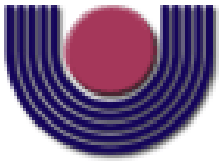

UNIOESTE – Universidade Estadual do Oeste do Paraná

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Informática

Curso de Bacharelado em Informática

Processo de Engenharia de Software para o Desenvolvimento de Aplicações Paralelas

Cleber Augusto Pivetta

CASCAVEL

2009

CLEBER AUGUSTO PIVETTA

PROCESSO DE ENGENHARIA DE SOFTWARE PARA O DESENVOLVIMENTO DE APLICAÇÕES PARALELAS

.....

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Informática, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. MSc. Guilherme Galante

CASCAVEL

2009

CLEBER AUGUSTO PIVETTA

PROCESSO DE ENGENHARIA DE SOFTWARE PARA O DESENVOLVIMENTO DE APLICAÇÕES PARALELAS

.....

Monografia apresentada como requisito parcial para obtenção do Título de *Bacharel em Informática*, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. MSc. Guilherme Galante (Orientador)
Colegiado de Informática, UNIOESTE

Prof. MSc. Luiz Antônio Rodrigues
Colegiado de Informática, UNIOESTE

Prof. Dr. Victor Francisco Araya Santander
Colegiado de Informática, UNIOESTE

Cascavel, 22 de novembro de 2009.

“...Liberdade, essa palavra que o sonho humano alimenta que não há ninguém que explique, e ninguém que não entenda...”

Cecília Meireles

AGRADECIMENTOS

Agradeço primeiramente a minha família, em especial aos meus pais, Elias e Dora, que durante toda vida me guiaram em um caminho de descobertas e aventuras, que apoiaram sem êxito em todo período de faculdade, nos bons e maus momentos.

Agradeço também a minha namorada Lays, pelos momentos de compreensão e carinho durante todos esses anos. Aos meus amigos da cidade de Palotina, onde nasci e conheci pessoas maravilhosas que estarão sempre em minha memória.

Gostaria de agradecer também, a todos os novos amigos que conquistei em Cascavel, estes que foram essenciais em todos esses anos de faculdade, pelas inúmeras horas de estudos, alegrias e descontração que partilhamos.

Aos professores do colegiado de informática, pelo apoio acadêmico, em especial ao meu orientador Guilherme Galante, por seu empenho e dedicação na orientação deste trabalho.

Lista de Figuras

Lista de Quadros

Lista de Tabelas

Lista de Abreviaturas e Siglas

MOODAP	Metodologia de desenvolvimento de aplicaçoes paralelas
DBC	Desenvolvimento Baseado em Componentes
RUP	Processo Unificado da Rational
UML	*Unified Modelling Language*
XP	*Extreme Programming*
UCP	Unidade Central de Processamento
PCAM	Particionamento, Comunicação, Aglomeração e Mapeamento
SEMPA	*Software Engineering Methods for Parallel Applications in Scientific Computing*
ASC	*Advancced Scientific Computing*
GSRS	*Global Software Requirement Specification*
LIT	*Literature Survey*
TCD	*Test Case Definition*
GSP	*Global Strategic Plan*
SEM	*Software Engineering Modules*
SRS	*Software Requirement Specification*
MDP	*Module Design Phase*
SCI	*Source Code Implementation*
TEST	*Testing*
DOC	*Documentation*
PRM	*Post Release Maintenance*
UMP2D	*Unified Methodology for Parallel Programs Development*
MIND	*Multiple Instruction stream over a Multiple Data stream*

API	*Application Programming Interface*
MPI	*Message Passing Interface*
PVM P	*Parallel Virtual Machine*
Pthreads	*Posix Threads*
IEEE	*Institute of Electrical and Electronics Engineers*
ISO	*International Organization for Standardization*
RMI	*Remote Method Invocation*
MDD	Métodos de Decomposição de Domínios

Sumário

Resumo

A programação paralela está entre as áreas mais estudadas na computação científica, isto se deve principalmente pela limitação imposta em arquiteturas com único processador. A proposta de utilização de vários processadores executando tarefas simultaneamente proporciona grandes vantagens, sobretudo um grande aumento do poder computacional. Porém, na maioria das vezes, uma aplicação paralela é desenvolvida baseado na experiência do desenvolvedor, e não por um processo definido, onde há fases e etapas de desenvolvimento. Isso remete a uma grande incerteza das práticas e custos ao longo do projeto. Desta forma, uma análise sobre algumas das principais metodologias tradicionais e paralelas é realizada visando absorver características importantes e necessárias neste tipo de aplicação. Como resultado disto, é proposta a metodologia MOODAP (Metodologia Orientada a Objetos para o Desenvolvimento de Aplicações Paralelas) onde descreve-se as principais técnicas e mecanismos utilizados neste contexto. Por fim, é realizado um estudo de caso visando validar as fases desta metodologia.

Palavras-chave: Aplicações Paralelas, Desenvolvimento, *Software*, UML, Metodologia, MOODAP.

Capítulo 1

Considerações Iniciais

1.1 Introdução

Desde o surgimento dos primeiros computadores digitais eletrônicos, a computação passou por um processo evolutivo intenso, a nível de *hardware* e *software*, a fim de proporcionar maior desempenho e ampliar o leque de aplicações que podem ser computacionalmente resolvidas de maneira eficiente [18].

Com o advento de novos tipos de aplicações, limitação de processamento em arquiteturas sequências além de um grande aumento de capacidade e velocidade das redes conectarem computadores [27], houve um conveniente aumento da possibilidade de desenvolver aplicações paralelas de alto desempenho, contrastando com a situação anterior da década de 90 onde aplicações paralelas eram confinadas apenas à pesquisa e instituições acadêmicas [25]. Agora o processamento paralelo vem ganhando maior atenção na indústria possibilitando a produção de *software* portável para plataformas de *hardware* paralelas e distribuídas com baixo custo de implementação de rede à alto desempenho dos processadores paralelos [22].

O projeto e o desenvolvimento de aplicações paralelas é uma das principais áreas de interesse no domínio da computação de alto desempenho e computação industrial. De fato a computação paralela está se tornando uma parte integral na maioria das aplicações, como por exemplo: pesquisa espacial, medicina, pesquisa genética, gráficos e animação, processamento de imagens entre outras [25].

Entretanto, o desenvolvimento de uma aplicação paralela não é um processo trivial [25]. Em um processo de desenvolvimento de uma aplicação específica, o desenvolvedor deve se perguntar questões importantes. Tais questões envolvem se a aplicação pode ser paralelizada,

como essa paralelização deve ser realizada, qual o tipo de arquitetura, que algoritmos devem ser utilizados e quais tecnologias de *software* são recomendadas [20].

Experiências têm mostrado que o desenvolvimento de *software* para sistemas paralelos ainda é menos produtivo que o desenvolvimento de programas sequenciais. Uma razão para isto é que não há ferramentas adequadas de desenvolvimento e análise para *softwares* paralelos [24].

Este problema vem sendo estudado com a intenção de encontrar uma forma de melhorar a metodologia de desenvolvimento de sistemas paralelos, baseando-se em técnicas da engenharia de *software*. O foco dessas técnicas é analisar a complexidade do sistema, encontrar a paralelização específica para uma determinada classe de aplicação, padronização na documentação e desenvolvimento de programas, portabilidade, modularidade e reusabilidade [24]. A razão de utilizar a engenharia de *software* em sistemas paralelos é tentar reduzir o custo e os esforços dos desenvolvedores, encontrando um ciclo de vida independente de projeto e modelos de programação [23].

1.2 Motivação

Em nível de *hardware*, muito já se desenvolveu em computação paralela, e muita experiência já foi adquirida. Porém, em nível de *software* paralelo, ainda existem muitas lacunas a serem preenchidas e muito ainda deve ser pesquisado até que se encontrem soluções apropriadas, que impliquem na utilização dessas arquiteturas paralelas com ganhos de desempenho significativos [18].

Desta forma, há uma grande necessidade de um processo de desenvolvimento completo para sistemas paralelos que leve em conta as exigências específicas dos grandes projetos científicos, onde o fluxo de dados é irregular e existe uma complexidade maior nas estruturas de dados [20].

Como as técnicas da engenharia de *software* atuais não são abrangentes o suficiente, existe a necessidade de estudar e/ou propor novas alternativas para resolver este problema. Uma melhor avaliação de métodos e tecnologias inseridos no processo de desenvolvimento de aplicações paralelas aparece como uma opção eficaz para determinar as necessidades e vantagens de sua utilização.

1.3 Objetivos

Neste trabalho propõe-se a realização de uma análise de processos de engenharia de *software* para o desenvolvimento de aplicações paralelas. Mais especificamente pretende-se realizar uma revisão dos métodos e técnicas que já foram utilizados neste contexto ou de alguma forma podem ser utilizados para este fim. De acordo com os resultados desta revisão pretende-se propor uma metodologia de desenvolvimento de aplicações paralelas considerando um processo completo de desenvolvimento onde vislumbra-se preencher as lacunas encontradas nas metodologias estudadas. Por fim é realizado um estudo de caso utilizando-se desta metodologia.

1.4 Organização do texto

Este trabalho está organizado da seguinte forma.

O capítulo 2 apresenta uma visão geral do processo de desenvolvimento de software, seus principais conceitos e características. No mesmo capítulo também são descritas as principais e mais utilizadas metodologias de desenvolvimento, expondo de maneira geral suas particularidades.

O capítulo 3 apresenta uma visão geral das características dos programas paralelos, bem como os modelos mais utilizados na programação paralela. Também são descritas três metodologias de desenvolvimento deste tipo de aplicação. Por fim são abordadas em uma breve discussão algumas questões relevantes relacionadas ao processo de desenvolvimento das metodologias citadas.

O capítulo 4 descreve a metodologia proposta. São apresentadas as cinco fases do ciclo de desenvolvimento da MOODAP (Metodologia orientada a objetos para o desenvolvimento de aplicações paralelas), bem como as informações básicas sobre as técnicas e mecanismos utilizados em um desenvolvimento de uma aplicação paralela.

O capítulo 5 apresenta um estudo de caso onde são demonstradas todas as fases da metodologia MOODAP. Neste estudo é abordada uma solução paralela para a resolução de sistemas de equações lineares.

Finalmente, no capítulo 6 encontram-se as considerações deste trabalho, bem como as perspectivas de trabalhos futuros.

Capítulo 2

Processo de Engenharia de Software

O desenvolvimento de software é uma atividade em constante expansão. A cada dia, novos domínios, cada vez mais complexos, são incorporados à área de atuação da Engenharia de *Software*. À medida que os sistemas de software requeridos crescem em complexidade, o processo de desenvolvimento também se torna mais complexo. Além disso, as corporações estão competindo em um mercado onde o ciclo de vida de produtos e serviços estão cada vez mais curtos. Como consequência disso, há uma grande necessidade de adaptar um processo de software maduro apropriado que atenda as demandas do mercado.

2.1 Definições

Para melhor compreender o assunto é necessário definir o que é um processo de *software*.

O processo pode ser tratado inicialmente como uma abstração do conhecimento, que deve se transformar em um produto de *software*. É uma incorporação de conhecimentos coletados, destilados e organizados ao longo do processo.

O processo de *software* é definido por Pressman [1] como “um arcabouço para as tarefas que são necessárias para construir softwares de qualidade”.

Sommerville em [2] define da seguinte maneira: “O processo é um conjunto de atividades e resultados associados que produzem um produto de *software*.”

Com estas definições pode-se constatar uma grande necessidade dos autores expressarem uma visão modular do processo, subdividindo-o em um conjunto de atividades. Como existem atividades que englobam outras atividades, neste contexto o termo fase será utilizado para descrever atividades de nível mais alto.

O processo de software consiste de fases e informações associadas que são solicitadas para o desenvolvimento de um software. Toda organização tem seu próprio processo de *software*,

mas geralmente essas abordagens individuais seguem um mesmo modelo de caráter mais genérico para o processo de software.

2.2 Fases do Processo de Software

Apesar da grande quantidade de processos de software, quatro fases são comuns a todos os processos [2]:

1. Especificação do software: funcionalidades e restrições de operação;
2. Desenvolvimento: projeto e implementação do software de acordo com a especificação;
3. Validação do Software: garante que o software produzido cumpra a especificação;
4. Evolução do software: adequação as novas necessidades requeridas pelos clientes.

A definição das fases do processo realizada por Sommerville [2] pode ser considerada mais completa levando em consideração outras literaturas como a de Schwartz [3]. Este último autor aponta como fases do processo a especificação de requisitos, projeto de sistema, programação além de verificação e integração. Apesar da similaridade na maioria dos aspectos, Sommerville incluiu o critério de evolução, que expressa a necessidade de um desenvolvimento contínuo do *software* mesmo depois de entregue ao cliente.

2.3 Atividades do Processo de Software

Para cada fase do processo de *software* existe uma série de atividades básicas que são executadas. Segundo Pressman [1], estas atividades constituem um conjunto mínimo necessário para se obter o produto de *software*. Levando em consideração as classificações de Pressman, Sommerville e Schwartz [1, 2, 3] é possível identificar as atividades relevantes para cada fase do processo de software:

1. Especificação do *software*

    (a) Engenharia de Sistema: estabelecimento de uma solução geral para o problema, envolvendo questões extra-*software*.

(b) Análise de Requisitos: levantamento das necessidades do sistema a ser implementado com os possíveis usuários. Esta atividade tem como objetivo produzir uma documentação especificando os requisitos.
(c) Especificação do sistema: descrição funcional do sistema. Pode incluir um plano de testes para verificar adequação.

2. Desenvolvimento

(a) Projeto arquitetural: é desenvolvido um modelo conceitual para o sistema. Dependendo do contexto, especificar os módulos, suas relações e documentação.
(b) Projeto de interface: para cada módulo, a interface de comunicação com outros módulos deve ser definida e documentada.
(c) Projeto detalhado: realiza a definição dos módulos que possivelmente são traduzidos para pseudo-código.
(d) Implementação: codificação do sistema em uma linguagem de computador.

3. Validação do *software*

(a) Teste de unidade e módulo: realização de testes individuais em cada módulo para verificar a presença de erros e comportamento.
(b) Integração: nesta atividade os módulos são combinados e testados em grupo.

4. Evolução do *software*

Fase na qual todo processo desenvolvido entra em um ciclo iterativo, proporcionando uma constante evolução do produto final.

Além das atividades relevantes em cada fase, o processo pode ser completado com atividades *guarda-chuva* que ocorrem durante todo o projeto. Estas têm por objetivo a gestão, o monitoramento e o controle do projeto. Atividades típicas desta categoria incluem [17]:

- Acompanhamento e controle de projeto de software: permite aos *stakeholders* avaliar o progresso com base no planejamento e tomar ações necessárias para manter o cronograma.

- Garantia de qualidade: as atividades necessárias para garantir a qualidade de *software* são definidas e conduzidas.
- Gestão de risco: avalia os riscos relacionados ao projeto e a qualidade do produto
- Revisões técnicas formais: os artefatos de engenharia de software são avaliados, visando descobrir e remover erros antes que sejam propagados para as fases seguintes.
- Medição: as medidas de projeto, processo e produto são definidas. Desta forma, o *software* desenvolvido deve satisfazer as necessidades do usuário.
- Gestão de configuração de *software*: as modificações ao longo do projeto devem ser gerenciadas.
- Gestão de reusabilidade: define critérios para a reutilização dos produtos de trabalho. Estabelece mecanismos para obter componentes reutilizáveis.
- Preparação e produção do produto de trabalho: define as atividades relativas a criação de produtos de trabalho como documentos, modelos, formulários e listas.

É importante ressaltar que não existe um processo de software genérico que possa ser aplicado de forma integral em quaisquer projetos. Variações na tecnologia e paradigma adotados no desenvolvimento, tamanho e complexidade do projeto, requisitos e métodos de desenvolvimento, entre outros fatores, influenciam na forma como um produto de *software* é adquirido, desenvolvido, operado e mantido [4]. Ao passar dos anos vários modelos de processo foram criados com o propósito de englobar a grande variedade de novos domínios de aplicações vigentes no mercado.

Na seção seguinte são analisados alguns dos principais modelos mencionados na literatura.

2.4 Modelos de Processo de Software

Desde 1960, surgiram várias descrições do clássico ciclo de vida de software. Basicamente, o propósito desses modelos está em prover um esquema conceitual para gerenciar o desenvolvimento e fornecer um suporte maior ao desenvolvedor da aplicação. Abaixo, consideram-se alguns requisitos importantes que devem ser proporcionados pelo modelo de processo de software [5].

- Efetividade: um processo efetivo deve produzir um software correto, no sentido de desenvolver exatamente o que foi requerido pelo cliente. O processo deve verificar se o produto final soluciona as suas necessidades.

- Manutenção: deve proporcionar rapidez e facilidade ao encontrar e solucionar falhas.

- Predicabilidade: qualquer novo produto em desenvolvimento precisa ser planejado. Isto é essencial pra predizer o tempo e recursos necessários para o projeto.

- Repetitividade: se um processo é definido para um tipo de aplicação, ele deve ser reutilizado em processos futuros. Produzir um novo processo de *software* para cada projeto traz um elevado custo desnecessário.

- Qualidade: o processo deve prover uma comunicação clara entre clientes e desenvolvedores. Isto permite assegurar que o produto se enquadre no seu propósito de criação.

- Aprimoramento: as constantes mudanças nos ambientes de desenvolvimento e a solicitação de novos tipos de produtos requerem aprimoramentos no processo. O processo definido deve identificar e melhorar seu próprio modelo.

- Acompanhamento: o processo definido deve permitir a administração, desenvolvedores, e cliente acompanhar o progresso do projeto. Este acompanhamento tem por objetivo aperfeiçoar o planejamento e a predicabilidade.

2.4.1 Modelo Cascata

O modelo cascata ou clássico foi proposto por Royce [6]. Até meados da década de 80 foi o único modelo de aceitação geral. A idéia principal deste modelo é que as etapas de desenvolvimento seguem uma sequência, de forma que uma nova etapa só poderá ter início quando a anterior estiver finalizada. O autor sugere laços de *feedback*, que permitem realimentar fases anteriores do processo, mas em geral o modelo cascata é considerado um modelo linear. A figura 2.1 ilustra o modelo.

Apesar do modelo cascata ter sido um dos paradigmas mais utilizados na engenharia de *software*, a sua aplicabilidade, em muitos campos tem sido questionada. Segundo Pressman [1], podem ser caracterizados como os principais problemas deste modelo:

1. Ausência de *feedback* entre as fases: cada fase deve ser feita apenas uma vez, não retornando a fase anterior.

2. Dificuldade na especificação de requisitos: resulta na incerteza do início de qualquer projeto, além da indeterminação do tempo necessário para cada fase de desenvolvimento do sistema.

3. Ausência de prototipagem: uma versão funcional do sistema não estará disponível até a conclusão do processo.

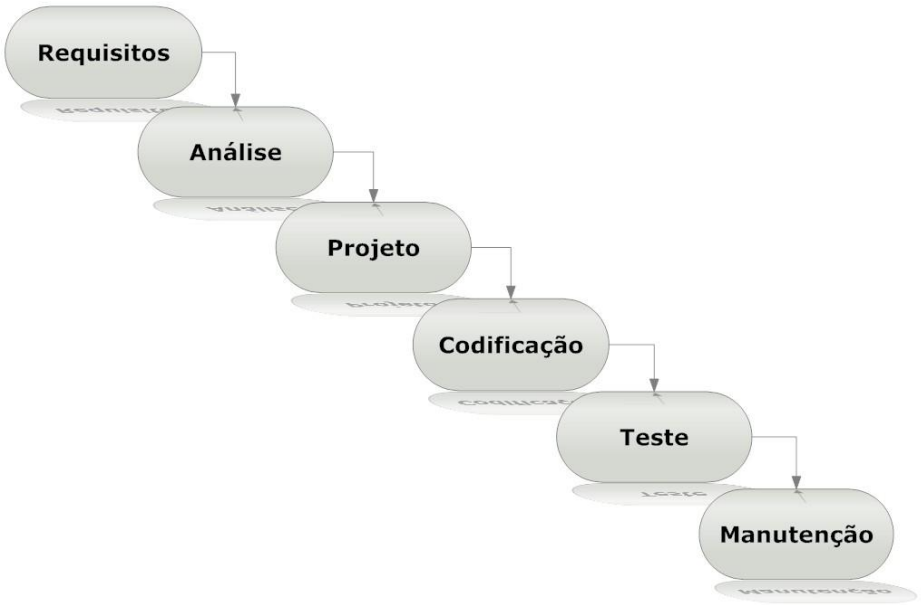

Figura 2.1 Modelo Cascata. Adaptado de Royce [6]

Pode-se deduzir que o modelo cascata se tornou inadequado para o desenvolvimento de aplicações atuais, pois a grande necessidade de modificações, tanto em características quanto em conteúdo de informação exprimiu a necessidade de um melhor intercâmbio entre as fases. No entanto, pode ser eficaz em processos nos quais os requisitos estão bem definidos e o processo ocorre de maneira sequencial.

2.4.2 Modelo Espiral

O modelo espiral foi proposto por Boehm [7] em 1988 com o objetivo de integrar as características de modelos já criados anteriormente adaptando-os a necessidades específicas de desenvolvedores ou às particularidades do software a ser desenvolvido.

A principal característica inserida no modelo proposto pro Boehm foi guiar seu processo com base em análise de riscos e planejamento que é realizado durante toda a evolução do

desenvolvimento. Riscos são circunstâncias adversas que podem surgir durante o desenvolvimento de software impedindo o processo ou reduzindo a qualidade do produto.

Este modelo organiza o desenvolvimento como um processo iterativo em que vários conjuntos das fases do processo se sucedem até se obter o sistema final. A figura 2.2 ilustra este modelo.

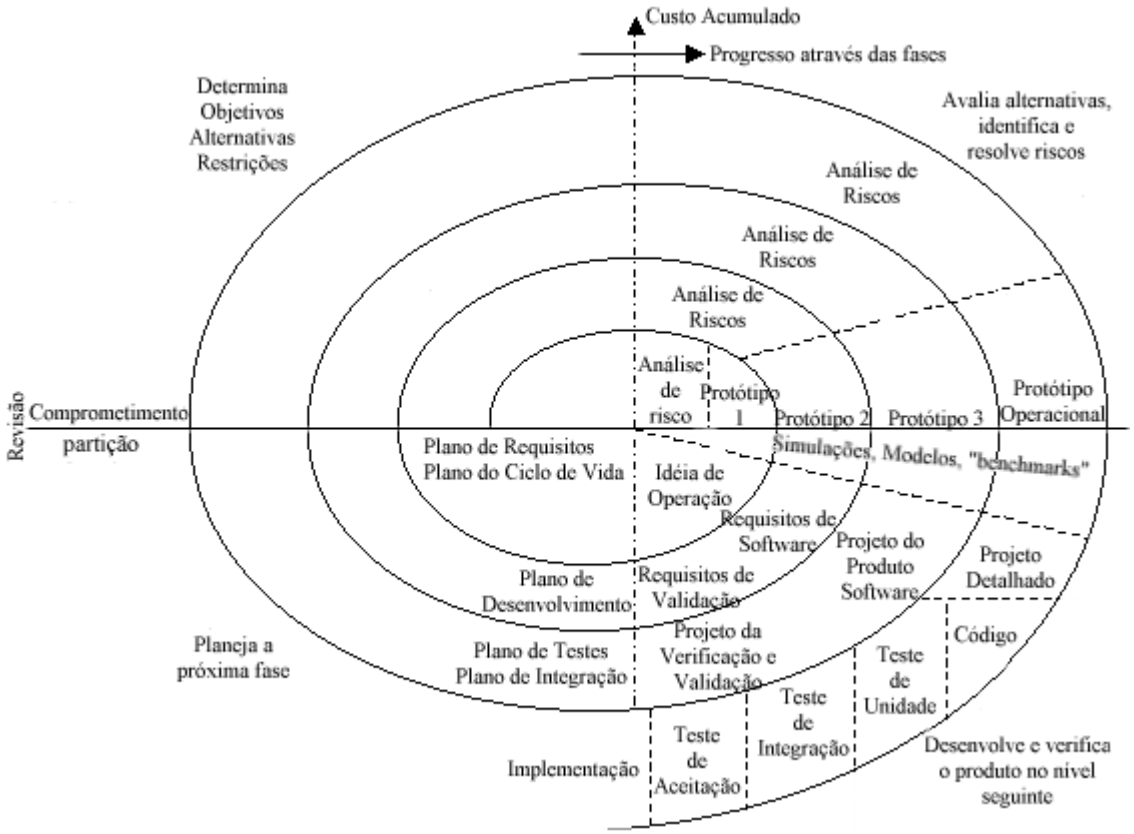

Figura 2.2 Modelo Espiral. Adaptado de Boehm [7]

É possível identificar que o modelo espiral possibilita maior integração entre as fases e facilita a depuração e a manutenção do sistema, além de permitir que o projetista e o cliente possam compreender e reagir aos riscos em cada etapa evolutiva.

Apesar de uma evolução significativa em relação ao modelo cascata, o modelo definido por Boehm não suporta o paralelismo entre as fases, ou seja, as fases continuam sendo executadas de maneira sequencial.

A incorporação de um processo iterativo reflete mais realisticamente o desenvolvimento de *softwares* de grande porte, porém um processo evolucionário requer maior competência principalmente na etapa de análise de riscos onde a não identificação e falhas no gerenciamento de erros podem acarretar em graves problemas no software desenvolvido.

2.4.3 Modelo de Desenvolvimento Baseado em Componentes

O desenvolvimento baseado em componentes (DBC) pode ser caracterizado pela composição de partes já existentes, ou pela composição de partes desenvolvidas independentemente e que são integradas para atingir um objetivo final [8]. Segundo Brown e Wallnau [10], um componente de software pode ser definido como "uma não-trivial, quase independente, e substituível parte de um sistema que cumpre uma função clara no contexto de uma arquitetura bem definida".

As atividades de modelagem e construção começam com a identificação de componentes candidatos. Esses componentes podem ser projetados como módulos de software convencional, classes ou pacote de classes orientados a objetos. Independente da tecnologia utilizada, o modelo de desenvolvimento baseado em componentes apresenta cinco fases essenciais utilizando uma abordagem iterativa para a criação de *softwares*, de acordo com a proposta de Brown [11]. A figura 2.3 ilustra este modelo.

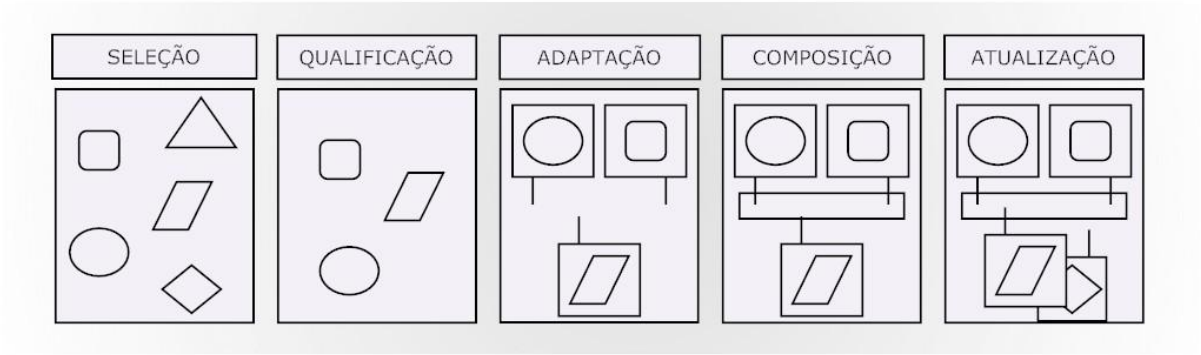

Figura 2.3 Modelo de desenvolvimento baseado em componentes. Adaptado de Brown [11]

A figura 2.3 demonstra apenas como tratar a inclusão de componentes ao sistema, porém um processo completo é composto de várias outras fases. Uma das metodologias mais utilizadas em desenvolvimento baseado em componentes é o Catalysis, na qual define-se as fases de análise de requisitos, especificação do sistema, projeto da arquitetura e projeto interno de componentes [21]. Estas fases são descritas brevemente abaixo.

1. Análise de Requisitos: está focada no entendimento do problema, bem como a delimitação do contexto do sistema, a definição dos requisitos (funcionais e não funcionais) e questões organizacionais.

2. Especificação do Sistema: descreve o comportamento externo do sistema, no qual procura-se modelar soluções de software para os modelos de domínio previamente identificados na especificação de requisitos.
3. Projeto de Arquitetura: o objetivo desta fase é a implementação interna do sistema, composta por duas partes relacionadas: a arquitetura da aplicação (implementa a lógica do negócio) e a arquitetura técnica (componentes de tecnologia e as dependências estáticas).
4. Projeto interno de Componentes: nesta fase é definida a estrutura interna, bem como suas interações que satisfaçam requisitos tecnológicos, comportamentais e de engenharia de software para cada componente.

A idéia de construir novas soluções a partir da combinação de componentes desenvolvidos ou adquiridos no mercado permite um aumento na qualidade do produto além de proporcionar suporte ao desenvolvimento rápido [12]. Estes sistemas possibilitam que suas partes sejam alteradas, removidas e substituídas sem que o sistema em si seja substituído.

2.4.4 Processo Unificado da Rational (RUP)

O RUP é um processo iterativo e incremental desenvolvido por três dos principais especialistas da indústria de software: Grady Booch, Ivar Jacobson e James Rumbaugh, sendo o resultado de mais de 30 anos de experiência acumulada [13]. Ele proporciona uma abordagem disciplinada para a atribuição de tarefas e de responsabilidades dentro de uma organização de desenvolvimento. O RUP utiliza algumas das melhores práticas do desenvolvimento do software moderno, no intuito de garantir alta qualidade para uma grande variedade de projetos e organizações.

O RUP foi criado para auxiliar o desenvolvimento de aplicações orientado a objetos, empregando uma utilização efetiva e vantajosa da Linguagem de Modelagem Unificada (UML), além disso, o processo está fundamentado em três princípios básicos: orientação a casos de uso, arquitetura e iteração.

Seu ciclo pode ser considerado tipicamente evolutivo. No entanto, uma forma de organização em fases é adotada para comportar os ciclos de desenvolvimento, permitindo uma gerência mais efetiva de projetos complexos. Ao contrário do tradicionalmente definido como fases na maioria dos modelos de ciclo de vida – planejamento, levantamento de requisitos,

análise, projeto, implementação e testes, o RUP utiliza fases ortogonais[1] a estas, descritas brevemente abaixo [14]:

1. Fase de Iniciação: o objetivo é atingir o consenso entre todos os envolvidos sobre os objetivos do ciclo de vida do projeto. Deve-se formular o escopo do projeto, além de preparar o seu ambiente.

2. Fase de Elaboração: o objetivo é assegurar que a arquitetura, os requisitos e os planos estejam bem definidos o suficiente para fornecer uma base estável para o esforço da fase de construção. Deve-se decidir basicamente sobre a organização do sistema, a seleção dos elementos e seu comportamento.

3. Fase de Construção: basicamente o objetivo desta fase é esclarecer os requisitos restantes e concluir o desenvolvimento do sistema com base na arquitetura definida.

4. Fase de Transição: consiste de várias atividades com o objetivo de assegurar que o software esteja disponível para seus usuários finais ajustando o produto com base em pequenos *feedbacks*. A figura 2.4 ilustra a visão geral do processo.

Estas fases ortogonais possibilitam um processo robusto e bem definido com a geração de artefatos[2] importantes. Assim, os riscos mais relevantes são combatidos primeiro, diminuindo as chances de fracasso do projeto. Todavia, este processo é inadequado para projetos de pequeno porte devido a sua alta complexidade, além da necessidade de uma grande experiência por parte da equipe de desenvolvimento na manipulação de ferramentas e métodos.

[1] Estas fases indicam a ênfase que é dada no projeto em determinado instante de tempo, desta forma o RUP divide o projeto em quatro fases no sentido de capturar a dimensão de tempo utilizada.

[2] É o produto de uma ou mais atividades dentro do contexto do desenvolvimento de um *software*. Em outras palavras, são as informações produzidas durante as etapas de projeto, como um modelo de caso de uso, um diagrama de classe, entre outros.

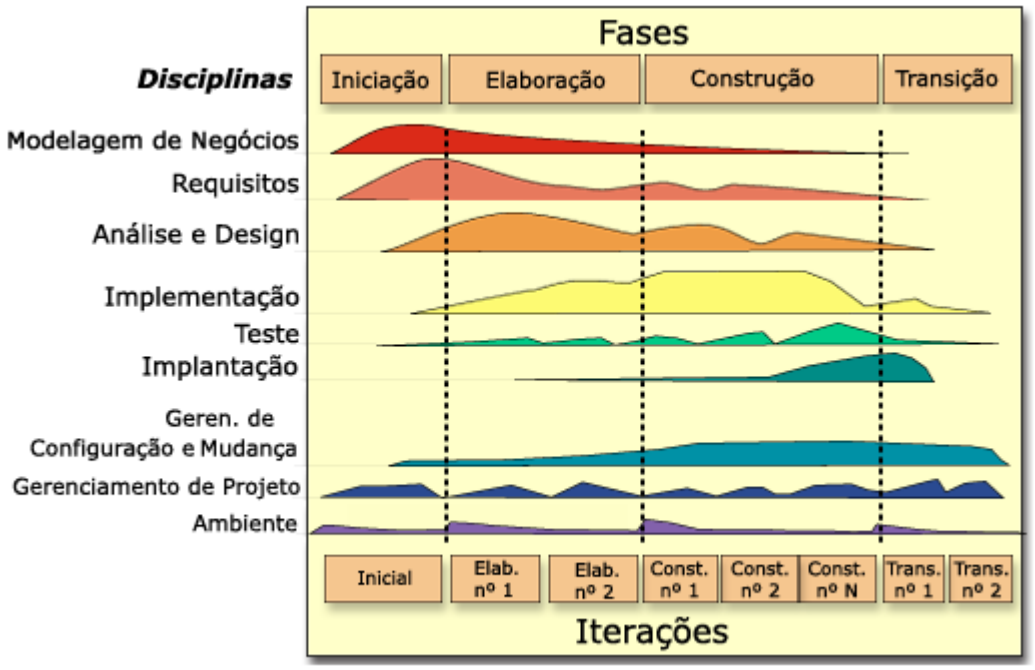

Figura 2.4: Visão geral do RUP [14]

2.4.5 Metodologias Ágeis

O termo "Metodologias Ágeis" tornou-se evidente no estabelecimento do Manifesto Ágil em 2001 [16], onde vários especialistas em processos de desenvolvimento de software representando métodos como Scrum e Extreme Programming (XP) apresentaram novos conceitos compartilhados por seus métodos. Estes conceitos são [15]:

- Foco nos indivíduos e interações ao invés de processos e ferramentas.
- Software executável ao invés de documentação.
- Colaboração do cliente ao invés de negociação de contratos.
- Respostas rápidas a mudanças ao invés de seguir planos.

Os conceitos citados acima demonstram uma perspectiva de desenvolvimento diferente das metodologias tradicionais. Não que as metodologias ágeis rejeitem os processos, ferramentas e planejamento, mas simplesmente demonstram a necessidade de focar o processo em indivíduos e interações. Uma visão muito similar à forma que pequenas e médias organizações trabalham e respondem a mudanças [16].

As Metodologias Ágeis vislumbram principalmente o termo "agilidade" no seu contexto de trabalho. Isto remete a necessidade de responder adequadamente e rapidamente às

modificações ao longo do processo. Além disso, o processo deve ser projetado de modo que permita à equipe de projeto adaptar tarefas e aperfeiçoá-las, focando nos produtos de trabalho mais essenciais e mantendo-os simples.

De acordo com Pressman [17], qualquer processo ágil de software é caracterizado de modo que atenda a três suposições-chave sobre a maioria dos projetos de software:

1. Os requisitos podem ser alterados. É difícil prever se as prioridades do cliente serão modificadas à medida que o projeto é desenvolvido.
2. É complicado determinar se a etapa de projeto está finalizada e pronta para a etapa de construção. Estas etapas devem ser realizadas em conjunto. Isto facilita a validação dos modelos de projeto à medida que são criados.
3. O planejamento das etapas de análise, projeto, construção e testes não são tão previsíveis.

Para gerenciar toda essa imprevisibilidade, um processo ágil de software inclui uma estratégia de desenvolvimento incremental, possibilitando a entrega de protótipos executáveis ao cliente em curtos períodos de tempo. Esta característica fornece o *feedback* necessário á equipe de software no sentido de adaptar as modificações adequadas ao produto em .43desenvolvimento.

Os benefícios que um desenvolvimento ágil oferece é tornar o processo mais ágil e flexível. Porém, existem pontos fracos na utilização destas metodologias. Não existe uma análise de riscos efetiva que busca evitar os erros. A análise de requisitos é conduzida de maneira informal, o que pode provocar certa insegurança quanto ao bom funcionamento do sistema, além da questão cultural que vincula o desenvolvimento a metodologias mais rígidas e mais documentadas.

2.5 Comentários Gerais

Neste capítulo abordou-se os principais modelos de processo de software citados na literatura, suas características e etapas de desenvolvimento. Também foi descrita a importância da utilização de um processo a fim de melhorar o gerenciamento e a qualidade do produto final.

Constatou-se que a definição de vários processos de desenvolvimento possibilita uma melhor adequação a um maior número de domínios de aplicação. Porém, novas tecnologias

são desenvolvidas a cada dia, e a necessidade de melhorar e adaptar os processos para estas mudanças requer uma constante evolução das técnicas de engenharia de software empregadas nas organizações.

No próximo capitulo serão abordadas as principais características e conceitos de computação paralela, bem como algumas metodologias de desenvolvimento específicas para este tipo de aplicação.

Capítulo 3

Conceitos de Computação Paralela

Até meados da década de 80, acreditava-se que o aumento de desempenho dos computadores seria vinculado somente à criação de processadores mais eficientes [19]. Porém, está idéia tomou uma nova direção com uma abordagem de processamento paralelo, no qual sua essência se baseia na união de dois ou mais processadores para a solução de um problema computacional.

Um programa paralelo possui uma série de características que o diferem de um programa sequencial. Este último possui um conjunto de instruções que são executas sequencialmente, já o paralelismo especifica programas cujas instruções podem ser executadas concorrentemente ou simultaneamente. Além de possuir estruturas comuns a programas sequenciais como laços de repetição, comandos de atribuição e comandos de decisão, um programa paralelo deve tratar aspectos específicos relacionados ao paralelismo, como por exemplo, sincronizações e comunicações entre processos [18].

3.1 Arquiteturas Paralelas

No desenvolvimento de um software paralelo o programador deve estabelecer uma interação entre os processos paralelos. Os dois principais paradigmas de programação paralela são memória distribuída e memória compartilhada [20].

3.1.1 Memória distribuída

Este modelo tem uma abordagem mais simples em nível de *hardware*. Esta abordagem é utilizada em computadores separados fisicamente conectados por uma rede, onde cada UCP (Unidade Central de Processamento) tem memória própria e a comunicação é realizada por

meio de troca de mensagens. Como exemplo de utilização de memória distribuída pode-se citar a computação em *cluster* de computadores. A figura 3.1 ilustra o modelo.

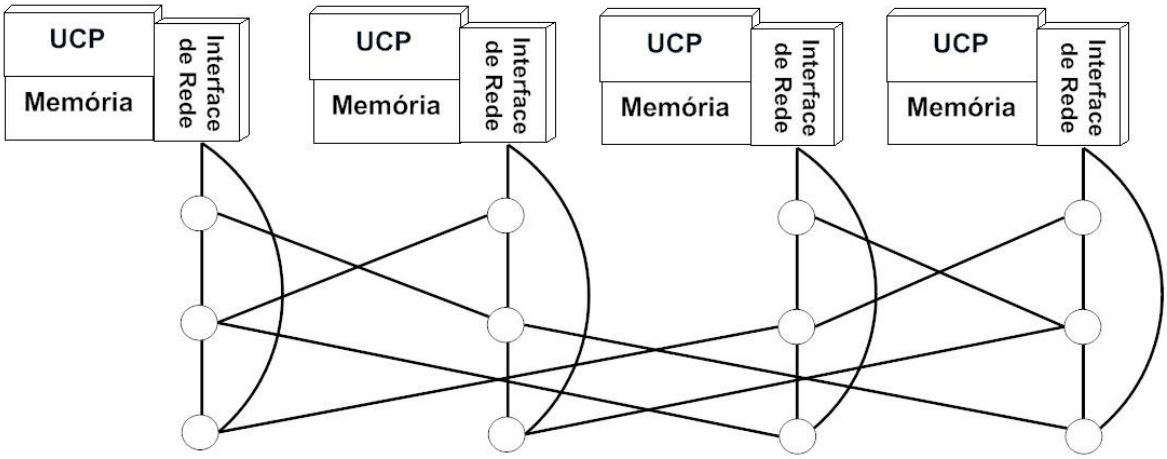

Figura 3.1: Modelo de Memória Distribuída. Adaptado de Dongarra et al. [20].

3.1.2 Memória Compartilhada

Com uma abordagem mais complexa, o modelo de memória compartilhada possibilita que todas as UCPs possam acessar o mesmo conjunto de dados para leitura e escrita. Isso faz com que o acesso as informações aconteçam mais rápido do que no modelo de memória distribuída, já que não necessita de recursos de rede para realizar as operações. Como exemplo de utilização de memória compartilhada pode-se citar uma máquina com vários processadores ou uma máquina com processador multicore[3]. A figura 3.2 ilustra o modelo.

Apesar de possibilitar esta vantagem, a consistência das informações deve ser mantida pelo desenvolvedor da aplicação. As operações devem estar sincronizadas de maneira que garantam exclusão mútua em seções críticas, ou seja, executadas de forma atômica.

[3] Este tipo de processador consiste de dois ou mais núcleos de processamento dentro de um mesmo *chip*. Desta forma o sistema operacional trata cada um desses núcleos como um processador distinto.

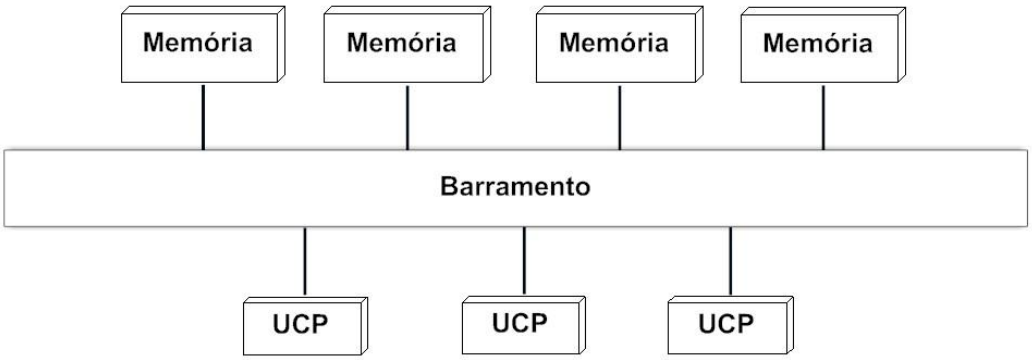

Figura 3.2: Modelo de Memória Compartilhada. Adaptado de Dongarra et al. [20].

3.2 Desenvolvimento de Software Paralelo

Considerando as metodologias de desenvolvimento de software citadas no capítulo 2, pode-se observar que estas visam um desenvolvimento mais genérico, ou seja, não leva em consideração um tipo de aplicação específica. Entretanto o desenvolvimento de uma aplicação paralela requer algumas informações associadas ou fases de desenvolvimentos próprias deste tipo de aplicação. Na literatura existem poucos trabalhos sobre metodologias específicas para este tipo de aplicação. Nesta seção descreve-se a metodologia tradicional PCAM proposta por Foster [27], a metodologia desenvolvida pelo projeto SEMPA [28] e a metodologia UMP^{2}D [29].

3.2.1 Metodologia PCAM

Esta metodologia estrutura o processo da fase de projeto em quatro etapas distintas: Particionamento, Comunicação, Aglomeração e Mapeamento. Nas duas primeiras etapas procura-se descobrir algoritmos visando concorrência e escalabilidade, diferente do foco das duas últimas, que está relacionado a questões de localidade e eficiência. A figura 3.3 ilustra a metodologia.

É interessante mencionar que a metodologia PCAM [27] é apresentada como um conjunto de atividades sequenciais. Porém, esta também considera que em temos práticos várias questões podem ser tratadas simultaneamente e que avaliações dentre o processo podem ocasionar em mudanças em etapas anteriores. A seguir são descritas as quatro etapas da metodologia, bem como seus princípios e características.

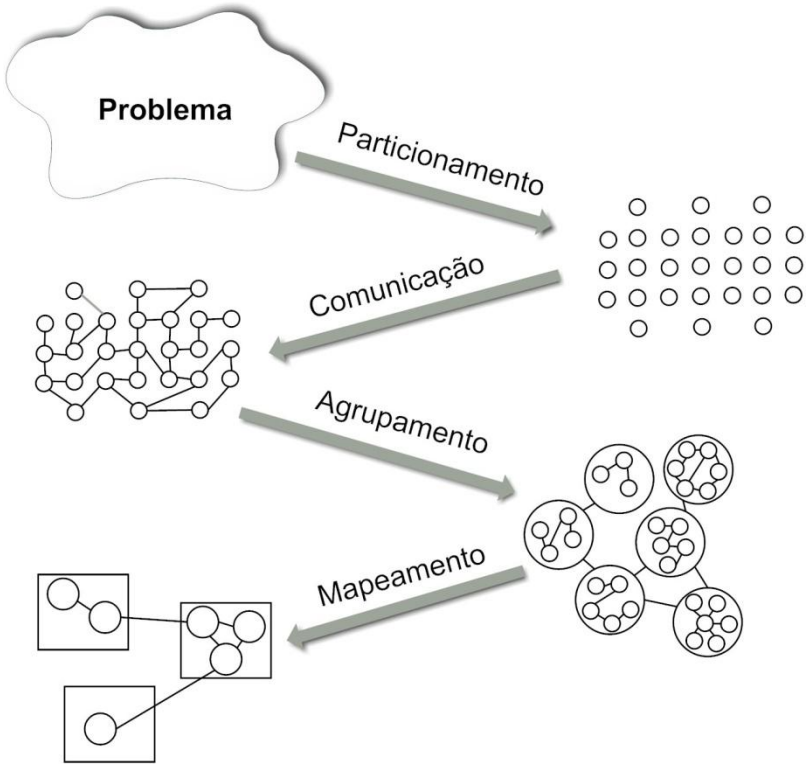

Figura 3.3: Metodologia PCAM. Adaptado de Foster [27]

Etapa 1: Particionamento

Esta etapa tem por objetivo buscar as oportunidades para a execução paralela. Seu foco está em definir um grande número de pequenas tarefas dentre o problema. Conhecido como granularidade fina, esta decomposição promove maior flexibilidade em termos de potencial de paralelismo.

O particionamento pode ser realizado utilizando duas técnicas de partição conhecidas como: decomposição em domínios e decomposição em funções. Ambas devem ser capazes de particionar tanto a computação associada, quanto os dados sobre os quais o cálculo é realizado.

A decomposição em domínios busca decompor os dados de um problema em pequenas partes de tamanhos semelhantes. Após isso, procura-se dividir os cálculos que devem ser executados, associando a cada operação os dados necessários. Esta divisão resulta em um número de tarefas, cada qual possuindo dados e um conjunto de operações sobre estes.

Ao contrário da decomposição por domínios, a técnica de decomposição em funções tem ênfase nos procedimentos. Esta tem seu enfoque inicial no processamento a ser realizado,

buscando dividir o processamento em diferentes tarefas e então identificar os dados necessários a cada tarefa.

Após a conclusão da etapa de particionamento, a partição deve obter um número de tarefas de ordem superior ao número de processadores. Somente desta forma haverá flexibilidade nas fases posteriores do desenvolvimento. Outro fator importante está relacionado às dimensões das tarefas, que devem estar com tamanhos semelhantes no intuito de facilitar o balanceamento de carga[4]. Estes fatores devem ser considerados com o objetivo de validar a solução e podem ser essenciais para decidir se há riscos em prosseguir com o modelo encontrado ou necessidade de reavaliar a especificação do problema.

Etapa 2: Comunicação

As tarefas definidas na etapa de particionamento podem ser executadas concorrentemente. Entretanto, muitas vezes podem não ter uma execução independente, ou seja, algumas tarefas precisam de dados que estão associados a outras tarefas. Isto remete ao conceito de comunicação entre tarefas.

Esta etapa compreende duas fases. Na primeira fase definem-se as estruturas que ligam direta ou indiretamente as tarefas consumidoras (que precisam dos dados) com as tarefas produtoras (que possuem os dados). Na fase seguinte são definidas as mensagens que serão enviadas ou recebidas nos canais.

A procura do melhor desempenho deve-se ainda definir e organizar as operações de comunicação necessárias á satisfação do desenvolvedor da aplicação. Os tipos possíveis de comunicação entre tarefas são definidos abaixo.

- Comunicação *local/global*: Em uma comunicação local cada tarefa se comunica com poucas tarefas vizinhas. De outra forma, na comunicação global cada tarefa se comunica com muitas outras tarefas.
- Comunicação *estruturada/não estruturada*: Em uma comunicação estruturada uma tarefa em conjunto com todos os seus vizinhos constitui uma estrutura regular. Em uma comunicação não estruturada as tarefas constituem grafos arbitrários.
- Comunicação *estática/dinâmica*: cada tarefa tem seus parceiros comunicantes fixos durante o programa. Na comunicação dinâmica, os parceiros de cada tarefa podem variar. Isto é determinado por cálculos da própria execução.

[4] Consiste basicamente em dividir a carga total de processamento pelos vários processadores do sistema.

- Comunicação *síncrona/assíncrona*: Em uma comunicação síncrona as tarefas se comunicam de forma coordenada. Em contraste, a comunicação assíncrona não prevê qualquer tipo de sincronização.

Finalizada esta etapa, pode-se esperar que o número de operações de comunicação seja otimizado para que não afete demasiadamente o desempenho da aplicação.

Etapa 3: Agrupamento

Nas fases anteriores a computação foi divida em tarefas e a comunicação entre elas foi definida. Desta forma, criou-se um algoritmo abstrato e não específico para um determinado tipo de máquina. A fase de agrupamento tem por objetivo fazer uma passagem do mundo abstrato para o real, de modo a adaptar o algoritmo genérico para uma máquina paralela específica, podendo esta ser de memória compartilhada ou distribuída.

No agrupamento, as tarefas identificadas na fase de particionamento são agrupadas novamente visando um número igual de tarefas e processadores, resultando assim em uma redução do custo de comunicação e simplificação do projeto. O agrupamento também é vantajoso sempre que a análise dos requisitos de comunicação revelar que determinadas tarefas não podem ser executadas concorrentemente.

Etapa 4: Mapeamento

A última fase da metodologia especifica onde as tarefas deverão ser executadas. Entretanto, o processo de mapeamento é um problema de grande dificuldade e deve-se procurar as melhor estratégia visando minimizar o tempo total de execução. Com o intuito de atingir este objetivo pode-se discutir duas alternativas. Na primeira estratégia as tarefas que podem ser executadas concorrentemente são vinculadas a processadores distintos para aumentar a concorrência. Na segunda alternativa as tarefas que tem alto nível de comunicação são alocadas no mesmo processador para aumentar a localidade.

O problema do mapeamento é conhecido com *NP-Completo*. Isto significa que não existe um algoritmo que possa encontrar o melhor mapeamento possível em tempo polinomial. O que existe são algumas técnicas especializadas e heurísticas efetivas. Dentre essas técnicas destacam-se o balanceamento de carga utilizado em algoritmos mais complexos de decomposição de domínios e algoritmos de escalonamento de tarefas utilizados para problemas que se baseiam em decomposições em funções.

3.2.2 Metodologia Proposta pelo SEMPA

O SEMPA (*Software Engineering Methods for Parallel Applications in Scientific Computing*) é um projeto interdisciplinar de pesquisa composto por engenheiros mecânicos, analistas numéricos e cientistas da computação da Universidade Técnica de Munique [28]. Este projeto foi fundado em fevereiro de 1995 com o objetivo de desenvolver métodos para aplicações paralelas no domínio da computação científica baseando-se em estudos de caso de práticas industriais.

No sistema de trabalho adotado pela ASC (*Advancced Scientific Computing*), uma parceria do projeto SEMPA [28], o processo de desenvolvimento considera uma combinação entre o modelo Cascata e o modelo Evolucionário[5]. Deste modo, requisitos podem ser especificados ou modificados ao longo do projeto. Neste sentido, o modelo propõe um controle global do projeto que é alcançado com o *loop* global, sendo este dividido em quatro tarefas.

1- *Global Software Requirement Specification* (GSRS): esta fase é a mais importante no processo, pois define o produto a ser construído. Geralmente uma constante interação entre o cliente e os profissionais da área garante isso.

2- *Literature Survey* (LIT): a principais referências necessárias ao projeto devem ser pautadas com antecedência para que estas sejam preparadas para a utilização.

3- *Test Case Definition* (TCD): dependendo do projeto, esta fase propõe definições detalhadas dos casos de teste para o sistema de teste que será preparado. O objetivo dos testes é manter uma estabilidade entre o código e os requisitos definidos na GSRS.

4- *Global Strategic Plan* (GSP): tem o objetivo de controlar as fases de projeto e implementação. O projeto é divido em vários módulos chamados SEM (*Software Engineering Modules*). Estes módulos são ranqueados baseando-se nas prioridades do cliente e custos de desenvolvimento. Também são definidas as responsabilidades pessoais e as estimativas de tempo, datas de entrega e *milestones*[6] para cada módulo SEM.

Dentro da GSP, o projeto como um todo é dividido em módulos com a estrutura similar a um modelo Cascata. Cada módulo segue os seguintes passos:

---

[5] Os modelos evolucionários são iterativos. Estes promovem aos engenheiros de software desenvolver versões cada vez mais completas do software. Como exemplo deste modelo cita-se o modelo em espiral descrito na seção 2.4.2 deste trabalho.

[6] Termo utilizado como designação de um ponto de controle dentro de um cronograma através da definição de pontos de checagem ou marcos de desenvolvimento, pode representar a conclusão de um conjunto de tarefas ou fase.

a. *Software Requirement Specification* (SRS): escrita e revisão do documento de especificação dos requisitos de software.

b. *Module Design Phase* (MDP): escrita e revisão do documento da fase de projeto do módulo.

c. *Source Code Implementation* (SCI) e *Testing* (TEST): escrita e implementação do código, teste de implementação, revisão da implementação e execução dos testes.

d. *Documentation* (DOC): escrita da documentação baseado nos documentos de projeto disponíveis e revisão da documentação

Depois desta etapa destinada a cada SEM, o processo é finalizado com uma nova etapa de documentação (DOC) seguida de um *release* do *software* (REL) e posterior manutenção do *release* (PRM). O processo proposto pela ASC pode ser observado na figura 3.4.

É interessante observar que existem pessoas ou grupos associados a cada tarefa. Em um processo de desenvolvimento da ASC as atividades são vinculadas em termos destes indivíduos ou grupos citados abaixo:

- Engenheiro de projeto: pessoa encarregada de desenvolver o software;
- Gerente de projeto: pessoa responsável pelo gerenciamento do projeto;
- Parceiros Externos: universidade ou parceria industrial;
- Clientes;
- Alta gerência da ASC: responsável pela coordenação dos projetos.

Baseado na figura 3.4, um novo modelo foi criado aplicado ao desenvolvimento de uma aplicação paralela. Este modelo considera um software existente e idealmente segue seis fases dentro do ciclo de desenvolvimento. O modelo é ilustrado na figura 3.5.

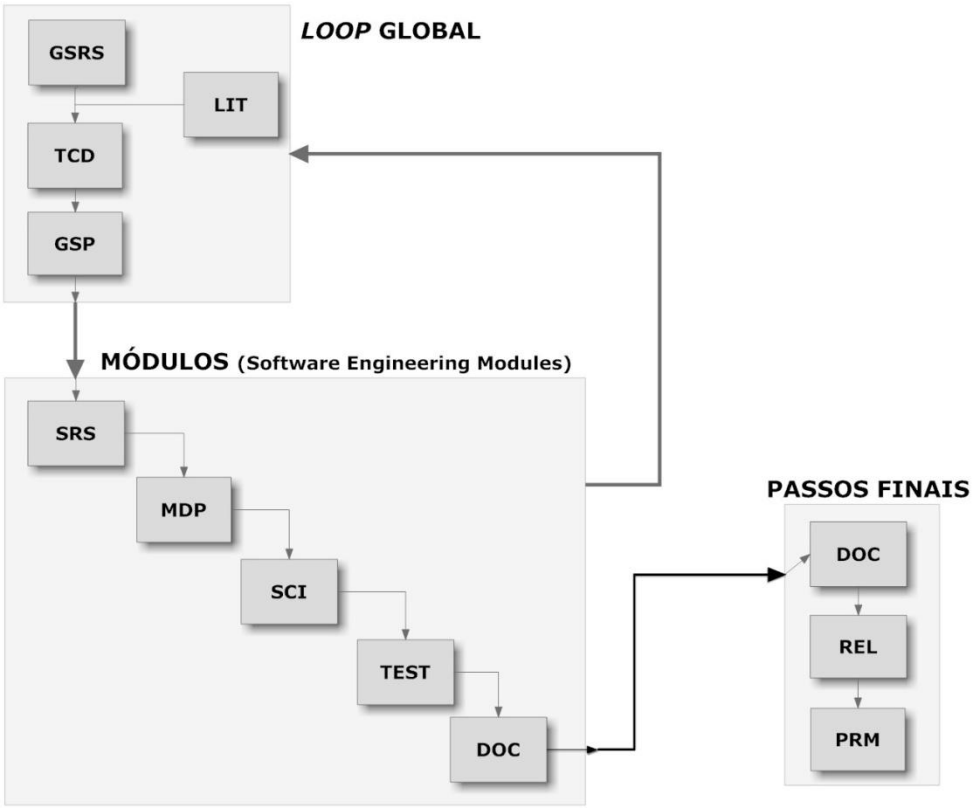

Figura 3.4: Conceito evolucionário de engenharia de software utilizado pelo ASC. Adaptado do SEMPA [28]

1. Análise e documentação do código sequencial: nesta fase os algoritmos e implementações sequenciais devem ser entendidos por cientistas da computação que possuem o conhecimento na área do domínio da aplicação. A documentação descreve algoritmos em nível de abstração, proporcionando uma noção de como o domínio da aplicação pode ser fragmentado em sub-rotinas. A documentação resultante desta análise serve como referência para todo o projeto.

2. Conceito de paralelização: com base na documentação do projeto, os cientistas da computação identificam as dependências em nível de algoritmo e as possíveis abordagens de paralelização. Deve-se determinar a abordagem mais eficiente levando em consideração a arquitetura utilizada, bem como a formulação do pseudocódigo desta solução.

3. Especificação global de requisitos: reuniões entre o grupo de desenvolvedores do *software*, peritos em *hardware*, gerentes e usuários definem os objetivos gerais do projeto. A GSRS (*Global Software Requirement Specification*) define as funcionalidades do programa a ser implementados, requisitos de desempenho, plataforma de software além do conjunto de situações de testes para validar e analisar o desempenho.

4. Definição global do planejamento de projeto dos SEMs: esta fase é destinada a implementação da solução adotada. O GSP (*Global Strategic Plan*) decompõe o processo de implementação em várias sub-tarefas de complexidade gerenciável. Cada SEM é caracterizado pela definição de um conjunto de requisitos funcionais e interface. A documentação é relativa as dependências entre cada módulo SEM e seu tempo de planejamento.

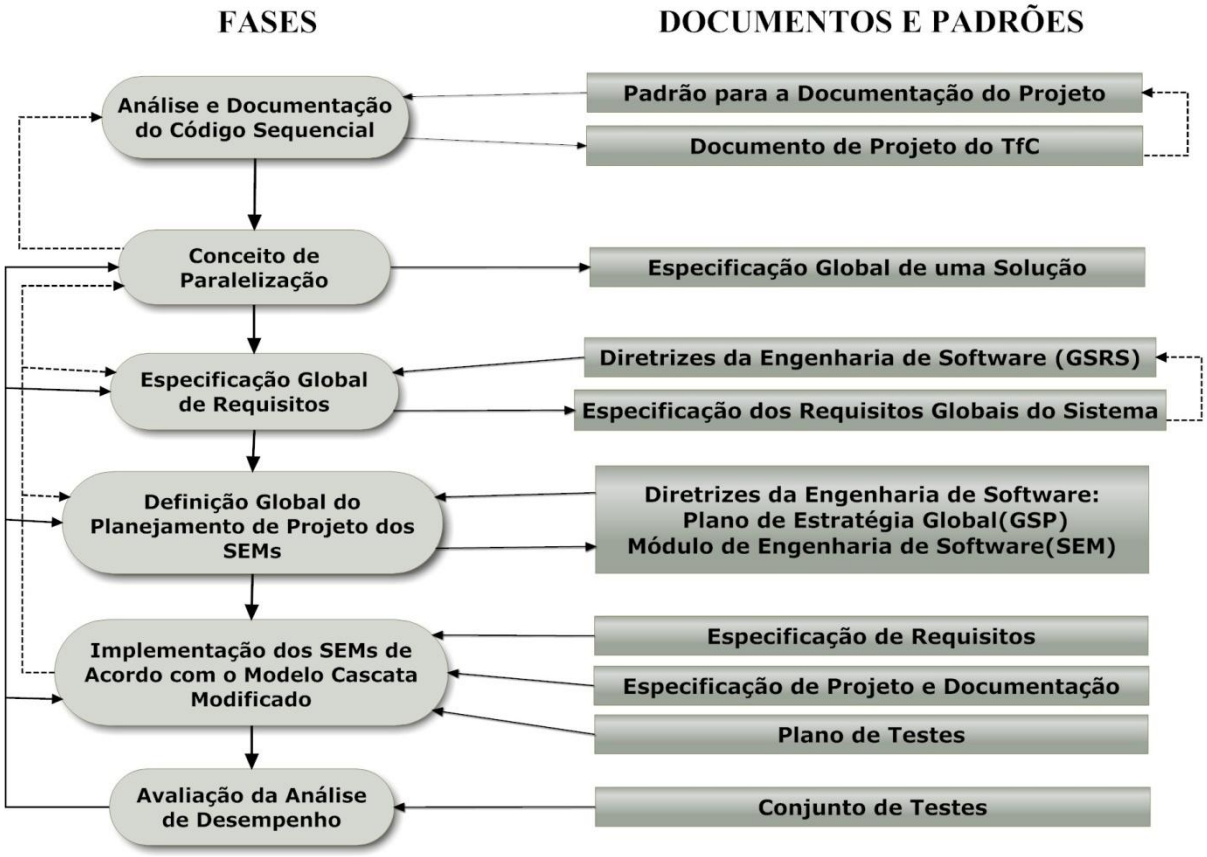

Figura 3.5: Projeto e Implementação de um programa paralelo: Os Processos de engenharia de software. Adaptado do SEMPA [28]

5. Implementação dos SEMs: cada SEM é implementado de acordo com o modelo Cascata modificado. Depois de cada estágio dentro do modelo Cascata, membros do projeto que não são os desenvolvedores, realizam uma revisão. Desta forma o processo de desenvolvimento avança para o próximo estágio ou retorna ao anterior.

6. Avaliação da análise de desempenho: após a conclusão dos SEMs, o programa paralelo é avaliado como um todo. Se passar em todos os testes, um *release* é destinado ao

usuário, este pode indicar os possíveis erros e novas funcionalidades a serem acrescentadas para os futuros *releases*.

3.2.3 Metodologia UMP^{2}D

A UMP^{2}D (*Unified Methodology for Parallel Programs Development)* [29] é uma metodologia orientada a objetos para o desenvolvimento de aplicações paralelas. Sua principal característica está relacionada à utilização da linguagem de modelagem UML e tem como objetivo suprir as principais necessidades do desenvolvimento deste tipo de aplicação, como balanceamento de carga e minimização da comunicação.

O ciclo de vida do UMP^{2}D é baseado em uma combinação do modelo Cascata e o modelo RUP. Desta forma, o processo de desenvolvimento é dividido em seis fases. O modelo é ilustrado na figura 3.6.

Figura 3.6: Ciclo de vida do UMP^{2}D. Adaptado de Olivete [29]

1. Coleta de Requisitos: nesta faze define-se o escopo da aplicação. Uma coleta de informações é realizada para servir como base nas etapas de modelagem e implementação. O diagrama de Casos de Uso pode ser utilizado nesta fase, no sentido de representar visualmente o contexto da aplicação e a interação com o usuário. Porém, na maioria das aplicações paralelas não há iteração com o usuário e este diagrama pode ser descartado.

2. Elaboração: esta fase compreende uma grande parcela no desenvolvimento da aplicação. É subdividida em três etapas que formam um ciclo iterativo interno: Análise, projeto e análise de desempenho.

- Análise: tendo como base os requisitos coletados na primeira fase, o desenvolvedor deve especificar o diagrama de classes definindo-se apenas os atributos e seus relacionamentos. Além disso, deve-se iniciar a elaboração do diagrama de atividades, que tem o objetivo de expressar o comportamento interno de cada caso de uso. Este diagrama tem a capacidade de expressar o paralelismo através de barras de sincronização.
- Projeto: esta etapa inicia-se com o refinamento do diagrama de classes, adicionando-se métodos e outros conceitos permitidos pelo UML, como relações de agregação, especialização e generalização. Também deve ser definida a arquitetura que será utilizada, bem como os processadores e os canais comunicação entre eles. Por último é realizado o mapeamento, onde cada método é ligado a um elemento de processamento. Para cada método, é necessário que o desenvolvedor atribua um valor estimado, de forma a conduzir posteriormente uma análise de carga de cada processador. Além do diagrama de classes utilizado no início da fase, deve-se elaborar o diagrama de sequência mostrando o paralelismo em nível de objeto, o diagrama de colaboração que permite visualizar as mensagens enviadas e recebidas pelos objetos e o diagrama de distribuição que demonstra quais objetos serão executadas em cada processador.
- Análise de desempenho: Baseando-se no valor de carga atribuído a cada um dos métodos, uma análise de desempenho é realizada visando os requisitos essenciais da aplicação, como por exemplo, se há muita comunicação entre as tarefas. Caso esta análise não for satisfatória, deve-se voltar ao inicio da iteração, ou seja, refazer ou alterar a etapa de análise.

3. Implementação: A partir do modelo criado nas fases anteriores, o desenvolvedor deve codificá-lo em uma linguagem de programação.

4. Testes: Nesta fase são executados conjuntos de testes no sentido de verificar a qualidade da aplicação resultante. Devem-se localizar os possíveis erros nas etapas anteriores e corrigi-los.

5. Análise de Paralelismo: Tem o propósito de analisar o desempenho do *software* através da visualização das informações de rastreio. Estas informações devem ser geradas durante a execução do programa ou através de monitoração *on-line*. Desta forma, as informações adquiridas nesta fase servem para a atualização do modelo, no sentido de atribuir valores reais a métodos e classes. As ferramentas utilizadas para esse propósito devem ser escolhidas de acordo com a linguagem e da biblioteca de comunicação utilizadas.

6. Transição: Na etapa final da metodologia, é realizada a instalação da aplicação na máquina destino para a utilização dos usuários. Ao final da etapa a aplicação deve estar na sua capacidade máxima para a resolução dos problemas por ela implementados.

3.2.4 Discussão sobre as metodologias de Software Paralelo

Analisando as metodologias de desenvolvimento de software paralelo citadas nas seções anteriores, pode-se identificar que estas se preocupam com eventuais mudanças que podem ocorrer em fases anteriores do desenvolvimento, resultantes de requisitos mal especificados ou estratégias ineficazes. As metodologias PCAM e SEMPA partem do princípio que o *software* desenvolvido já deve existir, ou seja, deve-se paralelizar um software sequencial. Por outro lado, a metodologia UMP^{2}D vislumbra o desenvolvimento de um *software* paralelo desde o início, não sendo adaptado de um software sequencial.

Considerando-se o modelo de processo, a metodologia PCAM sugere um modelo incremental, porém não especifica como devem ser realizadas as interações. Já na metodologia proposta pelo SEMPA, o modelo é resultante de uma combinação do modelo Cascata e o modelo Evolucionário. Desta forma, requisitos globais são definidos em uma etapa inicial e requisitos específicos são realizados em cada módulo SEM. A metodologia UMP^{2}D promove a utilização do modelo Cascata combinado com o modelo RUP, o que proporciona uma documentação mais abrangente, principalmente pelo fato de se utilizar os diagramas UML em cada fase de desenvolvimento.

Outra questão importante está atrelada ao quanto à metodologia abrange as fases de um desenvolvimento completo de *software*. Pode-se notar que a metodologia PCAM envolve somente a parte de tradução de algoritmos sequenciais em algoritmos paralelos, além de propor as etapas de uma forma genérica, não deixando claro quais são as técnicas que o desenvolvedor ou organização deverá utilizar em cada uma das etapas. De maneira diferente, o SEMPA propõe um modelo mais robusto onde há um controle maior de informações associadas ao desenvolvimento. Neste sentido, a metodologia SEMPA oferece melhor análise

e documentação dos artefatos ao longo do projeto. Apesar de ser mais detalhada, a forma de realizar cada etapa também é abstraída nesta metodologia. Semelhante a metodologia SEMPA, a UMP^{2}D também oferece um processo completo, onde as fases são associadas a uma documentação UML possibilitando melhor manutenção do *software*. Se considerar a questão organizacional, apenas o SEMPA no modelo proposto pela sua parceria ASC, descreve como seus *stakeholders* estão relacionados às tarefas inseridas no projeto.

Avaliando as metodologias de forma geral, fica claro que mesmo apresentando as etapas para a paralelização de aplicações, tais metodologias apresentam poucas soluções pontuais para cada uma das etapas envolvidas, ou seja, não expõe técnicas de forma objetiva para a utilização no processo, destaca-se apenas a utilização dos diagramas UML pela metodologia UMP^{2}D. Dessa forma, no próximo Capítulo, apresenta-se uma proposta de metodologia para o desenvolvimento de aplicações paralelas onde procura-se oferecer soluções para cada etapa do desenvolvimento de um *software* paralelo visando um processo mais robusto e uma melhor orientação ao desenvolvedor ou organização que deseja desenvolver este tipo de aplicação.

Capítulo 4

Metodologia MOODAP

Uma boa metodologia de desenvolvimento de software deve possuir aspectos que guiem o desenvolvedor na construção de um software de qualidade, bem como melhor organização dos artefatos produzidos e redução de prazos e custos de desenvolvimento.

Atualmente, o foco de desenvolvimento de sistemas está voltado ao paradigma de programação orientada a objetos, principalmente pelas vantagens associadas a esta tipo de desenvolvimento, tais como, reutilização de código, programação em nível mais elevado, modularidade, menor custo de desenvolvimento, facilidade de manutenção, entre outros. Neste sentido, a metodologia desenvolvida neste trabalho será focada neste tipo de desenvolvimento. Agregada a orientação a objetos, também será utilizada a *Unified Modelling Language* (UML), uma linguagem ou notação de diagramas para especificar, visualizar e documentar modelos de *software* orientados por objetos [34].

As características singulares de uma aplicação paralela necessitam de um tratamento diferenciado no aspecto de desenvolvimento da aplicação. Neste capítulo objetiva-se propor a metodologia MOODAP (Metodologia Orientada a Objetos para o Desenvolvimento de Aplicações Paralelas) onde procura-se desenvolver um *software* paralelo desde o início visando um processo mais robusto e objetivo em termos de técnicas e ferramentas utilizadas em cada fase do ciclo de vida.

Fundamentado em conceitos das metodologias de desenvolvimento descritas na literatura e considerando aspectos relevantes para este tipo de aplicação, é proposto o modelo da metodologia MOODAP, que considera o desenvolvimento desde o início da aplicação e contém as seguintes fases no seu ciclo de vida: especificação de requisitos, análise e projeto de *software*, implementação, testes de *software* e análise de desempenho.

As fases e as informações produzidas e utilizadas na metodologia podem ser observadas na figura 4.1.

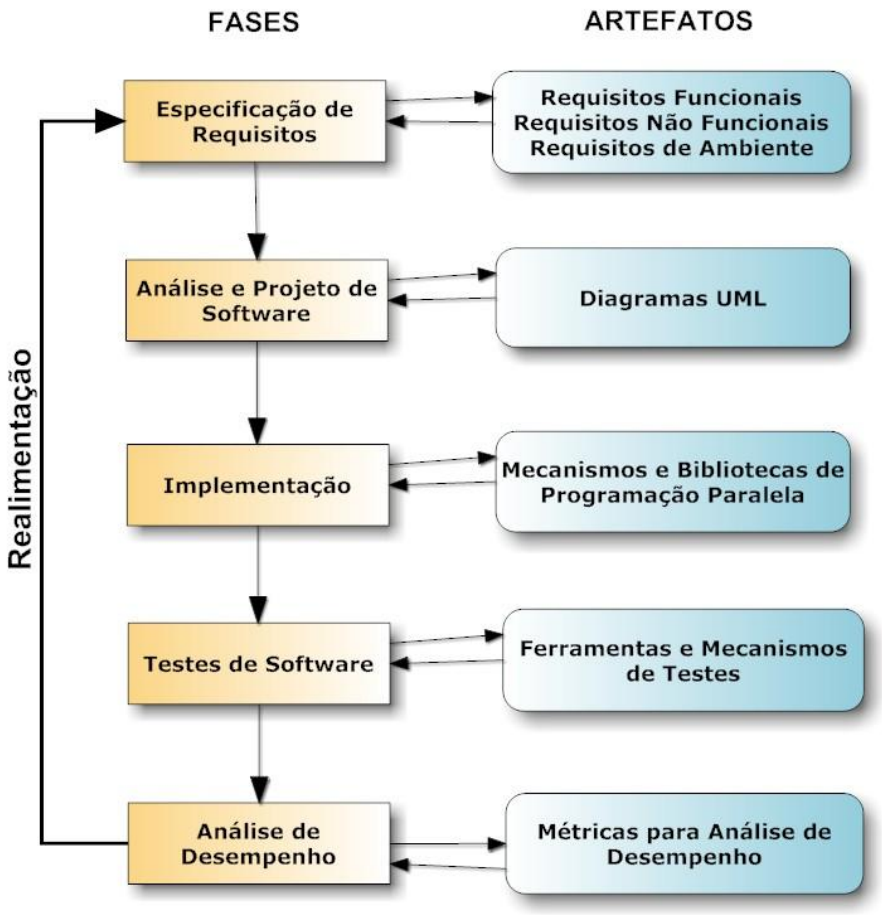

Figura 4.1 Metodologia Orientada a Objetos para o Desenvolvimento de Aplicações Paralelas – MOODAP

4.1 Especificação de requisitos

O processo de desenvolvimento de uma aplicação paralela inicia-se a partir da especificação dos requisitos. Esta etapa é fundamental, pois define o produto a ser construído. Neste contexto, a metodologia MOODAP subdivide esta fase em: requisitos funcionais, não-funcionais e requisitos de ambiente.

4.1.1 Requisitos funcionais

Os requisitos funcionais descrevem o comportamento do sistema. Este comportamento pode ser expresso através de serviços, tarefas ou funções que o sistema pode executar. Além disso, demonstra a interação do usuário com o sistema [32, 34].

Dependendo do foco da aplicação paralela em desenvolvimento, pode haver pouca interação com o usuário, desta forma não há necessidade de um levantamento de requisitos junto ao cliente, inviabilizando a construção de um documento mais específico. Como

exemplo pode-se citar alguns trabalhos científicos, onde as aplicações são executadas no estilo “caixa preta”, normalmente utilizando um *script* de entrada.

Porém, em uma aplicação comercial, pode-se considerar que as interações com o usuário podem ser descritas de forma semelhante a uma aplicação não paralela.

Caso de Uso*: <número> << o nome é um objetivo descrito com uma frase curta contendo um verbo na voz ativa >>*

INFORMAÇÃO CARACTERÍSTICA

Objetivo no Contexto*: <uma sentença mais longa do objetivo do caso de uso se for necessário>*

Escopo: *<Qual sistema está sendo considerado (por exemplo, organização ou sistema computacional)>*

Nível: *<um dos tipos de objetivo: objetivo de usuário, objetivo de contexto ou objetivo de subfunção>*

Pré-condições: *<o que é necessário que já esteja satisfeito para realizar o caso de uso>*

Condição Final de Sucesso: *<o que ocorre/muda após a obtenção do objetivo do caso de uso>*

Condição Final de Falha: *<o que ocorre/muda se o objetivo é abandonado>*

Ator Primário: *<o nome do papel para o ator primário, ou descrição>*

CENÁRIO PRINCIPAL DE SUCESSO

<coloque aqui os passos do cenário necessários para a obtenção do objetivo >

<passo #> <descrição da ação >

EXTENSÕES

<coloque aqui as extensões, uma por vez, cada uma referenciando o passo associado no cenário principal >

<passo alterado> <condição> : <ação ou sub.caso de uso >

<passo alterado > <condição> : <ação ou sub.caso de uso >

INFORMAÇÃO RELACIONADA (opcional)

Prioridade: *<Quão crítico é o caso de uso para seu sistema/organização >*

Desempenho alvo: *<o total de tempo que este caso de uso poderia demorar >*

Frequência: *<com que frequência espera-se que o caso de uso ocorra >*

Caso de Uso Pai: *<opcional, nome do caso de uso que inclui este >*

Casos de Uso Subordinados: *<opcional, ligações para sub.casos de uso >*

Atores Secundários: *<lista de outros sistemas necessários para realizar este caso de uso >*

Quadro 4.1 *Template* de caso de uso proposto por Cockburn [35]

Os casos de uso [34] são uma das técnicas mais utilizadas pelos desenvolvedores para representar os requisitos funcionais de forma textual e gráfica. Um caso de uso é composto

por cenários. Estes são definidos como uma sequência de passos que descreve uma interação entre o usuário e um sistema.

Não existe uma forma padrão de descrever o conteúdo de um caso de uso e diferentes formatos funcionam bem em diferentes casos [34]. A figura 4.1 ilustra a descrição textual pelo *template* proposto por Cockburn [35].

Além da descrição textual, a UML define um diagrama para representar os casos de uso. A sua representação pode ajudar no entendimento e funciona como um sumário gráfico do conjunto de casos de uso. Neste trabalho, será utilizada uma aplicação de remoção de ruídos de imagens como exemplo na confecção dos diagramas ao longo da fase de projeto. Este estudo se trata basicamente em remover ruídos de uma imagem original através de um filtro suavização (passa-baixa). A figura 4.2 ilustra um exemplo do diagrama de casos de uso para esta aplicação.

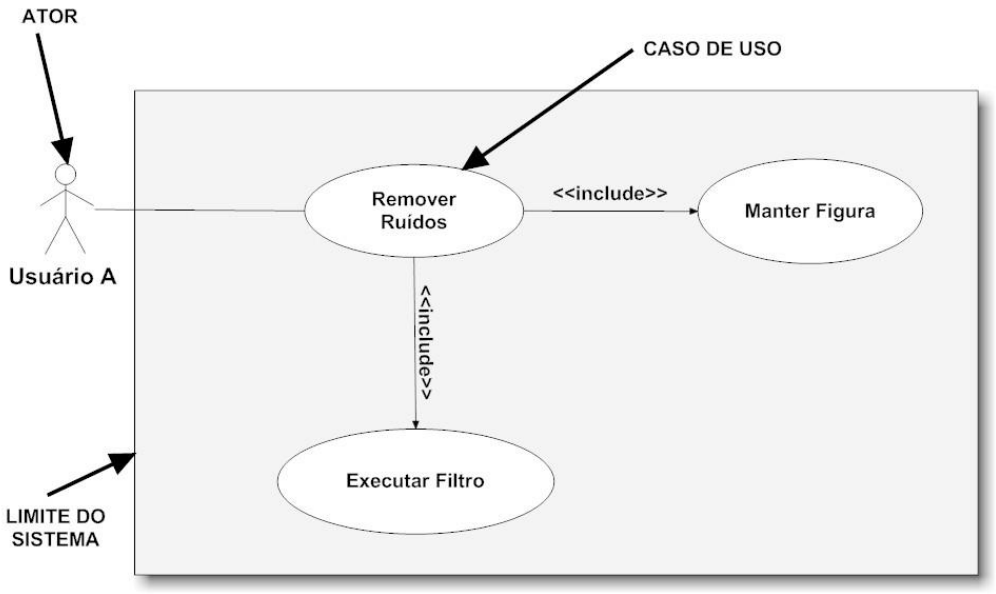

Figura 4.2 Exemplo de diagrama de casos de uso para o exemplo de remoção de ruídos.

É importante ressaltar, que a apesar dos casos de uso ser uma valiosa ferramenta para ajudar no entendimento nos requisitos funcionais de um sistema, outras alternativas podem ser utilizadas para a elicitação de requisitos. O *Extreme Programming* (XP) [15] possui estórias de usuário para guiar o desenvolvedor no que deve ser implementado. Essas estórias são descrições mais simples se comparadas aos casos de uso e fornecem uma boa maneira de dividir o sistema para a implementação em processo iterativo onde varias estórias são selecionadas e implementadas em cada iteração.

4.1.2 Requisitos não-funcionais

Os requisitos não-funcionais não estão preocupados diretamente com as funcionalidades do sistema, mas em fixar restrições sobre como os requisitos funcionais serão implementados. Estes requisitos incluem usabilidade, segurança, confiabilidade, eficiência, desempenho, entre outros [30].

Não há uma conformidade global sobre uma relação completa de todos os requisitos não-funcionais, porém, existem várias propostas para a classificação destes requisitos. Neste trabalho será utilizada a classificação proposta por Sommerville [31] que distribui os requisitos em três categorias: requisitos de produto, requisitos de processo e requisitos externos. A estrutura com a classificação dos requisitos pode ser observada na figura 4.3.

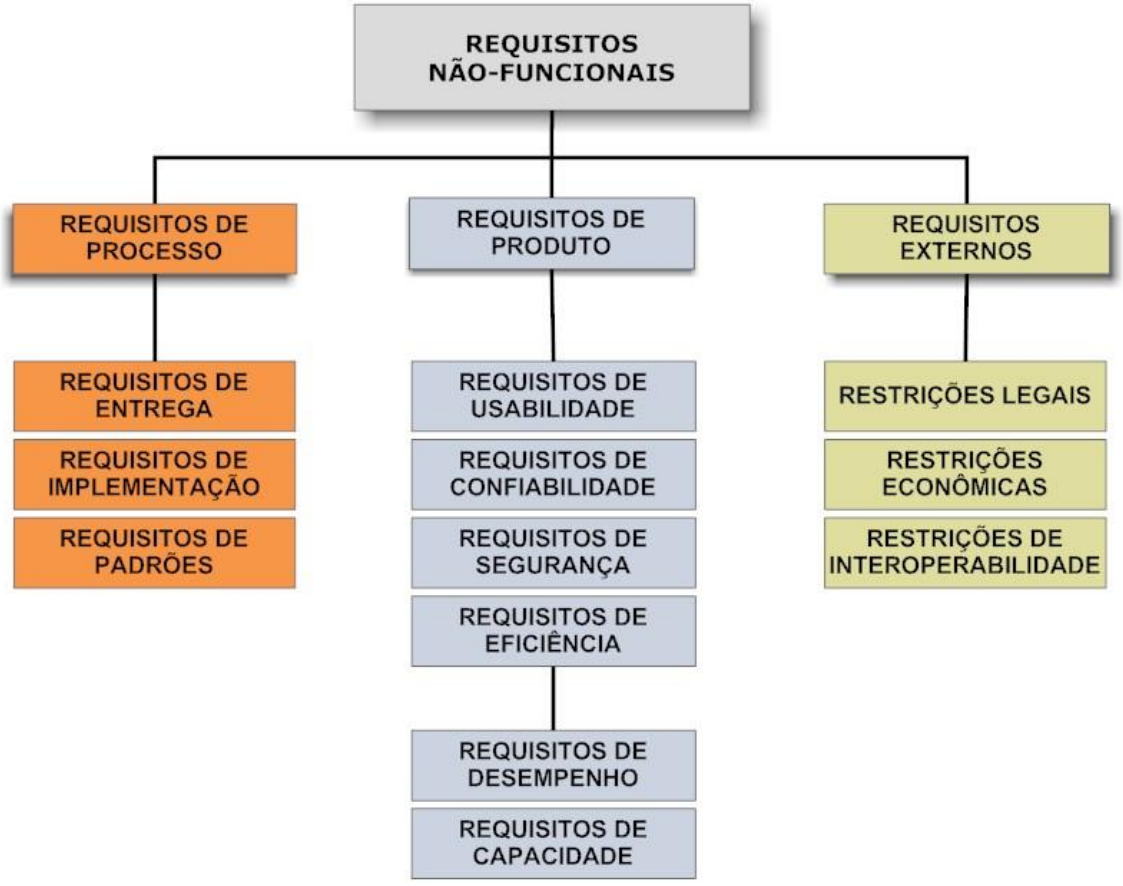

Figura 4.3 Classificação dos requisitos não-funcionais [31].

Neste contexto de requisitos não funcionais, pode-se destacar os requisitos de produto que especificam as características que o sistema deve possuir. Em uma aplicação paralela estes requisitos são consideramos essenciais, pois definem o comportamento do sistema em execução e limitações de projeto. A Tabela 4.1 ilustra como alguns requisitos não-funcionais de produto podem ser especificados.

Requisito não-funcional	Especificação (exemplo)
Confiabilidade	A taxa de ocorrência de falhas para o sistema Y deve ser menor que 0,01%. O tempo de reinicio após uma falha deve der menor que 1 segundo.
Desempenho	O sistema Y deve processar no mínimo 10 transações por segundo. O tempo de atualização da tela para o usuário deve ser menor que 1 segundo.
Capacidade	O código executável do sistema Y deve ser limitado a 1024 Kbytes, ou seja, especifica o tamanho máximo de memória que o sistema deve utilizar.
Segurança	O sistema Y deverá utilizar o algoritmo de criptografia RSA para todas as comunicações externas.

Tabela 4.1 Exemplo de especificação de requisitos não-funcionais de produto [30, 31].

Além dos requisitos de produto, outras questões importantes devem ser levadas em consideração. Dentre estas, destaca-se mecanismos de balanceamento de carga e tolerância a falhas.

Um dos maiores problemas relacionados a aplicações paralelas é o de melhorar a distribuição dos recursos computacionais entre os nós do sistema. A utilização de um esquema de balanceamento de carga visa maximizar a utilização dos recursos, possibilitando melhor desempenho do sistema.

Um esquema de balanceamento de carga pode ser classificado de acordo ao estado atual do sistema, em estático ou dinâmico. Os esquemas estáticos de balanceamento de carga são independentes do estado do sistema, ou seja, considera que o estado do sistema não se altera ou se altera muito pouco. O contrário acontece com os esquemas dinâmicos que sempre consideram o estado do sistema e podem realizar a distribuição de carga se necessário [38]. Apesar de ser uma área de pesquisa em constante evolução e apresentar inúmeras soluções

para este problema, a escolha de alguma alternativa para o balanceamento de carga não traz uma solução ótima, por se tratar de um problema NP completo, porém seu ganho de desempenho deve ser considerado e implementado em aplicações paralelas em aplicações mais robustas.

Grande parte dos sistemas que executam aplicações paralelas são classificados como *clusters* ou *grids.* Estas aplicações muitas vezes, executam uma grande quantidade de dados e levam um tempo considerável até o final do processamento, uma falha em algum nó do sistema pode comprometer toda a execução e não atingir o resultado esperado. Desta forma há necessidade de algum mecanismo que garanta a tolerância a falhas neste tipo da aplicação.

A maioria das técnicas para tolerância a falhas se encaixa em duas categorias principais: *checkpointing/rollbacking* e replicação [37]. A técnica de *checkpointing/rollbacking* utiliza um processo periódico de salvamento em um armazenamento não volátil de determinados estados do programa em cada nó do sistema, desta forma, caso ocorra alguma falha, os dados possam ser recuperados.

De maneira diferente, a técnica de replicação armazena cópias de informações relevantes para o sistema (como por exemplo, uma mensagem trocada entre nós), deste modo estas informações são consultadas no caso de uma falha.

Obviamente que a utilização de qualquer técnica de tolerância a falhas exigirá requisitos maiores do sistema que um que não seja tolerante a falhas. O problema está em balancear de forma eficiente a recuperação do sistema, a eficiência das técnicas utilizadas e seus requisitos de ambiente [37]. Pode-se analisar que estas técnicas foram desenvolvidas para atuar em sistemas com características e requisitos diferentes, dependo do caso pode ser mais custosa ou não. O que necessita de uma avaliação cuidadosa das particularidades de cada sistema.

É interessante ressaltar que alguns requisitos não-funcionais são dependentes da definição dos requisitos de ambiente. Isto pode ser observado considerando o requisito não-funcional segurança. Este requisito pode ser tratado de forma diferente se for aplicado a um *grid* ou *cluster*, já que a execução em um *cluster* é realizada localmente sem a necessidade de uma segurança maior do sistema, ao contrário de um *grid* que pode utilizar a internet para a troca de dados. Outro exemplo de dependência pode ser observado no requisito não-funcional capacidade, o qual depende do ambiente de execução, já que os nós do sistema devem atender a demanda de recursos computacionais, tais como memória ou disco.

4.1.3 Requisitos de Ambiente

Após a definição dos requisitos funcionais e não-funcionais, é necessário descrever alguns aspectos relacionados ao ambiente e arquitetura utilizada. Dependendo do conhecimento do cliente em relação às particularidades de um desenvolvimento de aplicações paralelas, esta tarefa de especificação pode ser desenvolvida em comum acordo entre cliente e desenvolvedores, ou como uma solução proposta pela equipe de desenvolvimento do projeto. Neste trabalho, definimos este levantamento de informações como requisitos de ambiente. A Tabela 4.2 ilustra os principais fatores que devem ser especificados neste etapa.

Requisito de ambiente	Descrição
Plataforma Computacional	Define em qual plataforma o sistema será executado (Windows, Linux, Solaris). Deve-se definir também se a estação de trabalho será homogênea ou heterogênea, permitindo maior portabilidade ou não ao sistema.
Arquitetura (processador)	A arquitetura de processador descreve o processador que é usado em um computador. Como exemplo pode-se citar: Pentium, AMD, SPARC, MIPS, PowerPC, entre outros.
Arquitetura Paralela	Define a estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo. Dentre a arquitetura MIND (*Multiple Instruction stream over a Multiple Data stream*), deve-se optar por memória compartilhada ou distribuída.
Quantidade de Processadores	Define o número de processadores ou nós que serão utilizados no sistema.
Linguagens e Bibliotecas	Qual a linguagem utilizada na implementação do sistema. Como Java, C, C++, Fortran. Definição dos padrões de comunicação como bibliotecas e APIs (*Application Programming Interface*). Podem ser utilizados mecanismos

	como: MPI, PVM, OpenMP, Pthreads, entre outros.
Capacidade de Memória	Especifica a quantidade de memória, em megabytes, para cada nó do sistema.
Capacidade de Disco	Especifica a quantidade de armazenamento em disco, em megabytes, para cada nó do sistema.
Rede	Definir a estrutura da rede e a adequação ao sistema proposto. Alguns aspectos devem ser considerados. Dentre eles, latência, banda e topologia. O tipo de rede também deve ser determinado: As mais utilizadas: Gigabit Ethernet, Infiniband, Myrinet, SCI.

Tabela 4.2 Exemplo de especificação de requisitos de ambiente.

A definição dos requisitos de ambiente requer um alto nível de conhecimento da equipe de desenvolvimento ou desenvolvedor, pois suas escolhas implicaram na redução ou aumento dos custos e no desempenho da aplicação. A Tabela 4.2 apresenta de forma sumarizada alguns aspectos relevantes que devem ser especificados nesta etapa, porém, em um projeto real deve-se dedicar um tempo considerável para a análise das alternativas dentre os aspectos citados acima e optar por aquelas que agradem o cliente e sejam eficientes em relação ao sistema.

Após a definição dos requisitos funcionais, não-funcionais e requisitos de ambiente, o desenvolvedor de uma aplicação paralela possui as informações necessárias para as próximas etapas da metodologia de desenvolvimento.

4.2 Análise e projeto de software

Esta fase inicia-se tendo como base as informações adquiridas com os requisitos especificados na fase anterior. A partir destes, é necessário criar um modelo conceitual com o objetivo de expressar as principais funções e relações do sistema proposto.

Em um desenvolvimento orientado a objetos, o diagrama de classes é comumente utilizado para mostrar a estrutura da aplicação, ou seja, a colaboração e interação entre estas entidades para a execução das funcionalidades do sistema.

Para vislumbrar um melhor entendimento, é necessário definir alguns conceitos básicos envolvidos na definição e representação das classes de um sistema.

Uma classe pode ser definida como uma “descrição genérica ou coletiva de uma entidade do mundo real” [36], ou seja, representa um modelo comum para um conjunto de entidades semelhantes. Em uma notação para classes em UML, uma classe é representada em três subdivisões, a identificação da classe, os atributos e os métodos [36].

- Identificação da classe: defini-se o nome da classe e, opcionalmente, um estereótipo e um identificador de pacote.
- Atributos: define-se as variáveis membro da classe que podem ser usadas por todos os seus métodos tanto para acesso quanto para escrita.
- Métodos: são declarados os métodos, ou seja, as funções que a classe possui.

Obviamente que o diagrama de classes apresenta uma gama maior de representações e detalhes em sua especificação, como por exemplo, os tipos de relacionamentos entre as classes e a visibilidade dos métodos [36]. Todavia, os detalhes do diagrama não serão discutidos neste trabalho, pois entende-se que o desenvolvedor possua alguma experiência em um desenvolvimento que utiliza UML.

Baseando-se no exemplo de diagrama de casos de uso desenvolvido na etapa anterior, pode-se definir um diagrama de classes ilustrado na Figura 4.4.

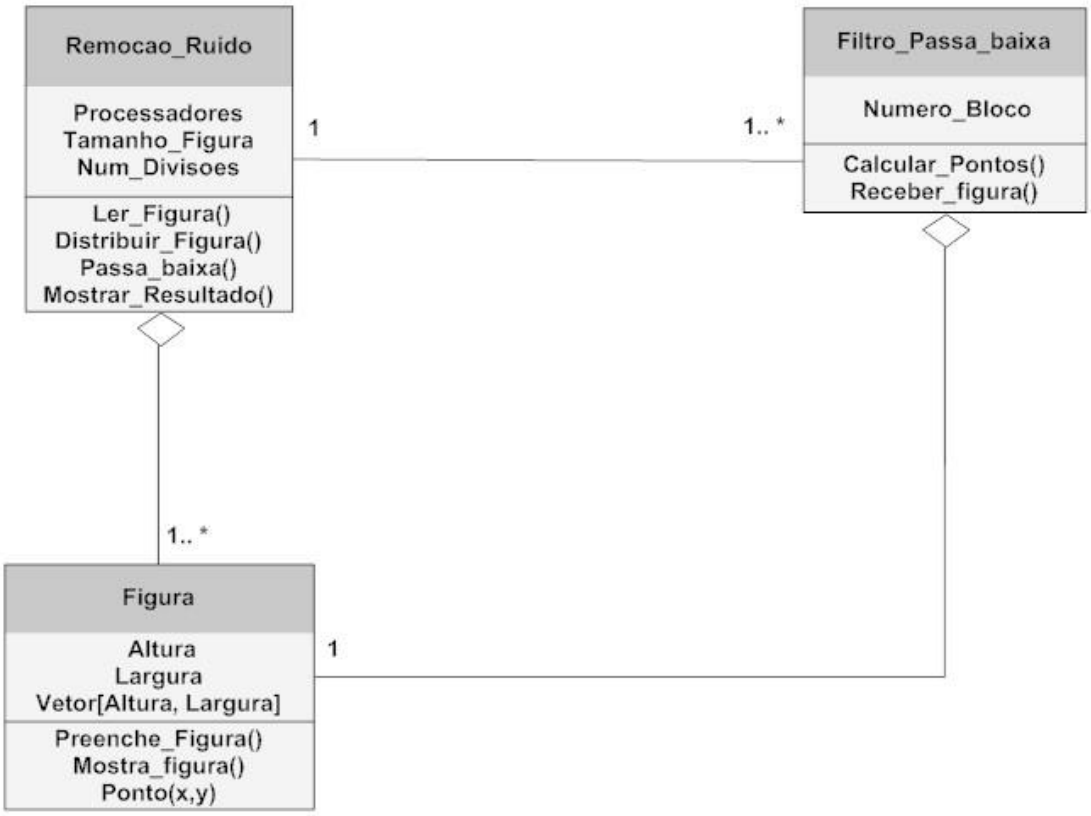

Figura 4.4 Diagrama de classes da aplicação Remoção de Ruídos.

Após a definição das classes e funcionalidades do sistema, o desenvolvedor da aplicação deve encontrar uma maneira de introduzir o paralelismo em partes do código que possuem potencial de execução em paralelo. Neste sentido, há necessidade de identificar e demonstrar quais atividades serão paralelizadas ou não. Para facilitar o entendimento da aplicação, o diagrama de atividades deve ser criado nesta fase para representar a execução de ações ou atividades dentro do sistema.

Uma característica importante relacionada a este diagrama, em especial ao desenvolvimento de aplicações paralelas, está na notação de sincronismo de concorrência. Esta notação representada por uma barra preenchida é utilizada tanto para abertura de sincronismo quanto para sincronismo de concorrências. A partir da abertura, os fluxos seguintes devem avançar concorrentemente. O sincronismo de encerramento de concorrências representa um ponto final da concorrência onde todos os fluxos concorrentes alcançam um ponto de sincronismo [36].

Pode-se afirmar que neste diagrama inicia-se o processo de particionamento descrito na metodologia PCAM, onde as oportunidades de paralelização são definidas visando o potencial de paralelismo da aplicação. Um exemplo de diagrama de atividades pode ser observado no exemplo do filtro passa-baixa na Figura 4.5.

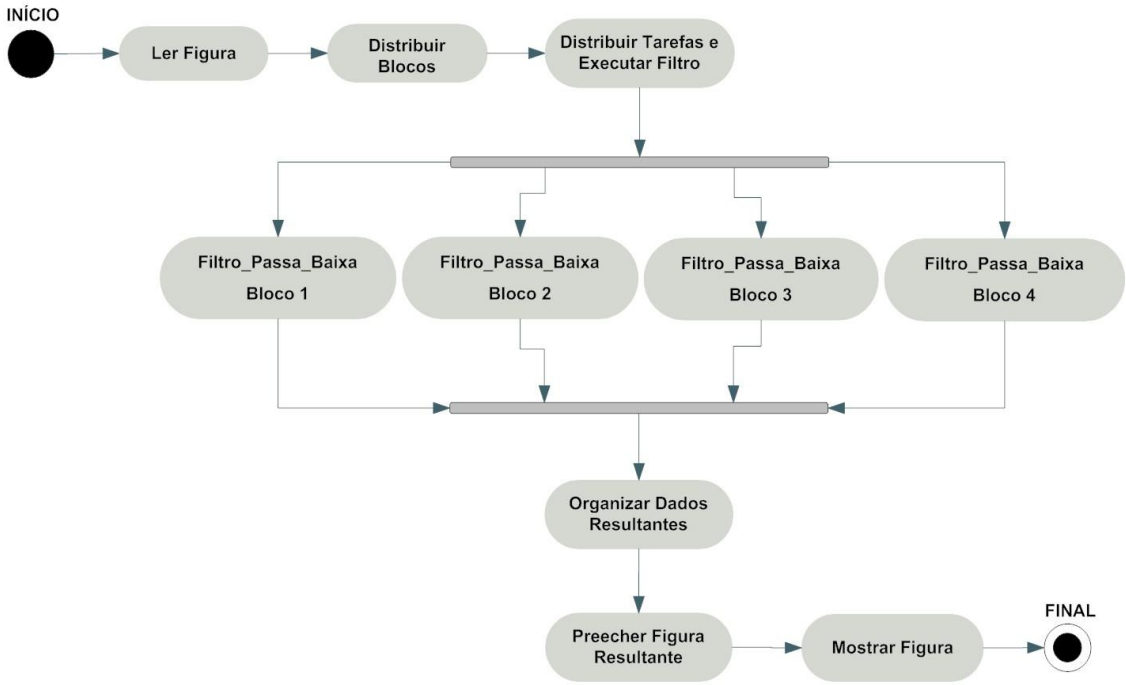

Figura 4.5 Diagrama de atividades da aplicação Remoção de Ruídos.

No diagrama de atividades da Figura 4.5, após a divisão da imagem em blocos é realizada a execução do filtro em paralelo, neste caso em quatro blocos. Posteriormente, as informações resultantes das execuções em paralelo são retornadas e as próximas atividades continuam de forma sequencial até a conclusão da ação no sistema.

O próximo diagrama a ser construído nesta etapa é o diagrama de sequência. Este diagrama oferece informações essenciais sobre os relacionamentos entre as classes, métodos e atributos das classes e o comportamento dinâmico dos objetos. Sua principal funcionalidade está em mostrar a comunicação entre os objetos ao longo de uma linha de tempo [34].

A notação utilizada na descrição do diagramas de sequência envolve a indicação do conjunto de objetos envolvidos em um cenário e a definição das mensagens trocadas entre estes objetos ao longo do tempo. Os objetos são colocados em linha na parte superior do diagrama. Linhas verticais tracejadas são traçadas da base dos objetos até a parte inferior do diagrama representando a linha de tempo. Desta forma, o ponto superior destas linhas indica um instante inicial e, à medida que se avança para baixo evolui-se o tempo.

Além da notação descrita acima, uma das mais importantes funcionalidades encontradas nos diagramas de sequência é a troca de mensagem. Esta primitiva é utilizada para indicar os momentos de interação ou comunicação entre os objetos. Utiliza-se como notação para trocas de mensagens segmentos de retas direcionados da linha de tempo de um objeto origem para a linha de tempo de um objeto destino [34]. O diagrama de sequência do exemplo de remoção

de ruídos ilustrado na Figura 4.6, apresenta as classes que estão sendo executadas em paralelo, as chamadas aos métodos e as mensagens trocadas entre essas classes. Neste diagrama o paralelismo é expresso com várias instâncias do mesmo objeto que será executada em paralelo.

De forma semelhante ao diagrama de atividades, o diagrama de sequência expressa os canais de informação e os elementos de processamento ou processadores onde serão executadas as ações em paralelo. Com a utilização do diagrama de atividades contempla-se a fase da definição da comunicação prevista na metodologia PCAM, onde o padrão de comunicação e o formato das mensagens são definidos.

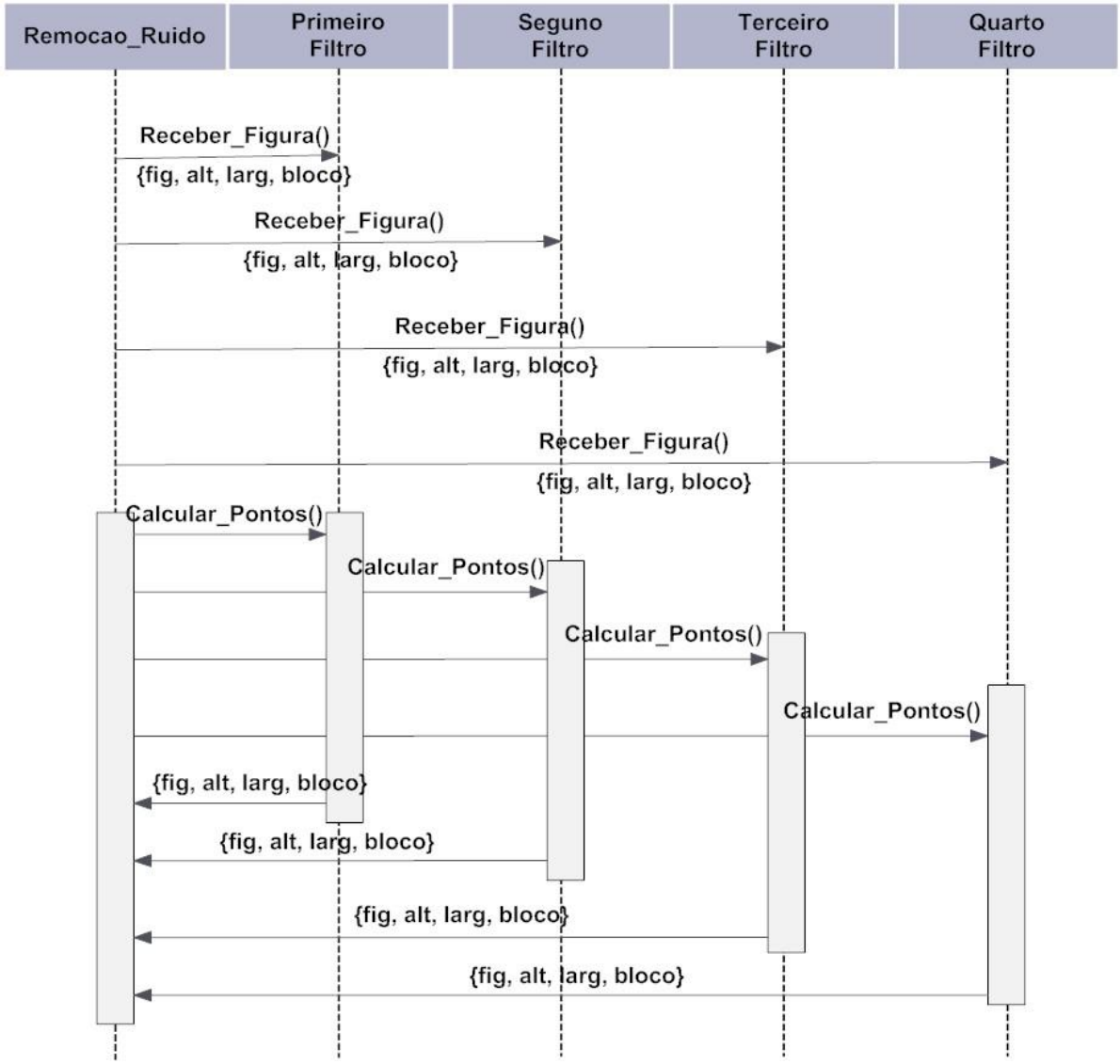

Figura 4.6 Diagrama de sequência da aplicação Remoção de Ruídos.

Outra questão importante que deve ser definida nesta etapa de projeto está relacionada ao mapeamento dos objetos nas máquinas de processamento do sistema, ou seja, especificar em

qual processador cada módulo ou parte do código será executado, também previsto na metodologia PCAM. Partindo dos requisitos de ambiente coletados na etapa de requisitos, como a quantidade de processadores e arquitetura utilizada, deve-se procurar encontrar um equilíbrio de carga para cada processador levando em consideração questões de balanceamento de carga inseridas no contexto de sistemas paralelos.

Neste sentido, o diagrama de distribuição (*deployment*) deve ser confeccionando para finalizar esta etapa da metodologia. Este diagrama ilustra os elementos de hardware ou nós do sistema e a troca de mensagens entre eles, além de onde os objetos são mapeados fisicamente. O diagrama de distribuição da aplicação de remoção de ruídos é ilustrado na Figura 4.7.

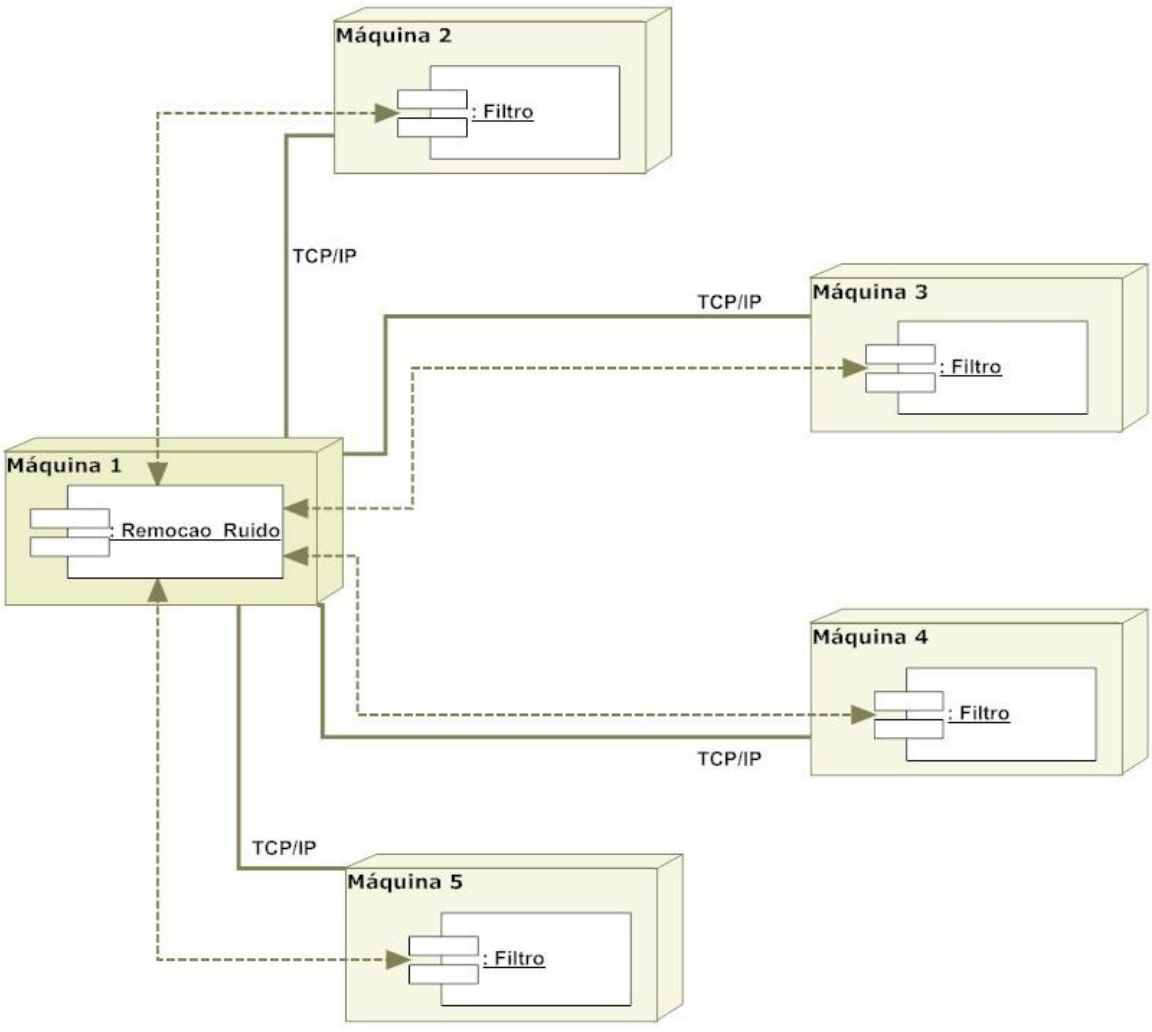

Figura 4.7 Diagrama de Distribuição da aplicação Remoção de Ruídos.

Após a confecção dos diagramas de classes, atividades, sequência e distribuição, conclui-se a fase de projeto da metodologia MOODAP. Considerando a metodologia PCAM, pode-se concluir que as fases de P – Particionamento, C – Comunicação e M – Mapeamento foram definidas com os diagramas utilizados. Porém, a fase A – Agrupamento que define possíveis

agrupamentos de tarefas que possuem muita comunicação entre elas, onde pode ser necessário agrupar tarefas em um único processador, requer uma análise mais eficiente sobre a comunicação entre as tarefas e nós do sistema. Isto deve reavaliado após a etapa de análise de desempenho, ou seja, após um analise numérica da qualidade da aplicação implementada.

4.3 Implementação

A fase de implementação inicia-se tendo por base os diagramas especificados na fase de análise e projeto. A partir destes diagramas, o desenvolvedor deve codificá-lo em uma linguagem orientada a objetos, ou até mesmo uma linguagem sequencial com C e Fortran, utilizando-se de artifícios, tais como bibliotecas de comunicação.

Abaixo serão descritas as principais ferramentas utilizadas na comunicação em aplicações paralelas.

4.3.1 MPI (*Message Passing Interface*)

O MPI é um padrão de interface para troca de mensagens em máquinas paralelas com memórias distribuídas. Esse padrão foi desenvolvido procurando fornecer uma base comum de desenvolvimento de programas paralelos em plataformas distintas. Assim, usando esta biblioteca o desenvolvedor consegue escrever um programa que rode em várias máquinas e essas não precisam ter necessariamente a mesma arquitetura.

O padrão MPI define a sintaxe e a semântica para 125 funções, divididas em comunicações ponto a ponto entre dois processos, operações coletivas entre processos, entrada e saída paralela (*parallel I/O*) e gerenciamento de processos para as linguagens de programação C, C++, Fortran 77 e Fortran 90 [39,20].

Dentre estas funções, destaca-se um conjunto de seis sub-rotinas que são suficientes para desenvolver a maioria das aplicações em MPI. A seguir uma breve descrição destas sub-rotinas em linguagem C [41].

Inicializar um processo MPI (MPI_INIT)

Esta rotina estabelece o ambiente de execução do MPI sincronizando todos os processos de inicialização de uma aplicação MPI. Esta deve ser a primeira rotina a ser chamada por cada processo.

```
int MPI_Init (int *argc, char ***argv)
```

Identificar um processo MPI (MPI_COMM_RANK)

Dentre um grupo de processos, esta primitiva identifica qual o número do processo. Este número deve ser identificado por um valor inteiro dentre um intervalo de 0 e N-1 processos.

```
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

Contador de processos MPI (MPI_COMM_SIZE)

Retorna o número de processos dentre o conjunto de processos.

```
int MPI_Comm_size (MPI_Comm comm, int *size)
```

Enviar mensagens no MPI (MPI_SEND)

A função MPI_SEND é utilizada basicamente para o envio de mensagens. As mensagens enviadas são compostas por duas partes: o dado, o qual deseja-se enviar, e o envelope com informações da rota dos dados. A mensagem também poderá conter uma variável inteira de retorno contendo o status da rotina.

```
int MPI_Send (void *sndbuf, int count,MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
```

Receber mensagens no MPI (MPI_RECV)

Muito semelhante ao MPI_SEND, porém, esta é utilizada para o recebimento de mensagens.

```
int MPI_Recv (void *recvbuf, int count,MPI_Datatype
datatype, int source, int tag, *status,MPI_Comm comm)
```

Finalizar processos MPI (MPI_FINALIZE)

Esta primitiva finaliza um processo para o MPI. Esta será a ultima rotina a ser executada por uma aplicação MPI onde sincroniza todos os processos na finalização. Segue a sintaxe desta primitiva.

```
int MPI_Finalize()
```

4.3.2 PVM (Parallel Virtual Machine)

O PVM é um conjunto integrado de ferramentas de *software* e bibliotecas que permitem o funcionamento de um grupo de computadores interconectados, sendo estes, mono, multiprocessados e/ou paralelos, como sendo um único computador paralelo [39].

O funcionamento básico do PVM pode ser dividido em duas partes distintas. A primeira parte é um *daemon,* chamado de pvmd3, que é executado em todos os nós no sistema. Este programa é responsável por todas as trocas de mensagens e coordenação das tarefas em execução.

A segunda parte é uma biblioteca de rotinas de interface. Nesta parte se encontram as primitivas necessárias para a interação entre as tarefas de uma aplicação. Estão contidas nesta biblioteca as rotinas responsáveis pela coordenação das tarefas, comunicação entre os computadores interligados, gerência de processos, além da verificação e manutenção do estado da máquina virtual. Para programação, essas bibliotecas são distribuídas em linguagens como Java, Python, Perl, C, C++ e Fortran [39].

A seguir são mostradas algumas primitivas básicas utilizadas na programação PVM [42]. Os exemplos abaixo estão na linguagem C.

Identificar um processo PVM (pvm_mytid)

Esta primitiva retorna o identificador pelo qual o processo requisitante é conhecido na máquina virtual. Esta função normalmente é a primeira função PVM chamada no programa e tem as funções de registrar o processo e informar ao processo o seu identificador na máquina virtual.

```
  printf("i'm t%x\n", pvm_mytid());
```

Disparar nova tarefa (pvm_spawn)

É usado para iniciar novas tarefas na máquina virtual. Em um modelo de programação mestre/escravo, esta primitiva é utilizada no programa mestre para lançar processos escravos para as máquinas pertencentes da máquina virtual.

```
  int cc = pvm_spawn("hello_other", (char**)0, 0, "", 1,
&tid);
```

Finalizar processo PVM (pvm_exit)

É a última função a ser chamada por um programa PVM. Esta função informa que processo está se desligando da máquina virtual. O PVM exige que um programa comece com pvm_mytid() e termine em pvm_exit().

```
  pvm_exit();
```

Enviar dados (pvm_send)

Esta primitiva envia os dados armazenados em um buffer de envio para uma dada tarefa. Para utilizar esta função deve-se utilizar anteriormente outras primitivas de gerenciamento de *buffer* descritos na literatura.

```
pvm_send(ptid, msgtag);
```

Receber dados (pvm_recv)

Esta função recebe uma mensagem enviada por outro processo e a coloca no *buffer* de recebimento. Caso não existam mensagens a ser recebidas, o processo fica bloqueado até que uma mensagem se faça disponível para recepção.

```
pvm_recv(ptid, msgtag);
```

4.3.3 Pthreads

Pthreads ou Posix Threads é um padrão que especifica uma API (*application programming interface*) para a escrita de aplicações *multithreaded*. Este padrão definido pela IEEE (*Institute of Electrical and Electronics Engineers)* e pela ISO (*International Organization for Standartization*) teve como objetivo suprir a falta de portabilidade de aplicações que utilizavam múltiplas threads [43].

Esta API manipula threads, que conceitualmente são múltiplos caminhos de execução que executam concorrentemente em uma memória compartilhada e que compartilham os mesmos recursos de um processo pai. Uma thread pode ser definida como uma processo simplificado, pouco custoso ao sistema operacional, sendo fácil de criar, manter e gerenciar.

O padrão Pthreads relaciona mais de 60 primitivas para a manipulação de threads nas linguagens C e C++ e alguns compiladores de Fortran (IBM AIX Fortran) que fornecem uma API Pthreads para Fortran. A seguir são descritas algumas das principais funcionalidades disponíveis nesta API em linguagem C [43].

Criar uma thread (pthread_create)

Cria uma thread levando em conta os atributos definidos em thread_id_attribute, o identificador da thread no parâmetro thread_id e a função que será executada em

thread_function usando o argumento arg. O retorno da função pthread_create pode ser 0 indicando sucesso, ou diferente de 0 indicando uma falha.

```
pthread_create( &thread_id, thread_id_attribute,
(void*)&thread_function, (void *) &arg);
```

Recuperar resultado de uma thread (pthread_join)

Muitas vezes os resultados obtidos por cada thread precisam ser unidos para fazer processamento sequencial, ou simplesmente para recuperar os valores de retorno. Neste sentindo pode-se utilizar a função pthread_join.

```
pthread_join(thread id,&retval);
```

Encerrar uma thread (pthread_exit)

Apesar da thread ser automaticamente encerrada quando a função thread_function é concluída. Pode-se solicitar explicitamente o encerramento da thread com pthread_exit. Após isso, os recursos de alocação da thread são liberados.

```
void pthread_exit(void* status);
```

4.3.4 OpenMP

OpenMP é uma API para o modelo de programação paralela com memória compartilhada para arquiteturas de múltiplos processadores. É composto basicamente por diretivas de compilação, bibliotecas de execução e variáveis de ambiente.

Disponível para uso em compiladores C/C++ e Fortran, esta API tem o paralelismo baseado no uso de múltiplas threads. O seu funcionamento inicia-se como uma thread principal, que executa sequencialmente até a primeira definição de uma região paralela ser encontrada. Nesse momento há uma bifurcação (*fork*) da execução, onde o thread mestre cria um conjunto de threads paralelos. Após a execução dos comandos na região paralela, as threads são sincronizadas (*join*) e finalizadas, permanecendo apenas a thread principal [44].

A seguir são descritos alguns exemplos da utilização da API OpenMP na linguagem C.

Variáveis de ambiente

São parâmetros definidos no ambiente operacional antes da execução do programa. As variáveis que controlam a execução do código paralelo devem vir sempre em letras maiúsculas.

– `OMP_NUM_THREADS`: identifica o número de atividades que serão executadas em paralelo.

– `OMP_DYNAMIC`: indica se o número de atividades a serem executadas em paralelo deve ou não ser ajustado dinamicamente.

– `OMP_SCHEDULE`: define esquema de escalonamento das atividades paralelas

Bibliotecas de execução

São rotinas de API, que possibilitam uma grande variedade de funções em uma aplicação OpenMP. Para um compilador C/C++ é necessário especificar a inclusão do arquivo “omp.h” no código fonte.

– `omp_set_num_threads()`: define o número de threads.

– `int omp_get_num_threads()`: retorna o número de threads.

– `void omp_set_nested()`: ativa ou desativa o paralelismo recursivo.

– `int omp_get_num_procs ()`: retorna o número de processadores disponíveis para a execução do programa.

Diretivas de compilação

As diretivas consistem em uma linha de código com significado especial para o compilador. Nas linguagens C e C++ as diretivas OpenMP são identificadas pelo #pragma omp.

– `#pragma omp_parallel`: define uma região paralelizável sobre um bloco estruturado de código. Esta é a diretiva fundamental do OpenMP.

– `#pragma omp sections`: especifica seções paralelas, onde cada seção será executada em um thread diferente.

– `#pragma omp parallel for`: determina uma região paralela e simultaneamente distribui as iterações do *loop* na região, por entre as threads de um grupo.

– `#pragma omp critical`: determina uma região do código que deve ser executada somente por uma thread de cada vez. Se uma thread estiver executando uma região crítica, as outras threads irão bloquear a execução quando alcançarem essa região.

4.3.5 Java RMI

O RMI (*Remote Method Invocation*) é uma tecnologia disponível para a plataforma Java que possibilita a chamada de objetos remotos da mesma maneira que objetos locais. O RMI

permite a comunicação de objetos executando em máquinas diferentes, independente da máquina virtual esta ou não na mesma máquina física.

Está técnica funciona como uma aplicação cliente/servidor, na qual o lado cliente submete uma tarefa que será executada por um objeto localizado no lado servidor. Após isso, o objeto executa a tarefa e devolve o resultado da execução para o cliente que requisitou a tarefa [47].

O Java RMI é composto por um conjunto de API's que inclui todas as classes necessárias à invocação de métodos remotos, um conjunto de ferramentas para executar o serviço de nomes e a geração de objetos *proxy*. Todos estes recursos são fornecidos com a distribuição padrão do kit de desenvolvimento do Java.

A API do Java RMI provê as classes utilizadas pelos clientes, servidores e pelo serviço de nomes. Abaixo são descritas alguns recursos disponíveis por esta API [48].

– A classe `RemoteObject`, do pacote `java.rmi.server`, é a superclasse dos objetos remotos, da qual derivam as principais classes do Java RMI.

– A classe `RemoteServer`, do pacote `java.rmi.server`, é uma classe abstrata que serve de base para as classes dos objetos que serão servidos remotamente. Desta classe deriva a classe `UnicastRemoteObject`, que implementa a comunicação RMI através de soquetes TCP/IP e faz com que os objetos continuem em execução contínua a espera de chamadas de clientes remotos.

– No pacote `java.rmi`, tem-se a interface `Remote`, de onde derivam-se todas as interfaces remotas. Esta interface é uma interface de marcação, servindo apenas para indicar que uma determinada interface é remota.

– A classe `Naming` é responsável pelo de serviço de nomes do Java RMI, do pacote `java.rmi`. Esta classe fornece os métodos para a consulta, inclusão e remoção de objetos no registro de nomes.

4.3.6 Comentários Gerais

Além das ferramentas citadas nas seções anteriores, existe uma série de outras ferramentas para o mesmo propósito encontradas na literatura, como as bibliotecas HPF, PESSL e OSLP. Porém, menos utilizadas que as descritas anteriormente ou mais específicas para determinado tipo de aplicação.

É importante destacar que ao final desta fase, a equipe de desenvolvimento consiga executar a aplicação utilizando-se de uma destas ferramentas em uma linguagem de

programação que suporte a ferramenta. Após a implementação, a fase de testes de *software* deve ser iniciada.

4.4 Testes de Software

A fase de testes de *software* tem como objetivo identificar erros em uma aplicação que muitas vezes passam despercebidos pelos desenvolvedores. Assim como acontece na programação sequencial, aplicações paralelas também necessitam de técnicas e ferramentas que auxiliem e suportem todas as características e problemas deste tipo de ambiente de programação. Desta forma, a fase de testes descrita neste trabalho terá o foco direcionado nas diferenças encontradas na aplicação de testes em uma aplicação sequencial em relação a uma aplicação paralela.

Um programa paralelo pode ser definido basicamente como uma coleção de diversos processos sequenciais que executam simultaneamente e que se comunicam. Desta forma, dificuldades básicas encontradas em um depurador sequencial, consequentemente são encontradas em qualquer depurador paralelo. Entretanto, aplicações paralelas apresentam outras questões relacionadas, tais como, ocorrência de vários processos simultâneos caracterizando um comportamento não-determinístico[7] e dificuldades relacionadas à concorrência e ao sincronismo entre os processos [45].

Dois problemas principais são causados pelo não-determinismo, o *irreproducibility effect* e o *completeness problem* [46]. O primeiro está relacionado a não capacidade de reprodução de um erro encontrado, já que o programa está sujeito a gerar a cada execução subsequente, como sequências diferentes de eventos sincronizantes ou diferentes caminhos de execução. Já o *completeness problem* está relacionado aos critérios de cobertura de teste. Este problema é encontrado quando se deseja testar o comportamento de um programa paralelo em todos os caminhos possíveis de execução, o que pode se tornar impraticável dependendo do número de caminhos a serem testados.

Uma solução para o *irreproducibility effect* é a criação de um mecanismo que crie re-execuções equivalentes a uma execução anterior "observada", mecanismo conhecido como *record&replay*. Neste mecanismo a ordem das mensagens é armazenada em um histórico de execução e posteriormente são utilizados para criar re-execuções equivalentes à execução "observada" [46].

[7] Não-determinístico: comportamento em que um evento tem mais de um caminho (estado) para percorrer dentro de uma aplicação.

Para o *completeness problem*, a maioria das soluções consiste em coletar informações referentes aos eventos de comunicação sincronizantes em uma execução inicial da aplicação, e assim identificar todas as condições de corrida existentes. Com isso, em uma segunda etapa, são criadas re-execuções artificiais através da manipulação das ordens dos eventos, permitindo que a aplicação tenha outros comportamentos. Técnicas como *event manipulation* e *artificial replay* conseguem investigar diferentes execuções para um mesmo conjunto de entradas possibilitando uma cobertura maior dos testes em aplicações paralelas.

Há uma grande quantidade de desafios relacionados à depuração de programas paralelos, como os problemas encontrados em ambientes onde a comunicação é baseada na troca de mensagens, tais como, MPI e PVM. Nesse tipo de sistema paralelo, um processo que termina por causa de um erro inesperado, pode propagar o erro pelo resto dos processos em execução, já que pode haver dependências de comunicação existentes entre estes processos, resultando em uma falha no sistema.

Várias ferramentas vêm sendo desenvolvidas visando uma melhor qualidade dos testes realizados em aplicações paralelas. Para o padrão MPI destaca-se a MPI-PreDebugger, uma ferramenta de depuração para localizar falhas em processos. Esta ferramenta distingue um erro original de código dos erros ocorridos com a comunicação durante a execução do programa. Se houver algum erro nessa comunicação, a execução do programa é interrompida e um arquivo de *log* é gerado descrevendo o comportamento dos processos durante a execução do programa [46].

As ferramentas de testes de software devem ser estudadas visando o mecanismo de troca de mensagens ou a biblioteca utilizada na fase de implementação. Esta tarefa pode ser complicada pelo fato de que ainda muitos estudos estão sendo realizados nesta área e poucas ferramentas estão disponíveis para depuração e testes em ambientes paralelos e distribuídos.

Ao finalizar a etapa de testes de *software*, o desempenho da aplicação deve ser analisado e possíveis otimizações deveram ser realizadas nas etapas anteriores.

4.5 Análise de Desempenho

O processamento paralelo visa obter um melhor desempenho em relação a uma aplicação sequencial. Isto se deve a possibilidade de tarefas serem executadas simultaneamente em vários processadores. Porém, às vezes o desempenho de uma aplicação paralela pode não ser satisfatório. Desta forma, a necessidade da utilização de métricas que determinem quantitativamente o desempenho alcançado.

Destaca-se entre as métricas de desempenho, o tempo de execução, o *speedup* e a eficiência [27].

4.5.1 Tempo de execução

O tempo de execução é o resultado da soma de três componentes de uma aplicação. O primeiro componente é o tempo de computação (T_{comp}). Este é responsável pelo tempo gasto em efetuar cálculos sem contar com o tempo de comunicação (T_{comn}) e tempo ocioso (T_{ocioso}). O segundo componente está relacionado a comunicação. Esta medida retrata o tempo gasto ao enviar e receber as mensagens pelas tarefas da aplicação. Por último tem-se o tempo ocioso, que ocorre quando o processador fica sem tarefas. Este componente pode ser minimizado com uma distribuição de carga adequada ou sobrepondo a computação com a comunicação. A fórmula para o cálculo do tempo de execução é dada por:

$$T_{exec} = T_{comp} + T_{comn} + T_{ocioso}$$

4.5.2 Speedup

Outra métrica importante é o *speedup* (ganho). Este representa o valor observado quando se executa uma determinada tarefa em um processador em relação a vários processadores. A fórmula para o cálculo é descrita a seguir.

$$Speedup = T1 / Tp$$

Onde $T1$ representa o tempo de execução em um processador e Tp representa o tempo de execução em p processadores.

Idealmente, o ganho deveria tender a p, porém, fatores como o tempo necessário para a comunicação entre tarefas, partes do código que são dificilmente paralelizáveis e granularidade ineficaz na aplicação invalidam esta premissa.

4.5.3 Eficiência

A medida de eficiência especifica a porcentagem de tempo que os processadores gastam realizando trabalho útil. Desta forma, tem-se uma medida mais conveniente da qualidade de um algoritmo paralelo. Esta métrica caracteriza o uso efetivo pelo algoritmo dos recursos de um computador paralelo de uma maneira independente do tamanho do problema. A fórmula para o cálculo da eficiência é descrita a seguir.

$$Ep = Sp/p$$

A eficiência é definida pela relação entre o *speedup* (Sp) e o número de processadores (p). O valor resultante deve variar de 0 a 1, o que representa a eficiência de 0% a 100%.

A partir dos dados coletados nesta fase, a equipe de desenvolvimento pode necessitar de uma readequação da comunicação da aplicação paralela desenvolvida. Pode ser conveniente agrupar tarefas que possuem elevada comunicação entre elas e que possivelmente estão mapeadas em máquinas distintas. A etapa A (agrupamento) descrita na PCAM deve ser realizada ao fim da etapa de análise de desempenho que consequentemente iniciará uma nova iteração nas etapas de projeto e implementação novamente.

4.6 Considerações Gerais

Neste capítulo foi descrito a metodologia MOODAP com o propósito de guiar o desenvolvedor em processo de desenvolvimento de uma aplicação paralela. Desde as etapas iniciais até a fase de análise de desempenho, procurou-se expor algumas das principais técnicas e mecanismos mais utilizados no contexto de aplacações paralelas, além da utilização de alguns diagramas UML relevantes nas fases de análise de requisitos e projeto. Obviamente que várias outras técnicas não foram mencionadas neste trabalho e podem ser estudadas e utilizadas em trabalhos futuros.

Ao fim das fases propostas nesta metodologia, o desenvolvedor deve possuir uma aplicação paralela executando em uma estrutura multiprocessada. A partir da conclusão da fase de análise de desempenho, o processo de desenvolvimento passa a ser iterativo se houver necessidade de uma otimização da aplicação em termos de estratégias de comunicação entre processos.

Para exemplificar a utilização da metodologia MOODAP, no próximo capítulo será realizado um estudo de caso no sentido de validar as fases do desenvolvimento em uma aplicação prática.

Capítulo 5

Estudo de Caso - Método de Solução de Sistemas de Equações Paralelo

O estudo de sistemas de equações é de grande importância, pois estes resultam de modelos discretos provenientes de vários tipos de aplicação, como programação linear, dinâmica dos fluídos, modelagem do clima e previsão meteorológica [51].

Neste estudo de caso apresenta-se um método de solução paralelo de sistemas de equações utilizado no modelo HIDRA [49]. Como entrada de dados tem-se um sistema de equações e como saída espera-se o resultado deste sistema, como mostrado na Figura 5.1.

Figura 5.1 Entrada e saída de dados para o método de solução.

O HIDRA é um modelo computacional paralelo para a simulação do escoamento e do transporte de substâncias, tridimensional e bidimensional, em corpos de água [49]. Um dos componentes importantes neste modelo são os métodos de solução de sistemas de equações.

Para o estudo de caso empregando a metodologia MOODAP será abordado o método de solução conhecido como Métodos de Decomposição de Domínios (MDD). Um MDD é caracterizado pela divisão do domínio computacional, que é particionado em subdomínios empregando algoritmos de particionamento. A solução global do problema é, então, obtida através da combinação dos subproblemas que são resolvidos localmente. Cada processador é responsável por encontrar a solução local de um ou mais subdomínios, que a ele são alocados e, então, essas soluções locais são combinadas para fornecer uma aproximação para a solução global [50].

Nas próximas seções, apresenta-se o uso da metodologia MOODAP no desenvolvimento desta aplicação.

5.1 Especificação de Requisitos

Requisitos Funcionais

Como definido na metodologia MOODAP, o processo de desenvolvimento inicia-se com a especificação de requisitos. Desta forma, são definidas as informações sobre cada caso de uso do estudo de caso realizado, segundo o *template* proposto por Cockburn [35]. É importante ressaltar que algumas informações deste *template* não serão descritas, pois dependendo da aplicação, não há necessidade de um grau de informação elevado.

Caso de Uso*: 1 – Executar Método de Solução*

INFORMAÇÃO CARACTERÍSTICA

Objetivo no Contexto: *executa o método paralelo para a resolução do sistema de equações.*

Escopo: *HIDRA – Modelo computacional paralelo com balanceamento dinâmico de carga para a simulação do escoamento e do transporte de substâncias, tridimensional e bidimensional, em corpos de água. Este módulo é o responsável pela resolução dos sistemas de equações gerados na discretização das equações do HIDRA.*

Pré-condições: *sistemas de equações previamente estabelecidos.*

Condição Final de Sucesso: *sistema de equações resolvido.*

Condição Final de Falha: *sistema de equações não resolvido; aborta-se a execução.*

Ator Primário: HIDRA

CENÁRIO PRINCIPAL DE SUCESSO

1. *Hidra executa o módulo de resolução de sistemas*
2. *Leitura dos arquivos contendo o sistema de equações*
   2.1<include> Manipular Arquivos
3. *Leitura dos arquivos contendo as informações de trocas de dados*
4. *Resolução dos sistemas de equações*
   4.1 <include> Resolver Sistemas de Equações
5. *Trocas de dados nas regiões de borda para manter a continuidade da solução*
   5.1 <include> Trocas Dados de Fronteira
6. *Repete-se os passos 3 e 4 até que se atinja a precisão desejada*
7. *Grava a solução em arquivo*

EXTENSÕES

1a. Hidra chama o módulo passando como parâmetro os arquivos a serem utilizados
1b. Processo mestre lê os parâmetros informados pelo usuário

2a.Processo mestre chama o Manipulador de arquivos com os parâmetros dos arquivos de entrada dos sistemas de equações
3a. Processo mestre chama o Manipulador de arquivos com os parâmetros dos arquivos de entrada troca de mensagens
4a. Processos Escravos resolvem os sistemas de equações
5a. Processos Escravos trocam dados para manter a continuidade da solução
7a. Processos Escravos enviam a resposta para nó mestre que gera os arquivos de saída
---

INFORMAÇÃO RELACIONADA

Prioridade: *alta*

Frequência: *ocorre todas as vezes que a solução de um sistema for necessária. Pode ocorrer a cada passo de tempo de uma simulação que emprega sistemas de equações*

Casos de Uso Subordinados: *Manipular arquivos; Resolver Sistemas através do Gradiente Conjugado; Trocar dados*

Quadro 5.1 Caso de uso Executar método de Solução

Caso de Uso*: 2 – Manipular Arquivos*

INFORMAÇÃO CARACTERÍSTICA

Objetivo no Contexto*: manipula arquivos de entrada e saída. Faz a leitura dos arquivos de entrada e a escrita dos arquivos contento a solução (Saída).*

Escopo: *Executar Método de Solução.*

Pré-condições: *chamada pelo caso de uso **Executar Método de Solução**; recebimento dos dados para escrita; recebimento dos parâmetros dos arquivos de entrada.*

Condição Final de Sucesso: *arquivo lido/escrito e dados enviados (no caso de leitura) para o caso de uso **Executar Método de Solução**.*

Condição Final de Falha: *erro de arquivo; aborta-se a execução.*

Ator Primário: HIDRA

CENÁRIO PRINCIPAL DE SUCESSO

Em caso de leitura
1. *Recebe o nome do arquivo a ser lido*
2. *Lê os dados do arquivo*

Em caso de escrita
1. *Recebe o nome do arquivo de saída*
2. *Recebe os dados a serem gravados*
3. *Escreve os dados no arquivo*

EXTENSÕES

Leitura

1a. Processo mestre lê os parâmetros informados

1b. Processo mestre verifica se os arquivos passados por parâmetros existem;

Se arquivos não existirem, a aplicação é finalizada com uma mensagem de erro

2a. Os dados lidos são divididos entre os processos escravos

2.b Processos escravos recebem os dados

Escrita

1a. Processo mestre lê os parâmetros informados

2a. Processos escravos enviam o conteúdo a ser gravado

2b. Processo mestre recebe conteúdo a ser gravado

3a. Processo mestre cria o arquivo de saída

3b. Processo mestre escreve o conteúdo no arquivo

INFORMAÇÃO RELACIONADA (opcional)

Prioridade: *alta*

Frequência: *ocorre todas as vezes que forem necessárias a manipulação de arquivos*

Quadro 5.2 Caso de uso Manipular Arquivos

Caso de Uso*: 3 – Trocar dados de fronteira*

INFORMAÇÃO CARACTERÍSTICA

Objetivo no Contexto*: Fazer a troca de dados entre os processos de modo a manter a continuidade da solução paralela.*

Escopo: *Executar Método de Solução.*

Pré-condições: *Chamada pelo caso de uso* ***Executar Método de Solução****; recebimento das informações sobre os dados que serão trocados.*

Condição Final de Sucesso: *Informações enviadas.*

Condição Final de Falha: *Erro na transmissão das informações. Execução abortada.*

Ator Primário: HIDRA

CENÁRIO PRINCIPAL DE SUCESSO

1. *Envia e recebe os dados para os respectivos processos*
2. *Altera o vetor de termos independentes utilizando os dados recebidos*

EXTENSÕES

1a. Processos escravos verificam lista de envio e recebimento de informações (recebido em 2a. de Manipular Arquivos)

1b. Processos escravos trocam informações com os processos relacionados na lista

2a.Processos escravos modificam os vetores dos termos independentes utilizando os dados recebidos de outros processos

INFORMAÇÃO RELACIONADA (opcional)

Prioridade: *alta*

Frequência: *ocorre todas as vezes que forem necessárias a troca de informações entre os processos envolvidos na resolução do sistema*

Quadro 5.3 Caso de uso Trocar Dados

Caso de Uso*: 4 – Resolver Sistemas de equações*

INFORMAÇÃO CARACTERÍSTICA

Objetivo no Contexto*: Resolver sequencialmente os sistemas de equações utilizando o método iterativo do Gradiente Conjugado.*

Escopo: *Executar Método de Solução.*

Pré-condições: *Chamada pelo caso de uso* ***Executar Método de Solução****; recebimento dos sistemas a serem resolvidos.*

Condição Final de Sucesso: *Sistemas resolvidos com sucesso; envio da solução para escrita em arquivo.*

Condição Final de Falha: *Erro na solução/solução não existente; aborta-se a execução.*

Ator Primário: HIDRA

CENÁRIO PRINCIPAL DE SUCESSO

1. *Resolve o sistema de equações*

EXTENSÕES

1a. Processos escravos resolvem suas respectivas partes do sistema de equações

INFORMAÇÃO RELACIONADA (opcional)

Prioridade: *alta* Frequência: *ocorre todas as vezes que forem necessárias a resolução de um sistema de equações*

Quadro 5.4 Caso de uso Resolver Sistema através do Gradiente Conjugado

Para facilitar o entendimento das funcionalidades do sistema, a Figura 5.2 ilustra o diagrama de casos de uso.

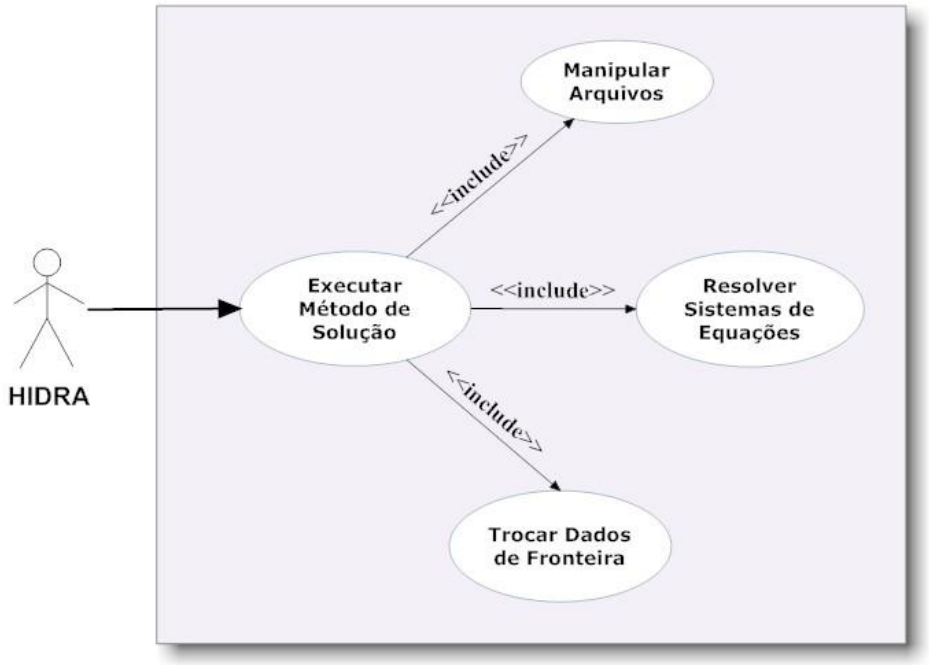

Figura 5.2 Diagrama de Casos de uso para o estudo de caso.

Requisitos Não Funcionais

Nesta aplicação, apenas os requisitos não funcionais de confiabilidade e desempenho foram considerados, isto se deve ao tipo de aplicação desenvolvida neste estudo. Caso fosse desenvolvida uma aplicação comercial, critérios como segurança, usabilidade e questões econômicas teriam que ser descritas de acordo com a necessidade do cliente e levadas em consideração nas etapas seguintes.

Confiabilidade: neste estudo de caso, a confiabilidade está associada à qualidade da solução numérica encontrada, ou seja, quão correta é a solução encontrada para um determinado sistema de equações.

Desempenho: espera-se que empregando o paralelismo obtenha-se ganhos de desempenho (*speedup*), de modo a reduzir o tempo de execução do sistema.

Requisitos de Ambiente

A seguir, são descritos os requisitos de ambiente da aplicação, estes que demonstram em qual estrutura física disponível para a execução da aplicação.

Requisito de ambiente	Descrição
Plataforma Computacional	Linux
Arquitetura (processador)	Pentium III 1.1 GHz
Arquitetura Paralela	Cluster de computadores (memória distribuída)
Quantidade de Processadores	20 nós de processamento com 2 processadores cada + 1 nó de controle (*front-end*) , totalizando 42 processadores
Linguagens e Bibliotecas	C/C++ e MPI
Capacidade de Memória	1 GB por nó
Capacidade de Disco	20 GB por nó
Rede	Gigabit Ethernet

Tabela 5.1 Requisitos de ambiente para o estudo de caso

5.2 Análise e Projeto de Software

Baseando-se nos requisitos coletados e definidos na etapa de especificação de requisitos, os diagramas UML de classes, atividades, sequência e distribuição são confeccionados nesta etapa da metodologia. Segue estes diagramas a seguir.

Diagrama de Classes

O diagrama de classes representado pela Figura 5.3 ilustra os atributos e métodos da aplicação para a resolução do sistema de equações. Na classe Solver Paralelo, o método

Ler_arquivo() manipula os arquivos de entrada. O método Obtem_Solucao() recebe a solução de cada nó de execução e o método Escreve_Solucao() escreve a solução final. Ainda nesta classe o método Verificar_Erro() testa se a solução encontrada possui um erro menor que o erro estimado.

A classe Solver Sequencial possui apenas dois métodos. Primeiro o método Receber_SE() carrega a matriz A e o vetor B para serem calculados com o método Calcular_Método(). Por último a classe Trocar Informações manipula o arquivo de trocas que são necessários para calcular a solução.

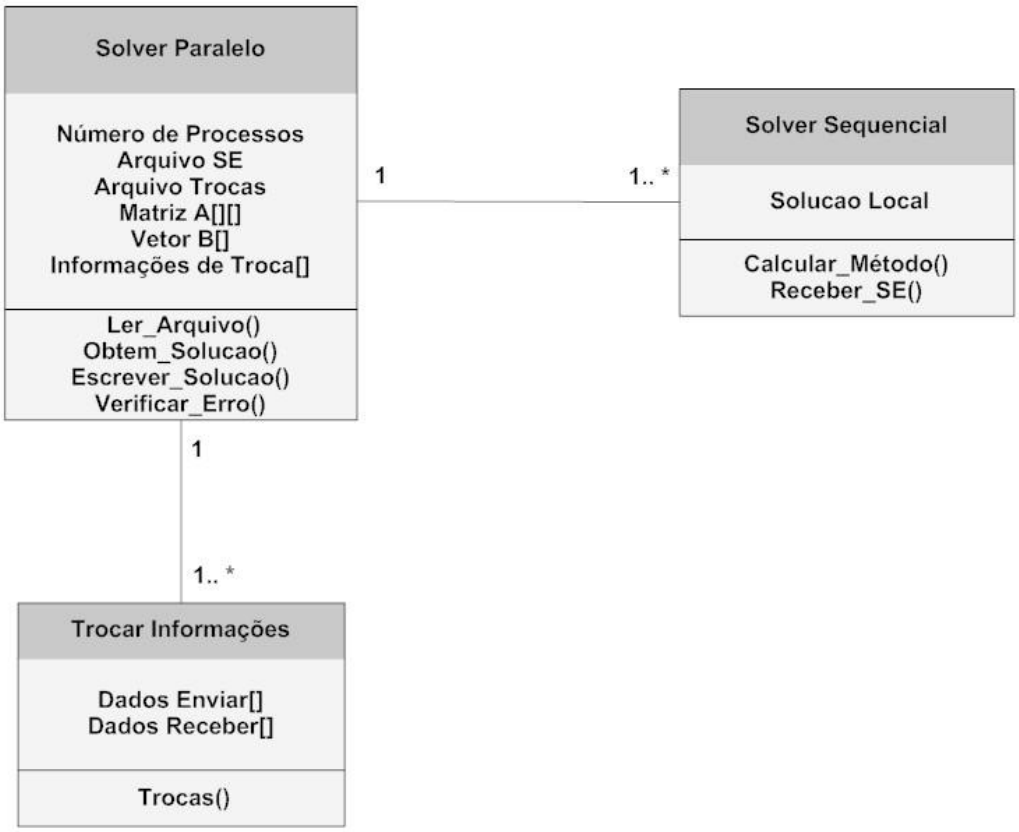

Figura 5.3 Diagrama de classes para o estudo de caso.

Diagrama de Atividades

O diagrama demonstra as atividades desta aplicação. O processo inicia-se com a leitura dos arquivos de entrada e informações de troca. Após estes passos, a execução da aplicação é subdivida em processos que ocorrem simultaneamente. Neste diagrama optou-se por ilustrar apenas 3 processos pelo fato de uma representação com mais processos ser inviável. Ao fim da execução paralela, o erro é calculado e se a condição for aceita a solução é escrita, caso contrário, novas iterações são executadas até a condição ser satisfeita.

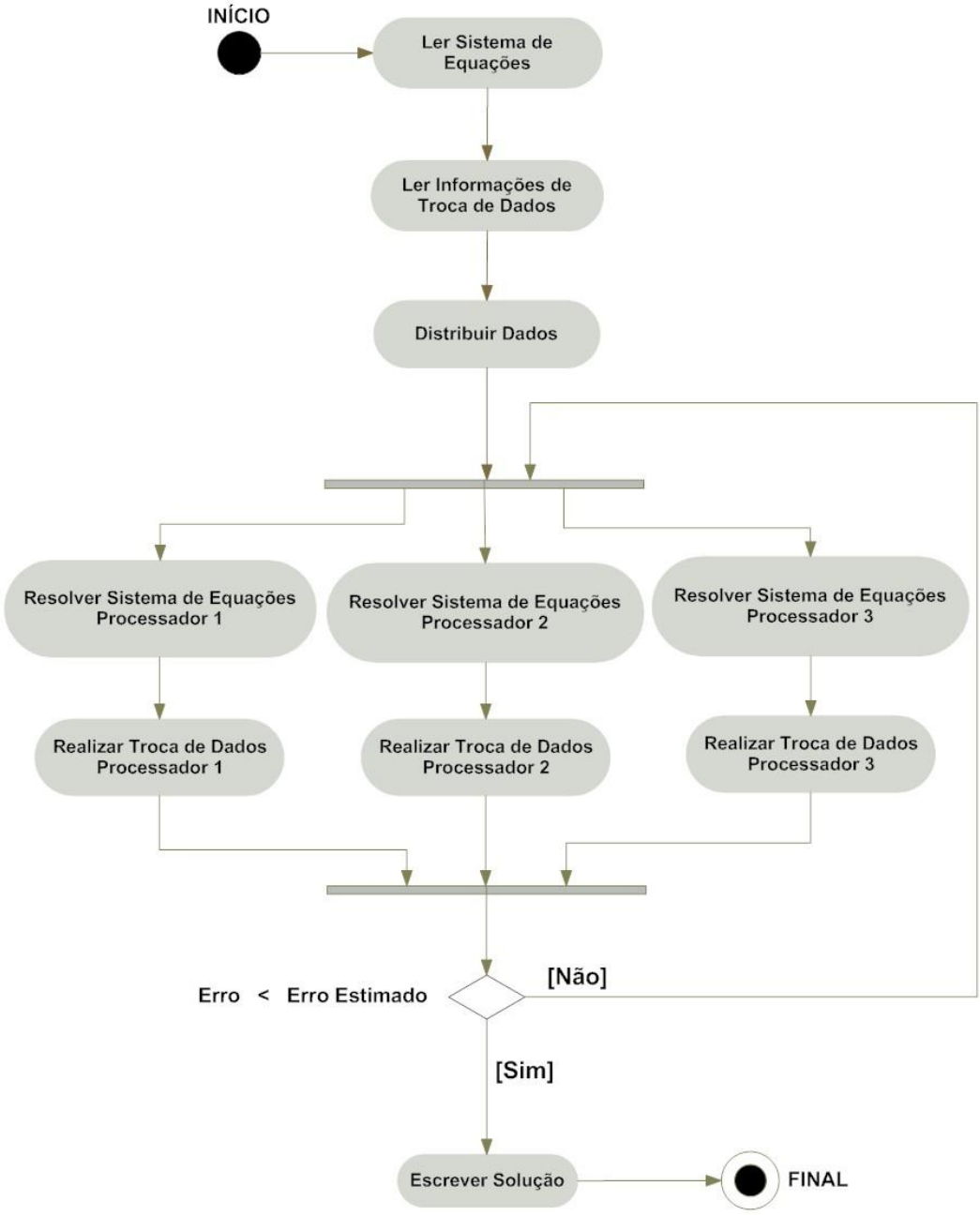

Figura 5.4 Diagrama de atividades para o estudo de caso.

Diagrama de Sequência

O diagrama ilustrado na Figura 5.5 demonstra as chamadas dos métodos e a comunicação entre os objetos do estudo de caso realizado. Muito semelhante as ações representadas na Figura 5.4, o sistema inicia com a chamada o subsistema Solver Paralelo, o objeto Solver Paralelo lê os arquivos de entrada e invoca os métodos Receber_SE() e Calcular() do objeto Solver Sequencial. O Solver Sequencial receber as soluções locais e testa a condição de erro.

Neste diagrama foram utilizados apenas dois processos paralelos para representar a execução simultânea.

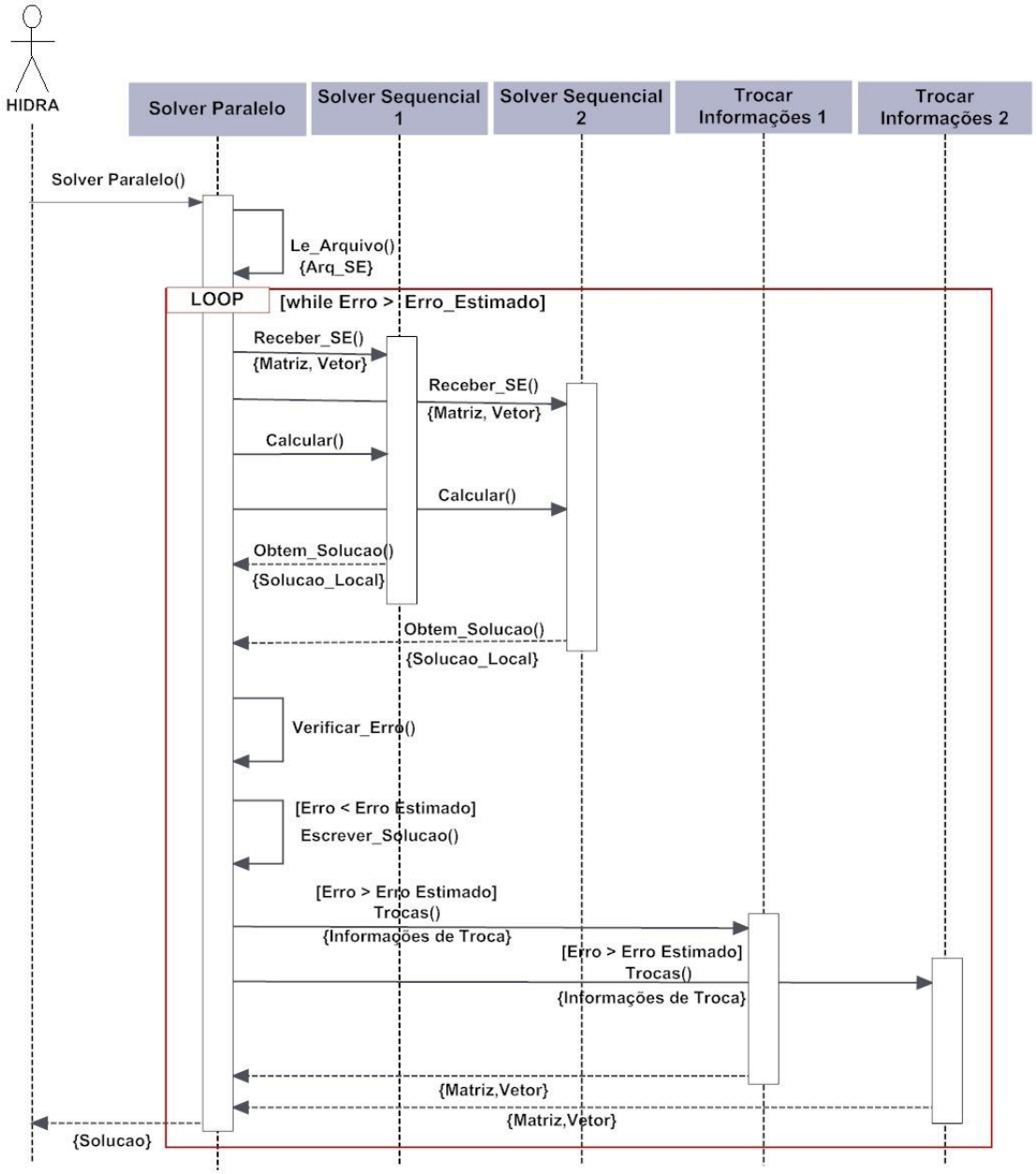

Figura 5.5 Diagrama de sequência para o estudo de caso.

Diagrama de Distribuição

Para este estudo de caso foram utilizados 21 nós de processamento, porém a representação de muitos nós em diagramas UML, como no diagrama de distribuição, pode ser considerada inviável. Para esta representação foi adaptado a notação (**...**) na vertical para suprir a necessidade de representar 21 nós de processamentos. Além disso, este diagrama demonstra onde os métodos são executados fisicamente.

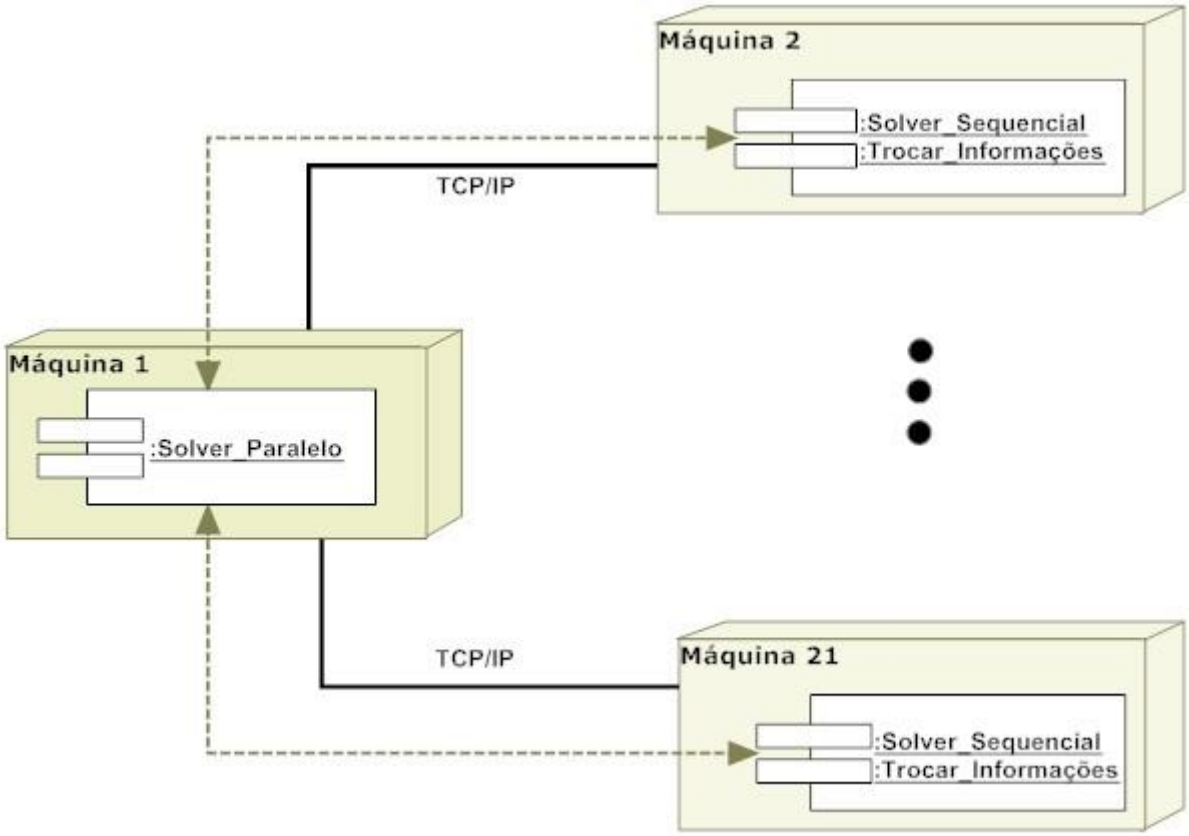

Figura 5.6 Diagrama de distribuição para o estudo de caso.

5.3 Implementação e Testes

O *solver* foi implementado em linguagem C/C++ utilizando as bibliotecas MPI e OpenMP para exploração de dois níveis de paralelismo. O MPI foi empregado na comunicação inter-processos, ou seja, é responsável pelas trocas de mensagens entre os nós do *cluster*. Já a biblioteca OpenMP foi utilizada para permitir o uso de *threads*, permitindo a exploração do paralelismo intra-nodal, aproveitando a característica multiprocessada das máquinas. A Figura 5.7 ilustra a utilização dos mecanismos de comunicação paralela neste estudo de caso.

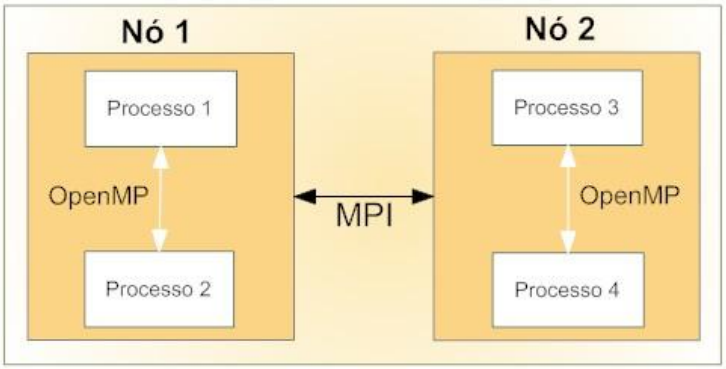

Figura 5.7 Níveis de paralelismo implementado no estudo de caso.

Para etapa de testes, apenas testes utilizando *benchmarking* foram efetuados, onde sistemas de equações com resultados já conhecidos foram resolvidos. O resultado obtido é comparado com o resultado conhecido para a validação da resposta. Esta decisão foi tomada em virtude de existirem poucas ferramentas disponíveis para testes em aplicações paralelas, e pelo fato de que ainda muitos estudos relacionados estão em andamento visando resolver problemas de não-determinismo e outros citados na seção 4.4.

Para complementar a fase de testes, o critério de comunicação entre processos pode ser avaliado utilizando-se a ferramenta *Jumpshot*. Esta que permite visualizar o tempo que cada processo gasta em cada evento, associando-lhe uma cor [40]. Um resultado do uso desta ferramenta pode ser observada na Figura 5.8.

Nas figuras (a) e (b), as áreas cinzentas ilustram o processamento, as azuis ilustram as operações de recebimento de mensagens, as vermelhas ilustram as operações de envio de mensagens e as verdes ilustram as operações de redução. A operação de redução é utilizada no cálculo do critério de convergência do método.

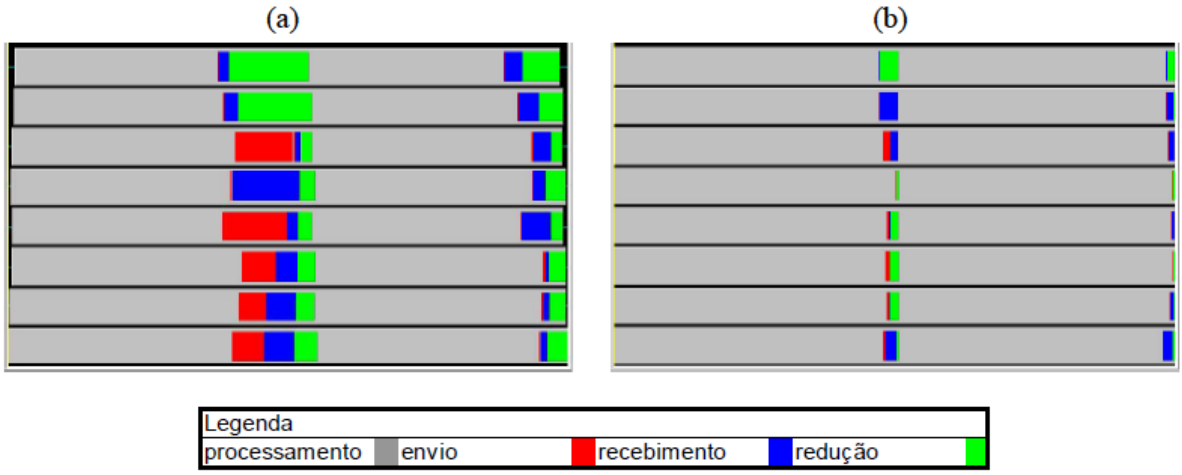

Figura 5.8 Execução do Solver Paralelo na ferramenta *Jumpshot*. (a) sistema com 11.506 incógnitas (b) sistemas com 184.096 incógnitas.

Analisando a figura 5.8, nota-se que o tempo gasto para a comunicação em uma execução com um número maior de incógnitas (b), proporciona um melhor desempenho se comparado a execução em (a). Isso justifica um aproveitamento maior de um grande número de processos para sistemas de equação maiores.

5.4 Análise de Desempenho

Como definido nesta etapa, as métricas de tempo de execução, *speedup* e eficiência foram calculadas para o Solver Paralelo.

Tempo de execução

Nas Figuras 5.9 e 5.10 são apresentados os tempos de execução calculados com o método MDD aditivo de Schwarz, utilizando acurácia de 0,1, 0,001 e 0,001, para sistemas de 11.506 e 184.096 incógnitas, respectivamente. O tempo de execução é dado em segundos.

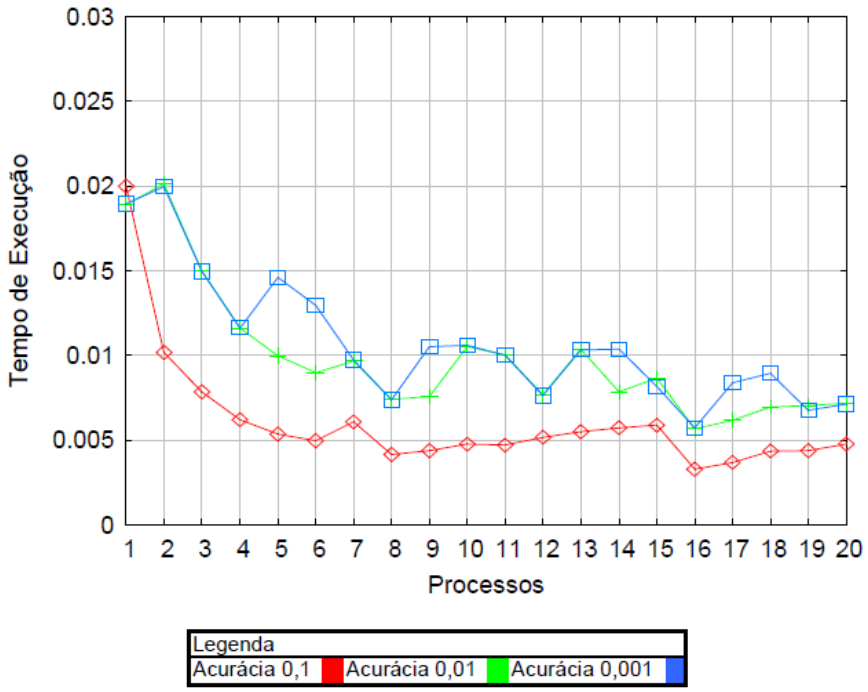

Figura 5.9: Tempo de execução do Solver Paralelo utilizando 11.506 incógnitas

A partir destes resultados, observa-se na Figura 5.9 que o melhor desempenho foi obtido foi com a execução utilizando 16 processos. Com quantidades de processos superiores a 16 ocorre um pequeno aumento de tempo de execução resultante da comunicação entre processos e pela redução de processamento necessário para cada subdomínio.

Com um sistema de maior porte, ilustrado na Figura 5.10. Os resultados obtidos apresentaram-se melhores em uma relação comunicação/processamento, sendo mais eficaz no critério de escalabilidade e reduzindo ainda mais o tempo de execução

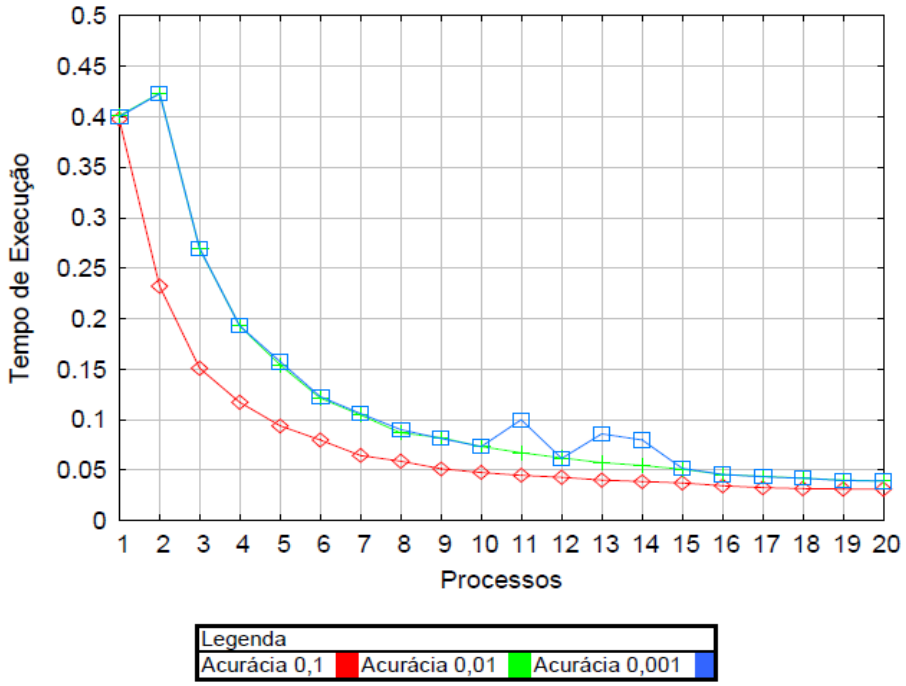

Figura 5.10: Tempo de execução do Solver Paralelo utilizando 184.096 incógnitas

Speedup

Nas Figuras 5.11 e 5.12 mostra a comparação dos *speedups* das execuções utilizando-se acurácia de 0,1, 0,01 e 0,001. O *speedup* é a razão entre o tempo de execução do algoritmo serial mais rápido e o algoritmo paralelo utilizando *p* processadores.

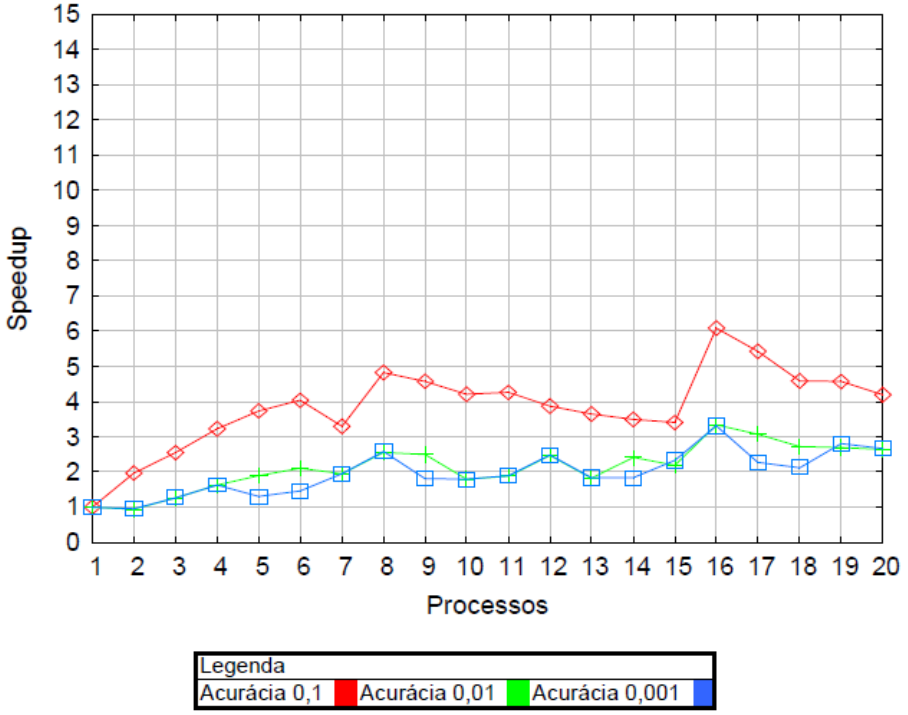

Figura 5.11: S*peedup* do MDD Aditivo de Schwarz utilizando 11.506 incógnitas

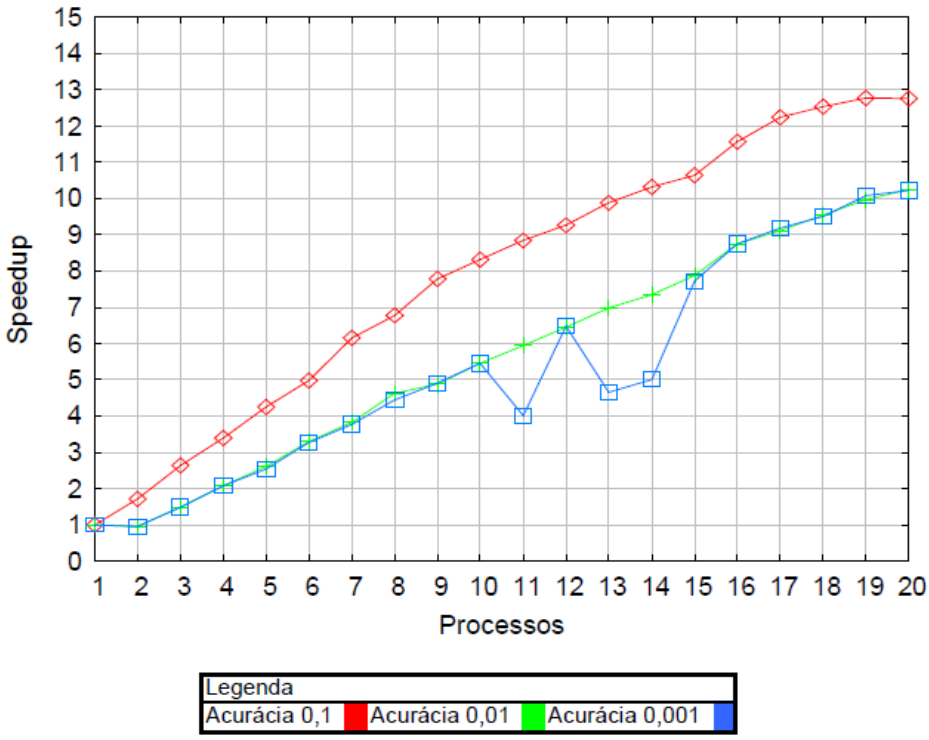

Figura 5.12: S*peedup* do Solver Paralelo utilizando 184.096 incógnitas

Considerando as figuras acima, observa-se que com o aumento do domínio computacional, ou seja, a quantidade de nós de execução no sistema houve aumento no valor dos *speedups*. Pode-se observar também que algumas quedas do valor do *speedup* encontradas nestes resultados são decorrentes do aumento do número de iterações para a convergência do método.

Eficiência

Observa-se nas Figuras 5.13 e 5.14 que a eficiência diminui com o aumento dos processos. Isso se deve ao aumento de subdivisões do domínio computacional, causando sobrecarga na quantidade de comunicações efetuadas.

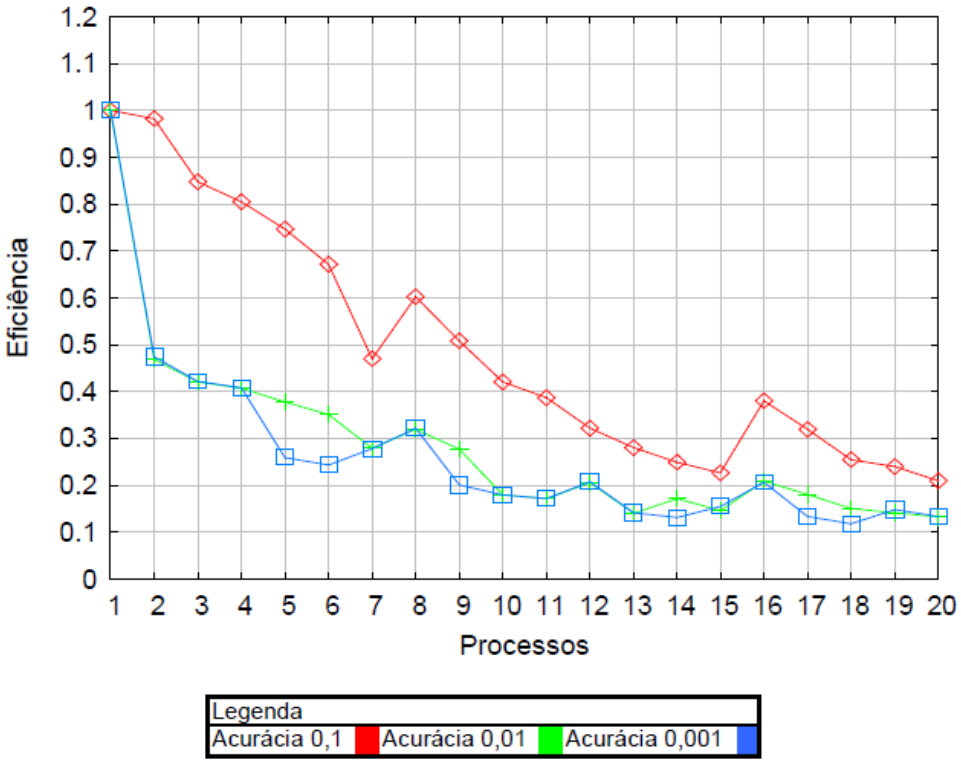

Figura 5.13: Eficiência do Solver Paralelo utilizando 11.506 incógnitas

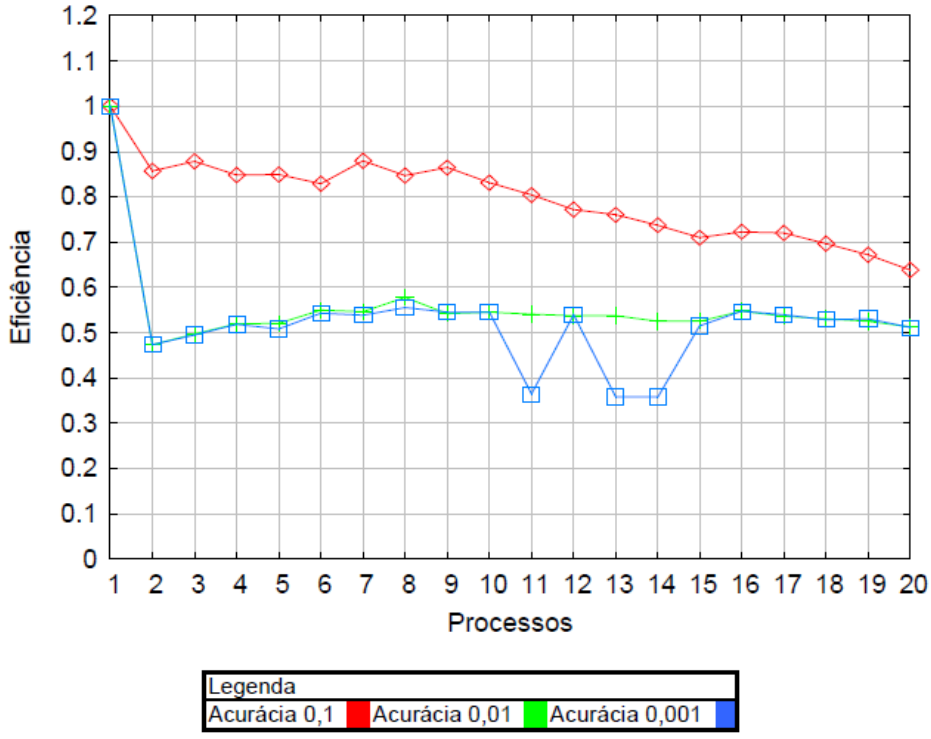

Figura 5.14: Eficiência do Solver Paralelo utilizando 184.096 incógnitas

Uma questão importante deve ser definida após os resultados obtidos. Esta questão se refere ao requisito não funcional confiabilidade que neste estudo de caso está associado a qualidade numérica da solução encontrada. Considerando esta premissa, pode-se observar que utilizando acurácia igual a 0,1 o tempo de execução é bem inferior aos apresentados utilizando-se acurácia 0,01 e 0,001, mas sua qualidade numérica é ruim. No entanto, os resultados obtidos entre as execuções utilizando acurácia de 0,01 e 0,001 são muito semelhantes, sendo mais vantajoso utilizar acurácia de 0,001, já que oferece melhor qualidade numérica.

Após esta análise de desempenho, a equipe de desenvolvimento deve verificar se o desempenho é satisfatório para a aplicação desenvolvida e a partir dos dados coletados deve-se decidir se uma nova iteração das fases da metodologia será desenvolvida. Esta nova iteração possibilitará uma revisão dos requisitos, projeto e implementação da aplicação e espera-se ao seu final, uma otimização dos resultados obtidos.

Capítulo 6

Considerações Finais

6.1 Conclusões

A necessidade de evolução no desenvolvimento de novas estratégias e mecanismos no contexto de aplicações paralelas se mostra cada vez mais evidente no cenário da computação de alto desempenho, principalmente pelo fato da limitação de arquiteturas sequenciais.

Neste trabalho apresentou-se aspectos do ciclo de vida dos modelos mais utilizados encontrados na literatura, tanto para aplicações sequenciais, quanto para aplicações paralelas e a partir disso buscou-se encontrar características importantes nestes modelos com objetivo de propor uma metodologia de desenvolvimento especifica para este tipo de aplicação.

Baseado nos modelos citados na literatura atual e nos paradigmas de desenvolvimento mais utilizados em aplicações comerciais optou-se por propor a metodologia para o paradigma orientado a objetos. Esta metodologia denominada MOODAP descreve cinco fases no seu ciclo de vida onde visam guiar o desenvolvedor na utilização de técnicas para a construção de uma aplicação paralela.

Aliado ao desenvolvimento orientado a objetos foi utilizado a linguagem UML para o desenvolvimento dos diagramas. Estes se mostraram ineficazes em alguns aspectos, como a não generalização para sistemas maiores onde há uma grande quantidade de nós e/ou processos. Porém devem ser utilizados como forma de ilustração para as ações dentro do sistema.

Para a etapa de testes, pouco foi discutido pela falta de ferramentas especificas para testes em aplicações paralelas. Já para a etapa de análise de desempenho, as informações resultantes das métricas utilizadas foram satisfatórias e determinaram com eficiência se as etapas anteriores foram bem sucedidas ou não.

Considerando o processo como um todo, a metodologia MOODAP teve êxito no sentido de preencher algumas lacunas encontradas nas metodologias existentes, como a falta de técnicas e mecanismos específicos para cada fase de desenvolvimento e a construção de uma aplicação desde o seu inicio.

No estudo de caso apresentado, o foco acadêmico da aplicação pode ter deixado de lado algumas características de aplicações comerciais onde apresentam maior quantidade de requisitos não funcionais e um projeto de *software* mais robusto onde são levadas em consideração questões organizacionais e de gerência de projetos.

Em linhas gerais, a metodologia MOODAP deve permitir ao desenvolvedor saber quais passos e quais problemas que permeiam uma aplicação paralela.

6.2 Trabalhos Futuros

Dentro do escopo deste trabalho, algumas perspectivas de novos trabalhos podem ser sugeridos como estudos futuros tendo como objetivo aperfeiçoar ou complementar este trabalho.

- Aperfeiçoar a metodologia MOODAP para tratar de questões de gerência de projeto e questões organizacionais. Para isso poderia ser utilizado modelos de qualidade de *software* como CMMI e MPS.BR e práticas de gerencia de projetos como PMBOK;
- Aplicar a metodologia MOODAP em um estudo de caso de uma aplicação robusta ou de caráter comercial. Desta forma outras características do processo poderiam ser avaliadas mais claramente;
- Analisar de forma mais especifica cada diagrama UML utilizado neste trabalho e propor alterações cabíveis para uma melhor adequação às necessidades de uma aplicação paralela;
- Buscar na literatura ferramentas de teste e análise de desempenho especificas para aplicações paralelas e propor estudos onde serão realizadas comparações entre estas ferramentas.

Referências Bibliográficas

[1] PRESSMAN, R. S. **Software engineering: A practitioner's approach**. Editora McGraw-Hill, 4ª Edição. 1997.

[2] SOMMERVILLE, I. **Software engineering**. Editora Addison-Wesley, 5ª Edição. 1995.

[3] SCHWARTZ, J. I. **Construction of software. In: Practical Strategies for Developing Large Systems.** Menlo Park: Editora Addison-Wesley, 1ª Edição. 1975.

[4] ISO/IEC 12207, **Information Technology – Software Life-Cycle Processes**, 1995.

[5] **Revista de membros da ACM (*Association* for *Computing Machinery*).** Disponível em: *http://www.acm.org/crossroads/xrds6-4/software.html*. Último acesso em: 20 de maio de 2009.

[6] ROYCE, W. W. **Managing the development of large software systems. In: Proceedings of IEEE**, WESCON, 1970.

[7] BOEHM, B. W. **A Spiral Model of Software Development and Enhancement**. Computer, 1988.

[8] SZYPERSKI, C. **Component Software: beyond-oriented programming**. Editora Addison-Wesley. 1999.

[9] FEIJÓ, R. H. B. **Uma Arquitetura de Software Baseada em Componentes Para Visualização de Informações de Informações Industriais**. Natal: UFRN, 2007. Tese (Mestrado) – Programa de Pós-Graduação em Engenharia Elétrica do Centro de Tecnologia da Universidade Federal do Rio Grande do Norte, Natal, 2007.

[10] BROWN, A. W.; WALLNAU, C. K. **"International workshop on component-based software engineering"**. 21ª International Conference on Software Engineering (ICSE'99), 1999.

[11] BROWN, A. W.; SHORT, K. **On Components and Objects: The Foundation of Component-Based Development**. International Symposium Assessment of Software Tools and Technologies. 1997.

[12] SPAGNOLI, L. A.; BECKER, K. **Um Estudo Sobre o Desenvolvimento Baseado em Componentes**. Porto Alegre. 2003.

[13] **Processo Unificado e RUP (Rational Unified Process).** Disponível em *http://cbt.br-web.com/arq/tcc/artigo_01.pdf*. Último acesso em 4 de abril de 2009.

[14] **Rational Unified Process: Visão Geral**. Disponível em *http://www.wthreex.com/rup/portugues/index.htm*. Último acesso em 8 de abril de 2009.

[15] SOARES, M. S. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software**. Conselheiro Lafaiete.

[16] **Manifesto for Agile Software Development**, Disponível em *http://agilemanifesto.org/*. Ultimo acesso em 5 de abril de 2009.

[17] PRESSMAN, R. S., **Engenharia de Software**. Editora McGrawHill, 6ª edição. 2006.

[18] SANTANA, R. H. C; Santana M. J; Souza M. A; Souza P. S. L; Piekarski A. E. T. **Computação Paralela**. Universidade de São Paulo Departamento de Ciências de Computação e Estatística, São Carlos, 1997.

[19] BUYYA, R. **High Performance Cluster Computing, Volume 2: Programming and Applications**, Editora Prentice-Hall. 1999.

[20] DONGARRA, J.; FOSTER, I.; FOX, G.; GROPP, W.; WHITE, A.; TORCZON, L.; KENNEDY, K. **Sourcebook of Parallel Computing**. Editora Morgan Kaufmann. 2002.

[21] D'SOUZA, D.; WILLS, A. **Objects, Components and Frameworks with UML – The Catalysis Approach**. Editora Addison-Wesley. 1999.

[22] LUKSCH, P. **Software Engineering Methods for Designing Parallel and Distributed Applications from Sequential Programs in Scientific Computing**. IEEE Computer Society Washington, DC, USA. 1997.

[23] MURPHY, C. Parallel **Software Engineering - Goals 2000**. COMPSAC – Computer Software and Applications Conference, 1995.

[24] LUKSCH, P.; MAIER, U.; RATHMAYER, S.; WEIDMANN, M. **SEMPA Software Engineering Methods for Parallel Applications**. Project Status. In Georg Scheuerer, TASCflow User Conference – 4ª Edição. 1996.

[25] GOSWAMI, D.; SINGH, A.; PREISS, B. R. A. ***Building*** **Parallel Applications Using Design Patterns**. New York. 2001.

[26] NORBERTO, C. E.; MONGELLI, H.; WUN S. **Algoritmos Paralelos usando CGM/PVM/MPI: Uma Introdução.** As Tecnologias da Informação e a Questão Social - 1ª Edição. Sociedade Brasileira de Computação. Porto Alegre. 2001.

[27] FOSTER, I. **Designing and Building Parallel Programs**. Editora Addison Wesley, 1995.

[28] LUKSCH, P.; MAIER, U.; RATHMAYER, S.; WEIDMANN, M. **SEMPA Software Engineering Methods for Parallel Applications**. Project Report. 28 de Agosto, 1998.

[29] OLIVETE, A. L.; TRINDADE, O. A Utilização da UML no Desenvolvimento de Aplicações Paralelas. XXVII Seminário Integrado de Software e Hardware, Curitiba, 2000.

[30] KOTONYA, G.; SOMMERVILLE, I. **Requirements Engineering : Processes and Techniques**. Editora John Wiley & Sons, 1998.

[31] SOMMERVILLE, I. **Software Engineering**. Editora Addison-Wesley. 6ª Edição, 2007.

[32] MALAN, R.; BREDEMEYER, D. **Functional Requirements and Use Cases**, Disponível em *http://www.bredemeyer.com pdf_files/functreq.pdf.* Último acesso em 10 de setembro 2009.

[33] ELSMARI, R.; NAVATHE, S. B. **Sistema de Banco de Dados**. Editora Pearson Addison Wesley. 4ª edição. São Paulo, 2005.

[34] FOWLER, M. **UML Essencial: um breve guia para a linguagem-padrão de modelagem de objetos**. Editora Bookman, 2000.

[35] COCKBURN, A. **Writing effective use cases**. Editora Addison Wesley, 2000.

[36] STADZISZ, P. C. **Projeto de Software usando UML**, 2002.

[37] CURVELLO, M. A. **Tolerancia a Falhas em Programas Paralelos,** Disponível em *http://www-di.inf.puc-rio.br/~endler/courses/DA/Monografias/08/MarcoAntonio-mono.pdf.* Último acesso em 20 de setembro de 2009.

[38] GORINO, F. V. **Balanceamento de Carga em Clusters de Alto Desempenho: Uma Extensão para Lam/Mpi**. 2006.

[39] MEFFE, C.; MUSSI, E. O. P.; Mello, L. R. **Guia de Estruturação e Administração do Ambiente de Cluster e Grid**. 2006.

[40] CHAN, A.; ASHTON, D.; LUSK, R.; GROPP, W. **Jumpshot-4's User's Guide**. Disponível em *http://www-unix.mcs.anl.gov/perfvis/software/viewers/*. Último acesso em 08/11/2009.

[41] **Introdução ao MPI**, Disponível em *http://www.cenapad.unicamp.br/servicos/ treinamentos/ apostilas/apostila_MPI.pdf*. Último acesso em 13 de agosto de 2009.

[42] SOUSA, S. H. G. **Uma Introdução ao PVM**, Disponível em *http://psgr.dep.fem.unicamp.br/index.php?option=com_content&task=view&id=51&Itemid 31*.Último acesso em 22 de setembro de 2009.

[43] LUNGARZO, G, O. **Uma introdução a Pthreads em linguagem C**, 2003.

[44] KÜSEL, R. A. M. **Apostila de Programacao OpenMP**, Disponível em *http://www.cenapad.unicamp.br/ servicos/treinamentos/openmp.shtml*. Último acesso em 23 de setembro de 2009.

[45] AMARAL, L. A. **Diminuição da Intrusao do Teste de Software em Programas Paralelos**. 2006. Dissertação de Mestrado (Mestrado em ciência da Computação). Pontifícia Universidade Católica do Rio Grande do Sul Faculdade de Informática - Pós-Graduação em Ciência da Computação, Porto Alegre.

[46] AMARAL, L. A. **Teste de Software em Programas Paralelos**. Disponível em *http://www.inf.pucrs.br/~eduardob/pucrs/research/students/LeonardoAmaral/TI/TI2_leonard o_amaral.pdf* . Último acesso em 23 de setembro de 2009.

[47] ALVES NETO, P. O; SILVA, R. S. Tecnologias **de Sistemas Distribuídos Implementados em Java: Sockets, RMI, RMI-IIOP e Corba**. Anuários de produção acadêmica docente. Vol. II, N^{o}.3. 2008.

[48] GONÇALVES, J. A. B. **Desenvolvimento de uma agenda distribuída utilizando Java RMI**. Dissertação, Faculdade de Jaguariúna. Jaguariúna, 2005.

[49] RIZZI, R. L. **Modelo Computacional Paralelo para a Hidrodinâmica e para o Transporte de Massa Bidimensional e Tridimensional**. 2002. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre-RS.

[50] GALANTE, G. **Métodos de Decomposição de Domínios para a Solução Paralela de Sistemas de Equações Lineares**. 2003. Trabalho de Conclusão (Bacharelado em Informática) — Universidade Estadual do Oeste do Paraná, Cascavel, PR.

[51] SAAD, Y. **Iterative Methods for Sparse Linear Systems**. PWS Publishing Company. 1996.