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

Implementações de Conexões de Componentes CCA Distribuídos usando Java e MPI

Paulo Henrique Lopes Silva

Fortaleza – Ceará
Agosto 2009

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

Implementações de Conexões de Componentes CCA Distribuídos usando Java e MPI

Autor
Paulo Henrique Lopes Silva

Orientador
Ricardo Cordeiro Corrêa

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
Agosto 2009

Resumo

Várias áreas das ciências e engenharias utilizam simulações computacionais como um dos pilares das técnicas que a metodologia aplica em busca da compreensão de fenômenos. A crescente e simultânea evolução dos modelos e técnicas de simulação, por um lado, e dos sistemas computacionais de alto desempenho, por outro tem impulsionado pesquisadores a abordarem problemas mais complexos, com um escopo cada vez maior. No bojo dessa evolução surge a demanda por mecanismos de *software* que possibilitem aproveitar melhor todos os recursos computacionais que estão se tornando disponíveis e atender à crescente demanda das aplicações científicas.

Este novo cenário trás consigo um aumento considerável na complexidade no desenvolvimento de *software* para aplicações científicas. Existem vários esforços em mecanismos para lidar com esta complexidade e estender o desenvolvimento de aplicações envolvendo a combinação de diversas simulações inter-dependentes.

A tecnologia de componentes, por meio da noção de partes de *software* independentes e reutilizavéis, surge como um mecanismo viável para atender as características dessa nova demanda das aplicações científicas. Entretanto, em meio aos seus benefícios, a inserção do modelo de componentes no desenvolvimento de aplicações científicas não deve comprometer seu desempenho original, principalmente nas que utilizam o paralelismo.

Uma questão inerente ao uso de componentes em aplicações de alto desempenho é o tratamento de comunicações. Neste trabalho, o problema do suporte a comunicação entre componentes *Forró* via MPI é abordado. Através de avaliações das execuções de alguns modos de implementação propostos, o objetivo é estudar soluções que possam auxiliar a resolução do problema mencionado.

Abstract

Several areas of science and engineering using computer simulations as one of the pillars of the techniques that the methodology applied in the quest for understanding of phenomena. The growing and simultaneous development of models and simulation techniques, on the one hand, and high performance computer systems, is driven by other researchers to address more complex problems, with an increasing scope. In the midst of these developments comes the demand for mechanisms of software enabling enjoy all the computational resources that are becoming available and meet the growing demands of scientific applications.

This new scenario brings with it a considerable increase in complexity in the development of software for scientific applications. There are several mechanisms in efforts to deal with this complexity and extend the development of applications involving the combination of several inter-dependent simulations.

The technology of components, through the concept of parts of independent and reusable software, is a viable mechanism to meet this new demand characteristics of scientific applications. However, amidst their benefits, the integration of the model components in the development of scientific applications must not compromise their original performance, especially in using the parallel.

An issue inherent in the use of components in high performance applications is the treatment of communications. In this work, the problem of support for communication between components *Forró* via MPI is discussed. Through assessments of executions of some methods of implementation proposed, the goal is to explore solutions that can help solve the problem mentioned.

Dedicatória

Dedico este trabalho a minha mãe Maria Luzia, ao meu pai José Gregório, a minha irmã Rafhaely e ao meu irmão Vitor. Obrigado pelo amor, apoio constante e confiança irrestrita em todos os momentos. Vocês se constituem como a minha maior motivação.

Agradecimentos

Gostaria de agradecer a Deus por me proporcionar saúde e perseverança para alcançar meus objetivos ao longo da minha vida.

Ao professor Ricardo Corrêa pela confiança depositada em mim e pela competência na orientação e no desenvolvimento deste trabalho.

A UFC e FUNCAP pela oportunidade e pelo financiamento para a realização trabalho.

A minha namorada Débora Jales pelo amor e confiança incondicional, que me ajudaram muito na realização deste trabalho.

Aos meus avós maternos Manoel Jacinto (*In Memorian*), Geraldina Gomes e paternos João Lino, Lídia pelo incentivo e confiança depositados a mim desde a infância. Juntam-se a eles, todos os meus tios e primos, em especial agradeço a tia Fátima, Nevinha, Elineuza, Marillac, Raimunda. Aos tios Genildo, Geovâneo, José Leite, Joaquim Neto e Francisco.

Aos primos Jeo e Lunardo. Às primas Silvia, Luana, Caroline, Juce e Taisa.

Aos amigos João Paulo Oliveira e Fernando Gomes pelo constante incentivo.

Aos colegas do mestrado, que me ajudaram muito no processo de ambientalização à cidade de Fortaleza e à Universidade. Agradecer também aos colegas de apartamento Cristiano e Saul e ao amigo João Borges.

Aos mestres e funcionários do departamento de computação da UFC e aos colegas do CENAPAD.

Enfim, agradeço a todos que de alguma forma me ajudaram durante esta etapa.

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Breve contexto histórico

A computação de alto desempenho (CAD) é um ramo da ciência da computação que vem evoluindo rapidamente ao longo dos anos, trazendo consigo muitos benefícios em diversas áreas. Ela, por exemplo, é usada no auxílio ao desenvolvimento de novos fármacos, na fabricação de materiais sintéticos, no projeto de componentes eletrônicos para diversos setores da indústria, como o automobilístico [18, 58]. Ainda ajuda os cientistas a investigar as variações climáticas [50] etc. Essas aplicações tem o desempenho como característica principal. Com isso, os desenvolvedores procuram sempre extrair o máximo do poder de processamento do sistema computacional utilizada para que possam resolver os problemas eficientemente.

Na história recente da computação de alto desempenho sabe-se da variedade e do rápido crescimento dos sistemas computacionais. Esta história começa na década de 1980, com os sistemas de computação vetorial como os da *Cray Research*, *Fujitsu* etc. Processadores vetoriais com memória compartilhada formam um sistema poderoso a ser usado eficientemente pelas aplicações de alto desempenho, como as simulações numéricas. Segue-se a este a disponibilização de sistemas paralelos formados por processadores, do tipo RISC, que compartilham memória. Estes sistemas, chamados SMPs (*Symmetric MultiProcessing*), foram usados em muitas aplicações, mas tinham a desvantagem de possuirem um grau de escalabilidade baixo. Fazem parte dessa história também, as sistemas com memória distribuída como supercomputadores ou MPPs (*Massively Parallel Processing*) fabricados pela Intel, Meiko, IBM etc. Eles são formados, essencialmente, por um conjunto de

processadores com suas próprias memórias e interligados por redes dedicadas e de alta velocidade. Outros sistemas para computação de alto desempenho são os COWs (*Clusters of Workstations*). Eles se assemelham estruturalmente aos supercomputadores, mas são formados por estações de trabalho ligadas por uma LAN (*Local Area Network*). A diferença é que os COWs podem ser obtidos por um custo bem inferior, por serem formados por computadores bem mais baratos e que vem crescendo muito em desempenho. Além disso, a rede que os interliga nao é tão dedicada quanto a dos supercomputadores [58].

A cada novo sistema computacional que surgia, um certo tempo era dedicado para que os desenvolvedores de aplicações de alto desempenho descobrissem as suas características com o propósito de extrair o maior desempenho possível. E devido a esta variedade de sistemas computacionais, algumas tentativas foram feitas na definição de modelos e padrões de programação, para diminuir o esforço humano nessa migração entre sistemas computacionais [58]. Para que estes padrões fossem aceitos pelos desenvolvedores, eles não deviam comprometer o desempenho das aplicações na busca por facilidades de programação. Entre os principais padrões para programação paralela estão MPI (*Message Passing Interface*) [4], HPF (*High Performance Fortran*) [59], GA (*Global Arrays*) [61] e OpenMP [60].

Todos os sistemas computacionais citados são usadas cotidianamente juntamente com o modelo de programação que mais se adequa à sua arquitetura de memória, compartilhada ou distribuída. Contudo, com o surgimento e a rápida disseminação da internet, iniciaram-se projetos na perspectiva de recursos dispersos e heterogêneos poderem cooperar em uma aplicação específica. Com isso, surgia mais um sistema poderoso, a grade computacional. Todos os sistemas de computação e os padrões de programação citados podem fazer parte de uma grade, aumentando ainda mais o poder computacional e o alcance que as aplicações podem ter. Entretanto, a complexidade no desenvolvimento de *software* aumenta consideravelmente [18, 41, 44].

Dadas as novas possibilidades, o desejo de abordar níveis maiores de detalhes nas aplicações científicas fez com que um número grande de especialistas envolvidos como físicos, químicos, biólogos, engenheiros e cientistas da computação, entre outros, se esforçassem para aperfeiçoarem ou criarem novos mecanismos (métodos científicos, algoritmos, modelos de programação e virtualização) para atender a esta nova demanda das aplicações, onde cada pequena aplicação pode fazer parte de uma aplicação maior, podendo envolver especialidades de varios domínios, tentando

diminuir a complexidade no desenvolvimento [18, 43]. Contudo, uma característica ainda permanece. O aperfeiçoamento ou a criação de artefatos de *software*, que permitam que os desenvolvedores possam programar em ambientes tão heterogêneos, não devem comprometer o desempenho das aplicações originais. Este é o principal requisito dos desenvolvedores de aplicações de alto desempenho.

1.2 A tecnologia de componentes

Uma das tentativas que fazem parte do esforço crescente dos desenvolvedores de *software* para oferecer um nível maior de abstração no tratamento da nova demanda das aplicações foi a criação da noção de desenvolvimento de *software* baseado em componentes. Nessa abordagem, componentes encapsulariam códigos funcionalmente independentes que podem ser combinados para formar uma aplicação. Esse fato trás maiores níveis de reuso e gerenciamento de *software*, pois os componentes podem ser aperfeiçoados ou substituídos sem comprometer a aplicação inteira [45].

Além disso, o desenvolvimento de componentes pode ser separado do desenvolvimento de aplicações. Os desenvolvedores de componentes se preocupariam apenas com o código que será encapsulado por um componente, enquanto que os desenvolvedores de aplicações se concentrariam na melhor forma de combinar componentes para atender aos requisitos de uma certa aplicação [1].

Vários domínios já utilizam a tecnologia de componentes em aplicações como: sistemas distribuídos, aplicações gráficas e *desktop* e recentemente em sistemas embarcados [17]. A seguir, serão apresentadas as principais características da tecnologia de componentes estabelecendo uma relação entre paradigmas de programação importantes e presentes no desenvolvimento de *software*.

1.2.1 Componentes vs objetos

O princípio fundamental dos componentes é o de ser uma peça de *software* independente com funcionalidade bem definida [11]. A possibilidade de se desenvolvê-los usando objetos faz com que seus conceitos sejam frequentemente confundidos. Além de ser uma unidade independente, um componente não possui estado persistente [12]. Em adição a isso, eles têm que claramente especificar o que disponibilizam para uso externo e o que requerem. Em outras palavras, componentes encapsulam sua implementação e interagem com o ambiente através de *interfaces* [1]. Por não possuir estado, componentes podem ser inseridos e ativados em um sistema particular que necessite da funcionalidade que um deles possa

oferecer. Já os objetos relacionam-se com conceitos como instanciação, identidade e encapsulamento. Eles possuem identidade única e estado, que pode ser persistente. O fato de possuir estado impossibilita uma maior flexibilidade dos objetos em relação aos componentes. Sendo assim, objetos não podem tão prontamente ser substituídos por outros, em algum sistema, por causa da dificuldade ou inviabilidade de reproduzir o estado do objeto a ser substituído pelo novo [12].

Os componentes de *software* foram desenvolvidos tentando conquistar o sucesso dos componentes de *hardware*, com entradas e saídas bem definidas. Além do reuso, outras características importantes podem ser citadas como: variedade de formas, extensibilidade, alto nível de gerenciamento de mudanças e a promoção da padronização, entre outras [1,11,12]. O objetivo é tentar aumentar a produtividade e conquistar a complexidade exigida pela nova demanda das aplicações, tanto no meio comercial, quanto no meio científico.

Contudo, ainda não existe um padrão na definição de componentes. O princípio de funcionamento é igual, mas o que existe, hoje, é a definição de componentes de acordo com certo modelo de componentes. Esse modelo especifica os aspectos de criação, configuração e gerenciamento, ou seja, define as regras de comportamento (interação) dos componentes [18]. Como exemplo de modelos de componentes pode-se citar: o COM (*Microsoft's Component Object Model*) [35], o CORBA (*Common Object Request Broker Architecture*) [34], o CCA (*Common Component Architecture*) [10] etc.

1.2.2 Componentes vs serviços

Como já mencionado, a tecnologia de componentes facilita o gerenciamento da complexidade inerente à nova demanda das aplicações e aumenta a reusabilidade. Por outro lado, o crescimento da demanda por aplicações e sistemas voltados para a área comercial, utilizando a internet como meio de comunicação, tem motivado a evolução da abordagem de desenvolvimento de *software* baseada em serviços [17]. Ela oferece grande flexibilidade de integração de sistemas distribuídos que são desenvolvidos em plataformas e tecnologias diferentes. Além disso também foca na reusabilidade e eficiência no desenvolvimento de *software*.

A tecnologia de serviços, assim como a de componentes, surgiu e está evoluindo em busca de oferecer ambientes cada vez mais interoperáveis. Ambas possuem abordagens similares. Isto pode resultar em um eficiente projeto que combine estes paradigmas. Contudo, é preciso que os conceitos dos dois sejam bem compreendidos

[17].

O princípio básico de funcionamento destas tecnologias é similar. Ambas Possuem partes independentes de *software* como a base do desenvolvimento de sistemas. Esse fato oferece um grande dinamismo na substituição destas partes. Entre algumas diferenças destaca-se o fato de que componentes necessitam de uma especificação explícita de um determinado modelo, enquanto que serviços alcançam um nível maior de independência através de um eficiente processo de descrição [17].

1.2.3 Características importantes dos componentes

A tecnologia de componentes possui outras características importantes, além das já citadas. A indenpendência funcional dos componentes fornece a possibilidade de se ter um fluxo de execução dinâmico das aplicações. Em contraste com métodos utilizados por programadores UNIX, que conectavam programas simples, através de *pipes* e redirecionamentos, a execução de uma aplicação com o conceito de componentes difundido atualmente não teria um fluxo linear, unidirecional [27].

Além disso, componentes podem ser trocados sem que haja uma nova compilação da aplicação. O processo de conexão é separado do processo de implementação das funcionalidades dos componentes. Este fato concede muitas vantagens a essa abordagem [18]. Componentes são conectados através de *interfaces*, que são uma espécie de contrato entre duas entidades. Nestas *interfaces* estão as funcionalidades que um componente disponibiliza e as que ele requer para a execução [11, 18, 20]. Com isso, otimizações podem ser realizadas separadamente nas implementações dos componentes, onde sempre existem formas de aperfeiçoamentos, sem a necessidade de alterar a sua *interface* [1].

Muitos componentes podem ter funcionalidades compatíveis com as características de determinados sistemas e através de um projeto bem definido podem ser reusados em uma variedade de aplicações. A crescente adoção da tecnologia de componentes no desenvolvimento de aplicações na indústria, no comércio e na ciência tem estabelecido um bom nível de reusabilidade, simplicidade de manutenção e acoplamento dessa abordagem (em relação a outros paradigmas) [18, 41, 44].

Para que os sistemas possam fazer uso destes benefícios eles devem adotar um certo modelo de componentes. Após isso, passam a utilizar plataformas para desenvolver aplicações de acordo com os padrões do modelo adotado (mais detalhes sobre modelos e plataformas serão apresentados nos capitulos 2 e 3). Modelos como o

COM (DCOM, COM+), o CORBA e o JavaBeans (*Sun's Enterprise JavaBeans*) [33] são bem aceitos e largamente utilizados em aplicações do meio comercial. Contudo, eles não possuem os requisitos importantes que são abordados pela computação científica como suporte a algumas linguagens e a tipos de dados, bem como sistemas operacionais utilizados somente para estas aplicações. Isto, por sua vez, não motiva a adoção destes pelos desenvolvedores de *software* da comunidade científica [1, 20]. Então, se faz necessária a criação de modelos específicos para as aplicações da comunidade científica [18].

1.3 Objetivo

O modelo CCA surgiu com o próposito de oferecer a comunidade científica os benefícios que a tecnologia de componentes pode trazer ao desenvolvimento de aplicações. Através de sua especificação, CCA determina, entre outras coisas, a definição dos componentes e o mecanismo de conexão entre eles (portas *provides/uses*). Diante deste cenário, encontra-se em desenvolvimento uma plataforma de desenvolvimento baseada no modelo CCA, chamada *Forro* [42]. É implementada em Java pelo grande número de características positivas que esta linguagem pode oferecer [66]. Entretanto, sendo voltada para aplicações de alto desempenho, esta plataforma deve ser capaz de manipular componentes e conexões em linguagem nativa, por razões de desempenho.

A partir deste contexto, esse trabalho busca ser um instrumento de estudos que auxilie o processo de integração entre o *Forro* e as aplicações MPI. Isto será realizado por meio dos seguintes ítens: primeiramente é preciso formular requisitos de aplicações de alto desempenho no que diz respeito ao suporte à comunicação em programas paralelos, levantar as principais funcionalidades requeridas pelas comunicações em programas paralelos baseados em MPI e identificar possíveis fontes de perda de eficiência em uma modelagem por componentes desses programas.

Após isso, é necessário levantar conceitos da engenharia de *software* disponíveis no modelo CCA e as dificuldades técnicas de compatibilidade dos conceitos de programação por componentes segundo CCA, sobretudo as conexões, com MPI. Como exemplo, o fato de chamadas a funções MPI serem transformadas em chamadas de método em uma porta. Seguindo se a isso é preciso listar soluções viáveis, a partir das possibilidades oferecidas pelo *Forro*, analisando criticamente as vantagens e desvantagens deste tipo de integração, e realizar avaliações experimentais.

1.4 Organização do documento

Esse documento está organizado da seguinte forma: no capítulo 2 será feita uma contextualização da abordagem de componentes e sua relação com a computação de alto desempenho. Em seguida, no capítulo 3, é apresentada uma descrição dos modelos de componentes voltados para esta abordagem. O capítulo 4 mostra algumas características da plataforma *Forro* e a possibilidade de integração com aplicações MPI, por meio da descrição de mecanismos para a comunicação de componentes *Forro* via MPI. No capítulo 5, está a descrição do processo de implementação de soluções propostas, tendo em vista os mecanismos oferecidos pelo *Forro*. Faz parte deste processo, a descrição dos modos de implementação propostos e a análise de suas execuções. No capítulo 6 estão as considerações finais e as propostas para trabalhos futuros. Logo após, estão as referências bibliográficas.

Capítulo 2

Componentes, Plataformas e Computação de Alto Desempenho

2.1 Componentes no desenvolvimento de simulações

Pesquisadores de várias áreas como físicos, químicos, biólogos etc, fazem uso da computação para tentar compreender melhor vários aspectos de fenômenos físicos. As aplicações de alto desempenho (simulações computacionais) possuem um papel fundamental no tratamento e na obtenção rápida de resultados de vários problemas destas áreas, beneficiando-as com descobertas importantes [44]. Entretanto, o desejo de abordar problemas mais complexos tem resultado em um aumento na variedade de especialidades (domínios) que uma simulação pode abranger. A complexidade no desenvolvimento destes *softwares* também aumenta [43].

O tratamento desta nova demanda de aplicações se torna possível por dois motivos. O primeiro é o crescimento da capacidade computacional nos sistemas de computação. Ele tem conduzido cientistas da computação a desenvolver algoritmos novos e mais complexos para aproveitar melhor as novas características destes sistemas [41]. O segundo está relacionado com os avanços científicos e, com isso, a possibilidade de tratar problemas considerados difíceis, até o momento. Contudo, estes problemas podem envolver uma vasta gama de áreas para uma resolução completa, precisa e que não perca em fidelidade [41].

Dados os fatos, pesquisadores ou até instituições raramente possuem recursos e especialidades suficientes em todas estas áreas. O que acontece é que eles concentram-se em sua própria especialidade no momento do desenvolvimento das simulações [43, 45].

A ocorrência deste cenário tem gerado enormes investimentos, por parte da ciência da computação, em modelos para melhorar o processo de desenvolvimento de *software* de simulação científica. Melhorar no sentido de oferecer mecanismos para combinar ou acoplar tecnologias de diferentes áreas em uma aplicação integrada sem perder sua precisão e fidelidade. Este acoplamento requer que a saída de um processo da simulação possua condições de ser utilizada como entrada em outro processo da simulação [45]. Outros fatores importantes para serem abordados nestes modelos são as questões relativas a interoperabilidade de linguagens e padrões utilizados nos códigos legados, que dificultam bastante este acoplamento na prática [18, 43].

As simulações científicas são, geralmente, formadas pela combinação de vários algoritmos com o objetivo de solucionar um conjunto de problemas específicos [18]. Uma das estratégias usadas para a integração destes algoritmos é a modularização, fazendo com que eles estejam em procedimentos independentes. Então, uma simulação, na forma tradicional, é composta por vários procedimentos que possuem parâmetros bem definidos para a execução [41].

A tecnologia de componentes remete, como já citado, a idéia de composição de aplicações através de partes independentes de *software*. Entretanto, esta noção de independência é estendida em relação ao modelo tradicional das simulações, pois componentes se comunicam através de *interfaces* e que não necessariamente mudam se houver a necessidade de troca de algum componente. A *interface* permanece a mesma, independente de qual implementação de um dado componente esteja pronta ou em execução [1]. Isto proporciona um controle maior no desenvolvimento e no auxílio ao tratamento da complexidade inerente da nova demanda de *softwares* de simulação científica [18, 41].

2.2 Elaboração de componentes vs configuração e execução

Uma das características da tecnologia de componentes é a possibilidade de encapsular códigos de linguagens nativas (que são geralmente usadas nas simulações). Com isso, tem-se a possibilidade de inserir esta tecnologia no contexto das simulações científicas. Entretanto, esta inclusão tem que ser realizada e apresentada de forma eficiente, não comprometendo aspectos importantes da simulação original, como o desempenho.

O desenvolvimento de componentes, neste caso, pode ser realizado desde a sua fase inicial, ou seja, desenvolver componentes que encapsulam códigos legados desde a concepção destes. Outra forma, é o desenvolvimento de componentes que

encapsulam códigos já existentes. Neste caso, existe a necessidade de desenvolver mecanismos para a comunicação com o código legado, de forma que este não precise de qualquer modificação [18]. Este procedimento deve ser realizado de forma eficiente para não comprometer o desempenho.

2.3 Plataformas de desenvolvimento

As plataformas de desenvolvimento são elementos muito importantes no contexto da tecnologia de componentes, pois eles funcionam como uma estrutura de base para o desenvolvimento de aplicações. Devido a isto, pode-se, frequentemente, encontrar nas literaturas que versam sobre este assunto a utilização do termo *framework* para nomear este tipo de estrutura. Portanto, um pequeno esclarecimento se faz necessário nesse momento.

A engenharia de *software* define *frameworks* como arcabouços de classes que funcionam como estrutura de suporte ao desenvolvimento e organização de projetos de *software*. Possuem bibliotecas de código e outros elementos que facilitam a composição de um novo projeto [46].

Contudo, a noção de *framework* utilizada pela comunidade de computação de alto desempenho é um pouco diferente. Neste caso, um *framework* também tem a função de auxiliar o desenvolvimento de aplicações, mas se diferencia pelo fato de ser um programa e de implementar características de certos modelos, auxiliando somente no desenvolvimento de aplicações restritas a determinados domínios. Ou seja, não é extensível ao ponto de poder auxiliar o desenvolvimento de qualquer aplicação, como o *framework* definido pela engenharia de *software*.

A noção ou definição de *framework* adotada pelos desenvolvedores de aplicações de alto desempenho, na realidade, se adequa bem mais ao que foi definido como plataforma de desenvolvimento. Entretanto, apesar de existirem algumas divergências entre sua definição, esta noção continua sendo muito usada pelos desenvolvedores de aplicações científicas. As estruturas de suporte mencionadas nesse texto possuem características compatíveis com a definição de plataforma de desenvolvimento, portanto é essa a nomenclatura que será utilizada.

No contexto do desenvolvimento de aplicações científicas, a adoção de plataformas de componentes se coloca como um importante auxílio neste processo, atribuindo-lhe mais rapidez e qualidade [22]. Eles realizam operações específicas como gerenciar a criação e as conexões de componentes [27]. Contudo, deve existir uma justificativa plausível para que os cientistas passem a utilizar este tipo de

plataforma. As novas demandas das simulações científicas requerem interação entre, possivelmente, vários domínios e modelos [43]. As plataformas devem participar neste momento intermediando esta interação de forma que ela não atribua custos computacionais extras. Portanto, a implementação de um mecanismo de integração deve ser realizada de maneira eficiente do ponto de vista da aplicação.

Através das plataformas, os usuários podem construir suas aplicações combinando os componentes de acordo com as funcionalidades requeridas [22]. Essas plataformas podem utilizar *interfaces* com o usuário de forma que estes possam configurar suas aplicações de maneira elegante. No momento da configuração da aplicação, ela tem a responsabilidade de instanciar e estabelecer as conexões entre os componentes [27]. A Figura 2.1 apresenta a *interface* de uma plataforma de componentes baseada no modelo CCA, a Ccaffeine [24].

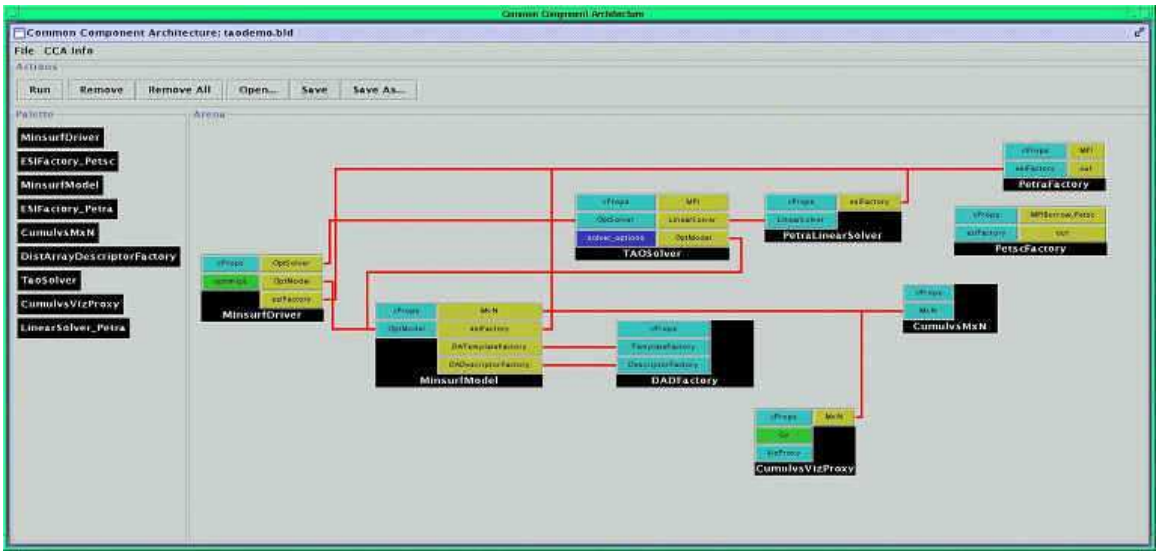

Figura 2.1: *A interface com o usuário de uma plataforma CCA em operação [27].*

A janela principal da *interface* com o usuário possui duas partes. Uma paleta no lado esquerdo e uma arena no lado direito. Na paleta estão listados todos os tipos de componentes disponíveis. Para criar uma instância de um componente de um certo tipo, basta clicar em um dos componentes da paleta e arrastar para a arena. Os nomes dos componentes devem ser únicos, então se o usuário não colocar um nome, a plataforma atribui um nome automaticamente. Os componentes instanciados são representados por caixas pretas e se comunicam com outras instâncias através de portas. Para conectar as portas o usuário clica em uma porta e a plataforma lista todas as candidatas para a ligação ou conexão. O usuário clica em uma delas e a

conexão será estabelecida [27]. Alguns conceitos particulares ao modelo CCA foram previamente expostos com o propósito apenas de apresentar de maneira mais clara o exemplo. Maiores detalhes são citados no Capítulo 3.

Apesar de concepções como estas serem bastante razoáveis do ponto de vista do usuário, a realização delas não é tão fácil assim. As plataformas de desenvolvimento existentes possuem muitas soluções parciais de alguns destes pontos e em outros eles ainda são dificilmente abordados [22, 46].

2.4 Conexões entre componentes

A comunicação entre componentes se trata de um aspecto de suma importância no contexto das aplicações científicas. Como já citado, a definição do conceito de componentes, hoje, é particular em cada modelo. Isto não é diferente para as formas de conexão entre componentes. Por isso, esta seção mostrará as formas de conexão entre componentes de uma maneira genérica. No capítulo 3, alguns modelos de componentes serão apresentados com atenção especial ao modelo CCA, que está em foco neste trabalho, e a forma de interação entre componentes CCA.

Os componentes que compõem uma aplicação (simulação), de pequeno ou grande porte, podem se comunicar através de conexões locais ou remotas. A conexão local ocorre quando os componentes estão em um mesmo espaço de endereçamento. Sendo assim, ela tem um custo equivalente ao de uma chamada a uma função ou a um método [27]. Na Figura 2.2, pode-se observar um exemplo de conexão local de componentes. Cada um dos componentes representa, individualmente, uma determinada funcionalidade e são conectados da maneira apresentada. Estas conexões podem formar uma aplicação, ou parte de uma aplicação, que será executada por estes componentes neste espaço de memória.

Já com a conexão remota, os componentes podem fazer parte de uma certa execução mesmo não estando na mesma memória. A interação entre componentes remotos ocorre através de algum mecanismo de comunicação distribuída como passagem de mensagens, chamada de procedimento remoto etc. O custo de comunicação, neste caso, é maior devido a infraestrutura de rede existente entre os componentes [18]. A conexão dos componentes deve existir no momento da configuração da aplicação, mas este custo não deve interferir no momento da execução da aplicação. Ele é um fator muito importante, pois um dos grandes objetivos no desenvolvimento de simulações é conseguir executá-las extraindo o máximo desempenho possível do sistema computacional subjacente. Este aspecto

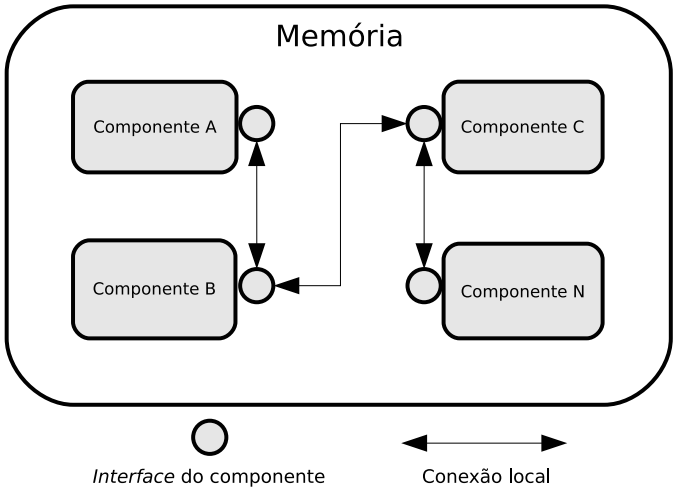

Figura 2.2: *Exemplo de conexões entre componentes na mesma memória.*

é ainda mais forte em simulações que fazem uso do paralelismo, onde as latências das interconexões paralelas são medidas em microssegundos [18]. A introdução de qualquer outra tecnologia, mesmo visando uma maior produção científica, tem que ser realizada de maneira a não comprometer o desempenho das simulações originais.

Na Figura 2.3 tem-se um exemplo de duas aplicações que se comunicam, entre si, com conexão remota e também com a conexão direta. As aplicações são diferenciadas, neste exemplo, pelos números após o nome dos componentes. A aplicação número um, envolve conexão direta entre os pares A,B e D,E e comunicação remota entre os componentes A e D. Na aplicação dois, a conexão remota ocorre entre C e E e conexão direta entre E e F. Outro fator, que também é mostrado nos exemplos, é que componentes podem ao mesmo tempo participar de conexões diretas ou remotas estando na mesma aplicação ou não.

As *interfaces* dos componentes descrevem exatamente o que eles disponibilizam e o que eles necessitam para executar [20]. No caso de comunicação remota, o processo de conexão pode envolver alguns passos. Até mesmo outros componentes, que terão funcionalidades específicas de conexão, poderão ser criados. Isto varia de modelo para modelo. Ou seja, neste processo, o *framework* pode até incorporar custos computacionais consideráveis, mas, diante da conexão estabelecida, as comunicações entre os componentes que encapsulam códigos legados não devem arcar com estes custos computacionais.

Por exemplo, os pesquisadores utilizam bastante a paralelização para desenvolver aplicações com o objetivo de obter mais eficiência e desempenho. Para que esta

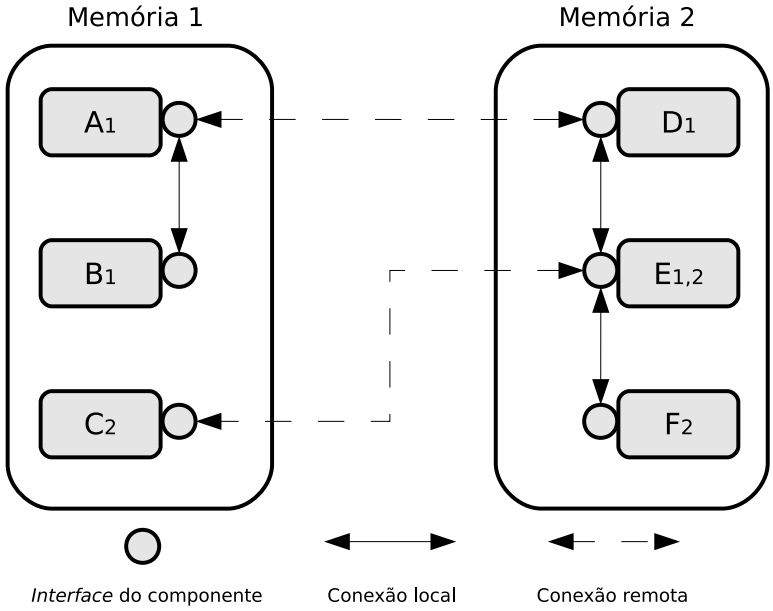

Figura 2.3: *Exemplo de conexões entre componentes remotos.*

abordagem de componentes seja aceita, a execução de uma aplicação deste tipo em uma plataforma de componentes deve ter desempenho igual ou bem próximo ao da execução original, sem a plataforma [18].

O objetivo com a adoção deste tipo de plataforma de *software*, baseada em componentes, é tornar mais fácil a vida de quem desenvolve simulações. Elas podem conceder um grande alcance no desenvolvimento de aplicações científicas. A possibilidade de desenvolver simulações que englobam especialidades de vários domínios se torna mais próxima devido ao que esta abordagem oferece. Devido a isto, simulações já existentes poderão ser incorporadas a estas plataformas, abrindo novas possibilidades de acoplamento entre elas. Entretanto, as principais características das simulações tradicionais devem ser tratadas com muito cuidado, para que se ganhe em alcance sem perder desempenho.

Pelo que foi apresentado, a tecnologia de componentes vem se tornando uma abordagem natural para o desenvolvimento de aplicações que possuam especialidades em vários domínios [18]. Ela adiciona características importantes e que podem acelerar a pesquisa e o desenvolvimento científico através do reuso e facilidade de integração [41, 45].

2.5 Requisitos da computação de alto desempenho

A computação, de uma forma geral, se coloca como uma importante área no processo de auxílio a resolução de vários problemas. O desenvolvimento de *software*, que ocupa um lugar importante neste contexto, começa desde a concepção de um determinado mecanismo para resolver um dado problema. Entretanto, o escopo destes problemas tem aumentado devido às novas necessidades dos usuários tanto na área comercial (integração de sistemas, por exemplo), como na área científica (desenvolvimento rápido, eficiente e multi-institucional de simulações). A abordagem de modelos e plataformas de componentes propõe a disponibilização de mecanismos para que os desenvolvedores possam tratar da complexidade inerente desta nova demanda de aplicações [18, 22].

Muitos modelos de componentes já auxiliam no processo de desenvolvimento de *software* na área comercial [1]. Contudo, para que possam beneficiar a computação científica de alto desempenho eles devem atender aos seus requisitos [18]. Na seção anterior foi mostrado um exemplo sobre as aplicações paralelas e o fato de que elas não podem sofrer com grandes latências de rede, quando incorporadas a uma plataforma. Outros aspectos devem ser abordados pelos modelos para que possam atender às características da computação de alto desempenho.

Além deste custo adicionado das redes, pode-se relatar também o custo relacionado com a virtualização. Algumas plataformas existentes, destinadas às aplicações de alto desempenho, são executadas sobre uma máquina virtual [19, 22]. Isto gera um custo devido à uma possível intervenção desta nas interações entre os componentes. As tarefas realizadas pela plataforma devem pesar o mínimo possível, na comunicação e na computação, entre processos em uma aplicação de alto desempenho.

A incorporação de códigos legados também é outro requisito importante [18]. Muitos destes códigos estão escritos em linguagens diferentes pelo fato de existirem várias vantagens e desvantagens entre elas [45]. O que vai definir a utilização de uma certa linguagem é o seu nível de adequabilidade às aplicações de um domínio específico. Com isso, é preciso que hajam, nos modelos de componentes de computação científica, mecanismos que garantam a interação entre códigos escritos nas linguagens mais importantes da comunidade e o suporte a aritmética complexa e matrizes multidimensionais (podendo ser alocadas dinamicamente) [41]. Esta interação deve ocorrer de forma a não agregar sobrecargas maiores na aplicação

e ser portável para muitas arquiteturas [44]. Muitas destas arquiteturas podem ser observadas na lista Top 500 [47].

Nesse trabalho, tem-se uma plataforma de componentes, *Forro*, que é executada por meio de uma máquina virtual. Ele oferece mecanismos para que o processamento das aplicações de alto desempenho seja realizado totalmente em código nativo. O objetivo é, a partir disso, propor soluções de implementação que auxiliem no processo de integração entre o *Forro* e as aplicações que utilizam o MPI como paradigma de comunicação paralela. Dentro do que o *Forro* oferece, a intenção é buscar formas de implementar as conexões entre componentes via MPI, conservando ao máximo seu desempenho original.

No próximo capítulo, serão apresentados alguns modelos de componentes e suas abordagens para o contexto da computação de alto desempenho. As características de cada um serão mostradas. O objetivo é compreender o modelo de programação proposto por cada um e investigar como eles abordam os requisitos de computação de alto desempenho.

Capítulo 3

Modelos de Componentes para Aplicações de Alto Desempenho

As aplicações de alto desempenho contemporâneas demandam por novos mecanismos de desenvolvimento que consigam atender seus requisitos fundamentais como eficiência e desempenho. O surgimento do paradigma da programação baseada em componentes, que habilita o desenvolvimento de aplicações a partir de outros componentes pré-construídos, aumentando o nível de reusabilidade, ganhou espaço na pesquisa científica e abrange diversos esforços na busca por modelos que atendam aos requisitos da comunidade de computação de alto desempenho [18, 43].

3.1 Modelos de componentes

A tecnologia de componentes já é utilizada, satisfatoriamente, no meio comercial [1]. Entretanto, a computação científica de alto desempenho não é beneficiada pelas soluções adotadas na área comercial [43]. Isto acontece pelo fato de as ferramentas de componentes desenvolvidas para aplicações comerciais não alcançarem os requisitos de execução rápida e escalável pretendidos pelos cientistas [1]. O fato de não existir um modelo de componentes que atenda a estes requisitos, impulsionou a pesquisa e o desenvolvimento de modelos específicos para aplicações ditas de alto desempenho [18]. Nas próximas seções serão apresentados alguns modelos de componentes voltados a este tipo de aplicação. O objetivo deste capítulo não é a apresentação detalhada do estado-da-arte sobre modelos de componentes para CAD. O leitor interessado pode consultar as referências [1, 5, 13, 15]. Busca-se com este capítulo apresentar duas informações relevantes no contexto do estudo que é apresentado nesse trabalho. A primeira informação é a variedade de questões, e

respectivas soluções, que aparecem quando se impõe os requisitos de CAD. Para atingir esse objetivo, escolhemos modelos que nos permitem discutir noções de conexão, componentes distribuídos, paralelismo, implementações em Java, utilização de MPI e computação em grade. Dentre os modelos escolhidos, é dado destaque ao modelo CCA, que será novamente bastante mencionado no capítulo 4. A segunda informação relevante deste capítulo é a demonstração da penetração do conceito de programação por componentes em CAD. Ao lado dos diferentes modelos que são apresentados, são citadas implementações de plataformas que atestam o fato de a área de CAD estar aceitando a programação por componentes como uma via para o futuro.

3.2 CORBA

CORBA (*Common Object Request Broker Architecture*) [29] é a solução de objetos distribuídos desenvolvida pela OMG [34] (*Object Management Group*). Esta arquitetura propõe criar as aplicações distribuídas cliente/servidor interoperáveis através do uso de uma IDL (*Interface Definition Language*). A IDL permite que os serviços sejam descritos de forma que clientes e servidores possam se comunicar mesmo estando implementados em linguagens diferentes [54]. A comunicação entre os elementos participantes ocorre por intermédio dos ORB's *(Objects Request Brokers)*, que se responsabilizam pelo transporte e tradução das solicitações e respostas. Desta forma, CORBA gerencia a heterogeneidade de linguagens, computadores e redes [5].

Resumidamente, para que uma aplicação seja criada, o desenvolvedor deve descrever a *interface* do serviço em IDL. O resultado da compilação desta descrição gera os chamados *stubs*, do lado do cliente, e os *skeletons*, do lado do servidor. A função do *stub* é interceptar as invocações do cliente e transmití-las ao servidor através dos ORB's. Já o *skeleton*, recebe as invocações do cliente e as repassam aos determinados serviços para que possam realizar a sua computação [5].

Sendo um modelo que propõe a interoperabilidade, CORBA chamou a atenção da comunidade científica, principalmente à comunidade de computação de alto desempenho, também por outros fatores como: suporte a tipos primitivos e tipos complexos, adaptação a sistemas distribuídos, independência de arquitetura e sistema operacional. A sua última versão engloba também o CCM [5] (*The CORBA Component Model*) promovendo a junção deste modelo com a flexibilidade da abordagem baseada em componentes.

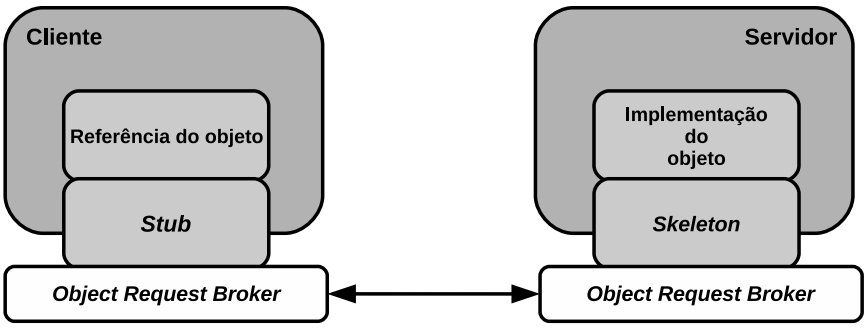

Figura 3.1: *Comunicação cliente/servidor no modelo CORBA.*

Oferecer formas de paralelismo é uma grande necessidade no contexto das aplicações de alto desempenho. O projeto GridCCM [5] é uma extensão do CCM e foi criado com o objetivo de adicionar o suporte ao paralelismo ao seu modelo de componentes. A especificação baseia-se no modelo de execução SCMD (*Single Component Multiple Data*) que define uma forma de execução para componentes paralelos similar ao SPMD (*Single Process Multiple Data*) para programas paralelos [54]. Em uma execução SPMD, cada processo executa o mesmo programa, mas com dados diferentes [1]. Cada programa troca dados através de bibliotecas de comunicação como MPI [4] (*Message Passing Interface*). A idéia do GridCCM é continuar com o modelo SPMD para a comunicação entre processos de um componente, utilizando CORBA para a comunicação com outros componentes [5]. A seguir serão mostrados outros projetos relevantes envolvendo o suporte ao paralelismo.

3.2.1 PARDIS

O ambiente PARDIS [3] foi umas das primeiras tentativas de introduzir objetos paralelos em CORBA. Nele está definido um novo gênero de objetos, chamados de objetos SPMD, que é uma extensão dos objetos CORBA. Questões como a distribuição de dados são gerenciadas por modificações na IDL. Isso oferece uma generalização da sequência CORBA de execução, denominada sequência distribuída. PARDIS objetiva a programação, com objetos SPMD, similar ao padrão de objetos CORBA. Ele também permite a sobreposição de execução entre clientes e servidores a sua habilidade de executar invocações do tipo não bloqueantes.

3.2.2 PaCO

PaCO ou *Parallel CORBA Object* [54] é outra tentativa para o suporte, por parte do CORBA, ao paralelismo. Implementa o conceito já definido de objetos

paralelos CORBA. PaCO estende a conhecida linguagem de definição IDL com novas funcionalidades. Isso resulta na capacidade de especificação do número de objetos CORBA que farão parte de um objeto paralelo e na distribuição de dados referentes aos parâmetros de uma operação. *Stubs* e *skeletons* são gerados pela IDL extendida e utilizam operações MPI para a redistribuição de dados e operações coletivas.

3.2.3 PaCO++

PaCO++ [6], *Portable Parallel CORBA Objects*, é uma continuação do projeto PaCO, compartilhando as mesmas definições de objetos paralelos e os mesmos objetivos. Este modelo permite que códigos SPMD sejam encapsulados em objetos paralelos. O paralelismo é suportado graças a uma camada de *software*, chamada de camada PaCO++. Este código é inserido entre o código do usuário e os *stubs/skeletons* do CORBA. Esta camada intercepta invocações dos usuários para gerenciar questões referentes a chamadas paralelas. Em resumo, um objeto PaCO++ é uma coleção de objetos CORBA que compartilham um contexto e que são capazes de invocar e receber invocações paralelas.

3.3 Fractal

Fractal [13] foi desenvolvido como um modelo de componentes que incorpora em sua especificação o suporte a características como: extensibilidade e adaptabilidade. Ele busca promover a noção de componentes dotados de mecanismos que possibilitam a realização de reconfigurações ou trocas. As características citadas ganharam notoriedade na nova demanda de aplicações de alto desempenho. Outros modelos comercialmente conhecidos não possuem suporte total à estas características.

A possibilidade de que as configurações de determinados componentes sejam alteradas, trás mais dinamicidade ao desenvolvimento de aplicações científicas, fazendo com que estas se adaptem a certas linguagens e plataformas diferentes, bem como ganhem em desempenho, pois as configurações dos componentes podem ser feitas de acordo com a arquitetura subjacente [13].

As principais características deste modelo são: hierarquia e composição de componentes (para ter uma visão uniforme das aplicações em vários níveis de abstração), compartilhamento de componentes (compartilha recursos mantendo o encapsulamento dos componentes), funções reconfiguráveis (implantar e configurar um sistema dinamicamente). Um componente Fractal é entendido como uma entidade executável que está encapsulada e tem identidade distinta [2, 13].

Com relação a sua estrutura, um componente Fractal é formado por duas partes [2, 13]: A membrana e o conteúdo. O conteúdo é composto de um conjunto finito de outros componentes, chamados de sub-componentes, que são gerenciados por um controlador anexado ao componente. Isto remete a idéia de hierarquia de componentes, tendo em vista que estes podem ser aninhados em qualquer nível arbitrário. O modelo Fractal, ainda distingue os componentes primitivos (que implementam serviços funcionais) e os componentes de composição que apenas servem para construir hierarquias de componentes, mas sem implementação de serviços.

Já em relação a comunicação, um componente pode interagir com seu ambiente através de *interfaces*. Como em outros modelos, estas *interfaces* podem ser clientes e servidoras. Uma *interface* servidora pode receber solicitações e retornar o resultado de uma determinada operação, enquanto que uma do tipo cliente emite chamadas à estas operações [30]. A interação entre estes componentes compreende a conexão de componentes primitivos e componentes de composição. A membrana é a parte responsável com funções de controle de um componente em particular. Ela pode ter *interfaces* externas e internas. *Interfaces* externas são acessíveis para outros componentes e permitem a reconfiguração de suas funções internas, enquanto que as *interfaces* internas são somente acessíveis aos sub-componentes de um componente [2, 13].

As comunicações entre componentes Fractal são realidadas de duas formas. A conexão básica é uma conexão entre uma *interface* cliente e uma *interface* servidora no mesmo espaço de endereçamento. Este tipo de conexão é chamado desta forma por estar, prontamente, implementada por ponteiros ou referências diretas [13]. Já uma conexão de composição envolve um número arbitrário de *interfaces*. Estas conexões são desenvolvidas de formas conhecidas como *stubs/skeletons* etc. Uma conexão é caracterizada por um componente Fractal que tem a função de gerenciar a comunicação entre outros componentes. O conceito de conexão em Fractal está relacionada com o conceito de conectores, pertencentes a outros modelos [13].

3.3.1 Julia

O principal objetivo do projeto Julia, foi implementar uma plataforma para a programação de membranas de componentes Fractal. Com isso, oferecer um conjunto extensível de objetos de controle, de modo que o usuário possa escolher livremente e montar os objetos controladores e interceptores, resultando no seu

próprio componente Fractal. Outro objetivo com o desenvolvimento do Julia foi proporcionar uma contínua transição entre a configuração estática e a configuração dinâmica. Visa a construção de sistemas de *software* com componentes escritos em Java [2, 13].

3.3.2 ProActive

Outro projeto envolvendo a construção de uma plataforma baseada em Fractal é o ProActive [15]. ProActive é visto como uma contribuição para o problema do reuso de *software*, integração e implantação para a computação paralela e distribuída. Ele propõe conceitos de programação, metodologias e uma plataforma para atender a natureza hierárquica, altamente heterogênea e distribuída da computação em grades, mas envolve também sistemas embarcados, computação ubíqua e internet, onde fatores como alto desempenho, alta disponibilidade e facilidade de uso são muito importantes [2, 26].

Além disso, ProActive não requer nenhuma mudança no padrão de Java de execução e não faz uso de compiladores especiais ou máquina virtual modificada. Como já mencionadao, ProActive aplica o padrão RMI em vez do tradicional padrão de passagem de mensagem. A justificativa é o esforço para alcançar o reuso de código [2, 26].

3.4 GCM

O GCM (*Grid Component Model*) é um modelo de componentes voltado para o desenvolvimento de aplicações para grades computacionais, que utiliza o modelo Fractal como base. Entre as principais características deste modelo pode-se citar: Abordagem da questão da implantação de componentes distribuídos. Eles necessitam ser implantados em diversos sistemas, que estão dispersos e são heterogêneos.

O modelo GCM oferece, além disso, um novo paradigma de composição coletiva de componentes. As aplicações em grades consistem em um conjunto de componentes que podem ser compreendidos como um grupo. A partir disso, a comunicação (um para muitos, muitos para um e muitos para muitos) entre eles pode acontecer de uma forma estruturada e de alto nível através das *interfaces multicast* e *gathercast*.

Outro aspecto que este modelo aborda é o suporte aos componentes autônomos, capazes de se auto adaptar a ambientes e requisitos em constante evolução. A idéia proposta é adotar a abordagem orientada a componentes para realizar a parte de

controle de um componente, diferentemente do que é feito em Fractal onde essa parte é realizada no modo orientado a objetos. Então, a parte de controle sendo realizada por um sub-componente concede mais dinamicidade ao modelo e aumenta a capacidade de adaptação dos componentes [62].

3.5 CCA

O modelo CCA [8], (*Common Component Architecture*), é uma especificação de um modelo de componentes para aplicações científicas de alto desempenho. Esta especificação está enfatizada na compreensão de como utilizar e implementar melhor a corrente prática do desenvolvimento de *software* baseado em componentes na área de computação científica de alto desempenho [18]. Além de definir a especificação, o esforço de desenvolvimento origina implementações que podem ajudar outros cientistas a desenvolver suas aplicações de acordo com o modelo [1, 20].

O CCA Forum [10], que compreende vários laboratórios do DOE (*U.S. Departament of Energy*) [9] e Universidades dos EUA, tem desenvolvido o CCA com o propósito de ganhar produtividade no desenvolvimento de *software* científico de alto desempenho e de alta qualidade de uma forma simples e natural do ponto de vista do cientista desenvolvedor.

O modelo CCA possui muitas semelhanças com modelos de componentes mais conhecidos e, também, requisitos de *softwares* científicos. Isto foi intencional e visa uma fácil adaptação de novos usuários de desenvolvedores. Alto desempenho e facilidade de uso são apectos de alta relevância no CCA. Além do mais, não existe nenhuma barreira tão forte que venha a impedir o desenvolvimento de plataformas para computação de alto desempenho baseados em modelos comerciais comuns ou a criação de mecanismos que possibilitem a integração entre componentes CCA e componentes baseados em outros modelos.

Os objetivos específicos que têm guiado o desenvolvimento do CCA compreendem aspectos como a heterogeneidade, ou seja, a possibilidade de compor aplicações executando em arquiteturas e linguagens distintas. A integração é outro aspecto importante, pois a maioria das simulações existentes foram desenvolvidas em linguagens como C ou Fortran. Então, o modelo deve suportar a integração dos códigos existentes sem modificar o padrão de programação utilizado originalmente. Acrescenta-se a isto, o fato de que o processo de configuração de componentes sob o modelo CCA (criação e conexão de componentes) não deve atribuir sobrecargas extras às aplicações. E para acompanhar as demandas, sempre variáveis, um modelo

de componentes deve ser bastante flexível, podendo se adequar aos requisitos de aplicações de determinados domínios. Com isso, o principal objetivo é alcançar a confiança de quem vai usar o modelo, permitindo facilidades de uso e reuso [1,18,20].

3.5.1 Conceitos importantes em CCA

Ainda não existe um conceito de componentes padrão na ciência da computação. Como já citado, a definição de componentes hoje é particular a um certo modelo. As diferenças existentes nesses conceitos ocorrem, essencialmente, no projeto e na execução dos componentes, mas o princípio fundamental dos componentes é servir como unidades independentes de *software* que podem ser combinadas na formação de aplicações [27]. No modelo CCA, estes componentes se comunicam através de *interfaces* chamadas portas. Elas podem ser compreendidas, de forma mais prática, como uma classe, uma coleção de subrotinas em Fortran ou estruturas em C. Existem dois tipos de portas. Portas *provides* são *interfaces* pelas quais os componentes disponibilizam determinadas funcionalidades. Portas *uses* são *interfaces* pelas quais os componentes requisitam determinadas funcionalidades [18,27]. A responsabilidade de gerenciar a criação e as interações entre componentes é atribuida a uma plataforma, que é uma aplicação que serve como base para a construção de aplicações com componentes CCA [27].

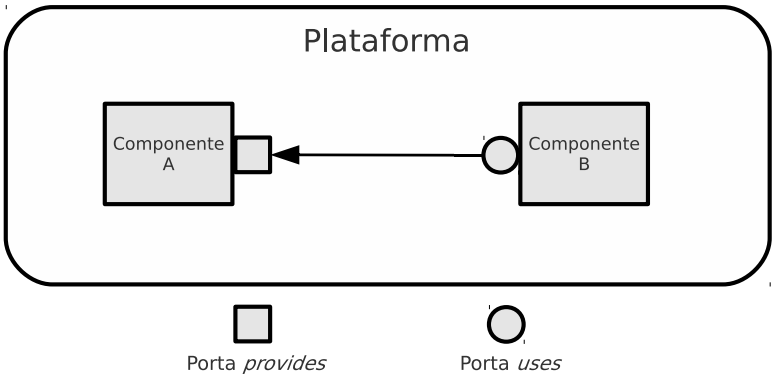

Figura 3.2: *Conexão simples entre componentes em CCA.*

Na Figura 3.2, observa-se uma interação simples entre componentes no modelo CCA. O componente B, através de sua porta *uses*, se comunica com o componente A, que disponibiliza alguma implementação através de sua porta *provides*. A porta *uses* se traduz por uma variável do componente B, por exemplo `portB`, cujo tipo é da *inteface* `Interface C`. A porta *provides* se traduz em algum mecanismo do componente A que permita a execução de cada método pertencente à `Interface C`.

A conexão liga as chamadas do tipo `portB.method()` à sua execução no componente A. O caso mais simples é conexão direta. Nesse caso, a porta *provides* é um objeto de uma classe que implemente a `Interface C`. A conexão é uma simples atribuição desse objeto à variável `portB` do componente B. Outras formas de conexão, mais complexas, são possíveis. Outro fato importante é que a conexão é feita em tempo de execução, após a instanciação do componente A e do componente B.

3.5.2 Interoperabilidade entre linguagens

O modelo CCA aborda esta questão da interoperabilidade através de uma SIDL *(Scientific Interface Definition Language)*. Ela é utilizada para descrever chamadas a *interfaces* de componentes. A SIDL aborda todas as características das IDLs encontradas em outros modelos, como a IDL da Microsoft para o COM [35] e a IDL do CORBA [29], além de incorporar os principais requisitos da computação científica como o suporte a números complexos e matrizes multidimensionais, bem como diretivas de comunicação que são importantes para componentes paralelos e distribuídos etc [18, 43].

Algumas outras tentativas foram feitas para contornar este problema, mas, neste contexto, a ferramenta considerada mais importante se chama Babel [43]. É uma implementação da SIDL que tem o propósito de abordar esta questão da interoperabilidade entre linguagens e reuso de *software* científico de alto desempenho reduzindo os custos computacionais deste processo. Adiciona suporte a orientação a objetos como a interoperabilidade entre C++, Java e Python, bem como com linguagens procedurais como Fortran e C. Com isso, habilita incorporação de componentes escritos nas linguagens mais usadas pela comunidade científica e escalabilidade para, possivelmente, suportar outras linguagens [1, 18, 43]

3.5.3 Componentes paralelos e distribuídos

O modelo CCA não impõe novos mecanismos para o desenvolvimento de aplicações paralelas. Sua abordagem principal é a integração. Esta proposta conserva o modelo de programação paralela tradicional. Com isso, os desenvolvedores podem continuar a implementar aplicações paralelas com seu paradigma preferido. Com relação ao desempenho dos componentes paralelos, este é considerado, virtualmente, idêntico ao da implementação original [1, 18]. Outro fato é que componentes paralelos podem envolver comunicações internas a eles.

A computação distribuída também é abordada no modelo CCA em virtude do crescimento das aplicações que oferecem serviços e das grades computacionais.

Na computação distribuída, a ênfase é na habilidade de integração de padrões e plataformas existentes. Uma aplicação pode ser composta por componentes paralelos e distribuídos. A integração de paradigmas de computação de alto desempenho paralela e distribuída é um campo de pesquisa interessante e desafiador [18,21]. Em relação ao modelo CCA, existe o aspecto de integração, em uma mesma plataforma, de protocolos de comunicação distintos. A coexistência, por exemplo, de conexões locais e remotas em uma mesma aplicação se torna interessante pelo fato de podermos utilizar tanto memória compartilhada quanto distribuída para o caso de *clusters multithreaded*. A implementação destas comunicações deve ser realizada com o máximo de atenção para a questão do desempenho, principalmente se tratando das conexões paralelas.

A computação distribuída levanta um número de questões que diferem da abordagem paralela [8, 18]. Entre elas está a comunicação entre componentes remotos. Se estiverem na mesma rede (LAN), a comunicação é comumente realizada através de paradigmas de passagem de mensagem como o MPI [4]. Entretanto, se estiverem em redes dispersas geograficamente, por exemplo, os mecanismos mais explorados são a chamada de procedimento remoto (RPC) ou a invocação de método remoto (RMI) [7]. Estas comunicações podem ser clandestinas (que ocorrem à margem da plataforma). Os componentes podem, também, efetuar comunicações não clandestinas. Este tipo exige uma intermediação, que pode ocorrer nas portas ou com uso de outros componentes intermediários.

3.5.4 Plataformas baseadas no modelo CCA

A seguir serão brevemente descritos algumas plataformas que implementam o modelo CCA. Nelas pode-se observar características interessantes do ponto de vista da forma de interação entre os componentes.

A abordagem oferecida pela **Ccaffeine** [24] é o suporte ao paradigma SPMD (*Simple Program Multiple Data*), que se caracteriza pelo fato de os processos que compõem um programa paralelo executarem as mesmas instruções, mas com dados diferentes. Ela possui uma parte implementada em C++ e sua *interface* gráfica com o usuário implementada em Java. Instâncias de componentes SPMD existem em processos comuns em todos os processadores participantes de uma execução. A rede de conexões é idêntica em cada processador. Por exemplo, uma porta pode ser utilizada em todos os processadores. Em CCAffeine, todas as conexões são feitas diretamente através da passagem de portas (ponteiros para uma função virtual) de

acordo com a especificação CCA. Utiliza passagem de mensagem para comunicações em memória distribuída (redes locais).

Ccaffeine é estruturada de forma que cada processador possua sua própria instância da plataforma. Este pode ser visto como um *container* que detém outros componentes. A visão destas instâncias SPMD indica a direção da integração de aplicações paralelas de larga escala.

XCAT [25] é uma plataforma CCA de componentes distribuídos que utiliza o modelo de *web services* como base da arquitetura. Possui implementações em C++ e Java. Cada porta *provides* em um componente XCAT é descrita usando um esquema XML (*eXtensible Markup Language*), sendo projetada como um *web service*. Os desenvolvedores da XCAT trabalham, frequentemente, usando documentos WSDL (*Web Services Description Language*) para este propósito. Estes documentos proporcionam uma descrição da *interface* e, também, disponibilizam informações pertinentes sobre os protocolos de comunicação para que serviços possam ser encontrados. O uso de WSDL pelo XCAT oferecerá a possibilidade de utilização dos mais conhecidos *client toolkits* para a invocação de métodos nas portas *provides* do XCAT [18].

SCIRun2 [23] é uma plataforma que combina a compatibilidade do modelo CCA com outros modelos de componentes comerciais. Utiliza RMI para conectar componentes em ambiente de memória distribuída. É *multithreaded* para facilitar a programação em memória compartilhada e, também, possui uma *interface* visual de programação. Sobretudo, esta plataforma oferece uma vasta abordagem, permitindo que cientistas possam combinar uma variedade de ferramentas para resolver um problema particular.

Aplicações nas áreas de combustão, geomagnetismo, álgebra linear esparsa [41, 45], química quântica [51], simulações nucleares [28], simulações envolvendo fusões [45], modelagem climática [50] têm adotado o modelo CCA. Outro exemplo importante que tem recebido mais atenção recentemente é a geração de qualidade de serviço computacional [52]. A motivação primária não é o reuso ou o compartilhamento de *software*. É aumentar a flexibilidade na exploração científica.

Esse capítulo apresenta uma breve descrição sobre alguns modelos e plataformas de componentes voltadas a aplicações científicas. Cada modelo descrito apresenta características importantes para o tratamento dos requisitos da computação e alto desempenho. Um destaque maior foi dado ao modelo CCA, pelo fato de ser um dos objetos de estudo desse trabalho. Com essa descrição pode-se observar que já

existem vários esforços na inserção de modelos e plataformas de componentes na computação de alto desempenho. Através das aplicações citadas como exemplo, percebe-se que a utilização dessa abordagem vem ganhando uma boa aceitação.

Continuando nessa linha, o próximo capítulo aborda uma análise feita sobre uma plataforma de componentes baseada em CCA chamada *Forro* e os mecanismos possíveis de integração com uma importante API de comunicação paralela, MPI. Esse procedimento exige muita atenção, pois as características que o MPI concede as aplicações não devem ser comprometidas.

Capítulo 4

A plataforma *Forro* e as aplicações MPI

Esse capítulo é destinado a mostrar algumas características importantes da plataforma *Forro* e que possuem relação com o escopo desse trabalho. Os aspectos envolvidos na integração do *Forro* com aplicações MPI serão descritos, bem como os mecanismos propostos para a comunicação entre componentes *Forro* utilizando o MPI.

4.1 Plataforma *Forro*

O *Forro* é uma plataforma de componentes, compatível com o modelo CCA, para aplicações paralelas e/ou distribuídas de alto desempenho. O leitor interessado em obter detalhes desta plataforma é aconselhado a consultar [68]. Nesse capítulo, são abordadas as características que dizem respeito mais diretamente ao escopo deste trabalho. Mais precisamente, levantamos questões que interferem nas conexões e que delimitam o espectro de solução de integração com MPI.

Além dos limites estabelecidos, detalhados de acordo com o modelo CCA na próxima seção, o principal aspecto da plataforma *Forro* que nos diz respeito é o fato de ela ser desenvolvida em Java. Não cabe nesse texto nos estendermos sobre os argumentos para tal escolha, mas esses estão ligados a características como: portabilidade, orientação a objetos e suporte a integração com códigos nativos. A importância dessas características ficará evidente à medida que suas principais propriedades forem sendo expostas.

Sendo uma plataforma voltada para aplicações científicas, o desempenho é um dos seus requisitos principais, no sentido abordado nos capítulos anteriores. O

fato de o *Forro* ser implementado em Java não se traduz na afirmativa de que o desempenho dessas aplicações seja comprometido, como visto adiante. Além disso, Java oferece características importantes às implementações de plataformas de computação distribuída. Adicionam-se as já citadas a segurança, a existência de APIs para comunicação que abstraem detalhes de configuração de aplicações e diminuem os esforços de utilização de recursos de rede, necessários em outras linguagens, uma possibilidade maior de reuso de código e a proximidade que possui com os conceitos de programação por componentes [7, 33, 57]. Estas características facilitam a realização de tarefas inerentes ao *Forro*, como instanciar componentes e efetivar conexões.

A maioria das aplicações científicas é desenvolvida em linguagens nativas como C/C++ ou Fortran. Com isso, é preciso que hajam estratégias para que elas possam ser incorporadas às plataformas de componentes. Para essas aplicações, o *Forro* oferece mecanismos para fazer com que o processamento seja realizado inteiramente em linguagem nativa. Toda a parte de configuração da aplicação continua sendo feita pela máquina virtual, mas a execução pode ser feita completamente na parte nativa.

Por meio da JNI (*Java Native Interface*) [57] as aplicações Java podem incorporar códigos escritos em linguagens nativas. Os métodos de uma classe Java podem ser declarados como nativos e acessados através de chamadas de função em uma biblioteca compartilhada (Ex.: *.so, *.dll). No *Forro*, aplicações escritas em linguagem nativa podem ser encapsuladas em componentes e continuar a realizar processamento na parte nativa.

Essa possibilidade de integração com códigos nativos faz surgir esforços para que as aplicações que usam o MPI possam ser inseridas em plataformas de componentes. Contudo, algumas questões tornam-se importantes neste contexto. Nesse caso, deve-se, sobretudo, conhecer como ocorre a conexão e a comunicação entre componentes CCA e, após isso, analisar quais os requisitos impostos pelo MPI para que essa integração possa acontecer.

4.2 Comunicação entre componentes em CCA

Essa é uma seção mais específica que abordará aspectos importantes relativos a comunicação entre componentes. Mostra como o modelo CCA especifica a comunicação, mediada pela plataforma de desenvolvimento subjacente, entre componentes locais ou remotos.

Tecnicamente, para que um componente se comunique com outros ele deve herdar a *interface* `gov.cca.Component` e implementar o método `setServices()`, que fazem parte da especificação CCA. Através deste método, um componente informa à plataforma a sua estrutura, ou seja, quais as funcionalidades que ele disponibiliza e as que ele requer [1, 20].

A Figura 4.1 mostra um exemplo de uma interação entre dois componentes CCA [1].

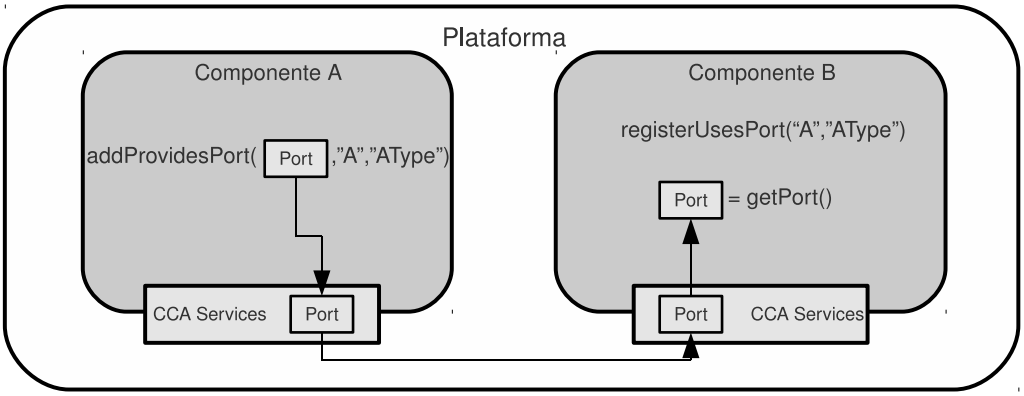

Figura 4.1: *Comunicação entre componentes no modelo CCA.*

Esta interação envolve passos como [27]:

- Um componente que precisa de uma porta registra sua necessidade à plataforma via invocação do método `registerUsesPort()` em seu objeto *Services*. Esta invocação especifica à plataforma o tipo da porta necessária e uma *string* que representa o nome único que o componente utiliza para identificar essa porta.

- Um componente que disponibiliza uma porta notifica à plataforma qual porta vai ser disponibilizada através da invocação do método `addProvidesPort()`, em seu objeto *Services*. Esta invocação especifica o tipo da porta que será disponibilizada, uma *string* referente ao nome único que o componente usa para identificá-la e uma referência ao próprio objeto.

- Estes componentes podem ser conectados de várias formas (a serem definidas pela plataforma subjacente). Outra função da plataforma é garantir que o tipo de porta solicitada e o tipo de porta disponibilizada sejam compatíveis.

- O componente que notificou sua necessidade à plataforma ganha o acesso à porta solicitada através da invocação do método `getPort()`, utilizando a

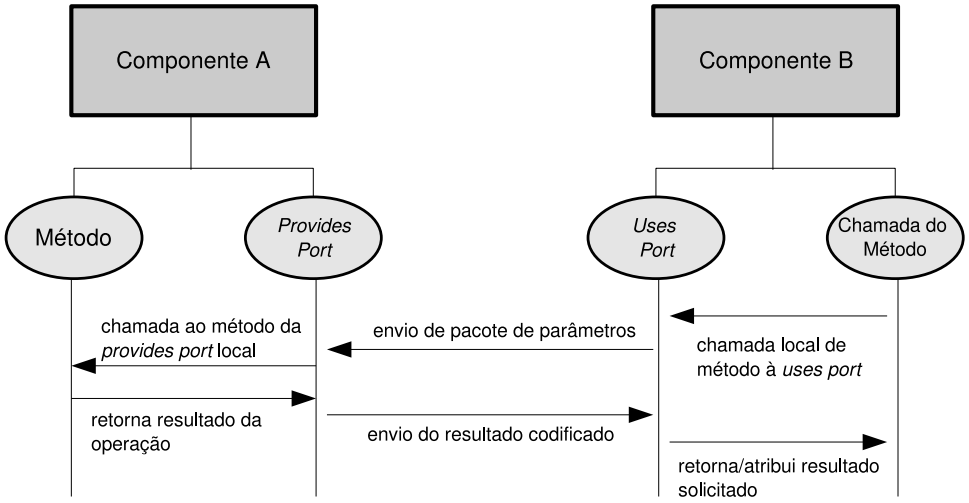

Figura 4.2: *Comunicação entre componentes remotos no modelo CCA.*

mesma *string* do passo 1.

Esta é uma descrição geral do processo de conexão entre componentes no modelo CCA [1]. No modo mais simples de conexão, as portas são ligadas diretamente. Esta é a chamada conexão local e é realizada pela plataforma. Interações entre componentes locais, que estão em um mesmo espaço de endereçamento, não devem possuir custo maior do que uma chamada a um método [9, 10]. Já para o caso da interação entre componentes remotos este mecanismo se torna inviável. Este tipo de comunicação envolve alguns outros aspectos em relação à comunicação entre componentes locais. A comunicação, como já citado, é mediada pela plataforma. Este realiza as chamadas a métodos remotos através de mecanismos como RMI, RPC ou passagem de mensagem [18].

Na Figura 4.2, pode-se observar uma outra forma de como poderia ocorrer a comunicação entre componentes. Com a conexão já estabelecida, os componentes interagem através de suas portas. O componente B realiza chamadas em sua porta *uses*, que foi conectada pela plataforma com a porta *provides* do componente A. No caso de conexão local, esta chamada será feita diretamente no respectivo método do componente A. No caso de conexão remota, esta chamada será efetuada, com o auxílio de RMI ou passagem de mensagem, que realizará a chamada ao método solicitado do componente A. Este tipo de comunicação possui custos adicionais óbvios em relação à local, como o mecanismo de *software* para auxiliar na chamada do método pela rede. Entretanto, são custos necessários para realizar chamadas a métodos de componentes remotos.

Como já mencionado, as operações de configuração de componentes realizadas pela plataforma, não devem interferir no desempenho das aplicações científicas. Ainda mais em relação às aplicações paralelas. O processo de criação e conexão de componentes não deve causar sobrecargas que comprometam o desempenho dessas aplicações. Ele é importante pelo fato de conceder facilidades na configuração e comunicação entre componentes que estejam geograficamente dispersos para formar uma aplicação. Contudo, o custo deste processo é mais sensível em aplicações paralelas e por isso deve ser minimizado para não comprometer a sua utilização.

4.3 Plataforma Forro em um ambiente MPI

4.3.1 Interação entre Forro e MPI

Uma aplicação MPI é formada por um conjunto finito de processos que são juntamente e cooperativamente inicializados e terminados. O MPI possui primitivas que contabilizam e identificam cada processo da aplicação, formando assim um grupo. Cada aplicação MPI pode ter mais de um grupo, mas devendo serem constituídos pelos processos inicialmente inicializados pelo MPI [32].

Diante disso, é importante que o *Forro* seja instanciando juntamente com cada processo que compõe a aplicação MPI, ou seja, instanciar o *Forro* a partir do MPI. Esse é um primeiro passo para diminuir possíveis custos nessa integração. Por outro lado, instanciar o *Forro* separadamente ao MPI dificulta a comunicação entre a máquina virtual e a aplicação MPI pelo fato de, nesse caso, estarem em processos diferentes.

Na Figura 4.3 está representado um cenário onde uma aplicação MPI é inicializada a partir do *Forro*. Nela observa-se um determinado número de máquinas onde o *Forro* e o MPI são inicializados. O *Forro* pode instanciar componentes em cada uma dessas máquinas. Neste exemplo, um componente é instanciado em cada uma das máquinas e a partir do processo criado para executar o componente da Máquina 1, por exemplo, uma aplicação MPI pode ser inicializada.

Observa-se, com isso, que o *Forro* e o MPI fazem parte do mesmo processo na máquina onde o *Forro* inicializou o MPI. Entretanto, no restante das máquinas o mesmo não ocorre, pois a inicialização do MPI gera a criação de novos processos, nas outras máquinas, que formarão a aplicação MPI. Devido a isso, o *Forro* e o MPI estarão em processos diferentes nas demais máquinas ocasionando custo adicional de comunicação entre eles. Outro fato importante é que mesmo com a instanciação do *Forro*, nas outras máquinas, juntamente com a inicialização do MPI, realizada pela

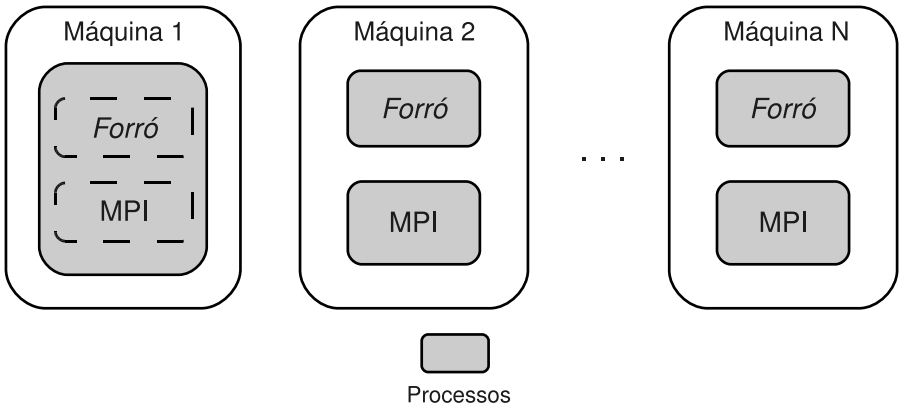

Figura 4.3: *Instanciar o Forro e MPI separadamente: em apenas uma máquina o Forro e o MPI fazem parte do mesmo processo. Nas demais, por estarem em processos distintos, a interação entre Forro e MPI produz custo adicional.*

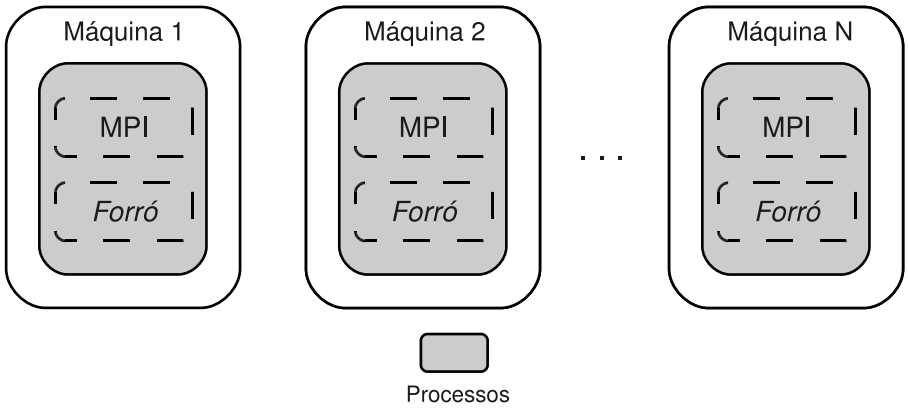

Figura 4.4: *Instanciar o Forro a partir do MPI: Forro e MPI no mesmo processo em todas as máquinas.*

instância do *Forro* na máquina 1, não evita que eles sejam executados em processos diferentes nas outras máquinas.

A Figura 4.4 representa a instanciação do *Forro* em um ambiente MPI de uma maneira mais eficiente. Nesse caso, o *Forro* é instanciado no momento da inicialização do MPI. Isso faz com que uma instância do *Forro* seja criada juntamente com cada processo que compõe a aplicação MPI, ou seja, *Forro* e MPI estarão no mesmo processo em todas as máquinas.

4.3.2 Encapsulamento em objetos

Outra questão que também deve ser analisada é o fato da adequabilidade requerida para que se possa utilizar o MPI em uma plataforma de componentes.

O padrão usado no MPI é o tradicional de chamada de funções, diferente do padrão de portas *provides/uses* especificado pelo modelo CCA. No *Forro*, a conexão direta é feita através da atribuição de um objeto a uma variável. Essa variável se traduz na porta *uses* e é declarada como o tipo de uma certa *interface* que possui os métodos necessários a uma certa execução. O componente faz a requisição à plataforma a partir desta porta. A porta *provides* é um objeto, uma implementação desta *interface*, que é disponibilizado pelo componente detentor desta porta. Então, o componente que fez a requisição receberá da plataforma este objeto e poderá fazer chamadas aos métodos requeridos.

Diante desse cenário, o padrão de programação tradicional usado com MPI deve ser um pouco modificado para ser inserido em um modelo de componentes. Entretanto, isto deve ser feito de forma a não comprometer o desempenho que o MPI pode oferecer as aplicações científicas. O *Forro* disponibiliza três modos para suportar à comunicação em suas conexões via MPI. Cada modo será descrito a seguir.

4.4 Mecanismos de suporte a comunicação entre componentes *Forro* via MPI

A possibilidade de chamadas a funções de linguagens nativas serem feitas a partir de métodos Java trás alguns benefícios importantes no que diz respeito à integração de aplicações em código nativo e o *Forro*, como já mencionado.

No caso da abordagem desse trabalho, deseja-se que os procedimentos de criação e conexão de componentes não interfiram na interação, por exemplo, entre processos de uma aplicação paralela que exige o máximo de desempenho possível do *hardware* subjacente. O modelo CCA especifica que não devem existir sobrecargas extras neste tipo de comunicação. A intenção é adicionar as características oferecidas pelo Java sem perder o desempenho original das aplicações que utilizam o MPI.

4.4.1 Comunicações clandestinas

O primeiro modo remete ao caso onde componentes se comunicam fazendo chamadas diretas ao MPI. Para isso, eles devem possuir um método com implementação nativa, que é a aplicação em si, fazendo chamadas ao MPI como `MPI_Send()` e o `MPI_Recv()`. Neste caso, ocorre uma sobrecarga mínima no momento em que o método com implementação nativa do componente é chamado para inicializar a aplicação. Durante a aplicação, não há sobrecargas provocadas

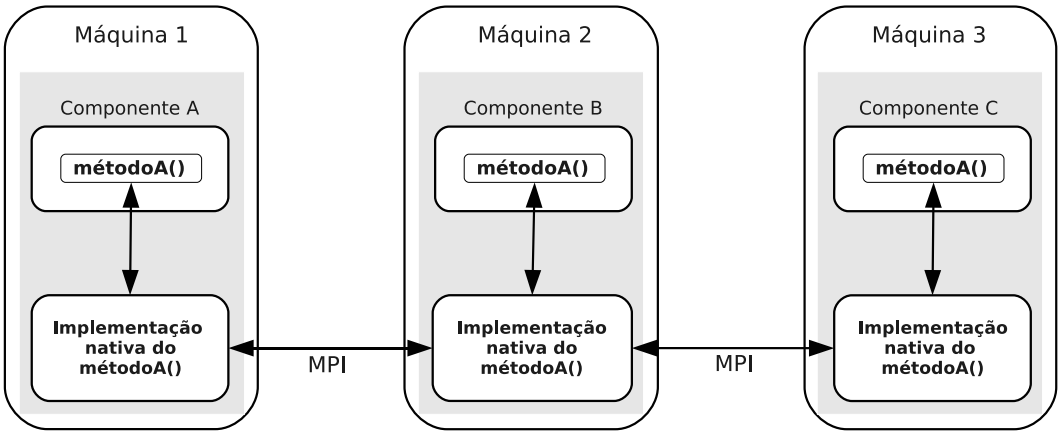

Figura 4.5: *Implementação de paralelismo na qual componentes fazem chamadas diretas ao MPI. Os componentes A, B e C são instanciados pelo Forro, cada qual com um método implementado em linguagem nativa. Esse método realiza alguma computação paralela.*

pelo *Forro*. A partir dele, o usuário criará e inicializará os componentes paralelos e a comunicação entre eles será realizada por chamadas a funções MPI feitas pela parte nativa dos componentes. Contudo, esta é uma comunicação que pode ser denominada de clandestina, pois não há conexão realizada pelo *Forro*.

Na Figura 4.5 observa-se um exemplo do que foi comentado no parágrafo anterior. Três componentes podem ser instanciados, cada um deles com um método que invoca uma implementação nativa. A partir disso, as comunicações podem ocorrer diretamente por meio do MPI, mas à margem do *Forro*.

4.4.2 Comunicações via conexões de portas

A outra possibilidade é a transformação de uma chamada MPI na invocação de um método em alguma porta. Nela estão inclusos o segundo e o terceiro modos. Essa possibilidade pode ser representada por meio de uma conexão típica entre uma porta *provides* e uma porta *uses* de um componente. A *Provides Port* encapsula, em seus métodos nativos (Java), as funções da biblioteca MPI (C/C++). A porta *uses* pertence a um componente da uma aplicação qualquer que fará chamadas aos métodos da *Provides Port*.

Nos exemplos das Figuras 4.6 e 4.7, as comunicações ocorrem através de conexões feitas dentro da máquina virtual. Diferentemente da representação das comunicações clandestinas, onde era necessário instanciar somente um componente por máquina que faz chamadas ao MPI diretamente na implementação nativa de algum de seus métodos, as comunicações via portas necessitam ter em cada máquina uma conexão

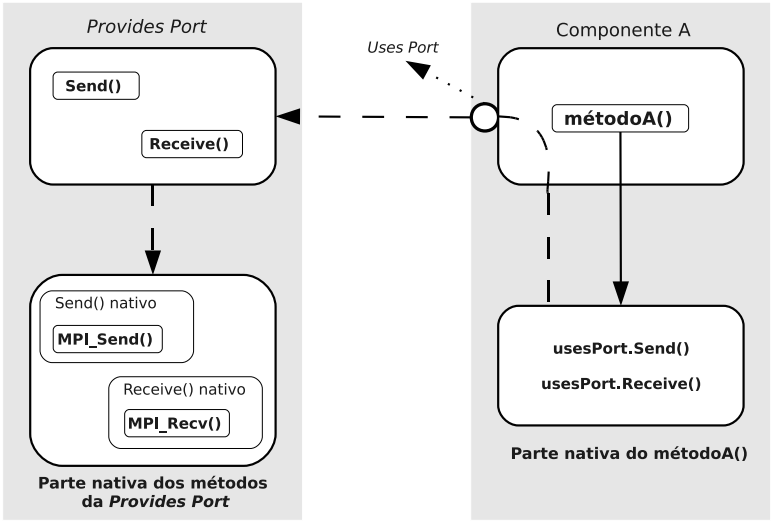

Figura 4.6: *Exemplo de chamadas MPI que se transformam em invocação de métodos.*

entre portas *provides/uses*. Esses cenários foram criados com o objetivo de explorar as possibilidades de integração entre Java e MPI e medir o quanto de custo está agregado nessas comunicações.

Na Figura 4.6 está representado o segundo modo de comunicação. Ele inicia logo após a conexão, realizada pela plataforma, entre a *Provides Port* e a porta *uses* do componente A (atribuição de um objeto do tipo *Provides Port* a uma variável no componente A, chamada *usesPort*, do tipo de uma *interface* que *Provides Port* implementa). O `métodoA()` desse componente corresponde a uma aplicação qualquer implementada em linguagem nativa. A seta preta contínua representa a interação do `métodoA()` com sua implementação nativa, ou seja a chamada do `métodoA()` e o ínicio da execução da aplicação na parte nativa. Mesmo com a passagem para o lado nativo, as chamadas oriundas do `métodoA()` à *Provides Port* ainda serão feitas por meio de um objeto Java, resultante do processo de conexão mencionado anteriormente.

Nesse caso, o código do `métodoA()` irá chamar métodos nativos da *Provides Port* e essa chamada passará pela máquina virtual antes. Por exemplo, a função `MPI_Send()` irá ser executada após a chamada ao método nativo `Send()` na parte Java da *Provides Port*. Além disso, outro tipo de sobrecarga ocorre no momento da chamada de método. Ela diz respeito a conversão de dados entre tipos nativos e tipos Java. Uma das características da integração entre Java é linguagens nativas é que na parte nativa é possível declarar variavéis com tipos de dados Java.

Um exemplo dessa conversão ocorre na chamada que parte do `métodoA()`, pois um método Java irá ser chamado. Para que a chamada seja feita corretamente os parâmetros do método que irá ser chamado devem ser convertidos para tipos de dados Java. Da mesma forma ocorre quando um método Java é chamado na *Provides Port* e sua implementação é nativa. Uma nova conversão ocorre, agora de tipos de dados Java para tipos de dados nativos, pois a execução do método irá ocorrer na parte nativa e com manipulação de dados com tipos nativos. Entretanto, nem todos os parâmetros precisam ser convertidos, pois a representação entre tipos de dados é a mesma. Mas para vetores ou matrizes, que frequentemente são usados em funções MPI como `MPI_Send()` e `MPI_Recv()`, essas conversões sempre irão acontecer pelo fato da representação de seus tipos ser diferente.

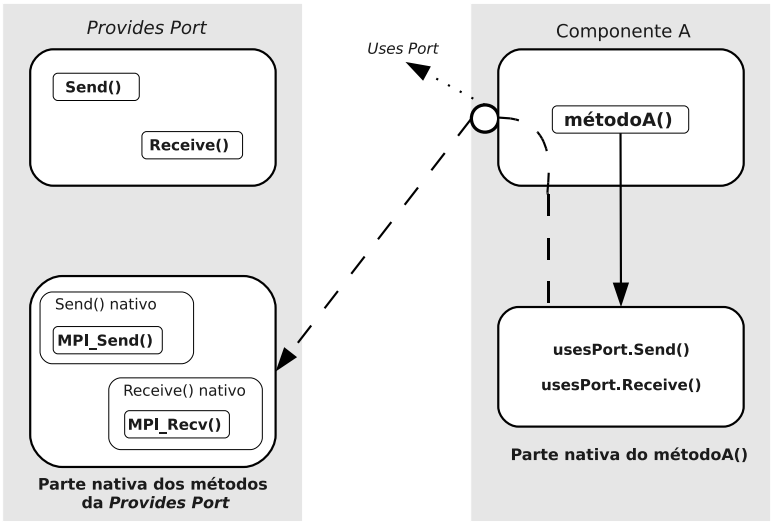

Figura 4.7: *Chamadas diretas às implementações nativas dos métodos da Provides Port.*

O terceiro modo é quase equivalente ao segundo. O que os diferencia é a possibilidade, que Java oferece, da realização de chamadas aos métodos da *Provides Port* diretamente em sua parte nativa. Nesse caso, o custo das comunicações tende a ser menor em virtude da não passagem das eventuais chamadas de método na parte Java da *Provides Port*. Por exemplo, uma chamada ao método `Send()` pode ser realizada diretamente em sua parte nativa, ou seja, na função nativa que implementa esse método. Na Figura 4.7 observa-se uma representação do terceiro modo de comunicação. As chamadas são efetuadas a partir do `métodoA()`, através da porta *uses*. Entretanto, as conversões de tipos ainda vão ocorrer nesse modo, pois a assinatura da função nativa que contém a implementação de um método da *Provides*

Port é a mesma em relação a este método na parte Java.

Como já mencionado, esses dois modos de comunicação via portas foram idealizados para que se pudesse explorar as formas de integração entre Java e MPI. Entretanto, a título de informação, o *Forro* possibilita que uma conexão seja realizada na parte nativa dos componentes, por meio da implementação de portas como objetos fora da máquina virtual, permitindo a comunicação direta em linguagem nativa. Com isso, a passagem das chamadas pela máquina virtual é retirada, mas o mecanismo que o *Forro* oferece para implementar portas é mantido.

Dadas as possibilidades e as afirmativas feitas sobre elas, o próximo passo é implementá-las e observar quais são os custos que essas soluções podem trazer. Para o segundo e terceiro casos é preciso saber o quanto de sobrecarga pode ser agregado a aplicação para saber se esse valor pode ser aceitável diante dos requistos das aplicações de alto desempenho. No capítulo seguinte serão apresentadas algumas experiências que servem como base para a descoberta de uma integração satisfatória entre Java, que é a linguagem em que o componentes estão implementados, e MPI, que é o paradigma mais usado em aplicações paralelas.

Capítulo 5

Processo de Implementação e Avaliação

5.1 Objetivo geral

Esse trabalho busca um mecanismo de integração entre componentes *Forro* e as aplicações MPI. Esse capítulo mostra uma avaliação experimental de soluções propostas a partir das possibilidades que o *Forro* oferece para encapsular aplicações MPI em componentes, mantendo as comunicações realizadas via MPI.

5.2 Descrição

As aplicações paralelas de alto desempenho, como já citado, possuem o desempenho como principal requisito. Principalmente as aplicações que têm dependêcia lógica entre seus processos. O paradigma dominante nestas aplicações é o SPMD, onde códigos idênticos são executados em todos os processos participantes da aplicação, que pode ser estendido ao MPMD (*Multiple Process Multiple Data*), onde tem-se códigos diferentes executando em todos os processos da aplicação.

O modelo CCA visa oferecer suporte tanto a programação paralela, quanto a programação distribuída, pelos motivos já apresentados. Com isto, pode-se integrar vários grupos de pesquisa ou instituições de forma colaborativa. A abordagem que o modelo CCA trás para suportar o paralelismo é a integração. Ou seja, os códigos de programas paralelos podem ser encapsulados por componentes CCA sem a ocorrência de modificação no modelo de programação usado no projeto original da aplicação. O objetivo é atrair muitos desenvolvedores de *software* que podem continuar usando seu modelo de programação favorito.

Com a possibilidade deste tipo de integração, o modelo CCA aborda o SCMD *Single Component Multiple Data*, que é a extensão lógica do paradigma SPMD. Nele, os componentes são entidades separadas e que devem ser instanciadas e configuradas identicamente em todos os processos participantes. Entretanto, após este processo de integração, o desempenho das aplicações paralelas pode ser comprometido. As plataformas criam os componentes e as ligações entre eles. Cada componente encapsula um tipo de código nativo. Com isto, pode-se ter uma sobrecarga indesejada nas interações entre os processos, desde que a interação entre os componentes não seja feita de forma eficiente. As operações internas da plataforma não devem adicionar sobrecargas extras na comunicação da aplicação encapsulada pelos componentes.

A partir disso, algumas experiências foram realizadas com o propósito de auxiliar o processo de integração entre o *Forro* e as aplicações MPI. Deseja-se observar com essas experiências os custos relativos a comunicação de componentes *Forro* (Java) que encapsulam códigos nativos.

5.3 Comunicação entre programas paralelos

Até o momento, foram estudadas e implementadas algumas soluções que integram programas que executam em uma máquina virtual e programas que executam fora da máquina virtual. Esse procedimento remete ao caso de programas SPMD. Os processos destes programas serão encapsulados em objetos Java, mas a comunicação entre eles poderá ser feita utilizando o paradigma de programação paralela original.

Dessa forma, tem-se a integração de programas que executam na máquina virtual Java e programas que executam fora dela. Com isso, a comunicação ainda será feita por MPI. Outro fator importante é a tentativa de obter uma boa *interface* entre Java e a biblioteca MPI.

A seguir serão apresentadas três experiências onde esse mecanismo foi utilizado e seus resultados.

5.3.1 Descrição das experiências

Os testes foram realizados *cluster* castanhao do Laboratório de PesquIsa em ComputAção (LIA), localizado no Departamento de Computação da UFC. No momento, não foi possível utilizar todos os nós do *cluster* em virtude do processo de atualização de seu sistema operacional e vários programas. Para essas experiências, foram disponibilizados dez nós nas redes *fast-ethernet* e *gigabit-ethernet*.

Três experiências foram realizadas:

- PingPong
- Operação coletiva *AlltoAll*
- Junção das operações coletivas *Broadcast* e *Reduce*

Cada umas delas é executada de forma que o tamanho da mensagem irá aumentando até chegar a um tamanho máximo. Vetores, declarados estaticamente, fazem o papel dessa mensagem que tem seu tamanho aumentado até um máximo. Para que certas anomalias sejam minimizadas, as experiências possuem 30 amostras para cada tamanho de mensagem. As amostras são necessárias para tentar evitar qualquer comportamento anômalo que uma execução, para um dado tamanho de mensagem, possa ter. Com esse número, pôde-se obter uma média satisfatória para cada tamanho de mensagem nas execuções. Entretanto, outros métodos de determinação deste número não podem ser descartados.

5.3.2 Modos de implementação

Nas experiências realizadas e que serão descritas a seguir foram utilizados 3 modos de implementação. Com isso, podemos observar melhor o desempenho computacional de diferentes implementações.

Estes modos foram implementados através da JNI (*Java Native Interface*), que possibilita que uma aplicação em Java seja integrada com aplicações escritas em código nativo. O modo 1 de implementação corresponde ao mecanismo representado na Figura 4.5, onde os componentes realizam comunicações clandestinas à plataforma. Os componentes simplesmente fazem chamadas a métodos com implementação nativa e as comunicações ocorrem via MPI diretamente.

O modo 2 se relaciona com o mecanismo representado na Figura 4.6 do capítulo 4. Neste caso, as chamadas MPI se transformam em invocação de métodos em uma porta. O modo 2 representará as comunicações com intervenção da máquina virtual. A conexão direta se traduz em uma atribuição de um objeto a uma variável, como já mencionado. Então foram implementados objetos Java para realizar esta "conexão". Assim, as chamadas serão efetuadas atráves de um objeto na máquina virtual.

O modo 3 representará as comunicações representadas na Figura 4.7. Este modo possui a mesma estrutura do modo 2. A diferença é que as chamadas aos métodos são feitas diretamente em suas implementações nativas. As chamadas ainda são feitas

através de um objeto Java, mas sem a passagem pelo lado Java do método chamado. Estas soluções foram propostas para saber o quanto de custo elas podem agregar nas comunicações e, com isso, auxiliar no processo de integração de componentes *Forro* e as aplicações MPI.

PingPong

A primeira experiência foi o PingPong. Nela, um vetor de inteiros é enviado de processo a processo com percurso de ida e volta. O tamanho do vetor vai sendo elevado até um tamanho máximo. Os processos não efetuam nenhuma computação. Com isso, o fator comunicação é melhor observado.

Resultados do PingPong

As Figuras a seguir mostram os resultados da execução do PingPong no cenário apresentado.

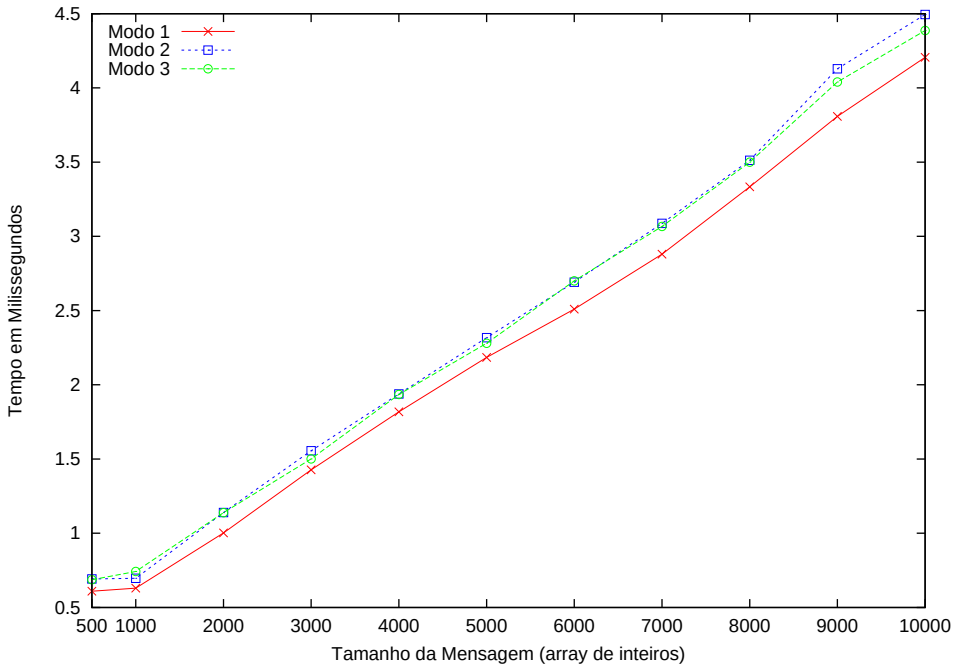

Figura 5.1: *PingPong na rede fast-ethernet.*

No gráfico da Figura 5.1, rede *fast-ethernet*, pode-se observar um comportamento com melhor desempenho da implementação no modo 1. Isso já era esperado pelo fato de não haver indireções nas chamadas as funções MPI. Já as implementações no modo 2 e no modo 3 apresentam um comportamento bem parecido com crescimentos constantes. A partir do tamanho 8000, a diferença entre eles aumenta um pouco. O

modo 3 obtem um melhor desempenho no final e adiciona 0,18ms (4,2%) em relação ao modo 1. Já o modo 2 adiciona 0,27ms (6,9%) em relação ao modo 1. Com estes tamanhos de mensagem já se pode perceber a interferência das indireções provocadas pelos modos 2 e 3, sendo que o modo 2 é mais prejudicado pela passagem no lado Java.

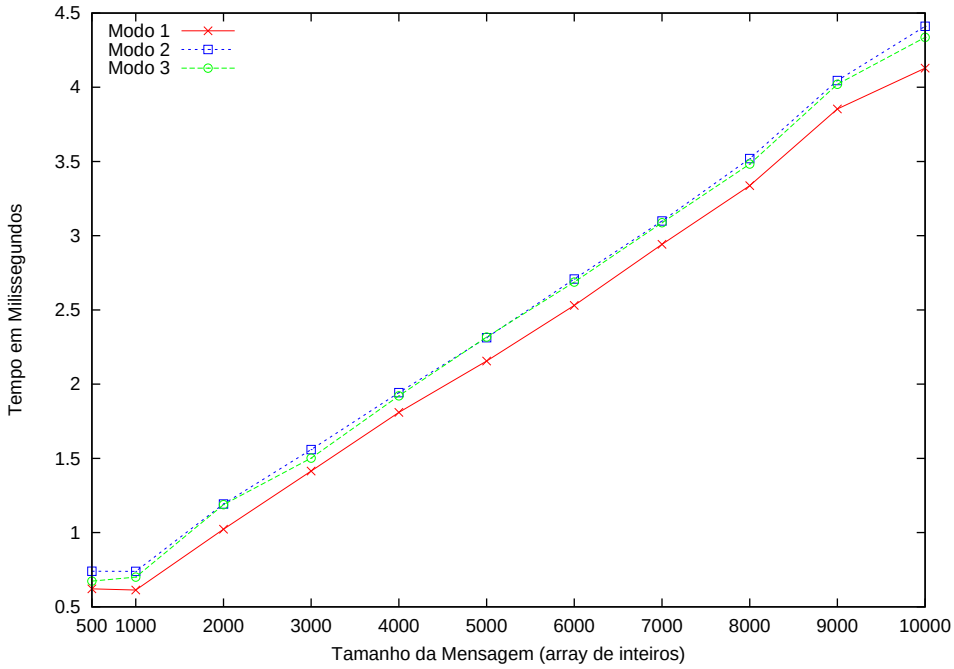

Figura 5.2: *PingPong na rede gigabit-ethernet.*

Na Figura 5.2 pode-se observar o gráfico relacionado as execuções do PingPong nos modos de execução apresentados na rede *gigabit-ethernet*. Coincidentemente, a implementação no modo 1 apresenta um melhor desempenho em relação as outras duas. As execuções nos modos 2 e 3 continuam a apresentar um comportamento parecido. A pequena diferença entre elas começa a surgir a partir do tamanho 7000. É no final onde ocorre a maior diferença entre elas. O modo 2 adiciona adiciona 0,29ms (6,79%) em relação ao modo 1. Já a execução no modo 3, adiciona 0,21ms (4,85%) em relação ao modo 1. Nesta rede pôde-se observar um melhor desempenho da execução no modo 1. Pôde-se observar também que a diferença entre as execuções nos modos 2 e 3 diminuiu, mas no geral, as indireções provocadas por elas ainda interfem em seu desempenho.

Operação coletiva *AlltoAll*

Nessa operação, cada processador envia uma mensagem distinta de tamanho m para todos os nós. Ela é, também, conhecida como operação de troca total de dados. Ela é usada em uma variedade de algoritmos paralelos como em transposição de matrizes, transformada de *Fourier* e operações de união em bancos de dados paralelos [55].

As experiências relativas a esta operação foram realizadas com 5 e com 10 nós do *cluster* com o objetivo de observar o que acontece com o comportamento das execuções dos modos de implementação com números diferentes de nós.

Resultados do *AlltoAll*

Na figura 5.3 pode-se observar a execução da operação *AlltoAll* na rede *fast-ethernet* com 5 nós. As execuções apresentam um comportamento bem parecido. Entretanto, desde o começo das execuções já se observa a interferência das indireções provocadas nos modos 2 e 3. A maior diferença entre elas duas é bem pequena (0,04ms). A execução do modo 2 adiciona 0.09ms (8,10%) em relação a execução do modo 1. Já a execução do modo 3 adiciona 0,05ms (3,95%) em relação a execução do modo 1.

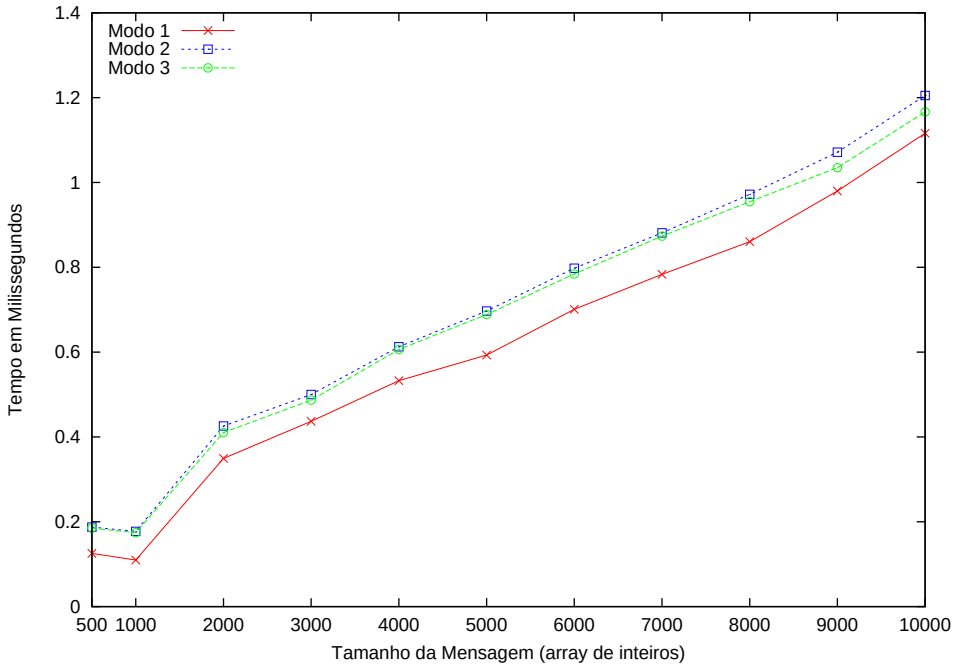

Figura 5.3: *Operação AlltoAll em 5 nós da rede fast-ethernet.*

A Figura 5.4 mostra o comportamento das execuções da mesma operação na rede *gigabit-ethernet*. A diferença no tempo total de execução dos modos de implementação nesta rede diminue quase pela metade. O comportamento das execuções do *AlltoAll* se assemelha ao da rede fast-ethernet, ou seja, desde o ínicio as execuções dos modos 2 e 3 já adiciona tempo em relação a execução do modo 1. Contudo, esta adição no tempo diminui em relação ao gráfico mostrado na Figura 5.3. A execução do modo 2 adiciona 0,05ms (8,10%) em relação a execução do modo 1. A execução do modo 3 adiciona 0,03ms (5,08%) em relação ao modo 1.

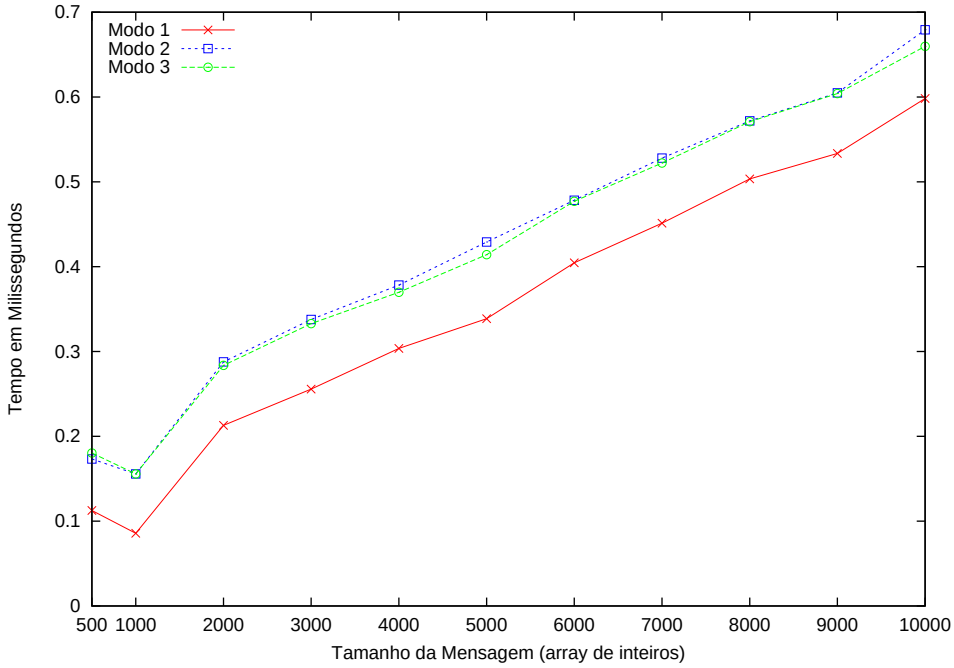

Figura 5.4: *Operação AlltoAll em 5 nós da rede gigabit-ethernet.*

Agora, serão mostrados os gráficos relativos às execuções da operação *AlltoAll* com 10 nós. O gráfico da Figura 5.5 mostra o comportamento das execuções dos modos de implementação na rede *fast-ethernet*. Elas apresentam um comportamento bem parecido durante toda a execução. A diferença entre os modos 2 e 3 em relação ao modo 1 são bem pequenas. A execução do modo 2 adiciona, em sua maior diferença, 0,022ms (6,34%) em relação ao modo 1. Já o modo 3, adiciona 0,163ms (4,5%). Visualmente a diferença entre as execuções diminui, mas tecnicamente continua seguindo um mesmo padrão, de acordo com o cálculo mostrado.

A Figura 5.6 mostra o comportamento das execuções na rede *gigabit-ethernet*. Os tempos totais de execução diminuem, mas não na mesma proporção do cenário

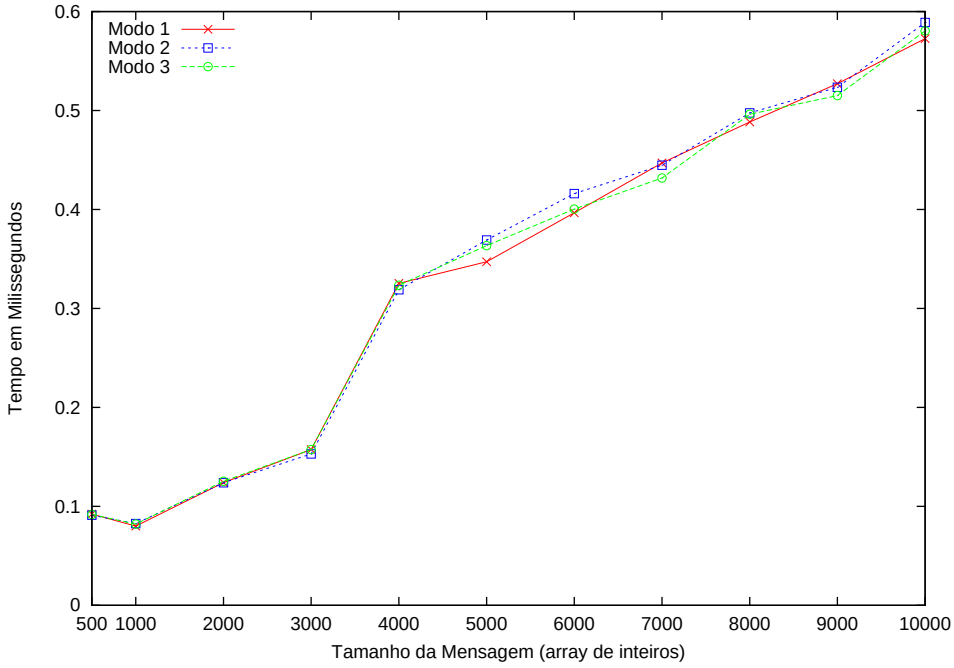

Figura 5.5: *Operação AlltoAll com 10 nós da rede fast-ethernet.*

mostrado com 5 nós. As execuções dos modos 2 e 3 mostram um comportamento bem semelhante, sendo que o modo 3 apresenta um melhor desempenho. A execução do modo 2 adiciona 0,0438ms (13,13%) em relação a execução do modo 1. A execução do modo 3 adiciona 0,387ms (10,6%).

O que se pode perceber, até o momento, é que quando os tempos das execuções são muito pequenos, uma pequena diferença entre os tempos das execuções já origina uma sobrecarga bem maior. Quando os tempos passam de 1ms, neste caso, as pequenas diferenças entre os tempos das execuções não geram sobrecargas tão maiores.

Com isso, pode-se concluir que os modos de implementação 2 e 3 podem adicionar pequenas sobrecargas em aplicações que possuem um tempo maior de execução.

Broadcast-Reduce

Essa experiência foi realizada utilizando duas funções de comunicação coletiva, assim como na experiência da operação *AlltoAll*, do MPI: O *Broadcast* (`MPI_Bcast()`) e o *Reduce* (`MPI_Reduce()`). O *Broadcast* envia a mesma mensagem para todos os processadores participantes da aplicação, enquanto que na operação *Reduce*, estas mensagens, de todos os processadores, são recolhidas pelo processador que chama esta função. Neste caso, o *Broadcast* envia o mesmo vetor para todos os

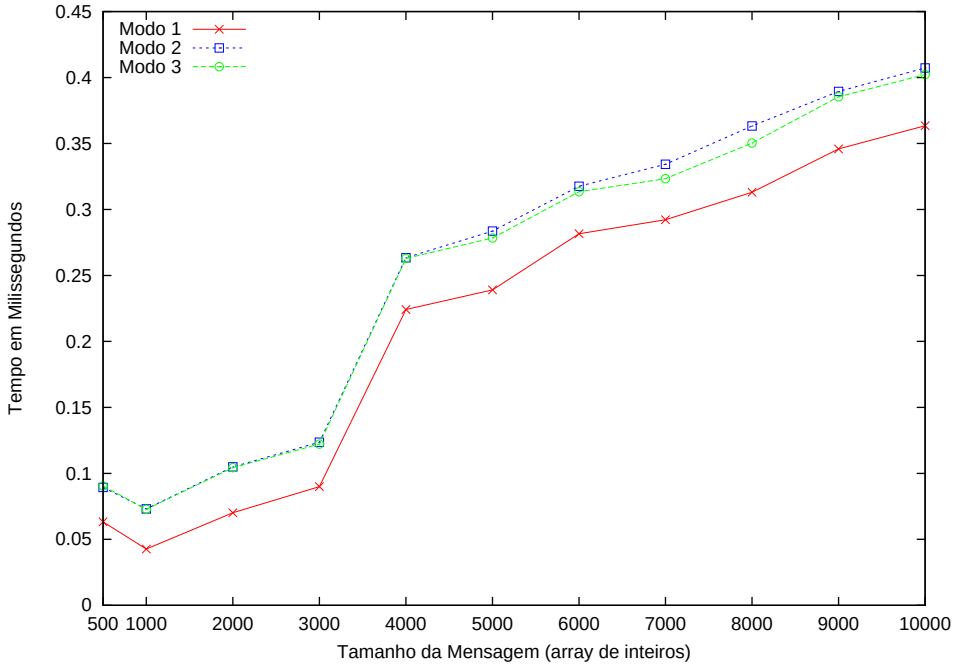

Figura 5.6: *Operação AlltoAll com 10 nós da rede gigabit-ethernet.*

processadores participantes e o *Reduce* recolhe estes vetores em um só vetor, através de alguma operação matemática ou lógica elementar realizada nos elementos dos vetores, como uma soma ou uma operação de maior que (>) ou menor que (<).

As experiências relativas a esta operação foram realizadas, também, com 5 e com 10 nós do *cluster* com o objetivo de observar o que acontece com o comportamento das execuções dos modos de implementação com números diferentes de nós.

Resultados do *Broadcast-Reduce*

Os gráficos das Figuras 5.7 e 5.8 mostram as execuções do *Broadcast-Reduce* em 5 nós das redes *fast* e *gigabit-ethernet*.

No gráfico da Figura 5.7, as execuções dos modos 2 e 3 apresentam uma grande diferença no início, em relação ao modo 1. Com o passar das execuções, a diferença vai diminuindo e os comportamentos vão se estabilizando. A maior diferença entre as execuções chega a 1,15ms e a menor diferença ocorre no final da execução (0,006513ms). Como houve o acontecimento de uma estabilidade no comportamento das três execuções, há a possibilidade de que a diferença entre os tempos das execuções não conservem a diferença dos tempos iniciais que são mostradas. Isto, pode fazer com que os modos 2 e 3 não adicionem grandes sobrecargas em relação ao modo 1.

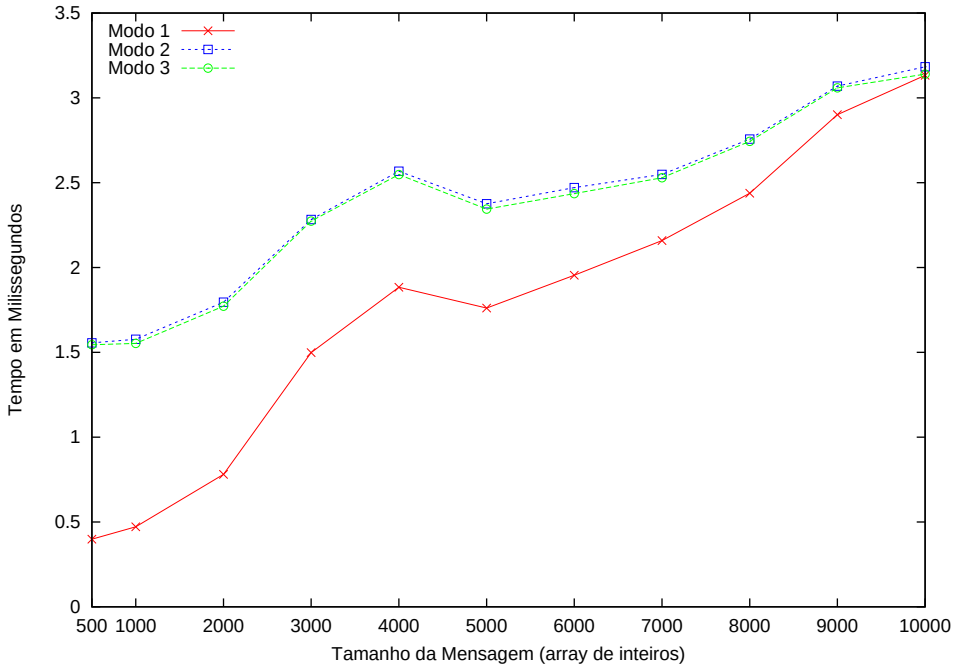

Figura 5.7: *Broadcast-Reduce em 5 nós da rede fast-ethernet.*

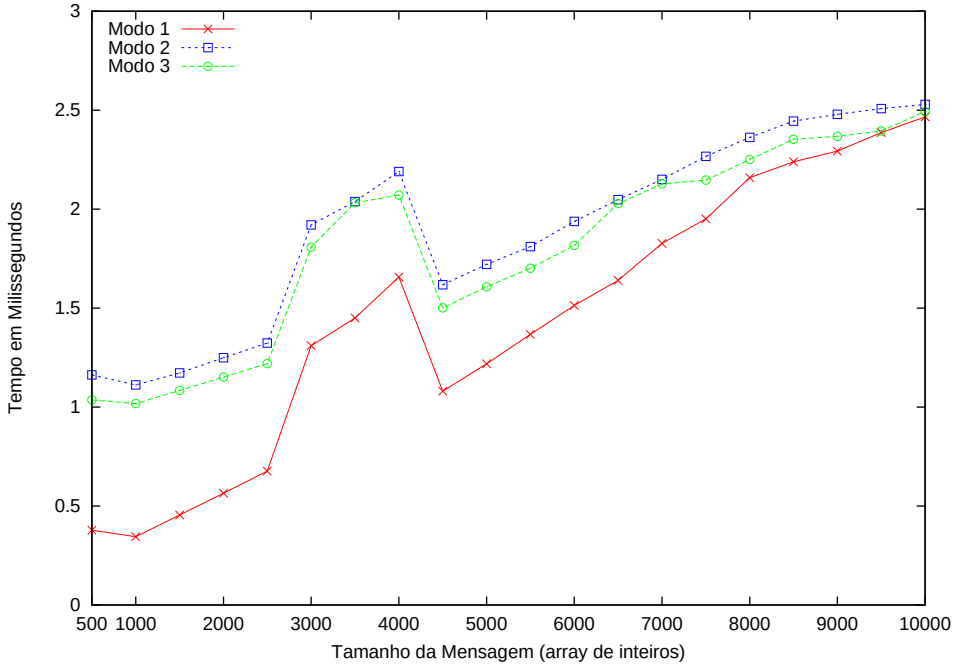

Figura 5.8: *Broadcast-Reduce em 5 nós da rede gigabit-ethernet.*

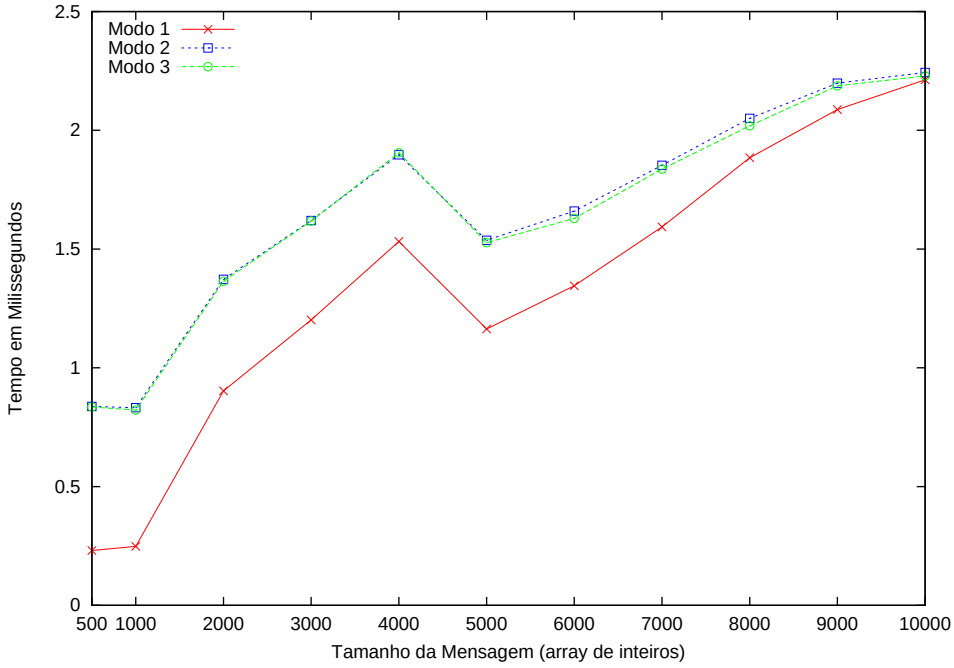

Figura 5.9: *Broadcast-Reduce em 10 nós da rede fast-ethernet.*

O gráfico da Figura 5.8, as execuções dos modos de implementação em 5 nós da rede *gigabit-ethernet* apresentam algumas semelhanças. O comportamento inicial das execuções mostram, também, a maior diferença entre elas, chegando a 0,79ms. Com o passar do tempo de duração das execuções, observa-se, também, uma estabilização em seus comportamentos. No final, a execução do modo 3 apresenta melhor desempenho em relação ao modo 2. Este adiciona 0,06ms (4,06%) em relação à execução do modo 1. Já a execução do modo 3, adiciona 0,03ms (2%) em relação à execução do modo 1. A mesma possibilidade de que as diferenças nos tempos não sejam tão grandes quanto às iniciais existe neste caso, como é mostrado no gráfico. Mesmo com o tamanho da mensagem aumentando, os comportamentos mostram uma estabilidade com passar o tempo.

Na execução do *Broadcast-Reduce* em 10 nós na rede *fast-ethernet*, gráfico da Figura 5.9, observa-se a preservação de uma característica mostrada nas execuções com 5 nós. No ínicio das execuções, as implementações nos modos 2 e 3 já apresentam uma grande diferença no tempo. Isso vai se estabilizando com o passar do tempo. Neste caso, a partir do tamanho 2000, as execuções dos modos 2 e 3 vão obtendo um comportamento mais estável. A maior diferença que a execução do modo 2 adiciona é de 0,37ms (31,9%). O modo 3 adiciona, em sua maior diferença, 0,35ms

(31,03%). Entretanto, no final das execuções, essas sobrecargas tem uma diminuição considerável. O modo 2 adiciona 0,0309ms (1,35%). Já o modo 3, adiciona 0,016ms (0,7%).

Com os resultados mostrados na maior diferença entre os tempos, não se pode descartar sumariamente os modos 2 e 3, visto que as execuções ainda não apresentam um comportamento uniforme e, também, pelo fato de que com um tamanho de mensagem maior a diferença entre os tempos tenha diminuido bastante. Supõe-se que a tendência, nesse caso, é que em execuções mais longas não haja uma variação deste tipo na diferença entre os tempos das execuções dos três modos.

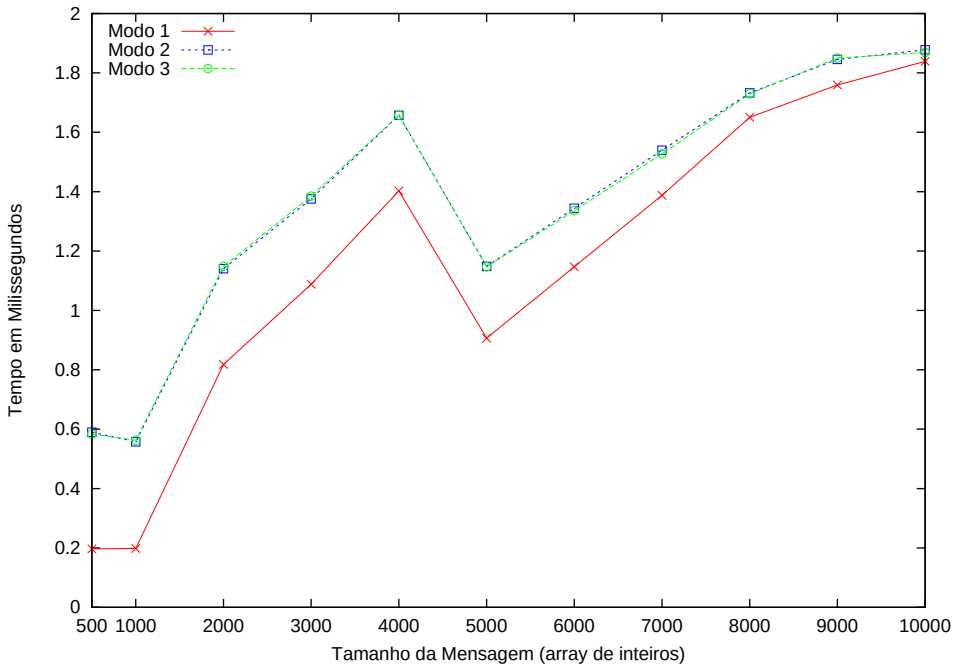

Figura 5.10: *Broadcast-Reduce em 10 nós da rede gigabit-ethernet.*

Na Figura 5.10 é mostrado o gráfico das execuções da mesma operação em 10 nós da rede *gigabit-ethernet*. O comportamento inicial dos modos 2 e 3 se assemelha aos comportamentos mostrados nos gráficos anteriores dessa operação. Além disso, eles tem comportamentos parecidos durante toda as suas execuções. A maior diferença entre os modos 2 e 3 em relação ao modo 1 é de 0,6367ms. Contudo, como ocorre no caso anterior, ao final das execuções a diferença diminui, fazendo com que o modo 2 adicione 0,41ms (32%). Já o modo 3 adiciona 0,03ms (1,64%) em relação ao modo 1.

A operação *Broadcast-Reduce* mostrou um comportamento diferente em relação

às outras experiências. Entretanto, se parece um pouco com a operação *AlltoAll* no fato de requerer uma duração maior das execuções para que se conseguisse uma maior estabilidade em seu comportamento. Mesmo com o tamanho de mensagem aumentando, os tempos nas execuções dos três modos não apresentavam uma variação constante. Isto so pôde ser possível de observar próximo ao final das execuções. Com isso, não se pode descartar as implementações dos modos 2 e 3 por uma grande diferença que pôde-se observar, principalmente na operação *Broadcast-Reduce*. Quando a variação dos tempos das execuções nos três modos se tornou constante, os modos 2 e 3, principalmente o modo 3, tiveram um desempenho próximo ao modo 1, que funciona sem provocar indireções.

No próximo capítulo, algumas considerações serão feitas sobre os modos de implementação propostos para que se possa observar seus pontos positivos e negativos.

Capítulo 6

Conclusões e Propostas de Trabalhos Futuros

Nesse capítulo serão mostradas as considerações sobre os mecanismos propostos por esse trabalho para auxiliar no processo de integração entre o *Forró* e as aplicações MPI. Após isso, serão mostradas algumas idéias para a realização de trabalhos futuros.

6.1 Conclusões

Os três modos de implementação propostos mostram que a integração Java/MPI pode ser bastante satisfatória. Eles foram implementados na tentativa de obter os valores que correspondem ao custo envolvido nas comunicações em programas paralelos encapsulados por objetos Java. Os valores mostram que o preço a ser pago por isso pode ser pequeno se a integração for bem realizada. Nos gráficos que mostram as execuções do PingPong e do *AlltoAll*, por exemplo, os modos 2 e 3 apresentam desempenhos com pequenas sobrecargas. O modo 3, principalmente, é que apresenta uma sobrecarga baixa.

As execuções do *Broadcast-Reduce* foram um pouco diferentes. De acordo com os gráficos, principalmente na parte final das execuções, a tendência é que essa operação precise de uma execução mais longa para que a variação entre os tempos das execuções dos três modos se estabilize e com isso se possa analisar o quanto de sobrecarga os modos 2 e 3 podem adicionar. Portanto, a diferença na variação entre os tempos das execuções nos três modos ocorrida no ínicio da operação não impossibilita, por si só, a utilização das implementações propostas. Quando essa variação se tornou constante, a análise foi feita e os modos 2 e 3 apresentaram um

bom desempenho em relação ao modo 1. Ou seja, o mecanismo proposto é viável, mas ainda pode ser melhorado.

Alguns aspectos, como esse que afetou mais a operação *Broadcast-Reduce*, não foram possíveis de serem realizados no cenário dessas experiências. Credita-se a isso, o fato dos vetores, o conteúdo variável das mensagens, terem sidos declarados de forma estática. Isso ocasionou um estouro na pilha dos processos quando a mensagem chegava a um certo tamanho, impossibilitando que a execução fosse mais longa e que conseguisse achar um ponto de convergência para que a análise fosse realizada de maneira mais contundente. Esse problema também ocorreu nas outras operações, já que em cada programa os vetores foram declarados de forma estática. Entretanto, com o cenários e as implementações propostas nas outras operações foi possível observar que esse tipo de integração pode ser satisfatória. Contudo, é um preço que pode ser pago em troca dos benefícios que a abordagem de componentes pode oferecer às aplicações científicas tradicionais.

6.2 Propostas de trabalhos futuros

Algumas questões que podem ser abordadas futuramente, são:

- Revisar as implementações nos modos 2 e 3, com o objetivo de melhorá-las para que possam obter sobrecargas cada vez menores.
- Repetir as experiências utilizando alocação dinâmica na declaração dos vetores que compõem as mensagens que serão trocadas entre os processos. Com isso, o objetivo é permitir que as execuções das experiências se tornem mais longas, que mensagens maiores possam ser trocadas e que se tenha a possibilidade de encontrar um ponto de convergência para cada uma das execuções.
- Integrar os modos de implementação propostos aos componentes *Forró*.
- Criar e executar componentes que encapsulem as principais bibliotecas para solução de aplicações científicas, como PETSc [68].
- No processo de configuração, realizado pelo *Forró*, estuda-se a possibilidade de inserir um canal dedicado por onde ocorrerá a interação entre os componentes. É uma característica que concede mais dinamicidade na comunicação entre componentes. Contudo, assim como apresentado na avaliação das soluções propostas nesse trabalho, a adição de qualquer nova funcionalidade ao

Forró não deve comprometer o desempenho das aplicações que realizam a comunicação via MPI.

Referências Bibliográficas

[1] 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*, vol. 18(2), pp. 215–229, 2006.

[2] Baude, F.; Caromel, D.; and Morel, M. From Distributed Objects to Hierarchical Grid Components. *Lecture Notes in Computer Science*, vol. 2888, pp. 1226–1242, 2003.

[3] 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)*. IEEE Computer Society, pp. 31. Washington, DC, USA, 1997.

[4] MPI. http://www-unix.mcs.anl.gov/mpi/. Acesso em Junho de 2009.

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

[6] Pérez, C.; Priol, T.; 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*, IEEE Computer Society, pp. 90274.1, Washington, DC, USA, 2004.

[7] Java RMI. http://java.sun.com/javase/technologies/core/basic/rmi. Acesso em Junho de 2009.

[8] The Common Component Architecture Technical Specification. http://www.extreme.indiana.edu/xcat/cca.html. Acesso em Junho de 2009.

[9] The DOE Common Component Architecture Project. http://www.extreme.indiana.edu/~gannon/cca_report.html. Acesso em Junho de 2009.

[10] The Common Component Architecture Forum. http://www.cca-forum.org/overview/index.html. Acesso em Junho de 2009.

[11] Wang, A. J. A.; Qian, K. Component-Oriented Programming. LNCS, Wiley Inter-Science. 2005.

[12] Szyperski, C. Component Software and the Way Ahead. pp. 1–20, 2000.

[13] Bruneton, E.; Coupaye, T.; Leclercq, M.; Quewewma, V.; Stefani, J-B. An Open Component Model and Its Support in Java. *Lecture Notes in Computer Science*. 2004.

[14] Leavens, G. T.; Sitaraman, M. Foundations of Component Based Systems. Cambridge University Press, 2000.

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

[16] Mathias, E.; Baude, F.; Cave, V.; Maillard, N. A Component-Oriented Support for Hierarchical MPI Programming on Multi-Cluster Grid Environments. *In SBACPAD*. IEEE Computer Society, pp. 135–142, Los Alamitos, CA, USA, 2007.

[17] Breivold, H. P.; Larsson, M. Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles. *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on.* pp. 13–20, 2007.

[18] Allan, B. A.; Armstrong, R.; Bernholdt, D. E.; Bertrand, F.; Chiu, K.; Dahlgren, T. L.; Damevski, K.; Elwasif, W. R.; Epperly, T. G. W.; Govindaraju, M.; Katz, D. S.; Kohl, J. A.; Krishnan, M.; Kumfert, G.; Larson, J. W.; Lefantzi, S.; Lewis, M. J.; Malony, A. D.; Mclnnes, L. C.; Nieplocha, J.; Norris, B.; Parker, S. G.; Ray, J.; Shende, S.; Windus, T. L.; Zhou, S. A Component Architecture for High-Performance Scientific Computing. *In Int. J. High Perform. Comput. Appl. Sage Publications, Inc.*, pp. 163–202, Thousand Oaks, CA, USA, 2006.

[19] Goscinski, W.; Abramson, D. Motor: A Virtual Machine for High Performance Computing. *In High Performance Distributed Computing, 2006 15th IEEE International Symposium on.* pp. 171–182.

[20] Armstrong, R.; Gannon, D.; Geist, A.; Keahey, K.; Kohn, S.; McInnes, L.; Parker, S.; 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.* IEEE Computer Society, pp. 13, Washington, DC, USA, 1999.

[21] Bramley, R.; Chiu, K.; Diwan, S.; Gannon, D.; Govindaraju, M.; Mukhi, N.; Temko, B.; Yechuri, M. A Component Based Services Architecture for Building Distributed Applications. *In HPDC '00: Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing.* IEEE Computer Society, pp. 51, Washington, DC, USA 2000.

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

[23] Zhang, K.; Damevski, K.; Venkatachalapathy, V.; Parker, S.G. SCIRun2: a CCA framework for high performance computing. *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on.* pp. 72–79.

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

[25] Govindaraju, M.; Krishnan, S.; Chiu, K.; Slominski, A.; Gannon, D.; Bramley, R. Merging the CCA Component Model with the OGSI Framework. *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid.* IEEE Computer Society, pp. 182. Washington, DC, USA, 2003.

[26] Baduel, L.; Baude, F.; Caromel, D. Asynchronous Typed Object Group for Grid Programming. *In International Journal of Parallel Programming.* vol. 35(6), 2007.

[27] Kumfert, G. Understanding the CCA Standard Through Decaf. *Lawrence Livermore National Lab.*, CA (US), 2003.

[28] Post, D. E.; Votta, L. G. Computational Science Demands a New Paradigm. *In Physics Today Journal*, vol. 58, pp. 35-41, AIP, 2005.

[29] CORBA. http://www.omg.org/technology/documents/formal/components.htm. Acesso em Junho de 2009.

[30] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., and Stefani, J. B. The FRACTAL Component Model and its Support in Java. *Softw, Pract. Exper.*, vol. 36, pp. 1257-1284, 2006.

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

[32] Snir, M.; Otto, S.; Huss-Lederman, S.; Walker, D.; Dongarra, J. MPI: The Complete Reference. vol. 1, MIT Press, 1998.

[33] Enterprise JavaBeans. Enterprise JavaBeans downloads and specifications. http://java.sun.com/products/ejb/docs.html. Acesso em Junho de 2009.

[34] OMG. CORBA component model. http://www.omg.org/technology/documents /formal/components.htm. Acesso em Junho de 2009.

[35] COM. Component Object Model specification. http://www.microsoft.com /com/resources/comdocs.asp. Acesso em Junho de 2009.

[36] Judd, G; Clement, M.; Snell, Q.; et al. Design issues for efficient implementation of MPI in Java. *In Proceedings of the ACM 1999 Java Grande Conference*. 1999.

[37] Morin, S.; Koren, I.; Krishna, C. JMPI: Implementing the message passing standard in Java. *In International Parallel and Distributed Symposium: IPDPS 2002 Workshops*. 2002.

[38] Mintchev, S. Writing programs in JavaMPI. *In Technical Report MAN-CSPE-02*. 1997.

[39] Baker, M.; Carpenter, B.; Fox, G.; et al. mpiJava: An Object Oriented Java interface to MPI. *In International Workshop on Java for Parallel and Distributed Computing*. 1999.

[40] Willcock, J.; Lumsdaine, A.; Robinson, A. Using MPI with C# and the Common Language Infrastructure. *In Concurrency and Computation: Prac. and Experience*. 2005.

[41] Kumfert, G.; Bernholdt, D. E.; Epperly T. G. W.; Kohl, J. A.; McInnes, L. C.; Parker, S.; Ray, J. How the common component architecture advances computational science. *In Journal of Physics: Conference Series.* vol. 46, pp. 479–493, 2006.

[42] Carvalho Junior, F. H.; Lins, R. D.; Corrêa, R. C.; Araújo, G. A. Towards an architecture for component-oriented parallel programming. *In Concurr. Comput. : Pract. Exper.* vol. 19(5), pp. 697-719, Chichester, UK, 2006.

[43] Epperly, T.; Kohn, S. R.; Kumfert, G. Component Technology for High-Performance Scientific Simulation Software. *In Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software.* pp. 69-86, Deventer, The Netherlands, 2001.

[44] Alexeev, Y.; Allan, B. A.; Armstrong, R. C.; Bernholdt, D. E.; Dahlgren, T. L.; Gannon, D.; Janssen, C. L.; Kenny, J. P.; Krishnan, M.; Kohl, J. A.; Kumfert, G.; McInnes, L. C.; Nieplocha, J.; Parker, S. G.; Rasmussen, C.; Windus, T. L. Component-based software for high-performance scientific computing. *In Journal of Physics: Conference Series.* vol. 16, pp. 536-540, 2005.

[45] Batchelor, D. B.; Berry, L. A.; Houlberg, W. A.; Jaeger, E. F.; Bernholdt, D. E.; Elwasif, W. R.; D'Azevedo, E. F.; Kohl, J. A.; Li, S. Applying Component Technology to Coupled Fusion Simulations. *In CompFrame 2005 Workshop on Component Models and Frameworks in High Performance Computing.* 22-23 June, Atlanta, Georgia, USA, 2005.

[46] Fayad, M. E.; Schmidt, D. C.; Johnson, R. E. Building application frameworks: object-oriented foundations of framework design. John Wiley & Sons, New York, NY, USA, 1999.

[47] Top 500 Supercomputing Sites. http://www.top500.org/. Acesso em Junho de 2009.

[48] SWIG homepage. http://www.swig.org. Acesso em Outubro/2008.

[49] Classic Interface Definition. http://www.cca-forum.org/bindings/classic. Acesso em Junho de 2009.

[50] Zhou, S. J. Coupling Climate Models with the Earth System Modeling Framework and the Common Component Architecture. *In Concurr. Comput. : Pract. Exper.*, vol. 18(2). pp. 203–213, Chichester, UK, 2006.

[51] Kenny, J. P.; Benson, S. J.; Alexeev, Y.; Sarich, J.; Janssen, C. L.; McInnes, L. C.; Krishnan, M.; Nieplocha, J.; Jurrus, E.; Fahlstrom, C.; Windus, T. L. Component-Based Integration of Chemistry and Optimization Software. *In Journal of Computational Chemistry.* vol. 25(14). pp. 1717-1725. Mathematics and Computer Science Division, Argonne National Laboratory, 2004.

[52] McInnes, L. C.; Ray, J.; Armstrong, R.; Dahlgren, T. L.; Malony, A.; Norris, B.; Shende, S.; Kenny, J. P.; Steensland, J. Computational Quality of Service for Scientific CCA Applications: Composition, Substitution, and Reconfiguration. *Mathematics and Computer Science Division, Argonne National Laboratory*, February 2006.

[53] Norris, B.; Balay, S.; Benson, S.; Freitag, L.; Hovl, P.; McInnes, L.; Smith, B. Parallel Components for PDEs and Optimization: Some issues and experiences, Parallel Computing. *Mathematics and Computer Science Division, Argonne National Laboratory*, vol. 28, pp. 1811-1831, 2002

[54] Rene, C.; Alleon, G.; Priol, T. Code coupling using parallel CORBA objects. *In Proceedings of the International Federation for Information Processing Working Conference on Software Architectures for Scientific Computing*, Kluwer, Ottowa, 105-118, 2000.

[55] Grama, A.; Karypis, G.; Kumar, V.; Gupta, A. Introduction to Parallel Computing. 2 ed. Addison Wesley, 2003.

[56] Rasmussen, C. E.; Lindlan, K. A.; Mohr, B.; Striegnitz, J.; Jlich, F. CHASM: Static analysis and automatic code generation for improved Fortran 90 and C++ interoperability. *In Proceedings of the Los Alamos Computer Science Symposium 2001 (LACSI?01)*, 2001.

[57] Java Native Interface: Programmer's Guide and Specification. http://java.sun.com/docs/books/jni/. Acesso em Julho de 2009.

[58] Chapman, B. The Challenge of Providing a High-Level Programming Model for High-Performance Computing. High-Performance Computing, pp. 21-49. John Wiley & Sons, Inc. 2006.

[59] HPF. High Performance Fortran. http://hpff.rice.edu/index.htm. Acesso em Junho de 2009.

[60] OpenMP. The OpenMP API specification for parallel programming. http://openmp.org/wp/. Acesso em Junho de 2009.

[61] Global Arrays. The GA Toolkit. http://www.emsl.pnl.gov/docs/global/. Acesso em Junho de 2009.

[62] Baude, F.; Caromel, D.; Dalmasso, C.; Danelutto, M.; Getov, V.; Henrio, L.; Pérez, C. GCM: A Grid Extension to Fractal for Autonomous Distributed Components. *In Annals of Telecommunications*. 2008.

[63] Chan, F.; Cao, J.; Guo, M. ClusterGOP: A High-Level Programming Environment for Clusters. High-Performance Computing, pp. 1-19. John Wiley & Sons, Inc. 2006.

[64] Mono. http://www.mono-project.com/Main_Page. Acesso em Junho de 2009.

[65] WeiQin, T.; Hua, Y.; WenSheng, Y. PJMPI: pure Java implementation of MPI. *In High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on.* vol. 1, pp. 533-535, 2000.

[66] the Java Grande Forum. http://www.javagrande.org/. Acesso em Julho de 2009.

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

[68] Araújo, G. A. Uma Nova Plataforma CCA para Aplicações de Alto Desempenho usando Conectores Exógenos. Tese de Doutorado - Departamento de Computação - Universidade Federal do Ceará. 2010.

[69] PETSc: Home Page. *Portable, Extensible Toolkit for Scientific Computation.* http://www.mcs.anl.gov/petsc/petsc-as/. Acesso em Julho de 2009.