INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SUL-RIO-GRANDENSE - IFSUL, CAMPUS PASSO FUNDO
CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET

VINÍCIUS CASTELANI RECK

ANÁLISE DE MIDDLEWARES DE GRIDS OPORTUNISTAS PARA AMBIENTES CORPORATIVOS E EDUCACIONAIS

Élder F. F. Bernardi

PASSO FUNDO, 2011

VINÍCIUS CASTELANI RECK

ANÁLISE DE MIDDLEWARES DE GRIDS OPORTUNISTAS PARA AMBIENTES CORPORATIVOS E EDUCACIONAIS

Monografia apresentada ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-Rio-Grandense, Campus Passo Fundo, como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet.

Orientador(a): Prof. Élder F. F. Bernardi

PASSO FUNDO, 2011

DEDICATÓRIA

Dedico esse trabalho a todos os leitores, que acreditaram no esforço realizado para concretização do mesmo.

AGRADECIMENTOS

Primeiro gostaria de agradecer as pessoas que estiveram ligadas diretamente a esse trabalho, são elas: Alexandre T. Lazzaretti, João Eduardo Pedrini, Anubis Graciela de M. Rossetto, José Antônio O. de Figueiredo e, especialmente, ao meu orientador Élder F. F. Bernardi, não só pelo excelente trabalho como orientador, mas como amigo, ainda, por seu apoio e ajuda que tive ao longo do ano. Outras pessoas que não tiveram ligação direta, mas que igualmente foram importantes: Meus pais e irmã, por todos os méritos e créditos que uma família pode ter na vida de cada um de nós; a minha namorada Vanessa, pelo amor, carinho, apoio e exemplo que tenho de sua parte; aos meus colegas Thomaz C. Xavier, Vinícius P. Lima, Giuseppe Matheus B. Pereira, José A. Almeida e Norton M. Vanz pela amizade e convivência. Para finalizar gostaria de agradecer a todos os professores, funcionários, colegas e amigos do IFSUL, que conheci ao longo do curso.

RESUMO

Mesmo com todos os avanços realizados na área de *hardware*, fornecendo novos e melhores equipamentos em curtos espaços de tempo, é comum encontrar cenários em que um único computador, mesmo este sendo de última geração, não seja suficiente para suprir toda a demanda de computação exigida. Dessa necessidade, surgiram os ambientes de computação de alto desempenho, construídos a partir de ambientes de computação paralela. Uma das formas mais acessíveis de trazer esses ambientes para o meio empresarial, é a utilização de computação oportunista. Onde as máquinas com ciclos ociosos são aproveitadas para realizar o processamento de grandes quantidades de dados em paralelo. Mas para que isso seja possível se faz necessário a utilização de *middlewares* de *grid* com características oportunistas. A motivação principal deste trabalho é realizar um estudo de alguns dos principais *middlewares* atuais e fornecer uma base de consulta para auxiliar na decisão a adoção de ambientes paralelos em empresas e instituições de ensino, de modo a fornecer as principais características desses cenários e sugerir *middlewares* que satisfação essas características. Para isso, foi realizado um estudo bibliográfico de alguns dos principais *middlewares*, seguido da eleição e implantação do mais apto a suprir as necessidades do cenário proposto. Para avaliar os ganhos de desempenho, utilizou-se uma aplicação *case* sobre o ambiente criado, o que retornou resultados positivos, comprovando os benefícios da adoção de ambientes paralelos oportunistas.

Palavras-chave: Computação Oportunista, Computação Paralela, Computação em Grade, Programação Paralela e Distribuída

ABSTRACT

Even with all the advances in the hardware area, providing new and better equipment in a short period of time, it is common to find scenarios where a single computer, even though last generation, not supply all the computing demand required. This need arose computing environments, high-performance build from parallel computing environments. One of the more affordable ways to bring these environments to the business, is the use of opportunistic computing. Where machines with idle cycles are utilized to perform the processing of large amounts of data in parallel. But to make this possible is necessary the use of grid with opportunistic characteristics. The main motivation of this work is to conduct a study some of the main current middleware and provide a basis for consultation to assist in the decision will adoption of parallel environments in corporations and educational institutions, to provide the main characteristics of these scenarios and suggest middleware that meet these characteristics.For this, was performed a bibliographic review of some key middleware, followed by the election and implementation of the fittest to meet the needs of the proposed scenario. To evaluate the performance gains, was used a case application on the environment created, which returned positive results, proving the benefits of adopting opportunistic parallel environments.

Keywords: Opportunistic Computing, Parallel Computing, Grid Computing, Parallel and Distributed Programming.

Lista de Figuras

Lista de Tabelas

Abreviaturas

API	*Application Programming Interface*
BMBF	*German Ministry for Education and Research*
BOINC	Berkeley Open Interface for Network Computing
BoT	Bag-of-Task
CA	*Certificate Authority*
CAS	*Community Authorization Service*
CIS	*Common Information Service*
CPU	*Central Processing Unit*
GPU	*Graphics Processing Unit*
GSI	Grid Security Infrastructure
GUI	Graphical User Interface
HiLA	*High Level API for Grid Applications*
HPC	High-Performance Computing
IDB	*Incarnation Data Base*
IIS	Information Services
JSDL	*Job Submission Description Language*
MOSIX	Multicomputer Operating System Unix
MPI	*Message Passing Interface*
MSDE 2000	Microsoft SQL Server 2000 Desktop Engine
OGF	*Open Grid Forum*
PD	Processamento Distribuído
POSIX	Portable Operating System Interface for Unix
PP	Processamento Paralelo

SGBD	Sistema de Gerenciamento de Banco de Dados
SMP	*Symmetric Multiprocessor*
SO	*Sistema Operacional*
SOA	*Service-Oriented Architecture*
SOAP	Simple Object Access Protocol
SWAN	Sandboxing Without A Name
UCC	UNICORE command line client
UNICORE	*UNiform Interface to COmputer REsources*
URC	UNICORE Rich Client
WEB	World Wide Web
WSRF	Web Services Resource Framework
XMPP	Extensible Messaging and Presence Protocol
XSLT	*Extensible Stylesheet Language for Transformation*

SUMÁRIO

1. Introdução

As rotinas e aplicações computacionais a nível acadêmico, científico e corporativo são cada vez mais complexas e dispendiosas, suas execuções demandam recursos que vão além da capacidade das arquiteturas de computadores tradicionais. Para suprir essas necessidades surgiram várias arquiteturas de processamento alternativas. Dentre essas, podemos destacar os supercomputadores, que oferecem grande poder de processamento e alto desempenho. São utilizados principalmente nas áreas de pesquisa científica e simulações. Seu uso está restrito a grandes corporações e consagradas instituições acadêmicas devido aos elevados custos para aquisição, manutenção e atualização destes equipamentos [1].

Outra alternativa são *clusters* computacionais, um ambiente de computação paralela formado por um conjunto de computadores, chamados nós, interligados por uma rede de interconexões [2]. Normalmente estão em gabinetes ou sala especialmente refrigerada, compostos por máquinas dedicadas e homogêneas, a níveis de configurações de *hardware* e *software*, além de várias ligações redundantes entre os nós.

A vantagem dos *clusters* comparando com supercomputadores está na arquitetura utilizada, pois em *clusters* se utilizam computadores convencionais, tornando mais viável sua implementação, porém, o valor investido em instalações adequadas e o próprio valor das máquinas dedicadas, ainda exigem um investimento significativo.

Mas a solução que vem se destacando no mercado emergente são os *grids* computacionais. Semelhantes ao *cluster*, provêm um alto poder computacional através do trabalho em conjunto de seus nós. No entanto seu diferencial está na utilização de dispositivos heterogêneos e o compartilhamento de recursos. Em um *grid* é possível anexar facilmente computadores de diferentes arquiteturas, tornando flexível a escolha de computadores para essa função. Além disso, aliado ao advento da Internet, sua escalabilidade ganha níveis globais, já que é possível conectar qualquer dispositivo, em qualquer lugar do mundo, para fazer parte de um *grid*.

De fato, *cluters* e *grids* resolveram a necessidade de tecnologias econômicas e computacionalmente viáveis, para suprir a demanda por maiores recursos computacionais. Mas ainda restou um desafio: como aplicar isto em pequenas e médias empresas, ou mesmo instituições menores? *Clusters* eliminaram a necessidade de adquirir supercomputadores vendidos na casa de milhões de dólares, mas ainda exigem um alto investimento em infraestrutura. *Grids* tornaram possível a adoção de computadores das mais variadas arquiteturas e eliminaram as limitações geográficas, porém ainda exigem um certo investimento. Como minimizar ainda mais estes valores, a ponto de tornar viável a qualquer organização usufruir tais tecnologias?

Pensando nesta linha, surgiu a computação oportunista [3], que visa utilizar o poder computacional "desperdiçado" de máquinas ociosas para suprir a demanda de poder computacional existente. Para isso, utiliza-se das tecnologias de *grid* aliadas a uma estrutura de gerenciamento

de nós autônomos. A ideia base é identificar computadores que não estão utilizando seu poder computacional ao máximo e enviar-lhes pequenas tarefas que compõem problemas de grande escala, de forma transparente ao usuário. Assim, é possível utilizar de infraestruturas já formadas para outros fins, com o intuito de diminuir ao máximo os custos necessários para sua utilização. Por exemplo: em uma empresa onde já existe uma rede de computadores previamente instalada, para utilização dos funcionários, é possível implantar um sistema de computação oportunista utilizando estes mesmos computadores para agilizar o processamento das faturas. Outro exemplo prático está em instituições de ensino que fornecem laboratórios de informática aos alunos. Muitas vezes estes computadores são deixados ligados em estado ocioso por longos períodos, que podem ser amplamente utilizados para resolver equações de grandes escalas, exigidas pelas mais diversas áreas da instituição.

Várias soluções de computação são apresentadas em forma de *middlewares*, implementando diferentes abordagens. Essas soluções possuem suas próprias características e particularidades que devem ser levadas em consideração na hora de decidir qual *middleware* será adotado. Um estudo aprofundado desses *middlewares* e o levantamento do cenário são essenciais para evitar futuros transtornos com a instalação e manutenção dos recursos. O que inicialmente seria uma escolha econômica, pode acarretar em custos elevados com suporte, ou mesmo com a não utilização dos recursos, devido ao alto nível de conhecimento específico exigido para utilização de alguns *middlewares*.

Por fim, este trabalho consiste em estudar e compreender as ferramentas existentes para o uso de computação oportunista, identificando suas particularidades, vantagens e desvantagens e avaliar qual está mais apta a resolver os problemas enfrentados em determinados cenários. Ao final deste trabalho será possível, as organizações interessadas em implantar uma solução oportunista para aplicações que demandam grande poder computacional, ter uma visão de computação paralela e as principais soluções existentes para sua implementação, com base suficiente para definir qual *middleware* melhor atende às suas necessidades. Para isso, será realizado um estudo aprofundado destes *middlewares*, bem como sua implantação, identificando problemas, facilidades, peculiaridades e dificuldades de cada ferramenta. Ao final será implementado uma das soluções propostas em um cenário de ambiente real, composto por máquinas convencionais de uso geral, com uma aplicação *case* real, que faz uso de modelos matemáticos e grandes quantidades de dados para realização de simulações agrícolas e assim, tentar comprovar os benefícios do uso da paralelização oportunista.

1.1 Objetivo Geral

Identificar, descrever alguns dos principais *middlewares grid* e implementar a mais adequada solução para um cenário real, que corresponda a realidade atual das empresas e instituições de ensino.

1.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

1. introduzir os conceitos de processamento paralelo;
2. analisar as peculiaridades de *grid* e computação oportunista;
3. identificar, descrever e avaliar implementações de alguns dos principais *middleware* de *grid* da atualidade, para fornecer uma fonte de consulta e auxílio a decisões, na escolha do *middleware* a ser adotado por empresas e instituições;
4. fornecer uma ferramenta para facilitar o uso do *grid*;
5. Implementar a paralelização de uma aplicação.

1.3 Organização do Documento

O restante do documento encontra-se organizado da seguinte maneira: no Capítulo 2 é feita a introdução aos conceitos de processamento paralelo, definição de um *grid* computacional, apresentando alguns conceitos de programação paralela e definição do *middlewares* estudados. No Capítulo 3 é abordada a ferramenta desenvolvida para auxiliar a submissão de tarefas ao *grid*. No Capítulo 4 estão relatados os experimentos realizados, bem como todos os procedimentos para que fosse possível a realização dos mesmos. O Capítulo 5 apresenta o fechamento do trabalho, com as conclusões obtidas e a relação dos trabalhos futuros.

2. Fundamentação Teórica

Atualmente o ambiente de trabalho está intimamente ligado à tecnologia, especialmente nas empresas que procuram estar atualizadas e preocupadas em manter a competitividade [4]. Do equipamento utilizado para bater ponto, das ferramentas de controle de produção, do sistema administrativo aos computadores utilizados pelos funcionários. Existe uma forte ligação com tendências tecnológicas, onde todas podem convergir a um único sistema de gerenciamento central. Este deve ser capaz de fornecer todos os recursos necessários, sem tornar-se um gargalo no sistema, comprometendo toda a estrutura planejada. Por este motivo, a disseminação de processamento paralelo e distribuído tem sido constante neste ambiente, visto as necessidades de alto processamento exigido pelos sistemas atuais.

Neste Capítulo serão explanados os conceitos envolvidos em processamento paralelo e distribuído e também uma visão inicial de *grids* e computação oportunista.

2.1 Processamento de Alto Desempenho

Computação de alto desempenho ou *high performance computing* (HPC) é o uso de arquiteturas especias para a execução de programas e aplicativos avançados, de forma eficiente, confiável e rápida [5]. Hoje amplamente utilizado para resolver problemas de grande escala, como por exemplo: previsões do tempo, problemas matemáticos complexos, simulações físicas, busca por petróleo e até mesmo na busca da cura para as mais diversas enfermidades conhecidas [5]. Nesse conceitos são utilizados múltiplos computadores trabalhando em paralelo, com o objetivo de resolverem um mesmo problema no menor tempo possível.

2.1.1 Processamento Paralelo

Segundo De Rose e Navaux em [5, p. 12], processamento paralelo se define como: "várias unidades ativas cooperando para resolver um mesmo problema, atacando cada uma delas parte do trabalho e se comunicando para troca de resultados intermediários ou no mínimo para divisão inicial do trabalho e para junção final dos resultados". Porém, uma aplicação só pode se aproveitar dos benefícios da HPC caso seja programada para tal, de forma a permitir que os processos internos sejam divididos em fluxos de processamento independentes. Assim, o sistema pode delegar estes fluxos a diferentes unidades de processamento.

2.1.2 Processamento Paralelo x Distribuído

No caso de processamento paralelo, o intuito é prover uma forma de otimizar o processamento das informações através de computadores organizados em uma sala, ou mesmo em uma

mesma estrutura(Rack). Já para o processamento distribuído (PD) a premissa é a mesma, mas com limites geográficos maiores. Quando se fala em processamento distribuído se arremeta a ideia de computadores interligados, mas distantes fisicamente, interligados por redes de interconexões ou mesmo através da Internet [5].

2.1.3 Arquiteturas Paralelas

Baseando-se no fato de um computador executar uma sequência de instruções sobre uma sequência de dados, diferenciam-se os fluxo de instruções e o fluxo de dados. Dependendo dos fluxos sendo múltiplos ou não, e com a combinação das possibilidades, Flynn [6, p. 06] propôs 4 classes demonstradas na Tabela 2.1.3:

Tabela 2.1 – Classificação conforme Flynn

	SD(*Single Data*)	**MD**(*Multiple Data*)
SI(*Single Instruction*)	**SISD** Máquinas von Newumann convenconais	**SIMD** Máquinas vetorias
MI(*Multiple Instruction*)	**MISD** Sem representantes(até o momento)	**MIMD** Multiprocessadores e multi-computadores

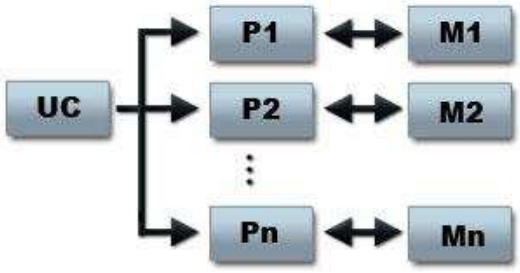
SISD: "um único fluxo de instrução atua sobre um único fluxo de dados" [5]. O fluxo de instrução alimenta uma unidade controladora (UC) que ativa a unidade processadora (P). A unidade (P) acessa a memória (M) através de um único fluxo de dados, realiza o processamento e reescreve o resultado na memória. Este processo está demonstrado na Figura 2.1. Temos como exemplo dessa classe os computadores pessoais com um processador com um único núcleo de processamento.

Figura 2.1 – SISD

MISD: "múltiplos fluxos de instruções atuariam sobre um único fluxo de dados" [5]. Na prática, diferentes instruções operam a mesma posição da memória ao mesmo tempo, executando instruções diferentes. Atualmente é improvável construir uma máquina que se comporte dessa forma, por isso está categoria não tem representantes.

SIMD: "uma única instrução é executada ao mesmo tempo sobre múltiplos dados" [5]. Uma única unidade controladora (UC) recebe um fluxo de instruções e envia a mesma instrução para os diversos processadores (Pn) que executam suas instruções em paralelo de forma síncrona sobre diferentes fluxo de dados. Este processo está demonstrado na Figura 2.2. Nesta classe podemos citar os computadores vetoriais.

Figura 2.2 – SIMD



MIMD: cada unidade controladora (UC) recebe um fluxo de instruções próprio e repassa-o para sua unidade processadora(Pn), que executa a instrução sobre um fluxo de dados próprio e de forma assíncrona [5]. Esta classe pode, ainda, ser dividida em duas subclasses, que se diferenciam entre si de acordo com a forma de acesso a memória. Máquinas com memória compartilhada são conhecidas como multiprocessadores, demonstrado na Figura 2.3, enquanto máquinas que não tem memória compartilhada são chamadas de multi-computadores, visível na Figura 2.4.

Figura 2.3 – MIMD: Multiprocessadores

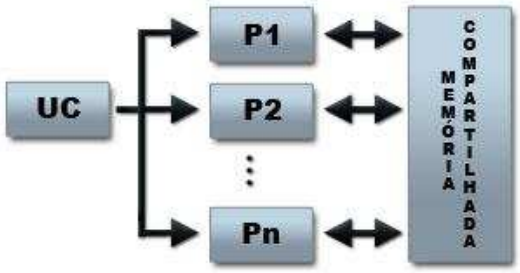
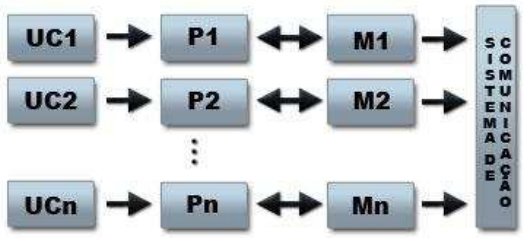


Figura 2.4 – MIMD: Multicomputadores



2.1.4 Classificação de Multiprocessadores Conforme a Memória

- Acesso uniforme à memória (UMA) onde a memória fica centralizada e a latência de acesso a à mesma é igual para todos os processadores.
- Acesso não uniforme à memória (NUMA), a memória é distribuída, implementada com múltiplos módulos associados um a cada processador. O espaço de endereçamento é único e cada

processador pode endereçar toda memória.

- Arquiteturas de memória somente com cache (COMA), todas as memórias locais são estruturadas como memória cache, com muito mais capacidade que uma cache tradicional.

Já os multi-computadores têm a forma de acesso à memória denominada como NORMA, sem acesso direto à memória de outra máquinas. Como uma arquitetura tradicional inteira foi replicada para construir está máquina, cada nó só pode acessar sua própria memória. O acesso a memória de outro nodo se dá via troca de mensagens, através de uma rede de interconexão.

2.1.5 Classificação Comercial de Máquinas Paralelas

Os principais modelos dessa classificação são [5]: Multiprocessadores simétricos (SMP), redes de estação de trabalho (NOW) e máquinas agregadas (COW). Salienta-se que todos os modelos aqui citados pertencem a classe MIMD.

1. No modelo SMP os processadores são completos e conectados através de uma memória compartilhada, através de um barramento, onde todos tem o mesmo tempo de acesso. Os processadores se comunicam através da memória compartilhada [7].

2. Um *Cluster* Computacional é um conjunto de computadores interligado por uma rede de interconexão onde trabalham em conjunto para solucionar problemas computacionais mais rapidamente. Os computadores pertencentes a um *cluster* recebem o nome de *nó*, cada *nó* recebe uma determinada tarefa para ajudar a solucionar um problema maior, ou ainda pode servir como reserva, caso ocorra uma falha em algum dos nós em trabalho.

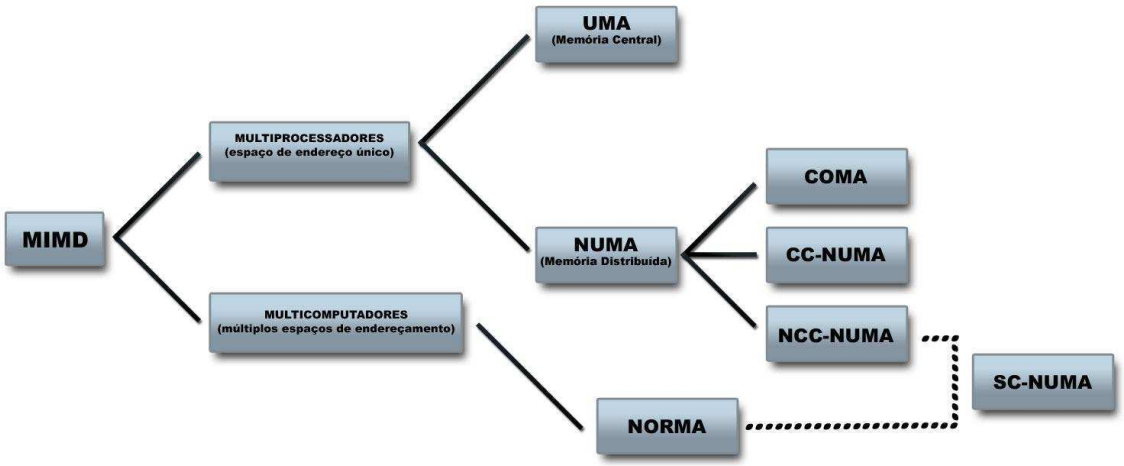
   (a) *Network Of Workstations*(NOW) são sistemas constituídos por nós não dedicados, interligados por redes *Ethernet*, constituindo assim uma máquina NORMA [5].

   (b) *Cluster of Workstations* (COW) possuem a mesma constituição de uma NOW, porém os nós são dedicados exclusivamente às aplicações paralelas. Assim, a máquina pode ser otimizada, configurando o *hardware* e os *softwares* dos nós, para o uso exclusivo de aplicações paralelas [5].

3. Por ora, pode-se definir *grid* como um grande numero de *nós* espalhados geograficamente interconectados por uma rede. Ou ainda um *grid* pode ser a união de vários *clusters* espalhados pelo globo. Mais adiante, na Seção 2.2, falar-se-a melhor sobre *grid*.

Na Figura 2.5 é possível visualizar a classificação conforme o compartilhamento de memória segundo Hwang [8].

Figura 2.5 – Classificação conforme o compartilhamento de memória



2.2 Grids Computacionais

O autor Dantas [3, p. 207] define *grid* com um conceito utilizado por vários autores:

> Um ambiente computacional distribuído paralelo que permite a compartilhamento, seleção, a agregação de recursos autônomos e geograficamente distribuídos. Estas operações e recursos podem ser utilizados durante a execução de uma aplicação, dependendo de sua disponibilidade,capacidade, desempenho e custo. O objetivo é prover aos usuários serviços com requisitos de qualidade corretos para o perfeito funcionamento de suas aplicações.

Um *grid* pode ser definido como: "uma plataforma heterogênea de computadores geograficamente dispersos, onde o usuário faz acesso ao ambiente através de uma interface única" [3, p. 203]. O princípio de um *grid* é fornecer acesso a um ambiente distribuído de forma transparente, onde o usuário possa fazer uso do processamento, do armazenamento e dos equipamentos do ambiente sem precisar conhecê-lo. A partir desta premissa, muitos autores fazem uma analogia do *grid* com a rede elétrica. O usuário conecta seu aparelho eletrônico na rede e instantaneamente tem acesso à energia necessária para fazê-lo funcionar, sem a necessidade de saber detalhes da origem desta energia.

2.2.1 Grids x Clusters

De uma forma simples, pode-se diferenciar *clusters* de *grid* pelo gerenciamento de recursos e serviços. Em um *cluster* tem-se uma entidade central, detentora do sistema, que gerencia de forma unilateral, por outro lado, o *grid* possuí varias entidades denominadas "organizações virtuais", que fazem uso dos recursos e serviços. [3].

2.2.2 Topologia de Grid

Grids não possuem uma regulamentação com relação a um modelo de arquitetura, mas Dantas [3] descreve dois possíveis modelos para adoção. Neste trabalho será abordado somente um deles, ilustrado na Figura 2.6 [3, p. 210], com o intuito de transmitir ao leitor um esclarecimento de *grids*.

Figura 2.6 – Topologia de Grid

Aplicações e Serviços: Neste nível, encontram-se os pacotes de aplicações e serviços que fazem uso do ambiente paralelo fornecido pelo *grid*, bem como as ferramentas de desenvolvimento. As aplicações variam de acordo com as necessidades das organizações virtuais, enquanto os serviços são responsáveis por prover diversas funções de gerenciamento dos recursos do *grid*.

Middleware: camada responsável por unificar o ambiente, através de protocolos e funções que visam abstrair a complexidade e compatibilidade de ambientes distribuídos. É através desta camada que um *grid* pode oferecer suporte a ambientes heterogêneos, como diferentes arquiteturas, sistemas operacionais, protocolos de redes entre outros.

Recursos: neste nível identificamos o *hardware* que compõem o *grid*. Podemos citar como exemplo: computadores, servidores de banco de dados e dispositivos de armazenamento.

Rede: neste nível encontram-se os dispositivos que farão as interconexões dos recursos. Como exemplo deste nível podemos citar os *switches* e roteadores.

2.2.3 Taxonomia de Computação em Grid

Computação em *Grid* pode ser classificada, do ponto de vista do usuário, quanto aos sistemas e serviços oferecidos. Segundo Krauter, Buyya e Maheswaran [9], os sistemas de Grid são divididos de acordo com suas principais atividades: *Computational Grid*(*Grids* de Processamento), *Data Grid*(*Grids* de Dados) e *Service Grid*(*Grids* de Serviços).

Computational Grid : sistema de *grid* com o objetivo principal voltado a fornecer um alto poder de computação para realizar tarefas complexas. Normalmente compostos por computadores com processadores robustos e grandes quantidades de memória. Este sistema se subdivide

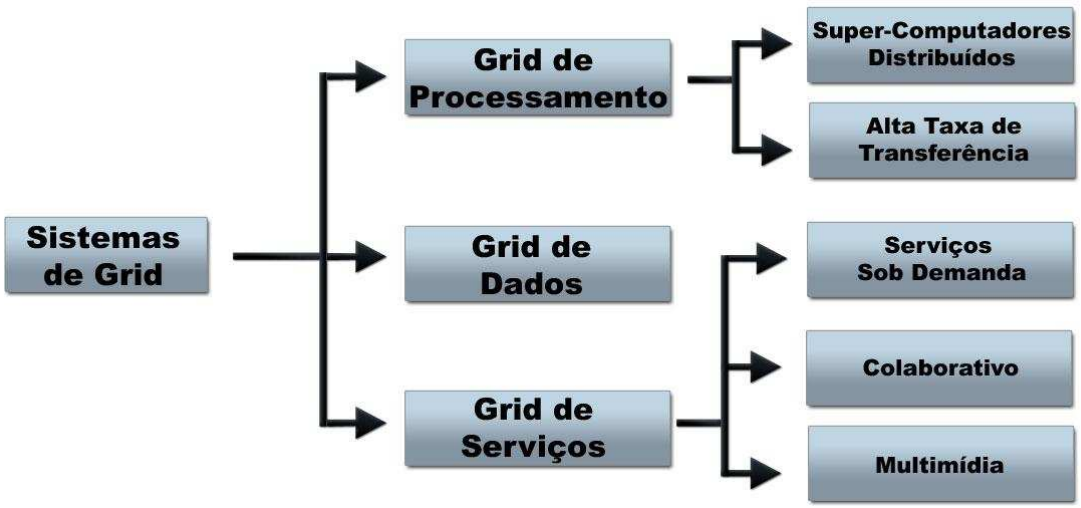
em duas partes: Computação de alto desempenho(*high performance*), destinado a execução de aplicações em paralelo focado no desempenho individual e cada aplicação. E computação de alta vazão(*high throughput*), destinado a distribuir aplicações com objetivo de promover o desempenho de todo o sistema.

Data Grid : sistema de grid voltado para armazenamento, transferência e alta disponibilidade de dados, acessíveis pelo usuário de forma transparente. Composto por computadores que prezam os componentes de transmissão, acesso ao disco e velocidade de barramentos, procurando uma melhor performance no tratamento da informação. Atualmente estes sistemas trabalham em conjunto com SBGDs fornecendo ao usuário uma visão única do sistema.

Service Grid : sistema de *grid* especializado na disponibilização de diversos serviços que dificilmente seriam viáveis com o uso de um único computador. Esses serviços podem ser classificados nas seguinte categorias [10]: serviços de computação, serviços de dados, serviços de aplicação, serviços de informação, serviços e conhecimento.

Na Figura 2.7 é possível visualizar a divisão proposta por Krauter, Buyya e Maheswaran [9].

Figura 2.7 – Taxonomia de Computação em Grid



2.2.4 Grids Oportunístas

Um *Opportunistic Grids*(*Grids* Oportunístas) pode exercer as características de uma ou mais das classificações propostas por Krauter, Buyya e Maheswaran [9], sendo que, na em grande parte dos casos, trata-se de um *grid* de processamento. Compostas, na maioria das vezes, por computadores convencionais. Este novo sistema tem como objetivo aproveitar melhor o uso de recursos já existentes em empresas e instituições.

De acordo com Dantas [3, p. 204], "a principal diferença entre a computação paralela oportunista, os *clusters* e *grids*, está baseada na utilização de ciclos ociosos de máquinas durante

a execução das tarefas paralelas". O autor [3] ainda ressalta que não existe nenhuma autoridade reguladora do uso de recursos disponíveis ou serviços providos por organizações virtuais. O conceito de computação oportunista está baseado na ideia de utilização das tecnologias consagradas de sistemas distribuídos, para prover um alto poder computacional, aliado ao enfoque para obter um melhor desempenho na presença de arquiteturas heterogêneas e na utilização de recursos da forma menos intrusiva possível aos nós integrantes do sistema.

Dantas [3, p. 205] faz um comparativo na Tabela 2.2 das características de *grid*, *cluster* e computação oportunista.

Tabela 2.2 – Diferenças entre configurações: Oportunista, *Cluster* e *Grid*

Características	**Oportunista**	**Cluster**	**Grid**
Domínio	Múltiplos	Único	Múltiplos
Nós	Milhões	Milhares	Milhões
Segurança do Processamento e Recurso	Necessária	Desnecessária	Necessária
Custo	Baixo, uso exclusivo de recursos ociosos	Alto, pertencente a um único domínio	Alto, todavia dividido entre domínios
Granularidade do problema	Muito grande	Grande	Muito Grande
Sistema Operacional	Heterogêneo	Homogêneo	Heterogêneo

2.3 Programação Paralela

Com o desenvolvimento do processamento paralelo, surgiram novos desafios na área de programação. Agora o programador precisa trabalhar com um novo ambiente de processamento e ficar atento às novas possibilidade e problemas deste ambiente. Se faz necessário uma visão da forma de como programar para explorar ao máximo as possibilidade de um sistema paralelo. Para isso podemos citar duas abordagens de programas paralelos:

Explícita: O programador fica responsável pelo paralelismo da aplicação, é ele quem define, através da programação do código fonte, como e quando a aplicação deve paralelizar suas tarefas. Esta abordagem traz a possibilidade de um maior controle sobre a paralelização, com isso é possível explorar de forma mais eficiente os recursos oferecidos pelo ambiente. Contudo, esta abordagem exige do programador conhecimentos avançados da linguagem de programação e de programação paralela, o que pode tornar dispendioso o seu desenvolvimento.

Implícita: Nesta abordagem a paralelização da aplicação fica sobre responsabilidade do compilador, o programador não precisa ter conhecimentos específicos de paralelismo e com isso pode programar de forma sequencial. Esta abordagem torna a tarefa de programação menos complexa,

o que facilita seu desenvolvimento, porém compromete o desempenho da aplicação, visto que sua paralelização fica limitada a qualidade do compilador [11].

Em programação paralela ainda podemos encontrar dois níveis de programação:

Intra-nó: Neste nível as tarefas são divididas entre os processadores e a memória é compartilhada. Esta arquitetura está descrita na seção 2.3.1 e pode ser visualizada na Figura 2.3.

Entre-nós: Aqui as tarefas são divididas entre os nós e a comunicação entre eles é feita através de mensagens. Também vimos está arquitetura na seção 2.3 visível na Figura 2.3.

2.3.1 Programação Intra-nó

A programação realizada neste nível pode ser feita pelo uso de processos leves (através de *threads*), gerando múltiplos fluxos de execução que são executados, cada um, por um processador diferente mas compartilhando a mesma memória.

A programação por *threads* é denominada multiprogramação leve pois exerce uma menor sobrecarga sobre sistema, são de fácil gerenciamento e compartilham os dados dentro do espaço de memória alocado pelo seu criador, ou processo pai. Dessa forma torna-se fácil a troca de informações entre os fluxos, aonde para isso, as *trheads* utilizam chamadas de leitura e escrita na memória.

Podemos implementar uma *thread* em três modelos diferentes: *User-level threading* (N:1), *Kernel-level threading* (1:1) e *Hybrid threading* (N:M) [12].

N:1: Neste modelo o sistema entende as *threads* como um único processo, associadas ao processo pai e recebem o nome de *threads* usuário. Este modelo é bastante útil em aplicações que necessitam de muitos fluxos independentes mas que não necessite de um alto poder de processamento. Bastante utilizado em servidores web para gerenciar as conexões ou mesmo em gerenciadores de *downloads* para trabalhar com múltiplos *downloads*, por exemplo. Contudo sua desvantagens está no fato de que o sistema operacional trata todos esses fluxos como sendo um único processo e, por consequência, recebem o mesmo tempo de processamento que outros processos com apenas um fluxo.

1:1: Este modelo trata cada *thread* com um processo separado fazendo uma melhor divisão do tempo de processamento com a carga de cada *thread* que recebe o nome de *thread* sistema. Entretanto torna-se pouco escalável, pois um elevado nível de processos pode sobrecarregar o sistema operacional.

N:M Adotado pela maioria dos sistemas operacionais modernos, este modelo trata-se de um híbrido dos outros dois. Onde *N threads* são mapeadas para *M* processos. Podemos dizer que as *threads* de usuários são divididas e encapsuladas em *threads* de sistema. Desta forma é possível aproveitar benefícios de ambos os modelos anteriores, porém este modelo possuí uma alta complexidade de implementação e um alto custo de gerenciamento de *threads*.

A seguir, na Tabela 2.3 encontra-se de forma resumida os modelos de *threads*, baseado na Tabela de Maziero [12, p. 21]:

Tabela 2.3 – Quadro comparativo dos modelos de *threads* .

Modelo	**N:1**	**1:1**	**N:M**
Resumo	Todos os N *threads* são mapeados em um único processo	Cada *thread* gerada recebe um espaço específico no núcleo	Os N *threads* de usuários são agrupados em M *threads* de sistema
Complexidade	Baixa	Média	Alta
Custo de gerência para o núcleo	Nenhum	Médio	Alto
Escalabilidade	Alta	Baixa	Alta
Divisão de recursos entre tarefas	Injusta	Justa	Depende

2.3.2 Programação Entre-nós

Neste nível, se faz uso de mensagens entre os nós para troca de informações, onde um nó pode enviar ou receber dados. Essa comunicação pode se dar de duas formas:

Síncrona: Um processo que deseja enviar dados a outro nó, envia uma mensagem ao seu destinatário e se bloqueia, permanecendo neste estado até receber uma mensagem de confirmação de recebimento. O mesmo ocorre com um processo que está esperando por dados, ficando bloqueado até recebe-los. Nesta forma deve-se ter o controle do tempo sobre os eventos ocorridos, a fim de gerenciar o envio das mensagens, para garantir um alto grau de sincronismo entre os nós evitando ao máximo a ociosidade de nós a espera de mensagens. Com este gerenciamento de tempo ainda é possível exercer um controle de falhas mais facilmente, visto que, sabe-se o tempo de execução de cada evento. Porém, um mal gerenciamento pode gerar *deadlocks* ou um grande desperdício de tempo, com máquinas ociosas a espera de mensagens.

Assíncrona: Nesta forma, o processo que envia a mensagem não é bloqueado e continua sua execução sem esperar uma mensagem de confirmação. Já o nó destinatário recebe a mensagem e armazena em um *buffer*[1], onde é consumido quando o processo destino está pronto para receber a mensagem. Assim o processo destino não precisa se bloquear até receber a mensagem, já que ela encontra-se no *buffer*. Nesta abordagem praticamente elimina-se a possibilidade de *deadlocks*[2] e reduz o *overhead*[3]. Porém, faz-se necessário um maior controle na troca de

[1] Buffer é um espaço de memória temporária utilizada para escrita e leitura de dados

[2] DeadLock caracteriza uma situação em que ocorre um impasse entre dois ou mais processos que ficam impedidos de continuar suas execuções.

[3] Overhead significa o tempo, espaço em disco ou processamento em excesso que o controle de uma aplicação utiliza para realizar uma determinada tarefa.

mensagens, para evitar a perda das mensagens e garantir que ela já esteja no *buffer* quando o processo consumidor precisar.

2.3.3 Classes de Aplicações Paralelas

Muitas das aplicações que fazem uso da infraestrutura dos *grids* possuem características e requisitos semelhantes quando abordadas pela sua estrutura de execução. Com o agrupamento destas semelhanças foram criadas classes de aplicações de acordo com a estrutura das aplicações. Tendo em mente essas classes, foram desenvolvidos escalonadores especializados para os principais tipos de aplicações: *parameter sweep*, BoT(*bag-of-tasks*) e *workflow* [13]. Abaixo temos a descrição dessas classes de acordo com Bernardi [14].

Parameter Sweep: Nesta classe as tarefas distribuídas são idênticas do ponto de vista do tipo de processamento realizado, porém a diferença está nos parâmetros destas tarefas, que mudam de tarefa para tarefa. Além disso essas tarefas são sequências e independentes, não existindo nenhum tipo de comunicação entre elas.

Bag-of-Tasks: Assim como as aplicações da classe *parameter sweep*, as aplicações *bag-of-task* são compostas por tarefas independentes e sem comunicação entre elas. Porém estas podem executar tarefas de processamentos totalmente diferentes, ao contrário das aplicações do tipo *parameter sweep*, que alteram apenas os parâmetros de entrada.

Workflow: Nesta classe encontram-se aplicações compostas por uma coleção de tarefas que obrigatoriamente devem seguir uma sequência de execução ordenada por dependências de controle e de dados.

Esses modelos de aplicações podem se implementados por diversos modelos de programação. Sendo o modelo conhecido como *master-slave* o mais utilizado. Neste modelo, *master-slave*, uma tarefa mestre delega a outras tarefas escravas o processamento a ser realizado. Este processamento costuma ser o mesmo para todas as tarefas escravas, estas que ao término do processamento retornam o resultado a tarefa mestre. Assim o processamento da aplicação é controlado pela tarefa mestre que cria e comanda as tarefas escravas e ao final recolhe os resultados.

2.3.4 Ferramentas

Com a disseminação da programação paralela, surgiram ferramentas para agilizar e organizar a construção de programas paralelos. Normalmente na forma de APIs ou bibliotecas, essas ferramentas procuram facilitar a programação paralela através da chamadas de métodos, ou até mesmo, classes especializadas na construção de múltiplos fluxos de processamento.

2.3.4.1 MPI

O MPI trata-se de um padrão desenvolvido através do esforço conjunto de um grupo de empresas que comercializam produtos de alto desempenho (HPC), junto de universidades e centros de pesquisa que utilizam HPC. É um padrão de troca de mensagens onde cada fabricante de *hardware* está livre para implementar as suas próprias rotinas utilizando características exclusivas do seu produto [15]. Por este motivo é bastante utilizado na programação entre-nós (veja Seção 2.3.2).

Apesar de muitas empresas fornecerem suas próprias versões comerciais, o padrão MPI encontra-se disponível gratuitamente na Internet. Atualmente na versão 2.2, a biblioteca é disponibilizada para as linguagens C, C++ e Fortran.

2.3.4.2 PThreads

Pthreads, ou ainda POSIX Threads, é um padrão POSIX para *threads*, um conjunto de APIs em C/C++ que permite desenvolver aplicaçõs com vários fluxos para programação intra-nó (veja Seção 2.3.1) baseado em *threads* [16]. Apesar de inicialmente ser escrita unicamente para sistemas *unix*, já existe uma versão para *win32*[4], permitindo que os desenvolvedores que optarem por está biblioteca não fiquem atrelados a sistemas *unix* [17].

2.3.4.3 TBB - Threads Building Blocks

Desenvolvido pela Intel, está biblioteca feita para C++ visa tirar maior proveito dos processadores multi-core produzidos pela Intel [18]. Através de paralelismo baseado em tarefas procura facilitar a vida do programador, abstraindo os detalhes da plataforma e os mecanismos de segmentação na programação intra-nó (veja Seção 2.3.1).

2.3.4.4 OpenMP

É uma API multiplataforma com suporte a memória compartilhada para programação paralela, desenvolvida para C/C++ e Fortran [19]. O OpenMP é sustentado por um grupo de empresas desenvolvedoras de hardware e software, o projeto procura ser portável e escalável, fornecendo aos programadores uma interface simples e flexível para desenvolvimento de aplicações paralelas, que vão desde o *desktop* a supercomputadores.

O OpenMP é composto por: diretivas para o compilador, biblioteca de rotinas de *runtime* e variáveis de ambiente, tornando-se um ambiente completo para programação paralela entre-nós (veja Seção 2.3.2).

[4]Win32 refere-se a versões do sistema Operacional Windows de 32bits

2.3.4.5 Cilk

Trata-se de uma plataforma de desenvolvimento para aplicações paralelas. Uma extensão da linguagem C, que tem como objetivo principal permitir que programadores sem conhecimento em paralelismo possam desenvolver aplicações e algoritmos paralelos. Isso torna-se possível através de marcações inseridas em um algoritmo tradicional, definindo onde os pontos de execução podem ser divididos em diferentes fluxos e de onde a aplicação não pode prosseguir sem que todos os fluxos estejam sincronizados. Dessa forma torna-se uma ferramenta interessante para programação intra-nó (veja Seção 2.3.1), do ponto de vista de programadores que estão acostumados a programação estrutural e desejam começar a paralelizar suas aplicações.

2.3.4.6 CUDA

Arquitetura de computação paralela da NVIDIA, permite um aumento significativo na HPC aproveitando o poder da GPU. Este projeto aposta na premissa de "co-processamento" onde a GPU trabalha em conjunto com a CPU para acelerar o processamento [20]. As novas placas de vídeo da NVIDIA já estão vindo com essa tecnologia e, aliada às ferramentas de desenvolvimento fornecidas pela empresa, prometem ser uma nova força no auxílio pela busca de alto desempenho. Como este paralelismo é feito na GPU juntamente com a CPU, ou seja na mesma máquina, é considerado uma programação intra-nó (veja Seção 2.3.1)

2.4 *Middlewares* para Computação Distribuída

Middleware é uma camada de software composta por diversas ferramentas que possibilitam um grupo de computadores unidos formarem um ambiente único, disponibilizando recursos computacionais de forma transparente aos seus utilizadores. Através do *middleware* os usuários têm acesso as ferramentas para controle de segurança e tolerância a falhas, informações de desempenho, monitoração de recursos, submissão de processo e alocação dos mesmos. O *middleware* é responsável pela autenticação dos usuários, autorizações de acesso, execução de aplicações, entrada e saída de dados, localização de recursos e balanceamento de carga [21].

2.4.1 *Middlewares* para Clusters

2.4.1.1 Beowulf

De acordo com o próprio site [22], temos a seguinte definição: "Os *clusters* Beowulf são *clusters* de desempenho escalável baseados em uma infraestrutura de hardware, uma rede do sistema privado e software de código aberto". O desempenho é melhorado proporcionalmente à inclusão de novas máquinas ao *cluster*. O *cluster* Beowulf pode trabalhar com no mínimo dois nós de forma independente, unidos através de uma rede comum, até um complexo aglomerado de mais de mil computadores.

Algumas das características do Beowulf são: executar somente sobre sistema operacional Linux; os nós devem ser preferencialmente iguais, de modo a tornar o sistema o mais homogêneo possível; o nó mestre pode ser utilizado como *gateway* para Internet, podendo trabalhar como servidor de armazenamento e monitor do sistema.

Os usos mais comuns são as tradicionais aplicações técnicas, tais como simulações, biotecnologia e *petro-clusters*, modelagem do mercado financeiro, mineração de dados e processamento de fluxo e servidores de Internet para jogos.

Os programas para este *middleware* são geralmente escritos usando linguagem C e Fortran e utilizando-se de trocas de mensagens para comunicação entre os nós.

2.4.1.2 openMosix

O openMosix é um projeto de código-fonte aberto, dito como um *fork* do projeto MOSIX da Hebrew University. Trata-se de uma extensão do *kernel* do *linux*, que possibilita gerenciar sistemas distribuídos de forma transparente, fornecendo ao usuário a imagem de um sistema único [23].

Dentre as suas características podemos destacar: a escalabilidade - que permite adicionar novos nós ao *cluster* de forma rápida de transparente, adaptatividade - que permite a inserção de computadores de arquiteturas diferentes e a não necessidade de recopilar aplicações - permitindo que qualquer aplicação se beneficie do *cluster*, sem a necessidade de alteração do código.

Apesar da grande comunidade de usuários, o openMosix foi encerrando em março de 2008 [24]. Segundo Moshe Bar, fundador do openMosix, o motivo pelo término do projeto se da pelo fato do rápido desenvolvimento das tecnologias de processamento, tornando obsoleto a tecnologia de *clustering*. Atualmente ele está envolvido em projetos de virtualização, como por exemplo o Xen[5]

2.4.2 *Middlewares* de Grids

Esta Seção descreve os *middlewares* de *Grid* selecionados para o estudo, trazendo uma abordagem imparcial de cada um, com suas principais características, arquitetura, usabilidade e peculiaridades.

2.4.2.1 Globus

Globus é um projeto de código-fonte livre, iniciando por volta de 1998 pela Globus *Alliance* [25], coordenado por Ian Foster e Carl Kesselman. Atualmente o site apresenta duas estratégias: Globus Online e Globus Toolkit. A primeira oferece uma infraestrutura pronta para o usuário, sustentada na nuvem, fornecida pela Globus *Alliance*, onde o usuário unicamente faz uso dos recursos oferecidos. Já o Globus Toolkit segue a linha do projeto original, fornecendo um *middleware* para construir *grids* customizadas. O GT, como é chamado no site dos desenvolvedores, é o elemento

[5]Xen é um projeto open-source permite a execução de vários sistemas operacionais, simultaneamente, sobre um mesmo hardware, através de tecnologias de virtualização.

chave que define os requisitos básicos exigidos para a construção de *grids* computacionais sugerido pelo projeto Globus. Segundo Bernardi [14], a principal característica do Globus é sua modularidade, que permite instalar e configurar o GT somente com os recursos que serão realmente utilizados. Ainda é importante salientar que o GT não engloba todos os projetos desenvolvido pelo Globus, mas sim um conjunto de ferramentas produzidas por alguns de seus projetos.

1. Aplicabilidade:

   O GT fornece um kit de componentes que visam fornecer as principais necessidades para um construção de um *grid* [26], entre eles podemos destacar:

   (a) Implementação de serviços de gerência de recursos, transferência de dados e descoberta de serviços;

   (b) Ferramenta para construção de *Web Services* em java, C e Python;

   (c) Infraestrutura de segurança baseada em padrões de autenticação e autorização;

   (d) APIs e *scripts* para acessar os serviços fornecidos pelo *grid*;

   (e) Documentação detalhada com especificações dos componentes, interfaces e exemplos de utilização.

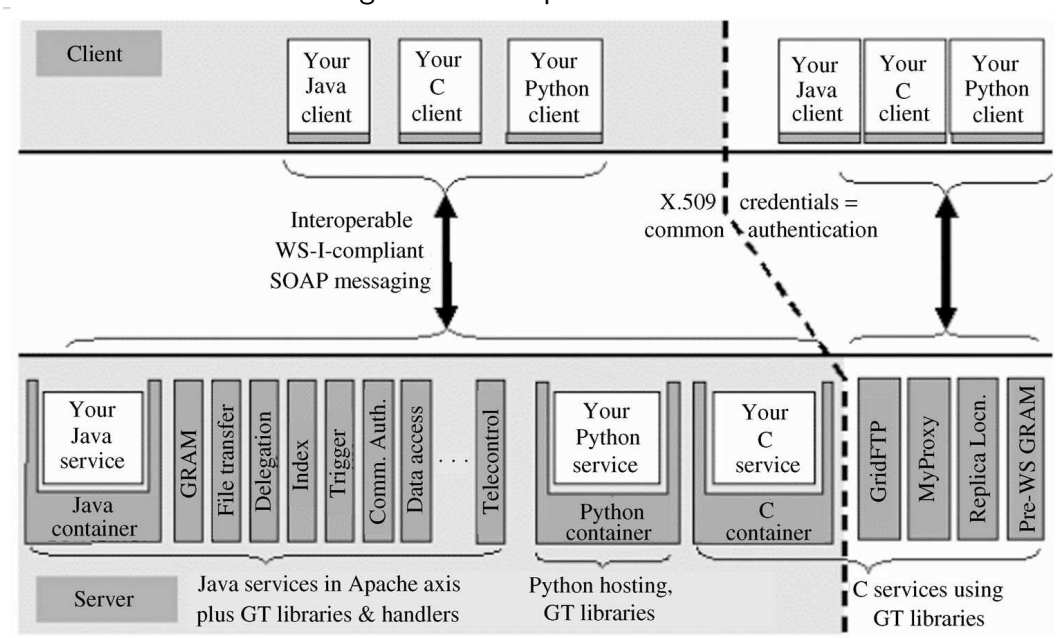
2. Arquitetura:

   Com exceção de alguns serviços que exigem um maior controle em níveis mais baixos da programação, aonde são desenvolvidos em C, o Globus possuí sua estrutura de de serviços implementada em Java. Para os serviços desenvolvidos pelos usuário, o GT se utiliza de uma estrutura baseada em *containers*, juntamente com um conjunto de bibliotecas de programação. Essa estrutura fornece todas as funcionalidades necessárias para hospedar uma aplicação personalizada e estão separados de acordo com a linguagem suportada: Java, C e Python. Os serviços personalizados, assim como os fornecidos pelo GT, ficam disponíveis aos seus consumidores através do protocolo SOAP(Simple Object Access Protocol) [26]. A arquitetura do Globus é ilustrada na Figura 2.8 [26, p. 514].

   Foster [26], ainda faz uma segunda abordagem da arquitetura do Globus, dividindo os serviços em cinco categorias: Segurança, Gerenciamento de dados, Gerenciamento de execuções, Serviços de informações e os *containers* (na Figura 2.9 [26, p. 515] chamado de *commom runtimes*). Ainda nesta visão, aparecem alguns componentes marcados como "*tech preview*", que correspondem aos componentes relativamente novos e que ainda estão em fase de testes, podendo sofrer modificações em versões futuras.

   ***Execution Management*:** corresponde aos componentes destinados ao controle do *grid*, dentre eles podemos destacar o *Grid Resource Allocation and Management*(GRAM). A partir de um *web service*, este componente permite gerenciar as tarefas de forma

Figura 2.8 – Arquitetura GT4



detalhada, podendo configurar e monitorar os recursos necessários para execução de cada tarefa, de forma a dar controle total ao usuário. Este controle pode incluir definição de recursos a serem utilizados, quantidade de dados que devem ser armazenados durante e após a execução, controle de acesso a tarefa, definição dos executáveis e seus parâmetros de inicialização, controle de acesso, entre outros.

O *Workspace Management Service* fornece controle para a criação de *sandboxes* através de máquinas virtuais ou contas Unix. Já o *Grid TeleControl Protocol*(GTCP) permite gerenciar os instrumentos conectados ao *grid*, como por exemplo, um microscópio.

***Data Management*:** contém o GridFTP, desenvolvido para prover funcionalidades de transferência de dados, que aliado ao *Reliable File Transfer*(RFT) permite realizar múltiplas transferências. Possuí o *Replica Location Service*(RLS), utilizado para ter acesso aos dados replicados no *grid*. O Data Replication Service(DRS) aliado ao RLS e o GridFTP prove os serviços necessários para gerenciar a replicação dos dados. E ainda o *Data Access and Integration* que fornece ferramentas para acessar e processar dados relacionais e estruturados.

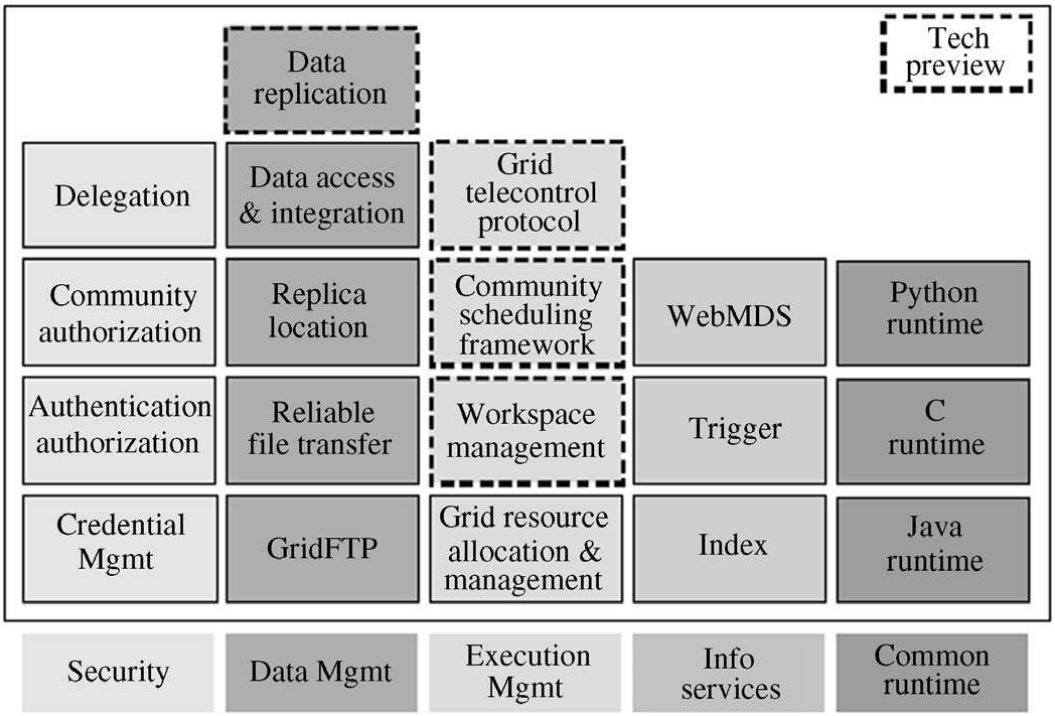
***Information Services*:** são os serviços responsáveis por monitorar e descobrir os serviços existentes no *grid*, implementando mecanismos que traduzem os serviços existentes em linguagem XML. Estes mecanismos implementam as especificações do WSRF e *WS-Notifications* em todos os serviços fornecidos pelo *grid*, incluindo os serviços desenvolvidos pelo usuário. Para os serviços que não possuem suporte WSRF ou *WS-Notifications* o Globus possui dois serviços agregadores: um de registro (Index) e um monitor de eventos (Trigger) que se encarregam de reconhecer os serviços disponíveis restantes. Já o

WebMDS através de XSLT permite criar *views* especializada dos XMLs gerados, entre outra gama de ferramentas que permitem aos usuários acessar e pesquisar os recursos disponíveis.

***Security*:** A segurança do Globus pode ser feita através de autorizações do tipo usuário/senha, mas por padrão utiliza credenciais com certificado X.509, com o MyProxy[6] como gerenciador das *public keys*. Através de protocolos, duas entidades podem validar as credenciais para que possam ser usadas no estabelecimento de um canal seguro entre as partes. Dessa forma os usuários poderão usufruir do *grid* tendo a segurança tratada de forma transparente pelo GSI através dos certificados.

***Common Runtime*:** responsável por fornecer uma estrutura completa para o desenvolvimento de aplicações personalizadas, aqui encontram-se os *containers* de programação. Fornecendo a estrutura necessária para desenvolver aplicativos que implementam uma interface de *web service*. Como mencionado anteriormente, são três *containers*, separados de acordo com a linguagem suportada. Todos implementam as implementações básicas de um *web service*, junto com diretivas de segurança e a combinação dos demais serviços fornecidos pelo *grid*.

Figura 2.9 – Arquitetura GT4



[6]MyProxy é um projeto desenvolvido pela Globus que combina um repositório de credenciais online com uma autoridade de certificação online para permitir aos usuários obter credenciais de forma segura

3. Instalação:

   Segundo Gláucio Souza em [27], para implantação do *middleware* se faz necessário uma série de etapas de configuração, que estão resumidas abaixo.

   Como pré-requisitos para instalação do Globus estão incluídos uma lista de pacotes de copiladores, bibliotecas e configurações de variáveis de ambiente necessárias para sua instalação, juntamente com um SGBD e seus respectivos *drivers*.

   Após configurar os usuários UNIX no ambiente, deve-se compilar o Globus e adicionar novas variáveis de ambiente que serão utilizadas pelo Globus. Ao término da compilação deve-se configurar o sistema de autenticação, utilizando um CA existente, ou em caso de testes, utilizar o *simpleCA*. Independente da escolha feita a partir dai inicia-se um processo de configuração de permissões e acessos através de certificados que serão utilizados pelos usuários para terem acesso aos serviços do *grid*.

   Também é preciso configurar cada serviço que será disponibilizado pelo *grid*: GridFTP, RTS,RFT, os *containers* e demais serviços. Ao término desta etapa realiza-se alterações no comando "sudo" para que o GRAM possa funcionar corretamente.

   São feitas configurações do MyProxy, como por exemplo coloca-lo na inicialização do sistema e configura-lo apara executar a partir de um *host* seguro. Além de realizar a instalação dos serviços responsáveis pelas autorizações dos usuários (CAS) e configura-los para interagir com o banco de dados escolhido.

4. Envio de Tarefas:

   É possível enviar uma tarefa de duas maneiras: primeira, para tarefas mais simples, utiliza-se o comando "globusrun-ws -submit -c" seguido pelo caminho da aplicação; segunda, com a possibilidade de enviar descrições, necessita de um arquivo XML que contém as informações da tarefa e utilizar o comando "globusrun-ws -submit -f" seguido pelo caminho do arquivo XML.

5. Considerações:

   O Globus é um projeto desenvolvido por algumas das principais universidades do mundo e conta com o apoio de diversas empresas consagradas na área de TI. Servindo de referencia quando o assunto trata de *grid* computacional, sua solução para sistemas distribuídos, o Globus Toolkit, provê um leque de funcionalidades que permite a implantação e o desenvolvimento de sistemas em *grid* [28].

   A abordagem do GT foi muito interessante, pois possibilita a instalação e configuração, somente dos serviços desejados, além da possibilidade incremental de novos recursos ao *grid*. A partir da versão 3.x e especialmente na versão 4 do GT sua estrutura foi voltada para fornecer

seus recursos em forma de *web-services*, adotando a ideia de "software for service-oriented systems" [26].

Porém sua implantação é a mais complexa e onerosa quando comparado a outras soluções aqui apresentadas, onde se faz necessário conhecimentos de nível avançado de computadores, conhecimentos de sistemas *UNIX* e de arquiteturas paralelas. Outro ponto relevante é sua incompatibilidade com ambientes Windows, dificultando sua adoção em cenários onde este SO domina. Mas o ponto mais crítico, está na necessidade de desenvolvedores com profundos conhecimentos em sistemas distribuídos e no próprio GT, para que possa ser aproveitado seu potencial, já que em sua versão *standard* não foi previsto um escalonador automático de tarefas, tendo que partir para soluções de terceiros, ou mesmo, passando este trabalho ao recursos humanos envolvidos.

Dessa forma é dedutível que o a solução proposta pelo projeto Globus, o Globus Toolkit, é uma escolha adequada para aplicar em grandes empresas ou organizações que necessitam de um sistema robusto e que, ao mesmo tempo, dê a possibilidade de total controle do *grid*.

2.4.2.2 OurGrid

OurGrid é um *middleware* de *grid peer-to-peer* [7] que dá suporte à execução de aplicações da classe BoT 2.3.3. O projeto é desenvolvido em JAVA e Shell Script, possuindo uma interface WEB. É mantido pelo Laboratório de Sistemas distribuídos da Universidade Federal de Campina Grande (UFCG) desde 2004.

A comunidade do OurGrid destaca-se pela grande participação de seus usuários e a rápida aceitação de novos membros, interessados em participar do projeto. Com um site bem estruturado e uma área especial para os desenvolvedores a comunidade evolui rapidamente, tornando o projeto mais robusto e atualizado. Além da interação através do fórum os organizadores da comunidade realizam encontros presencias entre os participantes, tornando a comunidade ainda mais social [29].

1. Aplicabilidade:

   O Ourgrid foi desenvolvido especificamente para resolver problemas computacionais que exigem alto poder de processamento e que possam ser tratados como aplicações BoT. O *middleware* foi desenvolvido em JAVA, o que possibilita que qualquer um dos componentes, seja para solicitar tarefas, seja para realizá-las, possar ser executado em diferentes SOs. Sendo assim, é possível montar um *grid* composto de máquinas com diferentes SOs, facilitando a implantação em ambientes heterogêneos.

2. Arquitetura:

[7] Peer-to-peer é um modelo de comunicação onde cada nó pode exercer funções de cliente e servidor.

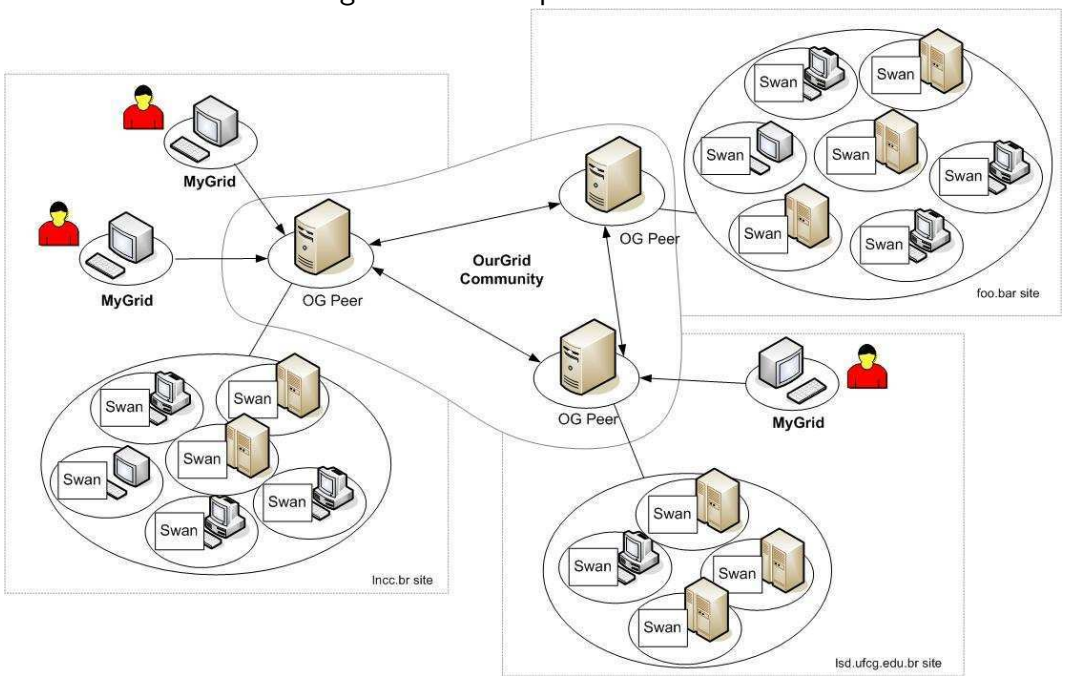
O Ourgrid é formado basicamente por três principais componentes: OurGrid *peer*, MyGrid *broker* e pelo *worker* que contém um SWAN (Sandboxing Without A Name). Um *grid* com o *middleware* OurGrid pode ser formado por um MyGrid *broker*, um OurGrid *peer* e diversos computadores com SWAN instalados. Esta composição é chamada de *site*, vários *sites* conectados formam uma comunidade. Podemos visualizar essa arquitetura na Figura 2.10 [30, p. 6]

MyGrid broker: provê uma *interface* web amigável para que o usuário possa submeter remotamente os *jobs* para o *grid*, além de fazer o escalonamento dos mesmos.

OurGrid peer: é o componente responsável por gerir os *workers* de um determinado *site*. Ele recebe as tarefas e faz o levantamento dos recursos que serão necessários para execução, caso não tenha recursos suficientes disponíveis pede ajuda a outro OurGrid *peer* da comunidade.

Workers: são os computadores responsáveis por efetivamente realizar as tarefas, neles está instalado o SWAN, uma máquina virtuai Xen com o acesso a rede desabilitado, que é responsável por executar, de forma segura para o *worker*, as solicitações vindas do seu OurGrid *peer*. Esta abordagem de *sandbox* foi adotada para garantir que os *workers* não sejam danificados por possíveis códigos maliciosos oriundos das tarefas. Isso poderia acontecer em momentos que eles executam tarefas solicitadas por OurGrid *peers* pertencentes a outros *sites* não confiáveis. Do ponto de vista de quem envia as tarefas o SWAN implementa um sistema de *sanity check* que verifica se a máquina virtual não foi alterada, garantindo a autenticidade e segurança dos resultados ao OurGrid *peer* [30].

Figura 2.10 – Arquitetura OurGrid



É importante salientar que a decisão de qual solicitação o OurGrid *peer* irá atender primeiro é baseada em um sistema de reputação [30]. Primeiramente ele procura atender todas as solicitações vindas diretamente do MyGrid *broker* próximo a ele, caso ainda existam recursos disponíveis, passa a atender as solicitações oriundas de outros Ourgrid *peers*. A partir dai é adotado um sistema chamado de *Network of Favours*, aonde visa dar prioridade aos Ourgrid *peers* que mais colaboram com a comunidade.

3. Instalação:

   As informações a seguir descrevem o processo de instalação do OurGrid, baseadas no documento de Mantovani,Junior e Rodrigues [31].

   Antes de instalar o OurGrid é necessário a instalação do *OpenFire*[8], pois é a partir dele que serão gerenciados os usuários do *grid*. Após o servidor estar devidamente configurado o próximo passo é cadastrar os computadores integrantes do *grid*, aonde cada computador seja *worker*,*peer* ou *broker* deve ser cadastrado como um usuário no servidor *OpenFire* .

   Para os usuário que desejam fazer uso dos recursos do *grid* é necessário instalar o *broker* em sua máquina local. Para isso, basta descompactar e executar o binário. Existem duas formas de trabalhar com o *broker*: executando em modo texto ou utilizar sua interface gráfica (GUI).

   Por fim é é preciso cadastrar os *peers* no *broker*, criando-se um arquivo texto com as informações dos *peers* e salvando-o com a extensão ".gdf". Após, carrega-se este arquivo no *broker*, seja por linha de comando ou pela interface gráfica. Ao final destes passo já é possível enviar tarefas ao *grid*, utilizando o *broker*.

   A instalação dos *peers* também se da por linha de comando ou interface gráfica, onde devem ser informados os campos de usuário e senha previamente cadastrados no servidor XMPP, além do endereço do servidor XMPP. Ainda é preciso cadastrar os usuários que poderão utilizar o *grid*, mesmo que já cadastrados no servidor XMPP. Então cadastra-se os *workers* que o *peer* irá gerênciar, utilizando-se um arquivo texto de extensão ".sdf" que conterá a descrição de todos os *workers*.

   O último passo é configurar os *workers*, também é preciso informar o usuário, senha e o endereço do servidor XMPP, além da chave pública do *peer* em que o *worker* será cadastrado.

4. Envio de Tarefas:

   Para submeter uma tarefa deve-se criar um arquivo texto do tipo ".jdf" com as informações da mesma. Este arquivo deve ser enviado para o *grid* através do *broker*, por linha de comando ou pela interface gráfica do mesmo [31].

[8] Openfire é um servidor de colaboração em tempo real de código-fonte aberto, que utiliza o XMPP como protocolo de transferência de mensagens instantâneas.

Neste arquivo de submissão temos os conceitos de *job* e *task*, o primeiro especifica atributos comuns de todo o trabalho, já o segundo especifica atributos particulares e comandos para as tarefas que compõem o aplicativo paralelo [32].

A cláusula *job* pode conter especificações do *worker*, como: quantidade de memória e SO necessário para executar o *job*. Já a cláusula *task* é composta por três sub-cláusulas: *init* que define quais arquivos serão transferidos para o *worker*, *final* que corresponde os arquivos que deverão ser retornados ao final da tarefa e o *remote* que contém a linha de comando a ser executada para iniciar *job*.

5. Considerações:

   O OurGrid foi projetado para resolver o problema de aplicações BoT e cumpre bem o seu propósito. Sua arquitetura é totalmente voltada para prover um maior desempenho e ao mesmo tempo segurança aos computadores envolvidos no *grid*. A possibilidade de compor o *grid* por computadores de SOs mistos é um ponto importante, visto que, nos dias atuais os ambientes computacionais estão cada vez mais heterogêneos.

   Outro ponto interessante é o emprego da tecnologia *peer-to-peer*, onde os *sites* que mais colaboram com a comunidade,serão privilegiados na distribuição dos recursos, incentivando o compartilhamento dos mesmos. Essa ideia pode ser aplicada tanto a uma comunidade privada, entre instituições, conglomerados de empresas, centros de pesquisas, como em uma escala global, aonde todos tem acesso e disponibilizam recursos através do *grid*.

   O OurGrid é uma excelente escolha quando as necessidades de uma determinada entidade estão limitado unicamente a um *computational grid* (veja a Subseção 2.2.3) de computadores com dedicação exclusiva, podendo ainda, fazer uso dos privilégios na distribuição de recursos para incentivar o compartilhamento dos mesmos.

2.4.2.3 UNICORE

UNICORE (*UNiform Interface to COmputer REsources*) foi fundada em 1997 por *German Ministry for Education and Research* (BMBF) . Consiste em um *middleware* de código-fonte aberto desenvolvido conforme os padrões definidos pela comunidade OFG (*Open Grid Forum*). Concebido com os princípios de SOA (*Service-Oriented Architecture*) e desenvolvido em JAVA, procura ser de fácil extensão e permitir a interoperabilidade com outras tecnologias de *grid* [33].

Possuí um site bem estruturado e com documentação de fácil acesso, manuais de utilização e instalação, tanto da parte do cliente quanto do servidor, são encontrados facilmente no site. Promovem uma reunião anual para usuários desenvolvedores e pesquisadores que trabalham com o UNICORE para compartilhar experiências, assistem palestras e discutem o futuro do projeto.

1. Aplicabilidade:

   Projetado para funcionar nos principais SOs da atualidade, o *middleware* pode ser facilmente instalado em qualquer plataforma graças ao seu prático instalador, tornando possível sua implantação em ambientes heterogêneos. Outro ponto interessante é o fato de ter seu código fonte aberto e desenvolvido sobre SOA, permitindo ter uma forma de entendimento e extensão de seus recursos.

   Seu cliente fornece uma maneira atrativa para trabalhar com aplicações da classe *workflow* (veja a Subseção 2.3.3), sendo seu principal foco, mas também é possível utilizá-lo para aplicações BoT. Sua principal utilização está em *grids* de âmbito científico, amplamente utilizado nesta área por toda a Europa.

2. Arquitetura:

   Sua versão mais atual, a UNICORE6, tem a arquitetura dividida em três camadas: *client layer*, responsável por fornecer uma interface de acesso a usuários; *service layer*, que contém todos os serviços e componentes fornecidos pelo *grid*; e *system layer*, responsável por fazer a ligação do *middleware* com cada máquina do grid. A seguir está descrito a arquitetura do *middleware* conforme o próprio site [33].

   ***Client Layer*:** Esta é a camada do topo, aonde se encontram as formas de acesso ao *grid*. O UNICORE *command line client* (UCC) que fornece uma interface de linha de comando para a submissão e monitoramento das tarefas, bem como recuperar os resultados obtidos pela execução das tarefas. UNICORE *Rich Client* (URC) faz o mesmo que o UCC, porem este tem uma interface gráfica desenvolvida em cima do Eclipse [9]. Já o HiLA (*High Level API for Grid Applications*) fornece uma API completa para os desenvolvedores que desejam criar um cliente personalizado para acesso ao *grid*. Ainda é possível utilizar ferramentas de terceiros para ter acesso aos serviços do *grid*, como por exemplo o GridSphere [10].

   ***Service Layer*:** Está é a camada do meio, contendo todos os serviços e componentes do *grid*:

   - O componente *Gateway* atua como ponto de entrada para o *grid* e realizando as autenticações de todas as solicitações de entrada.
   - O XNJS é o principal componente do *middleware*, oferece recursos de armazenamento, serviços de transferência de arquivos e serviços de gerenciamento de tarefas.

[9] Eclipse é uma comunidade de código-fonte aberto, cujos projetos estão focados na construção de uma plataforma de desenvolvimento aberta e extensível [34]

[10] GridSphere é um *framework* para o desenvolvimendo te portais para Internet que interagem com ambientes de *grid*

- O IDB (*Incarnation Database*) realiza o mapeamento dos arquivos JSDL[11] para um levantamento concreto dos recursos do *grid*. Nesta base ainda constam informações sobre os aplicativos e características dos recursos disponíveis.
- As autorizações de usuários são feitas através de certificados X509, mapeados para *login* e senha através do banco XUUDB. Porém essas autorizações podem ser feitas através de certificados de *proxy* ou ainda utilizando o UVOS, um serviço de *Virtual Organization* que pode ser usado para autenticações.
- Os serviços fornecidos pelo *grid* são registrados de forma centralizada através de um único componente, o *registry* e suas informações são fornecidas através do *Common Information Service* (CIS) , que pode fornecer essas informações na forma de um arquivo XML ou em uma versão texto para fácil leitura. Ainda informações de posicionamento geográfico da infraestrutura são fornecidas de forma visual com o auxílio do google maps.
- O UNICORE dá suporte para aplicações da classe *workflow* através de uma arquitetura de duas camadas. Uma das camadas denominada *Workflow Engine*, é responsável por gerenciar e coordenar a execução, enquanto a camada *orchestrator* é responsável por executar as tarefas individuais. Ainda é possível alterar o componente responsável pela execução por outro completamente diferente sem grandes modificações.

***system layer*:** A última camada do *middleware* que faz a comunicação direta com a máquina, para isso ela utiliza o componente TSI que traduz os comandos do *grid* para comando específicos do SO da máquina em que está instalado. Ainda nesta camada encontram-se o USpaces, que é responsável por fornecer um diretório específico para que o *middleware* possa trabalhar com os *jobs*. Neste diretório são armazenados os dados de entrada, arquivos temporários e dados de saídas gerados pelos *jobs* executados na máquina. Por fim o *External Storage* é utilizado para para transferir dados através da rede.

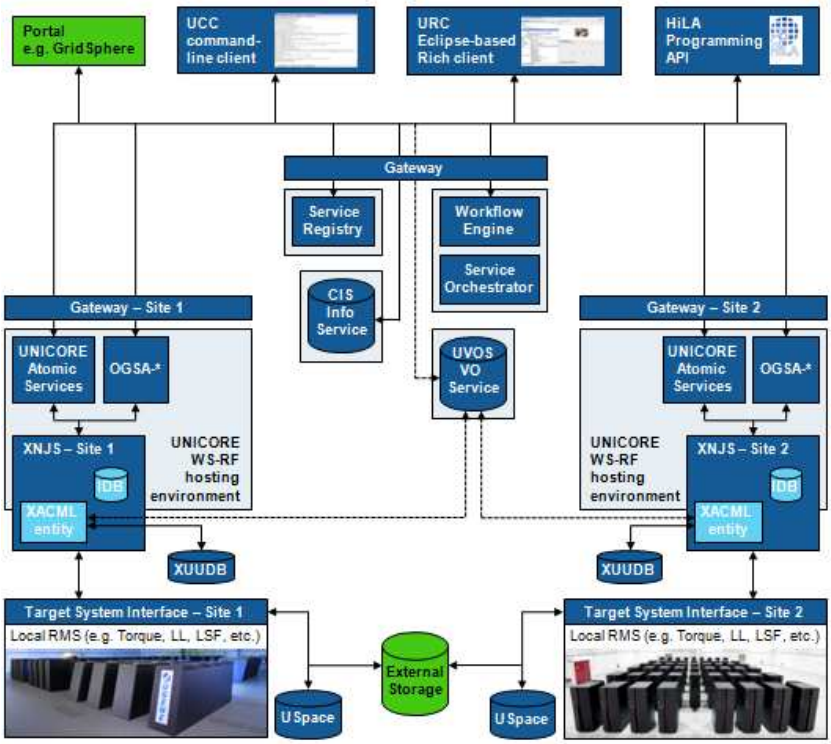
A seguir a Figura 2.11 exemplifica a arquitetura do UNICORE com dois *sites* e os serviços centrais agrupados abaixo do *Gateway* [33].

3. Instalação:

   Do ponto de vista do cliente o UNICORE oferece a escolha para *download* das versões URC, UCC e HiLA separadamente. O URC pode ser baixado na forma de um arquivo binário de extensão "jar" que possuí um instalador intuitivo e simples, ou ainda pode ser instalado no Eclipse como um *plugin*. Já o UCC esta disponível em pacotes binários do tipo "deb" e "rpm" enquanto o HiLA pode ser baixado no formato "deb" e "jar". Em ambos os casos é necessário identificar a localização do *keystore*[12].

[11] JSDL é usado para descrever os requisitos de recursos que um *job* precisa para ser submetido à execução [35]
[12] Keystore é um banco com as chaves privadas relacionadas aos seus certificados.

Figura 2.11 – Arquitetura UNICORE



O *server* também pode ser instalado através de um binário "jar" sem muito esforço, ao final da instalação terá o servidor rodando com os serviços básicos para o funcionamento de um *site*. Alguns componentes são baixados separadamente e instalados na forma de *plugins*, é o caso do CIS e UVOS, também instalados sem dificuldades.

Um ponto interessante é o fato da página na Internet oferecer um *liveCD* com o UNICORE já instalado e funcionando em uma distribuição Linux. Basta efetuar o *download*, gravar a imagem em um cd e iniciar o computador a partir do CD.

4. Envio de Tarefas:

   Segundo o próprio site do *middleware* [33], como o URC tem como base o Eclipse é possível criar um projeto para cada submissão de tarefa, isso torna-se interessante quando a aplicação é do tipo *workflow* pois é possível montar a sequência de execuções e reutiliza-la para futuras submissões. Não somente para editar e enviar um *workflow*, o URC também permite monitorar a execução do mesmo, de modo a detalhar todo o processo.

   O UCC também permite o envio de aplicações da classe *workflow*, porém o mais recomendado é a utilização do URC para tal. O UCC é uma alternativa para computadores de baixo poder de processamento que trabalham somente por linha de comando. Provê todas as funcionalidades necessárias para submissão de *jobs* por linha de comando, desde a descoberta de *sites* até a recuperação dos resultados.

5. Considerações:

   O UNICORE apresenta uma solução interessante para execução de aplicações da classe *workflow*, com seu URC intuitivo que permite uma fácil estruturação das execuções das aplicações, além da possibilidade de acompanhamento das mesmas. Aliado ao fato de sua fácil instalação e portabilidade torna-se uma boa escolha em ambientes heterogêneos aonde serão executadas aplicações da classe *workflow*.

   Por não suportar a paralelização a nível de código e, consequentemente, a falta de uma API própria para desenvolvimento de aplicações paralelas, torna-se um inconveniente para desenvolvedores mais experientes em programação paralela, que poderão sentir falta de tal recurso. Outro ponto a salientar é o fato do *midleware* não apresenta características oportunistas, sendo necessário uma dedicação quase que total, se não total, das máquinas para o ambiente de HPC.

2.4.3 *Middlewares* Oportunistas

Esta seção descreve os *middlewares* Oportunistas selecionados para o estudo, trazendo uma abordagem imparcial de cada um, com suas principais características, arquitetura, usabilidade e peculiaridades.

2.4.3.1 Alchemi

Alchemi é um *middleware* de *grid* desenvolvido pela *University of Melbourne* e conta com apoio do Governo Australiano e das empresas Microsoft e eWater. Destina-se a aplicações desenvolvidas na plataforma .NET da Microsoft e conta com uma API de desenvolvimento para tal, mas também permite a execução de aplicações desenvolvidas em outras plataformas, nesse caso aplicações da classe *parameter sweep* (veja Subseção 2.3.3).

1. Aplicabilidade:

   O Alchemi foi desenvolvido para executar em ambientes Windows, mas pode ser levado a outros ambiente com a utilização do Mono.[13] Possuí uma API própria para desenvolvimento de aplicações escritas em C#, e ainda fornece ferramentas e componentes para integração com o Visual Studio[14], formando um ambiente rico para desenvolvimento de aplicações paralelas.

   Para aplicações desenvolvidas em outras linguagens, não suportadas pela plataforma .NET, o Alchemi permite a paralelização dessas aplicações desde que pertençam à classe *parameter sweep* (veja Subseção 2.3.3).

[13] Mono é a implementação de código-fonte aberto e multi-plataforma de C# e o CLR, um binário compatível com Microsoft.NET [36]

[14] Visual Studio ou MS Visual Studio é uma ferramenta da Microsoft para desenvolvimento de aplicativos para ambientes Windows

2. Arquitetura:

   A arquitetura do Alchemi se divide em quatro componentes que envolvem o ambiente de *grid*: *Manager*, *Executor*, *Owner* e *Cross-Platform Manager* além das APIs e ferramentas de auxílio ao desenvolvimento.

   ***Manager*:** componente responsável por gerenciar e fornecer serviços que controlam as execuções das *threads*. Os *Executors* se registram no *Manager*, que por sua vez, controla a disponibilidade de cada um e distribui os *threads* recebidos através do *Owners* entre os *Executors* disponíveis.

   ***Executor*:** responsável por efetivamente executar as *thread*, podendo agir de duas formas: dedicado, quando o *Manager* gerencia os recursos fornecidos pelo *Executor* e definindo qual *threads* ele deve processar ou não dedicado, quando o *Executor* passa a trabalhar em momento de ociosidade, com a ativação da proteção de tela, ou através da definição manual do usuário, que escolhe uma tarefa a ser executada que encontra-se disponível um uma lista.

   ***Owner*:** As aplicações criadas com a API do Alchemi são executadas neste componente. É ele quem fornece uma *interface* para que o usuário envie sua aplicação para o *grid* e, posteriormente, receba os resultados. O *Owner* envia *threads* para serem processadas ao *Manager* e coleta as *threads* já processadas.

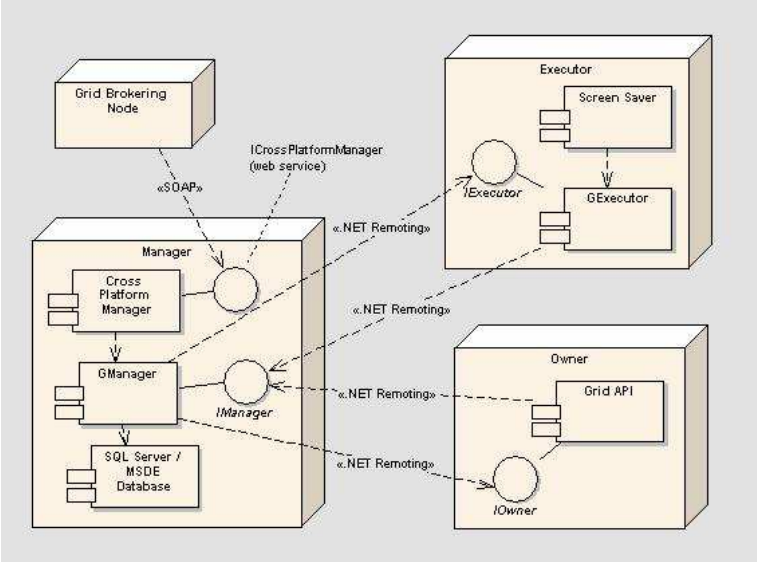
   ***Cross-Platform Manager*:** este é um sub-componente opcional do *Manager*. Trata-se de um *web service* que oferece uma parte das funcionalidade do *Manger*, para permitir que o Alchemi execute *jobs* genéricos, ou seja que não utilizam a API do Alchemi. Os *jobs* enviados são traduzidos para uma forma que o *Manager* compreenda e assim são agendados para execução como os demais.

   A Figura 2.12 [37, p. 3] demonstra um cenário de *grid* com o *middleware*. É possível visualizar o funcionamento do *grid*, onde o *Owner*, com o uso das APIs, envia as tarefas ao *Manager*, que por sua vez encaminha ao *Executor* e, ao receber o resultado, retorna-o ao *Owner* .Também é visível a integração do Alchemi com outros *brokers* através do protocolo SOAP, em que o *broker* se comunica com o *Cross-Platform Manager*, que traduz e envia a solicitação ao *Manager*.

   Além da forma ilustrada na Figura 2.12, que corresponde a um único *cluster*, podemos configurar o Alchemi para um modo *multi-cluster*, conectando os *Managers* de forma hierárquica. Neste cenário todos os *Managers*, exceto o que está no topo da hierarquia, são vistos como *Executors* pelos demais *Managers*.

   Por padrão todos os agendamentos oriundos do *Owner* local recebem a prioridade mais alta, a menos que seja alterada sua prioridade manualmente, enquanto os agendamentos vindos de

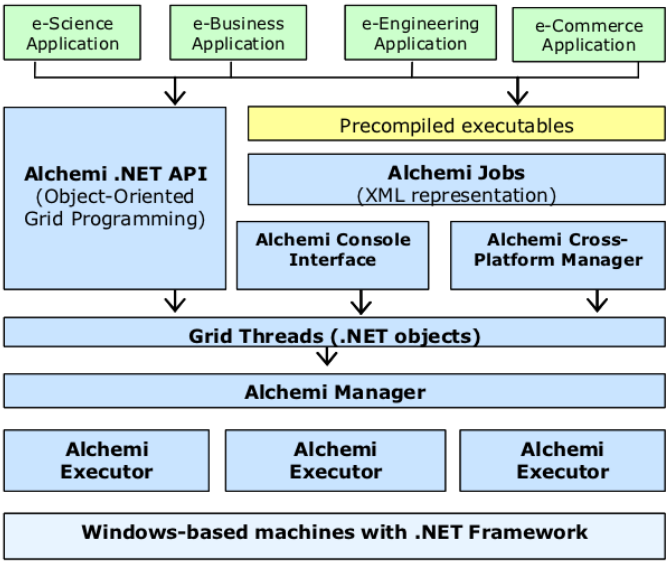
Figura 2.12 – Arquitetura Alchemi



outros *Managers* recebem prioridade normal. Desta forma, um *cluster* que tem seu *Manager* conectado a outro *Manager* de nível mais alto, manterá a prioridade da utilização dos recursos voltada aos agendamentos locais. Assim trabalha de forma a cooperar com outros *clusters* sem comprometer a *performance* para seus usuários locais.

Na Figura 2.13 [38, p. 8] está representado o diagrama de blocos do Alchemi. É possível visualizar o Alchemi *Console Interface*, que se encontra no Alchemi SDK. Através do Alchemi *Console* é possível gerenciar e monitorar o *grid*. Navegando em abas pode-se ter uma visão das estatísticas e informações dos recursos do *grid*, gerenciar os usuários, submeter e monitorar aplicações e *jobs*.

Figura 2.13 – Diagrama de Blocos do Alchemi



3. Instalação:

   Os pré-requisitos para instalar o *Manager* são: SQL *Server* 2000 ou MSDE 2000(Microsoft SQL Server 2000 Desktop Engine) e Microsoft .NET *Framework* 1.1 para ambientes *Windows* ou Mono para ambientes Linux.

   Sua instalação é simples, realizada como a instalação de um programa normal de Microsoft Windows, podendo ser instalado como uma aplicação normal ou um serviço. Será necessário um usuário e senha para conectar ao banco SQL *server* e algumas informações referentes a conexão. Na mesma tela já é possível iniciar o *Manager* clicando no botão "*start*".

   Para o *Cross-Platform Manager* é preciso ter instalado na máquina o Internet *Information Services* (IIS) e ASP.NET. O *Cross-Platform Manager* também é instalado de forma prática assim como o *Manager*, porém se for instalado em outra máquina ou mesmo se as configurações padrões do *Manager* forem alteradas é preciso especificar em um arquivo XML manualmente.

   Os *Executors* são instalados em cada máquina que será utilizada para realizar as tarefas, sua instalação é como o *Manager*, podendo executar como um aplicativo ou serviço. Para que funcione, basta configurar as informações de localização do seu *Manager*,suas credencias e se deve agir de modo dedicado ou não(veja Subseção 2.4.3.1).

   Já o SDK não possui um instalador, mas basta descompactar seu conteúdo no local mais conveniente. Nele esta contido o Alchemi *Console*, o arquivo "Alchemi.Core.dll", usado para criar os aplicativos que serão executados no *grid* e uma pasta de exemplos de aplicações.

4. Envio de Tarefas:

   O Alchemi aceita duas formas de envio de tarefas: através do envio de aplicações da classe BoT, para aplicações desenvolvidas em outras linguagens diferentes das suportadas pela plataforma .NET ou através da API Alchemi, onde é possível paralelizar a aplicação diretamente no código.

   No primeiro caso o envio de tarefas é feito através de linha de comando, onde envia-se um arquivo XML contendo as informações do binário, arquivos de entrada, comando para execução e arquivo de saída dos resultados. Para o segundo caso o Alchemi conta com um ambiente de programação com componentes prontos, que podem ser arrastados para dentro da aplicação, desenvolvido para facilitar o desenvolvimento utilizando Visual Studio.

5. Considerações:

   As ferramentas e componentes que são integrados ao Visual Studio encurtam a curva de aprendizado e adaptação, pois os desenvolvedores que trabalham com esta ferramenta já estão familiarizados com o ambiente de desenvolvimento. Os exemplos que acompanham o *middleware* e a própria API do mesmo - que permita ao desenvolvedor cria suas aplicações sem

a necessidade de utilizar lógicas específicas para ambientes paralelos - são outros facilitadores que colaboram para um rápida adaptação ao ambiente proporcionado.

A possibilidade de paralelizar aplicações do tipo BoT, para os casos em que as aplicações são desenvolvidas em linguagens sem suporte nativo pelo *middleware* e a praticidade da implementação facilitam na escolha para sua adoção. Podendo ser utilizado, até mesmo, levando em consideração somente estas características, pois prove um ambiente paralelo consistente para o trabalho com aplicações BoT e de rápida implementação em ambientes Microsoft Windows.

No entanto ferramentas como o Visual Studio e SQL Server, caso não opte pela versão gratuita, são ferramentas pagas que elevam os custos, forçando muitos desenvolvedores a adoção de ferramentas gratuitas. Sendo que o *middleware* não fornece suporte para estas ferramentas, de forma a impossibilitar um aproveitamento total dos recursos fornecidos pelo mesmo.

2.4.3.2 BOINC

Berkeley Open Interface for Network Computing (BOINC) é um *middleware* de *grid* oportunista com seu código-fonte aberto, desenvolvido pela universidade de Berkeley na Califórnia [39]. Sua principal finalidade é fornecer um ambiente de *grid* para o meio científico através de computação voluntária. É possível baixar um cliente no próprio site e escolher a qual projeto deseja colaborar. No entanto o BOINC tem seu uso não somente voltado a este fim, mas também para criar ambientes de HPC em universidades e empresas.

No site ainda é possível encontrar uma lista com os cem maiores colaboradores na computação voluntária, com estatísticas e gráficos dos projetos, com o intuito de promover e estimular a colaboração entre os participantes. O site também conta com fórum, listas de discussões, lista de algumas publicações e uma boa documentação sobre os projetos e o próprio *middleware*.

1. Aplicabilidade:

   O BOINC possuí APIs para auxiliar no desenvolvimento de aplicações específicas para o *middleware*. Com apenas algumas modificações no código é possível tornar qualquer aplicação paralelizável no *middleware*. Essas APIs estão disponíveis unicamente para linguagem C/C++ e Fortran, contudo existem outras APIs que podem ser utilizadas para gerar aplicações nativas para o *middleware* em outras linguagens. É o caso da API DC-API, que fornece meios para trabalhar com aplicativos desenvolvidos em Java [40].

   Embora o uso das APIs para o desenvolvimento das aplicações seja encorajado, outra forma de utilizar a infraestrutura oferecida pelo *middleware* é através do Boinc *Wrapper*, que executa as tarefas como sub-processos, gerenciando a comunicação com o Boinc Client [41].

   O BOINC *server* foi desenvolvido para executar em plataformas Linux, o que pode gerar uma certa complicação em ambientes totalmente formados por máquinas com Microsoft Windows. Todavia, uma alternativa para esse cenário é o usar uma máquina virtual, com o *server*,

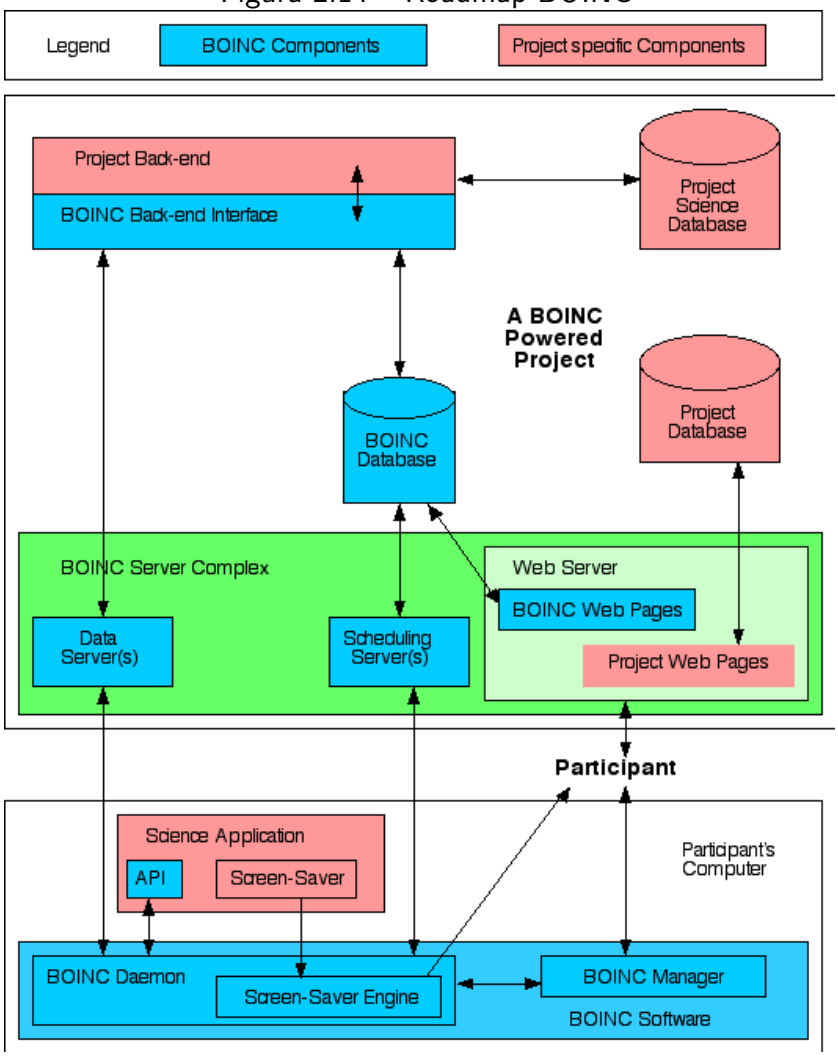
rodando sobre uma das máquinas com Microsoft Windows. Esta opção não acarreta em custos financeiros, pois existem alternativas de virtualização gratuitas, e até mesmo, de código-fonte aberto. Já os clientes possuem versões para Linux, Microsoft Windows e Mac OS.

2. Arquitetura:

   O BOINC possui o conceito de projeto, onde cada projeto possuí diversas aplicações, e cada aplicação diversas unidades de trabalho (*workunit*). Quando um novo recurso computacional é inserido na infraestrutura do *grid*, deve-se escolher o projeto com o qual pretende contribuir e conectar-se ao servidor deste projeto, que definirá os *workunits* as quais deve processar [41].

   Para compor um ambiente de HPC utilizando o BOINC é necessário somente uma máquina que receberá o BOINC *server* e no mínimo uma máquina com o BOINC *Client* para realizar as tarefas. Este BOINC *server* pode sustentar vários projetos, cada um com suas particularidades e um caminho único de acesso através do navegador [42]. Na Figura 2.14 [43], é visível uma separação entre componentes do servidor dos componentes particulares de cada projeto.

Figura 2.14 – Roadmap BOINC



Do lado dos clientes, que possuem o BOINC *Client* instalado, encontra-se o BOINC *Manager*, BOINC *daemon*, BOINC *Screen Saver*, BOINC *Client* API e *Science Applications*. As próximas informações descrevem cada um desses componentes e foram baseadas em [44] com algumas informações retiradas de [43].

BOINC Manager: é o componente responsável por fornecer uma *interface* gráfica de controle, de forma a permitir configurar o modo de trabalho e definir a quais projetos aquele cliente irá contribuir.

BOINC Daemon: responsável por controlar o *Science Applications* e gerenciar as tarefas recebidas, podendo armazena-las em *buffers*, para isso recebe comandos do BOINC Manager. Em sistema UNIX executa como um *Daemon*, enquanto que, em sistemas Microsoft Windows executa na forma de um serviço.

BOINC Screen Saver: componente que executa a proteção de tela no computador, onde demonstra algumas informações de caráter informativo, tais como: gráficos sobre a aplicação, informação da *Science Applications*, aviso de estado ocioso e aviso em caso do *Daemon* não estar executando.

BOINC *Client* API: permite a comunicação do BOINC *Client* com o *Science Applications* e facilitando o desenvolvimento, evitando a necessidade de recriar componentes que já foram desenvolvidos, testados e depurados.

***Science Applications*:** Corresponde às funções, rotinas ou programas que hospedados no BOINC *Client*, recebem os dados vindos do BOINC *server* e utilizando os recursos do cliente, geram as saídas correspondentes, remetendo-as novamente ao BOINC *server*. Um cliente poder ter quantos *Science Applications* ele precisar, que variam de acordo com a quantidade de projetos em que ele esta participando.

O BOINC *Server* é composto por um banco de dados, um servidor Web e por cinco *daemons* que encontram-se no BOINC *Core*. Esses componentes podem ser executados na mesa máquina ou em máquinas separadas, de acordo com a necessidade e infraestrutura desejada. Abaixo estão descritas as funcionalidades de cada componente.

***Validator*:** responsável por gerenciar os status de cada *workunit* e seus resultados. Ele acessa o BOINC *Database* frequentemente, verificando e atualizando os status dos *workunit*. Como a *workunit* pode ter vários status e cada status possui sub-status além dos resultados, este trabalho torna-se um dos *daemons* que mais utiliza a CPU da máquina.

Assimilator: verifica frequentemente por *workunits* finalizados e executa uma função pré-definida pelo admitsultados em um arquivo e a divisão e armazenamento dos resultados em outras mídias.

File Deleter: procura por *workunits* concluídas e que já passaram pelo *Assimilator* para excluir os arquivos utilizados no processamento. Um ponto importante a ressaltar é que ele somente excluir os arquivos e não os registros no banco de dados, assim é possível realizar consultas ao banco sobre *workunits* já finalizados.

***Scheduling Server*:** trata-se de um programa CGI que coordena os trabalhos enviados ao BOINC *Core*, de forma a organizar as tarefas para um melhor uso dos recursos disponíveis. Este *Server* comunica-se com os clientes de duas formas: manualmente, acionado através do BOINC Manager ou automaticamente, feita pelo BOINC *Daemon* para informar a existência de problemas na execução ou caso os *workunits* já tenham sido finalizados. Outro ponto a destacar é que ele não se conecta diretamente ao BOINC *Database* mas usa as informações salvas no *Feeder*.

***Feeder*:** trabalha como um *buffer* armazenando em uma espaço de memória compartilhado as consultas realizadas pelos clientes no *BOINC Database*. Dessa forma, o *feeder* permite uma forma rápida de acessar novamente essas informações, pois existe uma forte tendência de que elas sejam novamente necessárias em um curto espaço de tempo e quando acessadas no *feeder*, não geram consultas ao banco, agilizando o processo.

BOINC *Database*: trata-se de um banco de dados MySQL que tem como objetivo guardar as informações necessária para o funcionamento do *middleware*. Nele estão contidas as informações referentes aos usuário, descrições das aplicações de *workunits*, descrição dos clientes e também as tabelas usadas no site do projeto.

***Data Server*:** responsável por gerenciar e armazenar os dados envolvidos no processo de execução dos *applications*, prove um caminho para os clientes receberem os dados necessários para executar seus *workunits* e posteriormente retornar os resultados obtidos. O *Data Server* pode ser hospedado em uma máquina separada, contendo programas ou rotinas personalizadas para conversão e tratamento dos dados.

***Web Server*:** servidor web que contém as páginas do BOINC, por onde feito a interação com o usuário.

Ainda cada projeto pode possuir seu próprio banco de dados independente, utilizado para guardar dados específicos necessários para o projeto e sua página web no *Web Server*, com as particularidades do projeto, como é visível na Figura 2.14.

3. Instalação:

   Como pré-requisitos para instalação do *middleware* estão softwares como: Apache, juntamente com PHP e Python, para fornecer a interface de usuário e o MySQL, usado pelo *middleware* para gerenciar as tarefas [39]. Também é preciso criar um usuário chamado "boincadm" que deve ter acesso ao banco MySQL e permissões de leitura e escrita nas pasta do Apache.

Ao cumprimento de todos os pré-requisitos, a instalação se torna trivial, bastando entrar no diretório da aplicação e executar três comandos, dentre eles o conhecido $make$, para iniciar o processo de compilação da aplicação.

4. Considerações:

   O *middleware* possuí versões do cliente para os principais SOs da atualidade, colaborando assim para uma adoção em ambientes com heterogeneidade de SOs. Seu cliente é de fácil instalação e conta com duas formas de visualização: simples, que vem por padrão e trás apenas as informações necessárias para controlar o cliente e avançado, com informações e controle mais detalhado sobre o que esta sendo executado pelo programa.

   Um diferencial é a estrutura de gerenciamento dos projetos executados, onde um projeto pode ter várias aplicações vinculadas e cada aplicação é dividida e processada pelos clientes. É possível cadastrar vários projetos em cada cliente e gerenciar a quantidade de recursos disponíveis para cada projeto, além de, momentos em que um projeto esteja sem *workunits* para processar o próprio aplicativo do cliente divide os recursos inutilizados com os demais projetos cadastrados.

   Apesar do amplo suporte aos SOs oferecido pelo cliente, o *Manager* possuí versões apenas para Linux e Microsoft Windows, sendo o primeiro o mais utilizado. Os que desejam instalar o *manager* do *middleware* em ambientes Microsoft Windows poderão ter problemas com a falta de material de apoio ao assunto e o alto grau de complexidade para instalação. É preciso baixar o código-fonte do *midleware* e compilar na máquina desejada, para isso o *script* de compilação exige alguns requisitos, como por exemplo uma versão do Visual Studio instalado, entre outros detalhes que tornam este processo complexo.

   Por outro lado a instalação em ambientes Linux possuí boa documentação e uma maior facilidade para encontrar materiais de auxílio na internet, ou ainda, é possível utilizar a máquina virtual já configurada, disponibilizada no site, que agiliza sua implantação para teste e execução de projetos que não exigem grandes quantidades de processamento por parte da máquina em que se encontra o *manager*.

2.4.4 Comparação dos *Middlewares* Estudados

Ao longo deste capítulo foram abordados alguns dos principais *middlewares* da atualidade, com o intuito de levantar informações relevantes, juntamente com as principais características, dos mesmos. A partir disso foi possível observar as particularidades de cada um, onde tais particularidades são importantes no momento de escolha para adoção de um deles.

Características como linguagem nativa e plataforma são de caráter eliminatório, pois a incompatibilidade com uma delas torna o ambiente inutilizável, ou mesmo, impossível de ser implantado no cenário proposto. Por isso recomenda-se que estas sejam as primeiras características

a serem avaliadas, seguidas pela avaliação de características como segurança e suporte a classe de aplicações *Workflow*, que dependendo do cenário, podem ser de caráter mandatório. Um ambiente em que se faz necessário um alto grau de segurança deve-se avaliar as técnicas abordadas de cada *middleware* e as saídas encontradas para os pontos críticos de segurança em um sistema distribuído. Já para aplicações da classe *workflow*, torna-se imprescindível que o *middleware* forneça uma boa solução para o tratamento das mesmas.

No entanto, ainda existem outros fatores que podem influenciar na decisão e que também devem ser avaliados e comparados. Dentre esses fatores pode-se destacar a forma como são enviadas as tarefas, pois em muitos casos existe um grande número de tarefas a serem executadas e uma forma simples e eficiente de trabalhar com elas pode poupar tempo e complicações devido ao alto grau de complexidade para executar tarefas rotineiras. Nesta mesma linha temos as APIs, fornecidas por alguns *middlewares* que também tem forte peso no tempo de desenvolvimento e integração da aplicação com o ambiente.

Sendo assim, a seguir encontra-se uma relação de algumas das principais características de um *middleware*, seguidas pela Tabela 2.4 com as comparações entre os *middlewares* estudados.

1. **Linguagens nativas:** Linguagens suportadas pelo *middleware* para programação a nível de código em que o
2. **Envio de tarefas:** Formas de envios de tarefas oferececidas pelo *middleware*
3. **Plataforma:** Sistema Operacional em que o *middleware* pode executar
4. **Oportunista:** Se apresenta características oportunistas
5. **Segurança:** Forma pela qual o *middleware* realiza a autenticação dos usuários
6. **Troca de favores:** O *middleware* apresenta um sistema de benefícios conforme o nível de colaboração
7. **Workflow**: Suporte para aplicações da classe *workflow*
8. **APIs:** Fornece APIs para o desenvolvimento de aplicações

Apresentada a Tabela 2.4 com a comparação entre os *middlewares*, na Seção 4.3 será descrito o cenário do experimento e abordado os pontos destacados na tabela, sobrepondo-a sobre os pontos do cenário para auxiliar na definição de qual *middleware* será implantado.

Tabela 2.4 – Comparação entre os *Middlewares*

	Globus	**OurGrid**	**Unicore**	**Alchemi**	**Boinc**
Linguagens nativas	Java, C e Python	Nenhuma	Nenhuma	C#	C/C++ e Python
Envio de tarefas	Linha de comando	Linha de comando e Gráfica	Linha de comando e Gráfica	Linha de comando e Gráfica	Linha de comando
Plataforma	Linux	Linux, Mac OS e Windows	Linux, Mac OS e Windows	Windows	Linux, Mac OS e Windows
Oportunista	Sim	Sim, mas preferencialmente dedicado	Não	Sim	Sim
Segurança	MyProxy	OpenFire	Certificado x509 ou UVOS ou Certificados Proxy	Login/Senha	Login/Senha
Troca de favores	Não	Sim	Sim	Não	Não
Workflow	Não	Não	Sim	Não	Não
APIs	Sim	Não	Não	Sim	Sim

3. Ferramenta Desenvolvida

Para o presente trabalho foi desenvolvido uma ferramenta com o propósito de facilitar e agilizar o envio de tarefas para *grids*. A ferramenta foi projetada para fornecer um interface *web* que centraliza o gerenciamento das tarefas enviadas ao *middleware*.

Uma de suas principais características é a possibilidade de estender a ferramenta para comunicar-se com diferentes *middlewares*. Para isso bastando transcrever o comportamento particular do novo *middleware* em uma classe que estenda a interface *Middleware* e montar os arquivos de visualização necessários para possíveis interações com o usuário.

Entre as principais funções para que foi projetada, a ferramenta tem o intuito de:

- organizar os arquivos da tarefa;
- gerar os arquivos de configuração exigidos pelo *middleware*;
- submeter a tarefa;
- resgatar os resultados após o término da tarefa.

3.1 Motivação

O envio de tarefas a um *grid* não pode ser tratado como uma tarefa trivial, muitos *middlewares* impõem uma rígida estrutura de organização dos arquivos, além da geração de arquivos de configuração, que sofre um agravante, em casos que existam grandes quantidades de arquivos que precisam ser descritos, tornando o processo ainda mais oneroso e complexo. No entanto, a maioria dos *middlewares* não fornece uma ferramenta facilitadora que diminua o esforço necessário por parte do usuário, sendo que, o mais próximo encontrado, são ferramentas que auxiliam parte do processo, sem uma *interface* amigável e deixando grande parte do trabalho para utilizador.

Pensando nesse problema, veio a motivação para o desenvolvimento de uma ferramenta capaz de organizar e gerar os arquivos necessários através de informações recebidas do usuário por meio de uma *interface* amigável, diminuindo ao máximo a necessidade de configurações manuais. A ferramenta deve fornecer suporte ao diversos *middlewares*, para que não fique atrelada a nenhum *middleware*, forçando a utilização de um *middleware* específico aos que desejam fazer uso da ferramenta. Outro ponto importante é o fato de fornecer acesso remoto, permitindo que a submissão de tarefas possa ser feita a distância e ainda, fornecer uma forma fácil de identificação para acesso ao *grid*, sem necessidade da geração de credenciais por meio de certificados.

3.2 Tecnologias e Conceitos Abordados

Para o desenvolvimento da ferramenta foram abordados conceitos e tecnologias atuais, com o intuito de desenvolver uma ferramenta de fácil utilização e que possibilite sua extensão sem grandes modificações.

Dentre as abordagens adotadas, pode-se citar os conceitos envolvidos em um servidor *web*, conceitos de padrões de projetos, programação orientada a objeto e desenvolvimento de aplicações *web*. Já do ponto de vista das tecnologias pode-se citar a utilização de *Cascading Style Sheets*(CSS), *Javascript*, *JavaServer Pages*(JSP), Java e também conhecimentos do ambiente Microsoft Windows.

3.3 Desenvolvimento

A principal classe da aplicação chama-se App e é responsável por conter as principais variáveis e constantes do sistema, sendo chamada em diversos pontos ao longo do processo de execução da ferramenta. Para cada *middleware* suportado pelo sistema deve existir uma classe específica de mesmo nome, colocada dentro do pacote $middlewares$ e implementando a interface $middleware$. De forma que assim aplica-se o padrão *strategy*, para garantir uma maior modularidade para o suporte de novos *middlewares*.

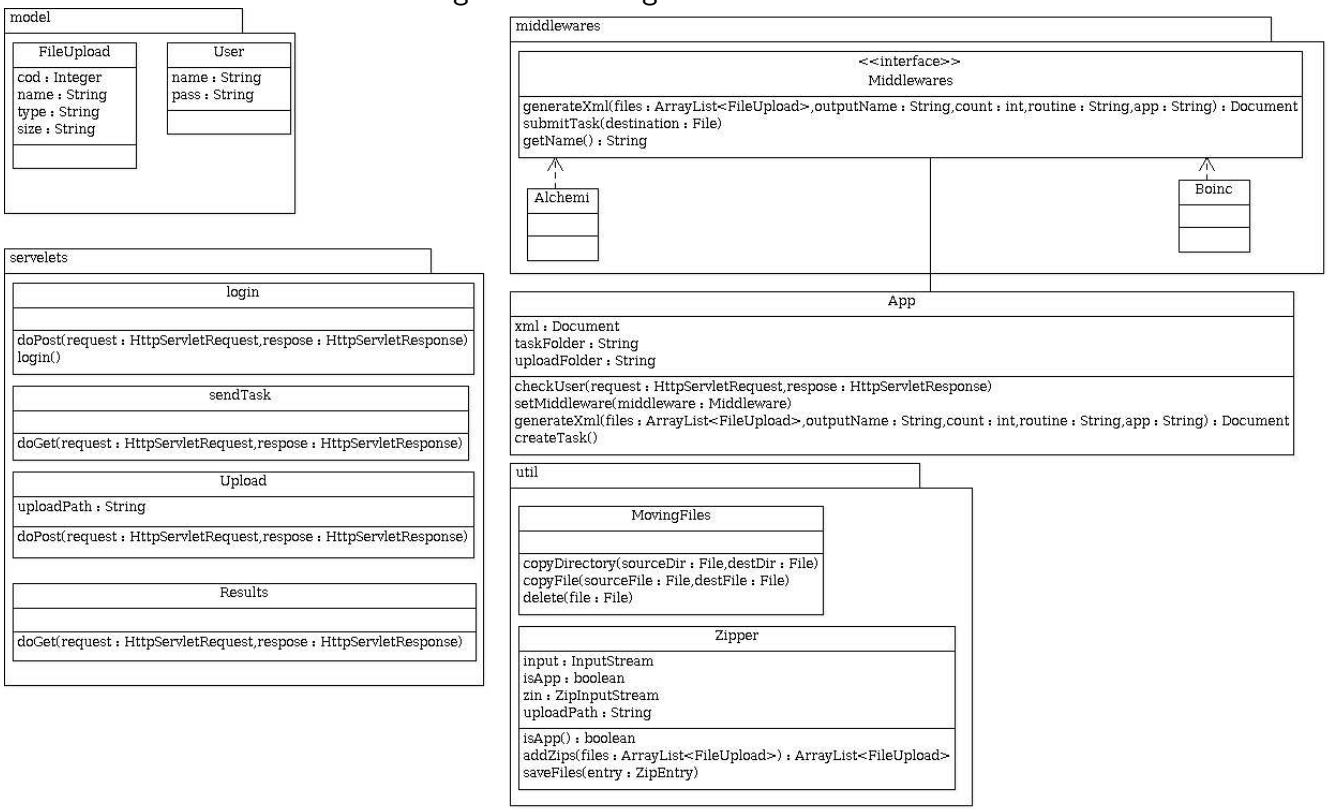
No pacote $servelets$ estão contidas as classes que estendem a classe $Servelet$ do java, sendo assim usadas com o propósito de processar as requisições enviadas ao servidor em cada etapa do processo. Elas fazem uso das classes do pacote $util$ e $model$ para trabalhar as informações recebidas, de modo a padronizá-las para que as classes que implementam a *interface* possam recebê-las e trata-las de modo singular, de acordo com a necessidade de cada *middleware*. Na Figura 3.1 é possível visualizar o diagrama de classes, sendo que nele encontram-se todas as classes implementadas até o momento, dando uma melhor visão do sistema.

Inicialmente a aplicação recebeu as implementações necessárias para se comunicar com o *middleware* Alchemi, sendo assim possível avaliar a estrutura de suprote proposta na aplicação. Algumas particularidade implementadas pela classe *middleware* diz respeito a responsabilidade de fornecer os arquivos necessários para a comunicação com o *middleware*, sendo esses, basicamente dois arquivos: $Alchemi.Core.dll$ e $alchemi_{j}sub.exe$, onde o $alchemi_{j}sub.exe$ é chamado via linha de comando, pois é através dele que a aplicação envia as tarefas ao *grid*.

É importante salientar que a aplicação, faz uso do executável "7zip", sob licença GNU LGPL. Sendo esse usado para compactar os arquivos resultantes da execução de cada *job* finalizado. Essa medida foi adotada para contornar o problema encontrado em casos que a tarefa a ser executada no *grid* não permite alterar o nome do arquivo de saída, consequentemente o *middleware* acaba sobrescrevendo os arquivos a cada resultado recuperado. O ponto positivo para a adoção desta solução foi a diminuição dos arquivos resultantes, que em alguns casos pode ultrapassar os 90%. Como os arquivos de saída são compactados, diminuem o tempo gasto no tráfego pela rede quando

comparados aos arquivos não compactados, agilizando o processo de recuperação dos dados.

Figura 3.1 – Diagrama de Classes



3.4 Execução

O aplicativo conta com um sistema de restrição básico, por isso a primeira tela visível, após acessar o endereço da aplicação, é uma tela de login padrão, sendo assim o aplicativo conta com um controle de permissões básico, dificultando o acesso a usuários não autorizados. Com o acesso ao sistema o usuário terá contato com a primeira etapa do processo, onde deve selecionar o *middleware* para qual deseja submeter a tarefa e enviar os arquivos do programa que será executado, juntamente com suas entradas. Para isso o usuário conta com duas possibilidades:

- Selecionar um-a-um os arquivos de entrada, recomendada somente em casos que não é possível realizar a compactação dos arquivos ou em casos que a compactação dos arquivos venha a ser demorada e não apresente grande diminuição no tamanho dos mesmos.

- Compactar todos os arquivos de entrada em um único arquivo ou, caso desejar, em vários arquivos de formato zip. Os benefícios por optar por essa forma de envio são: menor tempo de *upload* dos arquivos, pois a compactação diminuirá o tamanho dos mesmo e a simplificação

da seleção dos arquivos, pois é preciso selecionar apenas um arquivo, que conterá todos os arquivos de entrada.

Para o caso da aplicação submetida ser composta por mais de um arquivo, a mesma deve ser enviada pelo campo definido para tal, compactada no formato zip. Não se faz necessário qualquer identificação que indique o formato dos arquivos enviados, pois o próprio sistema reconhece quando se trata de um arquivo compactado e toma as medidas necessárias. O envio dos arquivos compactados, tanto para os de entrada quanto os pertencentes a aplicação, é encorajado, visto que os possíveis ganhos no tempo de envio, devido a diminuição de tráfego na rede, são significativos.

Com os arquivos carregados no servidor o próximo passo é definir como será organizado a execução. Nesse momento o usuário visualiza-rá uma tela próxima a mostrada na Figura 3.2, onde o campo $Comando$ representa o comando chamado para executar a aplicação, podendo esse trabalhar com coringas, que auxiliam na definição de parâmetros variáveis. Dessa forma é possível submeter aplicações que necessitam de entradas diferentes a cada execução, além de permitir o uso de contadores, que podem ser utilizados para diferenciar os arquivos de cada execução. Um possível exemplo de utilização é demonstrado na chamada do programa a baixo:

```
aplicativo.exe arquivo_variável.ext arquivo_variável2.ext contador saida.ext
```

Nessa situação tem-se o executável de nome $aplicativo.exe$ que recebe quatro parâmetros, sendo o primeiro e o segundo parâmetro correspondentes aos arquivos de entrada, que devem variar de *job* para *job*; o terceiro parâmetro corresponde a um valor numérico, que deve ser incrementado em cada *job* e o quarto corresponde ao nome do arquivo que deve receber a saída da execução.

Esse tipo de situação em especial pode gerar um problema sério ao usuário, pois ele terá que definir manualmente cada chamada do programa em cada um dos *jobs* sem mesmo poder fazer uso de replicação, pois terá que alterar cada parâmetro para receber arquivos de entrada diferente, além de incrementar o valor do terceiro parâmetro. Contudo, utilizando-se da aplicação o usuário poderá fazer uso dos coringas fornecidos pela aplicação, permitindo criar uma única linha de comando que irá se adequar para cada *job*. Sendo assim, abaixo está a solução para o exemplo utilizando os coringas da aplicação.

```
aplicativo.exe {param1} {param2} {ai} saida.ext
```

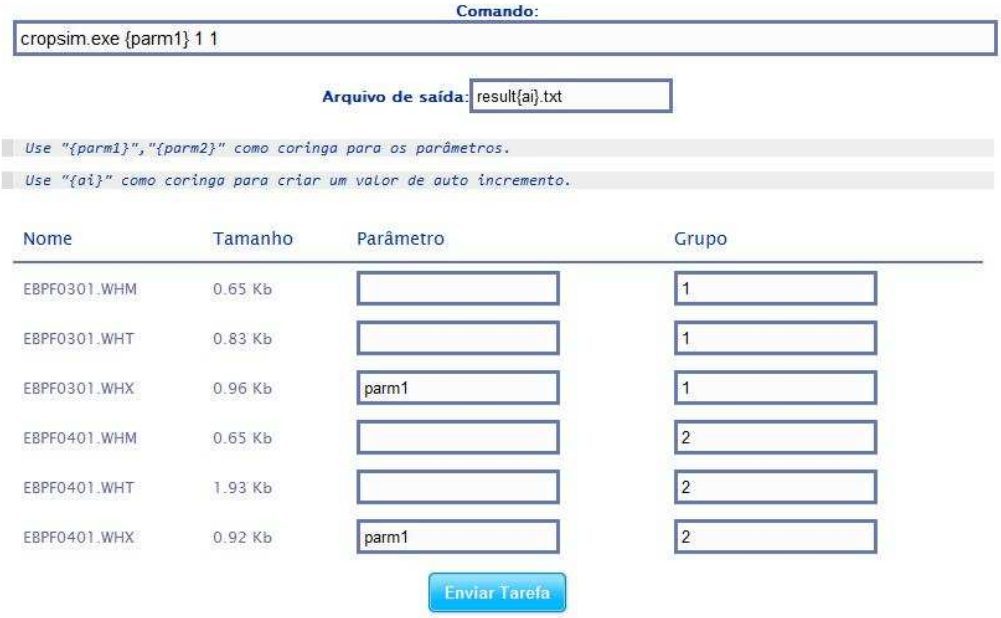
Com o comando a ser chamado pelo *middleware* para executar a aplicação já definida, é preciso identificar quais arquivos de entrada irão substituir os coringas. Para isso o aplicativo lista em uma tabela todos os arquivos de entrada enviados, onde cada arquivo de entrada recebe dois valores importantes, na coluna de parâmetro e na coluna de grupo. Na coluna de parâmetro deve-se definir quais os arquivos de entrada correspondem aos coringas definidos anteriormente, os arquivos que ocuparão a posição do coringa $param1$ devem receber como valor $param1$ na coluna dos parâmetros e os que ocuparão a posição do coringa $param2$ recebem o valor $param2$.

Há casos em que nem todos, ou mesmo nenhum, dos arquivos de entrada são passados por parâmetro, causando dificuldade para identificar a qual *job* pertence cada uma das entradas, especialmente para as aplicações que possuem mais de uma entrada para cada execução. Para contornar esse problema o usuário deve fazer uso da coluna "Grupos", pois adota-se a ideia que cada *job* possui um grupo de entrada de dados, sendo assim é preciso definir em cada arquivo a qual grupo ele pertence.

Na Figura 3.2 temos 2 grupos, sendo assim, serão dois *jobs*, os arquivos pertencentes ao primeiro *job* recebem o valor 1 na coluna $grupo$, enquanto os arquivos pertencentes ao segundo *job* recebem o valor 2. Ao final, terá um valor de grupos equivalente ao número de *jobs* q serão executados. Para os casos em que existam grandes quantidades de entradas, recomenda-se alterar o arquivo responsável por gerar a tabela, de modo a preencher automaticamente os campos de acordo com as necessidades da aplicação que será executada pelo *middleware*.

O campo $Arquivo\ de\ saída$ refere-se ao arquivo que deve ser resgatado ao final de cada execução, podendo ainda, serem vários os arquivos resgatados, nesse caso o usuário pode informar um-a-um separando os nomes por espaço. Outra possibilidade é fazer uso de coringas conhecidos da informática, é o caso do "?"e do "*". Por exemplo, se o usuário deseja resgatar todos os arquivos com a extensão "out", deve preencher o campo da seguinte forma $*.out$. Outro exemplo interessante seria a necessidade de resgatar os arquivos "saida_1.txt", saida_2.txt", saida_3.txt e todos os arquivos com extensão "rst". Para isso, bastaria fornecer o seguinte valor ao campo: $saida\_?.txt\ *.rst$.

Figura 3.2 – Tela de Configurações

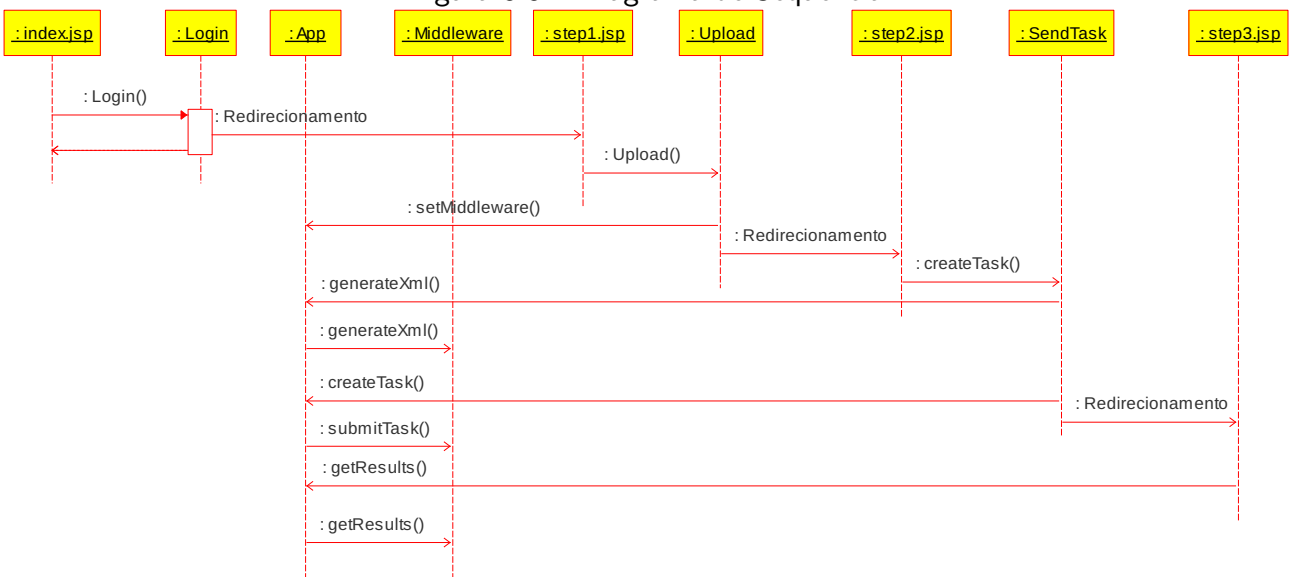


Ao enviar a tarefa o usuário será submetido a última etapa do processo, onde aguarda pelo término das execuções para solicitar o resgate dos arquivos resultantes. É importante salientar que

esse processo é referente a submissão de tarefas aos *grids* construídos com o *middleware* Alchemi e que pode sofrer pequenas modificações no processo quando utilizado com outros *middlewares*, visto que, cada *middleware* possuí suas particularidades, que podem necessitar de diferentes etapas para chegar até o momento de recuperar os resultados.

Por fim, a Figura 3.3 ilustra o diagrama de sequência, em que inicia com a página $index.jsp$, onde consta o formulário para efetuar *login* no sistema. Ao submeter o formulário o *servelet Login* é chamado fazendo as verificações necessárias e redirecionando para a página $step1.jsp$, caso esteja tudo certo. Essa página faz chamada ao *servelet* $Upload$, que recebe e organiza os arquivos enviados ao servidor, de modo a criar uma pasta especifica para aquela tarefa, além de ajustar o *middleware* selecionado e redirecionar para a página $step2.jsp$. Nesse momento é onde o usuário tem mais trabalho, pois precisa definir, entre os arquivos enviados, quais são e como serão organizadas as entradas para cada uma das execuções. Ao término desse processo, as informações são enviadas para o *servelt* $SendTask$, responsável por criar a tarefa no *middleware*, para isso, ele chama as funções da classe App, que faz uso das funções definidas na interface $middleware$ para delegar a tarefa para a classe do *middleware* selecionado. Durante esse processo o usuário é redirecionado para a página $step3.jsp$ onde aguarda para realizar o resgate dos resultados.

Figura 3.3 – Diagrama de Sequência



No próximo capítulo será comentado sobre o uso da ferramenta nos experimentos realizados para o presente trabalho, de modo a verificar os benefícios de sua utilização.

4. Experimentos

Neste capítulo serão abordados os experimentos realizados, bem como os procedimentos para que eles pudessem ser feitos e as conclusões obtidas.

4.1 *Middlewares* Avaliados

Com base nos estudos realizados e a partir das comparações feitas ao final do Capítulo 2.4 e com o auxilio da Tabela 2.4, chegou-se a definição sobre quais *middlewares* serão trabalhados na próxima etapa deste trabalho. Os *middlewares* escolhidos são: Alchemi e Boinc, pois ambos suprem as necessidades do cenário proposto na Seção 4.3 além do suporte a algumas das linguagens mais difundidas no âmbito acadêmico e empresarial.

Pode-se, ainda, citar a praticidade de implantação e utilização do Alchemi para ambientes em que se deseja um rápido e simples uso do ambiente paralelo e a estrutura organizacional dos projetos no Boinc, que possibilita um melhor gerenciamento dos projetos a serem trabalhado com ambiente, em locais que os recursos serão utilizados por vários setores com diversos projetos envolvidos.

4.2 Metodologia

Para o presente trabalho serão realizados oito baterias de testes utilizando o programa *case*, descrito na Sessão 4.5, em conjunto com a ferramenta desenvolvida durante esse trabalho, descrita no Capítulo 3. Como metodologia, foi definido a realização de 2430 execuções sequenciais do programa *case*, sobre o *middleware* Alchemi, no cenário proposto na sessão 4.3.2.1.

Com relação as baterias, foi definido que serão realizadas 5 execuções para cada bateria de testes. Sendo que, a bateria inicial será a utilização do *grid* com apenas um executor, servido como controle para os demais, seguido pelo acréscimo de um executor a cada bateria adjacente, onde, na última bateria, serão 8 executores. Ao final, serão levantados todos os resultados e eliminados o melhor e pior resultado de cada bateria, sendo feita a média dos 3 resultantes de cada bateria. Os números, tantos de baterias quanto de testes e execuções, foram definidos com base no tempo que se dispunha para realizar os testes,sendo assim, esses valores representando os maiores números possíveis de se trabalhar no tempo que se dispunha para realizar os testes.

Para avaliar os resultados foram trabalhados alguns conceitos comuns em computação paralela, sendo estes, descritos a baixo:

***Speedup*:** medida utilizada para avaliar a performance, sendo essa, utilizada para medir o ganho obtido com a execução paralela comparada com a execução sequencial. Sua fórmula é expressa a baixo:

$$Speedup = \frac{Desempenho\ sequencial}{Desempenho\ paralelo}$$

***Speedup* Linear:** definição para a situação em que dobrando o número dos processadores dobrar-se a velocidade, considerado o ideal quando se paraleliza uma aplicação, por isso, também pode ser chamado de *speedup* ideal.

Eficiência: está medida proporciona uma indicação da utilização efetiva dos processadores. Se o valor for abaixo de um indica que os processadores estão sendo sub-utilizados, caso fique acima de um significa que o problema pode ser melhor decomposto, sendo que um bom resultado se obtêm quando o valor fica a baixo e próximo de 1. Sua fórmula é expressa a baixo:

$$Eficiência = \frac{Speedup}{Número\ de\ Computadores}$$

4.3 Cenário de Aplicação dos Experimentos

Nesta Seção será apresentado o cenário proposto para implantação e teste dos *middlewares* apresentados no presente trabalho.

4.3.1 Descrição do Cenário

O cenário deve refletir ao máximo a atual situação das infraestruturas encontradas em empresas e instituição de ensino do país. Apesar das particularidades de cada situação, é possível levantar características semelhantes presentes em todos os cenários. Estas características estão descritas a baixo.

O grupo de máquinas utilizadas neste cenário deverá ser composto por máquinas atuais. Com um único processador composto por diversos núcleos, classificando as máquinas como pertencente à classe MIMD - Multiprocessadores, segundo a classificação de Flynn (veja Seção 2.1.3). E quando classificadas conforme o acesso a memória (veja Subseção 2.1.4) se enquadrem na classe UMA, pois possuem um acesso uniforme à memória.

Com a união dessas máquinas feita pelo *middleware*, formara-se-a uma nova máquina com arquitetura pertencente à classificação de Flynn correspondente a classe MIMD - Multicomputadores (veja Seção 2.1.3), pois teremos múltiplos processadores, cada um com sua própria memória e realizando a comunicação entre eles através de um barramento. Consequentemente sua classificação conforme a memória passara a ser NORMA (veja Subseção 2.1.4).

De uma forma mais comercial podemos definir este cenário com relação ao modelo físico (veja Subseção 2.1.5) como um NOW, visto que sua formação se dá através de nós com diferentes especificações de *hardware* e de uso não exclusivo para HPC.

É importante salientar que os softwares presentes nas máquinas também reflitam a realidade encontrada nos cenários empresariais e institucionais, por esse motivo as máquinas serão ser configuradas com sistema operacional *windows* [45] e com softwares instalados correspondentes aos usualmente utilizados pelas empresas e instituições.

4.3.2 Cenário da Realização dos Experimentos

O cenário escolhido será um dos laboratórios de informática do no Instituto Federal Sul-Rio-Grandese - campus Passo Fundo. A escolha deste cenário se dá devido a dois fatores predominantes: a existência prévia de uma infraestrutura de computadores em ambiente de rede e um grande período de ociosidade das máquinas.

Devido a esses fatores, o cenário escolhido se encaixa para as definições dos estudos realizados neste trabalho, descritos na Subseção 4.3.1.

4.3.2.1 Detalhamento do Cenário

O laboratório é constituído por nove máquinas de configuração semelhantes, no entanto 8 máquinas serão utilizadas para realizar o processamento e a nona máquina conterá a ferramenta para gerenciar o *middleware*. Sendo assim, foi realizado o levantamento do *hardware* das máquinas utilizadas no experimento e transcrito a baixo.

Processador : Intel Pentium 4 631, 3000 MHz

Placa Mãe: Phitronics P7V800-M

Memória do Sistema: 2 GB (PC3200 DDR SDRAM)

Adaptador Gráfico: VIA/S3G UniChrome Pro IGP (32 MB)

Disco Rígido: Maxtor 6 V080E0 SCSI Disk Device (80 GB, 7200 RPM, SATA-II)

Adaptador de Rede : Realtek RTL8169/8110 Family Gigabit Ethernet NIC

No que diz respeito ao *software*, as máquinas possuem instalado, como sistema operacional, o Microsoft Windows XP Professional com *Service Pack* 3 e uma gama se softwares aplicativos que incluem suíte *office*, servidor *web* Apache, JDK 6, Mozilla Firefox 4, PostgresSQL 9.0 entre outros aplicativos, comumente encontrados em computadores utilizados por instituições com cursos de informática.

4.4 Instalação

Nesta sessão serão descritos os processos de instalação de ambos os *middlewares*, com uma abordagem dividida em três etapas: requisitos, com o levantamento dos softwares necessários para que seja possível realizar a instalação; processo de instalação, descrevendo o processo de instalação do *middleware* e observações, com detalhes particulares encontrados durante o processo de instalação e que não se encontram em documentações.

4.4.1 Alchemi

A seguir estão detalhados os processos e particularidades encontrados para instalação do *middleware* no cenário proposto na Subseção 4.3.2. Tanto o Alchemi *Manager* quanto o Alchemi *Executor* foram instalados nas máquinas do laboratório, sendo que a máquina com o Alchemi *Manager*, apesar de possível, não recebeu um Alchemi *Executor*.

4.4.1.1 Requisitos

Para a instalação da máquina que recebeu o Alchemi *Manager* foi necessária a instalação do *framework* .NET 2.0 e do SGBD MSDE 2000, já para os computadores que receberam o Alchemi *Executor* foi necessário somente o *framework*.

4.4.1.2 Processo de Instalação

Para instalar o *framework* basta realizar o seu download, executá-lo e seguir as orientações. Já para o MSDE 2000, foi necessário executar a instalação no $prompt$ de comando, passando a seguinte linha de comando $setup.exe\ SAPWD = ' SA'\ DISABLENETWORKPROTOCOLS = 0\ SECURITYMODE = SQL.$ O procedimento de instalação por linha de comando é obrigatório, já que o instalador não possuí interface gráfica. Os parâmetros da instalação são obrigatórios, sendo que, o primeiro é referente à definição da senha do usuário administrador do banco de dados, o segundo habilita a conexão ao banco por todos os protocolos suportados e o terceiro permite a autenticação de usuários cadastrados na tabela de usuários do SGBD. Ao término da instalação faz-se necessário reiniciar a máquina, para que o SGBD seja iniciado com o sistema.

Após o cumprimento dos requisitos, realiza-se a instalação do Alchemi *Manager*, executando seu instalador. Ao término da instalação é solicitado a configuração do banco de dados a ser utilizado, onde deve ser informado os dados para conexão.

Com o final da instalação, encontra-se no menu de programas o grupo denominado $Alchemi$, onde consta o executável para configuração do banco, o executável do *Manager* e seus respectivos manuais de utilização. Ao executar o *Manager*, um ícone aparece no *system tray* e a janela do *Manager* aparece. É possível configurar a porta pela qual o *Manager* irá trabalhar e, caso necessário, indicar um outro *Manager* a quem se conectar, para formar um *multi-cluster*.

Para instalar os *Executors*, basta executar o instalador e o grupo $Alchemi$ também será criado no menu de programas, com o executável do *Executor* e seu manual. Após iniciar o programa é preciso informar o endereço e a porta onde encontram-se o *Manager*, também é possível definir se o *Executor* trabalhará de modo dedicado ou não.

4.4.1.3 Observações

Para este trabalho foi utilizada a versão 1.0.5, pois a versão 1.0.6 apresentava problemas no momento de recuperar os resultados das tarefas. Outro ponto descoberto, foi a não necessidade de instalar o banco SQL, pois o *middleware* tem a opção de gerenciar seu banco diretamente na memoria *ram* do computador, o que torna o processo de instalação ainda mais prático. Por fim, é importante salientar que os *Executor* só funcionam corretamente no modo dedicado após desativar o *firewall*, pois mesmo habilitando a aplicação no *firewall* continuou apresentando problemas.

4.4.2 Boinc

A seguir estão detalhados os processos e particularidades encontrados para instalação do *middleware* no cenário proposto na Sessão 4.3.2.

4.4.2.1 Requisitos

Para esse trabalho foi utilizado um servidor já configurado em uma máquina virtual, disponível no site do Boinc. No entanto, no site também é possível encontrar a lista de dependências necessárias para realizar a instalação do *middleware*.

4.4.2.2 Processo de Instalação

Apesar de ser possível baixar uma máquina virtual com o servidor instalado, é necessário registrar um projeto que deve conter a aplicação que irá ser executada. Para criar o projeto no Boinc é necessário executar uma série de passos, que estão descrito a baixo em ordem de execução.

- Pelo terminal deve-se entrar na pasta do Boinc, que neste caso encontra-se dentro da pasta *home* do usuário $boincadm$, para executar o comando $/tools/make\_project --url\_base\ http://ip\_do\_servidor/nome\_do\_projeto$ aceitando todos os avisos que aparecem. Ao final do processo será criado uma pasta chamada *projects* na pasta *home* do usuário ativo, que conterá a pasta do projeto.

- Como usuário *root*, é necessário alterar o arquivo $/etc/apache2/httpd.conf$, adicionando ao final as linhas contidas no arquivo $nome\_do\_projeto.httpd.conf$, que encontra-se na raiz do projeto criado. Após, deve-se reiniciar o servidor Apache.

- O próximo passo é adicionar a seguinte linha ao *crontab* do usuário atual: $00-595***/home/nome\_do\_usuario/projects/nome\_do\_projeto/bin/start --cron$. Essa linha garante que em um intervalo de 5 minutos será executado as rotinas responsável por manterem os dados do projeto atualizado.

- Como último passo, deve-se alterar a senha da área restrita do site do projeto. Isso é feito através da alteração do arquivo $htpasswd$, na pasta $/html/ops$, que encontra-se na raiz do projeto. Para esse procedimento pode-se executar o comando $htpasswd\ -c\ .htpasswd\ nome\_do\_usuario$ dentro da pasta ops, seguido pela senha desejada.

Com o projeto criado, o próximo passo é adicionar uma aplicação. Para isso deve-se acessar a pasta app, dentro da raiz do projeto, criar uma pasta com o nome da aplicação e colocar a aplicação, respeitando a nomenclatura: $nomeAplicativo\_versãoDoCliente\_Plataforma.extensão$, um exemplo seria: $nomeAplicativo\_5.0\_windows\_intelx86.exe$. Também deve-se adicionar o aplicativo no arquivo $project.xml$ que encontra-se na raiz. Isso é feito adicionando as seguintes *tags* ao fim do arquivo:

```
<app>
<name>nome_do_aplicativo</name>
<user_friendly_name>Nome do Aplicativo</user_friendly_name>
</app>
```

Para registrar o aplicativo no projeto executa-se os seguintes comando, de dentro da pasta do projeto e na ordem: $./bin/xadd$ e $./bin/update\_versions$. Depois do término da execução desses comandos o aplicativos estará devidamente registrado ao projeto e apto a receber *workunits*.

O próximo passo é criar os *templates* que definem como será trabalhado os arquivos de entrada e os de saída em cada *workunits*(veja Item 2.4.3.2). Para isso são criados 2 arquivos que devem ser dispostos na pasta $templates$ na raiz do projeto e que devem seguir a nomenclatura: $re\_nome\_aplicativo$, para descrever como será tratado os resultados e $wu\_nome\_aplicativo$, para descrever os arquivos de entrada.

É preciso adicionar os *daemons* que irão gerenciar a aplicação, para isso devem ser adicionadas novas entradas no arquivo $config.xml$, essas entradas estão descritas a baixo:

***feeder*:** responsável por garantir que os clientes receberão *workunits* para processar;

***transitioner*:** manipula a transição de estados de envio de *workunits* e recebimento dos resultados;

***file_deleter*:** remove os arquivos de entrada e saída após o término do *workunit*;

***sample_dummy_validator*:** verifica se os resultados recebidos são resultados válidos;

***sample_dummy_assimilator*:** verifica se existem resultados válidos ou resultados com erros.

Por último, é preciso registrar o *workunit* no *grid*, para isso executa-se o comando $create\_work$ passando por parâmetro as localizações dos arquivos criados. Um exemplo de criação de *workunit* poderia ser: $./bin/create\_work\ -appname\ nome\_aplicativo\ -wu\_namewu\_nome\_Aplicativo\_01\ -wu\_template\ templates/wu\_nome\_aplicativo.xml\ -result\_template\ templates/re\_nome\_aplicativo.xml\ -min\_quorum\ 1\ target\_nresults\ 1\ arquivo\_saida.txt$. A partir desse momento o *grid* detecta o *workunit* e delega trabalho aos seus clientes.

4.4.2.3 Observações

Antes de iniciar a criação do projeto foi executado os comando $\_autosetup$ e $./configure --disable-client\ --enable-server$, seguidos do comando $make$. Esse processo foi realizado para garantir que estava tudo corretamente instalado e funcionando.

No momento da criação do projeto foi utilizado o parâmetro url_base passando o ip do servidor, está medida foi tomada para evitar problemas com a url do projeto, pois o *middleware* utiliza o nome da máquina como caminho e os clientes do *middleware* não estavam conseguindo acessar o servidor pela url gerada. Existe uma série de outros parâmetros que podem ser passados pela linha de comando, no ato da criação do projeto, que podem ser encontrados na documentação do site do Boinc.

4.5 Estudo de Caso

Para realizar os testes do HPC construído, para verificação dos benefícios da paralelização e para comprovar as facilitações fornecidas pela ferramenta descrita no Capítulo 3, será utilizado um aplicativo fornecido pelos professores e pesquisadores do grupo Mosaico da Universidade de Passo Fundo, no qual fazem uso da aplicação para realizar pesquisas na área de plantio de cultivares.

Portanto, a aplicação a ser usada como estudo de caso será o CROPSIM [46], desenvolvido por L. A. Hunt and S. Pararaiasingham, que tem como propósito gerar um modelo que simula o crescimento e desenvolvimento da cultura do trigo.

Essa aplicação é particularmente importante para aqueles que desejam prever os resultados da colheita do trigo, visto que, o modelo pode gerar possíveis cenários futuros para a cultura do trigo baseado nos dados de entrada que alimentam a aplicação. Com os resultados das simulações é possível então, realizar uma previsão de alta precisão do desemprenho de safras futuras do trigo. Sendo assim, o aplicativo pode ser utilizado por cooperativas, instituições de pesquisas e demais empresas com interesse na cultura do trigo e que desejam uma previsão de colheitas futuras.

4.5.1 Funcionamento do Estudo de Caso

A aplicação recebe como entrada um único arquivo texto, que dita como deve ser executada a simulação, passando parâmetros e fazendo referência a outros arquivos que contém os dados à serem avaliados. Esses parâmetros contém diversas informações e fatores que influenciam diretamente no crescimento e desenvolvimento do trigo, como, por exemplo, temperaturas, épocas de semeadura, cultivares, irrigação, umidade, entre outros. O conjunto desses fatores é denominados de tratamento, sendo que, um arquivo de entrada pode ter vários tratamentos.

Basicamente a aplicação recebe o arquivo de entrada com as informações para a execução, juntamente com os arquivos contendo as entradas de dados e simula, através de modelos matemáticos, a produção da cultura do trigo. Essa simulação é feita com dados de cenários passados ou

futuros, sendo utilizado para extração de conhecimento através de informações geradas das saídas geradas pelo modelo, como por exemplo, verificação de mudanças climáticas globais, melhor época de plantio, melhor cultivar, melhor período de irrigação, entre outros.

4.5.1.1 Motivação

Para que os resultados produzidos pelo programa possam ser utilizados é necessário que uma grande quantidade de dados seja utilizada em sua entrada, dados esses, recorrentes de bases de dados com mais de 30 anos, que recebem novos leituras a cada dia. Dessa forma, o programa precisa ser executado milhares de vezes com entradas diferentes, gerando saídas diferentes a cada execução. Esse comportamento identifica uma aplicação da classe do tipo *parameter sweep*, explicada na Subseção 2.3.3, por isto passível de ser paralelizada. Dessa forma será possível agilizar o processo de cálculos do modelo matemático, colaborando para que o processo de simulação seja executado em um menor espaço de tempo, otimizando o processo.

4.6 Execução dos Testes

Para realizar as execuções foi adotado uma abordagem de decomposição de domínio, ou seja, em que existe uma divisão prévia dos dados em partições que posteriormente receberão a computação, sendo está a mesma para todos. Faz-se uso também do paradigma SIMD (veja Seção 2.1.3), em que todas as tarefas executam a mesma aplicação sobre diferentes entradas de dados, sendo assim também chamada de *Data Parallelism*.

4.6.1 Processo de Execução

A sala utilizada para os experimentos foi a de número 504, do prédio 5 do IFSUL, onde as máquinas encontravam-se ociosas, recebendo a instalação do *middleware* momentos antes da execução. Isso foi necessário pois as máquinas contam com um programa que restaura a configuração das mesmas para uma imagem padrão, a qual não consta a instalação do *middleware*.

Os testes foram realizados na sequência planejada, iniciando com uma máquina para realizar o processamento, acrescendo uma nova máquina a cada 5 execuções, até o total de 8 máquinas. Ao final de cada execução foi efetuado a manutenção do Alchemi *Manger*, eliminando completamente a aplicação dos registros do *middleware*. Para isso foi utilizado os recursos oferecidos pelo próprio Alchemi *Manager*.

Apesar do sucesso dos resultados, as execuções passaram por alguns contratempos, que atrasaram, mas não impediram a conclusão do cronograma planejado. Entre os principais problemas enfrentados, pode-se destacar as vezes em que o Alchemi *Manager* sofria um erro inesperado ao término da execução, forçando o seu fechamento e impedindo a obtenção dos resultados.

4.7 Avaliação dos Resultados

Ao término das execuções e após finalizar o levantamento dos dados obtidos, foi possível reunir e analisar os dados resultantes, mensurando suas variações de tempo e realizando o cálculo de *SeepdUp* e eficiência. Com isso foi possível montar a Tabela 4.1 com a exposição dos resultados.

Tabela 4.1 – Relação dos Testes

N Máquinas	Tempo(segundos)	SpeedUp	Eficiência
1 Máquina	694		
2 Máquinas	353	1,96415094339623	0,9820754716981
3 Máquinas	251	2,76494023904382	0,9216467463479
4 Máquinas	175	3,95066413662239	0,9876660341556
5 Máquinas	144	4,80831408775982	0,9616628175519
6 Máquinas	114	6,06997084548105	1,0116618075801
7 Máquinas	99	6,96321070234114	0,9947443860487
8 Máquinas	88	7,82706766917293	0,9783834586466

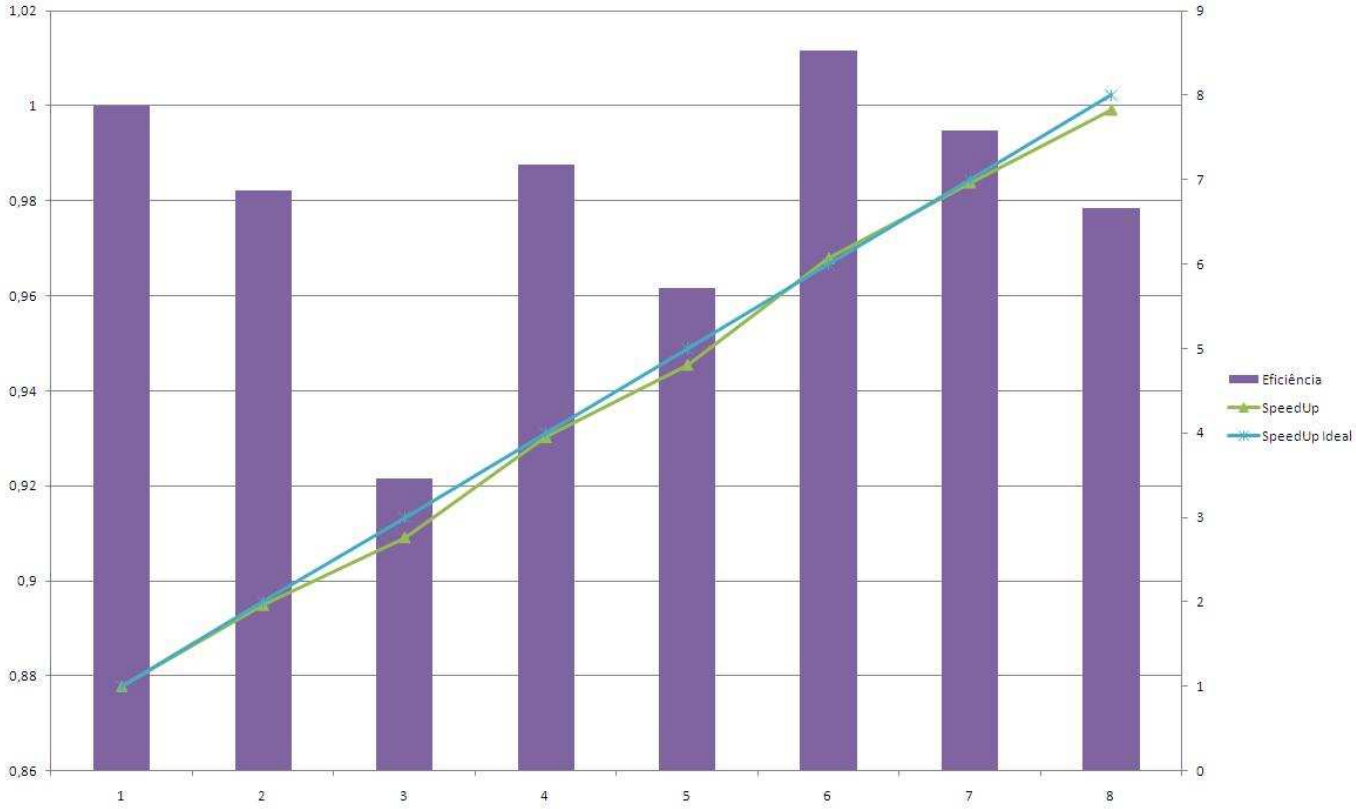
A partir da da leitura da tabela o próximo passo foi construir o gráfico de *speedup* e eficiência, demonstrados na Figura 4.1. Nele é possível notar que o tempo de execução é linearmente diminuído sempre que temos a duplicação do numero de máquinas executando as tarefas, caracterizando assim um *speedup* linear.

Com a demonstração do gráfico na Figura 4.1, termina a análise dos resultados, demonstrando o potencial da paralelização perante o cenário de testes trabalhado. A partir desses resultados é possível afirmar que os ganhos significativos do experimento encorajam a adoção de paralelização para aplicações que possuam grandes quantidades de dados, até mesmo para as de curto tempo de processamento.

Para exemplificar, pode-se sugerir um cenário hipotético em que o total de simulações necessárias seria de 48.000, com os dados do cenário proposto e usando a Tabela 4.1 como referência, o tempo total para realizar todas as simulações seria de, aproximadamente, 228 minutos utilizando apenas uma máquina. No entanto, quando trabalha-se com as 8 máquinas simultaneamente, o tempo, aproximado, seria de 28 minutos, consolidando uma diminuição de 200 minutos no tempo de execução.

No próximo capítulo serão apresentados os resultados obtidos em todos os pontos do trabalho, trazendo o fechamento do mesmo, juntamente com os possíveis caminhos para dar continuidade ao mesmo.

Figura 4.1 – Gráfico de *SpeedUp* e Eficiência



5. Conclusão:

Ao término do trabalho chega-se a três pontos principais: levantamento da documentação e avaliação dos *middlewares*, o desenvolvimento de uma ferramenta facilitadora com a possibilidade de integração com vários *middlewares* e os resultados da paralelização de aplicações, com a utilização de um *case* real em um ambiente real.

5.1 Avaliação Dos Middlewares

A partir do momento em que uma empresa ou instituição identifica a necessidade da implantação de um sistema distribuído para sanar a demanda por processamento, se faz necessário uma completa pesquisa e análise das necessidades que o ambiente deve cumprir, seguido pelo levantamento das soluções passíveis de serem adotadas. No entanto, somente isso não é suficiente para tomar uma decisão definitiva, é recomendável que mais de uma solução seja selecionada, já que, o processo de implementação pode gerar contratempos e, até mesmo, barreiras que tornem inviável a solução anteriormente selecionada.

Contudo a adoção de um *middleware* oportunista em ambientes que existam máquinas com períodos ociosos é encorajada, visto que, as vantagens da utilização de infraestrutura já existente são consideráveis, especialmente do ponto de vista financeiro. Outra possível situação seria de casos em que existe o interesse na implantação de HPC, onde o *grid* oportunista seria utilizado em um primeiro momento para avaliar os reais ganhos com a utilização do HPC, para posteriormente realizar os investimentos necessário para HPC dedicado.

5.2 Ferramenta Desenvolvida

A ferramenta demonstrou-se de grande utilidade, pois auxilia na geração dos arquivos necessários para submeter uma aplicação ao *grid*, arquivos estes, que podem ser de alta complexidade e extensos, podendo desencorajar o uso do *grid*, devido ao seu grau de complexidade e trabalho exigido por parte usuário. Ainda, destaca-se por fornecer uma interface amigável permitindo que uma pessoa, mesmo que sem domínio sobre *grids*, possa submeter sua aplicação de forma simples e prática e, ainda, automatizando grande parte do trabalho de geração das regras e configurações de execução.

5.3 Estudo de Caso

Como base nos resultados experimentais transcritos na Seção 4.7, pode-se concluir que a proposta de paralelização da aplicação CROPSIM obteve um bom resultado. A provável justificativa para esses resultado está no fato da serialização das execuções, visto que, apesar de rápidas,

o acúmulo de grandes quantidades executadas sequencialmente acarreta em uma longa fila de espera, gerando congestionamento e acúmulo de execuções, mesmo com grande parte dos recursos disponíveis.

Com a divisão das execuções entre os computadores, que realizam as tarefas em simultâneo, o resultado é uma vazão muito maior, trazendo ganhos significativo a cada nova máquina conectada ao *grid*. Isso acontece nesse cenário em especial pelo fato da aplicação *case* não necessitar de grande poder computacional, mas sim, estar limitada pela quantidade de tarefas simultâneas que um computador pode executar.

5.4 Trabalhos Futuros

Com os resultados positivos na paralelização da aplicação *case*, o próximo passo é trabalhar com outras aplicações de simulação, ou mesmo com ferramentas que trabalhem em conjunto com o *case*, possibilitando o aproveitamento do HPC produzido ao longo deste trabalho, juntamente com a ferramenta de submissão, descrita no Capítulo 3, construída para simplificar e automatizar a submissão de novas tarefas.

No que diz respeito à ferramenta produzida para esse trabalho, apesar de funcional, a ferramenta necessita de algumas modificações que permitam um melhor controle ao longo do processo de submissão e recuperação dos resultados, também necessita de melhorias de interface para facilitar ainda mais o processo de submissão, além de implementações de módulos que permitam a comunicação com um maior número de *middlewares*.

Por fim, o material produzido durante o trabalho permite ao leitor ter uma visão mais detalhada sobre os *middlewares* apresentado, fornecendo-lhe uma fonte de pesquisa para alguns dos principais *middlewares* da atualidade, além de trazer um experimento positivo para ser avaliado por aqueles que têm interesse em adotar um HPC. No entanto existe a possibilidade de dar continuidade ao trabalho acrescentando novos *middlewares* para serem estudados e implantados, para possíveis comparações de desempenho.

5.5 Conhecimentos Adquiridos

Ao longo desse trabalho foram trabalhadas, principalmente, áreas de computação paralela, no entanto, muitos outros conhecimentos foram adqueridos, conhecimentos esses que dificilmente poderiam ser escritos em uma monografia. Por esse motivo essa sessão tem como objetivo citar e representar os conhecimentos e experiências vivenciados ao longo do trabalho.

Com certeza, um dos principais ganhos ao longo do processo dizem respeito a experiência de vida, através do enfrentamento de dificuldades, relações intra-pessoais e cumprimento de prazos e metas, exigidos durante o todo o processo, aliados as amizades feitas e a inserção no ambiente acadêmico, tornam-se os principais ganhos com a realização desse trabalho.

Também é possível citar os conhecimentos e habilidades desenvolvidas que dizem respeito ao tema do trabalho, além de conhecimento nas áreas de projeto e desenvolvimento de aplicações e programação com linguagem Java. Outros conhecimentos que não estão diretamente ligados a área do trabalho, mas que também foram importantes: aprimoramento da produção textual, interpretação de textos, abrangência dos conhecimentos de língua inglesa e também, a introdução ao ambiente latex.

Para finalizar, gostaria de dizer que, com a conclusão do presente trabalho, obteve-se a presente monografia, a ferramenta para auxiliar a submissão de tarefas aos *middlewares* e a contribuição feita com a paralelização do aplicativo CROPSIM, que resultou em um artigo submetido à ERAD.

Referências Bibliográficas

[1] PITANGA, M. *Construindo Supercomputadores com Linux*. [S.l.]: Brasport Livros e MultimÃdia Ltda, 2008.

[2] ROSE, C. A. F. D.; NAVAUX, P. O. A. *Arquiteturas Paralelas*. [S.l.]: Editora Sagra Luzzatto, 2003.

[3] DANTAS, M. *Computação Distribuídas de Alto Desempenho: Redes, Clusters e Grids Computacionais*. [S.l.]: Axcel Books do Brasil Editora, 2005.

[4] G., J. E. L.; GOMES, C. de A. A tecnologia e a realização do trabalho. Revista de Administração de Empresas, 1993. Disponível em: <http://rae.fgv.br/>.

[5] ROSE, C. A. F. D.; NAVAUX, P. O. A. Fundamentos de Processamento de Alto Desempenho. In: *Anais: 2^a Escola Regional de Alto Desempenho*. [S.l.: s.n.], 2002. p. 3 29.

[6] FLYNN, M. J. *Some computer Organizations and their Effectiveness*. [S.l.]: IEEE Transaction on Computers, 1972. 948–960 p.

[7] FAVERO, A. L. *Suporte a Multiprocessadores Simétricos (SMP) em kernel Linux*. [S.l.]: Universidade Federal do Rio Grande do Sul, 2004.

[8] HWANG, K.; XU, Z. *Scalable Parallel Computing: Technology,Architecture,Programming*. New York, NY, USA: McGraw-Hill, Inc., 1998. ISBN 0070317984.

[9] KRAUTER, K.; BUYYA, R.; MAHESWARAN, M. A taxonomy and survey of grid resource management systems for distributed computing. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 32, p. 135–164, Fevereiro 2002. ISSN 0038-0644.

[10] BAKER, M.; BUYYA, R.; LAFORENZA, D. Grids and grid technologies for wide-area distributed computing. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 32, p. 1437–1466, Dezembro 2002. ISSN 0038-0644.

[11] LIMA, A. A. M. de et al. Utilização e configuração do cluster colorado da upf. 2007.

[12] MAZIERO, C. A. Sistemas operacionais ii - gerência de tarefas. 2008.

[13] SANTANA, M. N. P. Alocação de tarefas paralelas comunicantes em ambientes distribuídos heterogêneos. Universidade de Brasília, 2006.

[14] BERNARDI Élder F. F. Simple grid: Um framework para distribuição automática de aplicações comerciais em grades computacionais. Pontifícia Universidade Católica do Rio Grande do Sul, 2007.

[15] MPI, W. *Message Passing Interface Forum*. 2011. Disponível em: <http://www.mpi-forum.org/>.

[16] THREADS, W. P. *POSIX Threads Programming*. 2011. Disponível em: <https://computing.llnl.gov/tutorials/pthreads/>.

[17] WIN32, W. P. *POSIX Threads (pthreads) for Win32*. 2011. Disponível em: <http://sourceware.org/pthreads-win32/>.

[18] TBB, W. *TBB Home*. 2011. Disponível em: <http://threadingbuildingblocks.org/>.

[19] OPENMP, W. *OpenMP.org*. 2011. Disponível em: <http://openmp.org/wp/>.

[20] CUDA, W. *What is CUDA | NVIDIA Developer Zone*. 2011. Disponível em: <http://developer.nvidia.com/what-cuda>.

[21] CONTI, F. de. Grades computacionais para processamento de alto desempenho. Universidade Federal de Santa Maria, 2009.

[22] BEOWULF, W. *Beowulf.org: The Beowulf Cluster Site*. 2011. Disponível em: <http://www.beowulf.org/>.

[23] SALES, P. da S. B. Avaliação de desempenho de ferramentas de renderização de imagens em clusters openmosix e arquiteturas multicore. In: . [S.l.]: Escola Politécnica de Pernambuco - Universidade de Pernambuco, 2008.

[24] OPENMOSIX, W. *openMosix, an Open Source Linux Cluster Project*. 2011. Disponível em: <http://openmosix.sourceforge.net/>.

[25] ALLIANCE, W. G. *The Globus Alliance*. 2011. Disponível em: <http://www.globus.org/>.

[26] FOSTER, I. T. Globus Toolkit Version 4: Software for Service-Oriented Systems. In: JIN, H.; REED, D. A.; JIANG, W. (Ed.). *NPC*. [S.l.]: Springer, 2005. (Lecture Notes in Computer Science, v. 3779), p. 2–13.

[27] SOUZA, G. Guia para o globus toolkit 4. 2011. Disponível em: <http://saloon.inf.ufrgs.br/twiki-data/Projetos/Grade/GradeUFRGS/WebHome/Guia_para_o_Globus_Toolkit_4.0.pdf>.

[28] GOLDCHLEGER, A. Integrade: Um sistema de middleware para computação em grade oportunista. Universidade de São paulo, 2004.

[29] MOWBRAY, M. Ourgrid: A web-base community grid. HP Laboratories, Bristol, 2006.

[30] CIRNE, W. et al. Labs of the World, Unite!!! *Journal of Grid Computing*, v. 4, n. 3, p. 225–246, 2006.

[31] MANTOVANI, G.; JUNIOR, O. L. P.; RODRIGUES, R. M. da C. Tutorial de instalação da grade computacional. Universidade Estadual do Oeste do Paraná - Centro de Ciências Exatas e Tecnolóligas, 2009. Disponível em: <http://200.201.81.50/ guilherme/cursos/projens/grades.pdf>.

[32] OURGRID, W. *Writing Jobs*. 2011. Disponível em: <http://www.ourgrid.org/index.php?option=com_content&view=article&id=67&Itemid=54&lang=en>.

[33] UNICORE, W. *UNICORE - Distributed computing and data resources*. 2011. Disponível em: <http://www.unicore.eu/>.

[34] ECLIPSE, W. *Eclipse - The Eclipse Foundation open source community website.* 2011. Disponível em: <http://www.eclipse.org/>.

[35] ANJOMSHOAA, A. et al. *Job Submission Description Language (JSDL) Specification, Version 1.0*. [S.l.], jun. 2005.

[36] MONO, W. *Main Page - Mono*. 2011. Disponível em: <http://www.mono-project.com>.

[37] LUTHER, A. et al. *Alchemi: A .NET-based Grid Computing Framework and its Integration into Global Grids*. Australia, dez. 2003.

[38] HEAD, M. R. *Alchemi: A .NET Grid Application Framework*. [S.l.]: Binghamton University, 2007.

[39] ANDERSON, D. P. BOINC: A System for Public-Resource Computing and Storage. In: *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. [S.l.]: IEEE Computer Society, 2004. p. 4–10. ISBN 0-7695-2256-4.

[40] MAROSI, A. C. et al. Enabling java applications for boinc with dc-api. In: KACSUK, P.; LOVAS, R.; NÃ©METH, Z. (Ed.). *Distributed and Parallel Systems*. [S.l.]: Springer US, 2008. p. 3–12. ISBN 978-0-387-79448-8.

[41] PIROTTI, R. P. Comparação entre ferramentas de desktop grid - boinc, condor e xtremweb. In: . [S.l.]: Universidade Federal do Rio Grande do Sul, 2011.

[42] BOINC, W. *BOINC*. 2011. Disponível em: <http://boinc.berkeley.edu>.

[43] BOINC, W. W. *Unofficial BOINC Wiki*. 2011. Disponível em: <http://www.boinc-wiki.info/>.

[44] ULRIK, C.; JAKOB, S.; PEDERSEN, G. *Developing Distributed Computing Solutions Combining Grid Computing and Public Computing M.Sc. Thesis*. 2005.

[45] FOLEY, M.-J. *Forrester: Windows 7 powering 21 percent of corporate desktops; XP still at 60 percent*. 2011. Disponível em: <http://www.zdnet.com/blog/microsoft/forrester-windows-7-powering-21-percent-of-corporate-desktops-xp-still-at-60-percent/9723>.

[46] HUNT, L. A.; PARARAIASINGHAM, S. Cropsim - wheat:a model describing the growth and development of wheat. Department of Crop Science, University of Guelph, Ontario, Canada, 1995.