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

Infra-estrutura de Componentes Paralelos para Aplicações de Computação de Alto Desempenho

Jefferson de Carvalho Silva

Fortaleza – Ceará
Junho 2008

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

Infra-estrutura de Componentes Paralelos para Aplicações de Computação de Alto Desempenho

Autor
Jefferson de Carvalho Silva

Orientador
Prof. Dr. Francisco Heron de Carvalho Junior

Dissertação apresentada à Coordenação do Programa de Pós-graduação em Ciência da Computação da Universidade Federal do Ceará como parte dos requisitos para obtenção do grau de ***Mestre em Ciência da Computação****.*

Fortaleza – Ceará
Junho 2008

Resumo

A construção de novas aplicações voltadas à Computação de Alto Desempenho (CAD) têm exigido ferramentas que conciliem um alto poder de abstração e integração de *software*. Dentre as soluções apresentadas pela comunidade científica estamos particularmente interessados naquelas baseadas em tecnologia de componentes. Os componentes têm sido usados para abordar novos requisitos de aplicações de alto desempenho, entre as quais destacamos: interoperabilidade, reusabilidade, manutenibilidade e produtividade.

As abordagens das aplicações atuais baseadas em componentes, no entanto, não conseguem abstrair formas mais gerais de paralelismo de maneira eficiente, tornando ainda o processo de desenvolvimento difícil, principalmente se o usuário for leigo no conhecimento das peculiaridades de arquiteturas de computação paralela. Um tempo precioso, o qual deveria ser utilizado para a solução do problema, é perdido na implementação eficiente do código de paralelização.

Diante desse contexto, esta dissertação apresenta o HPE (*Hash Programming Environment*), uma solução baseada no modelo # de componentes paralelos e na arquitetura *Hash*. O HPE define um conjunto de espécies de componentes responsáveis pela construção, implantação e execução de programas paralelos sobre *clusters* de multiprocessadores. A arquitetura *Hash* é constituída de três módulos distintos: o *Front-End*, o *Back-End* e o *Core*. A contribuição principal deste trabalho reside na implementação de um *Back-End*, como uma plataforma de componentes paralelos que estende o Mono, plataforma de componentes de código aberto baseada no padrão CLI (*Common Language Interface*). Feito isso, unimos o *Back-End* às implementações já existentes do *Front-End* e do *Core*, ambos em Java e sobre a plataforma de desenvolvimento Eclipse, através de serviços *web* (*web services*). Ao final, apresentaremos um pequeno teste de conceito, constituído por um programa paralelo construído a partir de componentes #, segundo as premissas e conceitos apresentados neste trabalho.

Abstract

The development of new High Performance Computing (HPC) applications has demanded a set of tools for reconciling high level of abstraction with software integration. In particular, we are interested in component-based solutions presented by the scientific community in the last years. Components have been applied to meet new requirements of high performance applications such as: interoperability, reusability, maintainability and productivity.

Recent approaches for component based development in HPC context, however, have not reconciled more expressive ways of parallel programming and efficiency. Unfortunately, this issue increases the software development time and gets worse when users have poor knowledge of architectural details of parallel computers and of requirements of applications. Precious time is lost optimizing parallel code, probably with non-portable results, instead of being applied to the solution of the problem.

This dissertation presents the *Hash Programming Environment* (HPE), a solution based on the # (reads "Hash") Component Model and on the *Hash* Framework Architecture. HPE defines a set of *component kinds* for building, deploying and executing parallel programs targeted at clusters of multiprocessors. The *Hash* Framework Architecture has three loosely coupled modules: the *Front-End*, the *Back-End* and the *Core*. The main contribution of this work is the implementation of the *Back-End*, since we have an early version of the *Front-End* and *Core*, both developed in Java on top of the Eclipse Platform. The Back-End was implemented as a parallel extension of *Mono*, an open source component platform based on CLI (*Common Language Interface*) standard. Once independently done, we bound all the modules together, using web services technology. For evaluating the proposed *Back-End*, we have developed a small conceptual test application, composed by # components.

Agradecimentos

Gostaria de agradecer a todas as pessoas que me incentivaram e apoiaram, possibilitando meu sucesso no mestrado, em especial: ao meu orientador Heron, pela confiança em mim depositada, pela competência e empenho no desenvolvimento deste trabalho.

A meu ex-orientador de graduação, Vasco Furtado, pela oportunidade de trabalho com o grupo de Engenharia do Conhecimento da UNIFOR.

A UFC e a CAPES pela oportunidade e financiamento deste trabalho.

Aos meus pais Jacqueline e Jales por minha educação, formação e apoio que tem alicerçado todas minhas vitórias. Aos meus irmãos Luciana e Thomas, pela amizade e amor incondicional.

Aos meus professores e colegas de graduação da UNIFOR.

Aos meus colegas de mestrado e a todos os mestres e funcionários do Departamento de Computação da UFC.

Sumário

Capítulo 1

Introdução

O tema desta dissertação insere-se no contexto da emergente área de programação baseada em componentes voltada a aplicações de Computação de Alto Desempenho, notadamente oriundas das ciências computacionais e engenharias. Nas seções seguintes, apresentamos as motivações, os objetivos e as contribuições deste trabalho de dissertação.

1.1 Motivações e Perspectivas

Diversas são as áreas que demandam por Computação de Alto Desempenho (CAD). Dentre elas, podemos destacar a farmacologia (simulações químicas), otimização aerodinâmica (automobilística e aeroespacial), área financeira (*Line-of-Business applications*), climatologia (Simulações de clima e tempo), *Data Warehouse*, Mineração de dados, biologia computacional, geologia, astronomia, mecânica de fluidos, simulações de fraturas em dutos e bacias petrolíferas, sistemas de inteligência artificial, manipulação de grandes bancos de dados etc. Essas aplicações, em geral, fazem uso de complexos modelos matemáticos ou apenas exigem alto poder computacional para cálculos simples em extensas bases de dados. Por exemplo, nesse último, uma multiplicação de matrizes, representadas de forma vetorial, possui um algoritmo muito simples. No entanto, dependendo da ordem das matrizes envolvidas, a resolução pode levar muito tempo. Uma solução para tal problema seria adaptar o algoritmo para trabalhar em um ambiente paralelo, aproveitando ao máximo seus recursos.

Em documento produzido recentemente com o objetivo de estabelecer tendências sobre os grandes desafios científicos que estão por vir, a comunidade científica ressalta a relevância das técnicas de CAD para enfrentar esses desafios na próxima

década [74]. Este é um feito que se repete em aplicações industriais como é o caso das grandes simulações [48] que são cada vez mais desafiadoras nas indústrias de petróleo, automobilísitca, aeronáutica dentre outras. Todos essas são demandas extremamente exigentes do ponto de vista computacional. Além disso, vale ressaltar o reconhecimento da indústria do *software* pelo nicho de aplicações abordado por CAD [33, 79].

1.2 Evolução da Computação de Alto Desempenho

Durante a década de 80, estações de trabalho (*workstations*) ocuparam o lugar de minicomputadores e *mainframes*. Devido ao baixo custo e à grande demanda por este tipo de máquina, houve um crescente e contínuo investimento neste setor o que ocasionou sua evolução. Assim, o alto poder de processamento alcançado por estações individuais, motivou o uso desta tecnologia em supercomputadores, o qual deu origem a arquiteturas de processamento paralelo de larga escala(MPP de *Massive Parallel Processing*). Estas máquinas são constituídas de memória distribuída e processadores interconectados por uma interface de comunicação entre processadores de alta velocidade. Podemos citar como exemplos de MPP's as seguintes arquiteturas: Intel Paragon, Cray's T3D e T3E, IBM SP2, ASCI Red e Sun HPC. Uma outra abordagem consiste na ligação de várias estações de trabalho através de uma rede convencional, formando um computador paralelo ou NOW (*Network Of Workstations*).

Apesar do grande potencial, as MPP's e NOW's não conseguiram acompanhar a evolução dos processadores lançados para computadores pessoais, que custavam bem menos. Foi então que percebeu-se que computadores comuns ligados em rede e gerenciados por um *software* (ou até mesmo uma *middleware* complexa) podiam equiparar-se, em desempenho, aos supercomputadores, com a vantagem de possuir um preço bem mais convidativo. Surgem assim os primeiros *clusters* formados por computadores pessoais, com "*hardware* de prateleira". Além disso, o surgimento de sistemas operacionais gratuitos e abertos, como o Linux e suas várias distribuições, tornaram ainda menos custosa a adoção desses tipos de máquinas. Com a massificação do *hardware* e dos sistemas operacionais, o campo de pesquisa e implementação de *software* para gerenciamento de aplicações voltadas para *clusters* de relativo baixo custo aumentou. Tais aplicações tentam aproveitar ao máximo o desempenho destas arquiteturas visto que o tempo de latência de comunicação entre os processadores do *cluster* ainda era alto. Isso culminou com o reconhecimento

da computação em *cluster* como uma das áreas de interesse do IEEE, com caráter multidisciplinar, em 1999.

Além da computação em *clusters*, outro grande avanço na área de alto desempenho, notadamente a partir da década de 90, foi a computação em grades (*grid computing*). Na computação em grade, os computadores estão interconectados através de uma infra-estrutura de comunicação como a *Internet*, por exemplo, formando uma infra-estrutura de computação. Numa grade [30], os recursos são, geralmente, heterogêneos e as aplicações não devem requerer intensa comunicação entre seus processos para beneficiar-se da estrutura em paralelo.

Na década atual, surgiram as arquiteturas de processadores de vários núcleos (*multi-core*), onde um único processador pode possuir dois ou mais núcleos, tornando o paralelismo real em nível de processador. A indústria de microprocessadores lançou no mercado soluções *dual-core*, com dois núcleos, *quad-core*, com quatro núcleos e cogita-se o lançamento de processadores *octo-core*, com oito núcleos independentes. Apesar de todas as suas vantagens, a tecnologia *multi-core* desperta a seguinte preocupação nas disciplinas de desenvolvimento de *software*: como construir programas que aproveitem ao máximo a disponibilidade de recursos ociosos em computadores com múltiplos núcleos? A resposta está na implementação de programas paralelos, exigindo o particionamento de dados ou de funcionalidades para execução simultânea. De nada adianta um processador com múltiplos núcleos, se um *software* não é capaz de explorar concorrência. Mais interessante ainda é o particionamento dinâmico de programas de acordo com a disponibilidade de núcleos ociosos.

Além das arquiteturas, a área de CAD abrange desenvolvimentos nas áreas de algoritmos paralelos e ferramentas de programação paralela. As arquiteturas, como vimos, estão em constante evolução e, obedecendo a *Lei de Moore*[1], têm seu desempenho aumentado exponencialmente ao longo do tempo [73]. Apesar do *hardware* evoluir com relativa facilidade, a programação voltada para esses tipos de máquinas não é tão simples. Além de lidar com a implementação das funcionalidades do programa, o desenvolvedor deve ser responsável pela coordenação de um conjunto, às vezes muito grande, de tarefas que ocorrem em paralelo, em muitos processadores. O programador deve se preocupar com o balanceamento de carga, gargalos, acessos a dados compartilhados e *deadlocks* no sistema. Sistemas escritos

[1] o número de transistores por polegada quadrada em um circuito integrado dobra a cada 18 meses.

em linguagens diferentes terão dificuldade de "conversar" ou podem estar ligados a uma determinada arquitetura, o que torna sua migração complicada. Outros desejam interoperabilidade. No entanto, internamente, assumem representações de dados diferentes como, por exemplo, a representação dos tipos booleanos nas linguagens C++ e Fortran.

Para contornar tal problema, foram propostos artefatos para comunicação entre processos e *middlewares* para o gerenciamento dos nós de uma rede aumentando o poder de abstração, agilizando assim a implementação. No entanto, conciliar os requisitos de modularidade, eficiência, abstração, portabilidade e generalidade em computação paralela de alto desempenho é ainda reconhecido como um grande desafio entre os pesquisadores [75]. "Paradigmas de programação paralela que conciliem portabilidade e eficiência com generalidade e abstração estão ainda em falta" [40].

No âmbito comercial, existe uma tendência em desenvolver *softwares* complexos voltados a máquinas *desktop* simples. Essas aplicações tem como principal propósito atender as necessidades imediatas de seus clientes sem necessariamente se preocupar com o uso de recursos da máquina. Obviamente que existe um certo nível de otimização por parte dos desenvolvedores, principalmente na programação que exige alto processamento gráfico, como ferramentas de desenho (tratamento de imagens) e jogos.

A Computação de Alto Desempenho tradicional, por outro lado, teve seu início com programas não tão complexos, que apenas efetuavam cálculos matemáticos sem a preocupação com interfaces de usuário ou metodologias avançadas de Engenharia de *Software*. A diferença para o modelo comercial era que esses cálculos deveriam, e necessitavam, rodar em arquiteturas complexas como os primeiros supercomputadores e os *clusters*. O recurso mais exigido era o processamento e uso de memória, para assim obter um tempo viável de resposta. Era então uma implementação dependente do *hardware*.

Com o passar do anos, desenvolvedores CAD perceberam a necessidade de formas mais complexas de abstrações de seus programas. A evolução do *hardware* (incluindo as redes de comunicação), linguagens e paradigmas de programação bem como o sucesso da aplicação de técnicas avançadas de Engenharia de *Software* na área comercial, sobretudo o uso de componentes, a procura por ferramentas de alto nível aumentou. Interoperabilidade, reuso, separação de interesses, escalabilidade e usabilidade tornam-se novas necessidades de um bom *software* científico. Aplicações

essas, voltadas a arquiteturas como *clusters* e grades computacionais. Enfim, arquiteturas com grande disponibilidade de recursos. Sendo assim, os anos recentes testemunharam o crescimento na complexidade do *software* científico o qual deve apresentar a robustez existente nas aplicações comerciais e o bom uso das disciplinas de Engenharia de *Software*.

Nos dias atuais, a computação paralela se situa em um ambiente extremamente heterogêneo, tanto no nível da arquitetura (máquinas diferentes) quanto no nível se *software* (sistemas operacionais e linguagens de programação). As ferramentas de programação devem ser desenvolvidas para permitir a computação nesses sistemas com o mínimo de problemas e o máximo de produtividade.

1.3 Programação Orientada a Componentes e Computação de Alto Desempenho

A partir do momento que percebeu-se que a construção de programas deveria ser considerada como uma questão de engenharia, a noção de unir partes genéricas pré-fabricadas em partes específicas tornou-ser peça chave. Essas "partes pré-fabricadas" ficaram conhecidas como "componentes", os quais são utilizados na solução de aplicações maiores.

A motivação por trás do uso de componentes reside não apenas em questões de engenharia. Em [71], o autor enumera três argumentos para o uso de componentes no desenvolvimento de programas. O primeiro deles enfatiza o reuso de código, em que componentes feitos por terceiros podem ser agregados a outras aplicações. O desenvolvedor, dessa forma, perde menos tempo e dinheiro concentrando-se apenas no código estratégico da sua aplicação e reusando componentes que podem até ser "de prateleira". O segundo argumento explica que componentes podem ser usados no processo de montagem de várias formas diferentes, ou seja, de um mesmo conjunto de componentes (*core*) podemos construir diversos tipos de aplicações. O terceiro e último argumento afirma que programas modernos possuem uma dinamicidade muito grande, tendo portanto que ser reconfigurados constantemente. O uso de componentes nesse caso diminui o acoplamento geral da aplicação e mantém sua coesão através de interfaces bem definidas.

A definição de componentes em computação varia um pouco de autor para autor mas esses em geral concordam que um componente é uma unidade de programa independente, que deve satisfazer um conjunto de regras de comportamento e implementar padrões de interfaces, que o permite ser agregado a outros componentes,

sendo assim reusados [3, 71]. Componentes diferem de módulos convencionais de programa uma vez que podem ser implantados independentemente, sendo possível o seu compartilhamento por diferentes aplicações. Na área de desenvolvimento de programas comerciais, por exemplo, a tecnologia de componentes tem ajudado a desenvolver aplicações facilmente interoperáveis (família *MS Office*, por exemplo) e interface gráficas (GUIs) de alto nível.

No entanto, a computação de alto desempenho não foi beneficiada com esses avanços [6]. Devido a sua grande necessidade de computação rápida e escalável, a maioria das plataformas de componentes voltados às aplicações comerciais não se adequam aos requisitos dos cientistas. A computação científica necessita de um conjunto de ferramentas de componentes diferentes daquelas encontradas no setor comercial, que dêem suporte a tipos complexos e que otimizem a comunicação entre processos. Para preencher essa lacuna, foram propostos modelos de componentes voltados a computação de alto desempenho.

O modelo CCA [6] (vide Capítulo 2) foi inspirado no padrão industrial CORBA e COM. Esse modelo faz uso de um padrão de projeto chamado *provides/uses* o qual possibilita uma conexão direta entre os componentes de uma aplicação. Portas *provides* são as interfaces que um componente CCA oferece. Dependências em outros componentes são expressas através da porta *uses*. Além disso, CCA oferece suporte a tipos comuns em CAD e a possibilidade de configuração dinâmica dos componentes em tempo de execução. SciRun [34, 55], XCAT [32], Ccaffeine [1] e MOCCA [46] são exemplos de *frameworks* que utilizam o padrão CCA. Um *framework* " define o esqueleto de uma aplicação o qual pode ser customizado por um desenvolvedor de aplicações" [64]. Sendo assim, um *framework* provê uma API de serviços necessários a um certo domínio específico para a construção de aplicações.

Fractal [13] adota um conceito parecido com o CCA, onde o caráter hierárquico dos componentes é explorado. O seu padrão de projeto separa a interface da implementação e a forma de comunicação entre os componentes é assíncrona. ProActive [14] implementa o modelo Fractal e é voltado à programação em grades.

Infra-estruturas de componentes disponíveis para o meio científico devem ainda prover formas de paralelismo, pois esta é uma grande necessidade das aplicações de Computação de Alto Desempenho. Aliam-se então as vantagens de se trabalhar com componentes (interoperabilidade entre linguagens e suporte a tipos) com o suporte à programação paralela. Para tratar a questão do paralelismo, cada modelo segue uma abordagem específica.

A especificação do CCA inclui extensões SCMD (*Single Component Multiple Data*) para suportar o estilo de programação paralela conhecido como SPMD (*Single Program, Multiple Data*). De acordo com [6], o paradigma de programação SCMD, acoplado com um conjunto uniforme de conexões entre componentes, irá produzir um programa com características SPMD para a aplicação como um todo. É acrescentado, ainda, que o paradigma SCMD pode descrever estruturas mais gerais de uma aplicação, como por exemplo uma simulação SCMD conectada com uma ferramenta serial de visualização. Ou ainda múltiplas simulações paralelas, operando em conjunto, para simular um ambiente físico complexo.

Os *frameworks* baseados em Fractal tratam o paralelismo com o uso de comunicação em grupo, através de portas coletivas. A comunicação em grupo permite disparar chamadas de métodos em um grupo de objetos ativos do mesmo tipo compatível. De acordo com [25], este padrão de comunicação coletiva se aproxima daqueles encontrados em, por exemplo, MPI.

Devemos também mencionar as extensões paralelas baseadas em CORBA, como o PARDIS [41], GridCCM [57] e Paco/Paco++ [63].

Apesar da sofisticação, tais modelos de componentes não conseguem, ainda, abstrair formas mais gerais de paralelismo. Além disso, a programação nestes ambientes é complicada, exigindo um bom conhecimento dos detalhes técnicos de arquitetura. A situação se agrava quando o usuário que quer tirar proveito das vantagens do paralelismo não possui conhecimentos em computação (físicos, químicos, biólogos etc.). Um tempo precioso que deveria ser gasto, em sua maior parte, na resolução do problema de um domínio é usado para adaptar o programa em um ambiente paralelo. O Capítulo 2 explica em maiores detalhes o modelo Fractal e implementações baseadas no mesmo.

1.4 Objetivos

Diante do contexto exposto, o objetivo deste trabalho é construção do projeto e a implementação de parte um ambiente de programação paralela para aplicações de alto desempenho, baseado no modelo de componentes #. Este ambiente, chamado HPE (*Hash Programming Environment*), consiste no desenvolvimento de três módulos: o *Front-End* (módulo de interação com o cliente), o *Core* (repositório de componentes #) e o *Back-End* (implantação de componentes # em uma determinada arquitetura computacional). Esta dissertação pretende concentrar-se no projeto e implementação do *Back-end* e sua integração aos demais módulos, os

quais definem a contribuição principal do autor.

Dentre as técnicas de *Engenharia de Software*, deve-se ressaltar neste trabalho o estudo de novos paradigmas de composição de componentes paralelos, o qual inclui a *separação de interesses* de uma aplicação e a composição de *componentes naturalmente distribuídos e paralelos*, ou seja, a implementação do componente de forma hierárquica onde suas subpartes devem estar implantadas nos nodos da arquitetura alvo. A interoperabilidade de linguagens e o controle de versões fica a cargo da tecnologia implementada pela linguagem C#, em Mono, um ambiente livre compatível com sistemas Linux.

1.5 Organização da Dissertação

Neste documento, apresentamos uma solução de implementação para os problemas citados acima, fazendo uso da tecnologia de componentes. O Capítulo 2 descreve o estado da arte no tema pesquisado nesta dissertação, introduzindo a tecnologia de componentes e sua aplicação em Computação de Alto Desempenho. O Capítulo 3 apresenta o modelo de componentes #, enfatizando a noção de conectores como componentes #, por meio de um exemplo prático. O Capítulo 4 introduz ao leitor os principais conceitos do *framework* HPE, incluindo o ciclo de vida dos componentes # com detalhes da arquitetura *Hash*. O Capítulo 5 detalha a implementação do *Back-End* sobre uma plataforma CLI, o Mono, principal contribuição da dissertação proposta. O Capítulo 6 apresenta o estudo de caso, o qual serve como um teste de conceito. O Capítulo 7 conclui esta dissertação. Ao final, é apresentada a bibliografia a qual serviu de base para este trabalho.

Capítulo 2

Componentes e Computação de Alto Desempenho

A rápida evolução dos equipamentos computacionais (ou *hardware*, termo anglicano mais conhecido na área), associada ao seu barateamento ao longo dos anos, ampliou o uso de computadores para solução de problemas que exigem um grande esforço computacional. Entretanto, ainda faz-se necessário o estudo de padrões e o estabelecimento de normas para que a implementação das aplicações (ou *softwares*) capazes de realizar estas soluções supram as principais características de aplicações de CAD, como, por exemplo, o suporte a tipos complexos e processamento paralelo eficiente. Os termos anglicanos *software* e *hardware* serão utilizados por padrão nesse texto de agora em diante.

Este capítulo tem como objetivo apresentar os principais esforços na definição de padrões para aplicações CAD, além da implementação de diversos *frameworks*. Na Seção 2.1 fazemos uma breve descrição sobre aplicações e a complexidade *hardware versus software*. Na Seção 2.2 introduzimos a tecnologia de componentes e os seus benefícios na implementação de componentes de alto desempenho. A Seção 2.3 apresenta o modelo de componentes CORBA, seguido de extensões desenvolvidas para aplicações paralelas. A Seção 2.4 introduz o modelo de componentes CCA, desenvolvido pela comunidade científica com o intuito de atender aos requisitos de aplicações de seu interesse. A Seção 2.5 apresenta o modelo hierárquico de componentes Fractal. A Seção 2.6 aborda o modelo GCM de componentes paralelos voltados a ambientes de grades computacionais. A Seção 2.7 introduz uma forma de comunicação em grupo para a implementação do padrão SPMD sob a ótica da Orientação a Objetos. Finalmente, a Seção 2.8 conclui o capítulo com uma análise

crítica das soluções apresentadas e a contribuição desta dissertação.

2.1 Aplicações de Computação de Alto Desempenho

Aplicações de computação de alto desempenho têm invariavelmente como alvo arquiteturas de processamento paralelo, como *clusters* ou grades computacionais. No estágio atual, aplicações multidisciplinares para soluções de problemas desafiadores em Ciências Computacionais e Engenharia podem ser formadas por diversos módulos que muitas vezes não foram implementados pelos mesmos desenvolvedores e que podem estar localizadas em sítios diferentes, onde são divulgados apenas seus serviços. Para garantir um bom desempenho, muitas usam código nativo ou bibliotecas de comunicação de alto desempenho entre os processos, como o MPI.

Como exemplo, podemos imaginar um cenário onde uma aplicação física que deseje fazer alguma simulação precise de um módulo para soluções de problemas de Álgebra Linear para os cálculos. Se partimos do pressuposto que ambas as aplicações, de domínios diferentes, foram modularizadas corretamente, podemos admitir o reuso dos módulos de computação algébrica pela aplicação física de uma forma relativamente simples. Caso os dois problemas concordarem entre si quanto suas interfaces de comunicação, a troca de dados deverá ocorrer facilmente.

Podemos acrescentar ainda que dentro do próprio domínio da Física, poderíamos implementar submodelos aos quais se comunicariam através de interfaces bem definidas, com baixo acoplamento. O baixo acoplamento permite que os módulos sejam trocados em tempo de execução ou compilação, incrementando a quantidade de testes e possibilidades ao desenvolvedor.

O tipo de aplicações de nosso interesse nesta dissertação são tanto complexas quanto ao *software* como quanto ao *hardware* o qual servirá como base de implantação. Este é um tipo de cenário recente em computação paralela de alto desempenho. O cenário mais tradicional compreende um *software* de caráter simples, em geral monolítico, o que explica parcialmente a popularidade de linguagens como Fortran e C até os dias atuais em computação científica, mas um *hardware* complexo, em geral dotado de um conjunto de processadores e interfaces de comunicações específicas entre esses, as quais devem ser programadas tendo em vista suas características peculiares.

No meio comercial, tradicionalmente as aplicações são complexas, justificando o intenso estudo em disciplinas de Engenharia de *Software* voltadas a essa classe de aplicações desde as últimas quatro décadas, mas devem ser voltadas a

máquinas simples, em geral dotadas de um único processador ou assumindo algum mecanismo de virtualização, em geral ao nível do sistema operacional, que esconde as características mais peculiares da arquitetura. Esse contexto no meio comercial permanece o mesmo com o advento de grades computacionais como plataforma para aplicações comerciais, onde há uma tendência em esconder a complexidade da arquitetura em relação às aplicações através de *middleware*. Isso se deve aos diferentes requisitos de aplicações de computação de alto desempenho perante as aplicações comerciais tradicionais.

O *software* é complexo no meio comercial pois deve ser implementado seguindo um conjunto de regras e disciplinas de Engenharia de *Software*, gerando diversos tipos de documentação. Além disso, o mesmo deve agradar ao cliente em vários quesitos, dentre eles a facilidade no manuseio (o que motiva os estudos na área de IHC, Interface Humano Computador), otimização para fornecer resposta em tempo hábil, além da compatibilidade com diversas tipos de Sistemas Operacionais. Em contrapartida, o *hardware* alvo de aplicações comerciais deve ser simples, geralmente de prateleira e formado por um único processador, como na maioria dos *desktops* comuns. Não é comum a necessidade de uma configuração avançada ou a instalação de uma *middleware* de gerenciamento. A indústria de jogos é um bom exemplo. Nesse nicho, o *software* possui uma grande complexidade na implementação, pois engloba diversas áreas da computação, mas são voltados e devem ser compatíveis com a maioria dos *hardwares* disponíveis no mercado. Dessa forma, atinge-se o maior número de consumidores.

O *software* científico tradicional, por sua vez, compreende computações complexas, porém com pouca necessidade de um alto grau de modularização e estruturação do *software*. Ao programador, interessa apenas que o programa efetue o cálculo desejado e emita a resposta, que muitas vezes é entendida apenas pelo especialista.

O *hardware* alvo por sua vez não é um sistema *desktop* comum. São supercomputadores ou arquiteturas dedicadas como *clusters* ou grades computacionais. O desenvolvedor deve conhecer a fundo detalhes técnicos da máquina ou das *middlewares* instaladas para a execução do paralelismo de maneira eficiente de forma a minimizar o tempo de resposta, muitas vezes crítico.

Recentemente, observa-se uma tendência da complexidade do *software* científico é aumentar [15, 60, 65, 68]. O uso de novas técnicas de Engenharia de *Software* bem com a preocupação em desenvolver um produto usável não só no meio científico,

por especialistas, acarretaram uma série de estudos para o desenvolvimento de padrões a serem seguidos. Paradigmas como a Orientação a Objetos e Orientação a Componentes introduziram novas possibilidades às aplicações de computação de alto desempenho, mas precisam ser adaptadas segundo alguns requisitos peculiares destas aplicações. O *hardware* alvo em aplicações científicas continua sendo máquinas de alto desempenho que exigem um forte acoplamento com o software. Supercomputadores, *clusters*, e grades computacionais são largamente usados. Na realidade, um tempo viável de resposta de um *software* científico só é conseguido nessas arquiteturas. O uso de componentes auxilia na abstração das tecnologias envolvidas, encapsulando código de comunicação, código de computação e até mesmos os dados envolvidos. Nas seções seguintes, examinaremos como a tecnologia de componentes vem sendo usada no domínio da computação científica.

2.2 A Tecnologia de Componentes

Dentre as muitas definições do que são componentes, a de *Szyperski* [71] resume suas principais características: "Um componente de *software* é uma unidade de composição com interfaces de contrato bem especificadas e que possui apenas dependências explícitas de contexto. Um componente pode ser implantado independentemente e ser sujeito à composição por terceiros."

Visando as necessidades das aplicações de alto desempenho, a comunidade científica vem adotando o paradigma da orientação a componentes para melhor adaptá-lo as suas necessidades [75]. Como dito, componentes são módulos independentes, que podem ser produzidos por terceiros adaptados, com relativa facilidade, a uma determinada aplicação. Uma aplicação CAD, formada por diversos módulos, pode ser implementada seguindo a orientação por componentes. A comunicação entre esses módulos fica a cargo das interfaces existentes.

Módulos implementados em linguagens diferentes, tão comuns em CAD, podem interoperar através dessas interfaces. Código nativo ou código MPI, como exemplificado, pode ser encapsulado em componentes adequados e usados conforme a necessidade. Deve-se ainda acrescentar que, dependendo da arquitetura alvo de implantação do componente, o mesmo deve ser compilado de forma a se adequar a uma grade computacional, ou a um *cluster*, por exemplo. Enfim, a orientação a componentes largamente aproveitada no meio comercial, adequa-se às necessidades científicas tirando proveito de seus benefícios e aliando-se a performance de bibliotecas CAD.

Um componente deve então disponibilizar aos seus clientes as suas funcionalidades por meio de interfaces. Um componente escrito com a tecnologia de *Web Services*, por exemplo, publica sua interface por meio de um arquivo escrito em XML, o qual descreve os serviços que deverão ser utilizados pelos clientes. Componentes Java RMI utilizam o conceito de *stubs* e *skeletons*, para invocação remota de métodos entre objetos que residem em espaços de endereçamento de memória disjuntos.

Componentes podem também ser compostos por outros componentes, através da conexão de suas interfaces, formando aplicações maiores, ou componentes compostos (componentes hierárquicos). Desta forma, o acoplamento entre eles torna-se menor, permitindo que uma mesma aplicação possa escolher várias implementações para uma mesma funcionalidade.

É possível também que, em tempo de execução, uma aplicação "conecte-se" com aquele componente que melhor satisfaça sua necessidade, ou aquele escolhido por seu cliente.

Outra característica importante dos componentes é que os mesmos devem possuir meta-informações sobre os seus procedimentos de implantação. Um componente informa a sua arquitetura alvo de implantação detalhes pertinentes, por exemplo, a sua dependência sobre a quantidade de processadores ou a natureza dos mesmos. Componentes podem ainda explicitar quais sistemas operacionais são compatíveis ou quais outros componentes devem estar previamente instalados para seu funcionamento. Além disso, componentes devem prover suporte a sistemas distribuídos.

As seções abaixo apresentam diversos modelos baseados em componentes e se adequam à Computação de Alto Desempenho, como interoperabilidade, suporte a código nativo, suporte a arquiteturas distribuídas, conectores e separação em interesses. Cada um dos modelos e suas respectivas implementações contribuíram para a formação do Estado da Arte deste trabalho bem como a implementação do modelo desta dissertação.

2.3 CORBA

Visando um modelo que garantisse interoperabilidade entre diferentes linguagens, através de uma interface comum e um sistema de mapeamento de tipo, a infraestrutura CORBA (*Commom Object Request Broker Architecture*) [26] foi proposta. Atualmente na versão 4.0 de sua especificação, CORBA tem sido

adotada como um padrão industrial para aplicações baseadas em componentes e o desenvolvimento de *middlewares* distribuídos. CORBA foi criado por um consórcio chamado *Object Mangement Group* (OMG), envolvendo mais de oitocentas companhias. Com CORBA é possível que programas distribuídos troquem dados independentemente da linguagem em que foram implementados ou da plataforma nos quais estão implantados.

A motivação da sua implementação tem como objetivo a comunicação e troca de dados entre aplicações, muitas vezes, de diferentes domínios, linguagens de programação e ambientes de implantação. Obedecendo a sua especificação, os desenvolvedores esperam um aumento no reuso de código, ligado em tempo de execução e suporte a aplicações em sistemas distribuídos.

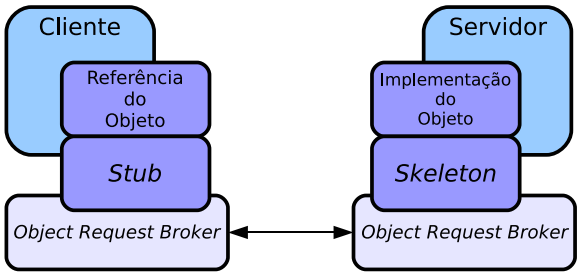

Figura 2.1: *Modelo Cliente-Servidor em CORBA*

A primeira versão estável de CORBA, chamada CORBA 1.0, foi lançada em 1991 e apresentava uma *Linguagem de Definição de Interfaces* (IDL, do inglês *Interface Definition Language*). Sendo assim, o desenvolvedor dessa tecnologia tinha a possibilidade de criar uma aplicação em uma determinada linguagem e gerar uma interface escrita numa linguagem comum entre outras tecnologias, a IDL.

A compilação da interface gera um conjunto de artefatos (*proxies*) responsáveis pela comunicação entre a aplicação criada e o cliente que a usará. Na Figura 2.1 vemos que do lado do cliente, uma referência ao objeto implantado no servidor é acessada por um código chamado *Stub*. O servidor disponibiliza formas de conexão a este objeto remoto por meio de sua classe *Skeleton*. O ORB (*Object Request Broker*) é o módulo intermediário responsável em tratar a requisição do cliente. A IDL também suporta diversos tipos de dados, desde primitivos como inteiros e booleanos aos mais complexos, como objetos e matrizes.

Inicialmente com suporte da linguagem C, CORBA hoje possui compatibilidade com diversas tecnologias como o Java, C++ e o *framework* .Net da Microsoft. Java possui uma implementação semelhante a CORBA, baseada em chamadas remotas

de método, o RMI (*Remote Method Interface*) que restringe a comunicação apenas entre objetos Java. A plataforma *.Net* também possui uma implementação de invocação remota de métodos, inclusa em sua biblioteca *System.Runtime.Remoting*. A vantagem da abordagem de invocação remota é que, além de serem mais eficientes que CORBA, permitem que uma aplicação faça uso de métodos remotos como se os mesmos fizessem parte de seu código. Entretanto, ambas soluções (Java e *.Net*) exigem que os objetos na comunicação estejam implementados sobre a mesma máquina virtual (JVM para o Java e CLR para o *.Net*).

```
component <nome>
    [ :  <base> ]
    [ supports <interface> [, <interface>]* ]
{
    <declaração de atributos> *
    <declaração de portas> *
};
```

Figura 2.2: *Código CIDL usado em CCM*

CCM, ou *CORBA Component Model*, difere do modelo dito clássico apresentado acima em alterar a sua linguagem de definição, a IDL, para a CIDL (Figura 2.2), objetivando a diminuição na complexidade de componentes CORBA (a partir da versão 3.0). A interface CIDL é então mapeada em uma IDL equivalente a qual será usada na comunicação entre componentes. CCM tem como premissa diminuir o tempo de desenvolvimento aumentando a abstração e reuso de código. Usando a palavra chave *component*, na CIDL, o desenvolvedor define o seu componente, as interfaces que o mesmo implementa e as portas (serviços) disponíveis aos seus clientes.

De acordo com o CCM, um componente é "uma unidade de *software* auto-contida constituída de seus próprios dados e lógica, com interfaces, ou conexões, bem definidas. É projetado para uso exaustivo no desenvolvimento de aplicações, com ou sem customização" [16].

As *facetas* ou interfaces de um componente CCM definem suas portas de acesso, expondo suas funcionalidades. *Receptáculos* são tipos de portas de configuração para especificar serviços requeridos de outros componentes. Muitas vezes, um componente CCM necessita "usar" um outro componente para poder concluir sua tarefa.

CCM também distingue-se por ser baseado em eventos. Os tipos de eventos se dividem em eventos *source* (entrada) e eventos *sink* (saída). Os eventos

tem a finalidade de diminuir o acoplamento no processo de comunicação entre componentes.

Apesar do apelo comercial, CORBA e seus "similares" chamaram a atenção também da comunidade científica sobretudo no que diz respeito a área de Computação de Alto Desempenho. A tecnologia de componentes, aliada ao modelo CORBA enfatiza as principais necessidades de um programa CAD das quais podemos citar: interoperabilidade entre linguagens; suporte a tipos primitivos e tipos complexos; adaptação a sistemas distribuídos; independência de arquitetura e sistema operacional. As subseções abaixo apresentam algumas ferramentas baseadas em CORBA para tal fim.

A especificação *Data Parallel CORBA* [27], proposto pelo consórcio OMG, consiste numa extensão ao modelo CORBA para o suporte à programação paralela, permitindo a implementação de objetos CORBA (baseados em IDL) paralelos. Objetos paralelos na realidade são um conjunto formado por implementações parciais dos mesmos executando em paralelo. Podem ser usados por clientes CORBA normais e também podem fazer uso objetos CORBA comuns.

A especificação para CORBA paralelo define uma semântica de particionamento e distribuição dos dados e requisições envolvidas no uso de objetos paralelos. Estes objetos são então, de alguma forma, distribuídos num arquitetura paralela, onde cada uma de suas partes é implantada em um processador diferente. Alia-se as vantagens do modelo CORBA (interoperabilidade e comunicação distribuída) com as vantagens do processamento paralelo.

Percebendo o potencial paralelo de CORBA, a comunidade científica baseou-se em seu modelo de componentes para a implementação de diversas infra-estruturas voltadas a Computação de Alto Desempenho. As extensões a seguir representam o produto da pesquisa sobre a tecnologia de CORBA paralelo sobre aplicações comuns no meio acadêmico, voltadas a arquiteturas robustas. É importante ressaltar que os trabalhos citados a seguir antecedem a proposta da especificação padrão *Data Parallel CORBA*, tendo na realidade exercido influência sobre a definição dele.

2.3.1 PARDIS

PARDIS (***PAR****allel* ***DIS****tributed*) [41] é um sistema distribuído na qual objetos representam computações paralelas SPMD. O Fato de usar CORBA em sua implementação permite que seus objetos possam interoperar com outros objetos CORBA em plataformas ou sistemas de *software* diferentes. Esses objetos interagem

através de interfaces definidas por uma IDL e cada um deles possui uma pequena aplicação encapsulada, servindo como bloco de construção para aplicações maiores. Além disso, o suporte a operações não-blocantes entre objetos permite ao PARDIS a construção de cenários concorrentes.

Para suportar o paralelismo, PARDIS estende o modelo de objetos CORBA à noção de objetos SPMD. Objetos SPMD, também chamados de objetos paralelos, podem ser definidos como objetos compostos por vários processos, visíveis ao servidor onde o objeto está implantado. Cada processo executa o mesmo programa sobre diferentes dados. O seu paralelismo é transparente às requisições dos clientes.

PARDIS implementa o conceito de programação paralela e distribuída. Ou seja, objetos SPMD instalados em um determinando sítio fazem uso da computação paralela localmente, gerenciada pelo PARDIS naquele domínio. Mas quando esses objetos comunicam-se com outros domínios (sítios remotos), através de interfaces CORBA, acessando outros objetos, fica então caracterizada a computação distribuída (comunicação remota).

Através da manipulação de dados distribuídos entre os processadores, PARDIS provê a generalização de *sequences* CORBA, chamadas *distributed sequences*. *Sequence*s CORBA é uma versão CORBA de um *array* unidimensional. Esse *array* permite dados de qualquer tipo, inclusive dados complexos. *Distributed sequences* é uma estrutura que torna possível a manipulação de dados espalhados entre vários processadores. A manipulação de estruturas de dados distribuídas busca a divisão da carga computacional entre o processos participantes gerando eficiência no cálculo do resultado.

2.3.2 PACO e PACO++

Para o projeto PARIS [54], PACO [61] foi a primeira tentativa de estender CORBA com paralelismo. Consiste na implementação de objetos paralelos para estender a IDL existente adicionando novos construtores. Estes construtores permitem especificar o número de objetos CORBA que fazem parte do objeto paralelo e os parâmetros operacionais de distribuição de dados. PACO faz uso de Fortran para definir distribuição de dados e MPI para comunicação entre processos (componentes CORBA com código MPI encapsulado).

Importante ressaltar que em nossa implementação planejamos a criação de componentes que especificam o ambiente instalado na arquitetura a qual o componente será implantado. Nesses componentes, assim como em PACO,

encapsulamos código MPI (chamada nativa) ou fazemos uso de outra tecnologia no lugar (alguma outra forma de comunicação inter-processos).

PACO++ [63] é a continuação do projeto PACO que compartilha a mesma noção de objetos distribuídos e os mesmos objetivos que dizem respeito a computação distribuída. PACO++ objetiva estender CORBA, não modificando o modelo e sim definindo uma extensão portável a qual seria compatível com qualquer implementação de CORBA. Sendo assim, a principal diferença entre PACO++ e PACO e PARDIS reside no fato de que estas duas últimas modificam a IDL já existente em CORBA. A modificação na IDL requer um novo compilador, tornando-o dependente da implementação de CORBA. Isso torna o código dos objetos paralelos menos portáveis e não possibilita a troca dinâmica de dados devido ao forte acoplamento entre os componentes.

O paralelismo em PACO++ é focado em aplicações SPMD, assim com o PACO e no PARDIS pois, segundo o autor, esse é um dos tipos de problema mais comum em computação paralela. Uma camada de *software* chamada PACO++ fica responsável pelo paralelismo. Ela é inserida entre o código do cliente e a implementação CORBA. Essa camada intercepta as chamadas ao CORBA e gerencia o paralelismo, tornando o código paralelo independente do CORBA.

Esta independência do ambiente é um requisito importante de Engenharia de *Software* pois garante que o programa responsável pela lógica de negócios possa ser alterado sem necessariamente interferir no código responsável pela comunicação.

2.3.3 GridCCM

GridCCM (CCM de *CORBA Component Model*) é uma extensão paralela do CCM, patrocinado pela INRIA. Para incorporar o paralelismo, GridCCM não faz nenhuma modificação na já existente IDL do CORBA. Na verdade, um arquivo XML auxiliar é usado para descrever o paralelismo entre componentes.

Assim como em Paco++, GridCCM não modifica o modelo de componentes CORBA. Ele apenas insere uma camada, chamada "camada GridCCM" responsável pelo paralelismo. Essa camada cuida do gerenciamento de objetos CORBA paralelos, situando-se entre o código do cliente o código de comunicação com CORBA.

O papel de GridCCM é permitir o gerenciamento transparente do paralelismo. O cliente envia os dados à camada GridCCM e essa por sua vez distribui e gerencia os mesmos entre os objetos CORBA distribuídos. A decisão de como os dados

serão distribuídos depende de restrições impostas pelo próprio cliente ou restrições impostas pelo servidor como a quantidade de recursos disponíveis para executar o componente, por exemplo.

O compilador GridCCM necessita de dois arquivos. Um arquivo definindo a IDL do componente e um arquivo XML que define a forma de paralelização do componente. A estratégia de particionamento dos dados é vista nesta dissertação como um novo componente que descreve como os dados devem ser distribuídos entre os processos. É portanto uma decisão que cabe ao cliente.

De acordo com [5], apesar dos modelos baseados em objetos CORBA, ou mesmo outras plataformas de componentes comerciais inspiradas em CORBA, como Java *Beans* (EJB) [29] e o COM da *Microsoft* [66], englobarem um grande quantidade de requisitos necessários às aplicações de alto desempenho, os mesmos ainda carecem de um suporte maior a tipos de dados voltados especialmente a esses problemas. Além disso, faz-se necessário padronizar o tipo de comunicação entre componentes otimizando ao máximo o envio de dados.

Em CORBA, não há suporte a tipos científicos como *arrays* multidimensionais ou números complexos, encontrados em linguagens como Fortran. COM, também voltado a aplicações corporativas, não possui abstrações para a construção de dados paralelos ou existentes em computação científica. COM também não suporta facilmente a implementação de herança ou herança múltipla, não sendo possível o uso de polimorfismo.

EJB, desenvolvido pela *Sun*, é uma solução baseada em Java que compete com a implementação da *Microsoft*. Ela não aborda a questão da interoperabilidade entre linguagens não sendo seu uso portanto adequado à computação científica. Mesmo usando a tecnologia JNI para interoperabilidade com código C e C++, as sucessivas chamadas a funções nativas pela JVM resultariam em quedas no desempenho da aplicação.

Na próxima seção, introduziremos o modelo CCA que objetiva atender requisitos de aplicações de computação de alto desempenho.

2.4 CCA

O *Common Component Architecture* (CCA) é um modelo de componentes para Computação de Alto Desempenho, desenvolvido por um consórcio formado por laboratórios de pesquisa e universidades dos EUA, patrocinadas pelo DOE (*U. S. Department of Energy*). De acordo com [6], CCA foi desenvolvido como um

modelo de componentes que evita conexões (*bindings*) dependentes da linguagem em que foram escritos. Esse artigo afirma ainda que, dependendo de como as conexões de linguagens são implementadas, aplicações que usam CCA não diferem em desempenho de aplicações escritas puramente em sua linguagem original. Isso se deve ao mecanismo de "conexão direta" do CCA, que permite que chamadas de funções entre componentes estejam diretamente conectadas de um módulo para outro, quando estes residirem no mesmo espaço de memória.

Considerando a variedade de aplicações científicas escritas em várias linguagens de programação diferentes, CCA propõe uma SIDL (*Scientific Interface Description Language*) [42] para garantir a interoperabilidade entre elas. A especificação CCA é puramente escrita em SIDL e, com a conexão de linguagem apropriada, especifica como deve ser um componente compatível com Fortran 77/90/95, C, C++, Python ou Java.

A ferramenta Babel [9], voltada a viabilizar a interoperabilidade entre linguagens comumente usadas em ciências computacionais, como C, C++, Fortran, Python e Java, tem mantido e desenvolvido a SIDL a qual provê suporte a tipos únicos necessários em computação paralela como números complexos, *arrays* multidimensionais e diretivas de comunicação paralela requeridas em componentes paralelos. Babel é responsável em analisar e gerar código (*proxies*) a partir de uma interface SIDL. O código gerado permite a troca de dados entre diversas linguagens de programação.

De acordo com sua especificação, um componente compatível com CCA deve possuir o método *setServices*, para comunicação com o *framework* de componentes (programa que cria e conecta componentes compatíveis com a especificação CCA). Em detalhes, através dos serviços do *framework*, o componente publica seu conjunto de portas aos demais componentes. Uma porta é um recurso que pode ser importado (porta do tipo *uses*) ou exportado (porta do tipo *provides*) por componentes.

O padrão *provides/uses* é comumente adotado em modelos de componentes de aplicações convencionais, como CORBA e COM/DCOM, os quais inspiraram muitas características do CCA. Neste padrão, uma porta descreve uma interface SIDL, contendo uma coleção de subrotinas implementadas em alguma linguagem. Desta forma, uma porta do tipo *provides* pode ser usada por componentes e uma porta do tipo *uses* poderá ser registrada para usar uma porta *provides*. Por exemplo, um componente A (Figura 2.3) publica uma porta *uses* através de uma chamada a um serviço do *framework* no método *setServices*. Esta porta, então, fica disponível ao

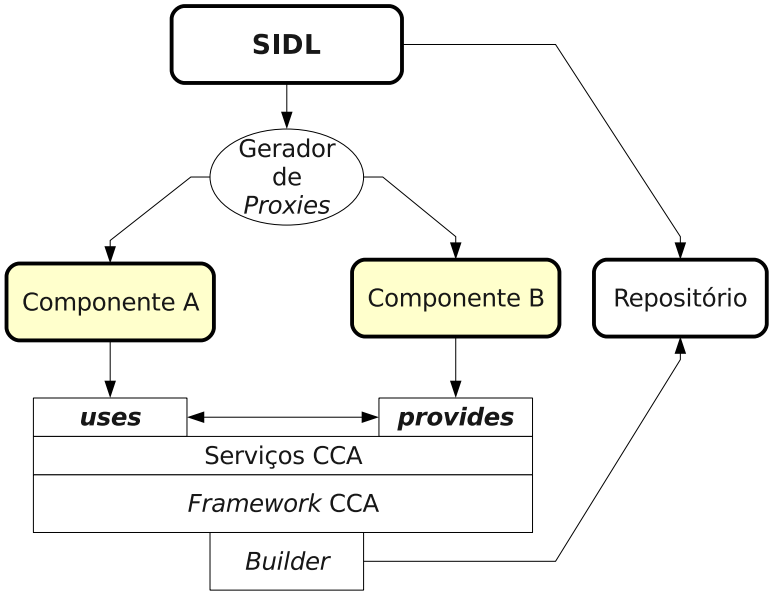

Figura 2.3: *CCA*

framework, o qual a conecta a uma porta *provides* de mesmo tipo pertencente a outro componente B. O componente A passa então a acessar os serviços do componente B através de um objeto do tipo da porta, a qual o representa.

A comunicação entre A e B é possível graças a geração de *proxies* (Gerador de *Proxy*), a partir da SIDL e através de alguma implementação (*framework*) do modelo. As definições das entradas e saídas dos componentes na SIDL são armazenadas em um repositório, o qual define uma API para a manipulação dos componentes implantados. Os *Builders* são responsáveis em montar a configuração. Através da API, o *framework* comunica aos *Builders* as atualizações referentes aos componentes.

O paralelismo em CCA é suportado através de suas diversas implementações, conhecidas como *frameworks*. Nas subseções seguintes, alguns exemplos de implementações do padrão CCA e o seu suporte a computação paralela.

2.4.1 CCAffeine

CCAffeine [1] é um *framework* baseado em Babel e suporta componentes paralelos baseados em MPI. Possui uma linguagem de *script* para a composição de aplicações e também uma interface gráfica (GUI). Componentes CCAffeine são criados dentro do mesmo processo portanto a comunicação entre componentes é feita através de chamadas de métodos locais.

2.4.2 XCAT

XCAT [32] é um *framework* distribuído baseado em Java, onde os componentes usam o protocolo SOAP para comunicação entre si (semelhante aos *Web Services*). XCAT pode usar ssh ou Globus para instanciar componentes componentes remotos, tornando-o mais apropriado a ambientes distribuídos. Possui também uma versão implementada em C++.

2.5 Fractal

O Fractal é um modelo de componentes modular e extensível que pode ser usado para projetar, implementar, implantar e reconfigurar sistemas e aplicações que vão desde sistemas operacionais a *middlewares* e interfaces gráficas para o usuário.

De acordo com [13], o modelo de componentes Fractal é adepto ao princípio da separação em interesses, também usado no modelo desta dissertação. Outras características incluem a separação nominal da interface e sua implementação, programação orientada a componentes e inversão de controle.

A separação nominal, também chamada de "padrão ponte" (*bridge pattern*), corresponde na separação do projeto dos interesses da aplicação, diminuindo assim o acoplamento. O segundo padrão corresponde à separação do interesse de implementação em vários interesses menores, implementados em entidades separadas chamadas componentes.

O último padrão corresponde à separação de interesses funcionais dos interesses de configuração. Ou seja, entidades externas de configuração ficam responsáveis em implantar os componentes que dizem respeito aos interesses funcionais. Dessa forma, caso alguma configuração deva ocorrer ao componente, não há a necessidade de o recompilar e o instalar novamente. Bastaria apenas mudar a entidade de configuração relacionada ao mesmo.

O princípio da separação em interesses é também aplicada a estrutura de componentes do Fractal, os quais são compostos de duas partes: o *content*, que diz respeito ao conteúdo que gerencia os interesses funcionais, e a *membrane*, que diz respeito aos interesse não funcionais (introspecção, configuração, segurança, transações e etc.). Além disso, um componente Fractal pode ser formado por outros componentes aninhados em seu interior, em diversos níveis (um conceito parecido com orientação a objetos, só que neste caso um componente é compartilhado por outros componentes, assim como objetos podem ser usados por outros objetos). As interfaces de configuração, relativas aos interesses não funcionais, permitem

serem implantados dinamicamente. Essas interfaces de controle podem ser inseridas diretamente no código ou através de ferramentas baseadas nelas, tais como ferramentas de implantação ou supervisão.

Fractal possui também um linguagem de descrição chamada Fractal ADL (*Architecture Description Language*), a qual provê um *schema* XML DTD (*Document Type Definition*) que tem por finalidade expressar a estrutura de um arquivo XML. Em Fractal, ele descreve, por exemplo, tipos de componentes, implementações de componentes, hierarquia de componentes e suas conexões. O DTD é também perfeitamente extensível para englobar outros interesses, inerentes a uma aplicação específica.

Extensões de Fractal, como as interfaces coletivas explicadas na subseção seguinte facilitam e elevam o nível de abstração para aplicações paralelas.

As subseções seguintes irão explanar alguns arcabouços os quais implementam o modelo Fractal.

2.5.1 ProActive

ProActive [11] é uma *middleware* em Java para computação paralela móvel e distribuída. Ela envolve orientação a componentes e objetos, a qual faz uso de componentes hierárquicos. ProActive é voltado para aplicações de meta-computação adaptada para aplicações em grades computacionais, podendo ser usado também em outros tipos de arquitetura, como *clusters*, por exemplo.

ProActive foi implementado usando a tecnologia Java RMI e também a API responsável pela reflexão (*Reflection*). Não requer nenhum tipo de modificação no ambiente de execução da JVM e nem necessita de um compilador extra.

Uma aplicação distribuída em ProActive é composta de pequenas entidades chamadas de *Active Objects*. Cada uma dessas entidades possui apenas um ponto de entrada, chamado de *root*, e possui sua própria *thread* de controle. As requisições a esta *thread* de controle são automaticamente armazenadas em uma fila de espera e atendidas de acordo com uma determinada política (FIFO por exemplo). *Active Objects* também podem ser movidos entre JVMs diferentes, através de um método de migração.

Uma outra característica muito importante em ProActive é a comunicação coletiva ou comunicação em grupo. Comunicação em grupo permite a invocação remota assíncrona para um grupo de objetos. A especificação de um grupo assemelha-se à criação de comunicadores por grupo em MPI, só que em mais alto

nível. O envio de um mensagem para um grupo é recebido por todos o processos pertecentes ao mesmo.

O objetivo do ProActive é combinar os benefícios gerados pela orientação a componentes com suas características (*Active Objects* e grupos de comunicação). Os componentes resultantes são chamados de "Componentes para Grade".

Para suportar o paralelismo, distribuição de dados e sincronização, tem sido proposto o uso de *interfaces coletivas* (*collective interfaces*) [12], baseado no modelo Fractal e na sua implementação mais importante, o ProActive. Interfaces coletivas expõem o comportamento coletivo dos componentes no nível de interfaces as quais oferecem serviços de envio de mensagem um-para-muitos (*one-to-many*) e muitos-para-um (*many-to-one*), respectivamente chamados de interfaces *multicast* e *gathercast*.

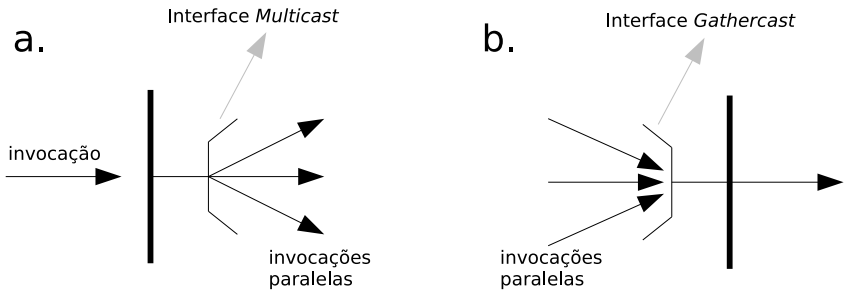

Figura 2.4: *O componente A "usa" uma porta provida pelo componente B*

Interfaces *multicast* (Figura 2.4 a) oferecem abstrações de comunicação *um-para-muitos*, as quais transformam uma única invocação em um lista de invocações. As invocações geradas são encaminhadas para servidores devidamente conectados. O resultado desta invocação pode ser uma lista de resultados ou uma redução. Invocações múltiplas aos servidores conectados ocorrem em *paralelo*.

Interfaces *gathercast* (Figura 2.4 b) oferecem abstrações de comunicação *muitos-para-um*, as quais transformam uma lista de invocações em uma única invocação. O objetivo é definir barreiras de sincronização e organizar os dados provindos das outras invocações. Os valores de retorno das chamadas são automaticamente encaminhadas aos processos requisitores.

A distribuição dos dados, usando interfaces coletivas, pode ser feita de duas maneiras distintas: o dado é copiado e enviado a cada um dos processos envolvidos (*broadcast*) na computação; o dado é particionado e pedaços menores que serão enviados separadamente a cada processo (*scatter*) envolvido na computação.

A forma de envio (*multicast* e *gathercast*) e o tipo de distribuição de dados pode ser configurado livremente pelo usuário. Através de *componentes controladores*, o desenvolvedor define o tipo de sincronização e o balanceamento de carga entre os processos. Ainda em [12], o autor afirma que o uso de interfaces coletivas facilita o desenvolvimento de aplicações se comparadas ao uso explicito de funções do MPI.

2.5.2 Julia

Julia [17] é uma outra implementação do modelo de componentes Fractal. Escrita na linguagem Java, Julia foi projetada para ser uma implementação leve e eficiente. Consiste em um *framework* que permite a criação e configuração de componentes Fractal, variando suas formas de acordo com a semântica associada ao componente. Julia provê um conjunto de semânticas de controle pré-definidas para componentes freqüentemente utilizados além de permitir a criação de semânticas personalizadas por usuário. Desta forma é possível redefinir ou customizar quaisquer aspectos de controle tais como o gerenciamento do ciclo de vida, criação de ligações, políticas de nomeação ou qualquer outro tipo de serviço técnico que seja desejado acoplar no modelo de componentes Fractal.

Julia usa ASM [19] para construção em tempo de execução de instâncias de componentes. ASM é usado em diferentes tipos de situação, dentre elas: gerar interceptadores e instâncias de interface Fractal; otimizar o código fazendo uso de estratégias para mesclar código, diminuindo o uso da memória; modularizar a escrita de classes de controle usando um algoritmo que gera *bytecode* de uma classe por meio de diversas camadas diferentes desenvolvidas independentemente.

2.5.3 AOKell

AOKell [18] é implementação da especificação Fractal patrocinada pelo INRIA e France Telecom. Este *framework* é responsável pela implementação de controladores de componentes e membranas. Uma membrana provê um nível de controle e supervisão do componente, influenciando no seu ciclo de vida e nas ligações entre componentes. O principal objetivo do AOKell é implementar um *framework* para programar controladores de componentes, os quais os usuários possam escolher e montar livremente objetos controladores para formar novas membranas em Fractal.

Comparado a outras implementações do modelo Fractal, AOKell distingui-se em prover uma abordagem baseada em componentes para implementação de controladores. As noções de uma interface cliente, uma interface servidora, um ligador (*binding*), um componente de composição são usadas no nível de controle de

uma mesma forma como são usadas no nível de negócios.

Uma outra característica importante a ressaltar sobre o AKOell é que o mesmo usa noções de Orientação a Aspectos (*Aspected Oriented Programming*) para confeccionar a "cola" que une a o código de aplicação e os componentes de controle. Cada componente é associado a um aspecto o qual monitora a execução do componente de aplicação e delega ao componente de controle a realização das funcionalidade de controle. O objetivo desta abordagem é o desacoplamento do código de controle do código de aplicação, semelhante ao padrão MVC.

2.6 O Modelo GCM

O modelo GCM (*Grid Component Model*) [56] foi idealizado pela comunidade *CoreGrid*, tomando como referência o modelo hierárquico de composição de componentes existente no Fractal e objetivando o seu uso em contextos de grades computacionais.

A hierarquia de componentes existente neste modelo possibilita ao desenvolvedor a composição de componentes GCM formados internamente por outros componentes GCM, e assim por diante. Fica abstrato aos usuários a noção de que seus componentes são formados por outros, a não ser que o mesmo queira explicitamente explorar seu componente.

Em adição ao estilo clássico de comunicação RPC (*Remote Procedure Call*), baseado em portas, GCM permite também o uso de portas de dados, *stream* e eventos no processo de interação de componentes. Padrões de interação coletiva também são suportados. As portas relativas aos dados permitem o compartilhamento de dados entre componentes de forma encapsulada e, ao mesmo tempo, preservam a realização de otimização *ad hoc*. Portas *stream* permitem a implementação do fluxo de dados em apenas uma via. Sendo assim, o seu uso explícito na comunicação entre componentes permite otimizações em tempo de execução. Já as portas baseadas em eventos podem ser usadas para prover a comunicação assíncrona entre componentes.

GCM pode suportar também diversos tipos de portas coletivas, incluindo aquelas que permitem a comunicação entre uma única porta *uses* e múltiplas portas *provides* ou a comunicação de múltiplas portas *uses* com apenas uma porta *provides*. Tal dinamicidade permite a implementação de todos os tipos de padrões de interação coletiva, derivados do uso de componentes compostos.

GCM adiciona ao modelo Fractal uma extensão para suporte a grades computacionais, as quais apresentam ambientes extremamente heterogêneos e

dinâmicos, GCM provê diversos níveis de gerenciadores autônomos em componentes, os quais se ocupam de interesses não funcionais da aplicação. Sendo assim, componentes em GCM possuem dois tipos de interfaces: as relativas aos interesses não funcionais e as relativas aos interesses funcionais. Implementações de interfaces não funcionais geram componentes que devem gerenciar as funcionalidades referentes às características não funcionais como eficiência e segurança. Implementações funcionais devem criar componentes cujas características afetam diretamente a computação. Cada componente GCM possui um ou mais gerenciadores que se comunicam com outros gerenciadores pertencentes a outros componentes através de suas interface não funcionais. Gerenciadores assumem estar presentes nos componentes responsáveis pelos aspectos que dizem respeito à grade computacional, contribuindo assim na eficiência de sua execução.

A arquitetura de componentes GCM é descrita fazendo uso de ADL (*Architecture Description Language*) a qual define o sistema de componentes usando composição e *bindings* de sub-componentes. Além disso, GCM também suporta a interoperabilidade em diversos níveis. O encapsulamento de componentes GCM dentro do padrão de *Web Services* permite a invocação de seus "serviços" por outros componentes.

Concluindo, é importante notar que este modelo de componentes é adequado tanto para a implementação de aplicações para Grades como até mesmo para a implementação de uma Grade, sendo que as duas se beneficiam das características acima explicitadas.

2.7 SPMD Orientada a Objetos

Os autores da referência [10] apresentam uma forma de comunicação em grupo (funcionalidade crucial para aplicações CAD) sob a perspectiva de orientação a objetos, chamada de SPMD Orientada a Objetos. Através de uma fábrica objetos, implementações "paralelas" compatíveis com as interfaces das classes originais são geradas e a comunicação entre os processos é feita através de invocação remota de métodos (RPC). Dessa forma, é possível uma maior flexibilidade na construção de componentes pois permite a composição avançada de blocos de computação paralelos voltados a arquiteturas em Grades e *Clusters*.

Também é apresentada uma extensão (generalização) do padrão ponto-a-ponto mostrado em *ProActive* para um modelo de grupos de Objetos Ativos (*Active Objects*) os quais possuem mecanismos para comunicação em grupo. O uso de

grupos facilita a implementação de modelos de alto nível como o *mestre-escravo*, além de poder reduzir o *overhead* na comunicação entre seus membros, otimizando a passagem de mensagens. É possível também alterar a camada de comunicação para melhorar o desempenho (usar *multicast* no lugar de RMI, por exemplo).

Agrupar membros que efetuam as mesmas tarefas é uma idéia razoável pois geralmente eles trabalham sobre o mesmo conjunto de dados. No modelo orientado a objetos, esta idéia é concretizada por um grupo de objetos que implementam uma interface comum. Quando estes objetos pertencem a um só tipo, dizemos que o grupo é **tipado**.

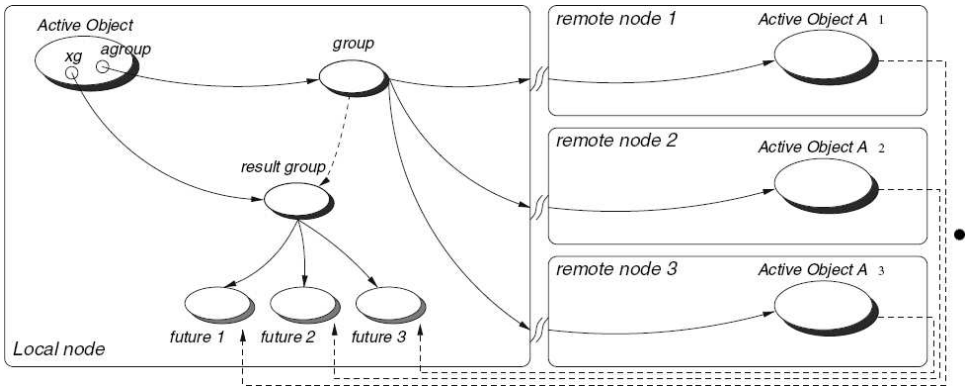

Figura 2.5: *Comunicação em grupo. Fonte: [10]*

A construção dos grupos é feita usando o *ProActive*. Um *stub* é responsável pela comunicação entre os objetos ativos remotos. Chamada a métodos é feita de forma transparente e um *stub* compatível com o tipo do objeto chamado é escolhido automaticamente. Uma função é usada para checar se a chamada é feita para um objeto apenas ou à um grupo de objetos (*group*). A chamada de método em um grupo é propagada a seus membros (*remote nodes*) usando *multithreading* (Figura 2.5). Os parâmetros da chamada são passados aos membros através de *broadcast*. Os resultados das operações são armazenados em um outros grupo, o grupo de resultados (*result group*). O grupo de resultados usa o mecanismo de espera-por-necessidade (*wait-by-necessity*). Quando um objeto invoca um grupo, fazendo um chamada de método, os resultados de sua chamada são armazenados em um outro grupo específico (*futures*) enquanto ele continua sua computação normalmente. O grupo de resultados passa os dados calculados apenas no momento em que o objeto chamador tiver necessidade. O objeto permanece bloqueado a partir do momento que requisita os resultados até a chegada dos mesmos.

A comunicação em grupo aliada ao padrão de objetos ativos encontrado em

ProActive pode ser de extrema utilidade na construção de aplicações paralelas e distribuídas, focando no modelo SPMD de programação. Esse enfoque dá origem ao modelo OO-SPMD (orientado a objetos). Segundo os autores [10], o OO-SPMD implementado no trabalho permite flexibilidade suficiente para o porte de aplicações a quaisquer protocolos de comunicação em baixo nível, bem como a portabilidade da aplicação para ambientes de grades, *clusters* ou mistos.

2.8 Conclusão

Este capítulo apresentou as principais propostas existentes no meio comercial e científico para tratar a orientação a componentes aliada ao processamento de alto desempenho. Vejamos agora as vantagens e desvantagens de cada proposta, contextualizando com o modelo que apresentamos nessa dissertação.

O modelo CORBA propõe uma linguagem universal entre as tecnologias e uma especificação de objetos paralelos. Entretanto, CORBA não suporta tipos de dados comuns em aplicações de alto de desempenho e nem implementa canais de comunicação eficientes entre componentes. Outros modelos como Java *Beans* e COM da *Microsoft*, inicialmente voltados ao nicho comercial, padecem dos mesmos problemas.

O modelo CCA foi uma forma de adaptar a IDL existente em CORBA para a aplicações que exigiam alto desempenho na comunidade científica. O resultado foi a SIDL, com suporte a tipos complexos. Além disso, o CCA provê um padrão de comunicação direta entre componentes, otimizando a troca de dados. No entanto, CCA não é em si um modelo de componentes paralelos, deixando a preocupação de paralelismo a cargo dos *frameworks* nele baseados, com destaque ao CCAffeine. Foi uma decisão proposital da comunidade responsável pela definição e manutenção da especificação do modelo CCA, tendo em vista que os requisitos de paralelismo ainda não eram muito bem conhecidos e amadurecidos na época de sua proposta.

O *framework* CCAffeine não oferece o suporte a formas mais gerais de paralelismo, estando restrito ao paradigma SCMD (*Single Component Multiple Data*), onde um componente CCA pode ser visto como um programa paralelo, o qual encontra-se replicado em diversos processadores. Cada instância do componente em execução em cada processador é dito pertencer a um mesmo *regimento de componentes*, portanto responsável por parte de uma computação paralela. Nos últimos anos, a comunidade CCA tem se empenhado na proposta de um modelo mais geral de paralelismo chamado MCMD (*Multiple Component, Multiple Data*).

Além disso, a ferramenta Babel tem sido extendida com o suporte ao PRMI (*Parallel Remote Method Invocation*), tornando possível a computação paralela distribuída em *frameworks* CCA. No entanto, continua problemático o suporte a chamadas de procedimento envolvendo M processos clientes e N processos servidores, uma instância do conhecido problema $M \times N$ abordado pela comunidade CCA. Portanto, formas mais gerais de paralelismo é um problema freqüentemente ainda endereçado pela comunidade CCA, o qual abordamos nesta dissertação através de um modelo de componentes inerentemente paralelo.

O Modelo Fractal e suas implementações, apesar de suportarem a composição hierárquica de componentes, não oferecem suporte à noção de componente paralelo e nem à noção de conectores responsáveis pela sincronização paralela. Para isso são necessárias extensões específicas em modelos de componentes CAD que suportam melhor a questão do paralelismo.

Em nosso projeto de dissertação, componentes são naturalmente paralelos. As subpartes desses componentes, chamadas *unidades*, são distribuídas entre diversos processadores e podem trabalhar sobre um conjuntos de dados distintos. A perspectiva orientada a interesses neste modelo de componentes, abordado no próximo capítulo, é a que mais se aproxima aos artefatos modernos de Engenharia de *Software*, ao contrário da perspectiva de decomposição baseada em processos freqüentemente enfatizada no projeto de aplicações que fazem uso de arquiteturas de processamento paralelo [23].

O padrão MPMD (*Multiple Program Multiple Data*) foi escolhido para a implementação dos componentes deste trabalho, ao contrário do estilo SPMD (*Single Program/Process, Multiple Data*) característico da maioria das soluções apresentadas neste capítulo. Entretanto, o seu uso não foi totalmente vetado. Em SPMD (ou às vezes SCMD), um mesmo programa (*Single Program*) é replicado em diversos processadores e trabalha sobre dados diferentes (*Multiple Data*).

Na abordagem de componentes apresentada nesta dissertação, nem sempre um componente representa a mesma computação replicada nos processadores (decomposição de domínio). No Capítulo 3 mostramos em mais detalhes como criamos componentes cujas subpartes compreendem computações diferentes (*Multiple Program*) sobre dados possivelmente diferentes (*Multiple Data*) (decomposição funcional). Além de componentes MPMD, mostraremos, numa mesma aplicação exemplos de componentes SPMD e o suporte do nosso modelo a esses tipos de abstração. Para tornar possível a comunicação inter-processo, fazemos

uso de conectores aplicados a CAD [20].

O uso de conectores tem como objetivo aumentar o nível de abstração e facilitar a interação entre componentes. Além disso, são os conectores próprios componentes do nosso modelo, com controle externo (coordenação exógena). No Capítulo 3 mostramos como usamos conectores na prática com auxílio de um exemplo de implementação. Em nosso modelo, como os conectores também são entidades programáveis, eles evoluem de acordo com as necessidade da aplicação [20]. Caso seja necessário alterar o ambiente responsável pela comunicação ou o tipo de dado a ser suportado por um conector, basta apenas trocar suas unidades por aquelas de interesse do desenvolvedor. Modelos como o Fractal e o CCA normalmente devem ser estendidos para suportar diferentes tipos de conectores [35], estando restritos a um conjunto pré-definido e restrito de conectores que nem sempre atendem a todos os requisitos das aplicações.

Concluímos que o uso de componentes naturalmente paralelos em nosso modelo aliado a maior flexibilidade proveniente da possibilidade de suporte a bibliotecas extensíveis de conectores, auxilia no desenvolvimento de aplicações paralelas e permite que a aplicação seja facilmente alterada, devido ao baixo acoplamento.

Capítulo 3

O Modelo de Componentes

Percebendo-se a necessidade de infra-estruturas de componentes adequados ao domínio de computação paralela e distribuída para aplicações de alto desempenho, o modelo # (lê-se *hash*) foi proposto [39].

Como discutido anteriormente, os modelos de componentes voltados a computação de alto desempenho [13, 25, 41] contém várias extensões de suporte ao paralelismo. No entanto, tais modelos não expressam formas mais gerais de paralelismo como em programação paralela baseada em passagem de mensagem [53, 69]. Esses modelos também consideram que processos são unidades básicas de decomposição de *software*. Ou seja, processos e interesses são colocados em uma mesma dimensão embora sejam ortogonais, como explicado a seguir.

Este Capítulo está organizado da seguinte forma: a Seção 3.2 apresenta um exemplo de paralelização de um problema matemático simples, buscando oferecer intuição sobre a idéia de decompôr programas paralelos por interesses ao invés de decompô-los por processos. Na Seção 3.3, o mesmo exemplo é definido em termos de componentes # e a noção de conectores é introduzida. Ao final, na Seção 3.5, é apresentada uma formalização, baseada em um cálculo de termos, para a composição de componentes do modelo # por sobreposição.

3.1 Fatiamento de Processos e Agrupamento por Interesses

Sob a perspectiva do modelo #, os processos que compõem um programa paralelo podem ser decompostos em várias fatias (*slices*) segundo algum critério[1] e que fatias de diferentes processos podem se agrupar em um interesse comum. Um

[1] Vale ressaltar que o fatiamento de programas (*program slicing*) segundo critérios específicos é uma área de pesquisa ativa em engenharia de software [72, 78], embora este trabalho não planeje a implementação de artefatos de fatiamento automáticos.

interesse "é uma abstração de uma solução canônica relevante para a solução de um problema dado" [51]. Interesses são unidades primárias da decomposição de um *software*. Eles podem ser funcionais, descrevendo computações, ou não funcionais, descrevendo aquilo que afeta computações. Migração de processos, tolerância a falhas, persistência, mecanismos de segurança, tempo de resposta, desempenho e localização física de um processo são exemplos de interesses não funcionais. Como interesses funcionais podemos citar: um pedaço de código que representa um cálculo significativo e operações de sincronização coletiva. Além disso, é possível ainda que exista uma hierarquia entre os interesses; por exemplo, um interesse que representa uma sincronização coletiva pode ser formado por um conjunto de operações *send/receive*.

Na programação paralela usual, interesses não são considerados peça principal do projeto. Muitas vezes, os interesses nem ao menos são considerados. O desenvolvedor preocupa-se inicialmente com a divisão do seu programa em processos que rodarão de forma independente. Os interesses surgem como conseqüência dessa abordagem que impossibilita uma visão mais geral da aplicação voltada ao paralelismo [38].

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 conseqüência.

3.2 Decomposição em Interesses

Para ilustrar a idéia do fatiamento (*slicing*) de processos em interesses, vamos considerar o seguinte exemplo: sejam A e B duas matrizes quadradas $n \times n$ e X e Y dois vetores $1 \times n$ (onde $n > 0$). Iremos calcular o escalar $r = (A \times X^T) \times (B \times Y^T)$. Consideremos um algoritmo paralelo para este cálculo em quatro processadores (P_0, P_1, P_2 e P_3) distintos, sendo que o processador P_0 que executa o processo *root* se responsabilizará em iniciar a computação (alocando memória para as matrizes e inicializando-as com valores numéricos) e recolher o resultado final, o escalar r.

O processador P_0, após inicializar as matrizes e os vetores, armazenará a metade superior (linha 0 até à linha $n/2-1$) da matriz A consigo e enviará a metade inferior de A (linha $n/2$ à 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 X inteiro ao processador P_1 e manterá uma cópia consigo.

De posse da matriz B e o vetor Y, P_0 enviará as metades superior e inferior da matriz B para os processadores P_2 e P_3, respectivamente. Já o vetor Y será enviado por inteiro aos processadores P_2 e P_3.

Após realizada a distribuição descrita acima, ficaremos com o seguinte cenário: P_0 de posse da metade superior de A (A_0) e do vetor X; P_1 de posse da metade superior de A (A_1) e do vetor X; P_2 de posse da metade inferior de B (B_0) e do vetor Y; finalmente, P_3 de posse da metade superior de B (B_1) e do vetor Y.

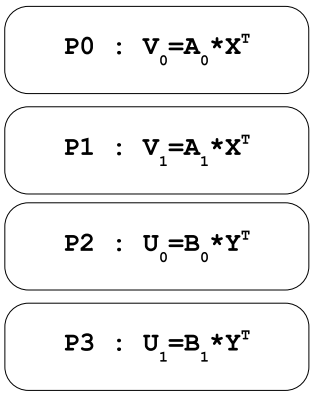

Figura 3.1: *Separação da computação nos processos envolvidos*

O objetivo desta distribuição é paralelizar o cálculo da multiplicação das matrizes nos quatro processadores, conforme mostrado na figura 3.1. Ou seja, o processador P_0 vai calcular $V_0 = A_0 \times X^T$ e P_1 vai calcular $V_1 = A_1 \times X^T$. O vetor V é o resultado dessa multiplicação, e o mesmo estará distribuído em P_0 (V_0) e P_1 (V_1). O mesmo ocorre para P_2 e P_3 onde é criado o vetor U que estará dividido em $U_0 = B_0 \times Y^T$ e $U_1 = B_1 \times Y^T$.

Até este ponto do algoritmo, foram calculados apenas o produto de matrizes por vetores de forma paralela, gerando dois novos vetores os quais chamamos de U e V. O próximo passo do problema inicial é o cálculo do produto vetorial $r = U^T \times V$, cujo resultado será armazenado em r. Este cálculo deverá ser efetuado de forma paralela nos processadores sendo que para isso faz-se necessária a redistribuição dos vetores apresentados na Figura 3.1. Vamos então dividir os vetores resultantes em cada processador, ou seja, V_0 que está localizado em P_0 será dividido pela metade em $V_{0-upper}$ e $V_{0-lower}$ (*upper* para a metade superior e *lower* para a metade inferior). Analogamente, nos outros processadores teremos que P_1 irá gerar $V_{1-upper}$ e $V_{1-lower}$, P_2 apresentará $U_{0-upper}$ e $U_{0-lower}$ e P_3 com $U_{1-upper}$ e $U_{1-lower}$ (Figura 3.2).

Esta nova configuração torna possível a redistribuição (Figura 3.3) dos dados nos diversos processadores. P_0 envia a P_1 $V_{0-lower}$. P_1 envia a P_2 $V_{1-upper}$. P_2 envia a

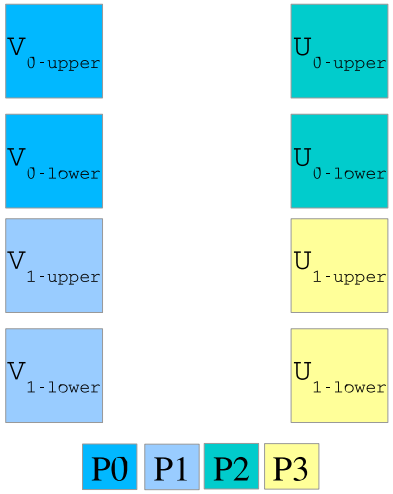

Figura 3.2: *Particionamento dos vetores U e V nos respectivos processadores. A legenda indica a cor de cada processador*

P_1 $U_{0-lower}$. P_2 envia a P_0 $U_{0-upper}$. P_1 envia a P_3 $V_{1-lower}$. P_3 envia a $P_2 U_{1-upper}$.

Após a redistribuição dos vetores *lower* e *upper*, cada processador deverá efetuar a multiplicação (Figura 3.4) dos vetores que foram recebidos e armazenar os dados em variáveis que representam o r particionado. Ao final, deverá ser feito o somatório desses valores no processador P_0. Logo $r = r0 + r1 + r2 + r3$.

O exemplo da multiplicação paralela explicado acima pode ser dividido em vários interesses. Alguns comuns a todos os processos envolvidos, e outros comuns a apenas dois processos, por exemplo. A Figura 3.5 ilustra como é feita separação do nosso exemplo em interesses. Abaixo, iremos enumerá-los, explicando-os de forma horizontal, concentrando-nos no processador P_0.

i. **Inicializar**: Este interesse pertence apenas à P_0. Ele se encarrega de inicializar as estruturas A, B, X e Y com valores inteiros aleatórios, para fins de teste.

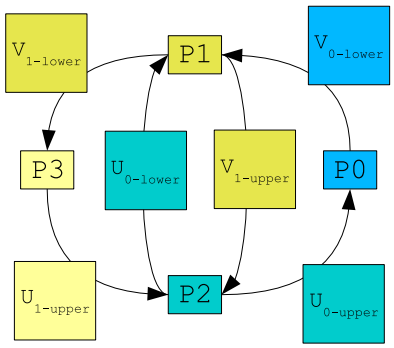

Figura 3.3: *Redistribuição dos vetores*

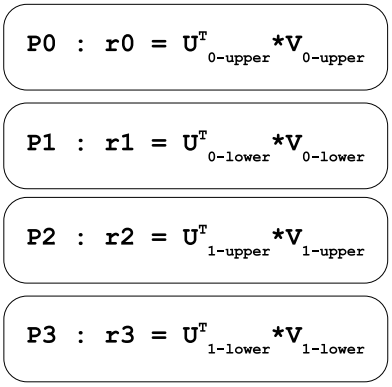

Figura 3.4: *Cálculo das partes de r em cada processador*

ii. Distribuir(A, X): Note-se que este interesse é comum tanto a P_0 quanto a P_1. Ele é responsável em distribuir as respectivas metades da matriz A e o vetor X (completo) entre esses dois processadores. Como explicado, P_0 irá receber A_0 e X^T. Já P_1 ficará com A_1 e também com Y^T.

iii. Distribuir(B, Y): Análogo ao interesse acima, este é comum a P_0, P_2 e P_3. Ele pertence a P_0 também pois este processador é quem inicia as estruturas de dados, no entanto, não efetua computação sobre as mesmas. Completando, ele distribui as respectivas metades da matriz B e o vetor Y entre P_2 e P_3. Como explicado, P_2 irá receber B_0 e Y^T. Já P_3 ficará com B_1 e também com Y^T.

iv. MultMatVet(A, X) e **MultMatVet**(B, Y): O primeiro multiplica A e X^T e armazena o resultado em V. O segundo, multiplica B e Y^T e armazena o resultado em U. Note que em P_0, apenas a metade superior de A é passada com parâmetro (A_0). E em P_1, a metade A_1 é usada. O mesmo vale para a matriz B, dividida entre os processadores P_3 e P_4. O vetores resultantes também são subdivididos em V_0, V_1, U_0 e U_1 e distribuídos respectivamente em P_0, P_1, P_2 e P_3. Importante ressaltar que esses interesses trabalham sobre dados diferentes. Em P_0 e P_1, *MultMatVet* atua sobre A, X e V. E em P_2 e P_3, atua sobre B, Y e U. Mesmo que em P_0, apenas a metade inicial de A seja usada e em P_1, a metade final, ainda assim esse dois processos compartilham o mesmo interesse, pois não houve a criação de uma nova estrutura de dados. Eles continuam atuando sobre a matriz A. Analogamente, o mesmo raciocínio vale para P_3 e P_4.

v. Redistribuir(U,V): Interesse comum a todos os processos pois os vetores U

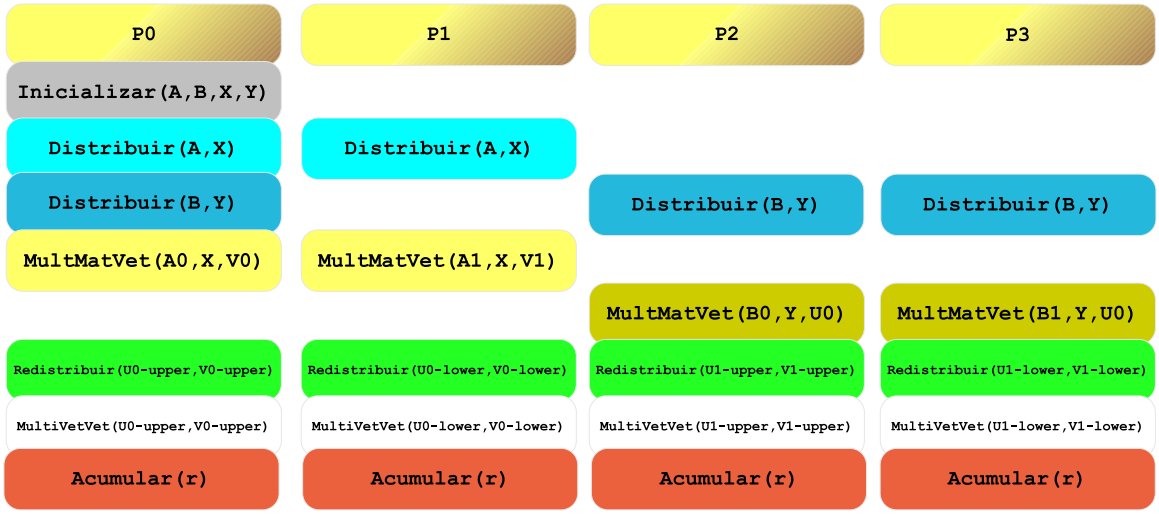

Figura 3.5: *Interesses comuns e específicos discriminados por cor em cada processador*

e V deverão ser repartidos entre os mesmos, sem a criação de novas estruturas de dados (Veja Figuras 3.2 e 3.3).

vi. **MultVetVet**(U,V): Outro interesse comum aos quatro processadores só que com a particularidade de que cada processador irá trabalhar com uma parte de U e V distinta das demais. Cada processador deve calcular, então, um escalar r.

vii. **Acumular**(r): Mais uma vez, um interesse comum a todos os processos. Em P_1, P_2 e P_3, a lógica será a de enviar o resultado calculado no interesse anterior em cada processador para P_0. P_0, além de enviar seu resultado a ele mesmo, irá somar os resultados provindos de outros processadores, e dele mesmo, no escalar r. O interesse é mútuo pois todos compartilham a mesma estrutura de dados, o escalar r.

Recapitulando, a multiplicação de dois vetores é um interesse comum a todos os processos. Cada processador necessita de uma rotina que efetua 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 (*root*). O recebimento do vetor X, é um interesse que diz respeito a apenas os processos P_0 e P_1. Já o vetor Y deve ser recebido por apenas os processos P_2 e P_3.

Visualizando o programa horizontalmente, por interesses, vamos de encontro à visão verticalmente centrada, nos processos, comum na programação paralela usual. Nos aproximamos então da abordagem usada na *Engenharia de Software*, as quais

assumem interesses como a unidade básica para a decomposição de um *software* [51]

Portanto, o modelo # propõe uma visão focada em interesses, quebrando assim o modelo tradicional de perspectiva orientada a processos dos artefatos de programação paralela existentes. A programação paralela sob a perspectiva #, passa então a ser orientada a interesses. 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. Por exemplo, o interesse **Acumular** nas unidades P_1,P_2 e P_3 apenas enviam seus dados à P_0. Esse, por sua vez, além de receber o dado calculado nele próprio, deverá fazer o somatório dos resultados provindos dos outros processadores. "Dessa forma, a síntese de um programa paralelo baseado em processos, capaz de executar de maneira eficiente em arquiteturas distribuídas contemporâneas, é possível a partir da decomposição baseada em interesses de um programa #. Conjectura-se sua generalidade, de forma que outras noções de componentes podem ser interpretadas em termos deste" [40].

3.3 Tratamento Uniforme a Conectores e Componentes

Um componente # é composto de várias unidades, cada qual constituindo a representação do componente em um dos computadores onde encontra-se implantada. Sob a perspectiva de processos, cada unidade constitui uma fatia de um processo pertencente a uma aplicação que faz uso do componente # (Figura 3.6). Portanto, podemos imaginar de forma intuitiva um componente # como um componente implantado em um conjunto de máquinas possuindo um papel possivelmente diferente, denotado pelas suas unidades em cada máquina. Isso generaliza a abordagem SCMD, comumente adotada em extensões paralelas para modelos de componentes, para uma abordagem MCMD (*Multiple Component Multiple Data*), padrão cuja importância tem sido reconhecida pela comunidade CCA para aplicações de Computação de Alto Desempenho [50].

Focando em interesses, programadores podem construir componentes # combinando-os através da sobreposição (*overlapping*). O compartilhamento de código entre os componentes é feito através de um mecanismo de fusão entre os interesses internos aos mesmos. Operações de Álgebra Linear, por exemplo, as quais são interesses de dois ou mais componentes #, podem se fundir para que as mesmas possam trabalhar sobre as mesmas estruturas de dados. O compartilhamento de estruturas de dados é fundamental para obter melhor desempenho em operações em um mesmo espaço de endereçamento, e tem sido explorados em modelos de

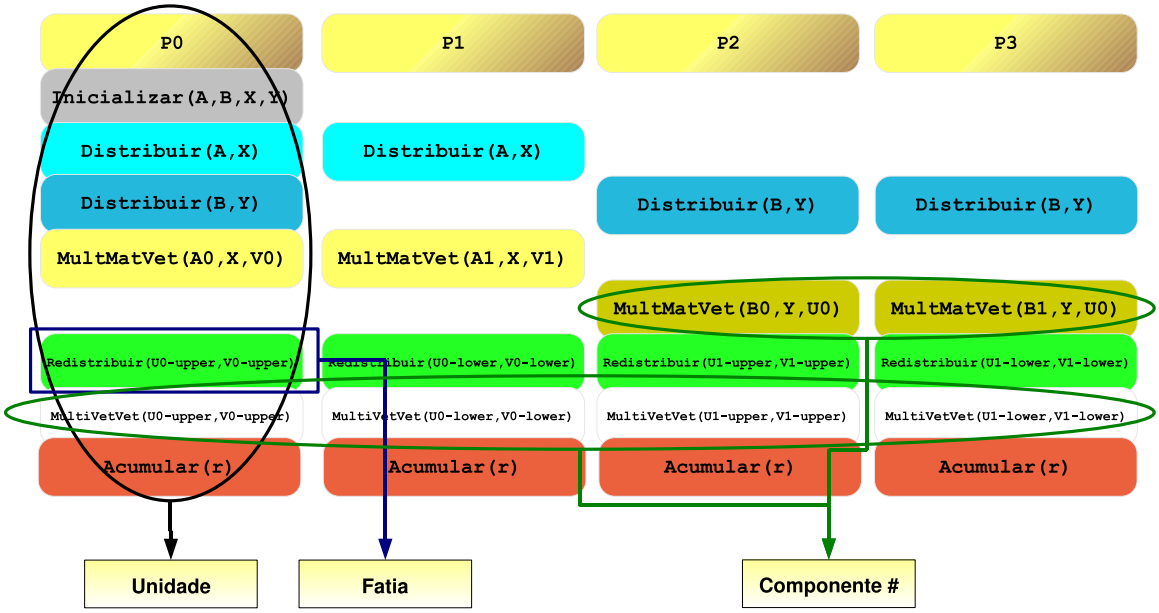

Figura 3.6: *Usando ainda o exemplo da seção anterior, um componente # é formado pela união das unidades pertencentes a um interesse referente a um conjunto de processos em um programa paralelo*

componentes como o Fractal [13].

De acordo com [49], um componente é uma unidade de computação a qual tem uma funcionalidade bem definida, independentemente implantável e sujeita a composição de terceiros e que pode ser agregada, por meio de conectores, a outros componentes para criar uma aplicação.

Conectores permitem a orquestração de um conjunto de componentes para atender alguma necessidade mais geral do que a funcionalidade individual de cada um. Segundo Allen e Garlan [2], conectores são constituídos de um papel (*role*) e uma cola (*glue*). Os papéis descrevem o comportamento local esperado de cada uma das partes que interagem. A cola atua como um especificação que determina as obrigações que cada componente possui na interação, além de sua coordenação.

Podemos afirmar serem os tipos de conectores suportados a característica que diferencia modelos e arquiteturas de componentes. Os modelos de componentes que tornaram-se populares no meio comercial suportam, em geral, conectores assimétricos que descrevem relações cliente/servidor entre componentes. Estes modelos tem servido como base inclusive para modelos voltados à computação de alto desempenho, como CCA e Fractal. Porém, em programação paralela é comum a existência de vários tipos de conectores, a maioria dos quais implementando relações simétricas, ou par-a-par (*peer-to-peer*) entre componentes [35].

A principal conseqüência do conceito de componente # é o tratamento uniforme entre os conceitos usuais de componente e conector sob a forma de um único conceito, o componente #. Torna-se possível que um *framework* ou infra-estrutura de componentes # suporte quaisquer tipos de conector que possa ser programado como um componente #, inclusive os que implementam a relação par-a-par entre componentes. O tratamento de conectores como componentes # é a conseqüência 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.

O componente # formado pelas unidades correspondentes às fatias de *Redistribuir(U,V)* é um conector cuja função consiste em repassar os dados U e V, calculados pelos componentes # *MultMatVet(A,X,V)* e *MultMatVet(B,Y,U)*, para o componente # *MultVetVet(U,V)*. Em termos mais gerais, o componente *Redistribuir* conecta um componente distribuído em N processadores (no caso, *MultMatVet(A,X,V)* e *MultMatVet(B,Y,U)* em dois processadores cada um), com um componente distribuído em M processadores (no caso, *MultMatVet(B,Y,U)* em quatro processadores), onde $M > N$. Os dados são transmitidos entre os componentes pela implementação do ambiente de comunicação entre processos.

Trabalhos importantes na área de *Engenharia de Software* têm proposto que conectores atuem em um papel preponderante na qualidade de um modelo de componentes [2, 45, 67]. Por exemplo, Mary Shaw [67] sugere que conectores em sistemas baseados em componentes sejam elevados à categoria de entidades de "primeira classe". Sendo assim, quando conectores são tratados em seu processo de desenvolvimento da mesma forma que outros componentes não menos importantes, incorporamos aos mesmos as vantagens inerentes à programação orientada a componentes. Por exemplo, podemos definir o tipo de dado a trafegar entre componentes e alterá-lo de acordo com a necessidade. Da mesma forma poderíamos definir e alterar um componente interno responsável pela comunicação entre processos. Os métodos usuais baseados em *imports* e *includes* usados na conexão de bibliotecas de componentes pecam por ignorar a importância das interações e conexões entre módulos. Como exemplo de modelos focados em conectores, citamos dois artigos no parágrafos que seguem.

Reo [4] é um modelo de coordenação para a composição de aplicações baseadas em componentes. Em Reo, os conectores são baseados em canais de comunicação. Um canal é um ponto de comunicação par-a-par com sua própria identidade e duas saídas distintas (*ends*). O uso de canais, formando os conectores, garante que

o código que une que os componentes possa ser adaptado com maior facilidade, diminuindo o acoplamento da aplicação. Nesta dissertação, é objetivo também implementar os conectores dos componentes # também como um código de mais alto nível, baseado em canais, com componentes #. Entretanto, o modelo # diferencia-se ao tratar componentes e conectores em uma mesma abstração, o componente #, permitindo o tratamento formal uniforme a esses [21].

Outro ponto a ressaltar é que componentes #, os quais implementam conectores, teriam uma coordenação exógena. De acordo com [4], uma coordenação exógena diz que as primitivas que causam e afetam a interação de uma entidade com outras residem em outras entidades. É possível que sejam feitas mudanças topológicas na aplicação sem interferir nem atualizar internamente os componentes da mesma, bastando apenas alterar as entidades responsáveis pela coordenação. Além disso, conectores são baseados em eventos, ou seja, eles são sincronizados a partir de eventos que disparam suas computações.

Em [45], Fiadeiro introduz um *framework* matemático, chamado de *CommUnity*, o qual apresenta formalismos para a construção dos processos que interconectam os componentes em um sistema complexo, separando-os da descrição da computação interna dos mesmo. Existe, então, uma preocupação em elevar o nível de abstração da "cola", ou conectores, que unem os componentes.

3.4 Sistemas de Programação # - Espécies de Componentes

O modelo # de componentes não define a natureza concreta de um componente #. Ele apenas delimita a maneira abstrata como componente captura o interesse da aplicação. A definição da natureza concreta de um componente # é feita através dos Sistemas de Programação #. Para tanto, estes devem definir *espécies* de componentes apropriados ao ambiente de computação alvo, associadas a requisitos de um certo nicho de aplicação. Uma *espécie* de componente agrupa componentes # que são definidos em termos das mesmas unidades da composição de um *software*. Além disso, componentes # de uma mesma espécie possuem o mesmo modelo de implantação em uma arquitetura como também as mesmas restrições de composição com outros componentes #.

A nossa implementação, baseada no modelo #, define um conjunto de espécies de componentes voltados a programação paralela de propósito geral, que serão usadas na montagem da aplicação exemplo. Outras implementações poderiam definir outras espécies, possivelmente de propósito especial, ou até mesmo reusar os conceitos de

$$
\begin{array}{lll}
\texttt{t} ::= & x & \\
& \lambda x.\ \texttt{t} & \textit{abstração} \\
& \texttt{t}\ \texttt{t} & \textit{aplicação} \\
& \mathbf{join}_{\kappa}\ \texttt{t}\ \texttt{t} & \textit{junção} \\
& \mathbf{fold}\ \oplus\ u_1\ u_2\ \texttt{t} & \textit{folding} \\
& \langle \mathcal{G}, \mathbb{C}, \mathbb{R}, \gamma, \rho, \kappa, c \rangle,\ \kappa{\in}\mathbb{K}\ c{\in}\mathbb{C} & \#\textit{-component} \\
 & & \\
\texttt{v} ::= & \lambda x.\ \texttt{v} & \textit{abstração} \\
& \langle \mathcal{G}, \mathbb{C}, \mathbb{R}, \gamma, \rho, \kappa, c \rangle,\ \kappa{\in}\mathbb{K}\ c{\in}\mathbb{C} & \#\textit{-component}
\end{array}
$$

Figura 3.7: # (Hash) Cálculo para sobreposição de componentes (sintaxe)

espécies implementadas por terceiros.

Estas implementações definem sua própria biblioteca de componentes #, que incluem desde componentes primitivos (para representações de tipos de dados, por exemplo) até componentes responsáveis por operações mais complexas (computações e conectores, por exemplo).

3.5 Sobreposição de Componentes - Semântica

Nesta seção é apresentada uma formalização da semântica da composição de componentes #, na forma de um cálculo de termos inspirado no λ-calculus. A Figura 3.7 apresenta os termos usados neste cálculo. As regras de avaliação de termos (semântica operacional) são mostradas na Figura 3.8.

Um termo (metavariável `t`) denota uma configuração. Configurações válidas sempre avaliam para componentes # (valores). Os termos *variável*, *abstração*, *aplicação* possuem significados provindos do λ-calculus. Já os termos *junção* e *folding* definem operações básicas de sobreposição.

Uma *abstração* modela uma configuração parametrizada por uma outra configuração de componente #. Por exemplo, considere $(\lambda x{:}\texttt{t})\texttt{t}_2$. Esta fórmula indica que todas as ocorrências livres de x em `t` devem ser substituídas por $\texttt{t}_2$. É possível definir configurações `t` cujo parâmetro x pode ser uma configuração de componente # arbitrária.

Uma *aplicação* define que o termo `t` à esquerda avalia para uma *abstração*.

O termo *componente* # é definido como uma tupla onde $\mathcal{G}$ é um grafo enraizado acíclico direcionado. As subárvores da raiz do grafo $\mathcal{G}$ denotam a hierarquia de fatias de cada unidade de um componente #. $\mathbf{C}$ denota o conjunto de instâncias de identificadores de componentes # de uma configuração, representada por c na tupla.

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}'_1}{\mathtt{t}_1\ \mathtt{t}_2 \rightarrow \mathtt{t}'_1\ \mathtt{t}_2}\ \textit{(E-App1)} \qquad \frac{\mathtt{t}_2 \rightarrow \mathtt{t}'_2}{\mathtt{v}_1\ \mathtt{t}_2 \rightarrow \mathtt{v}_1\ \mathtt{t}'_2}\ \textit{(E-App2)} \qquad \frac{\mathtt{t}_1 \rightarrow \mathtt{t}'_1}{\mathbf{fold}\ \oplus\ u_1\ u_2\ \mathtt{t}_1 \rightarrow \mathbf{fold}\ \oplus\ u_1\ u_2\ \mathtt{t}'_1}\ \textit{(E-fold1)}$$

$$(\lambda x{:}\mathtt{T}.\mathtt{t}_{12})\ \mathtt{v}_2 \rightarrow [x \longmapsto \mathtt{v}_2]\ \mathtt{t}_{12}\ \textit{(E-AppAbs)} \qquad \frac{\mathtt{t}_1 \rightarrow \mathtt{t}'_1}{\mathbf{join}_\kappa\ \mathtt{t}_1\ \mathtt{t}_2 \rightarrow \mathbf{join}_\kappa\ \mathtt{t}'_1\ \mathtt{t}_2}\ \textit{(E-Join1)} \qquad \frac{\mathtt{t}_2 \rightarrow \mathtt{t}'_2}{\mathbf{join}_\kappa\ \mathtt{v}_1\ \mathtt{t}_2 \rightarrow \mathbf{join}_\kappa\ \mathtt{v}_1\ \mathtt{t}'_2}\ \textit{(E-Join2)}$$

$$\mathbf{join}_\kappa\ \langle \mathcal{G}_1, \mathbb{C}_1, \mathbb{R}_1, \gamma_1, \rho_1, \kappa_1, c_1\rangle\ \ \langle \mathcal{G}_2, \mathbb{C}_2, \mathbb{R}_2, \gamma_2, \rho_2, \kappa_2, c_2\rangle \rightarrow \langle \mathcal{G}_1 \oplus \mathcal{G}_2, \mathbb{C}_1 \cup \mathbb{C}_2, \mathbb{R}_1 \cup \mathbb{R}_2, \gamma_1 \cup \gamma_2, \rho_1 \cup \rho_2, \kappa, c\rangle\ \textit{(E-Join3)}$$

$$\frac{u_1 \in \textit{nodes}(G),\ \ u_2 \in \textit{nodes}(G),\ \ v \notin \textit{nodes}(G),\ \ \{(\bot, u_1), (\bot, u_2)\} \subseteq \mathcal{G}}{\mathbf{fold}\ \oplus\ u_1\ u_2\ \ \langle \mathcal{G}, \mathbb{C}, \mathbb{R}, \gamma, \rho, \kappa, c\rangle \rightarrow \langle \mathcal{G} \cup \{(\bot, v), (v, u_1), (v, u_2)\}, \mathbb{C}, \mathbb{R}, \gamma \cup \{v \mapsto c\}, \rho \cup \{v \mapsto \rho(u_1) \oplus \rho(u_2)\}, \kappa, c\rangle}\ \textit{(E-fold2)}$$

Figura 3.8: Regras de redução para o cálculo # de sobreposição (semântica)

A função $\gamma{:}\textbf{\textit{nodes}}(\mathcal{G}){\rightarrow}\mathbb{C}$ imprime cada nó do grafo com o identificador de instância do componente interno que o possui como unidade. $\mathbf{R}$ representa o conjunto que denota os papéis atribuídos a cada fatia pela função $\rho{:}\textbf{\textit{nodes}}(\mathcal{G}){\rightarrow}\mathbf{R}$.

O interesse endereçado a um componente # é suficientemente definido por um conjunto de papéis delegados as suas unidades. Por exemplo, sejam u_1 e u_2 unidades de dois componentes # unidos (*joined*) C_1 e C_2, respectivamente. Seus papéis podem ser combinados pela aplicação do termo **fold** com o operador $\oplus$, definindo uma nova unidade a qual possui o papel $\rho(u_1) \oplus \rho(u_2)$. Por exemplo, se os papéis de u_1 e u_2 forem objetos Java das classes J_1 e J_2, o operador $\oplus$ forma uma unidade cujo papel é concretizado por um objeto de uma classe que possui os campos u_1 e u_2 dos tipos J_1 e J_2.

O elemento κ identifica a espécie do componente #. Ele é um elemento de $\mathbb{K}$, o conjunto das *espécies de componentes* suportadas pelo sistema de programação # em questão.

Usando uma notação inspirada em [59], a Figura 3.8 apresenta a semântica para a validação de configurações de componentes #, sobre termos `t`. Com esta semântica, é possível a contrução de componentes # baseados em configurações *bem-formadas*. Configurações bem-formadas dão origem a valores (componentes #).

As regras *E-App1*, *E-App2*, e *E-AppAbs* são regras de redução usuais de chamada-por-valor para o λ-calculus. As regras de congruência *E-Join1* e *E-Join2*

montam componentes # provindos de configurações aplicadas ao operador de junção, enquanto a regra de computação *E-Join3* mostra como unir dois componentes # dando origem a um novo componente #. O significado da regra de congruência *E-Fold1* e a regra de computação *E-Fold2* (onde $\perp$ é a raiz) é similar, mas de interesse a operação *folding*.

A representação hipotética da configuração de um componente # na Figura 3.9 exemplifica conceito de hierarquia. A, B, C, D, E, F e G são identificadores de instância para componentes # pertencentes ao conjunto $\mathbf{C}$. O componente A é formado pela sobreposição dos seus componentes internos B, C e D. Já em B, temos que E e F são seus componentes internos e E e G, componentes internos de C.

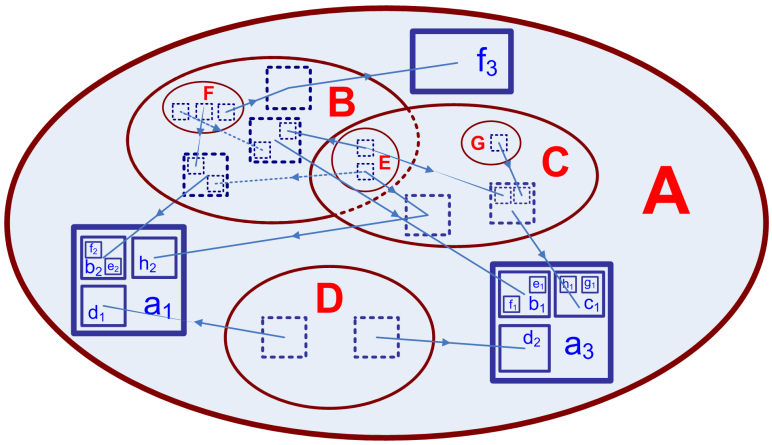

Figura 3.9: *Componente # hipotético, enfatizando o processo de sobreposição*

Na Figura 3.9 é apresentada a representação do componente A usando um grafo como proposto pela formalização teórica, ainda ilustrando o uso da função γ, cujas subárvores da raiz representam as unidades de A. São elas: *a1*, *f3* e *a3*. Para facilitar o entendimento, vamos nos concentrar na explicação da hierarquia de fatias da unidade *a1*, representada nas Figuras 3.9 e 3.10 utilizando diferentes notações. A unidade *a1* possui em seu interior três fatias: *b2*, *d1* e *h2*. Note que *b2* é formado pela fatia *e2* e pela fatia *f2*, onde a primeira veio do componente E (interno a B) e a segunda, veio do componente F, também interno a B (na verdade, *b2* é um nova fatia inicialmente formada dentro de B e depois importada a A, para dentro de sua unidade *a1*). Continuando, *a1* ainda é formado por *h2*, proveniente do componente H, interior à C e, finalmente, *a1* é formado por *d1*, interior ao componente D.

Na Figura 3.10, as outras duas unidades, *a3* e *f3*, possuem uma explicação de formação análoga a *a1*. Ainda na mesma figura, o grafo $\mathcal{G}$ demonstra o conjunto $\mathbf{C}$ e a função ρ, indicando a origem das fatias de cada unidade.

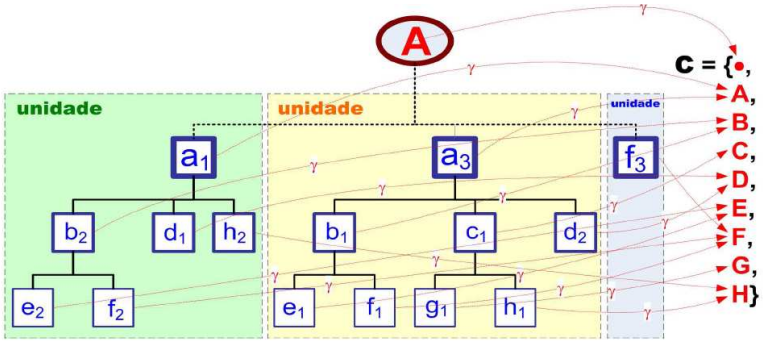

Figura 3.10: *Novas unidades formadas a partir da sobreposição de fatias dos componentes internos à A*

A Figura 3.11 ilustra o processo de sopreposição dos componentes B, C e D para formar o componente A, apresentando informalmente ao leitor o significado das operações de *junção* e *folding*.

Em **(2)**, a operação de *junção* dá origem a A (termo $\mathtt{t}$, resultante da junção de $\mathtt{t}_1$, $\mathtt{t}_2$, $\mathtt{t}_3$). Essa operação de junção apenas cria o novo componente fundindo as raízes de seus componentes internos.

Em **(3)** a operação *fold* atinge o primeiro ramo de A, dando origem a um novo termo $\mathtt{t}'$. Note que *a1* é formado por sobreposição de fatias, como explicado anteriormente no exemplo da Figura 3.10.

Em **(4)**, o novo termo $\mathtt{t}''$ surge da reaplicação da operação *fold* sobre $\mathtt{t}'$, dando origem a unidade *a2* cujo processo de formação é análogo à *a1*.

Finalmente, em **(5)**, o processo de fusão age sobre os componentes internos E e H, supondo que suas fatias e_i e h_i possuem o mesmo papel, ou seja, $\rho(e_i) = \rho(h_i)$, $i = 1, 2$. Sendo assim, elas são fundidas em eh_i, dando origem ao termo t''' .

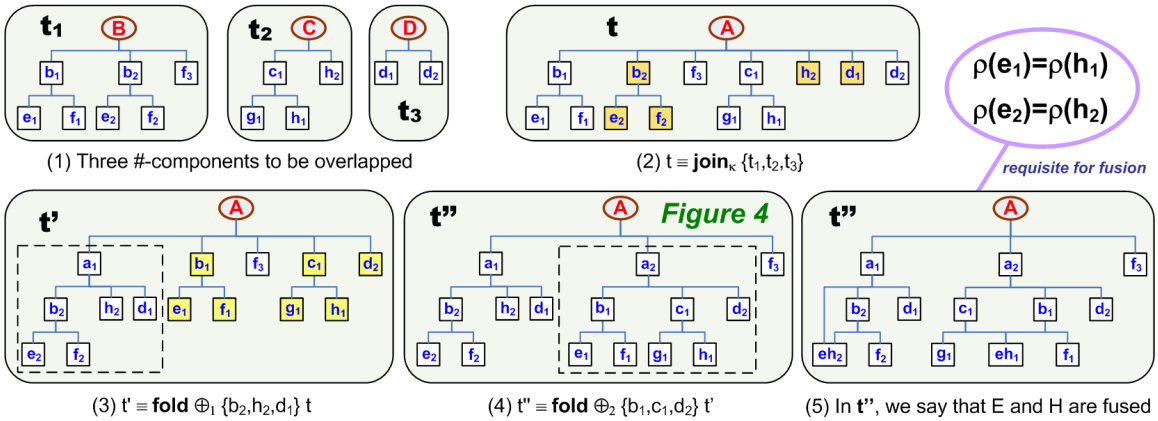

Figura 3.11: *Operações fundamentais de sobreposição sobre componentes internos à A*

3.5.1 Conclusão

Concluimos com o nosso exemplo didático que a separação de interesses de uma aplicação paralela facilita a composição de componentes #, inerentementes paralelos devido às suas unidades, as quais executam em processos diferentes. Cada interesse representa um componente #, o qual engloba uma tarefa associada a sua *espécie* de componente. Estes componentes devem então interagir entre si, através de seus conectores, com o intuito de executar a aplicação.

Capítulo 4

Um Arcabouço para Construção de Sistemas de Programação

Para que seja possível a concretização das idéias contidas no modelo # de componentes, a implementação de um arcabouço, ou *framework*, para construção de ambientes de desenvolvimento de aplicações de Computação de Alto Desempenho, foi idealizada obedecendo a arquitetura *Hash* [36, 40].

A partir dessa arquitetura, é possível seguir um padrão de implementação que levará ambientes de programação e de solução de problemas em áreas específicas a um nível maior de interoperabilidade. Enfim, é prevista a possibilidade de que vários *frameworks*, baseados na Arquitetura *Hash* e desenvolvidos por autores diferentes, cooperem entre si para a solução de problemas em ciências computacionais e engenharia. Sendo assim, propomos o *framework* HPE (*Hash Framework Environment*), sobre o qual instanciaremos o ambiente HPE, com espécies de componentes voltadas à programação paralela de propósito geral.

Este capítulo está organizado da seguinte forma: a Seção 4.1 identifica e relaciona as principais fases do ciclo de vida de um componente # no HPE. A Seção 4.2 apresenta os principais componentes da arquitetura Hash, objeto de estudo nesta dissertação. A Seção 4.3 apresenta ao leitor o HPE, com suas respectivas **espécies de componente** e sugestões de implementação.

4.1 Ciclo de Vida de Componentes

Uma infra-estrutura de componentes deve definir os estágios do ciclo de vida de seus componentes suportados. No caso do HPE, reconhecemos os seguintes estágios:

i. Descoberta: Nesta fase, o desenvolvedor faz uma requisição ao ambiente

sobre o conjunto de componentes disponíveis no momento. Estes componentes estão espalhados em diversos sítios. O mecanismo de descoberta deve permitir visualizá-los como um repositório de componentes de forma transparente quanto a localização.

ii. **Configuração**: 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 #.

iii. **Publicação**: A publicação consiste em tornar disponível a outros desenvolvedores, componentes criados por um programador em particular. Uma vez publicado, um componente pode ser descoberto e usado como um componente interno em uma nova configuração.

iv. **Implantação:** Quando o desenvolvedor cria um novo componente ou aplicação, ele deverá implantá-los em alguma plataforma de computação para poder executá-lo. A implantação consiste na montagem do código objeto e sua instalação, de acordo com a arquitetura alvo. No entanto, a geração de código objeto depende de qual *espécie* o componente pertence. Alguns componentes não dizem respeito a código fonte, portanto não gerariam código objeto. Tais componentes poderiam armazenar, ao invés de código fonte, metadados sobre um arquitetura em particular, por exemplo.

v. **Produção**: Colocar o sistema em fase de produção consiste em torná-lo disponível para execução e monitoração em tempo de execução.

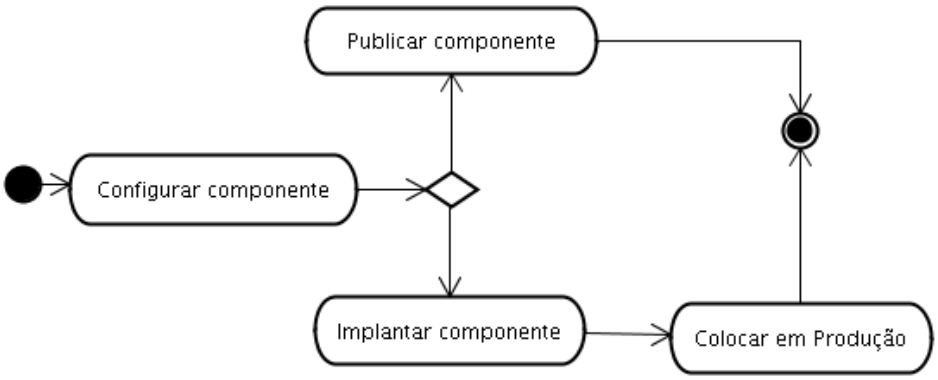

Figura 4.1: *Ciclo de Vida em sistemas de programação baseados no Framework HPE*

4.2 A Arquitetura *Hash*

A arquitetura *Hash*, utilizada no *framework* 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 # [40].

O *Back-End* é responsável por gerenciar as relações entre componentes # e as arquiteturas computacionais sobre os quais estes serão implantados e onde serão instanciadas quando em produção.

O *Core* tem como função oferecer serviços de configuração de componentes # e aplicações, através da sobreposição de componentes #. Os componentes serão descobertos a partir de um repositório distribuído de forma transparente ao desenvolvedor em um conjunto de *locations*. As *locations* servirão para que os desenvolvedores possam publicar seus componentes #.

O *Front-End* é a interface pela qual desenvolvedores de componentes e aplicações controlam o ciclo de vida deles por meio do acesso aos serviços do *Back-End* e do *Core*. A Figura 4.2 ilustra graficamente a estrutura e responsabilidades dos módulos da arquitetura *Hash*.

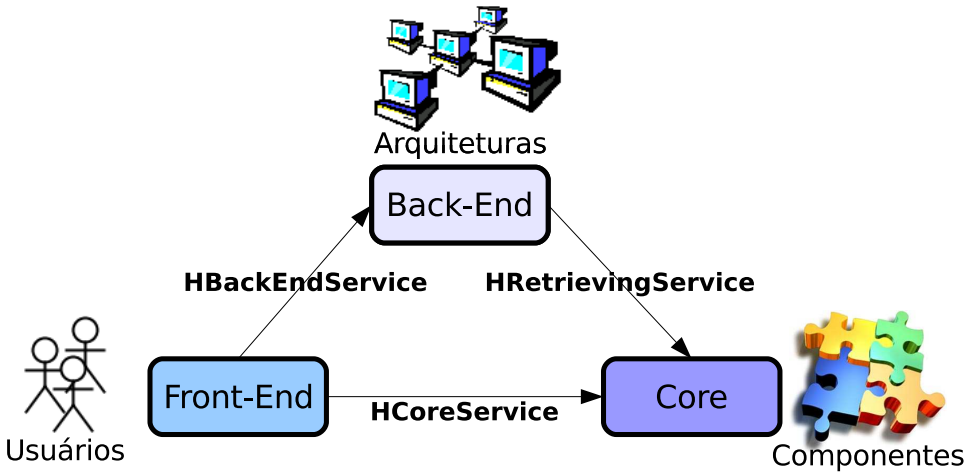

Figura 4.2: *Arquitetura HPE*

A Figura 4.3 nos mostra o diagrama de classes envolvendo a comunicação entre as interfaces e as entidades (atores) principais da nossa arquitetura. Iremos agora explicar em mais alto nível cada uma destas interfaces e seus serviços de interação.

4.2.1 A Interface *HCoreService*

A comunicação entre o *Front-End* e o *Core* é feita através da interface *HCoreService* a qual apresenta o *Front-End* como um cliente do *Core*, já que o primeiro irá se utilizar dos componentes disponíveis no segundo. Essa comunicação ocorre da seguinte forma: O *Front-End* inicia uma descoberta dos componentes acessíveis através do repositório mantido pelo *Core*, para daí escolher o que melhor se adequa a sua aplicação. Na fase de configuração, o usuário através do *Front-End* combina os componentes selecionados, utilizando serviços de configuração do *Core*.

Interfaces gráficas para manipulação de componentes # no *Front-End* são bem vindas para lidar com a configuração destes, obedecendo o padrão MVC (*Model-View-Controller*). A Visão corresponde aos serviços do *Front-End*, o Controlador corresponde aos serviços do *Core*, e o Modelo corresponde à representação usada internamente pelo Core para os componentes # configurados. A fase de publicação especifica quais os componentes criados pelo cliente deverão ser disponibilizados no *Core*, em alguma *location* registrada neste, para composição por terceiros. O serviço de recuperação busca os componentes selecionados pelo cliente.

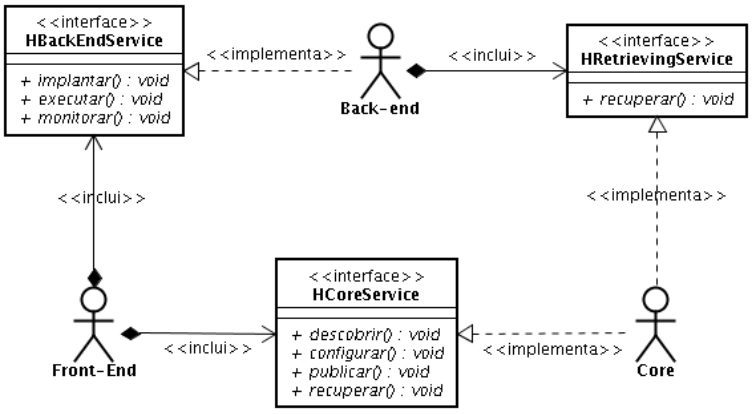

Figura 4.3: *HCoreService*

4.2.2 A Interface *HBackEndService*

A comunicação entre o *Front-End* e o *Back-End* compreende a interface *HBackEndService*, englobando três serviços: *implantação*, envolvendo a compilação dos componentes # na arquitetura suportada pelo *Back-End* e a instalação do código executável, preparando-o para a fase de produção; a *execução*, onde os componentes são carregados e executam na arquitetura; e a *monitoração*, para visualização do andamento da execução do componente #, visando depuração ou sintonização de

desempenho.

4.2.3 A Interface *HRetrievingService*

A comunicação entre *Back-End* e *Core* é feita através da interface *HRetrievingService*. O *Back-End* torna-se um cliente do *Core*, pois o primeiro deverá requisitar junto ao *Core* os códigos fonte para poderem ser montados em sua arquitetura. Portanto, o único serviço vislumbrado é o de recuperação.

A construção de plataformas de desenvolvimento baseadas na arquitetura *Hash* é realizada pela especialização de um conjunto de interfaces, com o objetivo dar suporte a várias **espécies de componentes**. Cada espécie descreve propriedades peculiares de certos componentes # que possuem interesses inter-relacionados, apresentando diferentes modelos de implantação e restrições de composição com outros componentes. Esta classificação se faz necessária tendo em vista que componentes # podem abordar diferentes classes de interesses, desde funcionais a não-funcionais, os quais não podem ser tratados de maneira uniforme. Exemplos de espécies serão apresentadas na descrição do HPE.

Uma vez que os componentes da arquitetura *Hash* são fracamente acoplados por meio de interfaces de serviço, torna-se possível uma sociedade de *Front-Ends*, *Back-Ends* e *Cores*, transparentemente localizados em um ambiente distribuído por meio de *Web Services* e tecnologias relacionadas, implementando as interfaces *HCoreService*, *HBackEndService*, e *HRetrievingService*. Como exemplo, um programador poderia usar *Web Services* para encontrar *Cores* compatíveis com uma classe de problemas cuja solução vem sendo configurada no *Front-End*. Uma aplicação médica localizaria apenas *Cores* relacionados às áreas de diagnósticos, biologia, fármacos, etc.

Ao achar os *Cores* compatíveis, seria possível acessar seus serviços para descobrir e configurar os componentes #. Além disso, seriam acessados *Back-Ends* apropriados para a execução da computação. Esta abordagem assemelha-se aos **Ambientes de Solução de Problemas** (*Problem Solving Environments* ou PSE's) [31], frequentemente adotados em ciências computacionais e engenharia com a finalidade de prover uma interface para o desenvolvimento de aplicações que seja mais próxima às abstrações no nível das aplicações, com as quais cientistas computacionais e engenheiros estão mais acostumados a lidar.

Em trabalhos futuros, deseja-se o uso da arquitetura *Hash* para integração de PSE's para a solução de grandes problemas em ciências computacionais e engenharia.

4.2.4 Componentes Abstratos e Concretos

A arquitetura *Hash* define dois tipos de componentes fundamentais. Os componentes concretos, ou componentes #, e os componentes abstratos.

Componentes abstratos compreendem componentes que não podem ser instanciados. Sua relação com componentes # é análoga a relação entre tipos abstratos de dados e módulos em linguagens estruturadas, ou a relação entre interfaces e classes em linguagens orientadas a objeto modernas. Por esse motivo, sua semântica é definida em termos de tipos existenciais. A forma geral de um componente abstrato no HPE é $\mathbf{C}\,[X_1{<:}\mathbf{T}_1, X_2{<:}\mathbf{T}_2, \ldots, X_n{<:}\mathbf{T}_n]$, onde $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_n$, $n{\geq}0$ são componentes abstratos com seus parâmetros supridos e $<:$ a relação de subtipos.

Um Componente Abstrato descreve um contrato que deve ser satisfeito pelos componentes # que o habitam. Uma formalização desse conceito baseado em Teoria das Instituições foi apresentado em [37]. Em uma mesma plataforma, deve existir um único componente # de um certo componente abstrato para cada combinação de parâmetros cuja implementação supõe-se apropriada. Por exemplo, um componente abstrato que descreve um canal de comunicação coletiva do tipo *All-To-All* pode estar habitado por vários componentes #, cada qual encapsulando uma implementação particular do canal apropriada a um certo tipo de plataforma de execução. Além disso, deseja-se o suporte ao controle *side-by-side*[1] de versões de um componente em uma mesma plataforma.

De maneira formal, as versões concretas (componentes #) de um componente abstrato definem instâncias específicas para cada componente abstrato aplicadas em contextos diferentes através do suprimento de sua lista de parâmetros tipos por componentes abstratos atuais. Assim, uma assertiva $\mathbf{C}\,[\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n]$ denota o componente # que implementa $\mathbf{C}\,[X_1{<:}\mathbf{T}_1, X_2{<:}\mathbf{T}_2, \ldots, X_n{<:}\mathbf{T}_n]$ especializado para ser aplicado em um contexto onde $X_1 = \mathbf{S}_1, X_2 = \mathbf{S}_2, \ldots, X_n = \mathbf{S}_n$.

Por exemplo, suponha o componente abstrato $\mathit{Channel}[X <: \mathit{Environment}, Y <: \mathit{Data}]$, cujos parâmetros de tipo X e Y denotam o ambiente para o qual o canal está implementado e o tipo de dado que será transmitido, respectivamente. Portanto, a partir desse componente abstrato é possível instanciar componentes # a partir da substituição de seus parâmetros por componentes abstratos que são subtipos de *Environment* e *Data*, definindo o

[1] Co-existência segura entre várias versões de um mesmo componente em uma mesma plataforma.

ambiente e estrutura de dados para o qual o componente é otimizado. Assim, um componente # que implementa $Channel[MPI, Array1D]$ seria especializado para transmitir vetores de uma dimensão através de MPI, enquanto outro que implementa $Channel[MPIBasic, Data]$, aplicável com segurança nesse contexto pois $Channel[MPIBasic, Data] <: Channel[MPI, Array1D]$, está implementado usando apenas as operações básicas do MPI e pode transmitir qualquer tipo de dado inclusive vetores de duas dimensões, sem muito refinamento. Exemplos mais práticos do uso de componentes abstratos e concretos serão apresentados no Capítulo 6.

O sistema de tipos proposto garante certo nível abstração aos programadores em relação aos detalhes da arquitetura sobre a qual um componente encontra-se implantado, deixando a responsabilidade pela escolha do componente mais adequado a um determinado contexto de execução à infraestrutura de componentes. Uma outra preocupação importante com respeito ao projeto de um sistema de tipos é a segurança (*safety*), propriedade que garante que programas bem formados executam de maneira correta. Esta dissertação não apresenta os detalhes formais do projeto do sistema de tipos de HPE, os quais serão detalhados em publicações posteriores a esta dissertação.

4.3 HPE: *Hash Programming Environment*

Baseado na Arquitetura Hash, o *framework* HPE foi proposto como uma base para implementação de ambientes de programação orientado a componentes # sobre a plataforma Eclipse. Sobre esse *framework*, temos instanciado o ambiente HPE, com a finalidade de construir aplicações paralelas (pois fazem computações simultâneas, em processos distintos) e distribuídas (pois os processos distintos podem estar localizados em *hosts* diferentes) de propósito geral, ou seja, as quais não estejam restritas a nichos específicos de aplicação.

O HPE, tem sido desenvolvido fazendo uso do GEF (*Graphical Editing Framework*) para a implementação de um *Front-End* que oferece um editor visual para composição de componentes # por sobreposição, obedecendo ao padrão MVC. Como conseqüência deste trabalho, planeja-se adaptar as interfaces entre *Front-End*, *Back-End* e *Core* para *Web Services*. Devemos porém ressaltar que o protótipo atual já implementa locations como *Web Services* acessíveis a partir de *Core's*.

O HPE estende o *framework* HPE pela sua especialização visando o suporte a um conjunto específico de espécies de componentes, para o suporte à programação

paralela de propósito geral sobre *clusters* de multiprocessadores, definidos a seguir.

i. **Arquiteturas**: Descrevem arquiteturas computacionais onde os componentes irão ser implantados. Suas unidades representam o nós de computação.

ii. **Ambientes**: Descreve a tecnologia de *software* utilizada a qual permite o paralelismo em uma arquitetura alvo. Unidades de componentes # desta espécie descrevem os serviços de um determinado ambiente, acessíveis aos processos envolvidos na computação.

iii. **Estruturas de Dados**: Uma estrutura de dados, possivelmente paralela, manipulada por uma ou mais computações em uma aplicação.

iv. **Computações**: Especifica a computação paralela, denotando um interesse funcional necessário em aplicações.

v. **Sincronizadores**: O meio o qual computações sincronizam seu estado, ou comunicação. Canais ponto-a-ponto, operações de comunicação coletiva e *bindings* para chamada remota de procedimentos são típicos exemplos de sincronizadores.

vi. **Aplicações**: Uma computação relativa a uma aplicação final, a qual inclui o código principal. Uma aplicação resulta no código contendo o método principal (*main*) a ser executado. Este método deverá chamar em cadeia as computações referentes à aplicação. Todo componente da espécie aplicação é também considerado da espécie Computação.

vii. **Qualificadores**: Uma característica não-funcional de um componente # a qual é relevante para a sua semântica ou desempenho. Por exemplo: o modo de comunicação de um canal (síncrono ou assíncrono); a estratégia de distribuição de uma grande estrutura de dados sobre os processadores; o algoritmo utilizado para solucionar sistemas lineares esparsos. Na verdade, qualificadores representam um papel importante para os tipos de componentes abstratos no sistema de tipos de componentes do HPE.

No *Front-End*, a configuração de componentes # a partir da sobreposição de outros deve ser realizada no nível dos componentes abstratos. Além disso, um componente concreto pode ser instanciado a partir de um componente abstrato. Cabe ao *Back-End* do HPE a responsabilidade de encontrar a implementação

adequada daquele componente abstrato disponível na arquitetura alvo, o que será detalhado no Capítulo 5. Isso permite lidar com heterogeneidade comum em arquiteturas de computação de alto desempenho. Abaixo, uma breve apresentação da nossa implementação dos componentes *Front-End*, *Back-End* e *Core*.

4.3.1 *Front-End*

O *Front-End* é o componente cliente do nosso sistema. Originalmente implementado na linguagem Java, ele faz uso de GEF (*Graphical Environment Framework*) para a criação de um *plug-in* para a plataforma Eclipse. Este *plug-in* torna possível a montagem de componentes # através de uma interface visual.

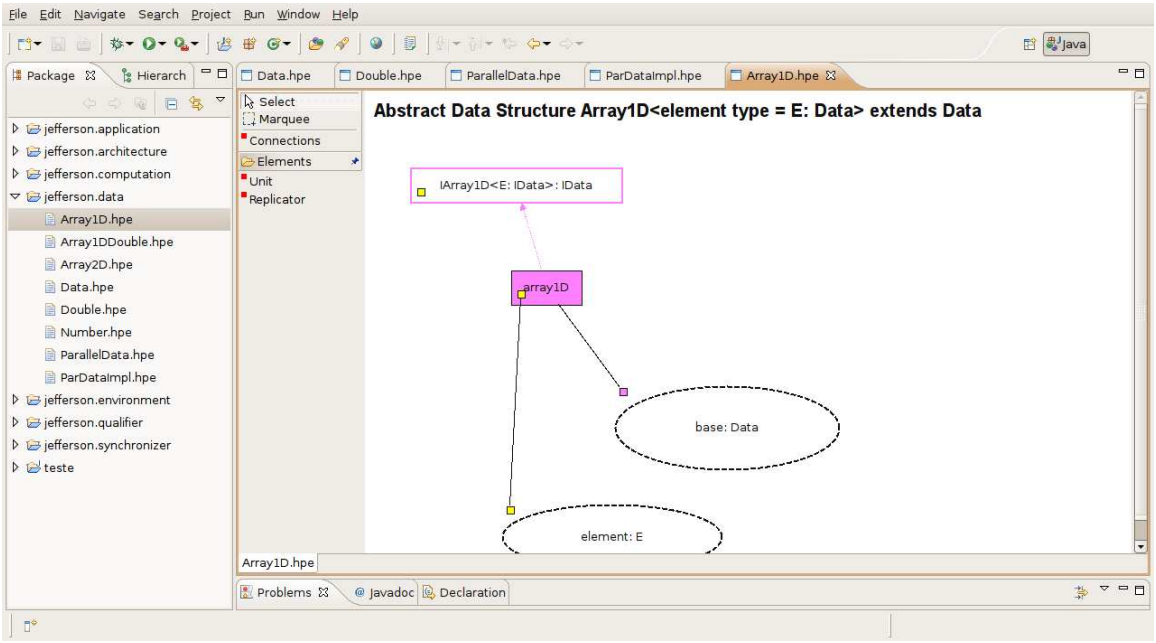

Figura 4.4: *Plug-in HPE executando na plataforma Eclipse*

Na Figura 4.4 apresentamos a montagem de um componente usando as abstrações gráficas embutidas no HPE. Essas abstrações facilitam ao desenvolvedor criar projetos de componentes paralelos que serão implantados em um *Back-End* específico. No Capítulo seguinte, a aplicação exemplo explanada anteriormente será projetada, para um cenário mais genérico no nosso *plug-in*.

4.3.2 *Core*

O *Core* resume-se em um respositório de códigos fontes, monitorados por um serviço *web* implementado em Java, disponível em um servidor Apache Tomcat. Nesta versão da implementação da dissertação, os códigos fontes necessários à

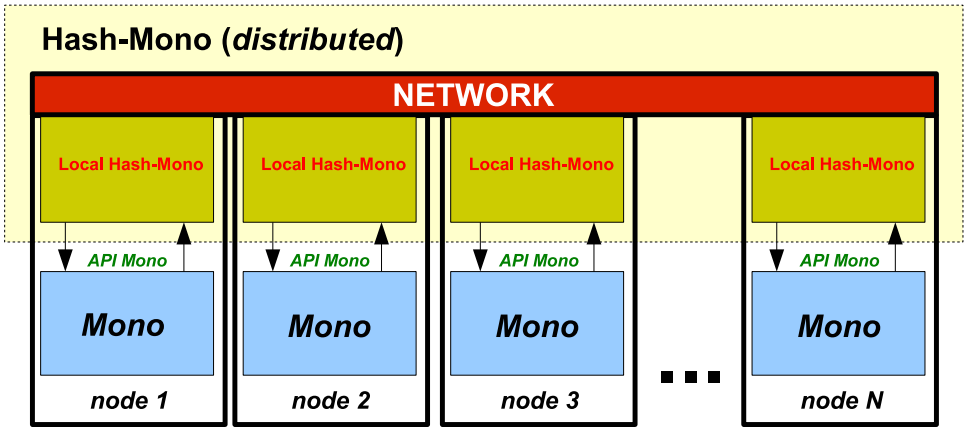

Figura 4.5: *Mono embutido na arquitetura Hash*

compilação dos componentes são acessados através da chamada de um método desse serviço, e baixados para uma pasta específica do *Back-End*.

Inicialmente, o *Core* armazena apenas os componentes necessários para a implementação e execução do teste de conceitos. Trabalhos futuros almejam sua extensão para inclusão de componentes básicos a outros nichos.

4.3.3 *Back-End*

Um importante aspecto a considerar no desenvolvimento deste projeto é a implementação do *Back-End*. O *Back-End* constitui uma infra-estrutura de componentes no sentido usual, responsabilizando-se pelos estágios do ciclo de vida a partir da implantação do componente. Esta infra-estrutura deve suportar a interoperabilidade entre componentes escritos em várias linguagens e o controle *side-by-side* de versões de componentes.

A plataforma CLI (*Common Language Infrastructure*) tem sido proposta pela indústria como um padrão para infra-estrutura de componentes com suporte embutido a código gerenciado, interoperabilidade de linguagens e controle de versão de componentes [70]. CLI incorpora importantes desenvolvimentos na área de compilação de linguagens de programação para linguagens intermediárias executáveis em máquinas virtuais, tendo como pressuposto aproximar o desempenho de código nativo através da compilação *Just-In-Time*. A interoperabilidade entre linguagens é obtida pela adoção de uma linguagem intermediária com suporte a um sistema de tipos polimórfico. A distinção entre código gerenciado e não-gerenciado (código nativo) é importante para aplicações de alto desempenho, onde em geral a maior parte do tempo de execução dos programas é gasto em trechos pequenos e

críticos do software.

As principais implementações do padrão CLI são o *.NET* (comercial) [44], e o Mono (código aberto) [7]. Uma alternativa para implementar o *Back-End* do HPE seria então extender o Mono para o suporte a componentes que estejam distribuídos, os componentes #. Para isso, seria suficiente a generalização do GAC (*Global Address Cache*), módulo responsável pelo gerenciamento dos componentes implantados em uma máquina, para torná-lo um programa distribuído em nós de um *cluster*, ao qual chamaríamos de DGAC (*Distributed Global Address Cache*). A arquitetura vislumbrada está na Figura 4.5. Note-se que se propõe, a princípio a extensão não-invasiva do Mono, utilizando os serviços "Mono Embutido", fornecido pela sua implementação atual. Detalhes sobre a implementação do HPE sobre CLI e Mono é o assunto a ser abordado no próximo capítulo.

Capítulo 5

Projeto e Implementação do *Back-End* do HPE

Como discutido no Capítulo 4, o HPE é formado por três módulos principais: o *Front-End*, o *Back-End* e *Core*. O principal objetivo desta dissertação é o projeto e implementação de um *Back-End* particular, para prova de conceito. Este é responsável pela implantação e execução de um componente # em uma plataforma de execução paralela, controlando as diversas versões do mesmo (se houver).

Em um breve resumo, as principais tecnologias usadas nesta implementação são as seguintes: *Web Services* Mono, instalados no servidor livre XSP, próprio à aplicações ASP.NET; O SGBD (Sistema Gerenciador de Banco de Dados) MySQL para armazenamento de informações; a biblioteca de comunicação remota entre objetos escritos em Mono (Mono RPC) e a tecnologia Java.

As seções seguintes detalham tecnologias envolvidas e a solução de implementação para o *Back-End* proposto. Assim, a Seção 5.1 apresenta uma visão geral sobre a especificação CLI e a plataforma Mono. A Seção 5.2 explana a tecnologia de *Web Services*. A Seção 5.3 descreve como um componente # é implementado sobre a plataforma CLI/Mono, ilustrando a abordagem por meio de um exemplo. A Seção 5.4 descreve o DGAC (*Distributed GAC*), uma generalização do GAC (*Global Assembly Cache*), módulo do Mono responsável pelo controle dos componentes implantados, de forma a suportar componentes paralelos (componentes #). A Seção 5.5 mostra como o *Back-End* comunica-se com o *Front-End*. A Seção 5.6 explica o procedimento de implantação de um componente # no *Back-End* de forma geral. Finalmente a Seção 5.7 descreve um pequeno exemplo da execução de um componente #, em alto nível.

5.1 CLI - *Common Language Infrastructure*

O CLI (*Common Language Infrastructure*) é uma especificação padronizada para ambientes de execução virtual, baseado em componentes, que surgiu a partir da iniciativa de grandes empresas de *software*, especialmente a *Microsoft*. A ECMA (*European Computer Manufacters Association*) é uma organização privada internacional de padronizações a qual mantém estreitas relações com o ISO (*International Standards Organization*). A ECMA é responsável pelo CLI (o qual é descrito na norma técnica ISO/IEC 23271:2003, acompanhado de um relatório técnico intitulado ISO/IEC 23272:2003). A linguagem C#, descrita pela norma técnica ISO/IEC 23270:2003, é usada nesta dissertação.

A especificação CLI define uma ambiente na qual códigos em linguagem de alto-nível diferentes possam ser executados independentemente de plataforma. O CLI é uma *especificação* e não deve ser confundido com uma de suas implementações, o CLR (*Commom Language Runtime*), que define uma máquina virtual capaz de compilar e executar código C# para diversas arquiteturas. O CLR permite que programadores ignorem diversos detalhes sobre a arquitetura específica que irá executar o código. O CLR é uma implementação proprietária desenvolvida pela *Microsoft* exclusivamente para sistemas *Windows*. A *Microsoft* no entanto liberou uma versão livre de uma implementação da especificação CLI, chamada SSCLI (*Shared Source CLI*). Essa, porém, apresenta diversas limitações comerciais e não supre várias necessidades de alguns programadores, sendo aconselhada apenas como uma versão voltada ao ensino e pesquisa.

O projeto Mono [7], liderado pela *Novell*, apresenta um conjunto de ferramentas de *software* compatíveis com o CLI. O Mono dispõe de uma compilador próprio para C# e uma máquina virtual semelhante ao CLR. Sua grande vantagem é compatibilidade entre diversos sistemas operacionais (incluindo Linux, *Windows* e *FreeBSD*) bem como um amplo suporte no que diz respeito as bibliotecas existentes no *framework* da *Microsoft*, o .NET. Além disso, Mono pode rodar aplicações Java, Python e é uma iniciativa livre (*Open Source*).

O Mono compila código C# fazendo uso das bibliotecas existentes em seu repositório. O código objeto é então submetido a sua máquina virtual (CLR) e depois ao sistema operacional apropriado (Figura 5.1).

O uso de um padrão aprovado por um comitê de renome como o ECMA é muito importante para o nosso trabalho pois garante que a linguagem de implementação

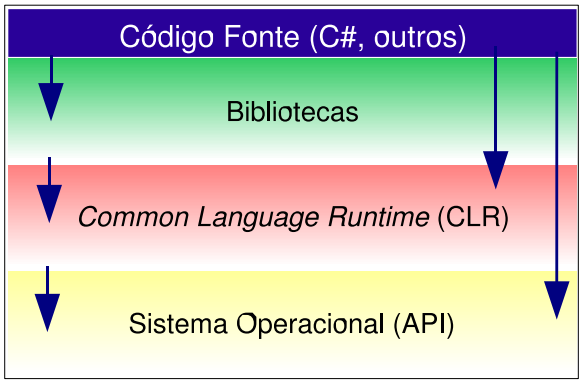

Figura 5.1: *Arquitetura Mono simplificada*

que usamos (no caso o C#) apresente várias vantagens provindos do CLI as quais podemos enumerar:

i. **Sistema compartilhado de tipos e geração de linguagem intermediária** - A especificação CLI provê uma linguagem intermediária chamada CIL (*Common Intermediate Language*), que suporta um sistema de tipos polimórficos chamado CTS (*Commom Type System*). O CTS é um padrão que permite programas escritos em linguagens diferentes compartilharem dados. Ele define um conjunto de tipos de dados independentes de arquitetura. Assim, um programador não se preocuparia se um tipo inteiro em C# seria compatível com um tipo inteiro em *Visual Basic* pois o mesmo é mapeado em um tipo comum.

   Além disso, CTS permite a descoberta de tipos atuais de variáveis no momento da execução, uma funcionalidade usada também nesta dissertação no processo de criação dinâmica de objetos.

ii. **Pacotes portáveis** (*Assemblies*) - Em CLI, componentes individuais são empacotadas em unidades chamadas de *assemblies*. Cada *assembly* pode ser carregada dinamicamente no ambiente de execução, através de um disco local, pela rede ou até mesmo criado *on-the-fly* por um programa. Nesse último caso, o Mono usa a API do *gacutil* (*software* utilitário) para gerar *dinamicamente* as unidades. Assim, *assemblies* podem ser utilizados por diferentes máquinas virtuais em diferentes ambientes.

iii. **Comunicação remota** - Conjunto de interfaces disponíveis para comunicação remota entre componentes escritos em linguagens padronizadas pelo CLI. Tais

componentes devem obedecer uma série de restrições, dentre elas a de ser *serializável* (implementar a interface *Serializable*).

iv. **Convenções de nomeação** - Um *assembly* gerado segundo a especificação CLI possui um conjunto de regras de nomeação, garantindo que várias versões de um mesmo componente coexistam em uma mesma máquina virtual. Os *assemblies* são armazenados no GAC (*Global Assembly Cache*), um repositório alocado em cada máquina virtual (vide seção 5.4).

v. ***Just-in-time compilation* (JIT) ou compilação sob demanda** - O Código interpretado por máquinas virtuais, chamado de linguagem intermediária (*bytecodes* por exemplo), não depende especificamente da arquitetura o qual foi compilado. O mesmo pode teoricamente executar em qualquer *hardware* contanto que possua uma versão da máquina virtual compatível. Visando otimização, a compilação sob demanda tem por objetivo transformar o código intermediário em código nativo à arquitetura, apenas na primeira vez em que for executado (*just-in-time*). O código nativo é então armazenado em memória principal, aumentando o desempenho da aplicação.

Em nossa implementação do *Back-End* escolhemos uma plataforma CLI, o Mono, principalmente devido as suas vantagens no que diz respeito à área de Computação de Alto Desempenho. A primeira delas é a interoperabilidade entre linguagens fornecida pela especificação. Diversos modelos em CAD são implementados em várias linguagens diferentes e é desejável a intercomunicação entre eles, visando o compartilhamento de código. Outras vantagens compreendem no controle *side-by-side* de versões, permitindo que a mesma versão de um componente # coexista consistentemente com suas outras versões e a compilação sob demanda (JIT) existente em Mono. O JIT permite que trechos de código sejam compilados para linguagem nativa. Muitas vezes em CAD pequenos trechos de código são responsáveis por um grande parte da computação, ocupando bastante processamento. A transformação do código intermediário responsável por esta computação em código nativo e o seu posterior armazenamento em memória principal auxilia no aumento do desempenho da aplicação, necessidade extremamente importante para CAD.

Figura 5.2: *Principais papéis no funcionamento de um Web Service (figura por H. Voormann)*

5.2 Tecnologia de *Web Services*

A W3C [77] define um serviço *web* como um sistema de *software* capaz de conectar aplicativos através de uma rede de computadores. O uso da linguagem XML (***EX****tensible* ***M****arkup* ***L****anguage*) permite a um serviço *web* escrito em uma linguagem como C#, por exemplo, ser "consumido" (jargão popular na literatura no que diz respeito a chamada de um serviço) por um cliente escrito em outra linguagem. Clientes acessam os servidores através de *proxies* gerados a partir de um arquivo WSDL (*Web Service Description Language*). O arquivo WSDL é fornecido pelo servidor como uma representação de seus serviços (os quais podem estar implementados em qualquer linguagem) na forma de XML. As mensagens trocadas entre clientes e servidores seguem o padrão SOAP (protocolo para troca de mensagens baseadas em XML).

De acordo com a Figura 5.2, um serviço *web* é registrado em um *Service Broker* chamado UDDI. O UDDI é responsável em tornar pública, através da representação WSDL, as informações do serviço (*Service Provider*), funcionando como uma espécie de "páginas amarelas". Uma vez registrado, o cliente (*Service Requester*) requisita o WSDL e passa a consumir o servidor via mensagens SOAP.

O uso de *Web Services* neste trabalho foi proposto como uma solução para a questão sobre o acoplamento fraco e distribuído entre os módulos HPE (*Front-End, Back-End* e *Core*). O WSDL permite que esses três módulos troquem mensagens independentemente da linguagem em que foram implementados, podendo residir em servidores distintos. O *Front-End* por exemplo, está sendo desenvolvido em Java e o *Back-End* em C#. A partir do arquivo WSDL, o *Front-End* gera classes (*proxies*, com o auxílio do *software* AJAX) em sua linguagem original para comunicação

remota com o *Web Service*, implementado em C#, instalado no servidor.

5.3 Representação de Componentes # no Mono

Como descrito no Capítulo 4, *componentes* # devem ser instanciados a partir de *componentes abstratos*, provendo-se valores atuais para seus parâmetros de tipos. Além disso, um componente # é formado de unidades, implementações concretas de suas unidades correspondentes do componente abstrato, as quais estão implantadas em processadores diferentes, tornando o mesmo naturalmente paralelo.

Na nossa implementação em Mono, as unidades de um componente # representam classes concretas escritas na linguagem de programação C#. Essas classes são implementações das interfaces associadas a suas unidades respectivamente correspondentes a componentes abstratos. Portanto, as unidades de um componente abstrato são interfaces escritas na linguagem C#. Como todo componente # é necessariamente a implementação de um componente abstrato então cada uma de suas unidades implementam uma unidade correspondente de um componente abstrato. Vejamos um exemplo intuitivo na Figura 5.3.

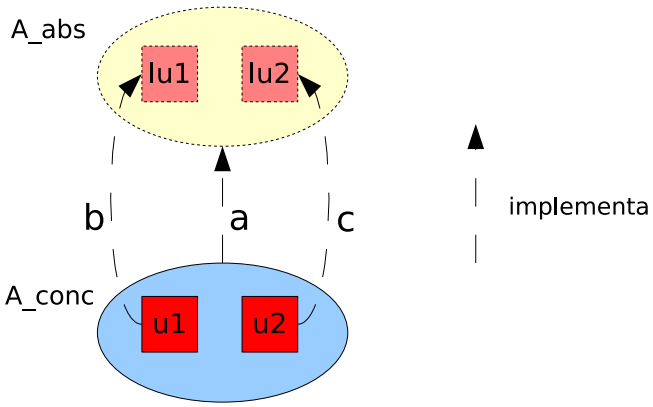

Figura 5.3: *Relação entre componentes # e componente abstratos.*

Seja o componente # concreto *A_conc* e o componente abstrato *A_abs*. O componente # concreto *A_conc* é a implementação de um componente abstrato (Figura 5.3, relacionamento `a`) *A_abs*. Para que isto seja verdade, cada uma das unidades de *A_conc* (*u1* e *u2*) deve implementar uma interface em *A_abs*. No exemplo temos que a unidades *u1* e *u2* implementam as unidades *Iu1* e *Iu2* pertencentes ao componente abstrato *A_abs* (Figura 5.3, relacionamentos `b` e `c`). Sendo assim, uma unidade do componente *A_conc* representa um classe C# que implementa uma interface a qual é uma unidade de *A_abs*.

Uma **unidade** especifica a menor parte de um componente #, a qual deverá

conter informações inerentes à execução deste, as quais dependem exclusivamente da **espécie** do componente. Uma espécie de componente (vide Seção 4.3) define as características de um componente e a forma como ele deverá ser tratado pelo *Back-End*, definindo suas restrições e modelos de implantação. Portanto, a forma de implantação de um componente # depende da espécie de componente a qual o mesmo pertence. Vejamos agora como esta dissertação implementou componentes abstratos e componentes # conforme as espécies descritas no Capítulo 4 obedecendo o que foi descrito nos parágrafos anteriores desta seção.

i. **Computação**: As unidades de componentes # da espécie *Computação* fornecem código fonte de classes que devem implementar a interface *ComputationKind* e a interface correspondente a unidade do componente abstrato o qual implementa, sobrescrevendo método *compute*. Este método corresponde a semântica de ativação para fatias de unidades de componentes # da espécie *Computação*. Representa a computação realizada. Portanto, computações são definidas como classes em C# que dão origem a bibliotecas Mono, após sua compilação. Estas bibliotecas são instaladas no GAC e reutilizadas por outros componentes, como veremos no exemplo usado para teste de conceito desta dissertação, no Capítulo 6.

ii. **Aplicação**: No nosso modelo, componentes da espécie *Aplicação* são especializações dos componentes *computação*, ou seja, aplicações também são computações. Portanto, classes associadas a unidades de componentes # da espécie *Aplicação* devem sobrescrever o método *compute*. Além disso, aplicações também devem possuir um procedimento *Main*, responsável pela execução do programa. Como decisão de projeto, componentes geram duas classes diferentes: uma responsável em sobrescrever o método *compute*, exigido pela interface da espécie *computação* e outra classe responsável em criar o código *Main*, capacitado em executar o componente ao chamar o método *compute* da classe que implementa a computação. As duas são compiladas em um mesmo arquivo executável.

   Por exemplo, seja um componente *A*, da espécie *Aplicação*. Este componente irá gerar duas classes: `A_compute` e `A_main`, as quais serão compiladas pelo DGAC em um mesmo executável. A classe `A_main` irá criar uma instância da classe *A_compute*. Então, em seu método *Main*, será chamado o seu método `compute`. No Capítulo 6, veremos mais detalhes deste processo em

uma aplicação usada como teste de conceito.

iii. **Estruturas de Dados**: As unidades de componentes da espécie *Estruturas de Dados* são objetos Mono responsáveis por representar estruturas de dados que serão usados pela computação. A nossa implementação usa objetos C# para encapsular tipos como matrizes, vetores, dentre outros. Uma implementação de um componente *Estruturas de Dados* é uma classe C# que ao ser compilada pelo DGAC gera uma biblioteca reusável por outro componentes. Portanto, um componente que efetua a multiplicação de matrizes pode usar como fatia interna um objeto gerado a partir de uma biblioteca que representa uma matriz. A implementação de estruturas de dados como tipos abstratos de dados é desejável porém não mandatória.

iv. **Arquitetura**: Os componentes da espécie *Arquitetura* são implementados com o uso de classes associadas a cada unidade, as quais oferecem serviços que permitem obter informações sobre aspectos estáticos e dinâmicos da plataforma de execução, como, por exemplo, os nós nas quais cada processo (unidade de um componente da espécie aplicação) deverá executar, a carga de trabalho dos nós, etc. Sua classes geram bibliotecas Mono reusáveis por outros componentes.

v. **Ambiente**: Esta espécie diz respeito as informações, oferecidas por um serviço implementado por uma classe Mono, sobre a tecnologia de *software/middleware* que está instalada na arquitetura gerenciada pelo *Back-End* (MPI, *OurGrid*, PVM, etc) para habilitação do processamento distribuído paralelo. Um componente ambiente tem como componente interno o tipo arquitetura utilizada e é parametrizado por este, de forma que um ambiente pode estar otimizado para uma determinada plataforma de execução. O componente ambiente irá gerar bibliotecas (*assemblies*) associadas a cada uma de suas unidades.

vi. **Sincronizadores**: Unidades de componentes da espécie *Sincronizador* são classes que devem implementar a interface *SynchronizerKind*, sobrescrevendo o método *synchronize*. Um sincronizador abstrato está parametrizado pelo ambiente sobre o qual executará, de forma que sua implementação é ótima para um determinado ambiente. O método *synchronize* implementa a semântica de ativação de fatias de unidades de componentes # da espécie *Sincronizador*.

Como exemplo de uso, um componente que faz a multiplicação de matrizes de forma distribuída, por exemplo, irá possui como fatia interna um sincronizador responsável em "espalhar" os dados entre os processadores, seguindo uma política definida por uma estratégia de particionamento.

vii. Qualificadores: Unidades desta espécie documentam características não funcionais em arquivos de formato XML. O XML é então usado para configurar a forma de execução de outras unidades, definindo modelos de comunicação e estratégias de distribuição de dados, por exemplo. Os arquivos XML são lidos através de interfaces Mono para manipulação de arquivos XML, existente na API do .NET. Componentes que utilizem um estratégia de particionamento utilizam esta interface e interpretam o XML, definindo como será o particionamento de seus dados pela arquitetura.

Figura 5.4: *Interface gráfica do HPE para criação de Espécies de Componentes*

A Figura 5.4 apresenta a interface gráfica disponível no *Front-End* para a criação de componentes # de uma Espécies de Componentes (*Component Kind*) selecionada. Cada Espécie criada gera um arquivo XML que é visualizado por meio de abstrações gráficas (Java 2D).

5.4 DGAC - *Distributed Global Assembly Cache*

Antes de explicar o que seria o DGAC, temos que deixar claro como funciona o GAC. O GAC (*Global Assembly Cache*) é um repositório de bibliotecas (no caso, arquivos de extensão `dll` ou *assemblies*) que podem ser usadas na compilação e execução do código fonte. Se um *assembly* deve ser acessado por múltiplas aplicações, então ele deve ser colocado em um diretório na qual a CLR (Máquina

Virtual CLI) tenha conhecimento. Este diretório é o GAC. Ao usuário, é permitida a instalação de *assemblies* personalizados. No entanto, a instalação de um *assembly* no GAC não é feita de uma forma ingênua, apenas copiando o arquivo dll em seu diretório. *Assemblies* os quais desejam ser compartilhados devem possuir "nomes fortes" constituídos de chaves com números muito grandes. Estas chaves são geradas pelo comando `sn` resultando em arquivos que são anexados ao *assembly* no momento de sua compilação. Feito isto, o *assembly* está apto para ser instalado no GAC. Esta tarefa exige o auxílio do programa *gacutil*, através de uma simples linha de comando. O Mono oferece suporte para instalação e criação de *assemblies* no GAC, bem como ferramentas para criação de "nomes fortes".

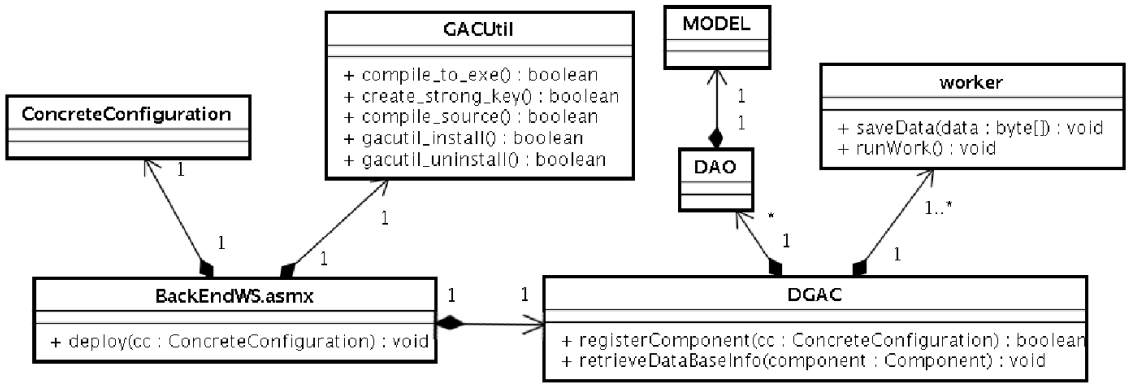

Figura 5.5: *Diagrama de classes simplificado do relacionamento entre DGAC e o Web Service*

O DGAC (*Distributed Global Assembly Cache*) é a extensão distribuída do GAC, proposta para suporte ao *Back-End*. DGAC constitui em n instalações separadas do Mono (cada uma com seu diretório GAC), em uma arquitetura distribuída de n processadores. Cada instalação é gerenciada por um único *Web Service* que faz interface com o *Front-End*. As informações relativas aos componentes # implantados são atualmente mantidas em um banco de dados, usando a ferramenta MySQL.

A Figura 5.5 mostra as principais entidades envolvidas na implementação do DGAC:

i. *BackEndWS.asmx*: esta classe diz respeito ao *Web Service* que ficará publicado no servidor. Seu método *deploy* recebe uma configuração de um componente # montado pelo cliente (*Front-End*). Esta configuração está representada através de um formato XML.

ii. *GACUtil*: classe Facade de comunicação com a ferramenta *gacutil*.

Responsável pela compilação e geração de bibliotecas e executáveis além da manipulação de arquivos.

iii. *DAO e MODEL*: classes que implementam o padrão arquitetural de comunicação com o nosso SGBD.

iv. *Worker*: processos independentes responsáveis em monitorar um GAC particular, recebendo requisições do DGAC.

v. *DGAC*: classe monitora dos processos *Worker* responsável em analisar a configuração concreta em XML, requisitar os arquivos fontes do *Core* e resolver as dependências do componente a ser instalado.

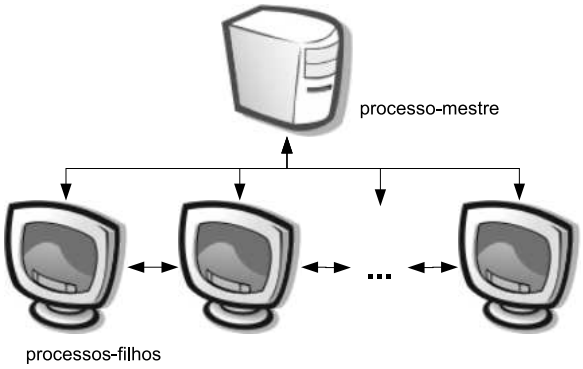

Figura 5.6: *O processo-mestre comunica-se com os processos-filho através da implementação de RPC (Remote Procedure Call) em Mono. Cada um dos processos-filho troca dados entre si por meio do ambiente instalado na arquitetura do Back-End.*

Independente da arquitetura, cada processador recebe uma instalação do Mono e conseqüentemente um GAC. Cada GAC é monitorado por um processo criado pelo DGAC (vamos chamar este processo, de processo-filho). Estes processos são iniciados independentemente e permanecem aguardando por solicitações do processo-mestre encontrado na máquina que faz a comunicação externa com o *Front-End*, ou seja a máquina a qual está instalado o *Web Service*. Na seções seguintes, apresentaremos como componentes # são implantados e postos a executar no Back-End, tornando mais claro o papel destes processos.

5.5 A Interface entre o *Front-End* e o *Back-End*

Como anteriormente mencionado, o *Front-End* irá comunicar-se com o *Back-end* através da tecnologia de *Web Services*.

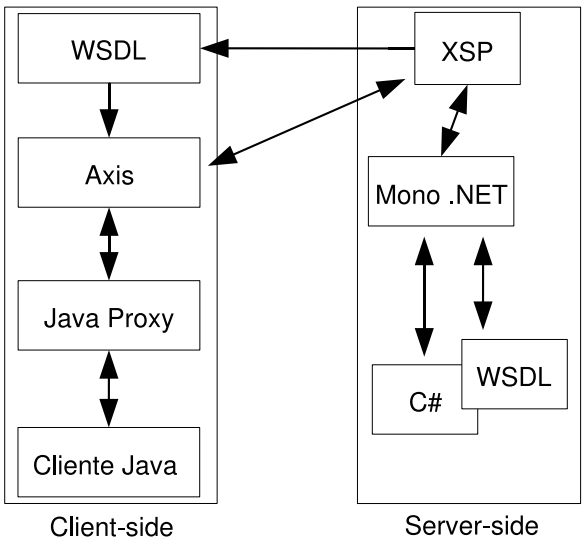

Figura 5.7: *Cliente (Front-End) e servidor (Back-End) envolvidos através de Web Services*

Para que seja possível publicar um serviço *web*, faz-se necessária a existência de um servidor que suporte a linguagem do serviço. Nesta dissertação, o *Back-End* está sendo implementado em C#. A linguagem C# provê um ótimo suporte a serviços *web* e é aceita por vários servidores *containers* dentre eles o Apache Tomcat, ISS (da própria *Microsoft*) e o xsp. O xsp é um servidor livre, simples, que roda tanto na plataforma Linux quanto *Windows* e por isso foi escolhido como nosso servidor de *Web Services*.

Na figura 5.7, um servidor xsp remoto hospeda uma interface C# a qual disponibiliza um conjunto de serviços que deverão ser transformados em uma linguagem intermediária, o WSDL (*Web Service Description Language*). O WSDL intersecciona o código C# apenas no que se diz respeito ao código que deverá ficar disponível no servidor xsp. O servidor xsp torna público o arquivo WSDL o qual pode ser transformado no lado do cliente em um *proxy*. Os *proxies* nada mais são do que representações locais ao cliente de um serviço que se encontra remoto. Em Java, e no nosso exemplo, o *framework Axis* [8] provê um conjunto de bibliotecas responsáveis em transformar o arquivo WSDL em um conjunto de classes proxy. O cliente, então, consome o serviço remoto fazendo simples chamadas de método às classes *proxy*.

A representação do nosso *Web Service* em um arquivo WSDL também torna possível que outras linguagens gerem seus próprios *proxies* e também consumam o serviço. Essa separação arquitetural entre *Front-Ends* e *Back-Ends* possibilita

o desenvolvimento de um conjunto de *Front-Ends* independentes da linguagem do serviço.

5.6 Implantação de um Componente # no *Back-End*

O principal serviço a ser consumido pelo cliente é o de instalar uma configuração concreta no *Back-End* (Figura 5.8, operação *implantar*), concretizando a implantação de um componente #.

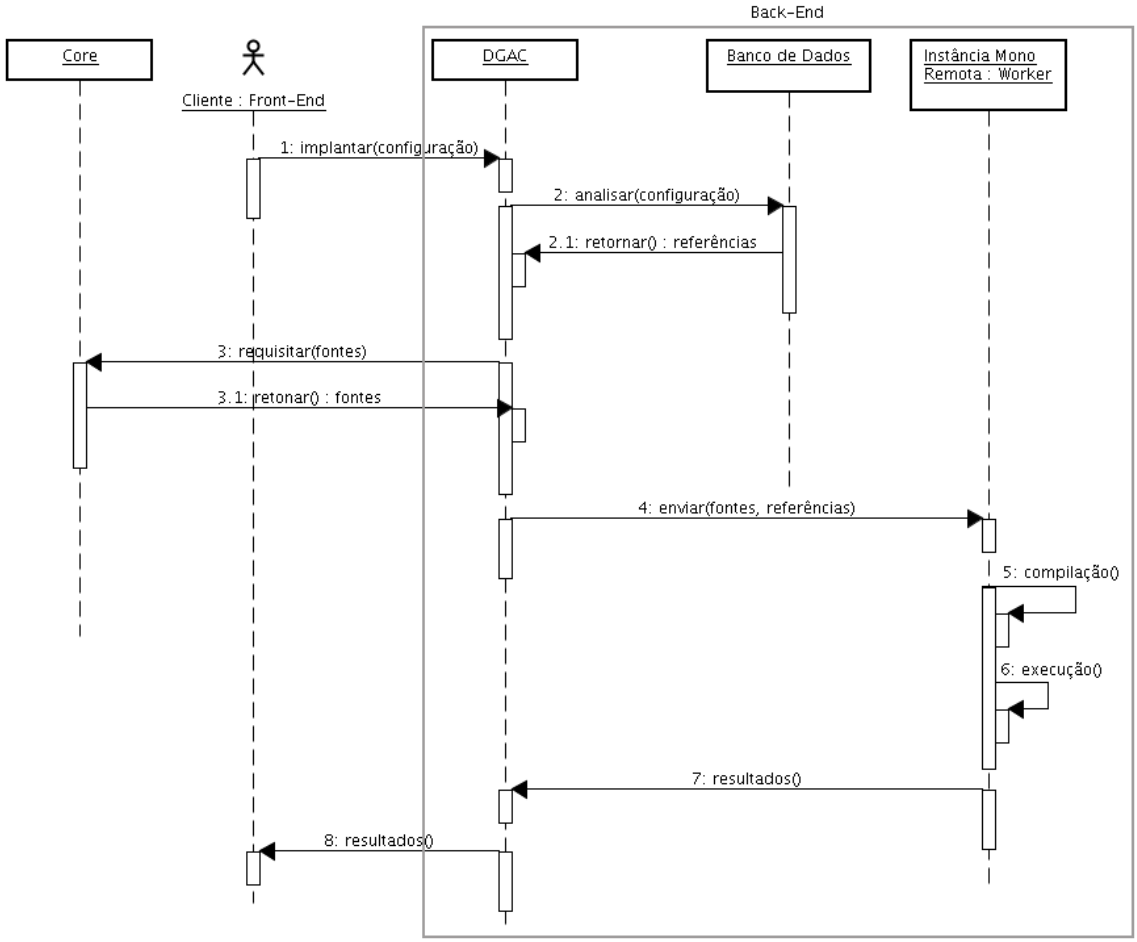

Figura 5.8: *Cenário de implantação de um componente #*

O cliente monta uma configuração de um componente # através do *Front-End* e submete-o ao serviço de implantação encontrado no *Back-End*, através de um objeto, atualmente serializado em XML. O DGAC deve então analisar esta configuração e resolver as dependências necessárias à compilação dos códigos fontes de suas unidades. Para isso, o papel do processo-mestre é receber a configuração de um componente # a ser instalado no *Back-End*. Esta configuração é analisada por

um método específico, o qual procura no banco de dados as descrições para seus componentes abstratos internos, os quais devem estar previamente instalados no DGAC.

Depois de analisar as informações no banco de dados, o processo-mestre está ciente onde (nós) estão instalados os componentes necessários (dependências) ao novo componente recebido como parâmetro. As dependências das unidades de um componente # correspondem aos *assemblies*, previamente instalados no DGAC, onde estão o código objeto da interface associada a unidade correspondente do componente abstrato que a implementa, além das interfaces associadas as unidades que constituem suas fatias, provenientes de seus componentes internos abstratos.

As versões concretas dessas interfaces (classes dos componentes # que implementam os componentes abstratos citados) são resolvidas em tempo de inicialização, discutido na próxima seção. Os códigos fonte das unidades do componentes # sendo implantado são requisitados pelo DGAC no *Core*, através de uma URI que disponibiliza um serviço (web service) de busca. Além de resolver estas dependências, o processo-mestre saberá para qual processo-filho deverá enviar o código fonte referente a unidade a ser compilada. Ele invoca remotamente o método *saveData*, existente em cada processo-filho. Este método recebe o código fonte (*array* de bytes), a lista de referências e o nome da unidade (arquivo *.cs*). O código fonte é **salvo** em uma pasta temporária, **compilado** e **executado**. Estes três passos são efetuados por uma classe dedicada em operações de *I/O*. Esta classe comunica-se com *shell* (console) do Linux e chama o utilitário *gacutil* para instalação das dlls e os comandos Mono `mcs` e `mono` para compilação e execução, respectivamente.

A lista de referências é transformada em uma *string* de compilação e é salva em um arquivo de extensão *.rsp*. Este tipo de arquivo funciona como um *build* para a aplicação. Ele é chamado pelo comando `mcs`, gerando o arquivo executável. Os arquivos executáveis gerados, no caso de componentes das espécie Aplicação, uma vez instalados podem ser executados pelas instâncias Mono dispostas em cada nó do cluster e e gerenciadas pelo *DGAC*. Os resultados da computação são enviados de volta ao DGAC, que os encaminha ao *Front-End*.

5.7 Execução de um Componente # no *Back-End*

Na fase de execução, dois métodos são de extrema importância: o método de resolução dos componentes internos de um componente concreto a ser executado, e o método de criação dinâmica de objetos relacionados aos componentes internos do

componente concreto encontrados no primeiro método.

Quando o método de resolução executa, ele inclui entradas no banco de dados sobre as implementações dos componentes internos encontrados para o componente concreto de entrada. Em seguida, o método de criação dinâmica pesquisa no banco de dados qual a implementação atual salva de cada componente interno para ser carregada. Obviamente, esse procedimento só ocorre na primeira vez que a unidade do componente é demandada. As demais chamadas apenas recuperam os dados já salvos em banco, para posterior criação dos objetos relacionados.

As subseções seguintes detalham estes dois métodos explicitando seu relacionamento. Um exemplo prático de seu uso será apresentado no próximo capítulo.

5.7.1 O Método de Resolução

Vamos agora analisar, em alto nível, o que ocorre no método de resolução. Neste método, o objetivo é encontrar a implementação de um componente abstrato (id_inner) interno a um componente concreto ($id\_concrete$). Achada esta implementação, uma entrada referente a mesma ($id\_concrete\_actual$) é adicionada ao banco de dados, para ser usada no momento da criação de um objeto.

A procura do componente # que melhor se aplica a um componente interno usado pela aplicação é realizada através da aplicação do algoritmo de resolução descrito na Figura 5.9. Suponha, por exemplo, um possível componente interno $Channel[X <: Environment, Y <: Data]$, cujos parâmetros de tipo X e Y denotam o ambiente para o qual o canal está implementado e o tipo de dado que será transmitido, respectivamente. O símbolo $<:$ denota a relação de subtipo. Portanto, este componente pode ser substituído por qualquer implementação que possua dois parâmetros que são subtipos de *Environment* e *Data*. Um componente # qualquer poderia demandar por um componente interno $Channel[MPI, Array1D]$, portanto especializado para transmitir vetores de uma dimensão através de MPI.

Caso não exista um componente # que seja uma implementação concreta de $Channel[MPI, Array1D]$, um processo de generalização fará uma busca por implementações menos específicas de canal como por exemplo uma implementação para o componente $Channel[MPIBasic, Data]$, aplicáveis com segurança neste contexto pois $Channel[MPIBasic, Data] <: Channel[MPI, Array1D]$. Obviamente, se nenhuma implementação for encontrada para o componente interno em questão, uma exceção é lançada em tempo de execução. Os detalhes do sistema

Algorithm *resolver_componente_concreto_interno*
Input *id_concrete id_inner*
Output implementação do componente *id_innner*, interno ao componente *id_concrete*

```
1  componente_concreto ←recuperar(id_concrete)
2  componente_interno ←recuperar(id_concrete, id_inner)
3  parametros_atuais ←componente_concreto.listar_parametros_atuais
4  aplicacao ←componente_interno.retornar_aplicacao
5  parametros_internos ←aplicacao.listar_parametros
6  for each pi in parametros_internos do
7      if pi é um componente then
8          aplicacao.adicionar(pi)
9      else
10         for each pa in parametros_atuais do
11             parametro_atual ←parametros_atuais.possui(pi)
12             aplicacao.adicionar(parametro_atual)
13 implementacao ←aplicacao.retornar_implementacao
14 id_concrete_actual ←implementacao.id_concrete
15 return id_concrete_actual
```

Figura 5.9: *Pseudocódigo da resolução para encontrar a implementação de um componente abstrato, interno a um componente concreto*

de tipos que garante a segurança do processo de generalização não são introduzidos nesta dissertação.

Os componentes encontrados neste método irão então possuir entradas no banco dados relacionadas a seu componente concreto que os tem como componentes internos. O método do Apêndice A.1 irá usar estas entradas para a criação dinâmica de objetos relacionados a cada componente interno, em tempo de execução.

5.7.2 O Método de Criação Diâmica

Em Mono, é possível que objetos sejam criados dinamicamente a partir de suas bibliotecas instaladas no GAC. Dado o nome da biblioteca (*assembly*), e o seu tipo, um objeto é criado em tempo de execução usando as propriedades de reflexão (*reflection*) da linguagem C# (Apêndice A.1, linhas 11-15).

Ao descobrir as implementações para cada componente interno a um componente concreto no método da Figura 5.9, o DGAC salva entradas no banco de dados que são recuperadas pelo método `createInstanceFor`. Este método recupera as informações no banco de dados (Apêndice A.1, linhas 02-09, classes DAO) e as usa na criação de um objeto interno ao componente concreto que será executado

(linhas 13-15). Através da característica da **reflexão**, presente na linguagem C#, o nosso método cria uma instância da classe realmente associada às variáveis de tipo pertinentes ao componente concreto, garantindo a segurança de tipos. O operador `typeof` é usado para retornar o tipo correto de uma variável de tipo. A partir desta informação, por reflexão, um novo objeto é criado de forma segura (*type safe*).

Capítulo 6

Estudo de Caso: Desenvolvimento e Implantação de Componentes no HPE

Este capítulo apresenta um estudo de caso simples que visa ilustrar os passos do ciclo de vida de componentes #. A aplicação escolhida é uma generalização para o programa paralelo apresentado no Capítulo 3 com o intuito de apresentar o modelo de componentes # para um número arbitrário de processadores. Consideramos importante ressaltar que essa aplicação, embora um simples caso de programa paralelo, descreve componentes de todas as espécies propostas para o HPE, ilustrando diversos aspectos de sua abordagem para composição de programas.

O capítulo está assim organizado: a Seção 6.1 descreve os detalhes da aplicação exemplo. A Seção 6.2 apresenta os principais componentes abstratos envolvidos na elaboração da aplicação. A Seção 6.3 mostra um exemplo de uma versão concreta de um componente abstrato. A Seção 6.4, por fim, explica o processo de implantação da aplicação, e sua execução.

6.1 Descrevendo a Aplicação Exemplo

O projeto de implantação da nossa aplicação define dois componentes, configurados a partir do *Front-End*: O componente *AppExampleOwner* e *AppExample*. *AppExampleOwner* é da espécie *Aplicação*. Como explicado no Capítulo 5 um componente da espécie *Aplicação* também *é* uma computação. Logo, internamente a *AppExampleOwner*, definimos um componente interno da espécie *Computação*, o componente chamado *AppExample*.

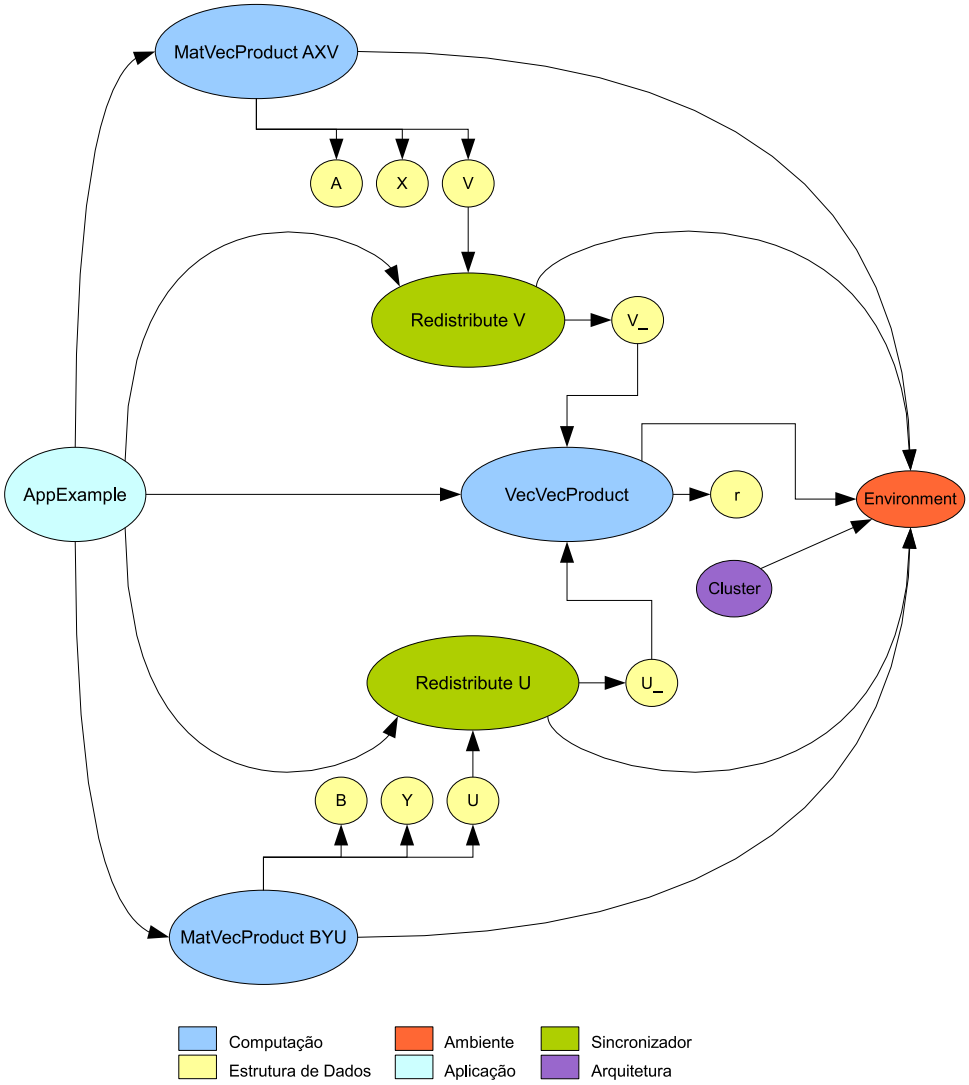

Figura 6.1: *Aplicação exemplo abstrata (AppExample), apresentando seus componentes internos. As cores discriminam a Espécie de Componente.*

AppExample define o código principal da lógica dessa aplicação enquanto *AppExampleOwner* define apenas o código que invocará a computação dando início à execução. *AppExampleOwner* foi projetado para execução em um conjunto de N processadores. Desses N processadores, um subconjunto P de processadores fica responsável em calcular o produto entre a matriz A com o vetor X. Outro subconjunto M deve calcular o produto da matriz B com o vetor Y. Portanto, $N = P + M$.

Apesar de *AppExampleOwner* ser responsável pela execução, estamos mais interessados em apresentar a lógica da computação contida no componente *AppExample*. Vamos, portanto, focalizar o processo de implantação desse último.

A Figura 6.1 dá um visão geral dos componentes envolvidos na montagem de *AppExample*, que é formado por um conjunto de componentes de diversas espécies explicitados pelas elipses. Cada elipse representa um componente #, possivelmente formado por outros componentes. Alguns componentes são básicos, não sendo formados pela composição de outros componentes.

Inicialmente, apresentaremos uma breve explicação desse exemplo nos parágrafos a seguir enfatizando o papel de cada componente interno e o significado das operações de entrada e saída. É importante ressaltar que a Figura 6.1 não condiz com o real projeto do componente *AppExample* e sim uma simplificação do mesmo, construída em uma ferramenta gráfica vetorial, com o objetivo apenas ilustrar a hierarquia de componentes. Optamos por esta abordagem pois uma tela da real representação gráfica de *AppExample* e *AppExampleOwner* causaria uma poluição visual incômoda a este texto, tanto ao leitor quanto ao nosso estilo de formatação. Entretanto, os componentes mais simples serão mostrados em sua forma real, assim como foram concebidos pelo *Front-End*.

Como podemos notar, as elipses são discriminadas por cores. Cada cor representa uma *Espécie de Componente* diferente. O componente *AppExample* possui diversos componentes internos, de outras espécies. O componente AXV, do tipo abstrato *MatVecProduct* é da espécie *Computação*. Seu objetivo é calcular a multiplicação paralela entre uma matriz e um vetor, representados pelos componentes abstratos da espécie *Estrutura de Dados* A e X, resultando em um outro vetor, representado pelo componente abstrato também desta mesma espécie V. O componente BYU é do mesmo tipo que AXV, porém aplicado aos componentes B, Y e U. Os componentes V e U são entrada para componentes da espécie *Sincronizador* de tipo abstrato *Redistribute*, cuja saída compreende as estruturas de dados $V\_$ e $U\_$, respectivamente. Esse componente distribui os vetores U e V, que estão redistribuídos em M e P processadores, nos N processadores. Então, são multiplicados paralelamente pelo componente VUr, de tipo abstrato *VecVecProduct*, atribuindo o resultado ao componente de espécie Estrutura de Dados r, que representa o escalar a ser acumulado no processador raiz.

Note-se que os componentes necessitam trabalhar sobre um ambiente de comunicação interprocessos. O componente da espécie *Ambiente* (*Environment*) está incluso nos componentes que necessitam deste tipo de serviço. Além disso, o componentes *Environment* tem como parâmetro interno o componente *Cluster*, da espécie *Arquitetura*. O componente *Cluster* indica à aplicação qual a arquitetura

paralela alvo.

A Seção seguinte apresenta os componentes **abstratos** usados na composição de *AppExample*.

6.2 Componentes Abstratos

Como mencionado no Capítulo 4, componentes abstratos possuem uma lista de parâmetros de tipos e são especificados a partir da sobreposição de componentes abstratos mais simples, ditos componentes internos. Componentes internos podem ser expostos ou não. Um componentes interno exposto c de um component c' é visível na configuração de um componente c'' que tem c' como componente interno, tornando possível o compartilhamento, uma vez que componentes que são primos próprios na hierarquia de componentes podem ser identificados pela mesma instância, desde que sejam do mesmo tipo abstrato.

Neste trabalho, e para o nosso exemplo, foi definido um conjunto de componentes básicos para a montagem dos componentes que serão usados no nosso projeto. Esses foram projetados pela ferramenta gráfica *Front-End* e cada um deles pertence a uma das **Espécies de Componentes** propostas na Seção 5.3. Nesta seção, usaremos os termos componente e componente abstrato indistintamente.

6.2.1 *PartitionStrategy*

Partition Strategy (Fingura 6.2), ou estratégia de particionamento, é o componente responsável por caracterizar a maneira pelo qual uma estrutura de dados é particionada, ou até mesmo replicada, entre as unidades de um componente onde é usado.

Indiretamente, mostra como essa estrutura de dados é distribuída entre os nós de um *cluster*. Este componente pertence à espécie *Qualificador*.

6.2.2 *Node*

Um componente *Node* (Figura 6.3) representa um determinado nó de um *cluster*. Através da interface *INode*, suas unidades oferecem serviços para recuperação de informações estáticas sobre o nó relativas à memória, à capacidade de processamento, à sua arquitetura, à pulsos de *clock*, ou mesmo dinâmicas, como sua atual carga de trabalho. Pertence à espécie *Arquitetura*.

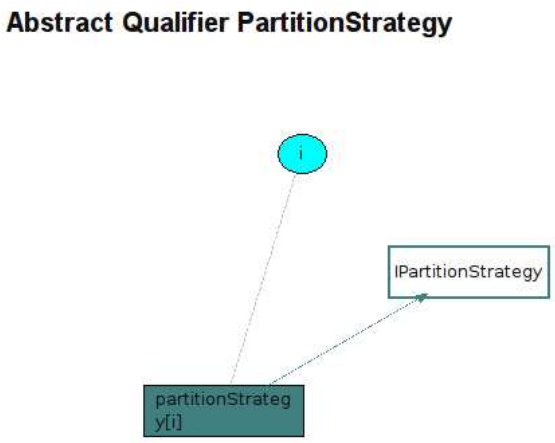

Figura 6.2: *Componente Abstrato PartitionStrategy*

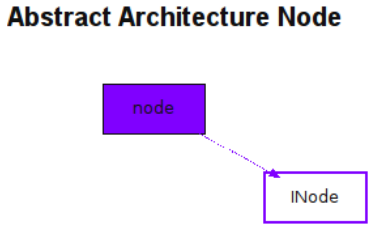

Figura 6.3: *Componente Abstrato Node*

6.2.3 *Cluster*[N<:*Node*]

Um componente *Cluster*, da espécie *Arquitetura*, representa o *cluster* onde um componente será implantado ou será posto a executar. Como podemos ver na Figura 6.4, este componente é parametrizado pelo tipo do componente interno *node*, de tipo *Node*, na variável N, de forma que implementações de *Cluster* podem ser especializadas para um certo tipo de nó. A implementação da interface *IClusterNode* a qual herda a interface *INode*, obriga as unidades ***clusterNode***[i], i=1...N, a definir um conjunto de serviços voltados a angariar informações sobre o *cluster*, como número de processadores disponíveis, carga de trabalho etc.

6.2.4 *Environment*[C<:*Cluster*[N<:*Node*]]

O componente *Environment* (Figura 6.5), da espécie *Ambiente*, é parametrizado pelo componente interno *cluster*, de tipo *Cluster*, na variável C. É sobre a arquitetura apresentada pelo componente interno *cluster* que o ambiente está instalado e irá prover a API de comunicação inter-processos. A interface *IEnvironment* oferece às unidades ***environment***[i], i=1...N, os serviços necessários

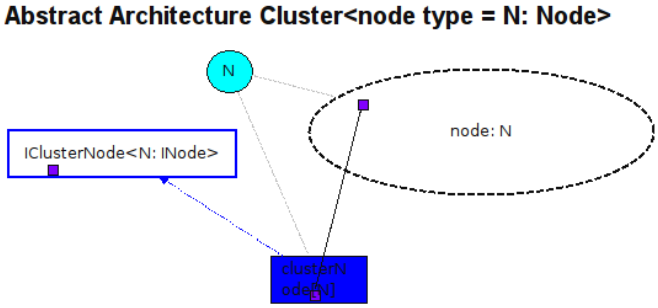

Figura 6.4: *Componente Abstrato Cluster*

à habilitação do paralelismo para os componentes que o utilizam. Observe-se que, uma vez que cada unidade do componente interno ***cluster*** é fatia de uma unidade ***environment*** , *IEnvironment* possuirá uma propriedade de tipo *IClusterNode*, que poderá ser útil na implementação do ambiente para acessar serviços que retornem informação sobre o estado dos nós individualmente e informações sobre a arquitetura.

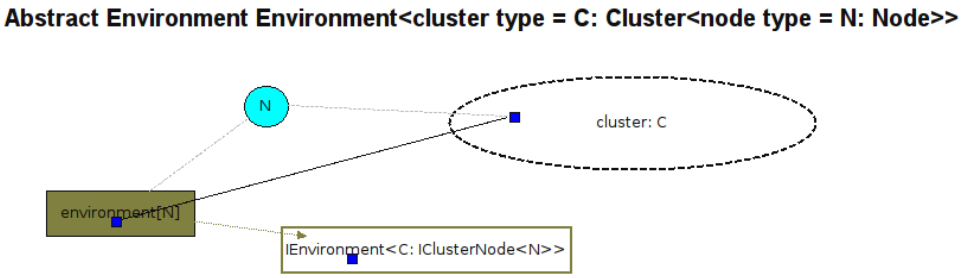

Figura 6.5: *Componente Abstrato Environment*

6.2.5 *MPI*[*C*<:*Cluster*[*N*<:*Node*]]

O componente *MPI*, da espécie *Ambiente*, é subtipo do componente abstrato anterior, *Environment*. Esse componente define um conjunto de serviços para a biblioteca MPI concreta que o implementa. Sendo assim, as primitivas de comunicação da biblioteca MPI são acessadas via chamada dos serviços destas interfaces por outros componentes. Usaremos a interface MPI.NET [28] para habilitação do MPI sobre a plataforma Mono, a qual define uma classe específica para instanciação de um ambiente MPI, perfeitamente ajustada as nossas necessidades.

6.2.6 *Data*

O componente *Data* (Figura 6.6), da espécie *Estrutura de Dados*, é o tipo base para definição de estruturas de dados seqüenciais para escalares, vetores, matrizes, etc. Sua única unidade ***data*** é representada pela interface *IData*, a qual oferece

serviços essenciais para retornar o tipo mais específico da estrutura de dados e para serialização, necessária a comunicação. No HPE, não é definida uma implementação concreta (componente #) para o componente abstrato *Data*. No HPE, são definidos os subtipos *Number* e *Double* para *Data*, representando respectivamente números de qualquer tipo e mais especificamente números de ponto flutuante de precisão dupla ($Double <: Number <: Data$).

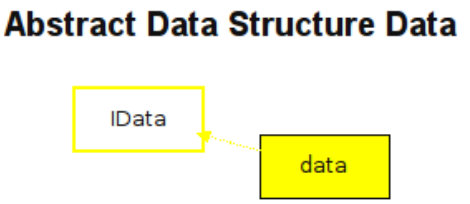

Figura 6.6: *Componente Abstrato Data*

6.2.7 *Array1D*[$E<:$*Data*] e *Array2D*[$E<:$*Data*]

Estes componentes representam *arrays* homogêneos de uma e duas dimensões, respectivamente, onde a variável E, do tipo *Data*, abstrai o tipo de dado dos elementos do *array*. Pertence à espécie *Estrutura de Dados*. A Figura 6.7 mostra apenas o componente *array* para uma única dimensão. O componente para duas ou mais dimensões deve ser compreendido por analogia. Observe-se que *Array1D* e *Array2D* são subtipos de *Data*. *Arrays* de uma e duas dimensões podem ser facilmente implementados em *buffers* contíguos de memória, permitindo que sejam facilmente serializados para comunicação. Isso é importante para implementações paralelas de aplicações de computação científica, onde *arrays* de múltiplas dimensões são freqüentemente usados e transmitidos.

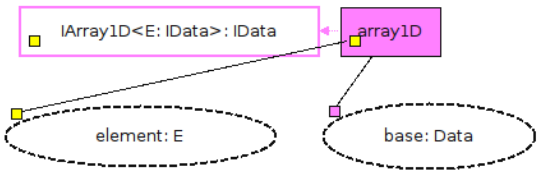

Figura 6.7: *Componente Array1D*

6.2.8 *ParallelData* [C<:*Cluster*[N<:*Node*]], E<:*Environment*[C], D<:*Data*, S<:*PartitionStrategy*]

O componente *ParallelData*, da espécie *Estrutura de Dados*, estende a noção de componente *Data* para um tipo de dado naturalmente paralelo. C, E, S, e D são parâmetros do componente *ParallelData*. Portanto, os componentes # que o implementam podem estar especializados para um certo ambiente E, sobre um certo tipo de *cluster* C, e para uma certa estrutura de dados de tipo D segundo uma determinada estratégia de particionamento S.

Como definido no projeto da Figura 6.8, esse componente é formado por $2 + N$ componentes internos, que definem: uma estratégia de particionamento p, de tipo S; um ambiente e, de tipo E, e N valores de alguma estrutura de dados d, de tipo D. Cada valor da estrutura de dados d está localizado em uma das unidades de um componente que faz uso de *ParallelData*. O conjunto deles forma uma estrutura de dados distribuída segundo a estratégia de particionamento p usando o ambiente e sobre o *cluster* c. É importante ressaltar que o componente interno e encontra-se exposto, de forma que é visível em uma configuração que usa *ParallelData*. Isso permitirá a fusão de ambientes entre componentes internos, de forma que todos estejam executando sobre a mesma instância de um ambiente.

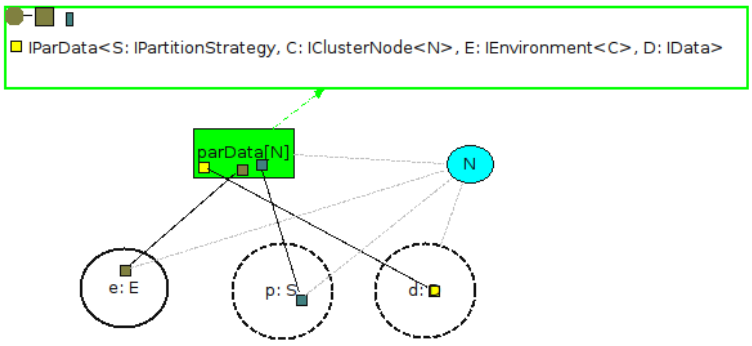

Figura 6.8: *Componente Abstrato ParallelData*

6.2.9 MatVecProduct [C<:*Cluster*[N<:*Node*]], E<:*Environment*[C], N<:*Number*, S_1<:*PartitionStrategy*, S_2<:*PartitionStrategy*]

O componente *MatVecProduct*, da espécie *Computação*, é responsável pelo cálculo da multiplicação de uma matriz A por um vetor X, armazenando o resultado em um outro vetor V. O enumerador N define a quantidade de processos nos quais *MatVecProduct* será distribuído. *MatVecProduct* tem como parâmetros de tipo: C,

representando o cluster para o qual é especializado; E, representando o ambiente para o qual é especializado; S_1, representando a estratégia de particionamento para a matriz de entrada; S_2, representando a estratégia de particionamento para o vetor de entrada e para o vetor de saída; e N, representando o tipo de número para o qual é especializado.

As estruturas de dados X e V são representadas por componentes internos dos tipos $ParallelData[C,E,Array1D[N],S_2]$, enquanto A é representado por um componente interno de tipo $ParallelData[C,E,Array2D[N],S_1]$. Portanto, tratam-se de vetores distribuídos de uma e duas dimensões segundo estratégias de particionamento S_1 e S_2, respectivamente.

Observe ainda que o componente interno e, exposto pelos componentes internos A, X e V, são fundidos, de forma que usem a mesma instância do ambiente. Além disso, os componentes internos A, X e V, assim como e, são expostos para configurações externas.

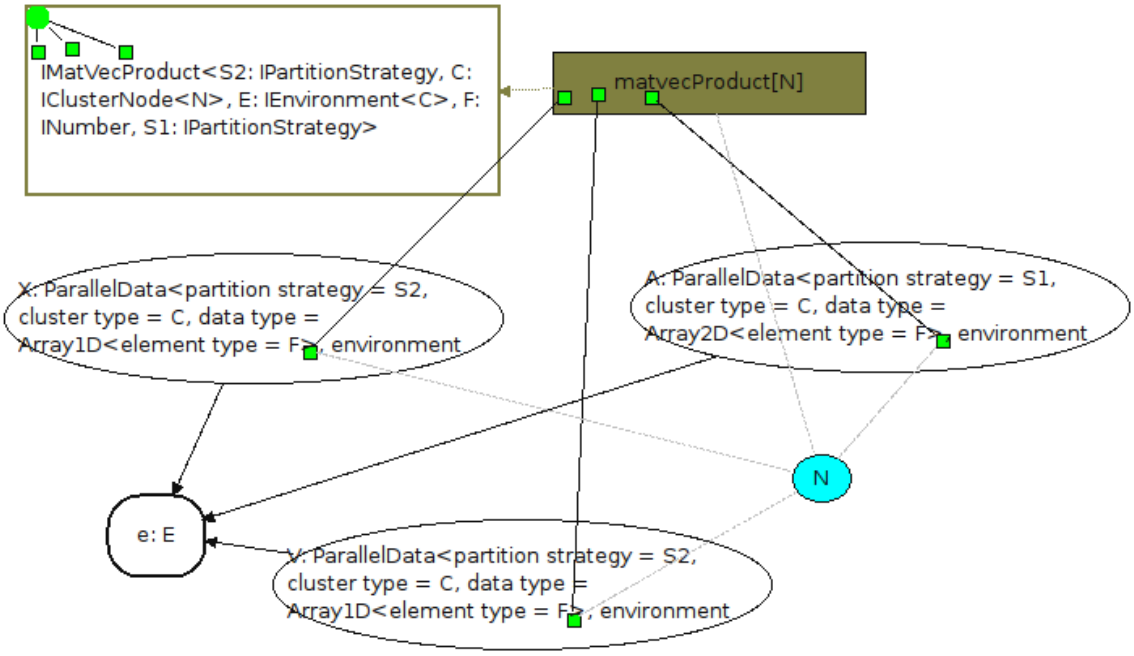

Figura 6.9: *Componente para multiplicação paralela entre um Vetor e uma Matriz*

6.2.10 *RedistributeData* [C<:***Cluster***[N<:***Node***]], E<:***Environment***[C], D<:***Data***, S<:***PartitionStrategy***]

O componente *RedistributeData*, da espécie sincronizador, tem por objetivo redistribuir dados distribuídos em um subconjunto de M processadores no conjunto

total de N processadores ($M < N$).

Em seu projeto no *Front-End*, *RedistributeData* possui os seguintes parâmetros de tipo, já comuns há outros componentes já descritos: um cluster $C <:$ *Cluster*, um ambiente $E <:$ *Environment*[C], uma estratégia de particionamento $S <:$ *PartitionStrategy* que será aplicada aos componentes internos A e B, e o tipo base da estrutura de dados paralela $N <:$ *Data*. Na Figura 6.10, o componente A, de tipo *ParallelData*[C,E,D,S], representa o dado de entrada, distribuído em M processadores, enquanto o componente B, de mesmo tipo, representa o dado de saída que deve estar distribuído nos N processadores. A e B são expostos.

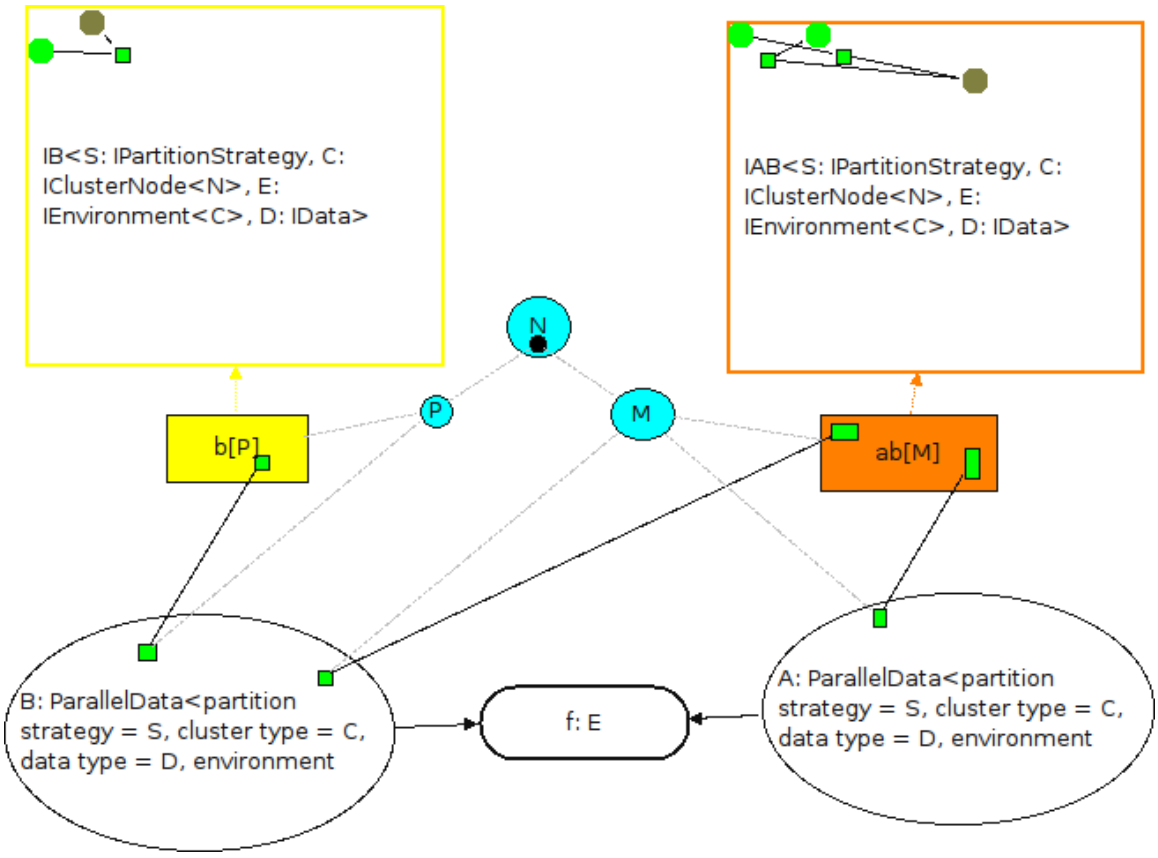

Figura 6.10: *Componente para redistribuição de dados nós de uma arquitetura*

6.2.11 *VecVecProduct* [$C<:$***Cluster***[$N<:$***Node***]], $E<:$***Environment***[C], $N<:$***Number***, $S_2<:$***PartitionStrategy***, $S_3<:$***PartitionStrategy***]

A configuração de *VecVecProduct* é muito semelhante a configuração do componente *MatVecProduct*. Seus parâmetros de tipos são semelhantes aqueles definidos para *MatVecProduct*, porém suas estratégias de particionamento são aplicadas a vetores de entradas (S_2) e a um escalar resultante (S_3). Tem como

objetivo realizar o produto interno de dois vetores U e V, representados por componentes internos de tipo $ParallelData[C,E,Array1D[N],S_2]$.

O resultado é armazenado em r, escalar representado também por um componente interno de tipo $ParallelData[C,E,N,S_3]$.

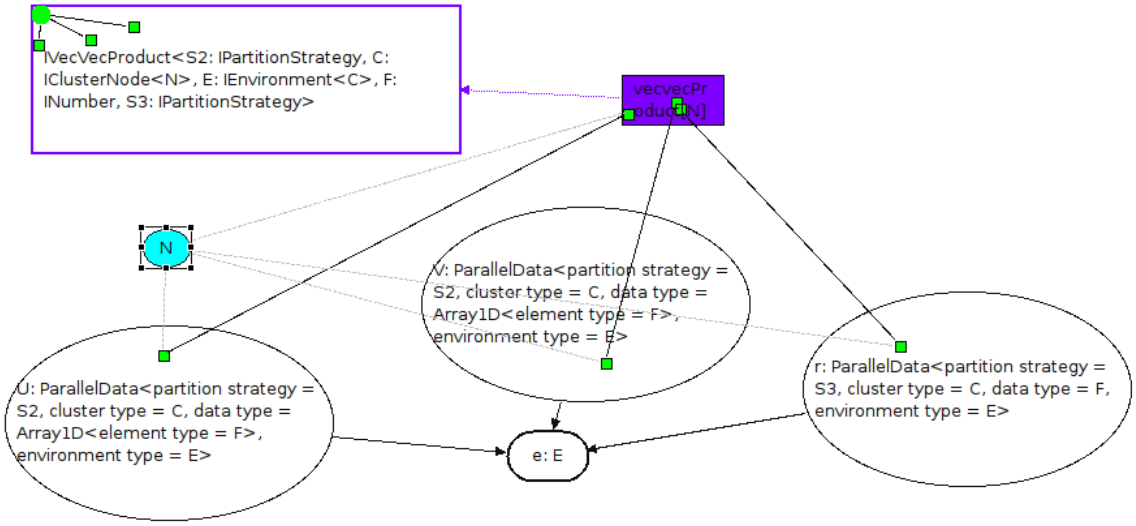

Figura 6.11: *Componente para multiplicação paralela entre vetores*

6.2.12 *AppExample* [C<:*Cluster*[N<:*Node*]], E<:*Environment*[C], N<:*Number*, S_1<:*PartitionStrategy*, S_2<:*PartitionStrategy*, S_3<:*PartitionStrategy*]

Iremos agora voltar a nos concentrar no componente *AppExample*, da Figura 6.1, da espécie *Computação*, interno ao componente *AppExampleOwner*, da espécie *Aplicação*. Seus parâmetros são C <: *Cluster*, E <: *Environement*[C], S_1, S_2, S_3 <: *PartitionStrategy* e F <: *Number*, abstraindo respectivamente o tipo do *cluster*, o ambiente para o qual suas implementações são construídas, as estratégias de particionamento adotadas por suas implementações para matrizes, vetores e o escalar resultante, respectivamente, e o tipo de número básico usado nas suas implementações. Esses parâmetros são trivialmente repassados aos parâmetros correspondentes de seus componentes internos *AXV*, *BYU*, *RedistributeV*, e *RedistributeU*. A fusão de componentes internos é usada sobre os componentes ambiente proveniente destes e entre estruturas de dados compartilhadas entre as operações.

As N unidades do componente *AppExample* são divididas em dois subconjuntos, representados pelas unidades p e q, com cardinalidades M e P, respectivamente.

Essas representam os processos do programa paralelo. Os M processos representados por p realizam o produto $A{\times}X$. Paralelamente, os P processos representados por q realizam o produto $B{\times}Y$. Em conjunto, todos os processos realizam a operação $U{\cdot}V$. Para isso os resultados das operações $A{\times}X$ e $B{\times}Y$, V e U, são redistribuídos em todos os processadores através das operações *RedistributeV* e *RedistributeU*, respectivamente. O resultado final da computação, o escalar r, encontra-se distribuído nos N processadores segundo a estratégia de particionamento S_3. Ao final do seu cálculo, cada processador sincroniza seus resultado acumulando-o em um processador principal (processador *root*), formando o resultado geral da aplicação, que é conseqüentemente retornado ao cliente.

Para facilitar a explicação iremos detalhar apenas as interfaces geradas para a unidade q de *AppExampleOwner* e *AppExample*. As interfaces geradas para a unidade p devem ser entendidas por analogia. Os arquivos fontes para estas interfaces são gerados pelo *Front-End* e armazenados no *Core*, sendo requisitados pelo *Back-End* no momento da implantação do componente concreto.

Para a unidade q do componente *AppExampleOwner* será gerado no *Front-End* a interface `IQOwner`. Analogamente, para *AppExample* será gerada a interface `IQ` (Apêndice A.2).

IQ e *IQOwner* são parametrizados pelos tipos explicados anteriormente, usando a notação *generics* do Mono. No corpo da interface `IQ`, são definidas funções de atribuição (*set*) para os parâmetros que representam as estruturas de dados a serem computadas (B, Y e R), além do ambiente que deverá ser usado para comunicação inter-processos (E). Essas estão associadas a fatias da unidade `q` que provém das unidades de componentes internos expostos. Esses componentes encontram-se privados em *AppExampleOwner*.

Tanto `IQ`, quanto `IQOwner` possuem o método *compute*, já que herdam da interface `ComputationKind` numa relação **é-um** (Apêndice A.2, linha 01). A implementação da interface `IQOwner` deve criar e atribuir a seu objeto interno do tipo `IQ`, as fatias internas `B`, `Y`, `R` e `Environment` e invocar seu método `compute`, como veremos adiante.

Resumindo, as unidades nomeadas q em *AppExample* e *AppxampleOwner* irão gerar duas interfaces: `IQ` e `IQOwner`, contidas nos arquivos fonte `IQ.cs` e `IQOwner.cs`, respectivamente. A criação dessas duas interface em separado foi um decisão de projeto na criação desta aplicação no *Front-End*. Com esta abordagem, visamos diminuir o acoplamento entre os componentes dando a liberdade de criar

implementações diferentes para cada interface. Na próxima seção iremos apresentar as implementações destas interfaces para um componente concreto que implementa o componente *AppExample* e o componente *AppExampleOwner*.

6.3 Componentes Concretos

Os Componentes abstratos apresentados na seção anterior devem possuir implementações concretas, ou componentes #, instaladas no *Back-End*. Estes componentes # são instanciados a partir de seus componentes abstratos suprindo-se sua lista de parâmetros de tipos. Entretanto, nesta seção, não iremos apresentar todos os componentes concretos. Iremos apresentar apenas a versão concreta do componente abstrato *AppExample* $[C{<}{:}Cluster[N{<}{:}Node]]$, $E{<}{:}Environment[C]$, $N{<}{:}Number$, $S_1{<}{:}PartitionStrategy$, $S_2{<}{:}PartitionStrategy$, $S_3{<}{:}PartitionStrategy]$, já que a mesma engloba internamente outros componentes concretos. No entanto, iremos mostrar a implementação da interface `IQOwner`, referente ao componente *AppExampleOwner*.

O componente # implementa o componente abstrato *AppExample* para os parâmetros atuais $C = Cluster[Node]$, $E = MPI[Cluster[Node]]$, $F = Double$, $S_1 = BalancedRows$, $S_2 = EqualPartition$, e $S_3 = PlaceInRoot]$. Ou seja, esta implementação é especializada para *clusters* genéricos baseados em MPI, distribuindo as matrizes de entrada homogeneamente por linhas entre os processadores, os vetores homogeneamente entre os processadores, armazenando o escalar resultante em um único processo, dito raiz, operando sobre números de ponto flutuante de precisão dupla. Estes parâmetros alimentam recursivamente os parâmetros de tipos dos componentes internos de *AppExample*. Para que este componente # possa executar, é necessário que estejam implantados no *Back-End* componentes # para os tipos atuais de seus componentes internos.

É importante ressaltar também que o exemplo apresentado também define um componente concreto para encapsular as funcionalidades da biblioteca de passagem de mensagens entre processos MPI. O *MPIImpl* é um componente concreto que implementa o componente abstrato *MPI*, baseado na implementação do padrão MPI para o ambiente .NET da *Microsoft*, chamada MPI.NET [28]. Este componente é exposto aos outros componentes da aplicação, sendo usado como ambiente padrão para a troca de dados entre as unidades instaladas em diversos processadores.

Através do *Front-End*, o programador gera os seguintes códigos fontes: `IQImpl.cs` (Apêndice A.5) para a unidade q do componente *AppExample*;

`IQOwnerImpl.cs` (Apêndice A.4) e `IQOwnerMainImpl.cs` (Apêndice A.3) para a unidade *q* do componente *AppExampleOwner*. Cada um destes fontes está relacionado com uma classe. As classes são implementações de suas respectivas interfaces, indicadas pelo nome do arquivo, apresentadas na seção anterior. Por exemplo, a classe `IQimpl`, contida no arquivo *IQImpl.cs* é uma implementação da interface `IQ`, contida no arquivo *IQ.cs*. O mesmo raciocínio deve ser aplicado aos outros arquivos. A classe `IQOwnerMainImpl`, por ter apenas um método (`Main`), não necessita de uma interface. Iremos explicar suas funções mais adiante.

A classe `IQImpl` implementa a interface `IQ`, sobrescrevendo seus métodos. A classe `IQOwnerImpl` implementa a interface `IQOwner`. Como explicado, na classe `IQOwnerImpl` haverá um objeto do tipo `IQ` (Apêndice A.4, linha 03). Finalmente, a classe `IQOwnerMainImpl` possui um objeto interno do tipo `IQOwner` (Apêndice A.3, linha 03).

Veremos na próxima seção como o DGAC implanta e executa estes arquivos.

6.4 Implantação do Exemplo

Componentes devem estar previamente implantados no *Back-End* para serem usados por alguma aplicação. Os componentes resultam em bibliotecas implementadas em Mono. Estas bibliotecas são referenciadas pelo componente da espécie aplicação que o cliente montou no *Front-End*. No nosso caso, a aplicação em questão é o componente *AppExample*, do apêndice 6.1.

Após projetar a aplicação *AppExampleOwner* junto com seu componente interno *AppExample* no *Front-End*, o usuário gera o código fonte referente as unidades executáveis em cada processo (apresentados nas Seções 6.2 e 6.3). Esse código fonte pode ser modificado pelo cliente, de acordo com a necessidade e é então publicado no *Core*. No momento da implantação, o *Back-End* requisita os fontes e os compila, gerando código executável que fará a computação.

Nesta seção iremos detalhar o processo de implantação desta aplicação, separando-o em duas partes: a compilação do componente # definido para *AppExample* e a execução deste componente. É importante ressaltar que a compilação gera um arquivo executável cujas referências compreendem apenas em **interfaces**, ou seja, unidade de componentes abstratos. No momento da execução é que estas interfaces serão instanciadas por **classes concretas**, ou seja, classes associadas às unidades de componentes # que implementam componentes abstratos. A descoberta dessas classes concretas é feita dinamicamente por um método do

DGAC chamado `createInstanceFor`, mostrado no Capítulo 5.

O objetivo desta implementação é apresentar um teste de conceito das idéias apresentadas na dissertação. Não visamos, ainda, testes comparativos ou otimizações do código paralelo.

6.4.1 Compilando o Componente # Aplicação

Para analisar o processo de compilação e geração do executável, descreveremos uma abordagem *bottom-up*, inciando da fatia mais interna (`IQ`) e finalizando em (`IQOwner`).

Analisando a declaração de variáveis em `IQImpl` (Apêndice A.5, linhas 03-09), cada objeto criado é declarado por uma interface (`calculateBYU` é um objeto da interface `IMatVecProduct`, por exemplo). Essas interfaces são representadas por bibliotecas Mono (arquivos dll). Para facilitar a explicação, nossa dissertação considera que as bibliotecas já estão pré-instaladas sendo necessária apenas a sua referência para o processo de compilação da classe de `IQImpl`.

A seguinte lista de bibliotecas será usada para a compilação de `IQImpl`: `IMatVecProduct.dll`, para a fatia `calculateBYU`; `IA.dll` e `IAB.dll`, para as fatias `redistributeU` e `redistributeV`; `IVecVecProduct.dll` para a fatia `calculateUVr`; a biblioteca `IParallelData.dll` para os vetores das fatias `u`, `v`, `u_`, `y` e `b` e finalmente `IEnvironment.dll` para o ambiente `environment`. Todas bibliotecas referentes à interfaces já compiladas.

As bibliotecas encontradas são referências apenas às fatias internas de `IQImpl.cs`. Como a classe `IQImpl` **implementa** a interface `IQ`, cujo fonte é mostrado na Apêndice A.2, a compilação de `IQImpl` deve referenciar a bibliotecas gerada por esse fonte (`IQ.dll`), formando assim a lista de referências. Supomos assim, que sua interface já está instalada.

De posse dessa lista de referências, o DGAC a envia a cada processo-filho (descritos no Capítulo anterior) que gera a seguinte *string* de compilação:

```
mcs IQImpl.cs -out:IQImpl.dll
    -r:IQ.dll -r:IMatVecProduct.dll -r:IRedAB.dll -r:IRedA.dll
    -r:IVecVecProduct -r:IParData.dll -r:IEnvironment.dll -r:ICluster
```

Essa *string* diz que `IQImpl` será compilado usando as bibliotecas precedidas por `-r:` e gerará uma biblioteca chamada `IQImpl.dll` através do argumento `-out:`. A saída é uma biblioteca pois `IQImpl` é da espécie *Computação*. Feito isso, cada

processo-filho instala a nova biblioteca no GAC de sua responsabilidade, tornando-a visível a outras aplicações. No entanto, pretendemos executar a unidade e bibliotecas só podem ser executadas se referenciadas por arquivos executáveis. Precisamos agora, criar a unidade executável que fará uso dessa nova biblioteca.

As classes `IQOwnerImpl` e `IQOwnerMainImpl`, explicadas anteriormente devem ser compiladas em um único arquivo executável. Para isso, uma nova lista de referência deve ser criada e repassada aos processos-filhos existentes no DGAC. O DGAC utiliza o mesmo processo usado na compilação de `IQImpl` para compilar `IQOwnerMainImpl`. Tal classe, por ser da espécie *Aplicação* deverá gerar um arquivo `exe`. A lista de referências para sua compilação compreende em seu código fonte, no código fonte de `IQOwnerImpl`, a biblioteca de sua interface `IQOwner` além de `IQ.dll` que é interface biblioteca anteriormente criada, `IQImpl.dll` e outras bibliotecas, ficando portanto:

```
mcs IQOwnerMainImpl.cs IQOwnerImpl.cs -out:IQOwnerMainImpl.exe
    -r:IQ.dll -r:IQOwner.dll -r:ICastanhaoNode.dll -r:IMPIBasic.dll
    -r:IDouble.dll -r:IEqualPartitionByRows.dll
    -r:IEqualPartition.dll -r:IReduceSum.dll
```

Cada processo-filho localizado em um processador criará esse arquivo executável usando as nossas classes de interface para comunicação com o *GacUtil* e *shell* do Linux. Após gerado o executável, o mesmo é salvo em uma pasta específica e iniciado pelo comando `mono` o qual chamará o seu método `Main`, desta forma:

```
mpiexec -np 8 mono IQOwnerMainImpl.exe.
```

O comando acima funciona pois todas as bibliotecas necessárias a este executável foram instaladas no DGAC e portanto são visíveis por outras aplicações. O processo análogo ocorre para a unidade *p*. A diferença são os dados de entrada e saída. Os detalhes da execução serão apresentados na próxima subseção.

6.4.2 Executando o Componente # da Aplicação

Nesta subseção apresentaremos em mais detalhes o processo de execução do arquivo `IQOwnerMainImpl.exe` criado anteriormente. É usada uma abordagem *top-down* explicando inicialmente o que ocorre na execução da classe `IQOwnerMainImpl` a qual chama em cadeia as classes `IQOwnerImpl` e `IQImpl`.

Ao executar o método *Main*, a classe `IQOwnerMainImpl` cria um novo objeto do tipo `IQOwnerImpl` (`iqowner`) da forma trivial, usando o operador `new`, chamando o seu construtor. O objeto `iqowner` é então fatia da aplicação `IQMainImpl`. Logo após, é feita a chamada `iqowner.compute()` (Apêndice A.3, linha 04).

A chamada ao contrutor de `IQOwnerImpl` (Apêndice A.4, linhas 32-38) cria dinamicamente os objetos `iqimpl`, `B,Y,r` e `e`, fazendo uso do método `createInstaceFor` do DGAC. Esse método, carrega dinamicamente uma biblioteca Mono referente a um componente concreto e o atribui a sua respectiva fatia. Portanto `createInstanceFor` irá carregar e atribuir à fatia `b`, o componente concreto `ParallelArray2DImplMPI`, implementação de `ParallelData` objetivando matrizes. Para `y`, o componente concreto `ParallelArray1DImplMPI`, implementação de `ParallelData` objetivando vetores. Para `e`, o componente concreto `MPI`, implementação de `Environment` objetivando o ambiente MPI. Finalmente, para `iqimpl`, o componente concreto `IQImpl`, implementação de `IQ`, previamente instalado na seção anterior. O método `createInstanceFor`, em sua primeira chamada, calcula o componente concreto associado ao objeto esalva sua referência no DGAC. Nas demais chamadas, `createInstanceFor` apenas consulta o que foi calculado na primeira chamada.

É muito importante notar que no momento da instanciação, no método compute, cada novo objeto é atribuido a sua respectiva fatia fazendo uso do métodos `set` (Apêndice A.4, linhas 07-32). Estes métodos são simples, mas não triviais. Cada `set`, além de atribuir o valor corrente calculado no método `compute` à fatia correspondente, atribui à fatia `iqimpl` também este valor. Por exemplo, vemos que no Apêndice A.4, linhas 15-16, o valor (`value`) corrente é atribuído à fatia `e` e posteriormente atribuído a fatia `e` pertencente a `iqimpl` (linha 16). Isso quer dizer que esta fatia é compartilhada entre estes objetos os quais acessam a mesma instância carregada pelo método `createInstanceFor`. O mesmo processo ocorre às fatias `b,y` e `r`, em seus respectivos métodos `set`.

A chamada `iqowner.compute()`, refere-se ao método `compute` da classe concreta `IQOwnerImpl` (Apêndice A.4, linhas 40-44). Nesse método as fatias `b` e `y` tem seus valores inicializados, o método `compute` de `iqimpl` é chamado e o valor de `r` é calculado em *IQImpl* é mostrado. Vejamos o que ocorre em *IQImpl* no próximo parágrafo.

Antes de chamar `compute` em `iqimpl`, `IQOwnerImpl` o tinha criado através do carregamento dinâmico de bibliotecas. Nesse processo de criação, o contrutor

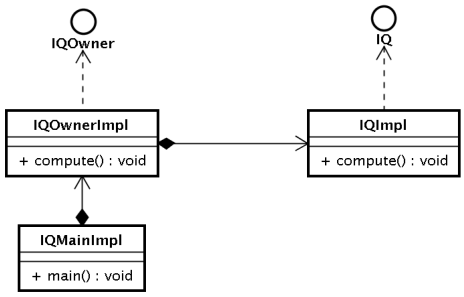

Figura 6.12: *Diagrama de classes referentes a unidade q*

de `IQImpl` foi chamado. Vejamos agora sua lógica que coompreende no código do Apêndice A.5, linhas 58-67. Nesse trecho, instâncias para as implementações de suas fatias privadas são criadas por meio do método do DGAC *createInstanceFor*. As fatias privadas coompreendem as implementações dos componentes que formam *AppExample*. Portanto, temos o objeto `this.calculateBYU`, o objeto `this.redistributeU`, o objeto `this.redistributeV` e o objeto `this.calculateUVr`. Além desses que representam computações, temos como fatias privadas os vetores `V_`, `U`, `U_`.

Assim como em `IQOwnerImpl`, o método `createInstanceFor` irá retornar componentes concretos e atribuí-los às fatias existentes em `IQImpl`. Portanto, para a fatia `calculateBYU` será atribuido o componente # `MatVecProductMPIImpl`. Para `redistributeU` e `redistributeV` será atribuido o componente # `RedistributeArray1DMPIImpl`. Para `calculateUVr` será atribuido o componente # `VecVecProductMPIImpl`. Para `V_`, `U`, `U_` será atribuido o componente # `ParallelArray1DMPIImpl`, representado o tipo vetor. As fatias ditas expostas são representadas pelos objetos `B` (Matriz); o objeto `Y`; o escalar `r` e o ambiente `E` que, como explicado, já foram criadas externamente por `IQOwnerImpl`.

Os métodos `set` atribuem seus valores correntes calculados no construtor as suas fatias internas. Por exemplo, no Apêndice A.5, linhas 16-24, a fatia `e`, criada exteriormente em `IQOwner` é atribuída as fatias `redistribute` e `calculate`. Sendo assim, existe uma inicialização em cadeia da fatia `e` que inicia em `IQOwner`, é atribuída a `IQImpl` e finalmente repassada as suas fatias internas. Podemos concluir que todos os componentes da aplicação `AppExample` compartilham o mesmo ambiente de comunicação inter-processos, o MPI, como previsto pela configuração.

Na implementação do método `compute` (Apêndice A.5, linhas 68-78), vemos a lógica da aplicação sendo iniciada. Primeiramente é chamado o método

`compute` do objeto `calculateBYU`, realizando assim a multiplicação da matriz B por Y e armazenando o resultado em U. Logo após, os vetores U e V são concorrentemente redistribuídos entre os processadores da aplicação através da chamada do método `synchronize` de seus respectivos sincronizadores (objetos `redistributeV` e `redistributeU`). Finalmente, o objeto `calculateUVr` computa o produto interno dos vetores distribuídos U e V, acumulando o resultado em r.

Capítulo 7

Considerações Finais

Em decorrência do crescimento em escala e complexidade de aplicações de Computação de Alto Desempenho, tanto no meio acadêmico quanto comercial, um grande esforço por parte dos pesquisadores para tornar a atividade de programação a mais produtiva possível e ao mesmo tempo atingir bom desempenho, está sendo realizado. Para isso, faz-se necessária a criação (e adaptação) de vários modelos que devem fazer uso de idéias provindas de *Engenharia de Software* específicas de nichos como *Sistemas Distribuídos*, por exemplo.

Esta dissertação de mestrado apresentou a o estado-da-arte das tecnologias de desenvolvimento de aplicações em computação de alto desempenho, principalmente no meio científico, e soluções baseadas em Programação Orientada a Componentes. No entanto, tais soluções ainda não garantem o poder de abstração e generalidade ao programador e/ou especialista. Em suma, modelos de componentes convencionais não são capazes de descrever padrões mais gerais de paralelismo sem recorrer a extensões ortogonais ao próprio modelo.

A contribuição do Modelo # (*Hash*), baseado em componentes, é a de tornar mais simples a criação de aplicações em computação de alto desempenho que desejem explorar o paralelismo, encapsulando a computação em componentes naturalmente paralelos e hierárquicos. A programação desses componentes é centrada em interesses tendo como conseqüencia sua divisão em processos, ou unidades paralelas. Essa abordagem centra a programação do usuário cliente no domínio do problema e não no código de paralelização. Esse último ponto, porém, ainda não foi objeto de investigação deste trabalho.

Para concretizar o Modelo #, foi proposta a Arquitetura *Hash* de *Frameworks*, a partir da qual definimos o *framework* HPE, o qual materializa os conceitos

inerentes aos componentes #. A partir deste, instanciamos o HPE, ambiente voltado ao desenvolvimento, implantação e execução de aplicações de computação de alto desempenho sobre clusters de multiprocessadores. A contribuição central deste trabalho reside na apresentação do projeto e implementação do *Back-End* do HPE sobre a plataforma CLI. Ao *Back-End*, foi submetida uma configuração de um programa paralelo baseado em componentes, o qual realiza uma operação simples de álgebra linear em paralelo. Esta aplicação foi construída a partir de um conjunto de componentes # básicos e compostos especialmente propostos para ela. Tais componentes foram construídos como configurações montadas no *Front-End*, buscando exemplificar alguns conceitos fundamentais deste estilo de programação. Além disso, a implantação e execução dessa aplicação serviu como teste de conceito para a implementação do *Back-End* desta dissertação.

7.1 Trabalhos Futuros

Trabalhos futuros sobre esta dissertação, e sobre o HPE como um todo, incluem:

- Implantação de *BackEnds* em *clusters* separados comunicando-se via componentes CCA. Desta forma, uma aplicação baseada em componentes # estaria instalada em dois *clusters*, ao mesmo tempo. Este estudo inclui a integração com o *framework* CCA Forró.

  O grupo pretende usar os *clusters* disponíveis ao grupo, como, por exemplo, os do CENAPAD [24] e o do LIA [43], chamado Castanhão, integrando-os via componentes de comunicação. Uma aplicação trocaria dados entre os *clusters* e, localmente em cada um deles, rodaria computações paralelas. Teríamos então uma aplicação naturalmente paralela (localmente no *cluster*) e distribuída (em *clusters* diferentes).

- Uso de ontologias para definição de espécies de componentes adequadas a nichos específicos de aplicações, os quais capturam o conhecimento de especialistas de uma determinada área, ao modo de *ambientes de solução de problemas* [31, 62, 76] e linguagens de domínio específico. Pretendemos criar regras de inferências que interpretem ontologias dando assim uma certa "inteligência" à aplicação. Regras de inferência propiciariam poder de decisão sobre a forma de distribuição das tarefas entre os processadores, dinamicamente. Processadores ociosos, ou com menos carga de trabalho,

seriam escolhidos prioritariamente, por exemplo. Espécies *Qualificador* são ótimos candidatos para este tipo de abordagem.

- Metodologias formais para especificação e derivação do código paralelo, fazendo uso do *Circus* [47] e redes de Petri [52, 58]. *Circus* é uma linguagem que define especificações de sistemas concorrentes. O seu objetivo é dar uma base sólida ao desenvolvimento de sistemas concorrentes e distribuídos, permitindo a análise de propriedades comportamentais e funcionais destes. Redes de Petri oferece um grande variedade de ferramentas para análise de propriedades estruturais, comportamentais e de desempenho, sendo seu uso alvo de investigação do grupo desde os trabalhos antecedentes com Haskell# [22].

- Criação da bibliotecas de componentes básicos, de propósito geral ou de domínio específico, para HPE. Pretendemos disponibilizar bibliotecas de componentes básicos em *clusters* de acesso a pesquisadores. Essas bibliotecas de componentes básicos serviriam de base para a montagem de componentes mais complexos e possivelmente aplicações que os usem. Além de componentes básicos para programação paralela em geral, podemos vislumbrar componentes básicos para as disciplinas de Álgebra Linear, Sistemas Lineares, Geometria Computacional, dentre outros nichos. Em particular, vislumbramos a construção de uma biblioteca de componentes paralelos baseado em funcionalidades desenvolvidas pelo grupo Pargo ao longo dos anos.

- Implementação de uma versão do *Front-End* compatível com um navegador *web*, podendo fazer uso da tecnologia JSF ou *applets* Java. Um *Front-End* disponível em um sítio na *Internet* poderia ser usado por diversos desenvolvedores, os quais seriam previamente cadastrados no sistema. Componentes seriam montados remotamente e suas configurações seriam salvas no mesmo sítio, para posterior acesso.

- Execução de *benchmarks* sobre aplicações matemáticas, possibilitando a avaliação de diversas configurações de componentes e o estudo de soluções de otimização do código paralelo.

Referências Bibliográficas

[1] Allan, B. A., Armstrong, R. C., Wolfe, A. P., Ray, J., Bernholdt, D. E., and Kohl, J. A. The CCA Core Specification in a Distributed Memory SPMD Framework. *Concurr. & Comput. : Pract. & Exper. 14*, 5 (2002), 323–345.

[2] Allen, R., and Garlan, D. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol. 6*, 3 (1997), 213–249.

[3] Andy Ju An Wang, K. Q. *Component-Oriented Programming*. LNCS. Wiley Inter-Science, 2005.

[4] Arbab, F. Reo: a Channel-Based Coordination Model for Component Composition. *Mathematical. Structures in Comp. Sci. 14*, 3 (2004), 329–366.

[5] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and Smolinski, B. 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, 1999), IEEE Computer Society, p. 13.

[6] Armstrong, R., Kumfert, G., McInnes, L. C., Parker, S., Allan, B., Sottile, M., Epperly, T., and Dahlgren, T. The CCA Component Model for High-Performance Scientific Computing. *Concurr. Comput. : Pract. Exper. 18*, 2 (2006), 215–229.

[7] Avery, J., and Holmes, J. *Creating .Net Applications on Linux and Mac OS X*. O'Reilly, 2007.

[8] AXIS The Axis Framework. http://ws.apache.org/axis/ - Acessado em Março/2007.

[9] BABEL Toolkit. https://computation.llnl.gov/casc/components/babel.html - Acessado em Dezembro/2007.

[10] Baduel, L., Baude, F., and Caromel, D. Asynchronous Typed Object Groups for Grid Programming. *International Journal of Parallel Programming 35*, 6 (2007), 573–614.

[11] Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., and Quilici, R. *Grid Computing: Software Environments and Tools*. Springer-Verlag, January 2006, ch. Programming, Deploying, Composing, for the Grid.

[12] Baude, F., Caromel, D., Henrio, L., and Morel, M. Collective Interfaces for Distributed Components. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 599–610.

[13] Baude, F., Caromel, D., and Morel, M. From Distributed Objects to Hierarchical Grid Components. *Lecture Notes in Computer Science 2888* (2004), 1226–1242.

[14] Bernabeu, J. M., Khalidi, Y. A., Matena, V., Shirriff, K., and Thadani, M. N. Solaris MC: A Multi-Computer OS. Tech. rep., Sun Microsystems, Inc., Mountain View, CA, USA, 1995.

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

[16] Breivold, Pei, H., and Larsson, M. Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles. *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on* (28-31 Aug. 2007), 13–20.

[17] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., and Stefani, J.-B. The FRACTAL Component Model and its Support in Java. *Softw, Pract. Exper 36*, 11-12 (2006), 1257–1284.

[18] Bruneton, E., Coupaye, T., Leclercq, M., Quewewma, V., and Stefani, J.-B. An Open Component Model and Its Support in Java. *Lecture Notes in Computer Science* (2004), 7–22.

[19] Bruneton, E., Lenglet, R., and Coupaye, T. ASM: A Code Manipulation Tool to Implement Adaptable Systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)* (November 2002).

[20] Carvalho Junior, F. H., Correa, R. C., Lins, R., Silva, J. C., and Araújo, G. A. High Level Service Connectors for Components-Based High Performance Computing. In *19th International Symposium on Computer Architecture and High Performance Computing* (Oct. 2007), pp. 237–244.

[21] Carvalho Junior, F. H., Lins, R., Corrêa, R. C., and Araújo, G. A. On the Design of Abstract Binding Connectors for High Performance Computing Component Models. In *Joint Workshop on HPC Grid Programming Environments and Components and Component and Framework Technology in High-Performance and Scientific Computing* (2007).

[22] Carvalho Junior, F. H., and Lins, R. D. Haskell$_{\#}$: Parallel Programming Made Simple and Efficient. *j-jucs 9*, 8 (aug 2003), 776–794.

[23] Carvalho Junior, F. H., and Lins, R. D. Separation of Concerns for Improving Practice of Parallel Programming. *INFORMATION, An International Journal 8*, 5 (Sept. 2005).

[24] CENAPAD. http://www.cenapadne.br - Acessado em Abril/2008.

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

[26] CORBA http://www.omg.org/technology/documents/formal/components.htm - Acessado em Dezembro/2007.

[27] Data Parallel CORBA http://www.omg.org/technology/documents/formal/data_parallel.htm - Acessado em Dezembro/2007.

[28] Douglas Gregor and Andrew Lumsdaine. Design and implementation of a high-performance MPI for C# and the common language infrastructure. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), ACM, pp. 133–142.

[29] Englander, R. *Developing Java Beans*. O'Reilly, 1997.

[30] Foster, I., and Kesselman, C. Globus: A Toolkit-Based Grid Architecture. In *The Grid: Blueprint for a Future Computing Infrastructure*. MORGAN-KAUFMANN, 1998, pp. 259–278.

[31] Gallopoulos, E., Houstis, E., and Rice, J. R. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Comput. Sci. Eng. 1*, 2 (1994), 11–23.

[32] Govindaraju, M., Krishnan, S., Chiu, K., Slominski, A., Gannon, D., and Bramley, R. 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, 2003), IEEE Computer Society, p. 182.

[33] HP HPC Solutions. http://www.winhpc.org/hphpc - Acessado em Junho/2007.

[34] Johnson, C., Parker, S., and Weinstein, D. Component-Based Problem Solving Environments for Large-Scale Scientific Computing. *Concurr. & Comput. : Pract. & Exper. 14* (2002).

[35] Junior, F. C., Araújo, G., de Carvalho Silva, J., Corrêa, R., and Lins, R. High-Level Service Connectors for Components-Based High Performance Computing. In *SBAC-PAD: Nineteenth International Symposium on Computer Architecture and High Performance Computing* (Washington, DC, USA, 2007), IEEE Computer Society, p. 10.

[36] Junior, F. C., Lins, R., Corrêa, R., Araújo, G., and Santiago, C. Design and Implementation of an Environment for Component-Based Parallel Programming. In *Proceedings of VECPAR'2006* (2006).

[37] Junior, F. C., Lins, R., and Martins. An Institutional Theory for #-Components. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF'2006)* (2006), pp. 152–157.

[38] Junior, F. C., and Lins, R. D. 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, 2005), ACM Press, pp. 1357–1364.

[39] Junior, F. H. C., and Lins, R. D. The # Model for Parallel Programming: from Processes to Components with Minimal Performance Overheads, 2005.

[40] Junior, F. H. C., Lins, R. D., Corrêa, R. C., and Araújo, G. A. Towards an Architecture for Component-Oriented Parallel Programming: Research Articles. *Concurr. & Comput. : Pract. & Exper. 19*, 5 (2007), 697–719.

[41] Keahey, K., and Gannon, D. PARDIS: A Parallel Approach to CORBA. In *HPDC '97: Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97)* (Washington, DC, USA, 1997), IEEE Computer Society, p. 31.

[42] Kohn, S., Kumfert, G., Painter, J., and Ribbens, C. Divorcing Language Dependencies from a Scientific Software Library. In *In 10th SIAM Conference on Parallel Processing* (Portsmouth, VA, 2001).

[43] LIA. http://www.lia.ufc.br - Acessado em Abril/2008.

[44] Liberty, J., and Hurwitz, D. *Programming ASP.NET*. O'Reilly, 2005.

[45] Lopes, A., Wermelinger, M., and Fiadeiro, J. L. Higher-order architectural connectors. *ACM Trans. Softw. Eng. Methodol. 12*, 1 (2003), 64–104.

[46] Malawski, M., Kurzyniec, D., and Sunderam, V. MOCCA - Towards a Distributed CCA Framework for Metacomputing. In *Parallel and Distributed ProcessingSymposium,2005. Proceedings.* (2005), 19th IEEE International, pp. 4–8.

[47] Manuela Xavier and Ana Cavalcanti and Augusto Sampaio. Type Checking Circus Specifications. *Electron. Notes Theor. Comput. Sci. 195* (2008), 75–93.

[48] Mattax, C. C., and Kyte, R. L. Reservoir Simulation, 1990.

[49] Mayes, K., Riley, G. D., Ford, R. W., Luján, M., Freeman, L., and Addison, C. The Design of a Performance Steering System for Component Based Grid Applications. In *Performance Analysis and Grid Computing* (2004), Eds. Kluwer Academic Publishers, pp. 111–127.

[50] MCMD-WG Toolkit. https://www.cca-forum.org/wiki/tiki-index.php?page=MCMD-WG - Acessado em Abril/2008.

[51] Milli, H., Elkharraz, A., and Mcheik, H. Understandig Separation of Concerns. In *In Workshop on Early Aspects - Aspect Oriented Software Development(AOSD'04)* (2004), pp. 411–418.

[52] Murata, T. Petri Nets: Properties Analysis and Applications. *Proceedings of IEEE 77*, 4 (Apr. 1989), 541–580.

[53] Pacheco, P. S. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[54] PARIS project team http://www.irisa.fr/paris/web/ - Acessado em Dezembro/2007.

[55] Parker, S. G., Johnson, C. R., and Beazley, D. Computational Steering Software Systems and Strategies. *IEEE Comput. Sci. Eng. 4*, 4 (1997), 50–59.

[56] Parlavantzas, N., Getov, V., Morel, M., Baude, F., Huet, F., and Caromel, D. Componentising a Scientific Application for the Grid. In *Achievements in European Research on Grid Systems* (London, UK, 2006), Springer-Verlag, pp. 109–122.

[57] Perez, C., Priol, T., and Ribes, A. A Parallel CORBA Component Model for Numerical Code Coupling. *International Journal of High Performance Computing Applications 17*, 4 (2003), 417–429.

[58] Petri, C. A. Kommunikation mit Automaten. *Technical Report RADC-TR-65-377, Griffiths Air Force Base, New York 1*, 1 (1966).

[59] Pierce, B. C. *Types and Programming Languages.* The MIT Press, 2002.

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

[61] Priol, T., and René, C. Cobra: A CORBA-Compliant Programming Environment for High-Performance Computing. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing* (London, UK, 1998), Springer-Verlag, pp. 1114–1122.

[62] Problem Solving Environments. http://www.cs.purdue.edu/research/cse/pses/ - Acessado em Abril/2008.

[63] Pérez, C., Priol, T., and Ribes, A. PACO++: A Parallel Object Model for High Performance Distributed Systems. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9* (Washington, DC, USA, 2004), IEEE Computer Society, p. 90274.1.

[64] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM 40*, 10 (1997), 39–42.

[65] Sarkar, V., Williams, C., and Ebcioğlu, K. Application Development Productivity Challenges for High-End Computing. In *IEEE International Conference on High Performance Computer Architecture (HPCA), Workshop on Productivity and Performance in High-End Computing* (2004), pp. 14–18.

[66] Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley and Sons, Inc., New York, NY, USA, 1998.

[67] Shaw, M. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *ICSE 93: Selected papers from the Workshop on Studies of Software Design* (London, UK, 1996), Springer-Verlag, pp. 17–32.

[68] Skjellum, A., Bangalore, P., Gray, J., and Bryant B. Reinventing Explicit Parallel Programming for Improved Engineering of High Performance Computing Software. In *International Workshop on Software Engineering for High Performance Computing System Applications* (May 2004), ACM, pp. 59–63. Edinburgh.

[69] Squyres, J. M., and Lumsdaine, A. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models*

and Systems for Grid Applications (St. Malo, France, July 2004), V. Getov and T. Kielmann, Eds., Springer, pp. 167–185.

[70] Standard ECMA-335: Common Language Infrastructure (CLI), 2006.

[71] Szyperski, C. Component Software and the Way Ahead. 1–20.

[72] Tip, F. A Survey of Program Slicing Techniques.

[73] Top 500 List of Super Computers. http://www.top500.org - Acessado em Outubro/2006.

[74] Towards 2020 Science. http://research.microsoft.com/towards2020science - Acessado em Fevereiro/2007.

[75] van der Steen, A. J. Issues in Computational Frameworks. *Concurr. & Comput. : Pract. & Exper. 18*, 2 (2006), 141–150.

[76] PSE, Virginia Tech. http://research.cs.vt.edu/pse/ - Acessado em Abril/2008.

[77] W3C The World Wide Web Consortium. http://www.w3.org - Acessado em Junho/2007.

[78] Weiser, M. Program Slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering* (Piscataway, NJ, USA, 1981), IEEE Press, pp. 439–449.

[79] Windows HPC. http://www.winhpc.org/ - Acessado em Janeiro/2007.

Apêndice A

Código Fonte

Neste apêndice, apresentamos o código fonte relacionado ao procedimento mostrado na Seção 5.7.2 e as classes explicadas no Capítulo 6.

```
01.  createInstanceFor(hash_component_uid, id_inner, id_unit) {
02.    ComponentDAO cdao = new ComponentDAO();
03.    Component c = cdao.rtrv_uid(hash_component_uid);
04.    InnerConcreteComponentDAO icdao = new InnerConcreteComponentDAO();
05.    IList<InnerConcreteComponent> icl = icdao.rtrvEnum(c.Id_concrete, id_inner);

06.    InnerConcreteComponent ic = icl.GetEnumerator().Current;
07.    int id_concrete_actual = ic.Id_concrete_actual;
08.    UnitDAO udao = new UnitDAO();
09.    IList<Unit> us = udao.rtrvEnum(id_concrete_actual, id_unit);

10.    Unit u = us.GetEnumerator().Current;

11.    string lib_path = u.Library_path;
12.    string lib_source_name = u.Library_module_name;

13.    Assembly a = Assembly.LoadFrom(lib_path);
14.    Type t = a.GetType(lib_source_name);
15.    Object o = Activator.CreateInstance(t);
16.  }
```

Figura A.1: *Recuperação de um componente em tempo de execução*

```
01.  public interface IQ<C, E, F, S1, S2, S3> :  ComputationKind
     where C:IClusterNode<INode>, E:IEnvironment<C>, F:INumber,
     S2:IPartitionStrategy, S1:IPartitionStrategy, S3:IPartitionStrategy
02.  {
03.     IE Environment {set;}
04.     IParData<S3, C, E, F> R {set;}
05.     IParData<S1, C, E, IArray2D<F> > B {set;}
06.     IParData<S2, C, E, IArray1D<F> > Y {set;}
07.  }
```

Figura A.2: *Interface IQ*

```
01.  public class IQOwnerMainImpl {
02.  static void Main(string [] args) {
03.      IQOwner iqowner = new IQOwnerImpl();
04.      iqowner.compute();
05.  }
06.  }
```

Figura A.3: *Método main em IQOwnerMainImpl*

```
01.  public class IQOwnerImpl<S2, C, E, F, S1, S3>:  IQOwner<S2, C, E, F, S1, S3>
     where S2:IEqualPartition, C:ICastanhaoNode, E:IMPIBasic<C>,
     F:IDouble, S1:IEqualPartitionByRows, S3:IPlaceInRoot{

02.    private E e = null;
03.    private IQ<S2, C, E, F, S1, S3> q = null;
04.    private IParData<S3, C, E, F> r = null;
05.    private IParData<S1, C, E, IArray2D<F» a = null;
06.    private IParData<S2, C, E, IArray1D<F» x = null;
```

```
07.  public IParData<S3, C, E, F> Q {
08.    set {
09.     this.r = value;
10.     this.q = value;
11.    }
12.  }
13.  public E Environment {
14.    set {
15.     this.e = value;
16.     this.q.Environment = value;
17.    }
20.  }
21.  public IParData<S1, C, E, IArray2D<F» B {
22.    set {
23.     this.B = value;
24.     this.q.B = value;
25.    }
26.  }
27.  public IParData<S2, C, E, IArray1D<F» Y {
28.    set {
29.     this.Y = value;
30.     this.q.Y = value;
31.    }
32.  }
33.  IQOwnerImpl() {
34.    this.iqimpl = DGAC.createInstanceFor("appexampleapp","compute","iqimpl");
35.    this.R = DGAC.createInstanceFor("appexampleapp","r","parData");
36.    this.B = DGAC.createInstanceFor("appexampleapp","b","parData");
37.    this.Y = DGAC.createInstanceFor("appexampleapp","y","parData");
38.    this.e = DGAC.createInstanceFor("appexampleapp","e","mpiBasic");
39.  }
40.  compute() {
41.    !attribute data to B,Y
42.    this.q.compute();
43.    !shows r
44.  }}
```

Figura A.4: *Classe IQOwnerImpl*

```
01.  public class IQImpl<C, E, F, S1, S2, S3>:  IQ<C, E, F, S1, S2, S3>
     where S2:IEqualPartition, C:ICastanhaoNode, E:IMPIBasic<C>,
     F:IDouble, S1:IEqualPartitionByRows, S3:IPlaceInRoot{
02.  {

03.    private IMatVecProduct<S2, C, E, F, S1> calculateBYU = null;
04.    private IRedistributeVec<S2, C, E, IArray1D<F» redistributeU,redistributeV = null;
05.    private IVecVecProduct<S2, C, E, F, S3> calculateUVr = null;

06.    private E e = null;
07.    private IParData<S1, C, E, IArray2D<F» b = null;
08.    private IParData<S2, C, E, IArray1D<F» y,u_,u,v = null;
09.    private IParData<S3, C, E, F> r = null;

10.  public IParData<S3, C, E, F> R {
11.    set {
12.     this.r = value;
13.     calculateUVr.R = value;
14.    }
15.  }

16.  public IParData<S3, C, E, F> Environment {
17.    set {
18.     this.e = value;
19.     redistributeU.Environment = value;
20.     redistributeV.Environment = value;
21.     calculateBYU.Environment = value;
22.     calculateUVr.Environment = value;
23.    }
24.  }

25.  public IParData<S1, C, E, IArray2D<F» B {
26.    set {
27.     this.b = value;
28.     calculateBY.B = value;
29.    }
30.  }

31.  public IParData<S2, C, E, IArray1D<F» V {
32.    set {
33.     this.v = value;
34.     redistributeV.V = value;
35.     calculateUVr.V = value;
36.    }
37.  }
```

```
38.  public IParData<S2, C, E, IArray1D<F>» U {
39.    set {
40.     this.u = value;
41.     redistributeU.U = value;
42.     calculateUVr.U = value;
43.    }
44.  }

45.  public IParData<S2, C, E, IArray1D<F>» U_ {
46.    set {
47.     this.u_ = value;
48.     redistributeU.U_ = value;
49.     calculateBYU.U_ = value;
50.    }
51.  }

52.  public IParData<S2, C, E, IArray1D<F>» Y {
53.    set {
54.     this.y = value;
55.     calculateBYU.Y = value;
56.    }
57.  }

58.  IQImpl() {
59.    this.calculateBYU = DGAC.createInstanceFor("appexample","BYU","matvecProduct");
60.    this.redistributeU = DGAC.createInstanceFor("appexample","redistributeU","ab");
61.    this.redistributeV = DGAC.createInstanceFor("appexample","redistributeV","b");
62.    this.calculateUVr = DGAC.createInstanceFor("appexample","UVr","vecvecProduct");
63.    this.U = DGAC.createInstanceFor("appexample","U_","parData");
64.    this.V = DGAC.createInstanceFor("appexample","V","parData");
65.    this.U_ = DGAC.createInstanceFor("appexample","U","parData");
67.  }

68.  void compute() {
69.      calculateBYU.compute();
70.      #pragma omp parallel sections
71.      {
72.       #pragma omp section
73.       redistributeV.synchronize();
74.       #pragma omp section
75.       redistributeU.synchronize();
76.      }
77.      calculateUVr.compute();
78.  }
79.  }
```

Figura A.5: *Código IQImpl, em C#*