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

Integração de Bibliotecas Científicas de Propósito Especial em uma Plataforma de Componentes Paralelos

Autor
Davi Morais Ferreira

Orientador
Prof. Dr. Francisco Heron de Carvalho Junior

Fortaleza – Ceará
Novembro 2010

Davi Morais Ferreira

Integração de Bibliotecas Científicas de Propósito Especial em uma Plataforma de Componentes Paralelos

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.

Orientador: Prof. Dr. Francisco Heron de Carvalho Junior

Fortaleza – Ceará
Novembro 2010

Resumo

A contribuição das tradicionais bibliotecas científicas mostra-se consolidada na construção de aplicações de alto desempenho. No entanto, tal artefato de desenvolvimento possui algumas limitações de integração, de produtividade em aplicações de larga escala e de flexibilidade para mudanças no contexto do problema. Por outro lado, a tecnologia de desenvolvimento baseada em componentes, recentemente proposta como alternativa viável para a arquitetura de aplicações de Computação de Alto Desempenho (CAD), tem fornecido meios para superar esses desafios. Vemos assim, que as bibliotecas científicas e a programação orientada a componentes são técnicas complementares na melhoria do processo de desenvolvimento de aplicações modernas de CAD.

Dessa forma, este trabalho tem por objetivo propor um método sistemático para integração de bibliotecas científicas sobre a plataforma de componentes paralelos HPE (*Hash Programming Environment*), buscando oferecer os aspectos vantajosos complementares do uso de componentes e de bibliotecas científicas aos desenvolvedores de programas paralelos que implementam aplicações de alto desempenho. A proposta deste trabalho vai além da construção de um simples encapsulamento da biblioteca em um componente, visa proporcionar ao uso das bibliotecas científicas os benefícios de integração, de produtividade em aplicações de larga escala e da flexibilidade para mudanças no contexto do problema.

Como forma de exemplificar e validar o método, temos incorporado bibliotecas de resolução de sistemas lineares ao HPE, elegendo três representantes signifcativos: PETSc, Hypre e SuperLU.

Palavras-chave: Computação de Alto Desempenho, Biblioteca científicas, Componentes de softwares, Álgebra linear.

Abstract

The contribution of traditional scientific libraries shows to be consolidated in the construction of high-performance applications. However, such an artifact of development possesses some limitations in integration, productivity in large-scale applications, and flexibility for changes in the context of the problem. On the other hand, the development technology based on components recently proposed a viable alternative for the architecture of High-Performance Computing (HPC) applications, which has provided a means to overcome these challenges. Thus we see that the scientific libraries and programming orientated at components are complementary techniques in the improvement of the development process of modern HPC applications.

Accordingly, this work aims to propose a systematic method for the integration of scientific libraries on a platform of parallel components, HPE (Hash Programming Environment), to offer additional advantageous aspects for the use of components and scientific libraries to developers of parallel programs that implement high-performance applications. The purpose of this work goes beyond the construction of a simple encapsulation of the library in a component; it aims to provide the benefits in integration, productivity in large-scale applications, and the flexibility for changes in the context of a problem in the use of scientific libraries.

As a way to illustrate and validate the method, we have incorporated the libraries of linear systems solvers to HPE, electing three significant representatives: PETSc, Hypre, e SuperLU.

Keywords: High-Performance Computing, Scientific Libraries, Software Components, Linear Algebra.

Dedicatória

Dedico este trabalho a Deus, minha herança. Dedico também aos meus pais José Deusimar e Maria Rejane, e às minhas irmãs Renata e Simone.

Agradecimentos

Em primeiro lugar elevo meu agradecimento a Deus, a quem dedico as primícias de tudo o que tenho e sou. A Ele, fonte de toda a minha força e motivação, em especial nos momentos onde os ventos sopravam contra, deixo meu maior agradecimento.

Imprimo também meu humilde agradecimento aos meus pais, José Deusimar e Maria Rejane, que amorosamente e gratuitamente ofertaram suas vidas para ajudar na construção de quem sou hoje, sem eles não seria o que sou.

Agradeço também, a presença e a amizade das minhas irmãs, Renata e Simone, que me tornam ainda mais feliz e realizado.

Reservo espaço também, para agradecer ao meu orientador e professor Francisco Heron de Carvalho Junior. Que com empenho e dedicação muito contribuiu para o enriquecimento e qualidade desse trabalho.

Meu agradecimento também, ao restante do corpo docente da UFC, por contribuírem na minha formação acadêmica e aos funcionários da UFC pelo solícito auxílio.

À Bruna Thalyta registro meu agradecimento por sua companhia e carinho durante este tempo.

Aos colegas de curso, a quem cito com estimo Cedma Santos, por enfrentar comigo as adversidades do mestrado, e Fernanda Lígia, pelas conversas cheias de motivação para a conclusão do nosso curso.

À FUNCAP pelo apoio financeiro.

Por fim, deixo meu sincero agradecimento a todos aqueles que contribuíram de forma direta ou indireta para a finalização desse trabalho e para a realização de mais um passo da minha vida.

"Aos olhos humanos parecia cumprir uma pena, mas minha esperança estava cheia de imortalidade."

Sabedoria 3,4

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Na história das aplicações em ciências e engenharia não é difícil perceber uma busca incessante por soluções computacionais de instâncias cada vez maiores de problemas a um custo menor de tempo e dinheiro. No entanto, ainda existem instâncias relevantes de problemas importantes que mesmo a um custo muito alto continuam sendo impraticáveis. A questão econômica relacionada à solução de tais problemas motiva a exploração e desenvolvimento de técnicas de CAD (Computação de Alto Desempenho), as quais integram *hardware*, *software*, linguagens e técnicas de programação que tornam possíveis aplicações cujos requisitos computacionais são inatingíveis sem o emprego de plataformas especiais de computação.

As técnicas de CAD podem ser aplicadas nas mais diversas áreas do conhecimento. Porém, é tradicionalmente associada àquelas oriundas das ciências computacionais e engenharias. A maior parte dos interessados nessas aplicações são instituições acadêmicas e indústrias de grande porte. Entende-se por uma aplicação de CAD qualquer aplicação cuja demanda por recursos computacionais, nas dimensões tempo e/ou espaço, excedem a capacidade de sistemas de computação (*hardware* e *software*) de uso comum. Para melhor mensurarmos a dificuldade de se conceber tais aplicações, segue uma breve descrição de algumas aplicações recentes:

Simulação do cérebro do gato A IBM em conjunto com a Universidade de Stanford e o Laboratório Nacional Lawrence Berkeley desenvolveram uma aplicação para simular o cérebro do gato, contendo 1 bilhão de neurônios. Através das simulações dessa aplicação espera-se obter um maior conhecimento a respeito do funcionamento do cérebro e avaliar como a estrutura do cérebro afeta a sua função. A simulação foi efetuada em um supercomputador Blue

Gene com 147.456 processadores. [79]

Simulação de tornados em alta resolução Ming Xue, da Universidade de Oklahoma, utilizando o supercomputador Ranger do TACC[1] (*Texas Advanced Computing Center*), conseguiu simular, em 2010, vários antigos tornados com uma fidelidade nunca antes obtida. O objetivo de Ming Xue era fazer uma análise mais detalhada dos tornados, para entender como e porque eles se formam. [89]

Modelagem do pulmão Ching-Long Lin e Eric Hoffman (professores da Universidade de Iowa), juntamente com Merryn Tawhai (Universidade de Auckland), lideram um projeto que visa avaliar algumas questões de doenças pulmonares, como a asma. Essa avaliação é feita através de simulações realizadas nos supercomputadores Lonestar e Ranger do TACC, onde um pulmão humano é modelado com ajuda de imagens de tomografias e, por fim, se utiliza DFC (Dinâmica dos fluidos computacional) para simular o fluxo de ar dentro do pulmão. [91]

Não é difícil perceber a alta complexidade dessas aplicações, fato que se repete em diversos outros problemas atuais. Para resolvê-las, é preciso não só conhecer algoritmos adequados, mas também dispor de uma grande capacidade computacional. Para isso, a arquitetura de processamento paralelo tem se mostrado mais adequada para a execução de aplicações de CAD do que a arquitetura de processamento sequencial [13].

1.1 Arquiteturas de Processamento Paralelo

A primeira maneira de se explorar o paralelismo em arquiteturas de computadores foi através dos processadores superescalares, onde o processador utiliza vários *pipelines* independentes para se conseguir um paralelismo no nível de instruções. A Control Data Corporation, em 1964, construiu a primeira máquina com esse recurso, o CDC 6600. Após o CDC 6600, vieram o CDC 7600, o IBM 360/91, entre outros. Atualmente, arquiteturas superescalares dominam o projeto de processadores.

Posteriormente aos processadores superescalares, surgiram os processadores vetoriais, baseados na arquitetura SIMD(*Single Instruction, Multiple Data*). Tais

[1] http://www.tacc.utexas.edu/

computadores podem realizar operações em vetores através de uma única instrução. Em 1976, a Cray Research[2], através de seu fundador Seymour Cray, criou o Cray I. Essa máquina tinha capacidade de realizar 160 milhões de operações de ponto flutuante por segundo e foi considerada o primeiro supercomputador, por superar o desempenho dos computadores da época em ordens de magnitude. Ainda na década de 80, surgiram outras máquinas vetoriais de outras companhias como a Fujitsu, Hitachi e NEC, as quais iniciaram uma forte competição pela construção de supercomputadores cada vez mais rápidos, a fim de atender especialmente as demandas do governo dos EUA e seus países aliados na contenda da Guerra Fria.

Apesar do considerável poder de processamento, tais máquinas não eram escaláveis e sua fabricação era extremamente onerosa, assim como sua aquisição e manutenção. O próprio Seymour Cray afirmou em 1974: "Em todas as máquinas que já projetei, o custo foi sempre um aspecto secundário. Importava descobrir como construí-las o mais rapidamente possível, ignorando completamente os custos". Dessa forma, foram investidos US$8,8 milhões no projeto, construção e implantação do primeiro Cray I no LANL (Los Alamos National Laboratory), valor alto mesmo para os padrões atuais. Foram fabricadas 16 máquinas Cray I, e estas custavam em média US$700 mil. Enquanto isso, outras indústrias de computadores desenvolviam processadores cada vez mais sofisticados para estações de trabalho, realizando um investimento em pesquisa e desenvolvimento substancialmente maior do que empresas como CRAY e NEC investiam no projeto e construção de novos supercomputadores. Isso era possível pela maior escala de mercado de estações de trabalho, comparado ao mercado de supercomputadores, que permitia melhor amortização dos custos de P&D. Por tudo isso, foi motivado o uso de processadores desenvolvidos para estações de trabalho em CAD, dando origem a arquitetura de supercomputadores de processamento paralelo massivo, conhecida como MPP (*Massive Parallel Processing*).

A arquitetura MPP surgiu por volta da década de 80, oferecendo computação paralela através de um grande número de processadores de estações de trabalho, cada qual com a sua memória física local, ligados por uma interconexão de alto desempenho de tecnologia proprietária. Tais processadores são independentes entre si, possuindo poucos recursos compartilhados, tornando assim o sistema mais escalável. Em MPP's, o controle do paralelismo é feito no próprio código da aplicação, o que torna sua programação mais complexa. Pertencentes a essa

[2]http://www.cray.com/

Tabela 1.1: Arquitetura dos 500 mais rápidos supercomputadores

Arquitetura	Quantidade	Porcentagem
Constelação	2	0.40%
MPP	74	14.80%
Cluster	424	84.80%

arquitetura temos o Cray XT, SGI Altix, NEC Vector, IBM pSeries e Hitachi SR16000. Uma outra forma semelhante de se obter computação paralela foi o uso de estações de trabalho interconectadas por rede de computadores convencional, abordagem que ficou conhecida como NOW (*Network of Workstations*). Embora essa arquitetura induza um custo menor que aquele de MPP's, a alta latência na comunicação é um fator crítico que compromete o desempenho global do sistema. No entanto, presta-se a viabilizar várias aplicações onde o requisito de comunicação não é crítico.

Em meados da década de 90, surgiu uma nova arquitetura de computadores paralelos distribuídos, os quais ficaram conhecidos como *Clusters*. Semelhante à arquitetura NOW, os *Clusters* originalmente eram formados por computadores pessoais (hoje dito *desktops*) dedicados ao processamento paralelo, interconectados por redes locais. *Clusters* possuem máquinas otimizadas para o trabalho paralelo, onde o sistema operacional é aprimorado para sistemas distribuídos e os protocolos de rede são implantados direto no *hardware*. Dessa forma, os *Clusters* aliam baixo custo, baixa latência na comunicação e boa escalabilidade. Somando a isso, o advento de sistemas operacionais gratuitos e livres reduziram os custos da pesquisa e desenvolvimento de aplicações para *Clusters*. Existem várias classes de *Clusters*, como a classe Beowulf[3], criado por Donald Becker da NASA.

Como era de se esperar, devido ao baixo custo e ao alto poder de processamento, a maior parte dentre os 500 mais rápidos supercomputadores do mundo são *Clusters*. Como podemos constatar na Figura 1.1, já dominam 84.20% da classificação Top500[4].

Como uma aparente evolução dos *Clusters*, surgiu o modelo de computação em Grade, que por meio de uma infraestrutura de computação, agregando recursos computacionais heterogêneos, oferece ao usuário acesso transparente aos recursos

[3]http://www.beowulf.org/
[4]http://www.top500.org/

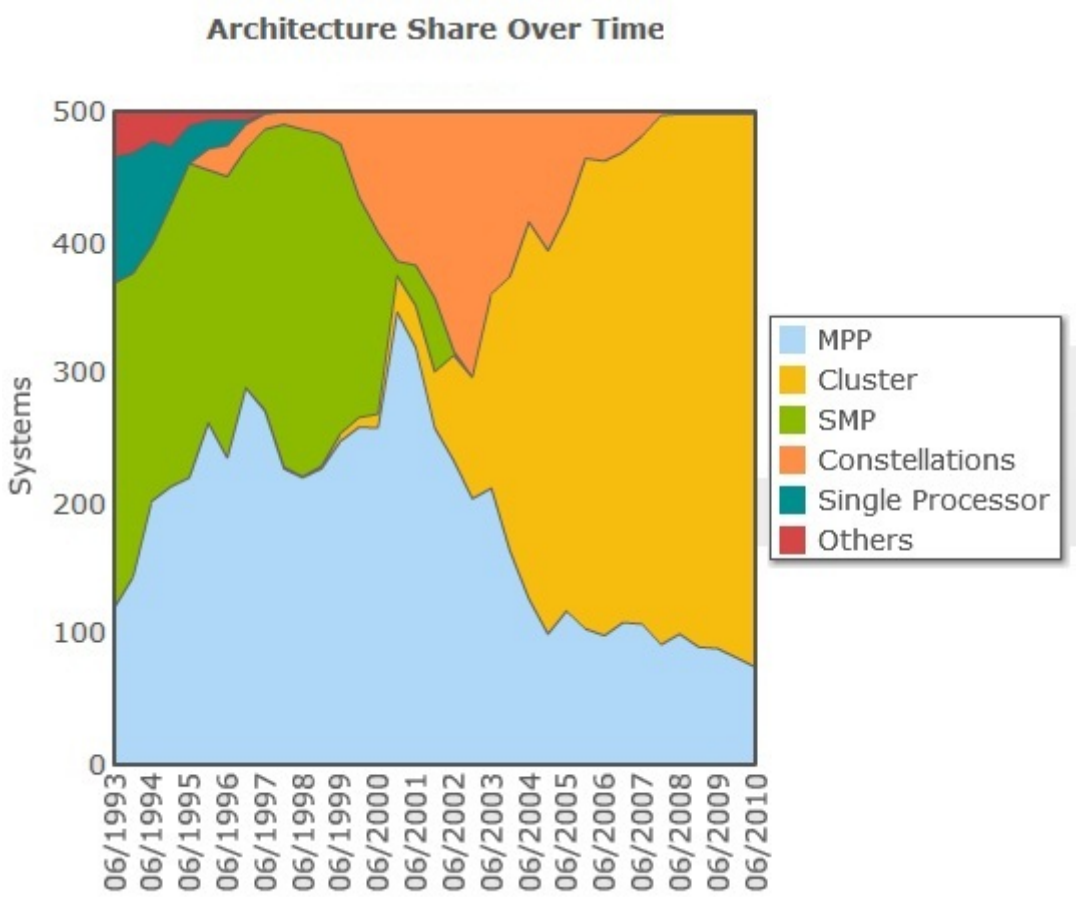


Figura 1.1: Arquiteturas dos 500 mais rápidos supercomputadores(figura por www.top500.org)

disponíveis. Grades podem ter propósito especial, quando se prestam a aplicações dentro de um determinado nicho específico ou mesmo a uma certa aplicação específica, é o caso do projeto Folding@home[5] que por meio do aproveitamento da ociosidade de computadores convencionais realiza inúmeras simulações do comportamento das proteínas para melhor entender o desenvolvimento de algumas doenças, como o Alzheimer, o Parkinson, alguns tipos de Câncer, dentre outras. Grades computacionais podem também ser de propósito geral, quando apenas oferecem o suporte à execução de programas paralelos, em geral usando o modelo *bag-of-tasks* [31].

Uma outra tecnologia emergente são os processadores com vários núcleos (*multi-core*), tornando o paralelismo real e explícito no processador. Intel e AMD já lançaram vários modelos com dois, três, quatro e seis núcleos. A tecnologia de processadores com vários núcleos está em constante desenvolvimento.

[5]http://folding.stanford.edu/

Recentemente, engenheiros da Universidade da Califórnia comprovaram em laboratório a viabilidade do processador AsAP (*Asynchronous Array of Simple Processors*), que possui 167 núcleos. [10]

É possível também mesclar algumas dessas arquiteturas, como Clusters de multiprocessadores, também conhecidos como constelações, Clusters com processadores de múltiplos núcleos, Grades de Clusters e MPP's, entre outros. De fato, arquiteturas híbridas, nas quais existe uma hierarquia em vários níveis de paralelismo, têm sido a tendência dominante no projeto de computadores paralelos para CAD. O grande desafio é a exploração eficiente do desempenho potencial dessas arquiteturas através do uso de interfaces de programação que promovam a produtividade no desenvolvimento de *software*.

1.2 Desafios em CAD

Como mencionado anteriormente, tecnologias de CAD são úteis para viabilizar aplicações em diversas áreas, como simulações químicas, otimização aerodinâmica, climatologia, mineração de dados, biologia computacional, mecânica de fluidos, simulações de fraturas em dutos e bacias petrolíferas, dentre outras. Sem muito esforço, podemos constatar que tais aplicações são de extrema importância para a nossa sociedade. Por exemplo, aplicações que ajudam a prevenir desastres naturais ou aplicações que auxiliam no desenvolvimento de novos fármacos, poderiam melhorar bastante a qualidade de vida. A importância dessas aplicações é um dos principais fatores que levam ao interesse e ao investimento em CAD por parte das universidades, dos centros de pesquisa, dos governos e das indústrias.

Aplicações desse porte requerem uma atenção ainda maior que as demais aplicações comerciais, não apenas pela necessidade de se tê-las, mas também pelos inúmeros desafios que surgem antes, durante e depois do seu desenvolvimento.

Um questão importante é a **multidisciplinaridade** em tais aplicações. Por exemplo, uma aplicação de previsão climática requer diferentes modelos físicos acoplados, como modelos para tratar os sistemas *oceano* e *atmosfera*.

Além do mais, os diferentes modelos envolvidos nas aplicações de CAD podem estar sendo executados em sítios computacionais geograficamente distantes. E realizar a **integração** dessas partes constitui um desafio, uma vez que essas podem ter sido desenvolvidas em ambientes computacionais diferentes, linguagens diferentes e etc.

Desenvolver e manter aplicações desse nível não é algo trivial, pois aplicações

desse gênero fazem uso de **modelos matemáticos complexos** e requerem um cuidado maior no gerenciamento de todas as funcionalidades que executam paralelamente, a fim de se ter um bom balanceamento de carga, evitar impasses (*deadlocks*), gargalos e outras eventuais falhas no sistema. Portanto, requerem não somente o conhecimento específico da área, mas também um vasto conhecimento em computação paralela. Portanto, a **implementação** e a **manutenção de códigos paralelos** é também um desafio em CAD.

Em aplicações de grande porte a reusabilidade de outros códigos de aplicação já implementados é um importante benefício para acelerar a produtividade e garantir a corretude da nova aplicação. No entanto, quando se trata de CAD oferecer e encontrar aplicações ou partes de aplicações que sejam reutilizáveis não é algo simples. Isso acontece pelo fato de que a maioria das aplicações de CAD são concebidas sem a utilização de artefatos da engenharia de software. Portanto, proporcionar a **reusabilidade** para as aplicações que demandam por CAD é também um desafio.

Outro desafio encontrado durante o desenvolvimento é que a execução da aplicação precisa ser **eficaz** e **eficiente**. E não é simples garantir a corretude de aplicações em um ambiente paralelo, devido às preocupações com a sincronização entre os processos, assim como também não é simples garantir a eficiência, pois aplicações desse porte exigem resultados rápidos para problemas complexos de serem resolvidos.

É fato que a indústria do *hardware* está em constante crescimento e sua evolução tem obedecido a Lei de Moore[6]. No entanto, as tecnologias de *software* não foram ainda capazes de explorar o máximo desempenho do *hardware* fornecido. Com isso, paradoxalmente apontam dois desafios. De um lado, o *software* precisa aproveitar o máximo do desempenho do *hardware*, e para isso necessitam ser específicas para um determinado *hardware* a fim de tirar o melhor proveito possível. Do outro lado, com o rápido avanço do *hardware*, a aplicação que fora desenvolvida para alguma arquitetura específica ficará obsoleta frente ao estado-da-arte do *hardware* antes do final do seu ciclo de vida útil. Assim, garantir em uma mesma aplicação a **portabilidade** e o **máximo aproveitamento das arquiteturas** é também um desafio.

Neste trabalho queremos contribuir com avanços nos desafios de integração, de

[6]A cada período de 18 meses, o número de transistores em um processador duplica-se e proporcionalmente o desempenho.

distribuição, de reusabilidade e de portabilidade entre aplicações sobre plataformas computacionais *Clusters*.

1.2.1 Componentes

Diante desses desafios, surge a necessidade de utilizar técnicas que possibilitem ganhos no que diz respeito ao gerenciamento e a comunicação de funções paralelas, a eficácia, a eficiência, a portabilidade, um melhor aproveitamento do *hardware* e um aumento na produtividade no desenvolvimento das aplicações.

Um dos meios de superar esses desafios é a utilização de partes genéricas de *softwares* pré-fabricadas na construção de outras partes maiores ou na construção de um novo *software*. A utilização desses blocos de *softwares*, chamados componentes, oferece benefícios ao desenvolvimento de novas aplicações, como, por exemplo, um considerável ganho na produtividade devido ao incremento do potencial de reuso de *software*. A configuração dos componentes e a criação de novas aplicações podem ser realizadas pelo estabelecimento de conexões entre os componentes, suprindo suas requisições. As conexões são estabelecidas após os componentes terem sido instanciados, podendo ser realizadas de diversas maneiras, definidas através das infraestruturas de componentes, dentre os quais podemos citar o CORBA, o JavaBeans e o .NET.

Os componentes para aplicações que demandam por CAD necessitam ser mais rápidos e escaláveis em processamento paralelo do que componentes de outros nichos de aplicação. Por conta disso, os modelos de componentes comerciais não se adequam a CAD [8, 90]. Para solucionar isso, foram propostos modelos de componentes voltados para CAD. Entre os quais, podemos citar o CCA [9], o GCM [15], o Fractal [23, 88] e o Hash [30].

Mais detalhes a respeito de modelos e infraestruturas de componentes, e de sua aplicação em CAD, encontram-se no Capítulo 2.

1.2.2 Bibliotecas Científicas

Um outro meio de transpor o desafio de oferecer mecanismos que facilitem o desenvolvimento de *software* com requisitos de CAD é a utilização de bibliotecas científicas para fins específicos [37]. Essas bibliotecas, cuja existência remonta aos primórdios da história das técnicas de programação de computadores, possuem características interessantes nesse contexto:

- Podem ser reutilizadas, aumentando assim a produtividade na construção de novas aplicações;

- São amplamente testadas nos mais diversos ambientes ao longo de muitos anos, possibilitando maior eficácia e eficiência;
- Utilizam os melhores algoritmos conhecidos para determinado problema, desse modo facilitando a eficiência da aplicação;
- Oferecem algoritmos voltados para o processamento paralelo, encapsulando a comunicação entre os processos;
- Podem ser implementadas para arquiteturas específicas, sem compromisso com generalidade, melhorando assim o aproveitamento do potencial de desempenho do *hardware* disponível.

Vale ressaltar que ao contrário dos componentes de softwares, as bibliotecas científicas não são instanciáveis e são integradas no processo de compilação. Dificultando assim, possíveis mudanças do cenário do problema em tempo de execução.

Existem inúmeros desafios durante a concepção de bibliotecas científicas, dentre os quais destacamos o tratamento da comunicação entre os processos e a portabilidade entre as arquiteturas, especialmente as arquiteturas paralelas. Tendo como primeiro objetivo superar esses desafios no desenvolvimento de bibliotecas científicas, foi proposto o MPI (*The Message Passing Interface*) [40], uma interface de comunicação por troca de mensagens sobre a qual bibliotecas científicas paralelas podem ser implementadas de maneira eficiente e portável, a qual transformou-se em um padrão de fato para programação paralela de propósito geral sobre arquiteturas distribuídas. Sobre o MPI, bibliotecas de comunicação mais específicas para as necessidades de bibliotecas científicas paralelizadas tem sido implementadas, como o BLACS (*Basic Linear Algebra Communication Subroutines*) [41], voltado a bibliotecas de suporte a álgebra linear computacional.

Detalhes sobre bibliotecas científicas, em especial no domínio de solução de sistemas lineares, tratado no estudo de caso desta dissertação, serão apresentados no Capítulo 3.

1.3 Objetivos da Dissertação

Como discutido, a utilização de componentes de *softwares* voltados para CAD fornece meios para superar os desafios no desenvolvimento de aplicações de alto desempenho. Da mesma forma, a utilização de bibliotecas científicas para fins

específicos oferece outros meios para transpor alguns limites no desenvolvimento de aplicações de alto desempenho. Portanto, utilizar as duas técnicas em um mesmo arcabouço de desenvolvimento de uma aplicação de alto desempenho poderá trazer benefícios ainda maiores.

Pensando dessa forma, o objetivo principal deste trabalho é oferecer um método sistemático para integrar bibliotecas científicas à infraestrutura de componentes paralelos HPE (*Hash Programming Environment*), baseada no modelo # de componentes, buscando oferecer aos desenvolvedores de programas paralelos que implementam aplicações que demandam por tecnologias de CAD os aspectos vantajosos do uso de componentes e de bibliotecas científicas.

Nossa proposta pode ser aplicada em diversos domínios de bibliotecas científicas. Devido à sua grande importância em aplicações computacionais, utilizamos para a validação da nossa proposta o domínio de bibliotecas científicas para a solução de sistemas lineares. Embora nosso esforço seja validar a proposta com um único domínio, a validação pode ser extrapolada para os outros domínios, desde que as bibliotecas pertencentes a um mesmo domínio possuam algumas características semelhantes, como os passos básicos para a execução da biblioteca e as funcionalidades de suas subrotinas. Para isso, como estudo de caso, utilizamos as bibliotecas PETSc, SuperLU e HYPRE.

Enumeramos ainda os seguintes objetivos específicos atingidos:

- Proposta de um conjunto de componentes paralelos, sobre a plataforma HPE, para aplicação em solução de sistemas lineares, tendo em vista a grande importância desse domínio de bibliotecas científicas em aplicações reais de grande interesse econômico e científico;

- Validação da plataforma de componentes HPE para integração com código nativo, especialmente com o código de bibliotecas científicas escritas em código nativo. Trata-se de um requisito importante para garantir o potencial de alto desempenho da plataforma, delegando ao código nativo a execução de partes do código com alta demanda computacional;

- Exercitar a possibilidade da inclusão de novas *espécies de componentes* ao HPE para suporte a domínios de aplicação específicos, onde os components possuem características especiais;

- Acrescentar mais evidências sobre a eficácia do uso de componentes para

aplicações de computação de alto desempenho, notadamente nos domínios das ciências computacionais e engenharias.

1.4 Estrutura do Texto

Além deste capítulo introdutório, que descreve as motivações, os objetivos e a estrutura desta dissertação, apresentamos no Capítulo 2 os componentes de *software*, assim como suas infraestruturas, especialmente aquelas voltadas para CAD. Especial ênfase será dada à plataforma de componentes HPE (*Hash Programming Environment*), objeto de estudo desta dissertação. No final desse capítulo é apresentado um estudo de caso experimental que evidencia a importância do objetivo dessa dissertação. No Capítulo 3 descrevemos as bibliotecas científicas destinadas para aplicações de CAD. Abordamos os principais domínios de uso das bibliotecas, destacando o domínio de sistemas lineares e descrevendo as bibliotecas científicas SuperLU, Hypre e PETSc, usadas no estudo de caso desta dissertação. No Capítulo 4 apresentamos a proposta do nosso trabalho, que é um método para a integração de bibliotecas científicas na plataforma de componentes paralelos HPE. No Capítulo 5, é apresentado um estudo de caso simples, que exemplifica o método proposto, aplicado sobre o domínio de bibliotecas para solução de sistemas lineares. As conclusões desta dissertação são apresentadas no Capítulo 6.

Capítulo 2

Componentes em Computação de Alto Desempenho

Um dos fatores que alavancaram a velocidade do crescimento da indústria do *hardware* foi a utilização de componentes pré-fabricados, que possuem um modelo computacional bem definido, a Álgebra Booleana. Esse foi um dos motivos pelos quais a indústria do *hardware* teve um rápido desenvolvimento. No entanto, a indústria do *software* não conseguiu acompanhar esse rápido avanço, tendo como consequência o que foi considerado na década de 70 como a crise do *software* [35]. Com o avanço do *hardware*, cresceu também a demanda por *software*s cada vez mais complexos. Todavia não existiam, naquela época, técnicas estabelecidas para o desenvolvimento de *software*s de grande escala, o que acarretou diversos problemas, como estouro de orçamento e prazo, *software*s de baixa qualidade com difícil manutenção de código e gerenciamento, entre outros.

Na tentativa de solucionar a crise do *software*, foram propostos vários artefatos de engenharia voltados ao desenvolvimento de *software*. Técnicas de modularizarão de programas e abstração emergiram como alternativas para lidar com a complexidade e escalabilidade do *software*, especialmente em aplicações do nicho corporativo. Dentre essas, estão os módulos, os tipos abstratos de dados e os objetos.

Uma das técnicas consideradas, inspiradas na indústria do *hardware*, foi a utilização de partes genéricas reutilizáveis, funcionalmente independentes, auto-implantáveis e sujeitas a composição por terceiros na construção de outras partes maiores com as mesmas características ou na construção de um novo *software* completo. Esses blocos de *software*s ficaram conhecidos como componentes de *software*. Existem várias definições para componentes de *software*, mas que diferem

em alguns pontos. Seguem duas definições mais abrangentes:

i. "Um componente de *software* é uma unidade de composição com interfaces de contrato bem especificadas e que possui apenas dependências explícitas de contexto. Um componente pode ser implantado independentemente e ser sujeito à composição por terceiros." [84]

ii. "Um componente de *software* é uma parte física de *software* executável com interfaces bem definidas e publicadas." [53]

Vale ressaltar a necessidade de distinguir a programação baseada em componentes dos demais estilos de estruturação de *software*, dentre os quais devemos destacar a a programação modular, a programação orientada a objetos (OO) e a programação baseada em tipos abstratos de dados (TAD). Enquanto os estilos OO e TAD enfatizam mecanismos de abstração que permitem a criação de programas de acordo com uma modelagem do mundo real representado através de objetos e tipos abstratos de dados, a programação baseada em componentes e a programação modular preocupam-se com maior ênfase na arquitetura do *software*. Na programação baseada em componentes, o programa é criado através da conexão de componentes de *software* pré-fabricados e pré-compilados, enquanto na programação modular a divisão das partes do *software* é observável somente durante o ciclo de desenvolvimento, e não em tempo de execução.

Vários são os benefícios de se utilizar componentes no desenvolvimento de *software*. O primeiro, e talvez o mais importante, é a reusabilidade, que permite que componentes feitos por terceiros possam ser reutilizados em outras aplicações. Outra característica importante é que componentes são substituíveis, possivelmente em tempo de execução. Ou seja, um componente pode ser substituído por outro que realize seu mesmo papel, podendo ser ou uma versão mais atual do componente ou um componente alternativo. É esperado também que os componentes possam ser compostos ou substituídos independentemente da linguagem de programação.

As aplicações de CAD também demandam por técnicas de engenharia de *software* para transpor alguns desafios [73, 76, 80]. Para alcançar tal objetivo é importante que os artefatos da engenharia de *software* tenham como premissas os benefícios de integração, distribuição e paralelização para a aplicação.

O paradigma de programação orientada a objetos já é bastante utilizado, com êxito, para lidar com a complexidade no processo de desenvolvimento de *software* de

larga escala. No entanto, esse paradigma possui limitações relacionadas ao requisito de integração entre objetos. Essa carência advém da carência de compatibilidade entre as linguagens OO e por não existir um padrão para a comunicação entre os objetos. Dessa forma, inviabiliza seu uso para a construção de aplicações de CAD, uma vez que nessas aplicações é comum que exista uma combinação de contribuições de diversas áreas do conhecimento, como equações diferenciais, otimização, álgebra linear, dentre outras. Por vezes, são compostas por partes, onde cada uma dessas partes é desenvolvida por diferentes instituições, de acordo com sua especialidade, como em sistemas multi-física. Infelizmente, os cientistas que as desenvolvem utilizam uma variedade de linguagens, tornando o trabalho de combinação das partes uma tarefa árdua. Por outro lado, também não se pode exigir que os cientistas utilizem apenas uma linguagem padrão. Portanto, quando se trata de aplicações de CAD, é indispensável que as partes sejam interoperáveis de forma independente de linguagens de programação específicas.

A programação orientada a componentes supre essas carências trazendo os benefícios da interoperabilidade entre linguagens, através de padrões para a comunicação entre os componentes definidos pela infraestrutura de componentes. Também é possível implantar e executar componentes de forma distribuída, suprindo a necessidade de distribuição das partes. Algumas infraestruturas de componentes voltadas para CAD buscam suprir o requisito de processamento paralelo, embora com limitações [27]. Portanto, ao utilizarmos componentes de *software* dispomos dos benefícios de integração, distribuição e paralelização, motivando a constatação de que a utilização de componentes supre a necessidade de integração entre as partes que compõem o *software* de CAD [90].

2.1 Infraestruturas de Componentes

Componentes estão intimamente associados a diferentes infraestruturas de suporte. As definições concretas de componentes e de suas formas de conexão e implantação em uma plataforma de execução definem uma infraestrutura de componentes. Em outras palavras, uma infraestrutura de componentes é definida por seus modelos. Para se criar e validar um novo componente em uma determinada infraestrutura, é necessário seguir o *modelo de componentes* da infraestrutura. Da mesma forma, para conectar dois ou mais componentes é preciso seguir o seu *modelo de conexão*. É também preciso seguir o *modelo de implantação* da infraestrutura para configurar componentes em uma aplicação [92]. São exemplos de infraestrutura de

componentes o EJB(*Enterprise JavaBeans*) [56], o CORBA [78], o .NET [43], o OSGi [71], o COM [20], e o Web Services [5], de uso disseminado em aplicações corporativas.

Quando se trata de aplicações de CAD, existem requisitos importantes que não são prioritários em aplicações corporativas, como o alto desempenho de execução e das interconexões entre os componentes, bem como o processamento paralelo escalável. Por conta disso, o uso das infraestruturas citadas torna-se inviável para alcançar os objetivos de uma aplicação em CAD [8, 90]. Para solucionar isso, ao longo da década atual, foram propostas infraestruturas de componentes voltadas para CAD, discutidas a seguir.

2.2 Componentes para CAD

Como mencionado anteriormente, a programação orientada a componentes supre os requisitos de integração, distribuição e paralelização de *software* com requisitos de CAD. Entretanto, as infraestruturas de componentes para fins de aplicações corporativas não atendem por completo requisitos mais específicos. Por exemplo, uma aplicação baseada no modelo de componentes JavaBeans não dispõe de interoperabilidade com outras linguagens além do Java. CCM (*Corba Component Model*) oferece meios para que a aplicação se comunique de forma transparente com outras aplicações escritas em linguagens diferentes, porém não oferece suporte a tipos de dados comuns em CAD, como números complexos e vetores multidimensionais, bem como impõe um modelo de conexão pouco eficiente, com maior ênfase nas questões de segurança e interoperabilidade em detrimento das questões de desempenho. O modelo de componentes COM sofre das mesmas deficiências do CCM.

Ainda que as infraestruturas de componentes para fins comerciais garantissem a interoperabilidade entre as aplicações de alto desempenho, o seu baixo desempenho na comunicação nos leva a desconsiderá-las. Essa excessiva latência se dá pelo fato de que a comunicação é sempre realizada indiretamente, o que as tornam inviáveis para aplicações de CAD, para as quais a velocidade da comunicação é um fator preponderante no seu desempenho global.

Além disso, as infraestruturas de componentes não foram desenvolvidas com o processamento paralelo em vista, tampouco o modelo de componentes sobre o qual estão baseadas. Extensões paralelas tem sido propostas, mas não conseguem atingir o nível de generalidade e flexibilidade de abordagens não componentizadas, como

a programação com MPI (*Message Passing Interface*) [40], levando ainda a quebra da unidade funcional dos componentes, devido ao tratamento de processos como componentes [30].

Visando superar essas limitações, foram propostas infraestruturas de componentes voltadas para aplicações de CAD, baseadas em modelos de componentes específicos para os requitos de aplicações CAD, como o CCA [9], o GCM [15], o Fractal [23, 88] e o Hash [30], este último objeto do trabalho desta dissertação.

2.3 CCA

O CCA (*Common Component Architecture*) foi proposto no final da década de 1990 por cientistas computacionais de laboratórios nacionais e universidades sediados nos Estados Unidos, financiados pelo DOE (U.S Department of Energy) no âmbito do CCTTSS (Center for Component Technology for Terascale Simulation *software*), parte do programa SciDAC (Scientific Discovery through Advanced Computing) [2,8,9].

O CCA toma por base modelos de componentes empregados nas aplicações comerciais, como CCM (Corba Component Model) e COM, buscando adaptá-los segundo os requisitos de desempenho de aplicações CAD. Semelhante ao CORBA e ao COM, o CCA é baseado no padrão *provides/uses*, onde cada componente provê recursos (portas do tipo *provides*) e depende de recursos (porta do tipo *uses*). É responsabilidade dos *frameworks* CCA intermediar o estabelecimento de conexões entre portas *uses* e portas *provides*, de mesmo tipo, entre componentes distintos.

Para o suporte à interoperabilidade entre aplicações desenvolvidas em linguagens distintas, bem como oferecendo um melhor suporte a aplicações dos domínios científicos, o CCA oferece a SIDL (*Scientific Interface Description Language*), uma IDL com suporte a tipos de dados comuns em CAD, como números complexos e arrays multi-dimensionais, para tornar o uso das aplicações científicas transparente ao desenvolvedor. A ferramenta Babel foi desenvolvida para permitir que componentes escritos em diferentes linguagens como Fortran 90, Fortran 77, C, C++ e Phyton possam se comunicar de forma transparente, como ilustrado na Figura 2.1. Dessa forma, o desenvolvimento das aplicações pode ser realizado de forma mais flexível. Experimentalmente, tem sido demostrado que fazer uso da interface SIDL afeta de forma insignificante o tempo total da computação [9]. Isso é possível devido às conexões diretas entre os componentes quando residem em espaço

de memória compartilhada, as quais não são possíveis em CORBA, onde a conexão entre os componentes é sempre intermerdiada através dos elementos *stub* e *skeleton*.

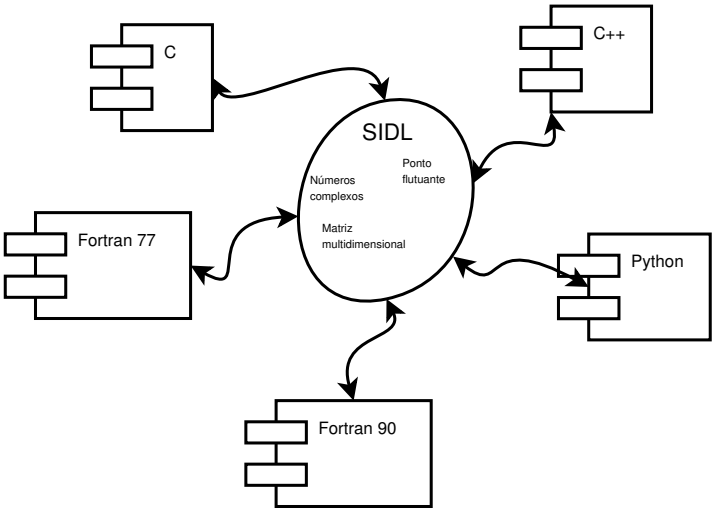


Figura 2.1: Interoperabilidade entre componentes através da SIDL

Frameworks CCA são programas que criam e conectam componentes para construir aplicações. São exemplos de *frameworks* que aderem ao modelo CCA: o SCIRun2 [72], o Ccaffeine [4], o DCA [17], o XCAT [50]. Para criar um *framework* CCA, o desenvolvedor deve necessariamente implementar interfaces descritas em SIDL na especificação oficial do CCA [87]. A seguir, apresentamos as interfaces que acreditamos serem as mais relevantes para o entendimento do modelo CCA (vide também Figura 2.2).

- AbstractFramework: publica métodos para obter o objeto do tipo Services necessário para a manipulação das portas dos componentes CCA, permitindo ainda que um programa externo, o qual deseja instanciar e conectar-se a componentes através do *framework*, seja enxergado como um componente pelo próprio *framework*;

- Services: publica métodos através dos quais os componentes que executam no *framework* podem declarar suas portas *uses* e/ou portas *provides*;

- BuilderService: publica métodos através dos quais o usuário do *framewrork* (programa, interface de linha de comando ou interface gráfica) pode instanciar componentes e conectar suas portas *uses* e *provides* previamente declaradas;

- ComponentRepository: publica um método através do qual os componentes

disponíveis em um repositório tornam-se visíveis aos usuários, a fim de serem instanciados por meio da interface BuilderService;

- Component: deve ser implementado por qualquer componente CCA, possuindo o método setService, através do qual o *framework* oferece ao componente o objeto do tipo Services necessário para publicação de suas portas;

Dessa forma os passos básicos para a composição e a realização de componentes CCA são:

i. O usuário escolhe um ou mais componentes publicados por meio da interface ComponentRepository;

ii. O usuário executa o método *createInstance*, da interface BuilderService, para instanciar os componentes com os quais ele deseja trabalhar;

iii. Ao ser criada uma instância de um componente, o *framework* chama automaticamente o método setService implementado em cada componente, causando a publicação de suas portas devido às chamadas aos métodos addProvidesPort e registerUsesPort da interface Services, as quais tornam-se visíveis ao usuário através do *framework*;

iv. O usuário executa o método connect, da interface BuilderService, para conectar par-a-par portas *uses* de componentes ditos clientes a portas *provides* de componentes ditos servidores, ambas de um tipo comum;

v. O método go, de cada componente que implementa a porta padrão Go, é executado, iniciando a lógica computacional implementada pelo componente;

vi. Através do método getPort, da interface Services, invocado como efeito da execução do método go, os componentes clientes podem acessar os serviços dos componentes servidores a eles conectados através das portas *uses*.

vii. Ao final, é executado pelo *framework* os métodos *releaseServices* e *shutdownFramework* para finalizar o *framework*.

Cada *framework* CCA desenvolvido durante a década de 2000 tem buscado exercitar com maior ênfase requisitos específicos de aplicações CAD, com vistas ao aprimoramento do CCA em futuras versões da especificação. Por exemplo, o CCAffeine enfatiza a interoperabilidade através do Babel, a conexão direta entre os

componentes, e o paralelismo através do estilo SCMD (Single Component Multiple Data), através do qual um componente paralelo é definido como um regimento de componentes iguais, cada qual executado em um dos nós de um *cluster* e comunicando-se com os demais através de troca de mensagens usando alguma interface usual, como MPI. Já o XCAT enfatiza a possibilidade de componentes estarem distribuídos, quando então comunicam-se através de invocações remotas de métodos (conexão indireta) ao invés de conexões diretas. O DCA, por outro lado, tenta integrar os aspectos de distribuição e paralelismo, oferecendo meios para componentes SCMD que residem em conjuntos distintos de nós de uma plataforma distribuída de execução se comunicarem através de PRMI (*Parallel Remote Method Invocation*) implementado sobre o MPI. De fato, o DCA foi especialmente desenvolvido para investigações sobre os acoplamentos $M \times N$ entre regimentos de componentes com números distintos de membros do lado cliente (M) e do lado servidor (N). Essa problemática é também de interesse dos projetistas do *framework* SciRun2, o qual ainda oferece suporte a interoperabilidade com componentes CORBA e Dataflow.

2.4 Fractal

O modelo de componentes Fractal [19, 22] foi proposto no início da década de 2000 por pesquisadores advindos da France Telecom e INRIA (*Institut National de Recherche en Informatique*), financiados no contexto to consórcio CoreGrid, envolvendo instituições de pesquisa européias. Foi projetado como um modelo de componente modular e extensível, com o objetivo do suporte ao projeto, implementação, implantação e reconfiguração de sistemas e aplicações, onde o componente pode fazer uso de diferentes formas de composições, diferentes formas de conexões e diferentes linguagens de programação.

O modelo Fractal possui as seguintes características:

- *Recursividade*: componentes podem ser compostos de forma hierárquica, formando um novo componente;
- *Reflexibilidade*: um componente pode explorar e expôr suas próprias características estruturais através de interfaces que oferecem serviços de reflexão;
- *Componentes compartilhados*: um componente pode ser compartilhado por diversos outros componentes;

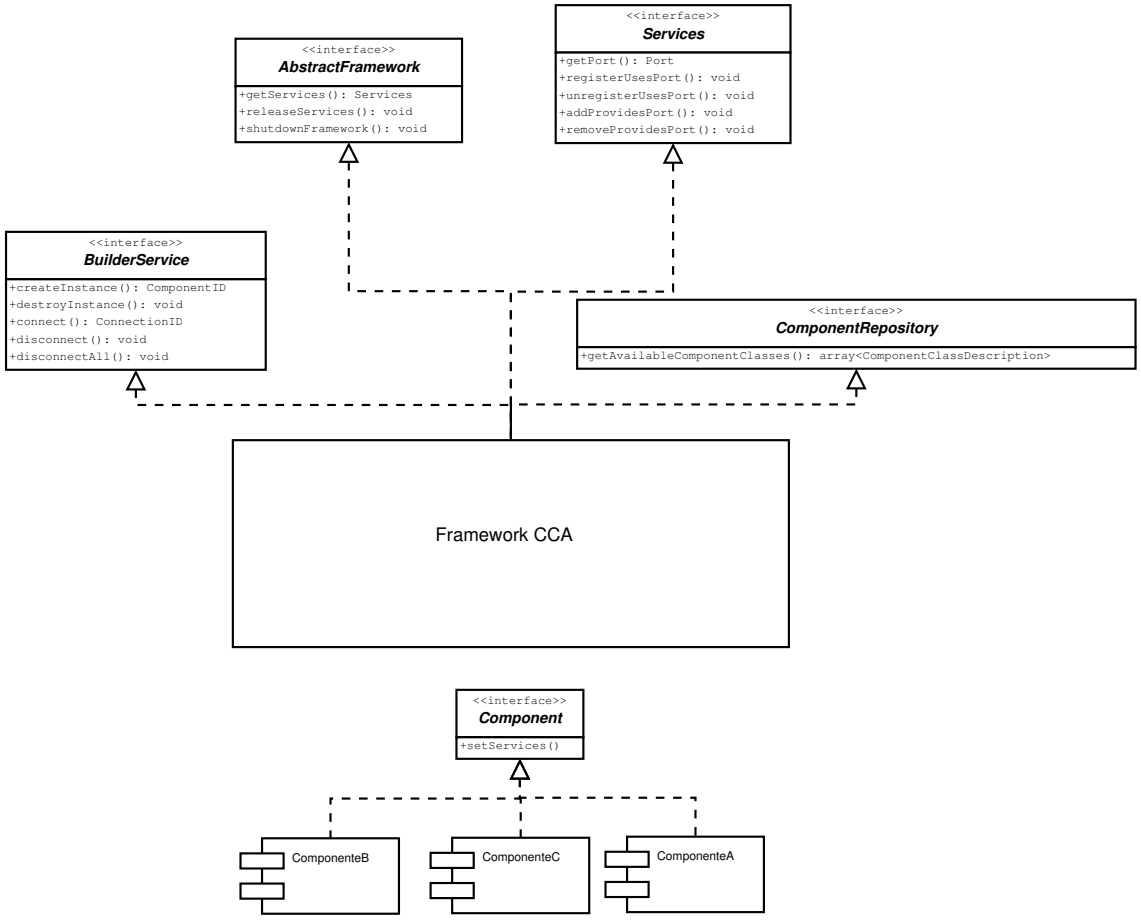


Figura 2.2: Principais interfaces da especificação do modelo CCA

- *Componentes ligação*: implementam a única abstração de componente para conexão existente no modelo. Tal implementação pode conter qualquer padrão de comunicação síncrona;
- *Execução independente do modelo*: componentes podem livremente executar com diferentes modelos de execução;
- *Aberto*: interesses não funcionais de um componente podem ser ajustados através de uma membrana de controle.

O componente Fractal é composto basicamente de três partes, ilustradas na Figura 2.3:

- **Interface**: constitui o meio de exposição dos componentes. Uma interface pode ser *servidora*, por onde o componente permite acesso aos seus recursos, ou *cliente*, por onde o componente acessa recursos exteriores. De fato, trata-se de um conceito semelhante as portas *uses* e *provides* de componentes CCA;

- **Membrana**: envolve o componente Fractal e possui *interfaces* para gerenciar os interesses não funcionais do componente (características internas, como configuração, segurança, transações, dentre outras), e *controladores*, para, dentre outras possibilidades, controlar o ciclo de vida do componente. Existem na *membrana* as *interfaces externas*, que são acessíveis por componentes exteriores, e as *interfaces internas*, que são acessíveis apenas por componentes internos à *membrana*;

- **Conteúdo**: consiste em conjunto de componentes internos à *membrana*, ditos sub-componentes, que gerenciam os interesses funcionais, os quais definem a computação propriamente dita realizada pelo componente.

O modelo não define os tipos de CONTROLADORES da MEMBRANA, mas os classifica em três níveis de acordo com a visibilidade do componente. No nível inferior, o componente é apresentado com uma caixa-preta e não oferece nenhuma visibilidade interior. No nível intermediário, o componente dispõe da interface Component, o qual permite descobrir as interfaces CLIENTES e SERVIDORAS do componente. No nível superior, o componente expõe mais detalhes sobre elementos da sua estrutura interna (conteúdo).

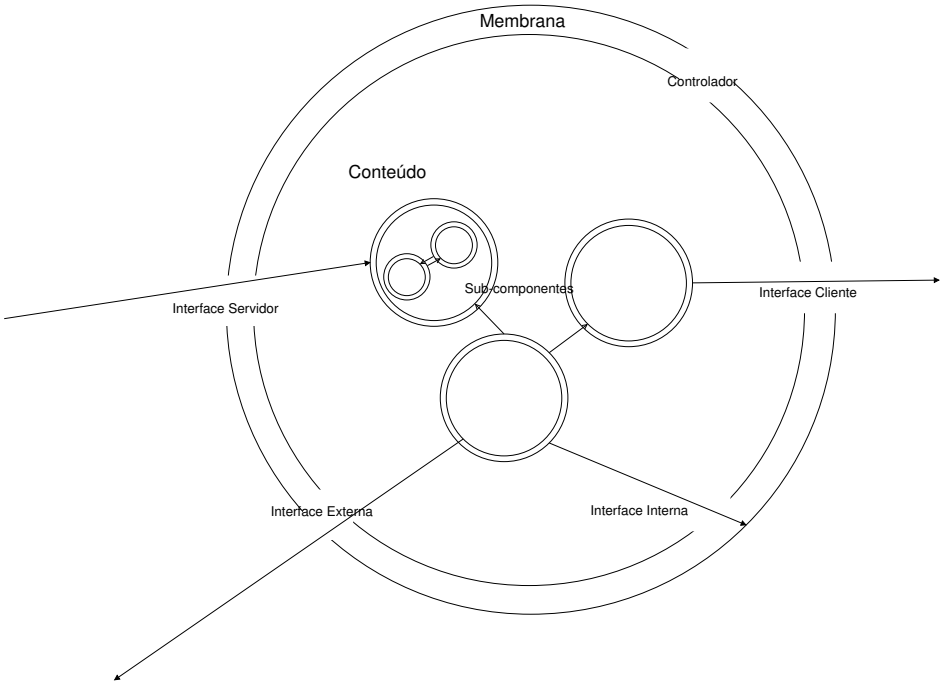


Figura 2.3: Estrutura de um componente Fractal

No modelo de conexão do Fractal, para existir a comunicação entre os

componentes deve-se primeiro ligar as INTERFACES dos componentes. Para realizar a ligação entre as INTERFACES o modelo Fractal dispõe de dois tipos de ligações (*bindings*). A ligação primitiva promove a ligação de uma INTERFACE CLIENTE com uma INTERFACE SERVIDORA em um mesmo espaço de endereçamento, enquanto a ligação composta promove a ligação de um número variado de INTERFACES. Depois de ligadas as INTERFACES, as requisições da INTERFACE CLIENTE para executar alguma operação de um outro componente devem ser aceitas pela INTERFACE SERVIDORA do componente solicitado.

Várias implementações foram desenvolvidas obedecendo o modelo de componentes Fractal, dentre as quais destacamos o JULIA [22] e ProActive [25]. Nessas implementações foram utilizadas diversas linguagens, como Java, C, C++, Smalltalk e .NET. O ProActive surgiu antes do próprio Fractal ter sido proposto. De fato, o modelo Fractal foi especificado na experiência dos projetistas e desenvolvedores do ProActive. Em versões posteriores à definição do modelo Fractal, o ProActive passou a aderir ao modelo Fractal, quando passou a ser referenciado como ProActive/Fractal.

2.5 GCM

Criado pela comunidade CoreGrid, o GCM (*Grid Component Model*) [48] é uma extensão do modelo Fractal visando o uso em Grades Computacionais. Os componentes GCM possuem as mesmas características básicas do modelo Fractal, como a recursividade, o compartilhamento de componentes, a separação dos interesses, dentre outras. Por ser uma extensão do modelo Fractal, destacamos nesta seção apenas as principais características agregadas ao modelo Fractal para dar suporte ao uso em ambientes de Grades computacionais.

A arquitetura dos componentes GCM é definida da seguinte forma: a especificação do componente é descrita usando uma linguagem ADL (*Architecture Description Language*) em um documento XML, no qual são definidos os componentes primitivos, os componentes compostos, as interfaces e os aspectos de Grades computacionais; deve-se implementar API's (*Application Programming Interface*) de tempo de execução em várias linguagens, possibilitando assim uma manipulação padrão dos componentes em execução, como o gerenciamento do ciclo de vida, otimização, monitoramento, reconfiguração, dentre outras; além da especificação do componente, é preciso também possuir algumas informações sobre o ambiente de Grade alvo, como as requisições do *hardware*, a localização do

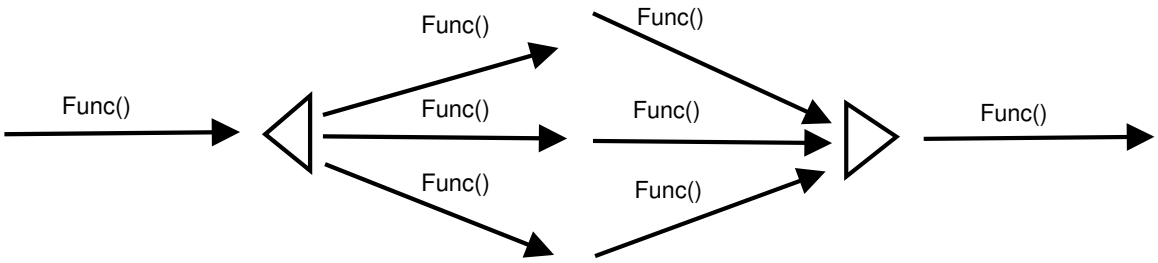
código na plataforma e a dependência entre versões, essas e outras informações são especificadas em um documento XML, permitindo assim a implantação dos componentes em vários contextos.

O modelo de componentes GCM também oferece interoperabilidade, tornando possível a sua implementação sobre diversas infraestruturas de grades, como também possibilitando que os componentes GCM se envolvam com um invólucro que segue o padrão Web Services (WS). Dessa forma, tanto o *framework* WS pode se favorecer do *framework* GCM, como os componentes GCM podem utilizar serviços no padrão Web Services.

No modelo Fractal, a comunicação entre os componentes pode ser de diversos tipos, bastando para isso que o tipo de comunicação seja especificado no tipo da INTERFACE, onde apenas as comunicações compatíveis podem ser ligadas (por exemplo, a *interface cliente* ser do mesmo tipo da *interface servidora*). Dentre esses tipos de comunicação possíveis no GCM, destacamos o tipo *Stream*, tão comum em Grades computacionais, que permite uma implementação explícita do fluxo de dados em apenas uma direção, possibilitando assim uma otimização do desempenho em tempo de execução.

Outra característica importante do modelo GCM é o suporte a *interfaces coletivas multicast e gathercast*, onde, através dessas novas interfaces, a comunicação coletiva é facilitada por não mais necessitar de componentes intermediários. Cada *interface coletiva* possui um *controlador* para realizar configurações e ajustes dinâmicos. A *interface multicast* tem a função de transformar uma única invocação em um conjunto de invocações. Já a *interface gathercast* tem a função de transformar um conjunto de invocações em uma uma única invocação. Podemos observar o comportamento básico das *interfaces coletivas* na Figura 2.4. Por meio dessas *interfaces* é possivel realizar comunicações de $1{\times}M$ e $M{\times}1$ [48]. Para suporte a comunicação $M{\times}N$, propõe-se o uso de M interfaces *gathercast* e N interfaces *broadcast* ou o uso de controladores, através dos quais o programador pode especificar as trocas diretas entre os processos clientes e servidores.

Para melhor se adequar ao ambiente heterogêneo de Grades Computacionais, o modelo GCM utiliza os componentes controladores, de forma dinâmica, de acordo com o contexto. Para tanto, os componentes controladores são tratados como sub-componentes, de forma que podem ser adicionados, conectados e desconectados de forma dinâmica. Visando ainda uma melhor adaptabilidade em ambientes heterogêneos, a definição dos componentes no modelo GCM segue os princípios



(a) Interface Multicast (b) Interface Gathercast

Figura 2.4: Interfaces coletivas no modelo GCM

da Computação Autônoma [83], que visa facilitar o gerenciamento de sistemas complexos incluindo elementos auto-gerenciáveis ao sistema, tendo por requisitos a auto-configuração, a auto-cura, a auto-otimização e a auto-proteção. Para isso, cada componente dispõe de interfaces para consultar e controlar o comportamento de aspectos não-funcionais. Assim, o uso de componentes controladores como sub-componentes, e o uso de componentes autônomos oferecem uma melhor adaptabilidade em ambientes heterogêneos.

Implementações sobre o modelo GCM já foram realizadas. Por exemplo, foi desenvolvida uma solução para garantir a interoperabilidade entre todos os *middleware* existentes [54]. Também tem sido desenvolvido o ProActive/GCM [6], incluindo o suporte a serviços de nuvens computacionais (*cloud computing*).

2.6 HASH

Visando aproximar as práticas modernas de engenharia e arquitetura de *software* da prática da programação paralela de propósito geral, foi proposto o modelo HASH de componentes [28], doravante referido com o símbolo #. Ao contrário dos outros modelos, que herdam a prática tradicional do desenvolvimento de programas paralelos sob a perspectiva orientada a processos, usando componentes para encapsulá-los, o cerne do modelo # é o desenvolvimento com enfoque em interesses, premissa fundamental na engenharia de *software*. Cada componente paralelo deve abordar um interesse de *software*, em contraposição à abordagem tradicional de decompor um interesse que deve ser tratado em vários nós de processamento em vários componentes, sob uma perspectiva de componentes distribuídos. Portanto, a partir da decomposição de um programa paralelo em seus interesses, através de componentes, podemos chegar à sua decomposição de processos de forma natural, mapeando-os aos nós de processamento, abordagem que acreditamos ser mais

compatível com os artefatos de engenharia de *software* hoje empregados [66].

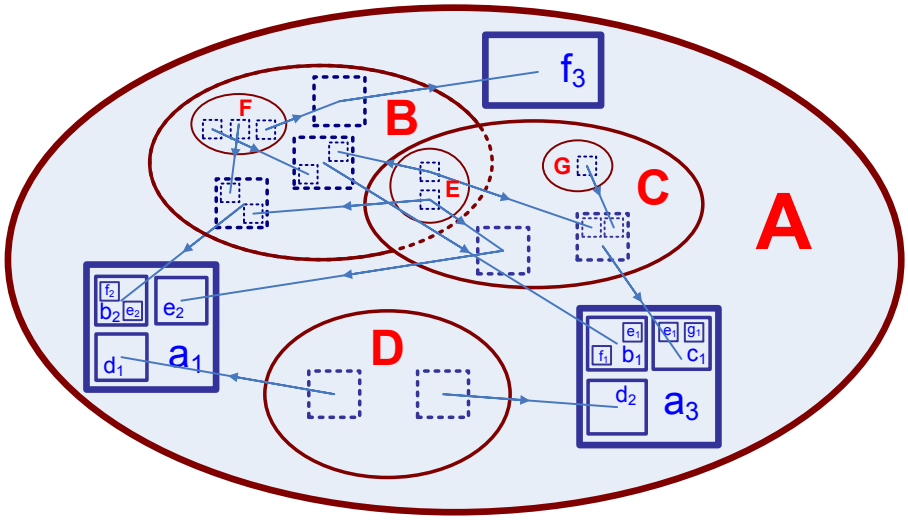


Figura 2.5: Sobreposição de Componentes #

Segundo o modelo #, um componente, dito componente-#, é formado por um conjunto de *unidades*, cada uma implantada em um dos nós de processamento de uma plataforma distribuída de execução paralela. Vários componentes, ditos componentes aninhados, podem ser compostos através da noção de *composição por sobreposição* para formar um novo componente, de mais alto nível, o qual implementa um dado interesse a partir dos interesses implementados por seus componentes aninhados. A composição por sobreposição é bastante simples, conforme ilustrado na Figura 2.5, onde as elipses representam os componentes-# A,B,C,D,E e os quadrados representam as unidades dos componentes, como as unidades a_1, f_3, a_3 e etc. A composição por sobreposição também tem sido formalizada algebricamente [29]. Cada unidade de um componente-# pode importar unidades de componentes aninhados distintos, os quais são ditas *fatias* da unidade que as importa, representado na Figura 2.5 através dos quadrados internos d_1, f_2, g_1 e etc. Um programa paralelo é então especificado por um componente de mais alto nível, cujas unidades correspondem aos processos que serão mapeados aos nós de processamento. Note na Figura 2.5 a possibilidade de compartilhamento de componentes, representada pelo componente E, compartilhado entre B e C.

Um forma mais prática de entender o modelo # a partir de seus princípios básicos é observar como um programa paralelo visto sob a perspectiva tradicional de processos pode ser decomposto em componentes #. Para isso, os processos que compõem a aplicação paralela podem ser subdivididos em várias fatias (*slices*) de

acordo com interesses da aplicação. As fatias (de diferentes processos) podem ser agrupadas em um interesse comum. Por exemplo, observe os processos P0 e P1 da Figura 2.6. Ambos são compostos de várias fatias, onde algumas das quais compartilham o mesmo interesse, como as fatias multiplicar de P0 e P1 e as fatias distribuir de P0 e P1.

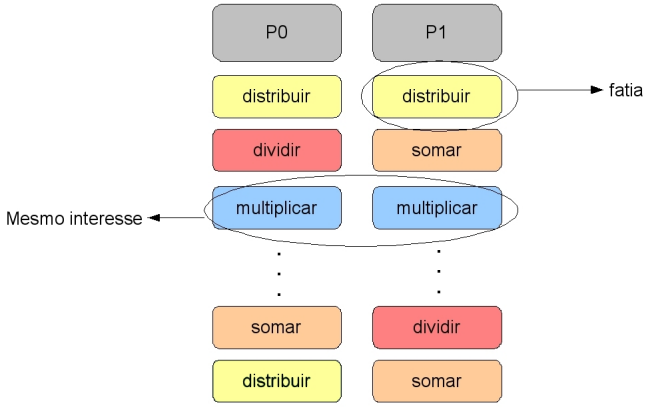


Figura 2.6: Orientado por interesses

Esses interesses podem ser funcionais, como a computação propriamente dita e operações de sincronização, ou podem ser não funcionais, como a definição de políticas de segurança e critérios de eficiência.

As fatias de processos que correspodem a um mesmo interesse, podem ser encapsuladas em um componente-#, onde cada fatia correspode a uma de suas unidades. Portanto, entende-se por unidade o papel que um processo realiza em um determinado interesse. Dessa forma, podemos entender que um mesmo componente-# pode ser implantado em um conjunto de máquinas, permitindo alcançar formas mais gerais de paralelismo. Podemos perceber melhor isso visualizando a Figura 2.7.

Quando se trata dos conectores entres os componentes, em geral, os modelos de componentes suportam conectores pré-definidos, geralmente assimétricos, do tipo cliente/servidor. No entanto, em computação paralela, faz-se necessário a existência de diversos de tipos de conectores que se adequam a cada situação particular na aplicação [27]. Uma das vantagens do uso do modelo # é que tanto os componentes como os conectores são considerados como componentes-#, de *espécies* distintas. Isso pode ser inferido pelo fato de que um mesmo componente # pode

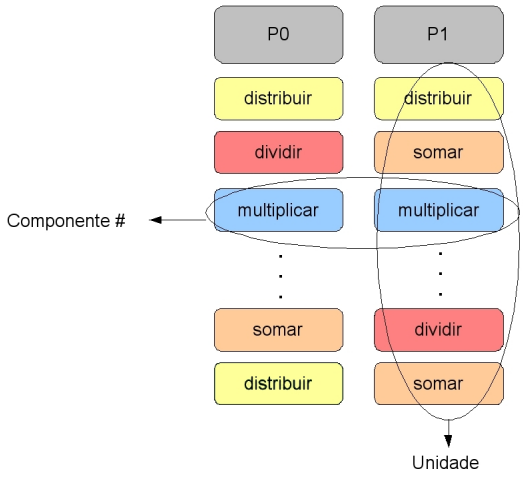


Figura 2.7: Um componente #

estar implantado em máquinas distintas, onde suas unidades desempenham o papel do conector, vide Figura 2.8. Sendo assim, podemos utilizar diversos tipos de conectores, definidos pelo usuário. Conectores simétricos, que descrevem iterações entre processos onde estes atuam como pares, podem ser facilmente especificados.

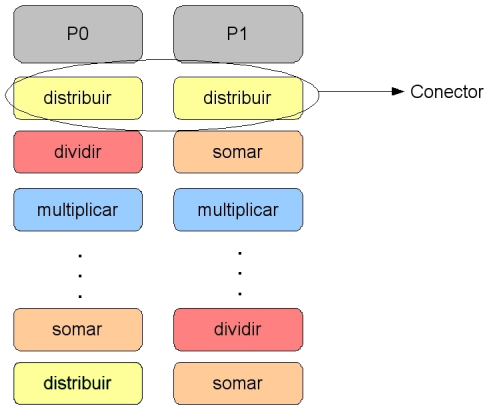


Figura 2.8: Conector como um componente #

2.6.1 Um exemplo prático

Para melhor entendermos, exemplificamos o modelo # através de um exemplo bem simples e prático. Para tal, usamos o algoritmo conhecido como Crivo de Eratóstenes para gerar os números primos menores ou iguais à n. Tomando uma lista ordenada com os números de 2 até n, eliminam-se todos os números compostos que

são múltiplos dos primos menores ou iguais a $\sqrt{n}$. Isso é feito da seguinte maneira: encontra-se o primeiro elemento e eliminam-se todos os seus múltiplos. Em seguida, encontra-se o segundo elemento, e eliminam-se todos os seus múltiplos. Tal processo é repetido enquanto o elemento cujos múltiplos devem ser removidos for menor ou igual a $\sqrt{n}$. Os números que sobrarem são necessariamente números primos. Por exemplo, para obter todos os primos de 1 até 25, basta eliminar todos os múltiplos de 2,3 e 5, pois esses são os primos menores ou iguais a $\sqrt{25}$. Na Figura 2.9, que ilustra esse exemplo, os números marcados são os números compostos eliminados.

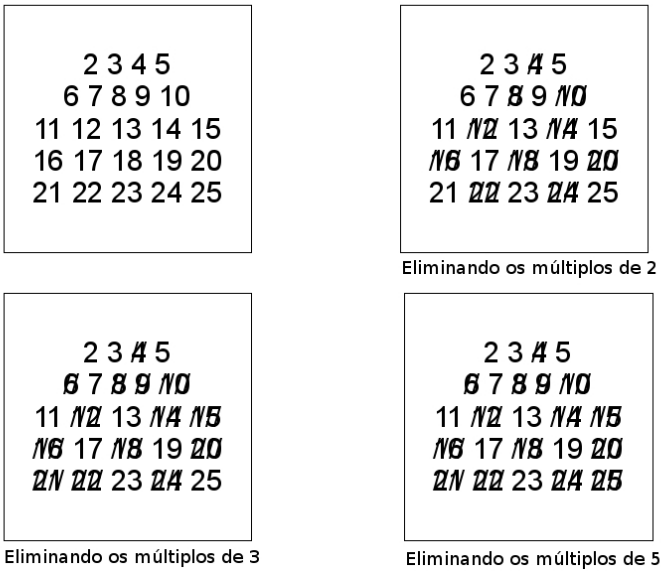


Figura 2.9: Crivo de Eratóstenes

Seguindo o exemplo, desejamos saber os números primos menores ou iguais a 25 de forma paralela, usando para isso a abordagem de componentes # com 3 processadores. Para isso, cada processo deve receber uma parcela da lista de números, ficando também responsável por marcar os números compostos da sua própria lista. Já o processo *root* fica responsável por distribuir as parcelas da lista entre os demais processos e por enviar o próximo número primo, em que seus múltiplos devem ser eliminados, ao restante dos processos. Dessa forma, o processo P0, através do método $distribuir(2, 25)$, distribui a lista de números para os processos P1 e P2, onde P0 fica com os números $2, 3, 4, 5, 6, 7, 8, 9$; P1 fica com os números $10, 11, 12, 13, 14, 15, 16, 17$; e P2 fica com os números $18, 19, 20, 21, 22, 23, 24, 25$. Após isso, P0 encontra o número primo 2, através do método $encontrarProxPrimo()$, e envia-o para os demais processos pelo método $enviarPrimo()$, então todos os processos executam paralelamente a eliminação dos

múltiplos de 2, através do método *eliminarMulti*(). Tal procedimento, de envio e eliminação, se repete para os números primos 3 e 5. Por fim, cada processo deve enviar o seu resultado ao processo P0, para gerar a lista final com os números primos.

Tratando a solução desse exemplo através do modelo de componentes #, podemos perceber, através da Figura 2.10, que existem componentes # que se comportam como conectores. É o caso do componente # formado pelas unidades correspondentes às fatias distribuir(2,25). Podemos perceber também, um outro componente # responsável pela computação propriamente dita, é o caso do componente # formado pelas unidades correspondentes às fatias eliminarMult(2).

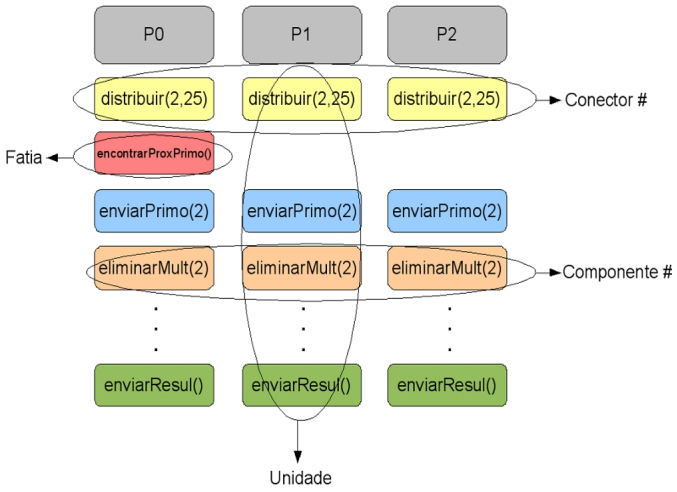


Figura 2.10: Uma abordagem usando o modelo de componentes # de uma solução paralela do Crivo de Eratóstenes

2.6.2 Sistemas de Programação

O modelo de componentes # não define a natureza concreta do componente-#, restringindo-se a definir a natureza de suas unidades de forma abstrata, como unidades de software de alguma forma conhecida, daí o uso do termo "unidade". O modelo # define apenas que os componentes devem ser formados por *unidades* e que estas devem estar implantadas nos diferentes nós da plataforma de execução paralela, e define também que os componentes podem ser compostos por sobreposição para formar outros componentes ou uma nova aplicação.

A definição da natureza concreta dos componentes # é realizada através dos sistemas de programação implementados sobre o modelo #, chamados de *Sistemas de Programação #*. Para isso, esses devem definir *espécies* de componentes de acordo com o ambiente de computação pretendido, onde uma espécie de componentes é um

conjunto de componentes # que compartilham características em comum, como o modelo de implantação, características do ciclo de vida e restrições de composição com outros componentes de sua própria espécie ou espécies distintas. Por exemplo, o sistema de programação # denominado HPE, que está descrito na Seção 2.8, suporta espécies de componentes destinados a programação paralela de propósito geral [30]. Um outro exemplo seria a criação de espécies de componentes para fins específicos, como a integração de bibliotecas científicas para a solução de sistemas lineares, que é a proposta deste trabalho.

2.7 O Paralelismo em Modelos de Componentes

Para lidar com o paralelismo nos modelos de componentes, especularam-se algumas ideias, como admitir que um único processo lide com as comunicações entre os componentes, tornando-se claramente um gargalo capaz de afetar significamente o desempenho e escalabilidade de um sistema. Outra proposta simples seria impor que todas as comunicações fossem realizadas através do *framework*, ou plataforma. No entanto, essa abordagem acarretaria a necessidade de modificações em códigos existentes, a maioria dos quais fazem uso de bibliotecas de trocas de mensagens padrões, como o MPI, o que não é uma tarefa nada trivial. Então, uma abordagem que se tornou comum baseia-se no princípo de que os próprios componentes paralelos devem escolher o padrão de comunicação desejado [74], de forma ortogonal a própria plataforma de componentes.

As primeiras experiências práticas de suporte ao paralelismo em modelos de componentes aconteceram com o CORBA, o qual, como mencionado anteriormente, suporta apenas componentes sequenciais, possivelmente distribuídos. Dentre as propostas de suporte a componentes paralelos no CORBA, nas quais um componente CORBA encapsula um programa paralelo, destacamos o PARDIS(*PARallel DIStributed applications*) [59], o PaCO(*Parallel CORBA Object*) [75] e o PaCO++ [34], os quais introduziram a noção de objetos paralelos. Enquanto no PARDIS um objeto paralelo é composto de *threads*, o PaCO trata o paralelismo encapsulando o MPI em um componente, onde o código MPI é visto como um objeto CORBA. Por fim, o PaCO++ traz uma extensão do PaCO para suportar a portabilidade. Ainda sobre o modelo CORBA foi desenvolvida a extensão GridCCM [74] para suportar o uso de componentes paralelos. Para tanto, para não precisar modificar o modelo, foi incluído um arquivo XML que descreve as operações paralelas e a distribuição dos argumentos dessas operações. Devemos ainda destacar o Data Parallel CORBA [70],

padrão proposto pela OMG para o paralelismo de dados, no estilo SCMD, usando CORBA.

Tratar o paralelismo nos modelos de componentes para CAD é uma questão bastante investigada. Um dos principais pontos discutidos é a forma com que os componentes se comunicam. De forma geral, nos modelos de componentes existentes, a comunicação entre os componentes segue o modelo de interação cliente-servidor, onde ora o componente se comporta como servidor oferecendo serviço aos demais componentes, ora se comporta como cliente requisitando os serviços de outros componentes. Dessa forma, o paralelismo é conseguido pela ação conjunta de um grupo de componentes distribuídos. No entanto, esse padrão de comunicação não é suficiente para ser empregado em aplicações de CAD, uma vez que aplicações desse nicho requerem não apenas esse modelo de comunicação, mas diversas outras formas de comunicação [30], onde os processos envolvidos atuam como pares cooperativos, e não como clientes ou servidores.

Embora o modelo de componentes CCA seja destinado para o desenvolvimento de aplicações de CAD, o paralelismo não é definido na sua especificação de componentes, deixando cargo dos desenvolvedores de *frameworks* compatíveis com o CCA a possibilidade de implementar a estratégia de suporte ao paralelismo desejada. Espera-se assim que a experiência adquirida possa guiar os projetistas da especificação CCA em direção a uma especificação geral de paralelismo no futuro. Por exemplo, no *framework* CCAffeine [4] o paralelismo é disponibilizado através do encapsulamento da sincronização por meio dos componentes, onde o paralelismo é tratado com um interesse ortogonal aos *frameworks*. Para isso, introduz o estilo SCMD (*Single Component Multiple Data*), onde um componente paralelo é definido por um conjunto de componentes iguais que formam um regimento e comunicam-se através de passagem de mensagens usando alguma interface conhecida para essa finalidade, como o MPI. Cada membro do regimento é um componente implantado em um dos nós de um cluster. Membros de regimentos diferentes dentro de um mesmo processador, compartilhando espaço de endereçamento, podem comunicar-se por meio de portas CCA convencionais. É dessa forma que dois componentes paralelos implantados em um *cluster* podem ser ligados por portas.

O *framework* DCA generaliza a noção de paralelismo proposta pelo CCAffeine, de forma que componentes paralelos são programas MPI que podem estar implantados em subconjuntos distintos (sobrepostos ou disjuntos) dos nós de processamento de um *cluster*. Para a intermediação da comunicação entre componentes paralelos que

possuem diferentes números de processos, o DCA implementa uma forma de PRMI (Parallel Remote Method Invocation) sobre o MPI, com poder expressivo suficiente para programação de acoplamentos $M{\times}N$ entre componentes paralelos. Apesar disso, o DCA impõe algumas restrições, motivadas por aspectos técnicos, na forma como os componentes paralelos podem interagir, especialmente por não haver um comunicador MPI comum entre os processos dos componentes cliente e servidor. Além disso, ao exigir a participação dos próprios componentes na redistribuição dos dados quase sempre necessária nos acomplamentos $M{\times}N$, aumenta-se o grau de acoplamento entre os componentes, pois ambos, do lado cliente ou do lado servidor, devem conhecer detalhes sobre a distribuição de dados no outro lado. Tal característica é indesejável, por promover a dependência entre os componentes.

Alguns esforços da comunidade CCA tem sido concentrados na especificação do suporte ao paralelismo MCMD (*Multiple Component Multiple Data*), o qual generalizaria o conceito de SCMD suportado pelo CCAffeine [1] permitindo que times de componentes distintos sejam lançados em subconjuntos do conjunto total de nós de processamento de um computador paralelo distribuído, sobrepostos ou disjuntos. Documentos que descrevem as especificações CCA para o suporte a MCMD, bem como a implementação de um protótipo de prova de conceito, podem ser obtidos no *wiki* oficialmente mantido pela comunidade CCA para essa finalidade[1].

Como descrito na Seção 2.4, o paralelismo no modelo de componentes Fractal se realiza por meio das *interfaces* dos componentes que ao se ligarem permitem a requisição de recursos entre os componente conectados. Já o modelo GCM, descrito na Seção 2.5, extensão do modelo Fractal, o paralelismo é tratado pelas mesmas *interfaces*, mas com a adição de interfaces coletivas, denominadas *gathercast* e *broadcast*, para tratar comunicação $1{\times}M$, $M{\times}1$ e $M{\times}N$. De fato, trata-se do paradigma de objetos paralelos distribuídos, onde a comunicação é realizada através de PRMI, herdando os problemas destacados com a abordagem DCA.

Uma observação que vale a pena ressaltar é que tais abordagens ainda não alcançaram o nível de expressividade da programação de passagem de mensagem, como o MPI, o qual permite a descrição de quaisquer padrões distribuídos de comunicação entre processos paralelos.

No HPE, por se tratar de um sistema de programação baseado no modelo #, o paralelismo é visto como uma característica inerente aos componentes, visto que estes são constituídos por unidades implantadas em nós distintos de um *cluster*,

[1] https://www.cca-forum.org/wiki/tiki-index.php?page=MCMD-WG

ainda passíveis de composição hierárquica. Tal característica põe o HPE em um nível de suporte ao paralelismo superior aos demais *frameworks* e plataformas baseados em CCA, Fractal e GCM, suportando o poder expressivo de troca de mensagens com MPI. Para demonstrar isso, um trabalho recente tem compatibilizado o HPE com o padrão CCA, constituindo um *framework* CCA capaz de conciliar a distribuição e o paralelismo de componentes sem necessidade de PRMI, através do uso de componentes que implementam as ligações CCA [26]. Essa elevação ao nível de componentes das ligações CCA torna possível a implementação específica de uma ligação baseada em um conhecimento prévio das características do *cluster* alvo e das estruturas de dados transmitidas através das portas CCA, promovendo conexões mais eficientes entre componentes. Outra característica importante desse *framework*, herdada do HPE, é o suporte a composição recursiva (hierárquica), suportada pelo GCM mais não presente de forma direta no modelo CCA.

2.8 HPE: Hash Programming Environment

Visando o suporte a aplicações de computação intensiva sobre plataformas de *cluster computing*, surgiu a plataforma de desenvolvimento e execução de componentes paralelos HPE (*Hash Programming Environment*), definido de forma a atender os requisitos de um sistema de programação # de propósito geral. Obedecendo aos requisitos necessários para ser um sistema de programação #, o HPE tem sido desenvolvido desde o ano de 2005 por pesquisadores pertencentes ao grupo de pesquisa ParGO (*Parallelism, Graphs and Optimization*)[2] da Universidade Federal do Ceará [30]. Seu protótipo mais recente encontra-se publicamente disponível em http://hash-programming-environment.googlecode.com, com o objetivo de experimentos de prova de conceito por parte do grupo de pesquisa.

No HPE, os componentes # possuem os seguintes estágios no seu ciclo de vida:

- *Descoberta*, onde o desenvolvedor requisita os componentes # necessários para composição de sua aplicação ou outro componente de forma transparente;
- *Configuração*, onde os componentes # escolhidos são configurados de acordo com a necessidade do desenvolvedor, a fim de compor novos componentes ou uma aplicação final;
- *Publicação*, onde o programador disponibiliza um componente # por ele desenvolvido para usuários potenciais;

[2] http://www.lia.ufc.br/~pargo/

- *Implantação*, onde o desenvolvedor implanta o componente # em alguma plataforma de computação paralela a fim de que este seja executado ou ligado a uma aplicação em um momento futuro;
- *Produção*, onde o sistema encontra-se disponível para a sua execução ou ligação a uma aplicação, bem como para o seu monitoramento.

Para honrar o modelo de componentes # o sistema de programação HPE deve atender a três requisitos:

- suportar a implantação e execução distribuída das unidades dos componentes #;
- suportar a composição hierárquica dos componentes # por sobreposição;
- oferecer espécies de componentes para programação paralela de propósito geral sobre *clusters*.

2.9 Espécies de Componentes

Para oferecer o suporte ao desenvolvimento de aplicações paralelas e distribuídas de propósito geral, o HPE disponibiliza as seguintes espécies de componentes:

- *Arquiteturas*, que descrevem plataformas de execução paralelas;
- *Ambientes*, que representam interfaces de *software* utilizadas para suporte ao paralelismo sobre a plataforma, como bibliotecas de passagem de mensagem e *middlewares* de suporte ao paralelismo;
- *Computações*, que implementam computações paralela, onde as unidades representam um papel funcional do processo;
- *Estruturas de Dados*, que representam as estruturas de dados paralelas processadas no curso das computações;
- *Sincronizadores*, que representam padrões de comunicação e sincronização entre os processos;
- *Aplicações*, que descrevem as aplicações finais, que podem ser executadas no *cluster* como um programa paralelo;

- *Qualificadores*, que descrevem propriedades de componentes, as quais podem afetar o desempenho, semântica ou funcionalidades suportadas por estes;
- *Enumeradores*, que especificam uma quantidade finita arbitrária de unidades e componentes aninhados de uma configuração.

2.10 A Arquitetura Hash

A arquitetura Hash é constituída por três serviços: o *Front-End*, o *Back-End* e o *Core* [32], como ilustrado na Figura 2.11.

O *Front-End* é voltado para os usuários de componentes, interessados em construir aplicações ou outros componentes. Através desse serviço, é possível construir componentes # possivelmente pela combinação de outros componentes #, obtidos do repositório distribuído mantido por um serviço *Core*, por sobreposição. Os componentes desenvolvidos podem ser também submetidos a um serviço *Core* para armazenamento e posterior recuperação por um usuário interessado. Por fim, através do *Front-End*, um usuário pode ainda implantar, instanciar, e executar um componente em uma plataforma de execução paralela (*cluster*), por meio do serviço *Back-End*. Enfim, é através do *Front-End* que o ciclo de vida dos componentes # é controlado, o que é realizado através dos serviços *Back-End* e *Core*, cujas finalidades foram reveladas pela descrição.

O *Back-End* é uma *infraestrutura de componentes paralelos*, o qual expõe uma interface externa para implantação, instanciação e execução de componentes #. Mais detalhes sobre a implementação do *Back-End* são fornecidos adiante.

Embora o HPE seja um ambiente de programação para aplicações paralelas de propósito geral, fazendo uso de seus serviços seria possível implementar *Cores* e *Back-Ends* específicos para algum tipo de problema, ao modo dos PSE's (*Problem-solving environments*) e linguagens de domínio específico (DSL). Podemos entender por PSE como "um sistema que provê todas as facilidades computacionais necessárias para resolver alguma classe de problema por meio da seleção automática ou semi-automática de métodos de solução, como também a incorporação facilitada de novos métodos de solução" [47]. São exemplos de PSE's o NetSolve e o WebFlow. Assim, podemos criar *Cores* voltados para a solução de sistemas lineares, solução de problemas de otimização, dentre outras possibilidades. Podemos ainda desenvolver espécies de componentes específicas para um determinado domínio, abordagem que é adotada na contribuição deste trabalho.

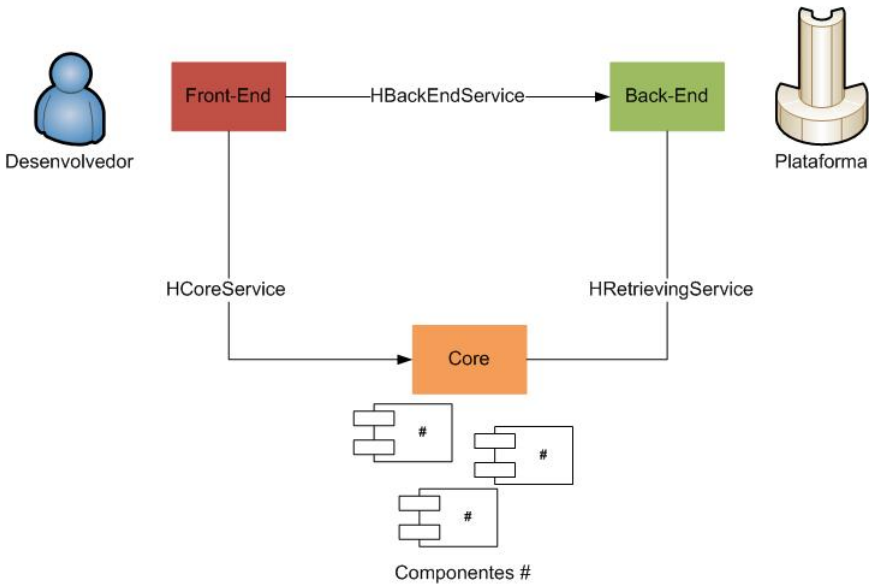


Figura 2.11: Arquitetura HPE

Ao se utilizar de interfaces, o *Front-End*, o *Back-End* e o *Core* podem interagir de maneira transparente no que diz respeito à arquitetura e à linguagem de programação, permitindo assim que o desenvolvedor possa eleger, dentre vários *Cores* e *Back-Ends* disponíveis, aqueles que mais se adequam às suas necessidades. Para isso, as interfaces entre o *Front-End* e o *Core* e entre o *Front-End* e o *Back-End* são implementadas utilizando Web Services [5], os quais permitem total transparência a respeito da linguagem na qual os componentes que formam o HPE são implementados, bem como de sua localização através de um endereço internet.

2.11 HTS: Sistema de tipos para componentes

O HTS (*Hash Type System*) é um sistema de tipos desenvolvido para suprir as necessidades de componentes # em ambientes de computação de alto desempenho, o qual foi adotado pelo HPE. Define dois tipos de componentes: os *componentes abstratos* e os *componentes concretos*, estes últimos também referidos como componentes-#.

Um componente abstrato define um contrato que deve ser obedecido pelos componentes concretos que o implementam para diferentes *contextos*. O contexto de um componente abstrato é definido por um conjunto de *parâmetros formais de contexto*, cada qual formado por uma *variável de contexto* e um componente abstrato (limite superior) para restringir os tipos de componentes abstratos possíveis para substituir a variável. Dessa forma, um componente abstrato pode ser descrito como

$C[X_1 <: T_1, X_2 <: T_2, ..., X_n <: T_n]$, onde C é o nome do componente abstrato, $X_1, X_2, ..., X_n$ são variáveis de contexto, e $T_1, T_2, ..., T_n$ são os seus respectivos *limites*, definidos por componentes abstratos aplicados a um contexto.

Um componente concreto é uma implementação de um certo componente abstrato para um certo contexto específico, definido pela substituição das variáveis de contexto por componentes abstratos que satisfaçam as relações de subtipo definidos pelos limites das variáveis. Devem implementar o interesse definido pelo componente abstrato. Através do contexto, os componentes concretos podem especializar um componente abstrato de acordo com suas necessidades. Por exemplo, podem definir a arquitetura para a qual o componente é otimizado, o ambiente de passagem de mensagens usado para a comunicação entre suas unidades, o método numérico usado para resolver um certo sistema linear, dentre outras possibilidades. Dessa forma, podem existir várias versões de componentes concretos vindo de um mesmo componente abstrato, com a condição de que exista apenas um componente concreto para cada *contexto* implantado em um certo ambiente mantido por um *Back-End*. Portanto, um componente concreto é descrito como uma implementação para uma *instanciação*, definida por $C[S_1, S_2, ..., S_n]$, onde C representa um componente abstrato $C[X_1 <: T_1, X_2 <: T_2, ..., X_n <: T_n]$ e $S_1, S_2, ..., S_n$ representam componentes abstratos que substituirão as variáveis de contexto. Para isso, devem satisfazer as restrições $S_1 <: T_1, S_2 <: T_2, ..., S_n <: T_n$.

O HTS oferece suporte a resolução de componentes aninhados de um componente-# composto dinamicamente, baseado na ideia da resolução de contexto. Uma vez que componentes aninhados são sempre tipados por *instanciações* (componente abstrato aplicado a um contexto específico), na fase de inicialização de uma aplicação, durante sua execução, é buscado um componente-# que satisfaça aquela instanciação no ambiente de componentes implantados na plataforma, o qual é gerenciado pelo *Back-End*. Quando encontrado, este componente será carregado e ligado ao componente aninhado. Caso contrário, é executado um procedimento de resolução, o qual busca generalizar os parâmetros de contexto da instanciação até que um componente-# que implemente uma versão mais genérica da instanciação original seja encontrada, carregada e ligada ao componente aninhado. Caso todas as generalizações possíveis sejam pesquisadas e não seja encontrado um componente-# compatível, uma exceção de erro de ligação é lançada, abortando a execução do componente. Esse procedimento é executado recursivamente a partir do componente-# aplicação, uma vez que a carga de um componente-# ligado a um

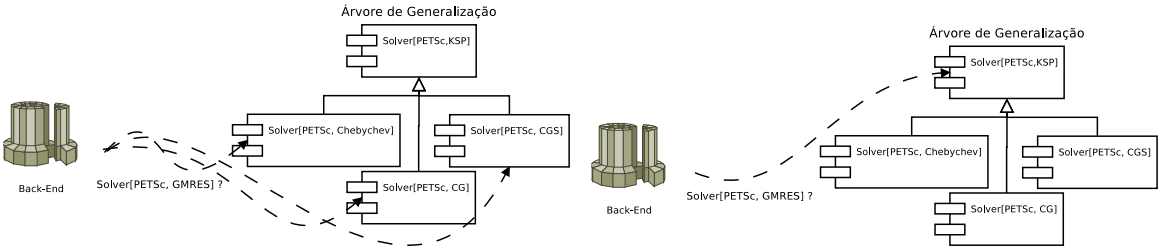
componente aninhado causa a tentativa de resolução dos componentes aninhados do componente-# recentemente carregado.

Para melhor esclarecer o HTS, descrevemos um exemplo simples. Seja o componente abstrato $Solver[X <: Library, Y <: MethodType]$ para solucionar sistemas lineares, onde o primeiro parâmetro representa a biblioteca científica utilizada e o segundo parâmetro o método numérico utilizado. Dessa forma, podemos especializar o componente abstrato *Solver* substituindo suas variáveis X e Y por subtipos dos componentes abstratos *Library* e $MethodType$, respectivamente. Portanto, o componente concreto $Solver[PETSc, GMRES]$ é uma *instanciação* do componente abstrato $Solver[X <: Library, Y <: MethodType]$ especializada para solucionar sistemas lineares utilizando a biblioteca científica PETSc com o método GMRES (*Generalized Minimal Residual Method*).

Exemplificaremos então a resolução dinâmica de componentes. Vamos apresentar essa noção por meio do exemplo da composição do componente abstrato $Solver[X <: Library, Y <: MethodType]$ e da recuperação de sua instanciação para o *contexto* $X = PETSc$ e $Y = GMRES$. Inicialmente o desenvolvedor, por meio do *Front-End*, instancia o componente abstrato *Solver* com os componentes abstratos *PETSc* e *GMRES* como *parâmetros de contexto* atuais. Em execução, o *Back-End* irá verificar no ambiente se existe um componente concreto para o dado contexto, como ilustrado na Figura 2.12(a). Caso não encontre o componente específico, o sistema irá procurar generalizar o contexto buscando um outro componente mais genérico compatível aos *limites* de subtipos de componentes estabelecidos nos parâmetros (Figura 2.12(b)). Por exemplo, o componente $Solver[PETSc, KSP]$ poderia ser escolhido para um *contexto* mais genérico com a biblioteca PETSc e o método iterativo KSP (*Krylov Subspace Methods*) que tem por subtipos mais conhecidos os métodos GMRES e CG (Gradiente Conjugado), dentre outros. Ainda não encontrando um componente concreto para o novo *contexto* generalizado, o sistema irá mais uma vez generalizar o contexto. Tal passo é repetido até o *limite* de subtipos de componentes ser alcançado. No caso da biblioteca PETSc, o componente *KSP* seria o limite para o método de solução. Enfim, não encontrando nenhum componente adequado é retornado um erro em tempo de execução.

2.12 A Implementação

Por ser compatível com o modelo de componentes #, os componentes participantes do HPE podem ser subdivididos em partes, chamadas unidades, que



(a) Busca de componente (b) Generalização do contexto

Figura 2.12: Resolução dinâmica de componentes

são implantadas em diferentes nós do ambiente paralelo. Além disso, os componentes podem ser compostos por sobreposição.

Para tanto, a versão atual do HPE implementa as funcionalidades dos seus serviços *Front-End*, *Core* e *Back-End* como descrito a seguir.

2.12.1 Implementação do Front-End

Como dito, o ciclo de vida dos componentes # é gerenciado através do *Front-End*. É interessante que essa manipulação possa ser feita através de interfaces gráficas. Para tanto, o *Front-End* do HPE foi implementado em Java utilizando o GEF (*Graphical Editing Framework*), sendo portanto apresentado como um *plug-in* para a plataforma Eclipse [67], voltado à configuração visual de componentes #. No entanto, a especificação de configurações de componentes # na forma textual, em alternativa à configuração visual, também é possível, por meio da linguagem HCL (*Hash Configuration Language*).

2.12.2 Implementação do Core

O *Core* foi implementado também em Java, como um repositório distribuído de configurações de componentes #, sobre um servidor Apache Tomcat [21]. De fato, um serviço *Core* agrega um conjunto de *locations*, acessíveis por meio de um endereço *web*, as quais armazenam componentes # organizados em pacotes. Assim, fornece ao *Front-End* uma visão uniforme dos componentes armazenados nas *locations*.

2.12.3 Implementação do Back-End

Como mencionando anteriormente, existem algumas características necessárias para o bom desempenho de uma infraestrutura de componentes para CAD, como o suporte a tipos de dados comuns em CAD e a interoperabilidade entre as linguagens de programação.

A plataforma CLI (*Common Language Infrastructure*) oferece um padrão para

infraestruturas de componentes que se adequam aos requisitos citados para CAD. CLI oferece o controle de versão *side-by-side* dos componentes, suporta o uso de ponteiros de forma segura e oferece a compilação *Just-in-Time* associado a possibilidade de pré-compilação de componentes computacionalmente intensivos para execução em modo nativo, aproximando assim o desempenho da execução do componente à execução do código nativo. O Mono [58], por ser uma implementação livre do padrão CLI, é empregado na implementação do *Back-End*.

O Mono faz uso do GAC (*Global Assembly Cache*), que tem por função armazenar os componentes compartilhados. Quando um componente requisitado não for encontrado no diretório da aplicação, ele é buscado no GAC. O GAC trata apenas os componentes implantados em uma mesma máquina. No entanto, os componentes # podem encontrar-se distribuídos em vários computadores. Dessa forma, é necessário uma extensão do GAC para trabalhar com componentes distribuídos para assim se adequar ao modelo de componente #. Com essa finalidade, o *Back-End* implementa o DGAC (*Distributed Global Address Cache*), o qual gerencia um conjunto de GAC's que estão instalados um em cada nó do *Cluster*. O gerenciamento é realizado por um conjunto de processos, chamados *workers*, que são controlados por um processo mestre que recebe requisições do *Front-End* para implantar e executar um componente #. Para manter as informações sobre os componentes (configuração e localização das unidades) é utilizado um banco de dados distribuído MySQL.

2.13 Estudo de caso

Para demonstrar a utilização do HPE apresentamos uma configuração de componentes # para solução do problema de integração numérica multidimensional. O problema é descrito matematicamente da seguinte forma:

$$\int \int \cdots \int_D f(x_1, x_2, \ldots, x_n) dx_1 dx_2 \ldots dx_n$$

O problema de integração numérica múltipla pode ser decomposto em problemas menores ao dividir o intervalo de integração em um número finito de subintervalos e aplicar em cada um dos intervalos algum método de quadratura numérica, como o método de Monte Carlo e o método de Romberg. O problema decomposto é possivelmente paralelizável. Para tanto, pode-se utilizar o paradigma mestre-escravo, onde um processo mestre divide e distribui os intervalos entre os

Parâm. de Contexto	Variável	Comp. Aninhado	Instanciação (Tipo do Componente Aninhado)
input_type	***I***	**input**	DATA
scatter_strategy	***S***	**scatter**	DISTRIBUTE[*environment_type*=***E***, *input_type*=***I***, *output_type*=***J***]
job_type	***J***	**job**	DATA
work_type	***W***	**work**	FUNCTION[*input_type*=***J***, *output_type*=***R***]
result_type	***R***	**result**	DATA
gather_strategy	***G***	**gather**	COMBINE[*environment_type*=***E***, *input_type*=***R***, *output_type*=***O***]
output_type	***O***	**output**	DATA
environment_type	***E***	**environment**	ENVIRONMENT

Tabela 2.1: Componentes aninhados de FARM

vários outros processos escravos, e então, cada processo escravo realiza a integração do intervalo confiado e envia o resultado ao processo mestre que é responsável por receber e somar todos os resultados para a obtenção do resultado final da integração múltipla.

Para a implementação desse caso de uso foi utilizada uma tradução da biblioteca NINTLIB [24] para C# paralelizada usando o MPI.NET, realizada por nosso grupo de pesquisa. Inicialmente, foi construído o componente # *FARM* como um esqueleto do paradigma mestre-escravo. Os componentes que fazem parte da configuração de FARM encontram-se apresentados na Tabela 2.1. No caso particular de FARM, cada componente aninhado está associado a um tipo (instanciação) que é também parâmetro de contexto da configuração. A seguir a descrição da função de cada um desses componentes aninhados na configuração de FARM:

- o **environment**, que define a interface de troca de mensagens usada para implementar a comunicação entre o mestre e os escravos.
- o **input**, que define a estrutura de dados de entrada do problema;
- o **scatter**, que *particiona* a estrutura de dados de entrada em um conjunto de *jobs*, e os *distribui* entre os processos escravos;
- o **job**, que define a estrutura de dados que define um *job*, entrada para os processos, ou unidades, trabalhadores;
- o **work**, que define a computação realizada por um escravo sobre cada *job* a fim de produzir um resultado que será enviado de volta ao mestre;
- o **result**, que define a estrutura de dados de saída dos processos escravos;
- o **gather**, que agrega todos os resultados calculados pelos processos escravos em um resultado final e o envia ao processo mestre;

- o **output**, que define a estrutura de dados do resultado final;

Para definir uma quantidade arbitrária de processos escravos no problema foi integrado um enumerador ao componente. O componente *FARM* está representado pela Figura 2.13, onde as elipses tracejadas representam os componentes aninhados privados, as elipses com linha cheia representam os componentes aninhados públicos, os retângulos representam as unidades, o circulo representa o enumerador e as setas representam o fluxo de dados.

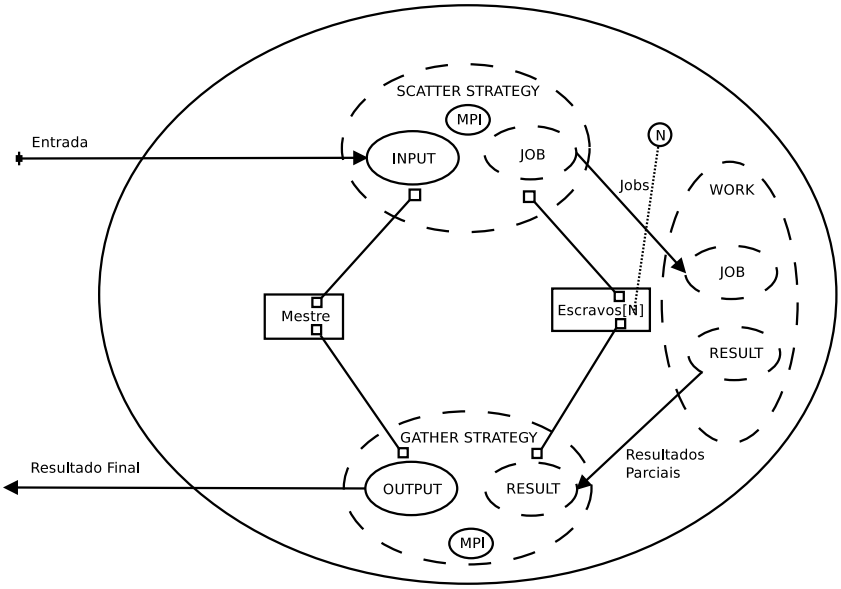


Figura 2.13: Configuração do componente *FARM*

Por fim, foi configurado o componente abstrato RombergIntegrator, com o contexto [*integrating_function*=***F***:PrimitiveFunction], o qual reutiliza o componente Farm, configurando-o como componente aninhado para o contexto do problema de integração numérica múltipla. Para a especificação do contexto (instanciação) é necessário suprir os parâmetros de contexto de Farm, da seguinte forma:

- o parâmetro *input_type* é suprido pelo componente abstrato IntegralCase, no contexto [*integrating_function*=***F***], que define uma função a ser integrada. A variável de contexto ***F*** define a função primitiva a ser integrada, configurada como parâmetro de contexto de *RombergIntegrator*, cujo identificador é também *integrating_function*;

- o parâmetro *scatter_strategy* é suprido pelo componente abstrato DistributeInterval, aplicado ao contexto [*input_type*=IntegralCase,

output_type=List[*element_type*=IntegralCase]], o qual define como os intervalos de integração serão divididos e distribuídos entre os processos escravos;

- o parâmetro *job_type* é suprido por uma lista cujos elementos são do tipo definido pelo componente abstrato IntegralCase, representado por List[*element_type*=IntegralCase];
- o parâmetro *work_type* é suprido pelo componente abstrato ApproximateIntegral, aplicado ao contexto [*input_type*=IntegralCase, *output_type*=Double], que computa a integração numérica múltipla sobre a função dada utilizando o método de *Romberg*;
- o parâmetro *result_type* é suprido por uma lista cujos elementos são do tipo definido pelo componente abstrato Double, representado por List[*element_type*=Double];
- o parâmetro *gather_strategy* é suprido pelo componente abstrato SumIntegrals, aplicado ao contexto [*input_type*=List[*element_type*=Double], *output_type*=Double], o qual soma os resultados das integrações sobre os intervalos;
- finalmente, o parâmetro *output_type* é suprido pelo componente abstrato Double.

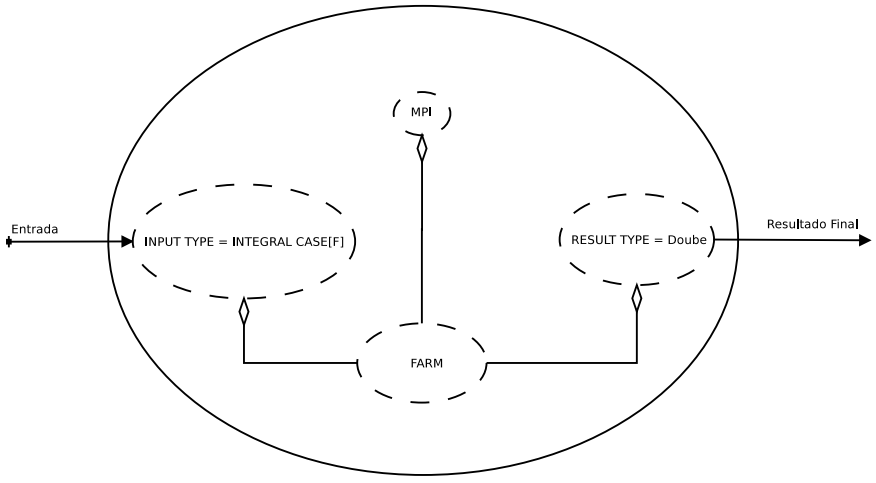


Figura 2.14: Configuração do componente *FARM*

	Número de Dimensões (n)											
	4				5				6			
P	T_{bind}	T_{par}	S	E	T_{bind}	T_{par}	S	E	T_{bind}	T_{par}	S	E
1	2,4s	6,9s	0,8	83,0%	2,4s	161,3s	0,8	82,6%	2,4s	3744s	0,8	84,9%
2	2,6s	3,6s	1,6	80,4%	2,6s	80,7s	1,7	82,5%	2,6s	1875s	1,7	84,7%
4	2,9s	1,9s	3,0	75,1%	2,9s	41,6s	3,2	80,1%	2,9s	963s	3,3	82,5%
8	4,7s	1,1s	5,3	66,8%	4,7s	21,3s	6,3	78,2%	4,7s	499s	6,4	79,5%
	$T_{seq} \approx 5,8s$				$T_{seq} \approx 133,1s$				$T_{seq} \approx 3181s$			

Tabela 2.2: Perfil de Desempenho do HPE - Integração Numérica Multi-Dimensional

O componente abstrato RombergIntegrator foi finalmente implementado pelo componente # **RombergIntegratorImpl** para o contexto [*integrating_function*=TestFunction]. Portanto este componente # irá ser carregado e ligado a uma aplicação que tentar realizar uma integração sobre a função definida pelo componente abstrato TestFunction, usado em nosso estudo de caso.

A utilização do componente Farm como componente aninhado e sua especificação para contexto de integração numérica múltipla pode ser observado pela Figura 2.14.

2.13.1 Avaliação de Desempenho do HPE

A aplicação de integração numérica multi-dimensional foi utilizada para realizar uma avaliação de desempenho da plataforma de componentes paralelos implementada pelo *Back-End* do HPE. Os resultados são apresentados nas Tabelas 2.2 e 2.3.

A Tabela 2.2 mostra resultados para a versão HPE em configurações onde a função foi integrada em respectivamente 4, 5 e 6 dimensões. Quanto maior o número de dimensões, muito maior é a intensidade computacional do problema de integração a ser resolvido. Os resultados mostraram boa escalabilidade, como

	Número de Processadores (P)							
	1		2		4		8	
metric	HPE	MPI.NET	HPE	MPI.NET	HPE	MPI.NET	HPE	MPI.NET
média	160,6s	148,4s	80,5s	74,3s	41,7s	38,1s	21,3s	19,7s
mediana	**161,2s**	**148,8s**	**80,6s**	**74,4s**	**41,6s**	**38,1s**	**21,3s**	**19,7s**
mínimo	142,5s	130,0s	73,4s	66,1s	38,6s	36,7s	20,3s	18,9s
máximo	173,9s	164,8s	86,5s	80,7s	44,7s	39,9s	22,3s	22,0s
desvio padrão	6,34s	5,44s	2,94s	2,47s	1,24s	0,65s	0,39s	0,39s

Tabela 2.3: Comparação entre HPE e C#/MPI.NET - Integração Numérica Multi-Dimensional (n=5)

	Número de Dimensões (n)		
	6	7	8
seq	4.84s	89.4s	1621s
1	4.98s (×1.0)	89.7s (×1.0)	1628s (×1.0)
2	2.50s (×1.9)	44.9s (×2.0)	814s (×2.0)
4	1.25s (×3.9)	22.7s (×3.9)	408s (×4.0)
8	0.64s (×7.6)	11.5s (×7.8)	205s (×7.9)

Tabela 2.4: Execução Nativa - C++ - Integração Numérica Multi-Dimensional

esperado, a partir de 4 dimensões. A tabela compara ainda o tempo total de computação paralela (T_{par}), sobre até 8 processadores, com o tempo de ligação dos componentes (T_{bind}), bem como com o tempo de execução sequencial (T_{seq}), e apresenta ainda o *SpeedUp* (S) e a Eficiência (E). O tempo de ligação mostra-se constante independente do número de processadores e tamanho do problema (n), o que garante boa escalabilidade da solução baseada em componentes para problemas que justificam o paralelismo, uma vez que a ligação de componentes, por serem paralelos, é o único peso significativo da implementação. A justificativa para o aumento do tempo no caso de 8 processadores é que só foi possível replicar a base de dados do DGAC em até 4 processadores, devido a restrições do MySQL-cluster, evidenciando que essa replicação é relevante para o desempenho total do processo de ligação.

Na Tabela 2.3, a versão HPE é comparada com uma versão escrita puramente em C#/MPI.NET, não baseada em componentes, para o caso $n = 5$. Os resultados mostram um impacto pouco significativo do uso dos componentes, em comparação com os benefícios advindos da sua utilização. Pode-se observar que para um mesmo número de processadores o acréscimo de tempo é praticamente constante, tal adição de tempo é justificada pelo carregamento dos componentes-# para sua execução.

Apresentamos ainda, na Tabela 2.4, os resultados da integração numérica em sua versão nativa, em Fortran, a partir da qual foi implementada versão C#. Os resultados evidenciam uma grande disparidade entre a versão nativa e a versão executada sobre máquina virtual (C#/Mono), no que concerne ao desempenho bruto.

Essa última constatação é bastante relevante no contexto do trabalho dessa dissertação, pois justifica o encapsulamento de bibliotecas científicas nativas e pré-paralelizadas para que programas HPE possam desfrutar do desempenho dos computadores paralelos modernos.

Capítulo 3

Bibliotecas Científicas

A capacidade de reutilização durante o desenvolvimento de um *software* tornou-se algo fundamental para que novos *softwares* sejam desenvolvidos mais rapidamente, com melhor qualidade e menor custo. Vários são os benefícios ao utilizar esse conceito, e quando se trata de aplicações de CAD isso se torna crucial. Desenvolver aplicações de CAD é bastante complexo, haja vista que essas aplicações fazem uso de modelos matemáticos complexos. Além disso, é também necessário o gerenciamento de todas as funcionalidades que executam paralelamente, a fim de se ter um bom balanceamento de carga, evitar impasses, gargalos e outras eventuais falhas no sistema. Portanto, requer não somente o conhecimento específico da área, mas também um vasto conhecimento em computação paralela. Utilizar técnicas que possibilitem a reutilização de partes de *softwares* é essencial para a eficiência e para a produtividade no desenvolvimento dessas aplicações.

Um artefato bastante utilizado na implementação de aplicações de CAD e que fornece o proveito da reutilização é o uso de bibliotecas científicas, geralmente voltadas para oferecer funcionalidades úteis para as ciências computacionais, como solucionar um sistema linear, multiplicar matrizes, dentre outros.

Utilizar bibliotecas científicas na construção de uma aplicação tráz não só o benefício da reutilização, mas alguns outros. Um dos principais que podemos citar é que tais bibliotecas já foram inúmeras vezes testadas nos mais variados ambientes, dando assim uma maior possibilidade da corretude do algoritmo. A ACM (*Association for Computing Machinery*) é uma das organizações que testam e disseminam algumas dessas bibliotecas. Outro benefício importante é que também é pressuposto que tais bibliotecas utilizem o melhor algoritmo conhecido para um problema determinado. Uma outra vantagem é que algumas dessas bibliotecas

científicas encapsulam a comunicação entre os processos, melhorando assim a produtividade do desenvolvedor e evitando possíveis erros nos códigos paralelos. Outro proveito para ampliar a velocidade de execução das aplicações é que algumas bibliotecas foram desenvolvidas para uma arquitetura específica buscando usufruir de todo os recursos computacionais disponíveis da arquitetura. Por outro lado, existem também algumas bibliotecas que se adequam a vários tipos diferentes de arquiteturas, possibilitando assim a portabilidade da aplicação entre arquitetura distintas ou mais atuais. Portanto, fazer uso de bibliotecas científicas, bem testadas e conceituadas, além de ter uma grande melhoria na produtividade do desenvolvimento das aplicações, é um caminho para se ter uma aplicação sem erros e eficiente.

Um exemplo de iniciativa histórica que evidencia a importância do uso de artefatos de reuso de software por parte da comunidade científica foi o CRPC (*Center for Research on Parallel Computation*)[1], projeto financiado pela NSF (*National Science Foundation*), principal agência de fomento à pesquisa dos EUA, envolvendo participantes da academia e indústria entre os anos de 1989 e 2000, com o objetivo de tornar a programação paralela mais fácil e acessível para os cientistas, realizando pesquisas nos *softwares* e algoritmos para usufruir o máximo possível dos computadores paralelos disponíveis. O efeito dessas pesquisas foi o surgimento de novos *softwares* e a publicação de novos algoritmos para serem usados pela comunidade científica. Nesses novos avanços são considerados, durante o desenvolvimento, os requisitos de portabilidade, flexibilidade e programabilidade. Dentre essas novas tecnologias, podemos citar novos padrões de linguagem, bibliotecas que encapsulam os melhores algoritmos, e ferramentas para ajudar no desenvolvimento e depuração de software paralelo.

No entanto, a implementação de bibliotecas científicas paralelas é algo extremamente difícil, devido à complexidade extra associada ao controle de concorrência e à distribuição de dados. Como estamos tratando de aplicações paralelas, esses dois pontos são cruciais no desenvolvimento. Quanto à concorrência, dois ou mais processos podem ser executados corretamente sozinhos, mas podem não ter o mesmo resultado quando são executados concorrentemente, devido a problemas relacionados à interferência entre processos [16]. Quanto à distribuição de dados, quando mal realizada, pode levar a problemas no desempenho e no cálculo do resultado final. Por exemplo, se uma biblioteca para resolver sistemas lineares está

[1] http://www.crpc.rice.edu/

esperando uma matriz em que os dados são armazenados por diagonal em um vetor, mas na verdade recebe uma matriz armazenada de outra forma, então a biblioteca ou perde drasticamente o desempenho, devido à redistribuição dos dados, ou no pior dos casos é executada incorretamente. Para agravar a situação, há ainda a questão da portabilidade, pois eram oferecidas diferentes interfaces de programação, o que tornava ainda mais difícil a programação paralela para os desenvolvedores, exigindo trabalhos dispendiosos de conversão da aplicação paralela entre diferentes arquiteturas. Tamanha era a complexidade que, até bem pouco tempo, poucos eram os casos de sucesso de bibliotecas científicas paralelas. Um exemplo é a ScaLAPACK [18], ainda assim, um sucesso bastante limitado frente ao seu potencial.

Fruto das pesquisas realizadas pelo CRPC, o MPI (*The Message Passing Interface*) [39] e o HPF (*High Performance Fortran*) [46] foram duas das principais contribuições do grupo para se conseguir um padrão de interface de programação paralela entre diferentes arquiteturas.

Através de interfaces bem definidas, o MPI fez com que as bibliotecas científicas paralelas passassem a ser escritas de forma mais eficiente, devido à comunicação ser encapsulada, e passassem também, a ser portáveis entre diversas arquiteturas. Outro fato que ajudou o surgimento de uma nova geração de bibliotecas científicas paralelas foi a criação de técnicas de engenharia de software que permitiram que as bibliotecas pudessem chamar algumas operações sem levar em consideração como os dados estão distribuídos, podendo sofrer impacto apenas no desempenho. Com esses meios, surgiu uma nova linhagem de bibliotecas científicas paralelas, que podem ser mais facilmente reutilizadas e compostas.

Outra questão importante para a reutilização de software é a catalogação. Há pouco tempo atrás, não era uma tarefa tão fácil, em aplicações científicas, saber qual algoritmo utilizar e onde obtê-lo. Existem inúmeros algoritmos científicos, podendo até mesmo um cientista especialista perder um tempo considerável buscando a melhor solução entre livros, artigos e *softwares*. Foi com o intuito de sanar esse problema que os laboratórios *AT&T Bell Laboratories, Oak Ridge National Laboratory* e a Universidade de Tennessee, juntamente com alguns outros colaboradores organizaram o Netlib[2], com o propósito de oferecer um repositório livre de *softwares*, documentos de interesse para a computação científica. Para garantir a segurança da informação, a coleção do repositório é replicada e sincronizada em diversos servidores do mundo inteiro. Outro projeto importante,

[2]http://www.netlib.org

é o GAMS (*Guide to Available Mathematical Software*[3]), desenvolvido pelo NIST (*National Institute of Standards and Technology*), que visa fornecer para os cientistas e engenheiros um melhor acesso aos componentes de *softwares* para utilizar em modelagens matemáticas e em análise estatística. Além de fornecer um repositório *online* dos componentes, o projeto permite também buscas baseadas no nome das bibliotecas, no nome das subrotinas, em palavras-chave e em tópicos da computação científica. Pode-se também utilizar filtros na pesquisa para refinar a preferência, como a precisão da computação ou a linguagem de programação. Apesar de todos esses esforços, ainda pode ser difícil se encontrar o algoritmo desejado ou até obtê-lo sem documentação.

Uma outra questão essencial diz respeito a problemas que não podem ser resolvidos com as bibliotecas existentes, ou, quando o são, não são capazes de utilizar de maneira satisfatória os recursos de processamento e memória. Nesses casos, é muitas vezes necessária a adaptação das bibliotecas para o problema determinado. Entretanto, isso não é uma tarefa fácil, pois pode ser preciso um grande número de alterações, o qual se soma ao desafio da eficiência e da corretude da biblioteca adaptada. Portanto, é conveniente a disponibilidade de ferramentas que ajudem o usuário a escolher o melhor algoritmo e, caso seja necessário, também adaptá-lo.

3.1 Principais domínios de bibliotecas científicas paralelas

Diversas são as áreas que se beneficiam do uso de bibliotecas científicas paralelas. Dentre esses domínios, podemos citar alguns de maior relevância, como bibliotecas voltadas para resolver sistemas lineares e não-lineares, bibliotecas desenvolvidas para resolver equações diferenciais parciais, bibliotecas para problemas de otimização e outras para problemas de autovalores. Todas esses domínios e outros muitos que não foram citados são de grande importância para a solução de alguns problemas computacionais que exigem um alto poder de processamento, justificando assim, quando possível, o uso de bibliotecas científicas paralelas.

Para este trabalho, nos restringiremos, como estudo de caso, apenas ao domínio de solução de sistemas de equações lineares.

3.1.1 As Bibliotecas Científicas e a Álgebra Linear

Grande parte das bibliotecas científicas é voltadaa para cálculos da Álgebra Linear, devido ao fato de a maioria dos problemas, tanto das engenharias quanto da

[3]http://gams.nist.gov

própria computação científica, exigir soluções de problemas da Álgebra Linear, que incluem desde uma simples inversão de matriz até os mais complexos problemas, de forma particular, a solução de sistemas lineares.

Entende-se como sistema linear um conjunto de n equações lineares com m incógnitas, em outras palavras o sistema $Ax = b$. Diversos são os métodos inclusos nas bibliotecas científicas para solucionar tal problema. Entre alguns mais difundidos, podemos citar o GMRES (*Generalized Minimum Residual*) e o BiCG (*Biconjugate gradient*) [49].

A própria comunidade da Álgebra Linear tem se empenhado em ajudar no desenvolvimento dos algoritmos das bibliotecas científicas. Um desses esforços é a criação de um padrão para as operações mais básicas da Álgebra Linear. É desejável que elas sejam implementadas de forma eficiente nas mais variadas arquiteturas de computadores disponíveis, sem desprezar a portabilidade das operações entre os diversos fabricantes.

Um resultado desse esforço foi a criação da interface de programação BLAS (*Basic Linear Algebra Subprograms*) [63], que contém rotinas básicas de operações entre vetores e matrizes. A BLAS é subdivida em três níveis. O primeiro nível abrange as operações de escalar, vetor e vetor-vetor. Já o segundo nível, contém as operações matriz-vetor. Por fim, o terceiro nível contém as operações matriz-matriz. Por conta de implementações da BLAS serem consideravelmente eficientes, portáveis e livres, essa interface é bastante utilizada no desenvolvimento de bibliotecas avançadas de álgebra linear, como a LAPACK (*Linear Algebra PACKage*) [7]. Caso o usuário deseje utilizar a BLAS de forma ainda mais otimizada, ele pode usar o ATLAS (*Automatically Tuned Linear Algebra Software*) [93] para gerar automaticamente uma versão otimizada da BLAS para sua própria arquitetura, ou ainda utilizar alguma biblioteca matemática desenvolvida especificamente para sua própria arquitetura e que contenha a BLAS. Por exemplo, podemos mencionar a ACML (*AMD Core Math Library*) [3] para os processadores AMD e a Intel MKL (*Intel Math Kernel Library*) [55] para os processadores Intel.

3.2 Portabilidade e Eficiência

Podemos perceber através da existência de várias implementações de acordo com a arquitetura para a biblioteca BLAS que um dos pontos que mais influenciam o tempo de execução de um algoritmo avançado de álgebra linear é a arquitetura do computador. Para se conseguir um bom desempenho, um dos principais cuidados

é diminuir o máximo possível a frequência com que os dados são transferidos entre os níveis de hierarquia da memória. Para tanto, é importante que os elementos que são utilizados sejam armazenados consecutivamente, evitando assim trocas desnecessárias de dados na memória.

Em uma matriz, há duas formas de armazenar seus elementos consecutivamente na memória, por linhas ou por colunas. Na linguagem de programação Fortran, os elementos da coluna na matriz são armazenados consecutivamente, enquanto na linguagem C são as linhas que são armazenadas consecutivamente. Assim, se desejamos implementar em C um algoritmo eficiente para trabalhar com matrizes é fundamental que seus elementos sejam percorridos linha a linha, potencializando um uso mais eficiente das múltiplas hierarquias de memória *cache* presentes pelos computadores modernos.

Além de escolher o tipo de armazenamento adequado para a matriz é importante alocar os elementos mais referenciados na memória cache. Por exemplo, se desejamos realizar o produto entre uma matriz $A \in \mathbb{R}^{m \times n}$ e um vetor $x \in \mathbb{R}^n$. A operação $y = Ax + y$ pode ser computada da seguinte forma:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j + yi \qquad i = 1 : m$$

Podemos observar que os elementos de y e x são os mais utilizados durante a computação. Então, podemos alocar os elementos de y e x na memória *cache*. Dessa forma, esses elementos só são carregados uma única vez e estão sempre disponíveis, proporcionando assim ganhos no desempenho. Separar a matriz por blocos também diminui a frequência de troca de dados na memória, trazendo portanto ganhos de desempenho.

Essas e outras características da arquitetura são consideradas na implementação das bibliotecas científicas, como as variantes da interface BLAS e as bibliotecas EISPACK [81], Linpack [60], LAPACK, e ScaLAPACK que foram desenvolvidas sobre a interface BLAS. A EISPACK e a LINPACK foram desenvolvidas para computadores superescalares. A LAPACK foi desenvolvida para que a EISPACK e a LINPACK funcionasse de forma eficiente em computadores com memória compartilhada. Já a biblioteca ScaLAPACK é uma extensão da LAPACK para computadores com memória distribuída [42].

Como vimos, os computadores estão ficando cada vez mais rápidos. No entanto, os *softwares* nem sempre conseguem utilizar o máximo dos recursos dos mais rápidos computadores disponíveis. Para se beneficiar dos novos computadores, é importante que a biblioteca seja portável. Para isso, a biblioteca deverá ser escrita em linguagem padrão, como o Fortran e C, sem fazer suposições sobre o ambiente e utilizar padrões de interfaces de programação como o MPI, OpenMP e HPF. No entanto, existe o outro lado da moeda. Como garantir uma eficaz portabilidade sem sacrificar o desempenho? Dispôr de desempenho junto com portabilidade é um desafio, devido à diversidade de arquiteturas existentes. Todavia, o que se sabe é que o melhor é garantir a portabilidade, mas sem desconsiderar a eficiência [38], uma vez que se as máquinas evoluem rapidamente e temos bibliotecas portáveis, podemos então utilizá-las nos mais atuais computadores. Portanto, atingiríamos o desempenho desejado rapidamente, não importando a arquitetura.

Um projeto importante nesta área, e que visa a portabilidade e o desempenho é a coleção ACTS (*Advanced CompuTational Software*), discutido na próxima seção.

3.2.1 ACTS Collection

ACTS Collection [44] é um conjunto de ferramentas para auxiliar a escrita de novos programas científicos de alto desempenho. Grande parte dessas ferramentas foram desenvolvidas pelos laboratórios do DOE (*U.S Department of Energy*) juntamente com algumas universidades. Teve como seu primeiro objetivo acelerar o uso e o desenvolvimento de programas de alto desempenho pelo DOE para problemas críticos. No entanto, hoje não tem apenas o intuito de serem usadas nos próprios programas do DOE, mas também é motivada para serem usadas fora dele. Essas ferramentas são geralmente bibliotecas escritas em C, C++ e Fortran para executarem em ambientes de computadores paralelos com memória distribuída, usando para isso o MPI.

Existem dois importantes requisitos durante o projeto e a implementação das ferramentas pertencentes a ACTS Collection: a portabilidade e o desempenho. Outro benefício é que não é preciso pertencer ao DOE para se conseguir informações úteis da ACTS Collection, como o procedimento para instalação, a manutenção e o suporte. Tais informações são amplamente divulgadas tanto dentro do DOE como fora dele.

A ACTS Collection é subdivida em quatro categorias: **ferramentas numéricas**, **ferramentas para desenvolvimento de códigos**, **ferramentas para execução**

do código e **ferramentas para desenvolvimento de bibliotecas**. As três primeiras categorias contêm ferramentas e aplicações para usuário final. Já a última não é destinada ao usuário final da ACTS Collection e é também pobremente documentada.

A categoria **ferramentas numéricas** contém bibliotecas que implementam alguns algoritmos numéricos padrões, tanto de forma serial como de forma paralela. Entre esses algoritmos, existem rotinas para solução de sistemas lineares, de equações diferenciais ordinárias (EDO), equações diferenciais parciais(EDP), dentre outros. As bibliotecas existentes são: Aztec [11], PETSc [12], SUNDIALS [52], Hypre [45], ScaLAPACK [18], SuperLU [33], OPT++ [65], SLEPc [51] e TAO [86]. Em geral, as bibliotecas dessa categoria podem ser chamadas a partir de programas escritos em C, C++ e Fortran.

A categoria **ferramentas para desenvolvimento de código** contém ferramentas que oferecem uma infraestrutura para gerenciar algumas tarefas complexas da programação paralela. As únicas que fazem parte dessa categoria são: Global Arrays [69] e Overture [14]. Por exemplo, através da ferramenta Global Arrays podemos desenvolver mais facilmente programas paralelos com vetores distribuídos entre os vários processadores.

A categoria **ferramentas para execução do código** contém ferramentas para auxiliar o suporte durante o tempo de execução da aplicação, como visualização remota e análise do desempenho. Até o momento, só existem duas ferramentas: TAU (*Tuning and Analysis Utilities*) [77] e CUMULVS [61]. Por exemplo, ao incluir algumas macros da ferramenta TAU em programas escritos em C, C++, Fortran e Java, podemos analisar a performance da aplicação.

A categoria **ferramentas para desenvolvimento de bibliotecas** contém ferramentas para o uso no desenvolvimento de outras bibliotecas. A única existente é a já mencionada ATLAS [93], que pode gerar automaticamente outras ferramentas numéricas otimizadas para as mais variadas arquiteturas e compiladores. Está focada nas operações nível 3 da interface de programação BLAS e em algumas operações que exigem um alto grau de otimização da biblioteca LAPACK.

3.3 Bibliotecas Científicas para Solução de Sistemas Lineares

Existe uma vasta quantidade de bibliotecas científicas para solucionar sistemas lineares. Neste trabalho, como estudo de caso, restringiremo-nos às bibliotecas PETSc, SuperLU e Hypre.

3.3.1 SuperLU

A biblioteca científica SuperLU [33] foi desenvolvida na linguagem de programação C, podendo também ser chamada em Fortran. Foi implementada pela divisão de ciência da computação da universidade de Berkeley e pelo NERSC (*National Energy Research Scientific Computing Center*[4]). Tem como objetivo principal oferecer chamadas de métodos para solucionar sistemas lineares assimétricos com matrizes esparsas. Para isso, ela inicialmente realiza uma fatoração LU. Após este passo, a partir de uma matriz triangular, ela obtêm a solução através do método da substituição.

A biblioteca é subdividida em três partes, de acordo com a arquitetura alvo: SuperLU, SuperLU_MT e SuperLU_DIST. Se o interesse do desenvolvedor for construir aplicações para executar sequencialmente, então ele deverá usar o subconjunto SuperLU. Se o interesse for executar paralelamente com memória compartilhada, então ele deverá usar o subconjunto Super_MT. Já se o interesse for executar sobre máquinas paralelas com memória distribuída, então ele deverá usar o subconjunto SuperLU_DIST.

A convenção de chamadas de métodos existentes na biblioteca SuperLU segue o mesmo padrão das bibliotecas Lapack e ScaLapack para as rotinas computacionais (por exemplo, fatoração) e para as rotinas de solução. Na chamada $xmmop(a1...an)$, x indica o tipo de dado que pode ser utilizado, mm indica o tipo da matriz, op indica a operação, $a1...an$ indicam os n argumentos. Pode-se incluir a letra p antes do tipo de dado, quando é desejado que a operação seja paralela. Por exemplo, o método *pdgssv* é um solucionador paralelo para sistema lineares com matriz do tipo genérica com o tipo de dado *double* (números de ponto flutuante de dupla precisão). No entanto, a convenção difere para as rotinas utilitárias, que servem para ajudar ao usuário criar e destruir matrizes mais facilmente. Por exemplo, o método $dCreate\_CompCol\_Matrix$ é usado para criar uma matriz armazenada por colunas.

Queremos apresentar os passos básicos necessários para a construção de um programa para solucionar sistemas lineares utilizando a biblioteca SuperLU. Enfatizamos que em nenhum momento desejamos dissecar todos os passos possíveis, pelo fato de existirem vários métodos e portanto várias combinações possíveis. Para tanto, apenas destacamos os passos essenciais para tal desenvolvimento. Para isso,

[4] http://www.nersc.gov/

construímos um diagrama de atividades utilizando a notação UML. Essa notação é utilizada nos demais diagramas de atividade subsequentes.

Considerando que se deseja implementar uma aplicação paralela com memória distribuída para solucionar sistemas lineares, devemos seguir os seguintes passos fundamentais, descritos na Figura 3.1: primeiramente é preciso inicializar o MPI e então mapear os processos existentes em uma malha de processos 2D. Após isso, é preciso criar a matriz A, criar o vetor b e ajustar as opções de como o sistema será resolvido. Depois, deve-se escolher o solucionador desejado e chamá-lo. Por fim deve-se finalizar o ambiente MPI, a malha de processos e liberar o armazenamento.

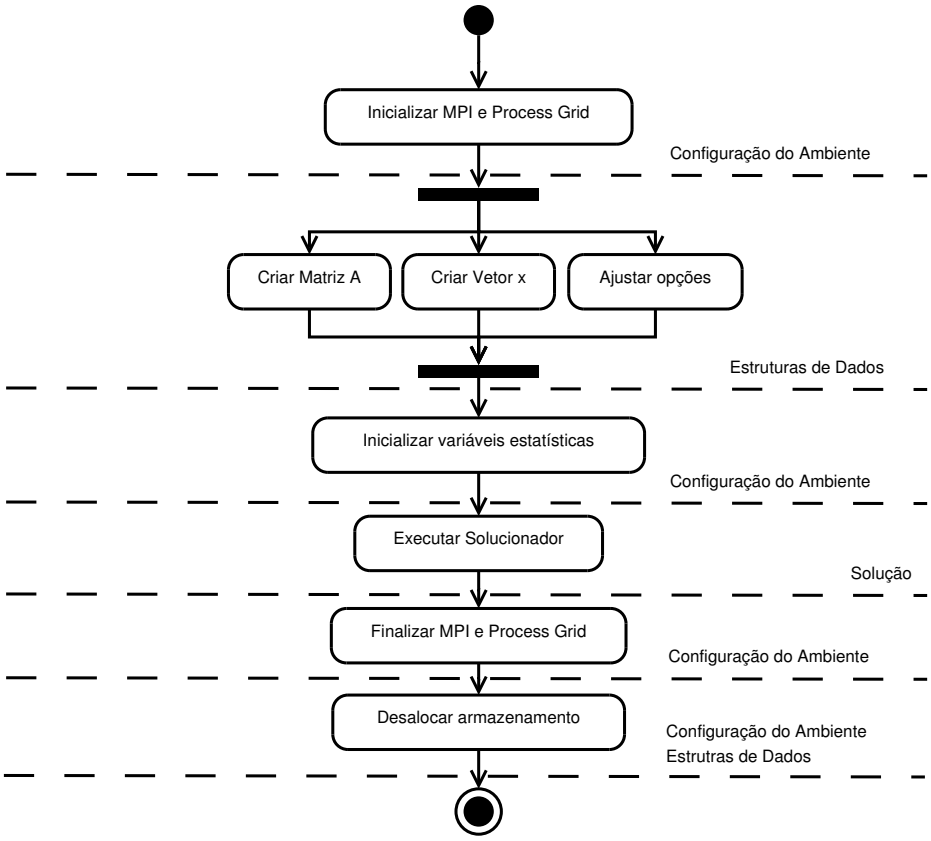


Figura 3.1: Diagrama de atividade para resolver sistemas lineares Ax=b com a biblioteca SuperLU

3.3.2 Hypre

A biblioteca Hypre [45] foi desenvolvida pelo CASC (*Center for Applied Scientific Computing*[5]) do Lawrence Livermore National Laboratory, com o objetivo de fornecer avançados pré-condicionadores paralelos para solucionar grandes

[5]https://computation.llnl.gov/casc/

sistemas lineares com matrizes esparsas. A documentação da Hypre cita ainda outras características importantes da biblioteca, como possuir pré-condicionadores extensíveis, métodos iterativos de subespaços de Krylov, interfaces intuitivas para a criação e manipulação de estruturas de dados, etc. É aplicável tanto em C como em Fortran.

Nas aplicações implementadas sobre a biblioteca Hypre, o desenvolvedor deve saber escolher com qual interface irá trabalhar, entre quatro interfaces possíveis: Struct, SStruct, FEI e IJ. A Struct é apropriada para executar em grades estruturadas. A SStruct é apropriada para executar em grades que são praticamente estruturadas, mas com apenas alguns elementos não estruturados. Já FEI é apropriada para usuários que obtém o sistema linear da discretização de elementos finitos. Por fim, a IJ é apropriada para as demais aplicações em que as outras interfaces não se compatibilizam.

A biblioteca Hypre segue uma convenção de chamadas de métodos própria, variando de acordo com a interface utilizada. Por exemplo, para a criação de uma matriz, pode-se chamar os métodos $HYPRE\_StructMatrixCreate$, $HYPRE\_SStructMatrixCreate$ e $HYPRE\_IJMatrixCreate$, quando se utiliza as interfaces Struct, SStruct e IJ, respectivamente. Existem, no entanto, outros métodos que não explicitam em sua chamada a interface utilizada, como exemplo, a chamada do solucionador $HYPRE\_BoomerAMGSolve$.

Seguindo o mesmo pensamento usado ao apresentar o diagrama de atividades da biblioteca SuperLU, queremos também apresentar apenas os passos fundamentais para solucionar sistemas lineares utilizando a biblioteca Hypre, sendo omitidos muitos outros métodos que poderiam ser incluídos, como métodos de pré-condicionamento.

Para desenvolver uma aplicação paralela, com memória distribuída e utilizando a interface Struct para solucionar sistemas lineares utilizando a biblioteca Hypre, devemos seguir os seguintes passos, vide figura 3.2: inicialmente é preciso criar, ajustar e montar a estrutura da grade. Depois criar e ajustar o objeto Stencil. Após isso, é preciso criar, inicializar, ajustar e montar a matriz A, o vetor b e o vetor x. Em seguida, é necessário criar, ajustar e chamar o solucionador desejado. Depois, pode-se obter algumas informações da solução. Por fim, é preciso desalocar o armazenamento.

3.3.3 PETSc

A biblioteca científica PETSc [12] tem como principal objetivo oferecer um conjunto de ferramentas escaláveis para solução de EDPs e outros problemas relacionados. Foi desenvolvida pelo *Argonne National Laboratory*[6] com o intuito de ser extensível, onde os usuários poderiam adaptar os solucionadores e as estruturas de dados, como poderiam também se comunicar, através de interfaces, com outras ferramentas, como BlockSolve95, ESSL, Matlab, ParMeTis, PVODE, and SPAI. Ela pode ser utilizada no desenvolvimento de aplicações nas linguagens C, C++, Fortran e mais recentemente em Python. Outras características importantes são: oferece rotinas para utilização de vetores e matrizes paralelas sobre MPI, é bem documentada e está em constante desenvolvimento de novas funcionalidades e otimização do desempenho.

Existem várias aplicações que fazem uso da biblioteca PETSc. Uma bastante interessante é a criação de uma infraestrutura computacional para simulação de cirurgia a laser no tratamento de câncer [36].

Como na biblioteca Hypre, a PETSc segue uma convenção de chamadas de métodos própria, porém mais intuitiva para a identificação dos objetivos dos métodos. Como podemos perceber pelos métodos $VecCreate$, $MatCreate$, $PetscPrintf$, $PetscInitialize$, que tem como objetivo criar um vetor, criar uma matriz, imprimir tipos de dados da biblioteca e inicializar um programa com PETSc, respectivamente.

Da mesma forma como nos diagramas de atividades anteriores, neste, queremos destacar apenas os passos básicos para solucionar um sistema linear. Desta vez, utilizando a biblioteca PETSc.

Para construir uma aplicação paralela para solucionar um sistema linear utilizando a biblioteca PETSc, podemos atender os seguintes passos, vide figura 3.3: primeiramente é preciso inicializar o programa PETSc. Depois, deve-se criar, ajustar as opções, setar valores e montar a matriz A. Deve-se também criar, ajustar as opções do vetor x e criar, ajustar as opções e valores do vetor b. Após isso, é preciso criar o objeto do solucionador, ajustar as operações do solucionador, ajustar as opções do solucionador e executá-lo. Em seguida, deve-se desalocar o armazenamento e, por fim, deve-se finalizar o programa.

[6] http://www.anl.gov/

3.3.4 Classificação das Subrotinas - PETSc × Hypre × SuperLU

Para a construção da solução proposta neste trabalho foi realizado um estudo intensivo sobre as bibliotecas científicas do domínio de solução de sistemas lineares. Um desses esforços foi a catalogação das subrotinas mais relevantes das principais bibliotecas. Para tanto, inicialmente, foi observado que em geral as bibliotecas desse domínio possuem os mesmos passos básicos para a obtenção da solução. Observando os Diagramas de Atividades 3.1, 3.2 e 3.3, podemos classificar os passos em passos de Configuração do Ambiente, passos de Estruturas de Dados e passos de Solução.

Para melhor entendermos a classificação das subrotinas foi produzida uma listagem com as subrotinas mais relevantes do domínio. Para isso, ao observamos que em geral as subrotinas das bibliotecas científicas do domínio de solução de sistemas lineares seguem a mesma classificação, resolvemos, sem perda de generalidade, incluir na listagem apenas as subrotinas das bibliotecas científicas destacadas nesse trabalho, a SuperLU, a Hypre e a PETSc. Por facilitar a legibilidade da listagem, decidimos decompô-la em quatro tabelas de acordo com sua classificação, são elas a Tabela 3.1 que lista as subrotinas da classe de Configuração do Ambiente, as Tabelas 3.2 e 3.3 que contêm as subrotinas da classe de Estruturas de Dados e a Tabela 3.4 que traz as subrotinas da classe de Solução. As subrotinas existentes em dada biblioteca estão marcadas com o símbolo $\surd$ na coluna referente à biblioteca a qual pertence.

Ao observarmos nossa listagem podemos constatar algumas semelhanças e diferenças entre as bibliotecas do domínio. Inicialmente, verificamos que em geral as bibliotecas realizam as mesmas operações, pouco variando de acordo com o nível de abrangência desejado da biblioteca. Notamos assim a existência de algumas subrotinas idênticas nas três bibliotecas, como a subrotina *Executar solver* da Classe Solução. Percebemos também a existência de algumas subrotinas que pertencem apenas a uma biblioteca, como a subrotina *Definir operandos* da Classe Solução.

No entanto, a partir do fato de uma subrotina não existir em uma biblioteca não podemos concluir que a biblioteca não realize o procedimento contido na subrotina ausente. Por exemplo, a definição dos operandos, realização da subrotina *Definir operandos* que existe somente na biblioteca PETSc, é realizada por subrotinas mais genéricas pertencentes as demais bibliotecas. Para esse exemplo, no caso da biblioteca SuperLU, os operandos são definidos através da subrotina *Executar solver*, já no caso da bibliteca Hypre, os operandos são definidos pela subrotina *Criar solver*.

Dessa forma, podemos concluir a semelhança entre as bibliotecas do domínio de

Tabela 3.1: Subrotinas da Classe Configuração de Ambiente da biblioteca SuperLU, Hypre e PETSc

Função	SuperLU	Hypre	PETSc
Inicializar ambiente			√
Finalizar ambiente			√
Setar opções padrão	√		
Inicializar controle estatístico	√		
Finalizar controle estatístico	√		
Inicializar grid	√	√	
Finalizar grid		√	
Definir extensões no Grid		√	
Montar Grid		√	
Definir variáveis do Grid		√	
Adicionar variáveis do Grid		√	
Definir relacionamento entre partes fora do Grid		√	
Adicionar um parte não estruturada		√	
Criar Stencil		√	
Destruir Stencil		√	
Definir Stencil		√	
Criar graph		√	
Destruir graph		√	
GraphSetStencil		√	
Adicionar non-stencil graph ao índice		√	
Montar Graph		√	

Tabela 3.2: Subrotinas da Classe Estruturas de Dados(Vetor) da biblioteca SuperLU, Hypre e PETSc

Função	SuperLU	Hypre	PETSc
Criar Vector		√	√
Inicializar Vector		√	
Definir valores Vector		√	√
Adicionar valores		√	√
Montar Vector		√	√
Duplicar vetor			√
Definir tipo de armazenamento		√	
Definir como complexo		√	
Definir tamanho do vetor			√
Definir opções do vetor			√
Definir todos os elementos com o mesmo valor			√
Retornar referência		√	
Retornar valores		√	√
Retornar o intervalo de índices pertencentes a este processador			√
Retornar numero de elementos do vetor			√
Retornar o tamanho do vetor			√
Imprimir Vector		√	√
Destruir Vector		√	√

solução de sistemas lineares no que diz respeito à realização de tarefas, e a diferença no que diz respeito à granularidade das subrotinas, tendo a biblioteca SuperLU a granularidade mais grossa entre as bibliotecas elegidas neste trabalho e a biblioteca PETSc a granularidade mais fina. A produção dessa listagem de subrotinas muito nos enriqueceu para justificar as decisões tomadas na implementação deste trabalho.

3.4 Integração de Bibliotecas Científicas em Infraestruturas de Componentes

Durante desenvolvimento de aplicações de CAD, os benefícios de produtividade, eficácia, eficiência e portabilidade são notórios quando se utiliza bibliotecas científicas. No entanto, tal abordagem não atinge alguns requisitos importantes do desenvolvimento de aplicações de CAD. Dois requisitos de CAD que podemos citar que não são atingidos através de bibliotecas científicas são: a necessidade de diversas trocas de solucionadores da aplicação com o mínimo possível de mudanças no código fonte; e a necessidade da integração entre aplicações.

Como sabemos, o paradigma de programação orientada a componentes permite a troca dinâmica de componentes em tempo de execução, como também suporta a integração entre aplicações através de modelos de componentes bem definidos. Dessa forma, podemos perceber que através do uso de componentes de *softwares* é possível alcançar os requisitos pendentes das bibliotecas científicas. Isso é possível ao se encapsular as aplicações e os solucionadores através de componentes de *softwares* e publicar suas funcionalidades através de interfaces padrões.

Trabalhos para a criação de interfaces padrões para bibliotecas científicas e sua apresentação como componentes de *softwares* já foram realizados. É o caso do ESI (*Equation Solver Interface*), do TSC, do CCA LISI (*LInear Solver Interface*), do Numerical Platon e da biblioteca SPARSKIT como componente CCA. A abordagem desses trabalhos e os seus resultados foram utilizados como base e comprovação do nosso trabalho proposto. Mais detalhes a respeito das motivações da apresentação de bibliotecas científicas como componentes de *softwares* serão abordados no Capítulo 4.

3.4.1 ESI

A interface ESI foi desenvolvida pelo DOE com participação de universidades e com o apoio de indústrias. O projeto foi escrito em C++ e tem como objetivo oferecer interfaces padrões para a solução de sistemas lineares. Dessa forma, é possível haver a interoperabilidade entre os solucionadores. A interface ESI dispõe de uma camada padrão que contém interfaces específicas para matrizes, vetores, operadores (solucionadores e precondicionadores), onde essas interfaces podem ser implementadas utilizando uma variedade de bibliotecas. Portanto, o desenvolvedor de aplicação pode acessar a camada padrão proposta pela interface ESI e escolher ou mudar os solucionadores de sua aplicação abstraindo-se da biblioteca utilizada.

Infelizmente, a interface ESI é pobremente documentada e já não é mais desenvolvida. Ainda que seu desenvolvimento esteja paralisado, a interface ESI é comumente utilizada com ponto de partida para a criação de novas interfaces padrões para bibliotecas científicas.

3.4.2 TSC (TOPS Solver Component)

Pesquisadores do projeto TOPS (*Terascale Optimal PDE Simulations*) e do CCTTSS (*Center for Component Technology for Terascale Simulation software*) desenvolveram o TSC (*TOPS Solver Component*) para solucionar problemas de sistemas lineares e não-lineares através de componentes de *softwares*. Os

componentes apresentados no TSC foram desenvolvidos de acordo com o modelo de componentes CCA e suas interfaces são especificadas em SIDL.

Inicialmente, foi construída uma interface padrão para facilitar a interoperabilidade entre os solucionadores. Essa interface é subdivida em dois tipos: TOPS.System e TOPS.Solver. A interface TOPS.System publica recursos para a definição do problema algébrico, já a interface TOPS.Solver publica os recursos para tratamento dos passos de execução dos solucionadores, como sua inicialização, definição dos seus parâmetros e sua execução. Por meio dessas interfaces, é possível encapsular um bom número de bibliotecas científicas, como a Hypre, a SuperLU e a PETSc. No entanto, essas interfaces não são genéricas suficiente para a maioria das bibliotecas paralelas.

Sobre a interface TOPS.System o desenvolvedor da aplicação cria o componente System para definir o sistema algébrico. O componente TSC é construído ao ser implementada a interface TOPS.Solver para cada biblioteca científica possível. Dessa forma, para resolver um sistema algébrico o desenvolvedor deve criar um componente System com a definição do problema, e então, é escolhido o componente TSC mais adequado ao problema definido no componente System.

3.4.3 CCA LISI (CCA LInear Solver Interface)

Desenvolvida por pesquisadores da universidade de Indiana, o projeto CCA LISI (*CCA LInear Solver Interface*) [64] oferece uma interface padrão entre diversas bibliotecas científicas para solução de sistemas lineares. A interface proposta, chamada *SparseSolver*, foi escrita em SIDL e, dessa forma, permite a interoperabilidade entre várias linguagens de programação.

É possível apresentar bibliotecas científicas do domínio de sistemas lineares como componentes CCA ao implementar a interface *SparseSolver* encapsulando as bibliotecas pretendidas. Dessa forma, os componentes CCA gerados podem ser utilizados sobre os *frameworks* CCA para o desenvolvimento de aplicações que requerem soluções de sistemas lineares.

Durante a definição da interface *SparseSolver*, alguns pontos foram considerados para alcançar uma maior generalidade entre as bibliotecas científicas.

O primeiro ponto considerado foi a constatação da semelhança entre as bibliotecas disponíveis. Foi notado que, em geral, as bibliotecas são subdividas em três partes: a definição do sistema linear através das estruturas de dados; a definição de opções, métodos e parâmetros dos solucionadores; e os solucionadores.

Outro ponto levantado foi a necessidade de permitir diversas opções para as chamadas dos solucionadores das bibliotecas, como chamadas únicas para um único sistema, vários solucionadores para um mesmo sistema e chamadas recursivas para um mesmo solucionador. Algumas dessas opções impõem a necessidade de meios para o reuso das estruturas de dados publicadas pela interface.

Outra questão considerada durante a definição da interface foi a escolha das estruturas de dados usadas para representar matrizes esparsas, dentre as quais COO (*coordinate format*), CSR (*compressed sparse row format*), CSC (*compressed sparse column format*), MSC (*modified sparse column format*), MSR (*modified sparse row format*) e um tipo não explícito chamado Matrix-Free. A forma com que os dados dessas estruturas estarão distribuídos entre os processadores segue o padrão de particionamento por blocos.

Como sabemos, os serviços dos componentes CCA são apresentados através de portas *uses/provides*. Dessa forma a aplicação e os solucionadores precisam utilizar tais portas para realizar a interação entre si. Para tanto, foi escolhido uma abordagem em que a aplicação deve possuir portas *uses* para ser acopladas com as portas *provides* fornecidas pelos solucionadores. Dessa forma, o desenvolvedor da aplicação fica mais abstraído das implementações no nível das bibliotecas, sendo necessário apenas invocar as portas *provides* dos solucionadores passando o sistema linear e esperar o resultado de volta.

Como nem todas as bibliotecas científicas compartilham métodos idênticos, a interface CAA LISI propôs publicar métodos da forma mais genérica possível para abranger uma maior cobertura das subrotinas das bibliotecas. Para tanto, são publicados o método para inicializar o sistema, os métodos para criar as estruturas de dados Matriz e Vetor, os métodos para definir o particionamento dos dados, o método para definir o tipo de armazenamento das estruturas de dados, o método para a execução do solucionador e os métodos genéricos *Set* para a definição de outros atributos de acordo com parâmetro *key* fornecido, como o tipo do solucionador, o tipo do precondicionador, o máximo de iterações e etc. A interface CCA LISI pode ser observada no Algoritmo 3.1

Algoritmo 3.1: Interface CCA LISI

```
1 package lisibase version 0.0 {
2   enum SparseStruct {
3         COO,
4         CSR,
```

```
5           MSR
6     }
7     interface SparseSolver {
8           int initialize(in long comm);
9           int setStartRow(in int startrow);
10          int setLocalRows(in int rows);
11          int setLocalNNZ(in int nnz);
12          int setGlobalCols(in int cols);
13          int setOffset(in int offset);
14          int setupMatrix[few_args](in rarray<double,1> Values(NNZ),
15                      in rarray<int,1> Rows(NNZ),
16                      in rarray<int,1> Columns(NNZ),
17                      in int NNZ);
18          int setupMatrix[media_args](in rarray<double,1> Values(NNZ),
19                      in rarray<int,1> Rows(RowsLength),
20                      in rarray<int,1> Columns(NNZ),
21                      in SparseStruct DataStruct,
22                      in int RowsLength,
23                      in int NNZ);
24          int setupRHS(in rarray<double,1> RightHandSide(NumLocalRow),
25                          in int NumLocalRow);
26          int solve(inout rarray<double,1> Solution(NumLocalRow),
27                              inout rarray<double,1> Status(StatusLength),
28                              in int NumLocalRow,
29                              in int StatusLength);
30          int set(in string key, in string value);
31          int setInt(in string key, in int value);
32          int setBool(in string key, in bool value);
33          int setDouble(in string key, in double value);
34          string get_all();
35          }
36    }
```

Em [64] foram realizados experimentos comparativos para a solução de sistemas lineares entre o tempo de execução de bibliotecas científicas como componentes CCA e o tempo de execução de bibliotecas científicas sem o uso de componentes. Obteve-se como resultado que o uso de bibliotecas científicas como componentes CCA, através da interface CCA LISI, acrescenta pouco tempo de execução.

3.4.4 Biblioteca SPARSKIT como componente CCA

A biblioteca científica SPARSKIT, já bastante conceituada pela comunidade científica, oferece ferramentas para computações com matrizes esparsas. No entanto, a SPARSKIT foi escrita em FORTRAN77 e disponibiliza suas funcionalidades através de interfaces com um difícil entendimento. Dessa forma, usar a biblioteca SPARSKIT em aplicações modernas é um grande desafio.

Com o objetivo de facilitar e ampliar o uso das funcionalidades da biblioteca SPARSKIT em aplicações modernas, foi que os pesquisadores J. Jones, M. Sosonkina e Y. Saad das universidades de Iowa e Minnesota disponibilizaram as funcionalidades da biblioteca SPARSKIT como componentes de *softwares* [57]. Para isso foi seguido o modelo de componentes CCA. Dispor a biblioteca SPARSKIT como componente CCA trouxe algumas vantagens para o desenvolvimento de aplicações de CAD, como a facilidade de acrescentar novos métodos, a facilidade da integração com outras aplicações e a interoperabilidade com diversas outras linguagens de programação.

Para tanto, foi desenvolvida classes de acordo com as funcionalidades da biblioteca SPARSKIT, onde cada classe implementa uma interface particular. As classes criadas foram: *Accelerator*, *Preconditioner*, *Matrix Generation*, *Matrix-Vector multiplication* e *Matrix Input/Output*. No entanto, por estar intrinsecamente associada à biblioteca SPARSKIT a interface desenvolvida não consegue abranger outras bibliotecas científicas além da biblioteca SPARSKIT.

Utilizar a biblioteca SPARSKIT como componente CCA não tem uma grande perda de desempenho na execução comparada com ao uso da biblioteca SPARSKIT original, de acordo com [57], tal overhead gira em torno de 28.6% e 29.6%.

3.4.5 Numerical Platon

A ferramenta Numerical Platon [85] foi proposta pelo CEA (*French Atomic Energy Commission*) com o objetivo de facilitar a reusabilidade, a flexibilidade, a portabilidade e a atualização das aplicações de alto desempenho. Embora desenvolvida inicialmente para a construção de aplicações do CEA, a ferramenta Numerical Platon está disponível para a utilização de terceiros sob a licença LGPL.

Para oferecer essas facilidades, a Numerical Platon dispõe de uma interface padrão entre várias bibliotecas científicas do domínio de solução de sistemas lineares, como PETSc, SuperLU, Hypre e solucionadores proprietários do CEA. A interface entre as bibliotecas é disponivel em várias linguagens de programação, como C, C++, FORTRAN, Ocaml e Python.

A ferramenta Numerical Platon é organizada de forma hierárquica possuindo níveis de abstração desde as operações entre as estruturas de dados Matriz e Vetor até as operações dos solucionadores e precondicionadores. Dessa forma, o desenvolvedor tem a liberdade de escolher com qual nível de abstração ele irá trabalhar. Por exemplo, pode-se trabalhar em nível mais abstrato simplesmente executando o solucionador desejado ou trabalhar em nível menos abstrato criando um solucionador específico através das operações entre matrizes e vetores.

A ferramenta Numerical Platon é apresentada como um conjunto de componentes, onde cada componente é uma interface padrão para as subrotinas das bibliotecas científicas do domínio de solução de sistemas lineares. Os componentes são classificados de acordo com os passos básicos para a obtenção da solução do sistema linear, são eles:

- *Environment*, que publica métodos de gerenciamento do paralelismo, medição do tempo, verificação de erros e utilização de *trace* para depuração.

- *NP Vectors e Matrices*, que encapsulam as estruturas dados matriz e vetor das bibliotecas científicas. Esses componentes não definem um formato padrão para matrizes e vetores, mas utiliza diretamente os formatos próprios de cada biblioteca. Os formatos de matrizes possíveis na ferramenta são, matriz densa, matriz simétrica densa, matriz esparsa armazenada por linha (CSR), matriz simétrica esparsa armazenada por linha (CSR_SYM), matriz esparsa armazenada por coluna (CSC) e matriz simétrica esparsa armazenada por coluna (CSC_SYM). Assim, se o usuário utilizar uma matriz no formato CSR e estiver trabalhando com a biblioteca PETSc o formato escolhido para a matriz será o CSR da biblioteca PETSc. Por esses componentes, também são publicados métodos para a manipulação e operação entre as estruturas de dados. Em geral, esses componentes são manipulados através dos outros componentes.

- *NP IO*, que publica métodos para salvar e ler objetos NP em arquivos. No caso de memória distribuída, para a escrita, todos os processos enviam os dados ao processo mestre, e esse salva os dados no arquivo. Já para a leitura, todos os processos podem ler seus dados locais em um mesmo arquivo.

- *NP Solvers and preconditioners*, que publicam os métodos de solucionadores e precondicionadores existentes nas bibliotecas PETSc, Hypre e SuperLU.

No total são 11 diferentes métodos de solucionadores e 5 métodos de precondicionadores.

A ferramenta Numerical Platon já é utilizada em aplicações sequenciais e paralelas do CEA. Com seu uso foi constatado que não existe *overhead* da ferramenta Numerical Platon comparada ao uso direto das bibliotecas. Por exemplo, em [85] foi mostrado que o uso da ferramenta Numerical Platon utilizando a biblioteca Hypre comparado ao uso direto da biblioteca Hypre para a solução de um sistema linear acrescenta apenas entre 0.2 e 1.4 segundos no tempo de execução.

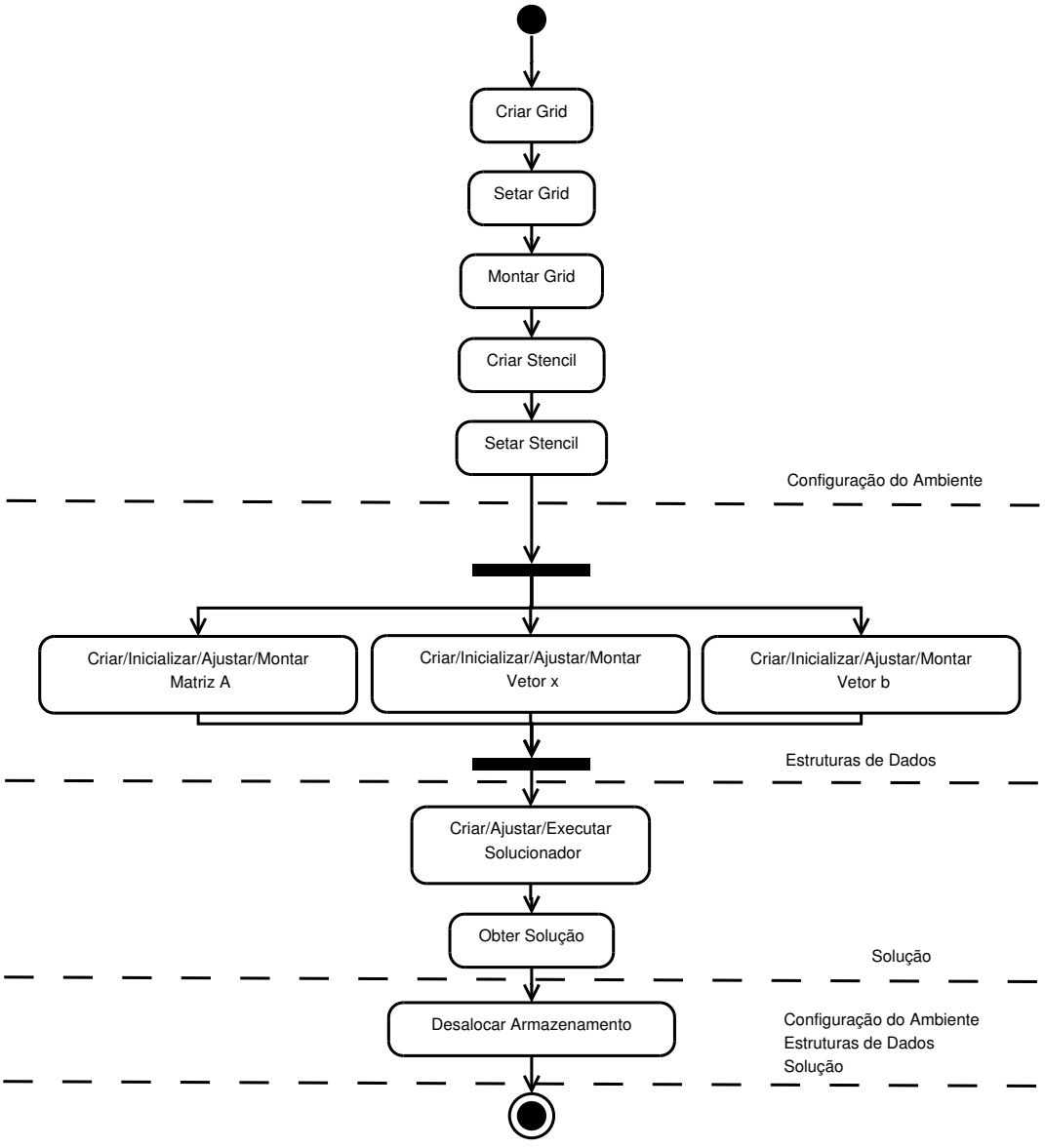


Figura 3.2: Diagrama de atividade para resolver sistemas lineares Ax=b com a biblioteca Hypre

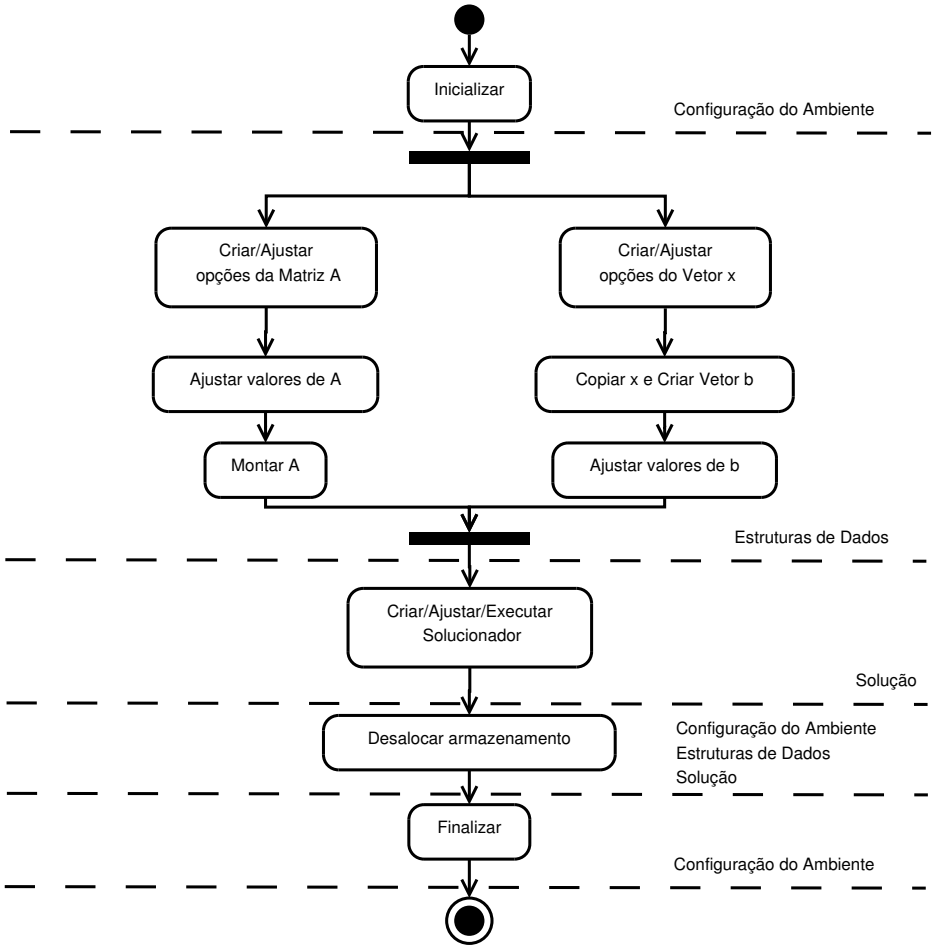


Figura 3.3: Diagrama de atividade para resolver sistemas lineares Ax=b com a biblioteca PETSc

Tabela 3.3: Subrotinas da Classe Estruturas de Dados(Matriz) da biblioteca SuperLU, Hypre e PETSc

Função	SuperLU	Hypre	PETSc
Criar Matriz	√	√	√
Criar Matriz densa	√		
Criar Matriz supernodal	√		
Criar Matriz AIJ Esparsa Sequencial			√
Criar Matriz AIJ Esparsa Paralela			√
Criar Matriz Densa Sequencial			√
Criar Matriz Densa Paralela			√
Criar Matriz Livre			√
Copiar Matriz	√		
Copiar Matriz Densa	√		
Inicializar Matriz		√	
Definir valores matriz		√	√
Adicionar valores		√	√
Montar Matriz		√	√
Definir opções			√
Definir propriedade simétricas		√	
Definir tipo de armazenamento		√	
Definir como complexa		√	
Definir nº máx de não zeros por linha		√	
Definir nº máx de não zeros por linha do bloco diagonal		√	
Converter Matriz Linha para Coluna	√		
Retornar a quantidade de não zeros	√		
Retornar a referência		√	
Retornar valores		√	
Retornar o intervalo de índices pertencentes a este processador			√
Imprimir Matriz	√	√	√
Destruir Matriz	√	√	√

Tabela 3.4: Subrotinas da Classe Solução da biblioteca SuperLU, Hypre e PETSc

Função	SuperLU	Hypre	PETSc
Criar Solve		√	√
Definir operandos			√
Definir método			√
Definir tolerância convergência		√	√
Definir máx de iterações		√	√
Definir opções		√	√
Definir Convergência			√
Definir précondicionador		√	√
Extrair précondicionador do solver			√
Use a zero initial guess		√	
Use a nonzero initial guess		√	√
Convergência padrão			√
Executar solver	√	√	√
Executar solver expert	√		
Executar solver que utiliza matriz ILU	√		
Executar solver expert que utiliza matriz ILU	√		
Executar solver que utiliza matriz LU	√		
Executar solver de uma equação da matriz	√		
Retornar nº de iterações		√	√
Retornar norma		√	
Obter solução			√
Acessar solução aproximada de uma iteração			√
Acessar residual			√
Destruir Solve		√	√

Capítulo 4

Bibliotecas Científicas no HPE

Como abordado no Capítulo 2, muitos são os benefícios ao se utilizar componentes de *software* com modelos orientados aos requisitos de CAD. Aliados a técnicas avançadas de engenharia de *software* e de suporte ao processamento paralelo, tais modelos de componentes são de grande valia durante o desenvolvimento de aplicações que requerem CAD. Também foram expostos em capítulos anteriores os ganhos ao se utilizar bibliotecas científicas na construção de aplicações de CAD, como a reusabilidade, o encapsulamento da comunicação e a utilização dos melhores algoritmos. Utilizar cada um desses recursos separadamente durante o desenvolvimento de aplicações de CAD já é algo bastante comum.

Disponibilizar as vantagens do uso dos modelos de componentes para CAD e do uso das bibliotecas científicas em um mesmo ambiente de desenvolvimento possibilita atingir benefícios ainda maiores. Podemos citar três benefícios que sustentam a realização de tal integração. São eles: o **tratamento da complexidade**, a **integração** e a **troca de solucionadores**. Sem muito esforço, podemos constatar que é de interesse dos usuários finais desfrutar dos ganhos ao realizar essa integração.

Assim como nas aplicações comerciais, é natural que com o crescimento da complexidade na implementação de aplicações nas áreas de ciências e engenharia, com requisitos de CAD, estas exigirão técnicas avançadas de engenharia de *software*. Portanto, é esperado que também os desenvolvedores que fazem uso das bibliotecas científicas irão precisar dos artefatos de engenharia de *software*, mais especificamente dos componentes de *software* para o **tratamento da complexidade** no desenvolvimento das aplicações.

Como sabemos, é bastante comum que em aplicações de CAD de grande porte exista a necessidade da integração entre as aplicações para a obtenção de algum

resultado, onde, por vezes, essas aplicações foram desenvolvidas em lugares e por pessoas distintas. Infelizmente, desenvolvedores que utilizam somente bibliotecas científicas não alcançam a integração entre as aplicações sem um excessivo esforço. No entanto, sabemos também que a integração entre aplicações construídas sobre modelos de componentes de *software* é garantida devido ao benefício da interoperabilidade assegurada em alguns modelos. Dessa forma, podemos garantir a **integração** entre aplicações, que fazem uso de bibliotecas científicas, ao apresentar as bibliotecas através de modelos bem definidos de componentes de *software*.

No caso de bibliotecas para solução de sistemas lineares, a escolha de um solucionador particular depende de alguns fatores, dentre os quais a forma de armazenamento e o padrão de dispersão dos elementos não-nulos da matriz, quando esparsa. No entanto, essa escolha não acontece de forma trivial devido à existência de uma grande quantidade solucionadores, especializados para contextos distintos. Embora o projeto Netlib tenha catalogado vários solucionadores, a escolha do solucionador mais adequado ao problema ainda é algo bastante dispendioso. A dificuldade da escolha é agravada pelo fato de que cada solucionador possui suas próprias estruturas de dados e protocolos próprios de invocação.

Dessa forma, se o desenvolvedor desejar testar qual o solucionador que melhor se comporta em sua aplicação, precisará modificar grande parte do seu código para cada solucionador testado. Por exemplo, em aplicações de CAD é comum a adição de novos requisitos, como um novo modelo físico, levando a uma possível necessidade de troca do solucionador escolhido, obrigando o desenvolvedor a modificar várias linhas do código já escrito para a correta execução do novo solucionador. Não é difícil perceber que o esforço necessário para todas essas modificações no código para cada solucionador pode ser bastante dispendioso.

Sendo assim, é necessário um artefato que possibilite a troca de solucionadores com o mínimo possível de mudanças no código fonte. Como já mencionamos, o paradigma de programação orientada a componentes vem suprir essa necessidade, uma vez que esse paradigma permite a troca dinâmica de componentes em tempo de execução de forma transparente. Portanto, ao apresentar os solucionadores como componentes de *software*, podemos realizar **trocas de solucionadores** com nenhuma ou pouca mudança no código fonte, possibilitando um maior aprendizado do comportamento dos solucionadores para uma escolha mais adequada.

Já existem alguns esforços da comunidade em implementar essa integração. É o caso da utilização da biblioteca SPARSKIT como um componente CCA, o ESI, o

TSC, o CCA LISI e o Numerical Platon. Esses trabalhos encontram-se descritos na Seção 3.4.

Dessa forma, podemos observar que a facilidade em tratar a complexidade na construção de aplicações de CAD, a facilidade em realizar a integração entre aplicações e a facilidade de realizar troca de solucionadores, levam os desenvolvedores de aplicações a requisitar que as bibliotecas científicas que eles utilizam sejam integradas com os modelos de componentes de *software* existentes.

Considerando tudo isso, desejamos propôr meios para integrar bibliotecas científicas à plataforma de componentes paralelos HPE, disponibilizando esse recurso não apenas para o desenvolvedor da biblioteca científica, mas a todos os usuários de bibliotecas científicas que desejam utilizá-las como componentes-#.

4.1 Perspectivas da Integração

Como premissa para definição do método que propomos para a construção de componentes # que implementam as funcionalidades de bibliotecas científicas, admitimos que o processo de desenvolvimento de componentes para o suporte a funcionalidade de bibliotecas científicas pré-existentes deve ser enxergado sob duas perspectivas:

- a perspectiva do *desenvolvedor de aplicações*;
- a perspectiva do *desenvolvedor da biblioteca*.

Sob o ponto de vista do desenvolvedor de aplicações, o método deve ser voltado a facilitar a produção das novas aplicações. Por exemplo, disponibilizar uma interface comum entre algumas bibliotecas, de modo que o usuário poderia alternar entre vários solucionadores sem precisar alterar o restante do código. Já sob o ponto de vista do desenvolvedor da biblioteca, o método teria apenas que oferecer quais os passos necessários que cada desenvolvedor de biblioteca deve seguir para disponibilizar sua ferramenta como um componente #. Ambas as perspectivas são observadas no método proposto nesta dissertação.

4.2 Níveis de Abstração

A fim de garantir o suporte às perspectivas descritas na seção anterior, os componentes propostos neste trabalhos estão classificados em dois níveis de abstração, ilustrados na Figura 4.1.

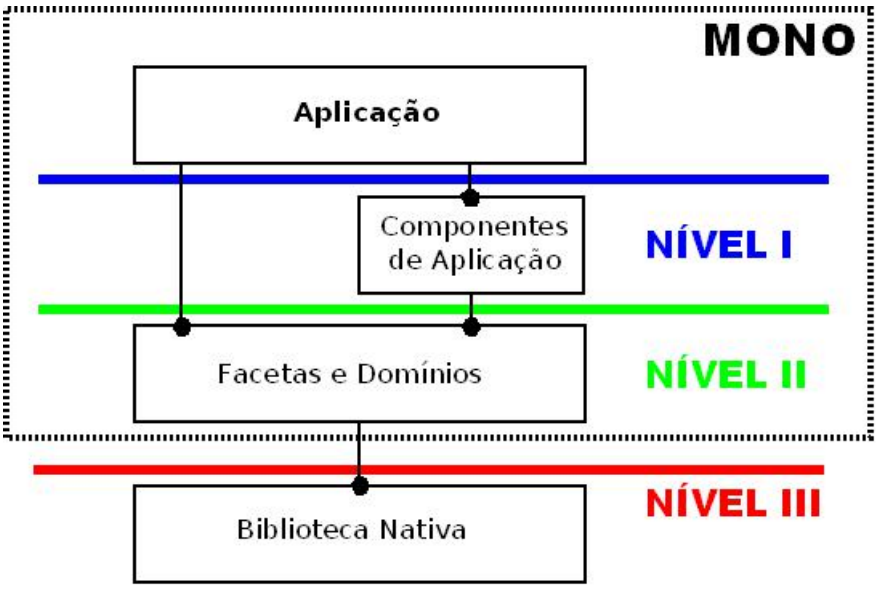


Figura 4.1: Níveis de abstração

No **nível I**, estão os *componentes de aplicação*, das espécies comuns do HPE, ou outro sistema programação # de interesse. Os *componentes de aplicação* oferecem funcionalidades computacionais, da forma mais abstrata possível, para a solução de problemas em algum domínio e são apresentados no paradigma de programação baseada em componentes. Esses componentes pertencentes ao **nível I** serão construídos pelo desenvolvedor da biblioteca, porém com o propósito de serem acessíveis ao usuário final (desenvolvedor de aplicação).

No **nível II**, estão os *componentes de biblioteca*, os quais constituem um invólucro sobre os recursos das bibliotecas nativas, representadas no **nível III**, e portanto possui um nível de abstração inferior comparada ao **nível I**. Esses componentes têm o objetivo de serem usados pelo desenvolvedor de *biblioteca*, mas podem ser acessados diretamente pelo usuário final (desenvolvedor de aplicação).

Os *componentes de aplicação*, no nível **nível I**, devem ser desenvolvidos utilizando as funcionalidades publicadas através dos componentes presentes no **nível II**. Por exemplo, pode-se criar um componente de aplicação para a solução de sistemas lineares realizando uma sequência de chamadas dos métodos apresentados no **nível II**. Para implementação dos componentes do **nível II**, a plataforma Mono, sobre a qual o ambiente de execução do HPE está implementado, oferece mecanismos simples de acesso ao código nativo.

Tal abordagem, permite ao desenvolvedor de aplicação trabalhar sob os dois níveis de abstração de acordo com as necessidades do desenvolvimento. Dessa forma,

para um desenvolvimento com *alta abstração*, o desenvolvedor de aplicação pode simplesmente requisitar um componente do **nível I** para a solução do problema de algum domínio, como pode também, chamar apenas as funcionalidades dos componentes do **nível II** para um desenvolvimento com *baixa abstração*, ou ainda, o desenvolvedor pode utilizar em uma mesma aplicação componentes do **nível I** e utilizar alguns métodos dos componentes do **nível II** para um desenvolvimento com *abstração híbrida*, por exemplo se antes da execução do componente do **nível I**, for necessário triangularizar a matriz fornecida, pode-se chamar um método de algum componente do **nível II** para tal. Portanto, o desenvolvedor tem total liberdade para utilizar o nível de abstração mais adequado.

O uso da classificação das abstrações já foi adotado em outros trabalhos do mesmo nicho [82, 85]. Para garantir uma maior flexibilidade, também resolvemos adotar uma classificação. No entanto, a classificação utilizada em nosso trabalho difere desses trabalhos anteriores, pois resolvemos classificar a abstração em níveis de generalidade entre as bibliotecas, onde o nível inferior deve ser mais específico para alguma biblioteca e o nível superior deve ser mais genérico entre todas as bibliotecas do mesmo domínio. Já nos outros trabalhos citados, a classificação é definida de acordo com a funcionalidade das subrotinas, onde o nível inferior apresenta métodos genéricos para operações entre matrizes e vetores, e o nível superior apresenta métodos genéricos de resolvedores do domínio de solução de sistema linear. A abordagem adotada nesta dissertação oferece total liberdade ao desenvolvedor para trabalhar no nível de generalidade desejado. A classificação considerada nos outros trabalhos já está incluída em nossa proposta através da classificação das *facetas* do domínio, a qual está explanada na Seção 4.4.

4.3 Método Proposto: Estudo de Caso

Para validação do método proposto, abordaremos o domínio das bibliotecas voltadas a solução de sistemas lineares no estudo de caso. Especificamente, integraremos as bibliotecas SuperLU, Hypre e PETSc, pertencentes a esse domínio, ao HPE. A escolha dessas bibliotecas se deve ao fato de já serem bastante utilizadas dentro do domínio abordado. Conjecturamos que o método proposto possa ser extrapolado para outros domínios, tendo em vista sua generalidade, pois as suposições usadas em sua definição são válidas para bibliotecas de outros domínios.

Ao observar os diagramas de atividade apresentados na Seção 3.3, desejamos constatar os aspectos comuns entre os passos necessários para se obter uma solução

de um sistema linear utilizando as bibliotecas SuperLU, Hypre e PETSc. Notamos que, em geral, para a construção de uma aplicação é preciso: criar e instanciar o ambiente que será implantado; criar e instanciar as estruturas de dados necessárias; criar, instanciar e executar o solucionador; por fim, liberar recursos de sistema eventualmente alocados. Observando também os procedimentos existentes nessas bibliotecas, verificamos que esses podem ser agrupados em três categorias principais (presente em todas as bibliotecas do domínio):

- os procedimentos para configuração, gerenciamento e monitoração da biblioteca e do ambiente;
- os procedimentos que lidam com a criação e atualização de estruturas de dados, incluindo vetores e matrizes de vários tipos;
- os procedimentos para a execução e manipulação de solucionadores;
- os procedimentos voltados a criação e configuração de pré-condicionadores da matriz.

De posse dessas informações, concebemos alguns artefatos para implementar a integração de forma mais abrangente possível, os quais podem ser reproduzidos em outros domínios ou envolvendo outras bibliotecas no domínio da álgebra linear.

4.4 Introdução de Novas Espécies: Domain e Facet

O uso de espécies de componentes possibilita introduzir no HPE restrições de configuração que permite implementar o método proposto nesta dissertação.

Para isso, são incluídas duas novas espécies de componentes no HPE, denominadas Domain e Facet. Os componentes abstratos da espécie Domain destinam-se a representar algum domínio específico. Por exemplo, a partir dela podemos representar o domínio de solução de sistemas lineares, o domínio de solução de problemas de otimização, dentre outros. Para cada um desses domínios, pressupõe-se a existência de vários conjuntos de funcionalidades de diferentes intentos, onde cada funcionalidade geralmente é expressa por uma subrotina. Cada conjunto é chamado de *faceta*, onde uma mesma faceta pode pertencer a mais de um domínio. Por exemplo, uma faceta que agrega subrotinas para a manipulação de matrizes pode também pertencer a vários domínios onde operações com matrizes são necessárias. Os componentes da espécie Facet são voltados para descrever as

facetas de um determinado domínio. Para cada *faceta*, deve existir um componente da espécie FACET que publica as subrotinas pertencentes à *faceta* determinada.

Um componente abstrato da espécie DOMAIN deve especificar entre seus componentes aninhados um ou mais componentes da espécie FACET, os quais podem ser públicos ou privados. Os componentes públicos são acessíveis por componentes aplicação (**nível I**) que usam o domínio, enquanto os componentes privados são usados apenas para implementação das funcionalidades dos componentes públicos, não visíveis aos componentes de aplicação. É possível ainda contextualizar o componente domínio pelas suas facetas, definindo parâmetros de contexto associados a componentes abstratos da espécie FACET.

Os componentes das espécies DOMAIN e FACET devem ainda possuir um parâmetro de contexto adicional do tipo LIBRARY, da espécie ambiente (ENVIRONMENT) do HPE, chamado *library*. Para cada biblioteca suportada, haverá um componente abstrato derivado a partir de LIBRARY, com o propósito de suprir o parâmetro de contexto *library*. A interface requerida por LIBRARY para cada unidade inclui informações básicas sobre a identificação e versão da biblioteca, bem como as estruturas de dados especifícas da biblioteca, como também configurações de ambiente que não dizem respeito à execução de subrotinas específicas. Dessa forma, deve-se criar componentes específicos, das espécies DOMAIN e FACET, para cada biblioteca pretendida. Por exemplo, deve-se criar um componente específico da faceta *matriz* para cada biblioteca de interesse, cuja interface oferece acesso às funcionalidades específicas da biblioteca relacionadas à construção e processamento de matrizes.

Os componentes-# (concretos) que representam as bibliotecas do domínio devem ser definidos a partir de instanciação dos componentes abstratos da espécie DOMAIN para o contexto determinado pelo parâmetro de contexto *library* e cada parâmetro de contexto $\langle facet \rangle$, onde $\langle facet \rangle$ representa o nome de um parâmetro de contexto cujo tipo é definido por um componente abstrato da espécie FACET.

Permitindo-se especializar o componente domínio de acordo com suas facetas, é possível ao componente domínio acessar funcionalidades específicas da biblioteca, e não estar restrito a uma interface genérica definida pelo componente faceta genérico. Portanto, para cada biblioteca, podem ser definidos componentes faceta específicos, por especialização das facetas mais gerais que sejam parâmetros de contexto do componente domínio implementado.

De posse dessas duas novas espécies de componentes, podemos representar

as funcionalidades de qualquer biblioteca científica de qualquer domínio como componentes-#, tendo como única restrição que a biblioteca a ser representada possua as mesmas *facetas* básicas (necessárias para execução) de qualquer outra biblioteca do mesmo domínio. Isso é possível, pois através da criação de uma componente abstrato # da espécie DOMAIN podemos introduzir restrições próprias de um dado domínio, como também através da criação de componentes abstratos da espécie FACET podemos introduzir restrições particulares das *facetas* do dado domínio. Tendo criado os componentes abstratos das espécies DOMAIN e FACET para representar um domínio de biblioteca, podemos então instanciá-los para qualquer biblioteca do domínio representado que possuam as mesmas *facetas* básicas.

Por exemplo, no nosso estudo de caso admitimos que o domínio de solução de sistemas lineares engloba cinco *facetas*, envolvendo subrotinas com os seguintes propósitos:

- configuração e gerenciamento do *ambiente*;
- construção e atualização de *vetores*;
- construção e atualização de *matrizes* (densas e esparsas);
- construção, configuração e invocação de *solucionadores*;
- construção, configuração e invocação de *pré-condicionadores*.

Sabendo disso, deve-se então criar um componente abstrato da espécie DOMAIN e criar um componente abstrato da espécie FACET para cada *faceta* básica. Portanto, para representar uma nova biblioteca do domínio de solução de sistemas lineares como componente #, a biblioteca deve ter como premissa a existência das cinco *facetas* básicas citadas.

4.5 Introduzindo um Novo Domínio de Biblioteca

Com o propósito de introduzir um novo domínio de biblioteca, devem ser obedecidos os passos descritos nos próximos parágrafos, já voltados ao estudo de caso com bibliotecas do domínio de solucionadores de sistemas lineares. De forma análoga, tais passos devem ser adotados aos outros domínios desejados.

4.5.1 Componentes de Biblioteca

Inicialmente, são criados os seguintes componentes de biblioteca (**nível II**), associados a um certo domínio:

- um componente abstrato da espécie DOMAIN para o domínio;
- componentes abstratos da espécie FACET, um para cada *faceta* do domínio.

No domínio de bibliotecas de solução de sistemas lineares, chamaremos o primeiro componente de LSSDOMAIN. Além disso, há os seguintes componentes da espécie FACET, associados a denominação da *faceta* correspondente e sua caracterização:

- LSSFACETSETUP (*faceta Configuração*): procedimentos para configuração, gerenciamento, e monitoração da biblioteca e do ambiente de execução;
- LSSFACETMATRIX (*faceta Matriz*): procedimentos para criação e manipulação de matrizes de múltiplas dimensões e de diversos tipos;
- LSSFACETVECTOR (*faceta Vetor*): procedimentos para criação e manipulação de vetores;
- LSSFACETSOLVER (*faceta Solucionador*): procedimentos para soluções de sistemas lineares de diversos tipos;
- LSSFACETPRECONDITIONER (*faceta Pré-Condicionador*): procedimentos para pré-condicionamento da matriz de entrada.

O componente que representa o domínio (ex: LSSDOMAIN) deve incluir em sua configuração um componente aninhado para cada *faceta* do domínio. Além disso, os tipos das *facetas* devem ser configurados como parâmetros de contexto, permitindo assim, que através dos tipos das *facetas* seja especificado algum cenário particular para a execução do componente. No caso do componente LSSDOMAIN, teríamos a seguinte configuração, escrita em HCL (*Hash Configuration Language*):

```
domain LSSDomain[library = L:Library,
                 setup = S:LSSFacetSetup[library = L],
                 vector = V:LSSFacetVector[library = L],
                 matrix = M:LSSFacetMatrix[library = L],
                 solver = R:LSSFacetSolver[library = L],
                 pre_conditioner = P:LSSFacetPreConditioner[library = L]]
begin
    enumerator n
    environment mpi : MPI
    environment library : L
    facet setup : S
    facet matrix : M
    facet vector : V
    facet solver : R
    facet pre_conditioner : P
    unit lssdomain
    begin
        slice n from n.index
        slice c from mpi.comm
        slice l from library.qualifier_unit
        slice s from setup.facet_unit
        slice m from matrix.facet_unit
        slice v from vector.facet_unit
        slice r from solver.facet_unit
        slice p from pre_conditioner.facet_unit
    end
end
```

onde:

- L, S, M, V, R e P são variáveis de contexto, representando os parâmetros denominados *library*, *setup*, *vector*, *matrix*, *solver* e *pre_conditioner*, respectivamente;
- *library* é um componente aninhado de visibilidade privada representando a biblioteca do domínio representado pela configuração, cujo tipo é delimitado pelo parâmetro de contexto enumerado no item anterior;
- **enumerator**, **environment** e **facet** são espécies de componentes.

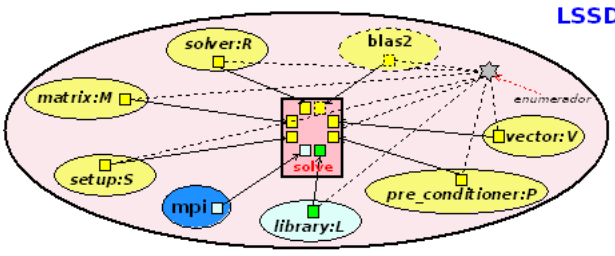
- ***setup***, ***matrix***, ***vector***, ***solver*** e , ***pre_ conditioner*** são componentes aninhados de visibilidade pública representando as *facetas* do domínio representado pela configuração, cujos tipos são delimitados pelos parâmetros de contexto enumerados no item anterior, respectivamente;

- *lssdomain* é uma unidade, a qual pode ser vista como um pacote que contém as interfaces associadas às suas fatias;

- l, s, m, v, r e p são as fatias da unidade *lssdomain*, provenientes de unidades dos componentes aninhados da configuração.

Usando essa abordagem, é possível especializar os componentes LSSFacetSetup, LSSFacetMatriz, LSSFacetVetor, LSSFacetSolver e LSSFacetPreConditioner através dos parâmetros de contexto, de acordo com a biblioteca. Além do mais, a inclusão de novas *facetas* depende da inclusão de novos componentes aninhados da espécie Facet no componente da espécie Domain que representa o domínio, oferecendo maior flexibilidade ao desenvolvedor da biblioteca.

A Figura 4.2 ilustra os componentes aninhados da espécie faceta que constituem a configuração do componente LSSDomain. Nesse componente, há um componente aninhado do tipo MPI, de nome ***mpi***, o qual é acessivel tanto aos componentes de biblioteca (**nível II**) quanto aos componentes de aplicação (**nível I**), a fim de compartilharem o mesmo ambiente de troca de mensagens baseado no padrão MPI. O componente aninhado ***mpi*** define a interface de troca de mensagens usadas para implementar o paralelismo da biblioteca. É importante portanto destacar o caráter paralelo dos componentes envolvidos, uma vez que suas unidades, correspondem a um conjunto de unidades homogêneas, notadamente implantadas em nós distintos de um *Cluster*, por estarem ligadas ao *enumerador* indicado na ilustração e representado pelo componente aninhado n, da espécie *Enumerador*, no código HCL apresentado anteriormente.




```
LSSDomain
  [ library = L : lss.Library,
    solver = R : lss.facet.Solver [library = L],
    matrix = M : lss.facet.Matrix [library = L],
    vector = V : lss.facet.Vector[library = L],
    pre_conditioner
          = P : lss.facet.PreConditioner[library = L],
    setup = S : lss.facet.Setup [library = L]
  ]
```

Figura 4.2: Configuração do componente LSSDomain

4.5.2 Componentes de Aplicação

Os componentes oferecidos no **nível II** oferecem um nível de abstração aquém daquele pretendido por uma interface de programação orientada a componentes. Então, o método proposto foi definido de modo a tornar possível a construção de componentes de mais alto nível, ditos *componentes de aplicação* (**nível I**), implementados sobre uma ou mais bibliotecas de um mesmo domínio. Assim, os componentes de domínio (**nível II**) podem ser usados para construir os componentes de aplicação (**nível I**), conforme descrito na Seção 4.2. Tais componentes podem ser implementadas sobre qualquer uma das bibliotecas do domínio. Tal abordagem segue a tendência da padronização das interfaces entre as bibliotecas, em função da experiência dos seus usuários.

Esses novos componentes pertencerão às demais espécies do HPE, onde a escolha da espécie dependerá da aplicação do componente. Por exemplo, no domínio estudado existirão os componentes *resolvedores* da espécie COMPUTATION, bem como os componentes *matrizes* e *vetores* da espécie DATA STRUCTURE. Note a relação entre os componentes de aplicação e as facetas para esse domínio.

Cada componente de aplicação terá que ter o componente abstrato do domínio como componente aninhado, de forma que possa ter acesso às suas facetas que sejam úteis na implementação. Deverá ainda ter o parâmetro de contexto *library*, do tipo LIBRARY, uma vez que poderá ser especializado para uma certa biblioteca. Outros parâmetros de contexto e componentes aninhados podem ainda ser acrescentados a um componente de aplicação, de acordo com as suas necessidades. Isso pode ser observado através da Figura 4.3, que ilustra o componente de aplicação SOLVER, o qual possui como componente aninhado o componente abstrato do domínio de solução de sistemas lineares LSSDOMAIN.

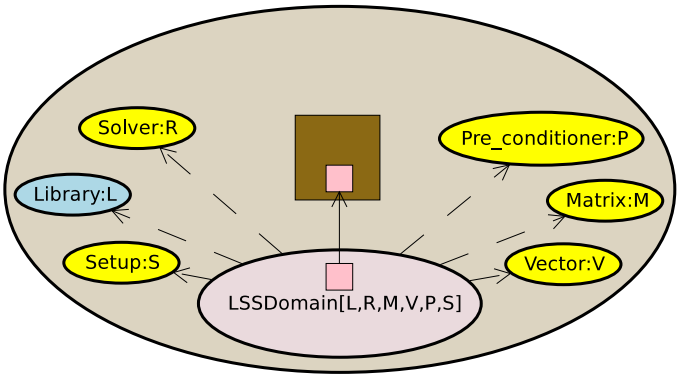


Figura 4.3: Componente SOLVER com o componente aninhado LSSDOMAIN

Tabela 4.1: Componentes de aplicação para o domínio de solução de sistema lineares

Componente	Espécie
SOLVER	*Computation*
MATRIX	*Data Structure*
VECTOR	*Data Structure*
LIBRARY	*Environment*
SOLUTIONMETHOD	*Qualifier*
MATRIXPROPERTY	*Qualifier*
ELEMENTDATATYPE	*Qualifier*
PRECONDITIONER	*Qualifier*

Tabela 4.2: Parâmetros de Contexto do Componente de Aplicação SOLVER

Parâmetro de contexto	Variável	Limite
library	*LIB*	LIBRARY
method	*MTH*	SOLUTIONMETHOD [*LIB*, *PRC*, *MPT*]
matrix_property	*MPT*	MATRIXPROPERTY [*LIB*]
numeric_type	*NUM*	NUMERICDATATYPE
matrix_type	*MAT*	MATRIX [*LIB*, *NUM*, *MPT*]
vector_type	*VEC*	VECTOR [*LIB*, *NUM*]
pre_conditioner	*PRC*	PRECONDITIONER [*LIB*, *MPT*]

O HPE oferece total flexibilidade para especificação dos componentes de aplicação. Como discutimos, o parâmetro de contexto *library* permite ao programador do **nível I** selecionar uma implementação (instanciação) específica do componente de aplicação para uma certa biblioteca. Sem o parâmetro de contexto, o programador fica restrito à biblioteca específica usada para implementar o componente de aplicação. Uma outra possibilidade é o uso de variáveis livres na configuração do componente de aplicação. Dessa forma, será selecionada qualquer uma de suas implementações sobre alguma biblioteca científica que esteja disponível. Os componentes qualificadores podem ser usados como parâmetros de contexto para orientar a escolha, mas na existência de mais de uma implementação, qualquer uma poderá ser escolhida arbitrariamente. As Seção 5.6 apresenta alguns exemplos de especificação dos componentes de aplicação para algum contexto.

A Tabela 4.1 apresenta os componentes de aplicação para o domínio de solução de sistemas lineares, propostos nesse trabalho, descrevendo suas espécies. O componente SOLVER é o mais importante deles, implementando solucionadores de sistemas lineares que podem ser especializados conforme:

i. a biblioteca usada na implementação;

ii. o algoritmo ou método utilizado na solução;

iii. alguma propriedade relevante da matriz de entrada para afetar a escolha do algoritmo ou estratégia de solução (diagonal, tridiagonal, tridiagonal em blocos, etc.) ;

iv. tipo numérico dos elementos da matriz (inteiros, ponto flutuante de precisão simples ou dupla, complexos, etc.)

v. a estrutura de dados usadas para representar a matriz, densa ou esparsa;

vi. a estrutura de dados usadas para representar o vetor;

vii. tipo do pré-condicionador a ser empregado.

Para isso, respectivamente, SOLVER possui os parâmetros de contexto definidos na Tabela 4.2, de onde podem também ser conhecidos os parâmetros de contexto dos demais componentes de aplicação.

Através dos parâmetros de contexto, como discutido na Seção 2.11, o programador é capaz de guiar o sistema de ligação de componentes do Back-End do HPE, baseado no sistema de tipos HTS, para escolher o componente-# mais apropriado. Por exemplo, um programador que use, como componente aninhado de uma aplicação, o componente SOLVER no contexto [***matrix_ property*** = BLOCKTRIDIAGONAL, ***numeric_ type*** = COMPLEX], deixando os demais parâmetros livres, estará demandando por um componente-#

Tabela 4.3: Tipos de armazenamento de matriz

Formato de armazenamento	SuperLU	Hypre	PETSc
Compressed Sparse Column	√		
Compressed Sparse Row	√	√	√

Tabela 4.4: Propriedades da matriz

Propriedades	SuperLU	Hypre	PETSc
Block			√
Symmetric block			√
Dense			√
General	√	√	√
Lower triangular, unit diagonal	√		
Upper triangular, unit diagonal	√		
Lower triangular	√		
Upper triangular	√		
Symetric lower	√		
Symetric upper	√		
Hermitian lower	√		
Hermitian upper	√		

que implementa um resolvedor apropriado para lidar com matrizes de números complexos esparsas cujo padrão de elementos não-nulos é tridiagonal-em-blocos. Neste caso, o uso da variável livre sobre os demais parâmetros de contexto determina que para tais contextos as escolhas são arbitrárias. Caso não exista no ambiente de execução um componente-# que implemente SOLVER para tal contexto, os parâmetros atuais de contexto podem ser generalizados até que um componente-# seja encontrado, o qual é retornado, ou não seja mais possível uma generalização, quando um erro de ligação ocorre. Por exemplo, uma possível generalização seria o contexto [$\boldsymbol{matrix\_property} = \text{FREE}, \boldsymbol{numeric\_type} = \text{COMPLEX}$], o qual englobaria componentes-# resolvedores que não fazem suposição a cerca da distribuição dos elementos não-nulos da matriz. Portanto, esse resolvedor é viável para uma matriz esparsa do tipo tridiagonal-em-blocos. Vários cenários de utilização para o componente Solver serão exercitados na Seção 5.5.

É desejável portanto que o desenvolvedor de biblioteca desenvolva vários componentes-# resolvedores, uns mais genéricos e outros mais especializados, variando o contexto, para o benefício dos desenvolvedores de aplicações. Dessa forma, ao introduzir-se um novo resolvedor apropriado para um certo contexto, o desenvolvedor de aplicação não precisaria modificar a aplicação que faz suposição a cerca desse contexto ou contextos mais especializados. Essa seria uma preocupação

Tabela 4.5: Métodos para a solução do sistema linear

Métodos	SuperLU	Hypre	PETSc
Richardson			√
Jacobi		√	
Chebychev			√
Conjugate Gradient		√	√
BiConjugate Gradient			√
GMRES		√	√
BiCGSTAB			√
Conjugate Gradient Squared			√
Transpose-Free Quasi-Minimal Residual (tfqmr)			√
Transpose-Free Quasi-Minimal Residual (tcqmr)			√
Conjugate Residual			√
Least Squares Method			√
LU	√		√
Cholesky			√
QR			√
XXt			√
XYt			√

inerente ao desenvolvedor da biblioteca, o qual poderá tornar disponível o componente especializado para que as aplicações que usam aquele contexto de resolvedor possam usufruir de melhora no desempenho em suas aplicações já existentes.

Alterações no contexto do problema também são viáveis sem a necessidade de grandes mudanças no código fonte. Por exemplo, se já tivermos uma aplicação de CAD que utiliza o componente SOLVER para resolver algum sistema linear e que tenha como entrada uma matriz simétrica, mas em outro momento a aplicação, por alguma outra requisição, precise resolver algum sistema linear que tenha como entrada uma matriz hermitiana, então basta ao usuário atualizar o contexto do componente MATRIX e então o HPE irá adaptar a execução do componente para o novo contexto, podendo até escolher a biblioteca e o método de solução mais apropriado para o problema. Em trabalhos futuros, poderíamos abordar a troca dinâmica de contexto. Nesse caso, a capacidade de auto-adaptação atingida com

Tabela 4.6: Tipos de precondicionadores

Tipos de precondicionadores	SuperLU	Hypre	PETSc
ParaSails		√	
Euclid		√	
PILUD		√	
diagonal		√	
Boomerang AMG		√	
Jacobi			√
Block Jacobi			√
SOR / SSOR			√
SOR with Eisenstat Trick			√
Incomplete Cholesky			√
Incomplete LU	√		√
Additive Scharwz			√
Linear Solver (ksp)			√
Combination of precondioners			√
LU			√
Cholesky			√
Shell (for user-defined pc)			√

as mudanças do contexto se assemelharia às aplicações com características de Computação Autônoma [83].

Ao observar os parâmetros de contexto adotados em nossos componentes de aplicação, percebemos que inúmeros são os possíveis cenários para a solução de um sistema linear. Sendo assim, resolvemos listar, entre os parâmetros de contexto possíveis, apenas aqueles que estão presentes nas bibliotecas adotadas para esse trabalho. Para tanto, as Tabelas 4.3, 4.4, 4.5 e 4.6 apresentam alguns dos possíveis contextos para os parâmetros MFT, MPT, MTH e PRC, respectivamente. Os contextos existentes em dada biblioteca estão marcadas com o símbolo √ na coluna referente à biblioteca a qual pertence. É importante ressaltar também que as tabelas referidas não apresentam todos os contextos possíveis das bibliotecas escolhidas para este trabalho, no entanto destacamos os principais cenários para os parâmetros de contexto.

Algoritmo 4.1: Interface do componente de aplicação SETUP

```
1  public interface IFacetSetup<L>
2  where L:ILibrary
3  {
4          int initialize(MPI_COMM_WORLD comm);
5          int set(string key, in string value);
6          int setInt(string key, in int value);
7          int setBool(string key, in bool value);
8          int setDouble(string key, in double value);
9          string get_all();
10 }
```

Para a definição das interfaces genéricas presentes em cada componente de aplicação tomamos por base a interface CCA LISI, descrita na Subseção 3.4.3. Optamos pela escolha da interface CCA LISI como referência para a definição de nossas interfaces, por considerarmos um trabalho já relevante sobre o tema e por julgarmos que a interface atende as necessidades de generalidade que desejamos. Sendo assim, cada componente de aplicação proposto possuirá uma interface genérica, entre as bibliotecas do domínio de sistema lineares, onde cada uma publicará rotinas que são mais comuns entre as mesmas facetas e publicará métodos *get/set* para os ajustes dos demais atributos de cada *faceta*. Os Algoritmos 4.1, 4.2, 4.3 e 4.4 apresentam as interfaces genéricas dos componentes de aplicação SETUP, VECTOR, MATRIX e SOLVER, respectivamente.

Algoritmo 4.2: Interface do componente de aplicação VECTOR

```
1  public interface IFacetVector<L>
2  where L:ILibrary
3  {
4      int setupRHS(double[] RightHandSide,
5                   int NumLocalRow);
6      int set(string key, in string value);
7      int setInt(string key, in int value);
8      int setBool(string key, in bool value);
9      int setDouble(string key, in double value);
10     string get_all();
11 }
```

4.6 Instanciação de uma Biblioteca

Para realizar a instanciação de uma biblioteca científica no domínio de solução de sistemas lineares no HPE iremos utilizar os componentes abstratos, criados neste trabalho, obedecendo os seguintes passos.

Primeiramente, deve-se criar um componente da espécie qualificador derivado a partir de LIBRARY para a biblioteca. Por exemplo, os componentes nomeados PETSc, SUPERLU e HYPRE. Conforme descrito na seção anterior, a interface das unidades requerida por LIBRARY pode ser estendida para as necessidades específicas da biblioteca. É orientado que sejam incluídos nesse componente as estruturas de dados particulares de cada biblioteca científica.

Algoritmo 4.3: Interface do componente de aplicação MATRIX

```
1  public interface IFacetMatrix<L>
2  where L:ILibrary
3  {
4      int setStartRow(int startrow);
5      int setLocalRows(int rows);
6      int setLocalNNZ(int nnz);
7      int setGlobalCols(int cols);
8      int setOffset(int offset);
9      int setupMatrix(double[] Values,
10                     int[] Rows,
11                     int[] Columns,
12                     int NNZ);
13     int set(string key, in string value);
14     int setInt(string key, in int value);
15     int setBool(string key, in bool value);
16     int setDouble(string key, in double value);
17     string get_all();
18 }
```

Em seguida, para cada componente abstrato que representa uma faceta do domínio (espécie FACET), iremos criar, por herança, componentes abstratos referentes às facetas da biblioteca particular. Por exemplo, no caso da biblioteca PETSc, os componentes criados serão PETSCSOLVER, PETSCMATRIX, PETSCVECTOR, PETSCUTILS, PETSCSETUP e PETSCPRECONDITIONER. As interfaces associadas às unidades desses novos componentes deverão publicar todas as subrotinas pertencentes a uma faceta da biblioteca determinada. Dessa forma, o desenvolvedor terá o controle de todas as subrotinas das bibliotecas científicas para

esse domínio.

Algoritmo 4.4: Interface do componente de aplicação SOLVER

```
1  public interface IFacetSolver<L>
2  where L:ILibrary
3  {
4      int solve(double[] Solution,
5                double[] Status,
6                int NumLocalRow,
7                int StatusLength);
8      int set(string key, in string value);
9      int setInt(string key, in int value);
10     int setBool(string key, in bool value);
11     int setDouble(string key, in double value);
12     string get_all();
13 }
```

Então, deve-se instanciar um componente concreto a partir do componente que representa o domínio pretendido, suprindo o parâmetro de contexto *library* pelo componente qualificador da biblioteca. As unidades do componente, correspondentes a classes C# no HPE, devem implementar todas as interfaces referentes a cada faceta do domínio.

Um componente concreto para suporte à biblioteca PETSc pode ser derivado a partir da seguinte instanciação de LSSDOMAIN:

$$\begin{aligned}\text{LSSDomain}\,[&\textit{library} = \text{PETSc},\\ &\textit{setup} = \text{PETScSetup}\\ &\textit{vector} = \text{PETScVector},\\ &\textit{matrix} = \text{PETScMatrix},\\ &\textit{solver} = \text{PETScSolver},\\ &\textit{pre\_conditioner} = \text{PETScPreConditioner}]\end{aligned}$$

Exemplificamos essa abordagem apresentando o diagrama de componentes UML na Figura 4.4 que representa a instanciação da biblioteca PETSc ao HPE. As setas tracejadas representam a relação de realização, enquanto as setas cheias representam a relação de generalização. As setas com o losango preenchido representam a relação de composição e a seta entre o componente PETSC e o pacote PETSC FONTE representa a relação de dependência.

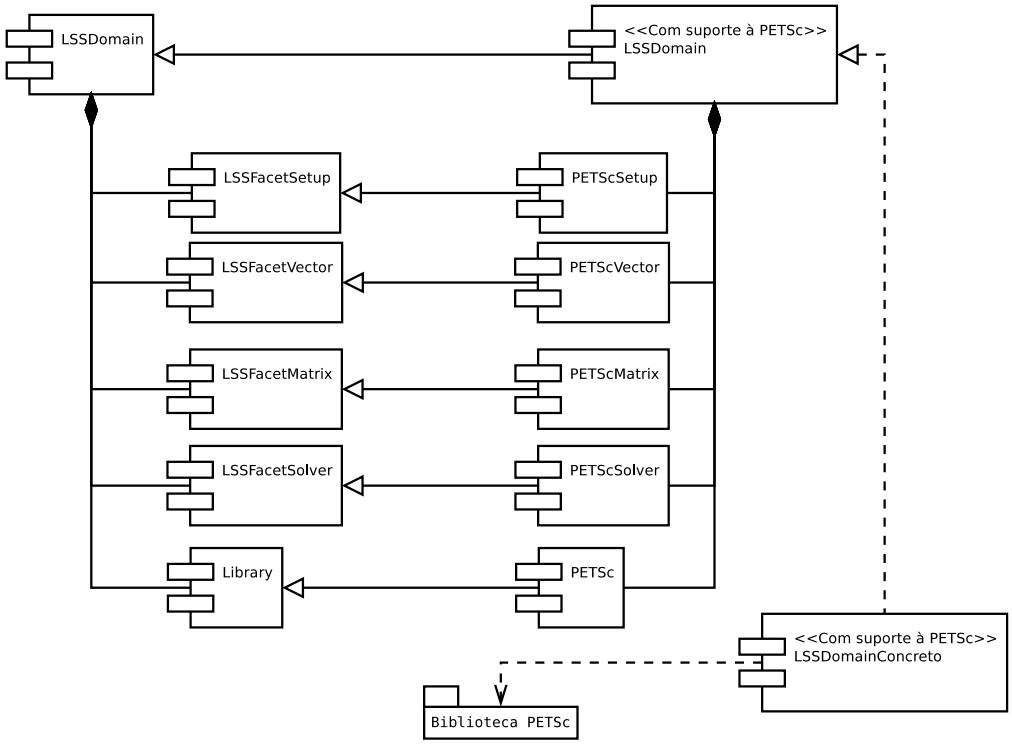


Figura 4.4: Instanciação da biblioteca PETSc

Capítulo 5

Estudo de caso: Equação de Poisson

Este Capítulo apresenta um estudo de caso simples que visa apresentar o uso dos componentes de aplicação pelo usuário a fim de resolver um problema prático de forma eficiente, permitindo que o próprio sistema carregue o componente que implementa a solução mais eficiente para o contexto de problema apresentado. Para tanto, resolvemos eleger o problema da resolução da Equação de Poisson. Embora simples, o problema é largamente utilizado em diversas áreas, que serão apresentadas mais a frente.

O objetivo principal do estudo de caso não é realizar uma avaliação rigorosa de desempenho dos componentes, pois o custo de computação da solução de grandes sistemas lineares, que justificam o paralelismo, facilmente torna pouco significativo o custo das ligações do código C# ao código nativo e a sua componentização. Para isso, outros trabalhos do grupo de pesquisa ParGO estão avaliando o custo da plataforma de componentes HPE sobre programas. O que temos observado em nossos estudos preliminares, como é o caso do problema da integração numérica multi-dimensional discutido na Seção 2.13.1, é que a utilização de aplicações construídas sobre o HPE mostra uma perda de desempenho da ordem de 5% a 10%.

O objetivo real do nosso estudo de caso é mostrar na prática os benefícios da utilização de componentes de aplicação para a resolução de algum problema, mais especificamente o problema da resolução da Equação de Poisson. Para isso, de acordo com o problema iremos descrever alguns cenários de uso, enfatizando a configuração do contexto através dos parâmetros de contexto do componente Solver.

5.1 A Equação de Poisson

Equações Diferenciais Parciais(EDP) são bastante utilizadas em diversas áreas da ciência para modelar sistemas físicos, permitindo simulá-los a fim de avaliar fenômenos naturais. As EDP's possuem a seguinte forma:

$$-a\nabla^2 u + b \cdot (\nabla u) + cu = f \ \ em \ \ \Omega \tag{5.1}$$

Em que se deseja calcular a função $u : \Omega \to R^d$, onde o domínio Ω é um subconjunto de R^d.

A Equação de Poisson é um caso particular da Equação 5.1, em que a é uma constante positiva e b e c são iguais a 0. Sendo assim, a Equação de Poisson pode ser escrita da seguinte forma:

$$-a\nabla^2 u = f \ \ em \ \ \Omega \tag{5.2}$$

Seja $u : R^d \to R$, sabemos que o Operador de Laplace ∇ é definido como:

$$\nabla^2 u = \nabla \cdot (\nabla u) = \triangle u = \sum_{i=1}^{d} \frac{\partial^2 u}{\partial {x_i}^2} \tag{5.3}$$

Sendo assim, pode-se reescrever a Equação 5.2 eliminando a constante a ao multiplicarmos os fatores por $\frac{1}{a}$ e utilizarmos o Operador de Laplace:

$$-\triangle u = -\sum_{i=1}^{d} \frac{\partial^2 u}{\partial {x_i}^2} = f \ \ em \ \ \Omega \tag{5.4}$$

Dessa forma, a representação da Equação de Poisson depende da dimensão do problema. Para o nosso estudo de caso, escolhemos tratar problemas com 2 dimensões, R^2. Para problemas em R^2 a equação pode ser representada da seguinte forma:

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y) \ \ em \ \ \Omega \tag{5.5}$$

5.2 Aplicações da Equação de Poisson

As particularidades da matriz resultante da Equação de Poisson nos permitirão realizar demonstrações do uso dos componentes de aplicação propostos nesse

trabalho. Embora essa tenha sido a principal causa da escolha do problema para o estudo de caso, também levamos em consideração a importância do problema para aplicações que possuem requisitos de CAD.

A Equação de Poisson é uma EDP com bastante utilidade em aplicações científicas, mais precisamente nas áreas da eletrostática, engenharia mecânica e física teórica.

Podemos, por exemplo, através da Equação de Poisson encontrar o potencial elétrico de uma distribuição de carga. Para esse problema, podemos representá-lo da seguinte forma:

$$\triangle\varphi = -\frac{\rho f}{\varepsilon} \tag{5.6}$$

Onde φ é o potencial elétrico procurado, ρ é a densidade de carga livre e ε é a permissividade do meio.

Pode-se ainda calcular o potencial gravitacional Φ de cada ponto do espaço associado à densidade de distribuição de massa μ, da seguinte forma:

$$\triangle\Phi = 4\pi G\mu \tag{5.7}$$

Outro argumento da importância da Equação de Poisson é a sua utilização em aplicações DFC (Dinâmica dos Fluidos Computacional), que servem para realizar simulações numéricas dos processos físicos existentes em escoamentos. Além do mais, aplicações desse nicho são de grande importância para a sociedade, como aplicações de aerodinâmica de veículos terrestres e aéreos, de previsão do tempo, de planejamento de recursos hídricos e da indústria de petróleo.

Por exemplo, se desejarmos realizar uma simulação de um escoamento permanente de fluidos bidimensionais, teremos que utilizar a Equação de Poisson representada da seguinte maneira:

$$-\frac{\partial^2 v_x}{\partial x^2} - \frac{\partial^2 v_y}{\partial y^2} = 0 \tag{5.8}$$

Onde v_x é a velocidade no eixo x e v_y a velocidade no eixo y.

Portanto, percebemos que o uso da Equação de Poisson é bastante relevante em aplicações científicas, o que também justifica nossa escolha como estudo de caso.

5.3 Discretização da Equação de Poisson

O primeiro passo para resolver problemas representados pela Equação de Poisson é a discretização da equação, onde transforma o problema contínuo em um problema discreto. Geralmente, as matrizes resultantes da discretização possuem algumas propriedades particulares, o que nos permitirá realizar testes em nossos componentes de aplicação para o contexto específico com as propriedades da matriz resultante.

Para tanto, faremos uso do método das diferenças finitas para a discretização da Equação de Poisson bidimensional, com o qual obtemos:

$$\frac{\partial^2 u(x,y)}{\partial x^2} = \frac{u(x+1,y) - 2u(x,y) + u(x-1,y)}{h^2} \tag{5.9}$$

$$\frac{\partial^2 u(x,y)}{\partial y^2} = \frac{u(x,y+1) - 2u(x,y) + u(x,y-1)}{h^2} \tag{5.10}$$

Portanto, podemos reescrever a Equação de Poisson como:

$$\frac{u(x+1,y) - 2u(x,y) + u(x-1,y)}{h^2} + \frac{u(x,y+1) - 2u(x,y) + u(x,y-1)}{h^2} = f(x,y)$$

$$\frac{1}{h^2}(u(x-1,y) + u(x+1,y) - 4u(x,y) + u(x,y-1) + u(x,y+1)) = f(x,y)$$

Percebemos assim que a discretização da Equação de Poisson conduz a um sistema linear $Au = f_{n\cdot n \times n\cdot n}$ que terá a seguinte estrutura:

$$A \times u = \begin{bmatrix} D & I & 0 & 0 & 0 & \dots & 0 \\ I & D & I & 0 & 0 & \dots & 0 \\ 0 & I & D & I & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & I & D & I & 0 \\ 0 & \dots & \dots & 0 & I & D & I \\ 0 & \dots & \dots & \dots & 0 & I & D \end{bmatrix} \times \begin{bmatrix} u_{11} \\ u_{21} \\ \vdots \\ u_{n1} \\ u_{12} \\ u_{22} \\ \vdots \\ u_{n3} \\ \vdots \\ u_{nn} \end{bmatrix} = \begin{bmatrix} f_{11} \\ f_{21} \\ \vdots \\ f_{n1} \\ f_{12} \\ f_{22} \\ \vdots \\ f_{n3} \\ \vdots \\ f_{nn} \end{bmatrix} = f$$

Onde I é uma matriz identidade $n \times n$ e D é uma matriz $n \times n$ com uma

estrutura tridiagonal da seguinte forma:

$$D = \begin{bmatrix} -4 & 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & -4 & 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & -4 & 1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 1 & -4 & 1 & 0 \\ 0 & \dots & \dots & 0 & 1 & -4 & 1 \\ 0 & \dots & \dots & \dots & 0 & 1 & -4 \end{bmatrix}$$

Observando o sistema linear $A \times u = f$ descrito acima, percebemos que a matriz A possui uma distribuição esparsa dos seus elementos não-nulos, os quais estão dispostos em blocos organizados nas três diagonais principais. Essa propriedade, característica das matrizes ditas *tridigiadonais-em-blocos*, nos permitirá utilizar componentes de aplicação apropriados para solucionar sistema lineares que a possuem.

Vemos assim, que tanto a importância do problema como a sua adequação para nossos propósitos de testes justificam a nossa escolha da Equação de Poisson como estudo de caso.

5.4 Métodos para a Resolução da Equação de Poisson

Uma vez obtido o sistema linear resultante da discretização da Equação de Poisson, é necessário agora resolvê-lo através de métodos numéricos. Como sabemos, os métodos diretos e os métodos iterativos são utilizados para esse fim.

Os métodos diretos, como a fatoração LU e a fatoração de Cholesky, retornam o valor exato da solução após um número finito de passos. No entanto, geralmente os métodos diretos consomem mais tempo e espaço de memória do que os métodos iterativos. Tais métodos também não são adequados a problemas com matrizes esparsas, uma vez que causam o preenchimento da matriz esparsa por elementos não-nulos, onde antes existiam elementos nulos. Os métodos diretos são mais adequados para sistemas pequenos.

Nos métodos iterativos, como o de Jacobi e o Gauss-Seidel, a solução é refinada em iterações sucessivas, onde a cada nova iteração o erro da solução é reduzido. As iterações serão interrompidas quando algum critério fornecido pelo usuário, como o erro mínimo ou o número máximo de iterações, for satisfeito. Em geral, esses

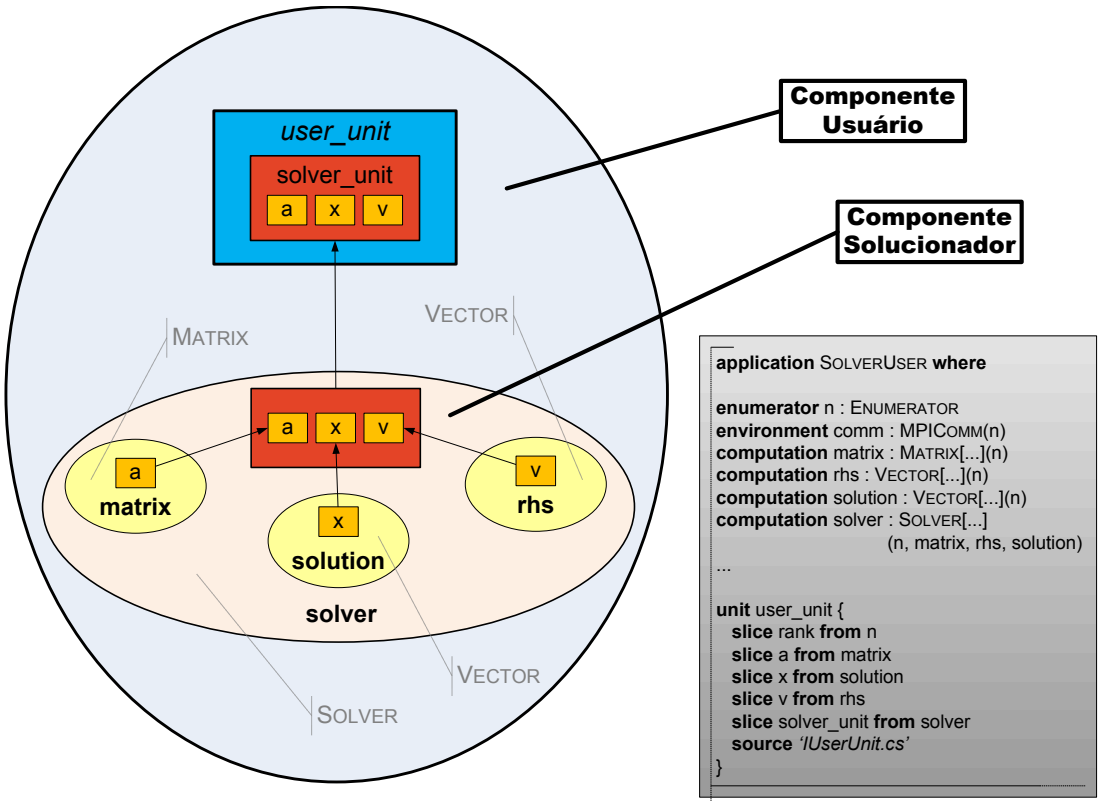


Figura 5.1: Usando o Componente SOLVER

métodos são mais rápidos e necessitam de menos memória do que os métodos diretos. Além do mais, tais métodos não alteram a estrutura da matriz do sistema linear. Portanto, os métodos iterativos são mais aplicáveis em sistemas lineares grandes e caracterizados por uma matriz esparsa.

Como vimos, a matriz resultante da discretização da Equação Poisson é esparsa, sendo assim, para o nosso estudo de caso, escolheremos a classe de métodos iterativos para aplicarmos em nossos componentes. Vimos também que a matriz do nosso sistema é bloco tridiagonal, o que nos permitirá escolher dentre os métodos iterativos aqueles que sejam específicos para esse tipo de matriz.

Para matrizes bloco-tridiagonais os métodos mais apropriados são de *Jacobi*, de *Gauss-Seidel*, gradiente conjugado e SSOR (Symmetric Successive Overrelaxation Method) [62]. Ressaltamos que foge do escopo deste trabalho a explicação de cada um desses métodos e o porquê de esses se comportam melhor do que outros métodos para solucionar sistemas lineares com matrizes bloco-tridiagonais.

Algoritmo 5.1: Esboço do Código da Classe que Implementa as Unidades do Componente SOLVER (C#)

```
1 public interface ISolverUnit<LIB, MTH, MPT, NUM, MAT, VEC, PRC> : IComputationUnit
```

```
2   where LIB:ILibraryUnit
3   where MTH:IMethodUnit
4   where MPT:IPropertyUnit
5   where NUM:INumericTypeUnit
6   where MAT:IMatrixUnit
7   where VEC:IVectorUnit
8   where PRC:IPreConditioner
9   {
10          IMatrixUnit<LIB, NUM, MPT> A {set;}
11          IVectorUnit<LIB, NUM> V {set;}
12          IVectorUnit<LIB, MTH, MPT> X {set;}
13          IMPIComm comm  {set;}
14  }
```

5.5 Uso do Componente de Aplicação Solver

Nesta seção, descrevemos alguns possíveis cenários do uso do componente de aplicação Solver para o problema da Equação de Poisson. Para tanto, apresentamos como o usuário deverá utilizar o componente Solver em uma aplicação. De agora em diante, chamaremos de **componente usuário** a aplicação ou componente que faz uso de um componente solucionador, do tipo Solver, para realizar seus propósitos. Para isso, o programador do componente usuário deve configurar, usando a linguagem HCL ou a linguagem visual do Front-End do HPE, um componente aninhado do tipo definido pelo componente abstrato Solver. Essa configuração está ilustrada na Figura 5.1, onde o componente aninhado **solver** representa o solucionador.

Ainda observando a Figura 5.1, o componente **solver**, por ser do tipo Solver, publica componentes aninhados que representam a matriz A e os vetores x e v do sistema $Ax{=}v$ a ser solucionado, respectivamente denominados **matrix**, **solution** e **rhs**. Por serem componentes aninhados públicos, o desenvolvedor terá acesso às suas funcionalidades particulares na programação das classes que representam as unidades do componente usuário, nomeadas user_unit na Figura 5.1, uma vez que suas unidades, nomeadas a, v e x na Figura 5.1, são fatias ditas públicas das unidades do componente solucionador, nomeadas solver_unit na Figura 5.1. De forma prática, a, v, e x são representados, no HPE, como propriedades públicas da classe que representa as unidades solver_unit no componente concreto, a qual deverá implemementar a interface ISolverUnit, esboçada na Figura 5.1, a qual representa as unidades solver_unit do componente abstrato Solver. Assim, o programador do

componente usuário deve acessar os métodos da interface de a e v para preencher os valores de entrada, bem como acessar os métodos da interface de x para ler o resultado, o que encontra-se ilustrado no código da Figura 5.3 (método go).

Algoritmo 5.2: Esboço do Código da Classe Base que Implementa as Unidades do Componente Usuário (C#)

```
1  /* NOTE: Automatically generated class.
2     The programmer do not need to access base classes */
3  public abstract class BaseUserUnitImpl : IUserUnit /* application kind */
4  {
5    protected ISolverUnit<...> solver_unit = null;
6    protected ISolverUnit<...> Solver { set { solver_unit=value; }}
7
8    protected ICSRMatrixUnit<...> a = null; /* ICSRMatrixUnit <: IMatrixUnit */
9    protected ICSRMatrixUnit<...> A { set { a=value; solver_unit.A=a; }}
10
11   protected IVectorUnit<...> v = null;
12   protected IVectorUnit<...> V { set { v=value; solver_unit.V=v; }}
13
14   protected IVectorUnit<...> x = null;
15   protected IVectorUnit<...> X { set { x=value; solver_unit.X=x; }}
16
17   protected IMPIComm comm = null;
18   protected IMPIComm Comm { set { comm=value; solver_unit.Comm=comm; }}
19 }
```

Os códigos C# apresentados nas Figuras 5.1, 5.2 e 5.3 têm o objetivo de oferecer uma visão mais prática da interface entre o componente usuário e o componente solucionador. A interface ISolverUnit está associada às unidades do componente abstrato Solver. As classes BaseUserUnitImpl e UserUnitImpl estão associadas às unidades do componente concreto usuário. Assumimos que o componente usuário é da espécie *aplicação*, embora poderia também ser da espécie *computação*. O código contém algumas simplificações para que o leitor concentre-se nos aspectos relevantes. Por exemplo, na classe BaseUserUnitImpl, nos abstraimos dos parâmetros de contexto aplicados às interfaces que definem os tipos das propriedades que representam as unidades dos componentes aninhados nas classes, respectivamente denominadas Solver, A, V, X e Comm. Este último representa o comunicador MPI, requerido por bibliotecas científicas paralelas implementadas sobre MPI. Devemos ainda ressaltar que somente a classe UserUnitImpl, dita uma classe de usuário (*user*

class) no HPE, é definida pelo usuário, sendo as demais geradas automaticamente a partir da configuração dos componentes envolvidos. Por esse motivo, visando oferecer um melhor nível de abstração ao programador do componente, a classe que representa uma unidade de um componente concreto é separada em duas classes, respectivamente chamadas *base class* (abstrata) e *user class*.

O componente Solver possui ainda parâmetros formais de contexto que governarão a escolha do solucionador apropriado, delimitando restrições relevantes como propriedades da matriz de entrada, o método utilizado para a solução do sistema e etc, enumerados na Tabela 4.2. O programador deve selecionar os parâmetros reais de contexto que suprirão os respectivos parâmetros formais que o interessam, obedecendo as restrições de limites de cada variável de contexto. Alguns parâmetros, ditos parâmetros livres, podem não ser supridos. Observando as Tabelas 4.3, 4.4, 4.5 e 4.6, podemos constatar que inúmeros são os cenários possíveis ao suprimos os parâmetros de contexto do componente Solver. Dessa forma, na Seção 5.6, que segue a esta, pretendemos descrever cenários possíveis para solucionar a Equação de Poisson, com o objetivo de mostrar os benefícios da definição do contexto no componente Solver e os benefícios da componentização das bibliotecas científicas.

Algoritmo 5.3: Esboço do Código da Classe de Usuário que Implementa as Unidades do Componente Usuário (C#)

```
1  /* NOTE: User-defined class.
2     Users configure the task performed by the component in user classes */
3  public class UserUnitImpl : BaseUserUnitImpl
4  {
5    /* Only override if there is some user-defined binding */
6    override public void setServices(Services services) {
7            base.setServices(services);
8            ...
9    }
10
11   /* the go method is required by the GoPort interface
12      provided by application components to be connected to the framework. */
13   public override void go() {
14         // Preencher e configurar a matriz a
15         ...
16         // Preencher e configurar o vector b
17         ...
18         // Configurar o pré-condicionador (se houver)
```

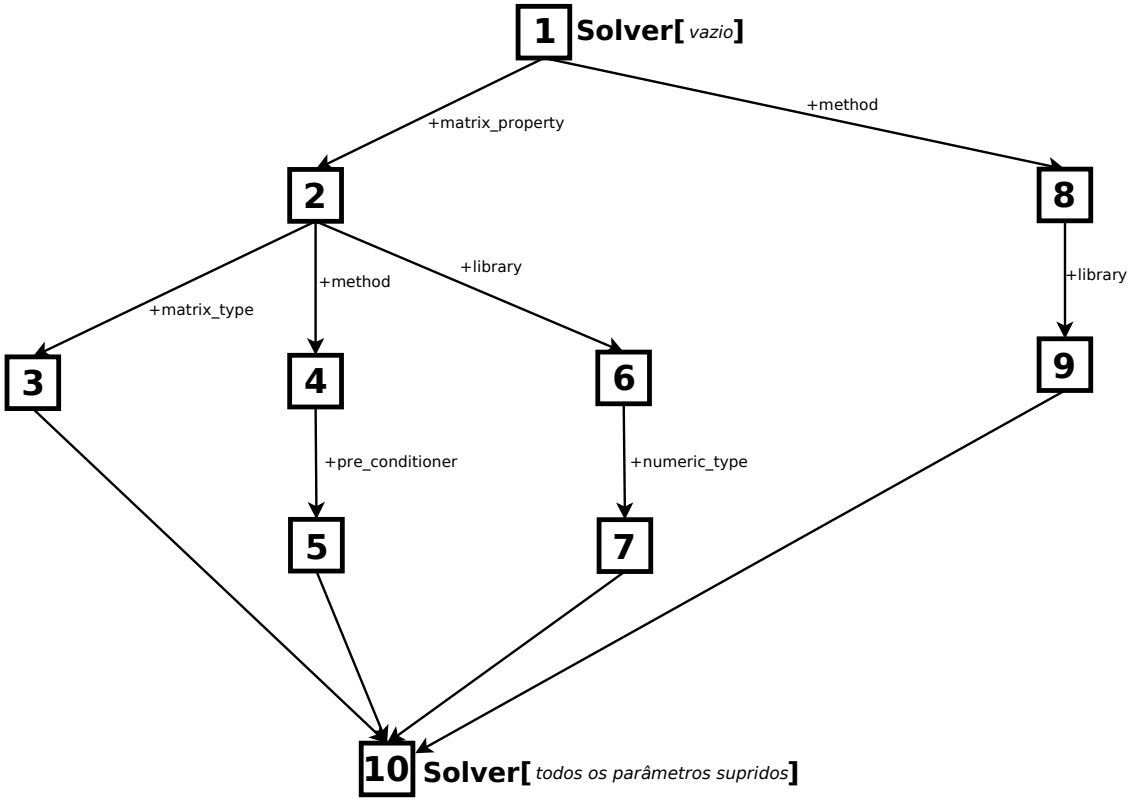


Figura 5.2: Cenários

```
19          /* NOTE: Os métodos de configuração do pré-condicionador
20             pertencem a interface do objeto solver_unit. */
21          ...
22
23          solver_unit.compute();
24
25          // Acessar o resultado do vector x
26          ...
27      }
28  }
```

Por fim, no código do método de execução das unidades do componente usuário (denominado *compute* para unidades de componentes da espécie *computação* e *go* para unidades de componentes da espécie *aplicação*), o desenvolvedor realiza uma invocação ao método *compute* do objeto solucionador, iniciando a computação da solução do sistema linear fornecido. Isso encontra-se ilustrado no código da Figura 5.3, onde uma chamada à execução do solucionador é exemplificada.

5.6 Escolha dos Parâmetros de Contexto: Análise de Cenários

Esta seção tem por objetivo demonstrar como a utilização dos parâmetros de contexto do componente SOLVER pode ser empregada para oferecer um maior nível de abstração aos usuários sobre o uso de bibliotecas científicas, permitindo a escolha e troca de solucionadores segundo as necessidades do componente usuário e obedecendo às limitações da plataforma HPE. Serão apresentados um total de dez cenários, nos quais demonstra-se possíveis variações na configuração dos parâmetros de contexto de SOLVER e suas consequências. Os cenários e suas relações estão ilustrados na Figura 5.2.

5.6.1 Cenário 1: Ausência de Restrições de Contexto

SOLVER []

Neste cenário, nenhum parâmetro de contexto é suprido. Todos são parâmetros livres. Dessa forma, o HPE não terá como ser guiado para escolher o solucionador (componente concreto) mais adequado para o problema, deixando a escolha do solucionador arbitrária.

Para o problema em questão, não é recomendável a escolha arbitrária do componente solucionador adequado. Poderia acontecer que o HPE escolhesse um método de solução específico para matrizes pentadiagonais, quando a matriz é, no caso da Equação de Poisson, bloco-tridiagonal. Tal escolha, poderia acarretar comprometimento do desempenho na execução, bem como o uso desnecessário de memória. Caso a intenção do usuário seja dizer que a matriz de seu sistema é de um tipo genérica, então deverá usar o parâmetro de contexto *matrix_property* para informar isso. Portanto, afirmamos ser muito importante, para o contexto específico de solução de sistemas lineares, que pelo menos seja determinado a propriedade da matriz através do parâmetro de contexto *matrix_property*, como veremos nos cenários a seguir.

Com relação a matriz e ao vetor de entrada e saída, a escolha arbitrária fará com que o acesso a essas estruturas de dados só seja possível por meio da interface genérica dos componentes MATRIX e VECTOR, acarretando uma possível sobrecarga para o solucionador escolhido para reorganização dos dados de entrada obedecendo as restrições dos componentes concretos de MATRIX e VECTOR selecionados pelo

sistema. Entretanto, na maioria dos casos, essa sobrecarga deve ser desprezível, quando a simplicidade da interface genérica seria justificada.

5.6.2 Cenário 2: Especificando Propriedades da Matriz de Entrada

$$\textsc{Solver}\,[\mathit{matrix\_property} = \textsc{BlockTridiagonalMatrix}]$$

Neste cenário, apenas o parâmetro de contexto *matrix_property* é suprido, especificando que a matriz característica do sistema linear tem a propriedade bloco-tridiagonal. Dessa forma, o HPE irá adotar, dentre os componentes concretos do tipo SOLVER implantados na plataforma, aquele que supre o parâmetro *matrix_property* com BLOCKTRIDIAGONALMATRIX, ou algum dos seus supertipos, quando não estiver disponível. Supõe-se que o componente concreto escolhido é o que melhor se adequa para o contexto de sistemas lineares com matriz bloco-tridiagonal. Então, o próprio HPE irá optar, orientado pelo parâmetro de contexto *matrix_property*, o método de solução e, portanto, a biblioteca mais apropriada para esse cenário.

Como no cenário anterior, será utilizada uma interface genérica para a entrada dos valores da matriz do sistema. Espera-se nesse caso que o componente concreto solucionador escolhido utilize uma estrutura de dados especializada para lidar com matrizes esparsas bloco-tridiagonais, embora tal requisito não seja mandatório. O componente que implementa essa estrutura de dados será encarregado de mapear a entrada do usuário através da interface genérica para a estrutura especializada.

5.6.3 Cenário 3: Restringindo o Formato da Matriz Esparsa

$$\textsc{Solver}\left[\begin{array}{c}\mathit{matrix\_property} = \textsc{BlockTridiagonalMatrix},\\ \mathit{matrix\_type} = \mathrm{CSR}\end{array}\right]$$

Neste cenário, os parâmetros de contexto *matrix_property* e *matrix_type* são supridos. Assim, o componente SOLVER é qualificado para solucionar sistemas lineares que possuam matrizes bloco-tridiagonais e que sejam armazenadas usando o formato CSR (*Compressed Sparse Row*). Semelhante ao cenário anterior, o HPE irá utilizar os parâmetros de contexto para adotar um componente implantado na

plataforma que seja mais adequado ao contexto fornecido pelo desenvolvedor.

Assim como existem métodos em cada biblioteca que são específicos de acordo com as propriedades da matriz (bloco-tridiagonal, triangular, densa, etc), a implementação do método pode ser otimizada para um certo formato de armazenamento da matriz esparsa, como os descritos na Tabela 4.3 (além desses, existem outros menos comuns). Sendo assim, a definição do formato da matriz pode ter influência relevante na eficiência e no consumo de memória da execução do método. É tarefa do desenvolvedor escolher o tipo de armazenamento adequado para o seu problema. Caso não seja o tipo mais apropriado, os resultados poderão não ser os esperados. Todavia, uma vez não estando satisfeito com o tipo de armazenamento escolhido, o desenvolvedor tem liberdade e facilidade para escolher outro tipo e comparar os resultados. A facilidade de troca de contexto (no caso, troca do formato da matriz) será abordada mais adiante.

Embora a representação da matriz seja a mais adequada, pode acontecer que não exista nenhum componente SOLVER implantado que trate matrizes com a representação fornecida pelo usuário. Dessa forma, poderia acontecer de ser usado um componente SOLVER que acesse a representação genérica da matriz, por meio da resolução dinâmica de componentes, o que acarretaria uma perda de desempenho, não necessariamente significativa, mas sem afetar a acurácia dos resultados.

Vemos assim que a definição do parâmetro de contexto *matrix_type*, além de *matrix_property*, constitui-se ferramenta importante para otimizar o desempenho do componente usuário, fazendo suposições sobre o formato da matriz de entrada. Consideramos isso como uma grande vantagem, uma vez que devido ao grande número de métodos de solução e de bibliotecas científicas, nem sempre o desenvolvedor sabe, dentre todos os métodos e bibliotecas, aqueles que mais se adequam ao seu problema. No entanto, esse serviço não restringe o desenvolvedor a utilizar apenas os métodos e as bibliotecas escolhidas pelo HPE. O desenvolvedor tem total liberdade para escolher o método e a biblioteca que julgar mais eficiente. Veremos isso nos próximos cenários.

5.6.4 Cenário 4: Restringindo o Método de Solução

$$\text{SOLVER}\left[\begin{array}{c} \textit{matrix\_property} = \text{BLOCKTRIDIAGONALMATRIX}, \\ \textit{method} = \text{GAUSSSEIDEL} \end{array}\right]$$

Neste cenário, o método de solução é escolhido pelo programador do componente usuário, através do parâmetro de contexto *method*, além da propriedade da matriz. Tal contexto encara a situação em que dadas as propriedades da matriz de entrada o desenvolvedor julga conhecer o método de solução mais apropriado para o sistema. Dessa forma, é esperado que o usuário combine o método de solução com a representação da matriz para o qual aquele método é adequado, e que um componente para esta combinação esteja implantado na plataforma.

5.6.5 Cenário 5: Incluindo um Pré-Condicionador

$$\textsc{Solver}\left[\begin{array}{c} \textit{matrix\_property} = \textsc{BlockTridiagonalMatrix}, \\ \textit{method} = \textsc{GaussSeidel}, \\ \textit{pre\_conditioner} = \text{I\_PLUS\_R} \end{array}\right]$$

Neste cenário, o programador determina o uso de um pré-condicionador aplicado ao método Gauss-Seidel, através do parâmetro de contexto *pre_conditioner*. No exemplo, supõe-se o uso do pré-condicionador I+R [68], um dos muitos propostos para o método Gauss-Seidel. Este cenário pode acontecer quando o desenvolver desejar otimizar ainda mais sua solução através do uso de pré-condicionadores do sistema. Como sempre, espera-se a escolha de combinações entre esses parâmetros que façam sentido na prática.

5.6.6 Cenário 6: Restringindo a Biblioteca a ser Utilizada

$$\textsc{Solver}\left[\begin{array}{c} \textit{library} = \text{PETSc}, \\ \textit{matrix\_property} = \textsc{BlockTridiagonalMatrix} \end{array}\right]$$

Neste cenário, o desenvolvedor deseja que o componente utilize a biblioteca PETSc para resolver um sistema linear com uma matriz bloco-tridiagonal, suprindo-se os parâmetros de contexto *library* e *matrix_property*, respectivamente. Portanto, difere dos demais por não delegar essa escolha ao sistema de resolução de componentes. É esperado que seja encontrado no ambiente um componente que aplique o método mais apropriado na biblioteca PETSc para matrizes bloco-tridiagonais. Caso não seja encontrado nenhum componente específico para

esse cenário, o contexto pode ser generalizado em ambos os parâmetros, embora na prática só faça sentido para o segundo, uma vez que a generalização de PETSc é Library, o qual não constitui uma biblioteca concreta.

5.6.7 Cenário 7: Informando o Tipo Numérico do Sistema

$$\text{Solver}\left[\begin{array}{c} library = \text{PETSc}, \\ matrix\_property = \text{BlockTridiagonalMatrix} \\ numeric\_type = \text{Integer} \end{array}\right]$$

Neste cenário, o parâmetro de contexto *numeric_type* é empregado para restringir o tipo numérico do sistema, no caso números inteiros. Semelhante ao cenário anterior, onde o componente é qualificado para solucionar sistemas lineares com matrizes bloco-tridiagonais com a biblioteca PETSc, além disso é definido o tipo numérico dos valores não nulos da matriz como inteiros. Sendo assim, o HPE irá buscar um componente implantado na plataforma cuja implementação seja otimizada para cálculos numéricos com números inteiros, possivelmente usando mecanismos de baixo nível. Sendo assim, a definição do parâmetro *numeric_type*, embora menos comum que os demais parâmetros, também poderia afetar o desempenho do programa.

Como nos cenários anteriores, caso não seja encontrado nenhum componente especializado com inteiros, o parâmetro será generalizado. Tal generalização poderia ser Integer $<:$ SinglePrecisionFloatPoint $<:$ DoublePrecisionFloatPoint, onde $<:$ denota a relação de subtipos. Nesse caso, quando não encontrando nenhum método de solução especializado para inteiros, então uma implementação especializada para números de ponto flutuante, de precisão simples ou dupla, poderia ser utilizada.

5.6.8 Cenário 8: Restringindo Apenas o Método de Solução

$$\text{Solver}\,[method = \text{GaussSeidel}]$$

Neste cenário apenas o parâmetro de contexto *method* é suprido para informar o método específico que será utilizado, diferenciando-se do Cenário 4, no qual a

propriedade da matriz de entrada também é informada. Neste caso, existindo na plataforma um componente SOLVER que tenha o método fornecido pelo contexto, então tal componente será adotado independente da biblioteca que implemente este método ou qualquer propriedade da matriz de entrada. Esse cenário captura a situação onde o programador sabe exatamente que método aplicar para resolver o problema, pouco conhecendo sobre a estrutura da matriz de entrada, sobre a qual prefere abstrair-se.

5.6.9 Cenário 9: Restringindo a Biblioteca e o Método de Solução

$$\text{SOLVER} \left[\begin{array}{c} \mathit{library} = \text{PETSC}, \\ \mathit{method} = \text{GAUSSSEIDEL} \end{array} \right]$$

Neste cenário, acrescenta-se ao cenário 8 a restrição sobre a biblioteca que será empregada. É esperado que o usuário tenha conhecimento se a biblioteca pretendida implemente o método escolhido. Este é um uso menos abstrato do solucionador, onde o programador explicitamente delimita características específicas do solucionador pretendido, abstraindo-se da definição das propriedades do problema. Podemos afirmar que se trata de uma forma de utilização mais próxima ao uso convencional dos solucionadores em um contexto não orientado a componentes.

5.6.10 Cenário 10: Uso de Todos os Parâmetros de Contexto

$$\text{SOLVER} \left[\begin{array}{c} \mathit{library} = \text{PETSC}, \\ \mathit{matrix\_property} = \text{BLOCKTRIDIAGONALMATRIX} \\ \mathit{numeric\_type} = \text{DOUBLEPRECISIONFLOATPOINT} \\ \mathit{matrix\_type} = \text{CSR} \\ \mathit{method} = \text{GAUSSSEIDEL} \\ \mathit{pre\_conditioner} = \text{CW} \end{array} \right]$$

Neste cenário, todos os parâmetros de contexto são selecionados. O programador do componente usuário tem total controle sobre a escolha do solucionador, confiando no sistema de resolução de componentes para escolher versões mais genéricas do

contexto quando um solucionador específico para o contexto informado não esteja disponível. Note que a restrição sobre a biblioteca diminui bastante a amplitude de escolhas do sistema de resolução. Recomenda-se como boa prática, para o domínio de solução de sistemas lineares, somente restringir a escolha da biblioteca quando o usuário tiver total conhecimento sobre as potencialidade de cada biblioteca suportada. Caso contrário, é desejável delegar a escolha da biblioteca para o sistema de resolução, deixando livre o parâmetro *library*.

5.7 Reconfiguração Orientada pelos Parâmetros de Contexto

Como já mencionado em capítulos anteriores, a intenção do nosso trabalho não restringe-se a construção de um invólucro sobre as bibliotecas científicas, mas oferecer um meio pelo qual usuários de bibliotecas científicas possam usufruir destas com benefícios advindos do uso da programação baseada a componentes.

Uma das vantagens do uso de componentes é a possibilidade de reconfiguração de componentes, cujo grau de suporte varia bastante entre plataformas de componentes, desde uma abordagem estática, na qual alterações não podem ser realizadas durante a execução do componentes, até abordagens dinâmicas, onde os componentes podem suportar diferentes graus de autonomia.

No HPE, as configurações de componentes são estáticas, com exceção do caso de conexões entre componentes da espécie *serviço* (Service), recentemente incorporada para suporte ao modelo CCA [26]. O caráter estático das configurações do HPE é justificado pela necessidade de garantir o alto desempenho das conexões entre os componentes, bem como minimizar qualquer sobrecarga sobre sua execução. De fato, a possibilidade de reconfiguração dinâmica não é considerada pelos proponentes do HPE um requisito relevante no contexto da programação paralela, assumindo importância somente na ligação entre programas paralelos, o que é suportado pelos componentes da espécie *serviço*, os quais permitem que componentes da espécie *aplicação* estabeleçam conexões dinamicamente reconfiguráveis, por meio de ligações que seguem o padrão CCA.

Logo, não há no HPE mecanismo hoje implementado para suportar a troca dinâmica de solucionadores, embora isso seja possível indiretamente, encapsulando-se um componente SOLVER em um componente da espécie aplicação que oferecesse as funcionalidades do solucionador através de porta *provides* do padrão CCA, suportadas graças aos componentes da espécie *serviço*. Essa possibilidade não foi exercitada nesta dissertação, mas não há desafio técnico para

sua implementação na versão atual do HPE, sendo uma mera consequência do projeto dessa plataforma.

Entretanto, o HPE facilita algumas formas de reconfiguração estática que não são suportadas por bibliotecas científicas nativas, através da reconfiguração de variáveis de contexto. Por exemplo, suponha que, utilizando diretamente uma biblioteca científica, desejamos criar e imprimir uma matriz esparsa armazenada segundo o formato CSR (*compressed row format*). Inicialmente, deve-se criar uma matriz CSR e chamar uma função específica de impressão de matrizes CSR. No entanto, quando, por alguma circunstância, precisamos mudar o tipo de armazenamento da matriz para CSC (*compressed column format*), precisaríamos modificar duas funções do programa usuário para ser possível imprimir a matriz no formato CSC. Por outro lado, se estivéssemos usando as bibliotecas como componentes de softwares, teríamos o componente MATRIX, para criar matrizes, e o componente PRINTMATRIX, para imprimir matrizes. Então, se desejamos criar e imprimir uma matriz CSR, basta utilizar o componente MATRIX, informar seu tipo de armazenamento através dos parâmetros de contexto, e depois utilizar o componente PRINTMATRIX passando a matriz já criada. Se pelas mesmas circunstâncias anteriores, precisamos mudar o tipo de armazenamento da matriz para CSC, agora bastaríamos apenas mudar um parâmetro de contexto do componente MATRIX. Nesse exemplo simples, deve-se observar que realizar mudanças no contexto do problema é muito mais fácil se a aplicação for construída através de componentes de software.

Em nossa proposta, essa vantagem é mais notável quando outros parâmetros de contexto precisam ser modificados, como os parâmetros referentes ao método de solução, pré-condicionador e à biblioteca científica. Certas mudanças de contexto consideradas relevantes para o domínio de solucionadores de sistemas lineares são discutidas a seguir.

5.7.1 Reconfiguração do Método de Solução e do Pré-Condicionador

Nossa proposta oferece ao desenvolvedor a flexibilidade para realizar trocas de métodos de solução que estejam disponíveis na plataforma, uma vez que a escolha do método de solução é determinada pelo parâmetro de contexto *method*, o qual pode ser permutado de forma transparente ao usuário por ser da espécie qualificador.

Portanto, o usuário de nossos componentes poderá eleger um método de solução para seu problema e caso não fique satisfeito com o resultado, a escolha de um outro método de solução que seja adequado ao seu propósito pode ser feito de forma

simples, substituindo-se o parâmetro real de contexto *method*. De forma análoga, também podemos realizar mudanças do pré-condicionador através do parâmetro de contexto *pre_conditioner*. É verdade que o desenvolvimento de aplicações através de bibliotecas científicas também permite a troca do método de solução de forma simples, na maioria dos casos bastando a mudança de apenas uma linha de código. No entanto, o desenvolvedor precisa conhecer bem o uso da biblioteca e precisará buscar no corpo do código a função que define o método de solução e realizar a troca. Já com o uso de nossos componentes, o usuário não precisará conhecer bem a biblioteca e nem pesquisar entre linhas de código as funções que precisam ser substituídas. Basta apenas escolher outro método de solução ou pré-condicionador através do parâmetro de contexto.

Argumentamos que a abordagem proposta aumenta a produtividade dos desenvolvedores de aplicações científicas, além de otimizar o desempenho das aplicações, pois os componentes propostos oferecem a possibilidade que sejam realizadas avaliações experimentais entre os métodos de solução para a escolha daquele que mais se adequa ao problema determinado. É importante ressaltar que uma biblioteca não possui todos os métodos de solução. Então, o ganho de produtividade será ainda maior quando o usuário de nossos componentes desejar escolher métodos de bibliotecas diferentes. Uma vez que se estivesse usando apenas as bibliotecas científicas, teria que criar um novo programa para cada biblioteca.

5.7.2 Reconfiguração da Biblioteca Científica

Assim como a troca do método de solução, a substituição da biblioteca científica utilizada também acontecerá de forma transparente ao usuário. Isso é possível pois, a biblioteca que será utilizada por nosso componente é definida através do parâmetro de contexto *library*, da espécie qualificador.

Infelizmente, se um desenvolvedor que utilize apenas bibliotecas científicas desejar alterar a biblioteca utilizada do seu programa já criado, terá grande complicações. Precisará aprender o uso da nova biblioteca que pretende usar, tarefa que potencialmente requer muito trabalho, e precisará reescrever praticamente todo o programa com a nova biblioteca. Por exemplo, se tivéssemos escrito uma solução da equação de Poisson através da biblioteca Hypre, mas que por algum motivo desejássemos obter a solução do mesmo problema através da biblioteca PETSc. Então, além de ter que aprender uma nova sintaxe teríamos que mudar várias linhas de código do programa já escrito. No entanto, se utilizássemos os componentes de

software propostos neste trabalho, precisaríamos mudar apenas um parâmetro para substituir a biblioteca Hypre pela biblioteca PETSc em nosso programa de solução da equação de Poisson.

5.7.3 Reconfiguração da Matriz

É possível também realizar a reconfiguração do contexto da matriz do sistema sem muitas dificuldades. Como mostrado nos cenários da Seção 5.6, existem três tipos de mudanças no contexto da matriz do sistema: o tipo de armazenamento da matriz, o tipo de dado dos valores da matriz e as propriedades da matriz, em geral relacionado o padrão da distribuição dos valores não-nulos da matriz esparsa.

Nos três casos citados, a troca de contexto acontece de forma transparente ao usuário. Isso é possível, pois, o componente MATRIX possui uma interface genérica para a criação de matrizes, na qual, dependendo do contexto definido, um método de criação de matrizes específico para o dado contexto pode ser invocado. Assim, o desenvolvedor pode alternar entre diferentes tipos de armazenamento, diferentes tipos de dados dos valores da matriz e diferentes propriedades da matriz. Dentre essas mudanças, a mais provável de acontecer é a substituição do tipo de armazenamento, pois sem modificar o problema inicial pode-se alcançar resultados melhores com a troca do tipo de armazenamento. Já a mudança do tipo de dado dos valores da matriz e a mudança das propriedades da matriz é mais raro de acontecer na prática, uma vez que se mudássemos esses contextos estaríamos modificando o problema em questão.

5.8 Avaliação de Desempenho

Com mencionamos no inicio deste capítulo não pretendemos, com este estudo de caso, realizar uma avaliação rigorosa de desempenho dos componentes, pois o custo das ligações do código C# ao código nativo e a sua componentização, comparado ao alto custo de computação da solução de grandes sistemas lineares, pode ser considerado desprezível. Entretanto, para constatar essa afirmativa realizamos alguns testes de desempenho sobre os componentes de aplicação propostos neste trabalho.

Para tanto, optamos por comparar o desempenho de execução do componente de aplicação SOLVER, que utiliza a biblioteca científica Hypre, com o desempenho de execução da biblioteca Hypre sem componentes. Para isso, elegemos o problema da Equação de Poisson de duas dimensões para a realização dos testes. As aplicações

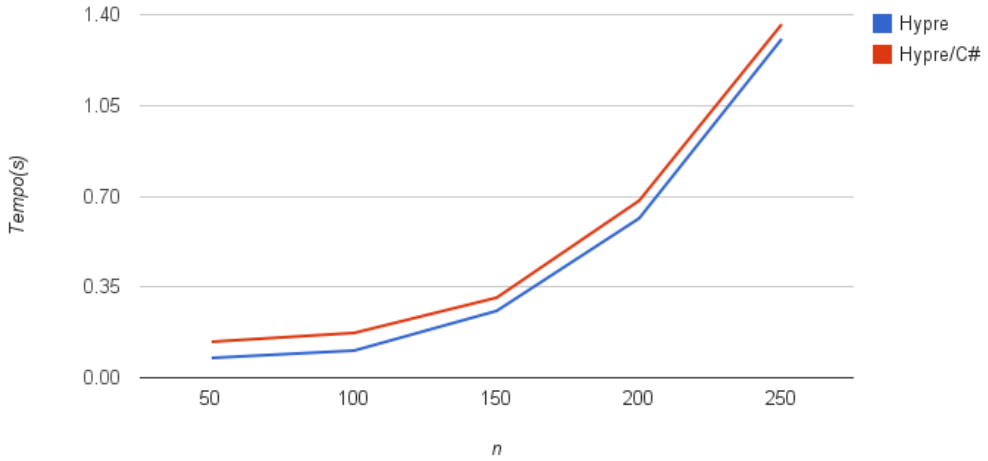


Figura 5.3: Desempenho da solução do problema da Equação de Poisson com 1 processador

foram executadas em uma máquina com processador Intel i5-750(*Quad-core*), com 2Gb RAM e com sistema operacional Unix.

Em nossos testes optamos por avaliar problemas com n iguais a 50,100,150,200 e 250 o que resulta em sistemas lineares com matrizes tridiagonais-em-bloco de dimensões 2500×2500, 10000×10000, 22500×22500, 40000×40000 e 62500×62500 respectivamente. Os procedimentos de avaliação foram repetidos para até três processadores.

Os resultados dos testes comparativos estão apresentados através das Figuras 5.3,5.4 e 5.5 que exibem os testes de desempenho para 1 processador, 2 processadores e 3 processadores respectivamente. Onde a curva de cor azul representa o tempo de execução da solução da Equação de Poisson para diversos tamanhos através da biblioteca Hypre, e de forma análoga, a curva de cor vermelha representa o tempo total de execução através dos componentes # que realizam chamadas à biblioteca Hypre. Ainda nessas figuras, o eixo vertical marca o total de tempo em segundos para a execução de algum problema de tamanho n, marcado pelo eixo horizontal.

Observando os resultados dos testes comparativos constatamos uma sobrecarga constante ao tempo total de execução da solução. Essa sobrecarga é devida à ligação entre código C# ao código nativo e sua componentização. Observamos ainda que tal sobrecarga pode ser considerada desprezível comparada ao tempo total de execução da solução realizada sem o uso de componentes. Uma vez que tal sobrecarga não

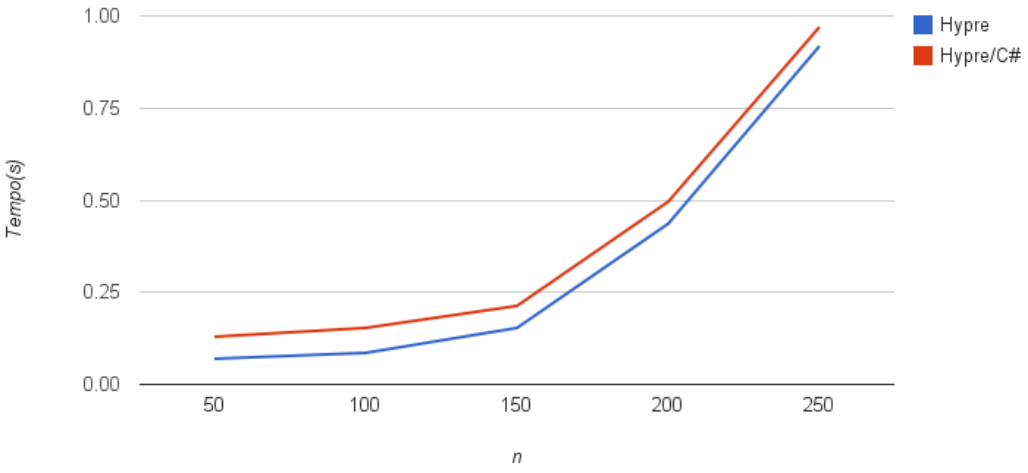


Figura 5.4: Desempenho da solução do problema da Equação de Poisson com 2 processadores

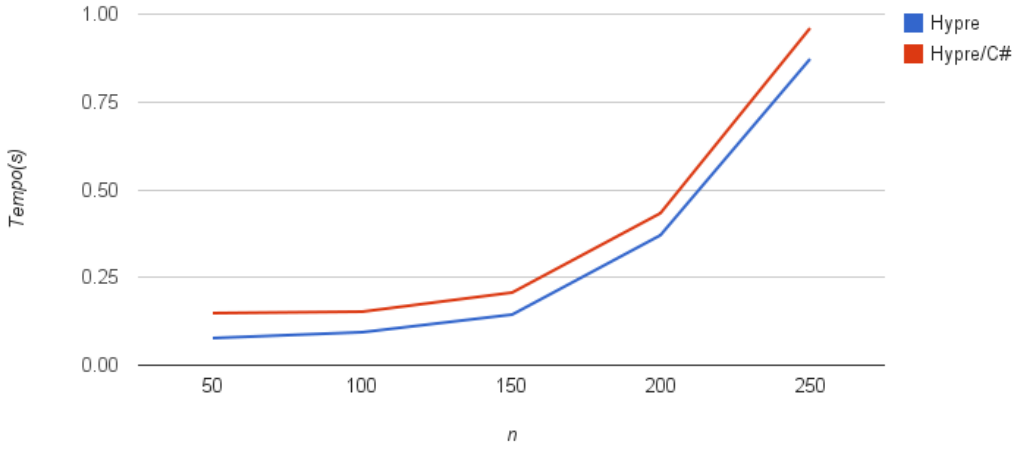


Figura 5.5: Desempenho da solução do problema da Equação de Poisson com 3 processadores

Tabela 5.1: Sobrecarga de tempo de execução pelo uso de componentes de software

Componente	Média de Sobrecarga
COMPONETE-#	0,06 segundo
CCA-LISI	0,1 segundo
NUMERICAL PLATON	0,8 segundo

alcança sequer 0,09 segundos. Dessa forma, constatamos a veracidade do que já havíamos afirmado no inicio do capítulo.

Observando ainda os trabalhos relacionados ao nosso trabalho, consideramos relevante ressaltar que nossos componentes de softwares apresentaram uma sobrecarga menor do que os componentes de softwares CCA-LISI, que apresentaram sobrecarga entre 0,07 e 0,13 segundos, e do que os componentes de softwares da ferramenta Numerical Platon, que apresentaram sobrecarga entre 0,2 e 1,4 segundos. Uma vez que nossos componentes de softwares apresentaram sobrecarga entre 0,04 e 0,08 segundos conforme apresentado ta Tabela 5.1.

Capítulo 6

Conclusões e Propostas de Trabalhos Futuros

A importância das aplicações provenientes dos domínios das ciências computacionais e engenharias tem crescido de forma evidente na última década, sendo hoje consideradas propulsoras da inovação tecnológica na indústria e do avanço do conhecimento científico na academia, fatos que evidenciam sua notória influência sobre o modo de vida da sociedade contemporânea. Novas técnicas computacionais aplicadas à simulação de sistemas naturais e artificiais e à descoberta científica tem sido especialmente possibilitadas com a rápida evolução das tecnologias que oferecem recursos de processamento de alto desempenho, assim como das novas tecnologias que permitem a integração de softwares geograficamente distribuídos. Estas últimas, tem oferecido suporte a um grau de colaboração e integração de recursos, humanos e computacionais, nunca antes alcançado para solução de grandes desafios científicos e tecnológicos.

Tendo em vista esse contexto, esta dissertação apresentou contribuições na integração de tecnologias recentemente empregadas em aplicações que demandam por CAD (Computação de Alto Desempenho), em especial os componentes de software, com tecnologias tradicionalmente utilizadas nesse contexto, como as plataformas distribuídas de computação paralela e as bibliotecas científicas de propósito especial.

A tecnologia de componentes tem sido empregada com sucesso em várias aplicações nas áreas de ciências computacionais e engenharia, especialmente como consequência dos esforços da comunidade envolvida com a especificação e disseminação dos modelos de componentes CCA e Fractal/GCM. A existência de

uma comunidade firmemente envolvida com o propósito de alavancar o uso de componentes em CAD tem possibilitado o surgimento de *workshops* especializados nesse tema, como o CBHPC (*Workshop on Component Based High Performance Computing*), o qual se encontra em sua terceira edição no ano de 2010, fruto da fusão dos eventos Compframe *(Workshop on Components and Frameworks for High Performance Computing*) e HPC-GECO (*HPC Grid programming Environments and COmponents*), respectivamente organizados pelas comunidades de pesquisadores em torno dos modelos CCA e Fractal/GCM, respectivamente. Tal fato evidencia a importância atribuída a esta linha de pesquisa emergente nos últimos anos. O trabalho de nosso grupo de pesquisa tem apresentado contribuições nessa área no que concerne ao problema da integração do processamento paralelo aos componentes, reconhecendo desafios ainda não tratados nesse contexto.

As biblioteca científicas tem sido empregadas em aplicações científicas e de engenharia desde os primórdios da história da computação. Podemos citar como pioneiros os trabalhos de Grace Hopper e seus colegas na Universidade de Harvard, na década de 1940, quando propuseram as primeiras biblioteca de subrotinas matemáticas, na tentativa de aumentar a produtividade na programação do Mark I, um dos primeiros computadores de que se tem notícia a ser aplicado em problemas reais no auge do esforço da marinha norte-americana na II Guerra Mundial.

A iniciativa pioneira de Grace Hopper levou a outros desenvolvimentos, como o surgimento dos primeiros compiladores e linguagens de programação de alto nível, alavancando o conceito que chamavam naquela época de *programação automática*. Desde então, várias bibliotecas científicas tem sido utilizadas pelas comunidades científica e de engenharia, a maioria das quais escritas nas linguagens C e Fortran, implementando os algoritmos mais eficientes para solução de problemas diversos, muitas vezes orientadas às plataformas de execução paralela, sendo validadas ao longo das décadas pela própria utilização, levando ao seu conceito próprio de corretude e confiabilidade por *reputação*.

Este trabalho reconhece o valioso legado dessas bibliotecas, enfatizando como seu objetivo principal oferecer meios de integrá-las às plataformas de componentes contemporâneas. Em particular, abordamos a integração a uma plataforma de componentes paralelos chamada HPE (*Hash Programming Environment*), proposta pelo nosso grupo de pesquisa com o intuito de lidar com as características intrínsecas de plataformas distribuídas modernas de computação paralela. A plataforma HPE está atualmente implementada sobre a plataforma de execução virtual Mono,

adicionando o desafio da integração entre o código gerenciado, sobre o qual a plataforma de componentes está implementada, e o código nativo, sobre o qual a maioria das bibliotecas científicas encontram-se implementadas.

6.1 Contribuições

Enfatizamos como a principal contribuição desse trabalho a especificação de um método para integrar bibliotecas científicas à plataforma HPE, descrita no Capítulo 4, a qual foi exemplificada e validada aplicando-a ao problema de integração de bibliotecas do domínio da solução de sistemas lineares, cuja gama de aplicações nas ciências e engenharia é de conhecimento amplo. Portanto, isso caracteriza o produto dessa integração também como contribuição importante desta dissertação.

Devemos ainda ressaltar algumas ideias de melhoramentos que surgiram para a plataforma HPE, em consequência de dificuldades enfrentadas para implementação deste trabalho, como o uso de parâmetros de contexto livres e a ideia de introduzir dependências funcionais entre parâmetros de contexto de um componente abstrato, as quais propomos denominar *dependências contextuais*. Esta última possibilidade não foi discutida ou utilizada nesta dissertação, tendo em vista a pouca maturidade dessa ideia na conclusão desta dissertação, mas o citamos como trabalhos futuros a serem abordados.

Também não foi possível enfrentar o problema da troca dinâmica de solucionadores, mesmo utilizando os componentes CCA agora suportados pelo HPE, embora consideramos pouco relevante para o objetivo primordial de validar o método de integração, principal objeto desta dissertação.

6.1.1 Considerações sobre Avaliação de Desempenho

Ressaltamos que o estudo de caso apresentado no Capítulo 5 teve como objetivo demonstrar a usabilidade dos componentes de aplicação propostos no Capítulo 4, dessa forma apresentamos apenas algumas avaliações de desempenho não tão rigorosas.

De fato, optamos por incluir uma avaliação de desempenho dos componentes não tão criteriosa nesta dissertação, por a considerarmos pouco relevante frente aos desafios enfrentados com a integração das bibliotecas nativas na plataforma Mono, apesar do suporte que essa plataforma oferece para tal finalidade, bem como frente ao esforço empregado no estudo das bibliotecas PETSc, HYPRE e SuperLU, assim como outras que foram preliminarmente analisadas com a finalidade de escolher um

subconjunto representativo de bibliotecas para solução de sistemas lineares.

A menor relevância deve-se ao fato de que a plataforma HPE foi projetada de forma a não inserir sobrecarga na execução das computações ou conexões entre componentes, como evidenciado na Seção 2.13.1. Dessa forma, o desempenho de um programa paralelo baseado em componentes HPE cujo tempo de processamento é dominado por chamadas a subrotinas de uma biblioteca científica componentizada é muito próximo ao mesmo programa paralelo escrito na linguagem nativa e utilizando o MPI. De fato, o custo adicional deve-se prioritariamente ao custo do encapsulamento das subrotinas da biblioteca na plataforma de execução virtual, medido pelo custo de chamadas da máquina virtual ao código nativo. Porém, com o desenvolvimento na implementação de máquinas virtuais motivado pela grande disseminação de linguagens como Java e C#, tal custo é hoje insignificante frente à complexidade computacional das operações implementadas pelas subrotinas solucionadoras de bibliotecas para solução de sistemas lineares sobre computadores paralelos, o que é de fato que justifica sua paralelização.

Embora não seja o objetivo deste trabalho, constatamos com algumas avaliações que os componentes # possui uma sobrecarga desprezível comparada ao tempo de computação das bibliotecas científicas. Além do mais, os componentes # apresentaram sobrecarga inferior aos componentes de softwares dos principais trabalhos relacionados ao nosso.

6.2 Perspectivas de Trabalhos Futuros

No decorrer da realização da pesquisa que levou às contribuições desta dissertação, algumas ideias de novas contribuições a serem avaliadas surgiram, as quais enumeramos a seguir:

- Utilizar os componentes propostos neste trabalho para implementar aplicações reais que demandam por funcionalidades no domínio de *solução de sistemas lineares*, de forma a melhor validar o método, aprimorá-lo, e permitir que seja aplicado em ambiente de produção. Sugerimos então implementar aplicações já em uso ou requisitada por integrantes do grupo de pesquisa ParGO, o qual integramos, ou em colaboração com projetos de grupos de pesquisa da engenharia. Disponibilizaríamos assim os benefícios do programação orientada a componentes para os pesquisadores e desenvolvedores dos grupos que nos rodeiam, permitindo maior *feedback*.

- Utilizar a metodologia proposta neste trabalho de incorporação de bibliotecas de outros domínios ao HPE. Por exemplo, o domínio de solução de equações de autovalores, uma vez que a implementação desse domínio pode ser construída sobre o domínio de sistemas lineares, já implementado nesse trabalho. Portanto, além de aumentar a gama de domínios específicos no HPE, seria também validado o benefício de **integração** e **interoperabilidade** dos componentes, pois as bibliotecas integradas serão de domínios diferentes ou até escritas em linguagens diferentes e deverão trabalhar em conjunto.

- Utilizar os benefícios da compatibilidade do modelo de componentes # com outros modelos de componentes para integrar os componentes propostos neste trabalho de mestrado aos componentes propostos em trabalhos relacionados que foram desenvolvidos sobre o modelo de componentes CCA, como o TSC e o CCA LISI. Para tanto, tem sido desenvolvida a compatibilização do HPE com o padrão CCA.

- Investigar o uso de dependências funcionais entre parâmetros de contexto de componentes abstratos, forma a qual identificamos como útil para oferecer um maior nível de abstração ao programador de componentes de aplicação para solução de sistemas lineares. A essas dependências funcionais, daremos o nome de *dependências contextuais*. Por exemplo, ao propôr os componentes faceta específicos de uma biblioteca em particular, por exemplo PETSC, uma dependência contextual poderia estabelecer a seguinte relação:

$$\text{LSSDomain} : [\mathit{library} : \text{PETSc}] \rightarrow \left[\begin{array}{l} \mathit{setup} : \text{PETScSetup}, \\ \mathit{vector} : \text{PETScVector}, \\ \mathit{matrix} : \text{PETScMatrix}, \\ \mathit{solver} : \text{PETScSolver}, \\ \mathit{pre\_conditioner} : \text{PETScPreConditioner} \end{array}\right]$$

.

Dessa forma, o programador de componentes de aplicação (nível I) precisaria apenas suprir o parâmetro de contexto library, sendo os parâmetros reais dos demais parâmetros de contexto inferidos a partir da dependência contextual informada pelo programador dos componentes no nível II (facetas). Poderíamos ainda generalizar a ideia, permitindo a definição de dependências contextuais ao nível dos parâmetros de contexto do componente abstrato, ao

invés de combinações específicas de parâmetros reais. Por exemplo:

$$\text{LSSDomain} : [\textit{library}] \rightarrow [\textit{setup}, \textit{vector}, \textit{matrix}, \textit{solver}, \textit{pre\_conditioner}]$$

. Assim, para uma certa escolha para *library* somente poderia existir uma única escolha para os demais parâmetros de contexto no ambiente. Para o caso do componente domínio LSSDomain, esta seria uma suposição realista, uma vez que os componentes facetas são apenas descrições das subrotinas das bibliotecas nativas, o que não deve variar.

Referências Bibliográficas

[1] Wiki of MCMD-Workgroup at CCA Forum. Disponível em <https://www.cca-forum.org/wiki/tiki-index.php?page=MCMD-WG>, Acesso em: Janeiro/2010.

[2] The Common Component Architecture Forum. Disponível em <https://www.cca-forum.org>, Acesso em: Julho/2009.

[3] AMD Core Math Library (ACML). Disponível em <http://www.amd.com/acml>. Acesso em: Janeiro/2010.

[4] Allan, B. A., Armstrong, R. C., Wolfe, A. P., Ray, J., Bernholdt, D. E., and Kohl, J. A. The cca core specification in a distributed memory spmd framework. *Concurrency and Computation: Practice and Experience 14*, 5 (2002), 323–345.

[5] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[6] Amedro, B., Baude, F., Caromel D., Delbe, C., Filali, I., Huet, F., Mathias, E., and Smirnov O. *An Efficient Framework for Running Applications on Clusters, Grids and Clouds*. Springer, 2010, chapter 10, pp. 163–178.

[7] Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammerling, S., Demmel, J., Bischof, C., and Sorensen, D. Lapack: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE*

conference on Supercomputing (Los Alamitos, CA, USA, 1990), IEEE Computer Society Press, pp. 2–11.

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

[9] Armstrong, R., Kumfert, G., McInnes, L. C., Parker, S., Allan, B., Sottile, M., Epperly, T., and Dahlgren, T. The CCA component model for high-performance scientific computing. *Concurr. Comput. : Pract. Exper. 18*, 2 (2006), 215–229.

[10] Asynchronous Array of Simple Processors Project. Disponível em <http://www.ece.ucdavis.edu/vcl/asap/>. Acesso em: Maio/2009.

[11] Aztec Project Description. Disponível em <http://www.cs.sandia.gov/crf/aztec_descrip.html>. Acesso em: Janeiro/2010.

[12] Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008.

[13] Barghoorn, M. High performance computing through parallel processing. In *APL '00: Proceedings of the international conference on APL-Berlin-2000 conference* (New York, NY, USA, 2000), ACM, pp. 32–34.

[14] Bassetti, F., Brown, D., Davis, K., Henshaw, W., and Quinlan, D. Overture: an object-oriented framework for high performance scientific computing. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 1–9.

[15] Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, W., Henrio, L., and Prez, C. GCM: A Grid Extension to Fractal for

Autonomous Distributed Components. *Annals of Telecommunications 64*, 1 (2009), 5–24.

[16] Ben-Ari, M. *Principles of Concurrent and Distributed Programming (2nd Edition) (Prentice-Hall International Series in Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[17] Bertrand, F., and Bramley, R. DCA: a distributed CCA framework based on MPI. In *Proceedings of the HIPS2004 - 9th International Workshop on Highlevel Parallel Programming Models and Supportive Environments* (2004).

[18] Blackford, L. S., Choi, J., Cleary, A., Petitet, A., Whaley, R. C., Demmel, J., Dhillon, I., Stanley, K., Dongarra, J., Hammarling, S., Henry, G., and Walker, D. Scalapack: A Portable Linear Algebra Library for Distributed Memory Computers - design issues and performance. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)* (Washington, DC, USA, 1996), IEEE Computer Society, p. 5.

[19] Blair, G. S., Coupaye, T., and Stefani, J.-B. Component-based architecture: the fractal initiative. *Annales des Télécommunications 64*, 1-2 (2009), 1–4.

[20] Box, D. *Essential COM*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. Foreword By-Booch, Grady and Foreword By-Kindel, Charlie.

[21] Brittain, J., and Darwin, I. F. *Tomcat: the definitive guide, 2nd edition*. O'Reilly, 2007.

[22] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper. 36*, 11-12 (2006), 1257–1284.

[23] Bruneton, E., Coupaye, T., Leclercq, M., Quma, V., and Stefani, J.-B. The Fractal Component Model and Its Support In Java. *Software - Practice and Experience 36* (2006), 1257–1284.

[24] Burkardt, J. NINTLIB - Multi-dimensional quadrature. Disponível em <http://people.sc.fsu.edu/ burkardt/f_src/nintlib/nintlib.html>. Acesso em: Setembro/2009.

[25] Caromel, D., Klauser, W., and Vayssiere, J. Towards Seamless Computing and Metacomputing in Java. In *Concurrency Practice and Experience* (Sept. 1998), Geoffrey C. Fox, Ed., vol. 10, Wiley & Sons, Ltd., pp. 1043–1061. http://www-sop.inria.fr/oasis/proactive/.

[26] Carvalho Junior, F. H., and Correa, R. C. The Design of a CCA Framework with Distribution, Parallelism, and Recursive Composition. In *The 2010 Workshop on Component-Based High Performance Computing (CBHPC'2010)* (Oct. 2010), G. Allen and T. Kielmann, Ed., ACM.

[27] Carvalho Junior, F. H., Correa, R. C., Lins, R. D., Silva, J. C., and Araújo, G. A. On the Design of Abstract Binding Connectors for High Performance Computing Component Models. In *Joint Conference on HPC Grid programming Environments and Components (HPC-GECO), and on Components and Frameworks for High Performance Computing (4^{th} CompFrame)* (2007).

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

[29] Carvalho Junior, F. H., and Lins, R. D. An Institutional Theory for #-Components. *Electronic Notes in Theoretical Computer Science 195* (Jan. 2008), 113–132.

[30] Carvalho Junior, F. H., Lins, R. D., Corrêa, R. C., and Araújo, G. A. Towards an architecture for component-oriented parallel programming. *Concurr. Comput. : Pract. Exper. 19*, 5 (2007), 697–719.

[31] Cirne, W., Brasileiro, F., Sauvé, J., Andrade, N., Paranhos, D., Santos-neto, E., Medeiros, R., and Gr, F. C. Grid computing for bag of tasks applications. In *In Proc. of the 3rd IFIP Conference on E-Commerce, E-Business and EGovernment* (2003).

[32] de Carvalho Junior, F. H., and Lins, R. D. A Type System for Parallel Components. *CoRR abs/0905.3432* (2009). informal publication.

[33] Demmel, J. W., Gilbert, J., and Li, X. S. Superlu user's guide. Tech. rep., Berkeley, CA, USA, 1997.

[34] Denis, R., and Priol, T. Portable parallel corba objects: an approach to combine parallel and distributed programming for grid computing. In *In Proc. of the 7th Intl. Euro-Par'01 Conference (EuroPar'01* (2001), Springer, pp. 835–844.

[35] Dijkstra, E. W. The humble programmer. *Commun. ACM 15*, 10 (October 1972), 859–866.

[36] Diller, K. R., Oden, J. T., Bajaj, C., Browne, J. C., Hazle, J., Babuka, I., Bass, J., Bidaut, L., Demkowicz, L., Elliott, A., Feng, Y., Fuentes, D., Goswami, S., Hawkins, A., Khoshnevis, S., Kwon, B., Prudhomme, S., and Stafford, R. J. Computational infrastructure for the real-time patient-specific treatment of cancer. Tech. rep., University of Texas at Austin, 2008.

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

[38] Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., and White, A., Eds. *Sourcebook of Parallel Computing.* Morgan Kaufmann, 2003.

[39] Dongarra, J., Otto, S. W., Snir, M., and Walker, D. An Introduction to the MPI Standard. Tech. Rep. CS-95-274, University of Tennessee, Jan. 1995. http://www.netlib.org/tennessee/ut-cs-95-274.ps.

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

[41] Dongarra, J., and Whaley, R. C. *A User's Guide to the BLACS v1.1*, 1994.

[42] Dongarra, J. J., and Walker, D. W. Software libraries for linear algebra computations on high performance computers. *SIAM Review 37*, 2 (1995), 151–180.

[43] .NET Framework Developer Center. Disponível em <http://msdn.microsoft.com/pt-br/netframework>. Acesso em: Julho/2010.

[44] Drummond, L. A., and Marques, O. A. An overview of the advanced computational software (acts) collection. *ACM Trans. Math. Softw. 31*, 3 (2005), 282–301.

[45] Falgout, R. D., and Yang, U. M. hypre: A library of high performance preconditioners. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part III* (London, UK, 2002), Springer-Verlag, pp. 632–641.

[46] Forum, H. P. F. *High Performance Fortran, Language Specification, Version 2.0*, Jan. 1997.

[47] Gallopoulos, E., Houstis, E., and Rice, J. Computer as thinker/doer: problem-solving environments for computational science. *Computational Science Engineering, IEEE 1*, 2 (1994), 11 –23.

[48] Basic Features of the Grid Component Model. Disponível em <http://www.coregrid.net/mambo/images/stories/deliverables/d.pm.04.pdf>. Acessado em Julho/2010.

[49] Golub, G. H., and Van Loan, C. F. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[50] Govindaraju, M., Krishnan, S., Chiu, K., Slominski, A., Gannon, D., and Bramley, R. Merging the cca component model with the ogsi framework. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2003), IEEE Computer Society, p. 182.

[51] Hernandez, V., Roman, J. E., and Vidal, V. Slepc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Softw. 31*, 3 (2005), 351–362.

[52] Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw. 31*, 3 (2005), 363–396.

[53] Hopkins, J. Component primer. *Commun. ACM 43*, 10 (2000), 27–30.

[54] Huang, W., Wu, Y., Liu, J., Yang, G., and Zheng, W. A component based interoperability solution over existing grid middleware. In *GCC '07: Proceedings of the Sixth International Conference on Grid and Cooperative Computing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 550–557.

[55] Intel Math Kernel Library (Intel MKL). Disponível em <http://software.intel.com/en-us/intel-mkl/>. Acesso em: Janeiro/2010.

[56] Enterprise Javabeans Specifications. Disponível em <http://java.sun.com/products/ejb/docs.html>. Acesso em: Julho/2010.

[57] Jones, J., Sosonkina, M., and Saad, Y. Component-based iterative methods for sparse linear systems: Research articles. *Concurr. Comput. : Pract. Exper. 19*, 5 (2007), 625–635.

[58] Kaan, C. *Mono.NET goes LINUX*. Franzis-Verlag GmbH, Munich, Germany, Germany, 2007.

[59] Keahey, K., and Gannon, D. Pardis: A parallel approach to corba. In *In 6th IEEE International Symposium on High Performance Distributed Computation* (1997), pp. 31–39.

[60] Klema, V. Linpack user's guide. *Automatic Control, IEEE Transactions on 25*, 3 (Jun 1980), 614–614.

[61] Kohl, J. A., Wilde, T., and Bernholdt, D. E. Cumulvs: Interacting with high-performance scientific simulations, for visualization, steering and fault tolerance. *Int. J. High Perform. Comput. Appl. 20*, 2 (2006), 255–285.

[62] Krupka, J. Parallel Solvers of Poisson's Equations. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Science and Engineering, 2010.

[63] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw. 5*, 3 (1979), 308–323.

[64] Liu, F., and Bramley, R. Cca-lisi: On designing a cca parallel sparse linear solver interface. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA* (2007), IEEE, pp. 1–10.

[65] Meza, J. C., Oliva, R. A., Hough, P. D., and Williams, P. J. Opt++: An object-oriented toolkit for nonlinear optimization. *ACM Trans. Math. Softw. 33*, 2 (2007), 12.

[66] Milli, H. M. H. Understanding separation of concerns. *In Workshop on Early Aspects - Aspect Oriented Software Development (AOSD'04)* (2004).

[67] Moore, B., Dean, D., Gerber, A., Wagenknecht, G., and Vanderheyden, P. *Eclipse Development Using the Graphical Editing Framework and Eclipse Modelling Framework.* IBM International Technical Support Organization, Feb. 2004. http://www.ibm.com/redbooks.

[68] Munenori, M., Hisashi, K., Toshiyuki, K., and Hiroshi, N. The gauss-seidel method with preconditioner (i+r). *Transactions of the Japan Society for Industrial and Applied Mathematics 13*, 4 (2003), 439–445.

[69] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., and Aprà, E. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl. 20*, 2 (2006), 203–231.

[70] OMG Standard. Data Parallel CORBA, v1.0. Tech. Rep. 2006-01-03, Object Management Group, Object Management Group, Jan. 2006.

[71] The OSGi Architecture. Disponível em <http://www.osgi.org/about/whatisosgi>. Acesso em: Julho/2010.

[72] Parker, S. G., and Johnson, C. R. Scirun: a scientific programming environment for computational steering. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 1995), ACM, p. 52.

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

[74] Pérez, C., Priol, T., and Ribes, A. A parallel corba component model for numerical code coupling. In *Proc. of the 3nd International Workshop on Grid Computing, LNCS* (2002), Springer-Verlag, pp. 417–429.

[75] René, C., and Priol, T. Mpi code encapsulating using parallel corba object. *Cluster Computing 3*, 4 (2000), 255–263.

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

[77] Shende, S. S., and Malony, A. D. The tau parallel performance system. *Int. J. High Perform. Comput. Appl. 20*, 2 (2006), 287–311.

[78] Siegel, J. Omg overview: Corba and the oma in enterprise computing. *Commun. ACM 41*, 10 (1998), 37–43.

[79] IBM's Blue Gene Supercomputer Models a Cat's Entire Brain. Disponível em <http://www.popsci.com/technology/article/2009-11/digital-cat-brain-runs-blue-gene-supercomputer>. Acesso em: Março/2010.

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

[81] Smith, B. T., Boyle, J. M., and Dongarra, J. J. *Matrix Eigensystem routines - Eispack guide.* 1976.

[82] Sosonkina, M., Liu, F., and Bramley, R. Usability levels for sparse linear algebra components. *Concurr. Comput. : Pract. Exper. 20*, 12 (2008), 1439–1454.

[83] Sterritt, R., Parashar, M., Tianfield, H., and Unland, R. A concise introduction to autonomic computing. *Adv. Eng. Inform. 19*, 3 (2005), 181–187.

[84] Szyperski, C. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[85] Sécher, B., Belliard, M., and Calvin, C. Numerical platon: A unified linear equation solver interface by cea for solving open foe scientific applications. *Nuclear Engineering and Design 239*, 1 (2009), 87 – 95.

[86] TAO User's Manual. Disponível em <http://www.mcs.anl.gov/research/projects/tao/docs/manual/manual.html>. Acesso em: Janeiro/2010.

[87] The CCA Core Specification. Disponível em <http://www.cca-forum.org/download/spec/cca-0.8.1.sidl>. Acesso em: Maio/2009.

[88] The Fractal Project. Disponível em <http://fractal.ow2.org/>. Acesso em Maio/2009.

[89] Titanic Twisters. Disponível em <http://www.tacc.utexas.edu/feature_stories/2009/titanic_twisters.php>. Acesso em Janeiro/2010.

[90] van der Steen, A. J. Issues in Computational Frameworks. *Concurrency and Computation: Practice and Experience 18*, 2 (2006), 141–150.

[91] Virtual Breath. Disponível em <http://www.tacc.utexas.edu/feature_stories/2009/virtual_breath.php>. Acesso em Janeiro/2010.

[92] Wang, A. J. A., and Qian, K. *Component-Oriented Programming.* Wiley-Interscience, 2005.

[93] Whaley, R. C., and Dongarra, J. J. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 1–27.