Flávio Viotti

Escalonamento Hierárquico de Tarefas do tipo Bag-Of-Tasks com Compartilhamento de Arquivos

Santos

Setembro 2008

Flávio Viotti

Escalonamento Hierárquico de Tarefas do tipo Bag-Of-Tasks com Compartilhamento de Arquivos

Dissertação apresentada como requisito para obtenção do grau de Mestre em Informática junto à Universidade Universidade Católica de Santos.

Orientador: Hermes Senger

Santos

Setembro 2008

V799e VIOTTI, Flávio

Escalonamento Hierárquico de Tarefas do tipo Bag-Of-Tasks com Compartilhamento de Arquivos / Flávio Viotti - Santos:

[s.n.] 2008.

117f.; 30 cm (Dissertação de Mestrado - Universidade Católica de Santos, Programa em Informática).

I. VIOTTI, Flávio. II. Título

CDU 681.3:004 (043.3)

Dedicatória

Dedico à minha esposa, meu pai e a minha mãe *(in memorian)*,

Por muito que fizeram, e tiveram paciência comigo, e em especial à minha mãe que sempre sonhou em ver a conclusão desse trabalho.

Agradecimentos

Ao meu orientador professor doutor Hermes Senger sou grato pela orientação e paciência, ao Colégio Singular em especial ao meu coordenador prof. Celso Denis Gallão e o Diretor prof. José Carlos Fazzole Ferreira sou grato pela disponibilização do laboratório de informática, à Adriane Fontanna diretora da FATEC - São Caetano do Sul sou grato pela disponibilização do laboratório de informática.

Sou grato aos meus amigos do trabalho que me ajudaram nas correções de escrita e formatação do texto.

Aos amigos do peito que pacientemente aguardaram o término desse trabalho e que compreenderam que uma amizade vale muito.

Em especial agradeço a Deus por ter me dado força e inteligência, e colocado em meu caminho uma pessoa muito especial que é minha esposa Fernanda, por ter entendido a importancia desse trabalho em nossas vidas.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

BoT	Bag-of-Tasks
LHC	Large Hadron Collider
CERN	Centro Europeu de Pesquisas Nucleares
EGEE	Enabling Grids for E-Science in Europe
OV	Organização Virtual
RMI	Remote Method Invocation
WSDL	Web Services Description Language
XML	eXtensible Markup Language
SOAP	Simple Object Access Protocol
HTTP	Hyper Text Transfer Protocol
OGF	Open Grid Forum
OASIS	Organization for the Advancement of Structured Information Standards
GGF	Global Grid Forum
WSDL	Web Service Description Language
GT4	Globus Toolkit 4
GSI	Grid Security Infrastructure
SSL	Secure Sockets Layer
CAS	Community Authorization Service
RSL	Resource Specification Language
PSA	Parameters Sweep Applications
TCA	Tarefas que Compartilham Arquivos
MCT	Minimum Completion Times
MET	Minimum Execution Time
SA	Switching Algorithm

KPB	K-Percent Best
OLB	Opportunistic Load Balancing
IFA	Input File Affinity
GET	Serviço de Gerenciamento do Estado das Tarefas
WEKA	Waikato Environment for Knowledge Analysis

Resumo

Aplicações Bag-of-Tasks compostas por tarefas independentes que compartilham arquivos são freqüentes em diversas áreas da ciência. Exemplos de tais aplicações incluem buscas massivas (como por exemplo, quebramento de chaves), mineração de dados, simulação pelo método de Monte Carlo, manipulação de imagens, entre outros. Grades computacionais são bastante favoráveis para a execução desse tipo de aplicação, provendo capacidade computacional através da agregação de recursos distribuídos. Entretanto, a baixa escalabilidade freqüentemente limita o desempenho na execução de tais aplicações. Esta dissertação apresenta uma proposta de implementação de um escalonador de tarefas para grades computacionais, que tem como principal objetivo melhorar a escalabilidade de aplicações Bag-of-Tasks que compartilham arquivos, coordenando de forma mais eficiente a transferência de arquivos pela rede e a distribuição das tarefas entre as máquinas que compõem a grade, com o objetivo de melhorar a escalabilidade das aplicações. As técnicas de escalonamento hierárquico e de agrupamento de tarefas são implementadas pelo escalonador, e resultados experimentais mostram que a implementação proposta permite ganhos de desempenho.

Palavras-chave: Grade Computacional; Agrupamento de Tarefas; Escalonador; Modelo Hierárquico; Globus.

Abstract

Applications composed by independent tasks that share files are frequentley used in several areas of science. Some examples of such applications include the massive search (e.g. cryptography keys breaking), data mining, simulation of Monte Carlo Method, image manipulation among others. The use of Grid Computing is very favorable in the execution of thise kinds of applications, providing computing capacity through the distributed resources aggregation. However, the low scalability frequently limits the execution development of such applications. This dissertation presents an implementation proposal of Broker for Grid Computing, which has as its main objective to improve the scalability application of Bag-of-Tasks that share files that will coordinate in very efficienty way file transferences via network and tasks distribution among the machines that compose the grid, aiming to improve the scalability applications. The techniques of scalable hierarchy and task grouping are implemented by the broker and the experimental results show that this implementation proposal a significant development.

Key-words: Grid Computing; Tasks Grouping; Broker; Hierarchical Model; Globus.

1 Introdução

O avanço das tecnologias de informação e comunicação (TIC) e o desenvolvimento de novas aplicações científicas que produzem grande quantidade de informações e impõem a necessidade de oferecer infra-estruturas computacionais capazes de efetuar o processamento e armazenamento dessas informações. Novas aplicações requerem cada vez mais o uso de computadores interligados e a agregação de recursos computacionais. Um ambiente capaz de suprir essas necessidades é a Grade Computacional, que possibilita a agregação e coordenação de recursos computacionais e dispositivos especializados geograficamente espalhados e pertencentes a diversas instituições. Com Grades Computacionais é possível resolver problemas de grande porte em diversas áreas da ciência e engenharia.

A vantagem em executar aplicações em uma grade computacional é a capacidade de processamento obtida através da agregação de grandes quantidades de recursos computacionais. Um exemplo de aplicação típica que se beneficia do uso de grades computacionais são as aplicações *Bag-of-Tasks* (BoT). Tais aplicações podem ser decompostas em conjuntos de tarefas independentes, que compartilham arquivos de dados e não se inter comunicam, podendo ser executadas em qualquer ordem.

Apesar de sua aparente simplicidade, exemplos de aplicações BoT podem ser encontrados em várias áreas, incluindo mineração de dados, buscas massivas, simulações pelo método de Monte Carlo, fractais, biologia computacional e computação gráfica (CIRNE et al., 2003). Um modelo tipicamente utilizado para a implementação de tais aplicações em aglomerados e grades computacionais é o modelo mestre-escravo (master-slave), que consiste em um nó da grade ou aglomerado que é normalmente utilizado para dar início e co-

ordenar a execução das tarefas nos nós escravos. Geralmente, existe um único repositório que armazena os arquivos que serão enviados aos processadores que irão executar as tarefas. Uma limitação freqüente nesse tipo de arquitetura é a ocorrência de gargalos no processador responsável pelo controle da aplicação, bem como no computador que implementa o repositório de arquivos.

Um aspecto importante a ser considerado no escalonamento das tarefas é o fato de que os arquivos poderão ser compartilhados entre diversas tarefas. Tal compartilhamento deve ser observado no sentido de evitar a retransmissão desnecessária de arquivos para os nós da grade. Esse problema é conhecido na literatura especializada como escalonamento de tarefas independentes que compartilham arquivos, e será referenciado neste trabalho por *TCA* (GIERSCH; ROBERT; VIVIEN, 2006). O gerenciamento desse compartilhamento deve ser de responsabilidade do escalonador de tarefas. Além disso, o escalonador deve também considerar a arquitetura que será utilizada na execução das tarefas.

1.1 Objetivo

O objetivo do presente trabalho consiste em propor e implementar uma ferramenta capaz de escalonar as tarefas de uma aplicação baseada em tarefas independentes que compartilham arquivos. O escalonador deverá levar em conta o compartilhamento dos arquivos, de modo que uma vez transmitido um determinado arquivo para um processador, esse arquivo permaneça disponível de modo que outras tarefas que dele dependam, possam ser escalonadas no mesmo processador, e tal arquivo não seja retransmitido desnecessariamente.

Em ambientes de grades computacionais ou aglomerados, tais retransmissões podem comprometer o desempenho das aplicações. Com a finalidade de melhorar a eficiência do escalonamento, o mecanismo proposto deverá utilizar as seguintes técnicas:

- Agrupar tarefas que possuam dependências de arquivos em comum, conforme proposto por Silva (SILVA; SENGER, 2008). Esse agrupamento pode proporcionar uma redução no total de transmissões de arquivos.
- Organizar a arquitetura em uma forma hierárquica, conforme proposto por Senger et al. (SENGER; SILVA; NASCIMENTO, 2006). Essa hierarquia pode aliviar o gargalo que se forma sobre o nó mestre.

A abordagem proposta tem como objetivo minimizar o problema das retransmissões de arquivos e do gargalo sobre o nó mestre.

1.2 Metodologia

Inicialmente, será realizado um estudo das heurísticas de escalonamento e agrupamento de tarefas já existentes, que serão discutidas no capítulo 2. A partir desse estudo será proposto uma arquitetura que utilizará a plataforma de grade computacional Globus Toolkit versão 4. Essa arquitetura será desenvolvida através da implementação de um protótipo utilizando a linguagem JAVA, com o qual serão gerados resultados para validação e prova do conceito.

1.3 Organização do Trabalho

O capítulo 2 apresenta o ambiente de grades computacionais, com especial foco nas ferramentas para suporte à execução de aplicações. No capítulo 3 será discutida a questão do escalonamento de tarefas do tipo BoT em grades computacionais, bem como os principais problemas que limitam a escalabilidade no caso de compartilhamento de arquivos. O capítulo 4 propõe uma arquitetura de um escalonador voltada para a execução de aplicações BoT que compartilham arquivos, com base na abordagem proposta por Silva et al. (SILVA; SENGER, 2008) e Senger et al. (SENGER; SILVA; NASCIMENTO, 2006). O capítulo 5 apresenta uma análise dos resultados obtidos com a execução do escalonador. O capítulo 6 apresenta uma conclusão para o trabalho.

2 Grades Computacionais

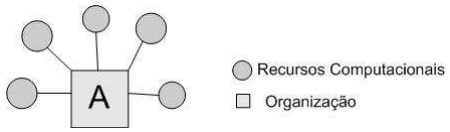
Diversas aplicações necessitam de grande poder computacional. É possível citar como exemplo o Large Hadron Collider (LHC), construído no Centro Europeu de Pesquisas Nucleares (CERN). O LHC é um acelerador de partículas que irá produzir uma enorme quantidade de informações, aproximadamente dez petabytes (10^7 gigabytes) por ano. Caso essas informações fossem gravadas em CD-ROM, eles poderiam formar uma pilha de aproximadamente 20 km de altura.

Essas informações necessitam ser processadas e armazenadas em algum local. Com a tecnologia existente, processar e armazenar todas essas informações em um único local seria impossível. Há estimativas de que 100.000 processadores de última geração seriam necessários para o poder computacional do LHC. Infelizmente, o CERN não dispõe de tantos recursos de processamento e armazenamento. Uma solução para esse problema seria a utilização de grades computacionais (SOTOMAYOR; CHILDERS, 2006).

A agregação de recursos computacionais para processamento e armazenamento disponibilizados em diversos locais, pode fornecer poder computacional suficiente. O EGEE(Enabling Grids for E-Science in Europe), faz exatamente isso: provê infra-estrutura computacional e armazenamento por toda a Europa para atender às necessidades do LHC.

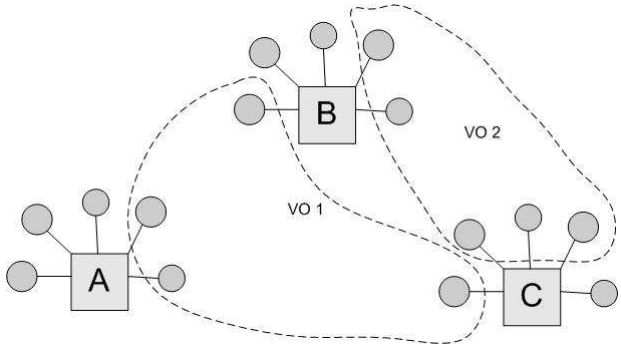
Com a utilização de grades computacionais é possível agregar diversos recursos de diversas organizações, e disponibilizá-los em uma organização virtual (OV) para resolver problemas específicos. Uma organização virtual pode ser formada conforme mostra a figura 1 e 2.

Figura 1: Uma Organização e seus recursos



Fonte: (SOTOMAYOR; CHILDERS, 2006)

Figura 2: Organização Virtual

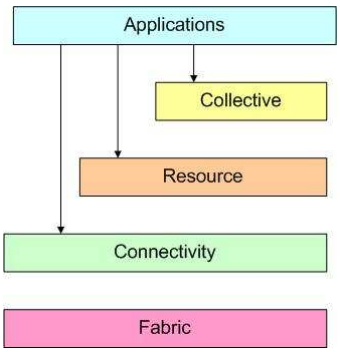


Fonte: (SOTOMAYOR; CHILDERS, 2006)

2.1 Arquitetura em Camadas

O estabelecimento, gerenciamento e exploração de organizações virtuais dinâmicas e multi-institucionais requer a criação de novas tecnologias. Essas novas tecnologias necessitam de uma arquitetura capaz de especificar componentes fundamentais, seus propósitos e funções, bem como o modo como esses componentes interagem (FOSTER; KESSELMAN; TUECKE, 2001).

Figura 3: Arquitetura de Grade Computacional



Fonte: (SOTOMAYOR; CHILDERS, 2006)

A arquitetura em camadas pode ser dividida da seguinte maneira conforme ilustrado na figura 3 e descrito a seguir:

- *Fabric*: É a camada que refere-se aos recursos que serão compartilhados na grade computacional. Esses recursos podem ser computadores individuais, aglomerados, supercomputadores, armazenamento em rede, banco de dados e etc.

- *Connectivity*: Esta camada define todos os protocolos para que os recursos possam se comunicar. Esses protocolos podem ser TCP/IP, HTTP, DNS e etc. Contudo, sistemas de grades computacionais também requerem segurança na comunicação. Portanto esta camada possui uma série de protocolos com o intuito de suprir tais necessidades. É possível citar alguns conceitos sobre segurança tais como: autenticação, autorização, criptografia de chaves públicas e autoridade certificadora.

- *Resource*: Esta camada refere-se a todos os serviços e protocolos que possibilitam o gerenciamento de recursos individualmente. Esse gerenciamento inclui tarefas como inicialização, monitoramento e contabilização desses recursos. Essa camada utiliza os serviços prestados pelas camadas inferiores para implementar:

  - Protocolos de Informações, que são utilizados para obter informações sobre o estado de um recurso, configuração, carga corrente e política de uso; e
  - Protocolos de Gerenciamento, que são utilizados para negociar acesso a um recurso compartilhado, reserva antecipada, qualidade de serviço, regular aspectos como contabilidade, pagamento entre outros.

- *Collective*: Esta camada faz a coordenação da iteração de múltiplos-recursos e constrói uma camada para prover serviços. Com isso é possível agregar e coordenar um grupo de recursos para executar uma tarefa. É possível exemplificar alguns tipos de serviços encontrados nessa camada:

  - Registro de Recursos: possibilita descobrir novos recursos na organização virtual e examinar suas propriedades.

  - Alocação e Escalonamento de Serviços: a execução de aplicações em sistemas de grade computacional implicam no descobrimento de recursos adequados. O serviço de alocação descobre esses recursos e os aloca. O serviço de escalonamento decide quais aplicações devem ser executadas, bem como seu local.
  - Serviço de Monitoramento: possibilita o monitoramento dos recursos disponíveis para a verificação de seu funcionamento.
  - Serviço de Gerenciamento de Dados: quando uma aplicação é executada pode existir a necessidade de informações que encontram-se armazenadas em bancos de dados. O serviço de gerenciamento de dados permite o acesso e transferência de informações localizadas nesses bancos de dados.

- *Applications*: Essa camada refere-se as aplicações que são executadas na grade. Essas aplicações não necessitam interagir diretamente com a camada Collective e possuem acesso livre e direto às camadas Resource e Conectivity quando necessário.

Atualmente, existem diversas infra-estruturas que possibilitam a utilização de sistemas de grade computacional, segue abaixo algumas dessas infra-estruturas.

- EGEE: Enabling Grids for E-Science in Europe (http://public.eu-egee.org/): Um ambicioso projeto em grade computacional que proporcionará aos cientistas acesso a diversos recursos computacionais espalhados entre 27 países.
- NEESit: (http://it.ness.org/): Provê uma extensa infra-estrutura para o NEES (Network for Earthquake Engineering Simulation).
- TeraGrid: (http://www.teragrid.org/): Um sistema em grade que provê uma poderosa infra-estrutura para pesquisas científicas abertas. Em 2004 o Teragrid possuía 20 *teraflops* de poder computacional e um pentabyte em armazenamento distribuído.
- Access Grid: (http://www.accessgrid.org/): Um sistema em grade utilizado para reuniões distribuídas, trabalhos colaborativos, seminários, conferências, tutoriais e treinamento.

- eDiaMoND: (http://www.ediamond.ox.ac.uk): O projeto eDiaMoND é um exemplo de como grades computacionais podem ser utilizadas para e-Health.

2.2 Histórico

A evolução histórica das tecnologias de Grade Computacional pode ser dividida em três gerações.

2.2.1 Primeira Geração

A primeira geração ocorreu entre os anos oitenta e meados dos anos noventa. Essa geração teve como objetivo atender às necessidades de grupos de aplicações específicas que demandavam muito poder computacional. Os principais projetos dessa fase foram o FAFNER e I-WAY.

O projeto FAFNER era um consórcio liderado pela Bellcore, Boston University, Syracuse University, Co-Operating Systems e Oxford University. Seu finalidade era coordenar esforços distribuídos dentro de uma rede hierárquica de servidores web, para fatorar números grandes. Seu principal objetivo era testar os limites de segurança do algoritmo de criptografia RSA.

O projeto I-WAY iniciado em 1995, tinha como objetivo implementar um *Backbone* capaz de conectar 17 centros de supercomputação dos EUA. Houveram tentativas no sentido de criação de soluções padronizadas para aspectos de segurança, escalabilidade e heterogeneidade no suporte à execução de aplicações paralelas complexas e de grande porte.

2.2.2 Segunda Geração

Diversos projetos tentaram resolver problemas tais como heterogeneidade, escalabilidade, múltiplos domínios administrativos, gerenciamento de recursos e sobrevivência a falhas. Foi utilizado o conceito de *middleware* para solucionar esses problemas.

O *middleware* implementa diversos modelos de programação e provê transparência sobre detalhes da arquitetura, rede, sistema operacional, hardware, linguagem de programação e localização física. Altas latências de comunicação geradas pela distância entre os diversos recursos levaram a adaptações nas aplicações. Essas adaptações tornaram as aplicações mais escaláveis. Os principais projetos dessa geração foram Globus, Legion e Condor.

2.2.3 Terceira Geração

Essa terceira geração foi marcada por uma nova visão sobre Grade Computacional, que vai além dos desafios ou aspectos puramente tecnológicos. Novos tipos de aplicações podem ser executadas em uma grade computacional. Para que isso aconteça novos requisitos são impostos a essa tecnologia, tais como reutilizar componentes de informação ou código, utilização cada vez maior de metadados, descoberta automatizada sobre funcionalidades e disponibilidade de recursos heterogêneos geograficamente distribuídos. Existe também a capacidade de manipulação de recursos heterogêneos tais como encontrar, agregar e combinar.

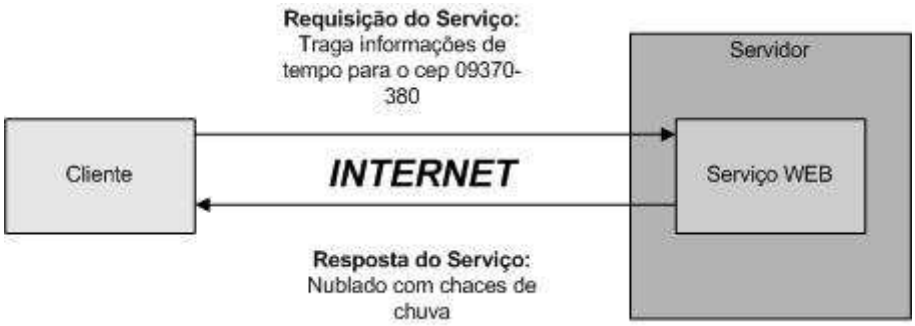
2.3 Serviços WEB

Desenvolver sistemas capazes de "conversar" entre si e com outros sistemas, legados ou não, sempre foi um desafio para a computação. Diante desse desafio algumas tecnologias tais como CORBA, DCOM, *Remote Method Invocation* (RMI), entre outras, foram propostas. Um grande problema relacionado a essas tecnologias é a interoperabilidade entre elas, que de uma forma geral conseguem se comunicar somente entre si. Uma tecnologia que auxilia a troca de informações, independente de plataforma é o XML. Diversas empresas tais como a Microsoft, HP, IBM, Sun Microsystem, etc, iniciaram pesquisas nessa área. O resultado dessas pesquisas no desenvolvimento de protocolos baseados em XML foi a especificação do XML-RPC e o SOAP. Essas especificações serviram como ponto de partida para a especificação de serviços web.

Um serviço web é uma abstração que pode ser implementada sob a forma de um agente de software, que envia e recebe mensagens. A troca de mensagens é especificada através de um documento que segue a especificação WSDL (CHINNICI et al., 2007), e define aspectos do serviço relacionados ao formato das mensagens, mecanismo de transporte da mensagem, serialização dos dados que compõe a mensagem e qual a localização na rede o agente pode ser invocado. Essas mensagens podem ser transportadas por diversos protocolos, sendo o mais utilizado o HTTP.

Conforme ilustrado na figura 4, um serviço web é um componente localizado em um servidor que recebe uma mensagem contendo uma requisição. Essa requisição solicita a execução de um determinado serviço. O serviço solicitado é então executado. O resultado é enviado para o cliente utilizando uma mensagem. O cliente recebe a mensagem e utiliza seu conteúdo.

Figura 4: Serviço WEB



Fonte: Próprio autor.

Ferramentas que se utilizam de serviços web podem desfrutar de algumas vantagens com relação a outras tecnologias, essas vantagens são:

- Serviços web são independentes de plataforma e linguagens, uma vez que usam o padrão XML(FALLSIDE; WALMSLEY, 2004). Isso permite que um cliente possa desenvolver módulos de aplicações em diferentes linguagens e executá-los em diferentes plataformas.

- Grande parte dos serviços web existentes utiliza o protocolo *HTTP* para transporte de suas mensagens. Essas mensagens na maioria das vezes não são bloqueadas pelos *Proxies* e *Firewalls* por serem mensagens *HTTP*.

- Aplicações clientes podem "aprender dinamicamente" quais operações são suportadas por um determinado serviço, bastando para isso ter acesso à sua interface em WSDL.

Por outro lado, a utilização de serviços web pode trazer algumas desvantagens, uma delas seria a sobrecarga, pois transmitir todos o dados em XML pode ser menos eficiente se comparado a um código binário. O que se ganha em portabilidade perder-se em eficiência.

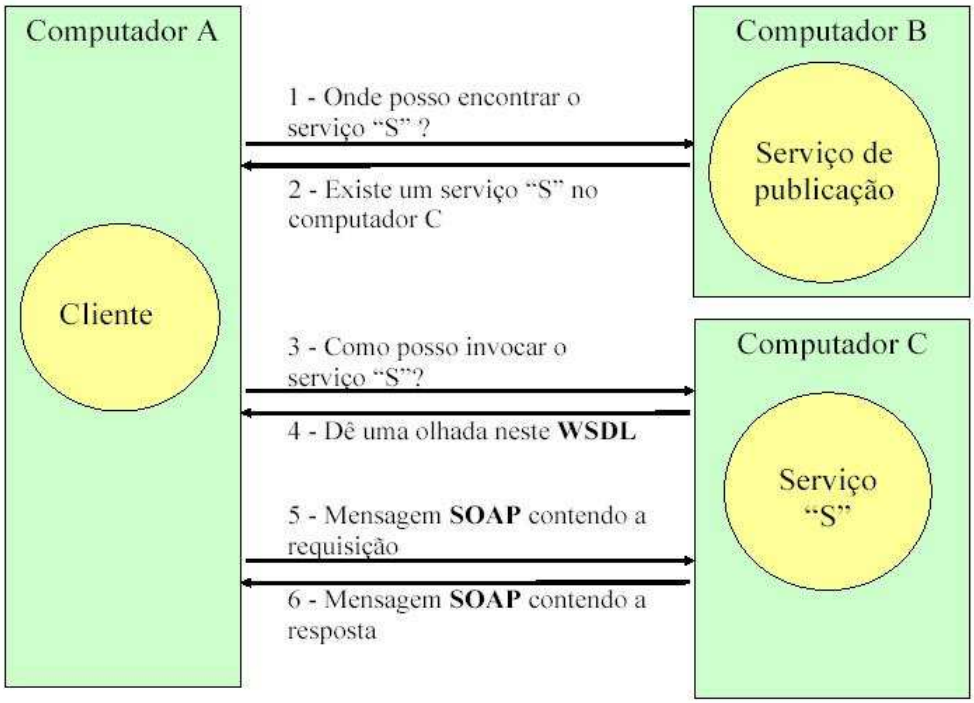
Para a utilização de um serviço web é preciso a execução de três etapas, sendo elas: publicação do serviço, descoberta do serviço e utilização do serviço. A seguir são descritos cada uma dessas etapas.

A *Publicação do Serviço* ocorre fazendo-se o registro do serviço em um determinado provedor. Esse provedor de serviços fica responsável por disponibilizar informações padronizadas sobre os serviços nele registrado, como por exemplo onde esse serviço está disponível. Todas essas informações são disponibilizadas através de uma linguagem XML.

A *Descoberta do Serviço* é feita através de uma solicitação do cliente para o provedor de serviços. Essa solicitação também é feita através de um serviço web. De posse do local onde o serviço encontra-se implementado, é possível utilizar o serviço web.

A utilização do serviço deve ser feita primeiramente obtendo informações sobre como deve ser feito sua invocação. A resposta para essa pergunta é feita através da especificação WSDL, e em seguida o serviço pode ser utilizado. O funcionamento de um serviço web mais detalhado pode ser observado conforme figura 5.

Figura 5: Invocação do Serviço WEB



Fonte: (SOTOMAYOR; CHILDERS, 2006)

2.3.1 Arquitetura de Serviços WEB

A arquitetura de serviços web possui como parte fundamental o uso das especificações SOAP (GUDGIN et al., 2007) e WSDL. Além disso, é possível destacar quatro aspectos que são tratados pela arquitetura (SOTOMAYOR; CHILDERS, 2006):

- **Processos de Serviços:** Processar os serviços pode envolver outros diversos serviços web. Aqui estão envolvidos aspectos como, por exemplo a descoberta de serviços, orquestração e coreografia de serviços, entre outros.

- **Descrição de Serviços:** Uma das características dos serviços web é a auto-descrição. Isso quer dizer que, uma vez localizado o serviço que se deseja utilizar, é possível pedir que esse serviço forneça sua auto-descrição informando quais operações o mesmo suporta e como deve ser feita sua invocação. Essa descrição é feita através da linguagem WSDL.

- **Invocação de Serviços:** A utilização de um serviço web envolve a descoberta do serviço, sua descrição e como ele deve ser invocado. A comunicação que será feita entre o cliente e o servidor requer trocas de mensagens. Essas mensagens são criadas de acordo com o protocolo SOAP que especifica como deve ser o formato das requisições para o servidor e, a maneira com que o servidor deve enviar as respostas para o cliente.

- **Transporte de Mensagens:** Todas as mensagens que necessitam ser transmitidas entre o servidor e o cliente podem ser transmitidas utilizando diversos protocolos, porém o protocolo HTTP é o mais usual, pois esse é o protocolo para acesso convencional de páginas web na Internet.

2.4 Visão Geral da Arquitetura OGSA

A execução de aplicações em sistemas de grade computacional requer uma melhor integração, virtualização, e administração de recursos e serviços dentro de uma organização heterogênea, dinâmica e distribuída (FOSTER et al., 2002) (FOSTER; KESSELMAN; TUECKE, 2001). Para suprir tais requerimentos, é necessário investir em uma padronização de forma que diversos componentes de um ambiente de computação moderno possam ser descobertos, acessados, alocados, monitorados e etc. A padronização favorece a interoperabilidade, portabilidade e reuso dos componentes e sistemas. Isso também pode contribuir para o desenvolvimento de segurança, robustez e sistemas de grade escaláveis (FOSTER et al., 2005).

A composição de um sistema de grade computacional necessita de alguns componentes básicos, tais como componentes para suporte aos gerenciamento de Organizações Virtuais (OV), serviços de descoberta e gerenciamento de recursos para a grade, serviços de execução e gerenciamento de tarefas e outros grupos de serviços tais como segurança, manipulação de dados, etc (SOTOMAYOR; CHILDERS, 2006). Tais componentes necessitam de uma interação constante, a fim de prover suporte ao funcionamento de uma grade computacional. Diante desse problema, houve a necessidade de uma padronização

destes serviços. Com essa padronização, é possível definir uma interface para cada tipo de serviço. A arquitetura OGSA desenvolvida pelo OGF, define um padrão comum e arquitetura aberta para aplicações baseadas em Grade Computacional.

A padrão OGSA é direcionado para um conjunto de requerimentos funcionais e não funcionais. Esses requerimentos puderam ser observados com a análise de alguns casos de uso em (FOSTER et al., 2004). E esses casos de uso ilustram algumas características e usos típicos de ambientes e aplicações de uma grade computacional. Essas características são interoperabilidade e suporte para ambientes heterogêneos, compartilhamento de recursos entre diversas organizações, técnicas para otimizar a alocação de recursos por parte do usuário e disponibilização desses recursos por parte da organização proprietária do mesmo, garantia da qualidade do serviço (QoS), execução de tarefas, serviços de manipulação de dados, segurança para acesso a recursos e organizações virtuais, redução do custo administrativo visto que é de grande complexidade a administração de uma grade computacional, escalabilidade e disponibilidade para melhorar a tolerância a falhas.

2.4.1 Estrutura dos Serviços OGSA

A arquitetura OGSA contém em termos de definições os serviços, as interfaces desses serviços, o *estado* individual ou coletivo dos recursos pertencentes à esses serviços, e a interação entre esses serviços dentro de uma arquitetura orientada a serviços (SOA). Esses serviços são representados na figura 6 em forma de cilindros. Os serviços são construídos com base nas definições dos padrões de serviços web com semânticas, adições, extensões e modificações que são relevantes para grades.

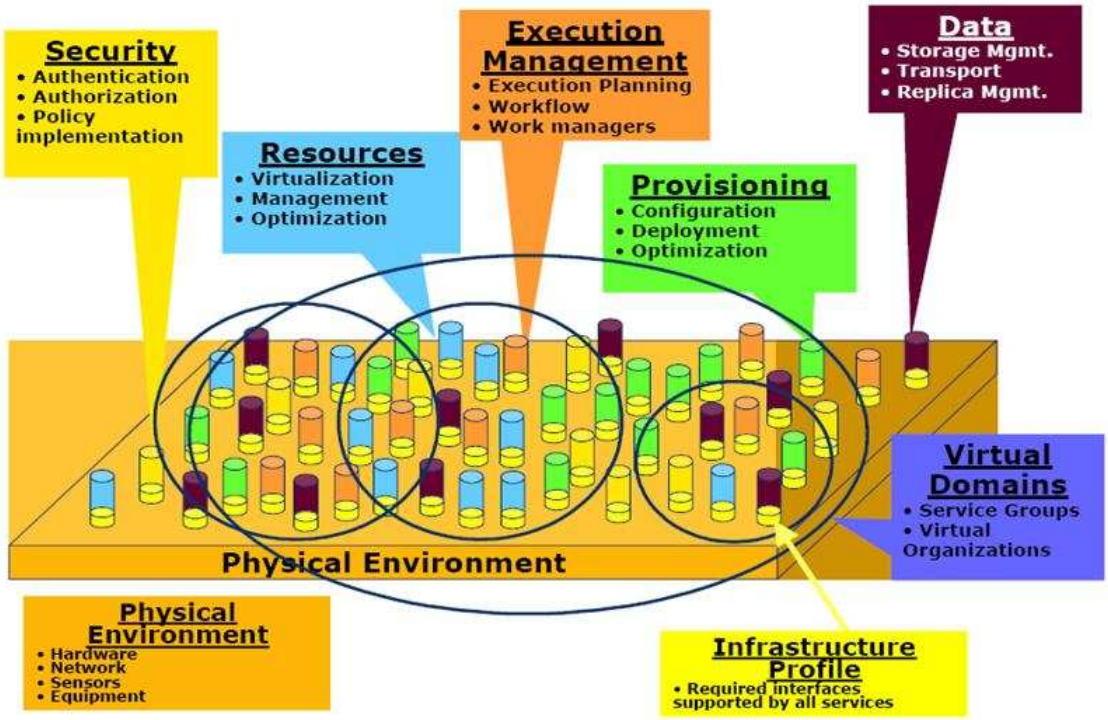
Uma importante motivação para o OGSA é o "Paradigma de Composição", onde um conjunto de capacidades é construído ou adaptado conforme suas necessidades, desde um conjunto minimalista de capacidades iniciais, até satisfazer sua necessidade. Nenhum conhecimento anterior dessa necessidade é assumido. Isto provê uma adaptatividade, flexibilidade e robustez maiores no que diz respeito a mudanças requeridas pela arquitetura.

A definição OGSA provê uma arquitetura plana, ou seja, não é composta por camadas. Isso proporciona que os serviços implementados nessa arquitetura serão livres para existirem isoladamente ou fazerem parte da interação de um grupo de serviços.

Aplicações que utilizam o padrão OGSA podem requerer que seus serviços possuam recursos *statefull*, contudo, os serviços web não são capazes de possuir recursos *statefull* de uma maneira padrão. Com a utilização das especificações WSRF (que será visto mais adiante na seção 2.6) é possível possuir recursos *statefull* em serviços web.

A estrutura da arquitetura OGSA, ilustrado na figura 6, possui diversos grupos de serviços, onde é possível relacionar alguns desses grupos (não todos) disponíveis nessa arquitetura, tais como segurança, recursos, provisionamento de recursos, manipulação de dados, e gerenciamento de execução de tarefas. Neste trabalho será dado um enfoque maior com relação ao gerenciamento de execução de tarefas.

Figura 6: OGSA Framework



Fonte: (FOSTER et al., 2005)

2.5 Serviços de Gerenciamento de Execução

Os serviços de gerenciamento de execução (OGSA-EMS) existente na arquitetura OGSA é destinado a dar suporte para a execução e o gerenciamento de unidades de trabalho. Exemplos de unidades de trabalho podem incluir aplicações OGSA ou aplicações legadas (um servidor de banco de dados, um *servelt* sendo executado em um container de aplicações JAVA, etc). Os EMS definem um conjunto de serviços que tratam do gerenciamento da execução das aplicações em diversos aspectos.

Para a execução de tarefas ou unidades de trabalho em um ambiente de grade computacional é necessário dispor de alguns mecanismos capazes descobrir onde existem recursos disponíveis na grade, para atender as necessidades das unidades de trabalho, tais como CPU, memória, bibliotecas disponíveis, e licenças disponíveis. Com esses recursos conhecidos, é preciso determinar quais serão utilizados, levando em consideração algoritmos capazes de otimizar sua execução e adequação às políticas existentes na organização proprietária do local onde esses recursos estão disponíveis para a execução da tarefa. Antes do início da execução desta tarefa deve-se preparar o ambiente para a execução da mesma, ou seja, efetuar transferência de arquivos quando necessário, e a configuração do ambiente de execução. Uma vez iniciada a execução da tarefa, é preciso monitorar sua execução a fim de identificar alguma falha na execução ou até mesmo providenciar a transferência da tarefa para outro local, preservando o trabalho já executado através de *checkpoints* durante sua execução.

Os serviços EMS devem ser capazes de identificar e resolver os problemas descritos acima. Para isso, são feitos vários agrupamentos compostos de vários serviços, que podem proporcionar a identificação e conseqüentemente a resolução desses problemas. Esses agrupamentos formam classes de serviços que compõem os serviços EMS. Essas classes de serviços ou categorias são denominadas de *Resources*, *Job Management and Monitoring Services*, e o *Resource Selection Services*, que serão discutidos a seguir.

2.5.1 Resources

Resources são compostos por dois serviços. O primeiro é o *Service Container*, que contém entidades de execução que podem ser tanto *jobs* quanto serviços. *Containers* disponibilizam informações sobre as propriedades dos recursos, que podem descrever tanto informações estáticas como, por exemplo, o tipo de executáveis que são aceitos, versão do sistema operacional, bibliotecas instaladas, políticas e ambiente de segurança, como também informações dinâmicas, tais como a carga do sistema e informações sobre a qualidade de serviço (QoS). O segundo é o *Persistent State Handle Service (PSHS)*, encarregado em disponibilizar o caminho para o acesso a informações persistentes através de métodos para se referenciar um *"resource handle"* (RH). Um *resource handle* é um nome abstrato dado a um recurso.

2.5.2 Job Management

Para o gerenciamento de um *Job* é preciso primeiramente definir o que é um *Job*. O OGSA-EMS define um *Job* como sendo a extensão de uma noção tradicional de trabalho, que encapsula todos os conhecimentos necessários sobre a unidade de trabalho, e é também é a menor unidade gerenciada. Cada *job* é nomeado distintamente através de um *resource handle* (RH), que é criado no instante em que é requisitado. Ele permite também que seja gerenciado seu estado de execução (iniciado, suspenso, reiniciado, finalizado e completado). A representação desse *job* pode ser feita por um documento JSDL (ANJOMSHOAA et al., 2005). Todas as definições existentes nesse documento poderão ser acessadas através das propriedades do *job*.

Uma vez definido a menor unidade que pode ser gerenciada, existe um serviço denominado *Job Manager* (JM), que encapsula todos os aspectos de execução de um *job*, ou conjunto de *jobs*, desde seu início até seu término. Esses *jobs* podem estar estruturados (exemplo: um Workflow ou um gráfico de dependência) ou não estruturados (exemplo: um *array* de *jobs* que não se interagem). O JM é responsável por organizar os serviços utilizados para dar início a um *job* ou conjunto de *jobs*, fazendo uma interação com o

Serviço de Execução Planejado que será visto a seguir, configuração do sistema, *containers* e o serviço de monitoramento.

2.5.3 Resources Selection Services

O *Execution Planning Services* (EPS), é um serviço com a finalidade de construir um mapeamento entre tarefas e recursos, chamado "agendamento". Esse agendamento é a relação entre serviços e recursos, e também pode ser considerado como uma lista alternativa, que basicamente diz "se parte desse agendamento falhar, use esta outra". Um EPS não possui a capacidade de promover a execução desse agendamento. A execução do agendamento é feito pelo serviço JM. O EPS usará, para criar seu agendamento, o serviço de informações em conjunto com o *Candidate Set Generator* (CGS). Primeiramente o EPS receberá informações sobre os recursos disponíveis através do CGS, e em seguida buscará informações mais precisas com o serviço de informações para cada recurso, e então construirá seu agendamento com base em alguma função de otimização.

O *Candidate Set Generator* tem como finalidade determinar o conjunto de recursos com o qual a unidade de trabalho poderá ser executada. O CGS irá verificar dentre todos os recursos disponíveis aquele que satisfizer as exigências da unidade de trabalho. Essas exigências estão contidas no documento JSDL que descreve a unidade de trabalho.

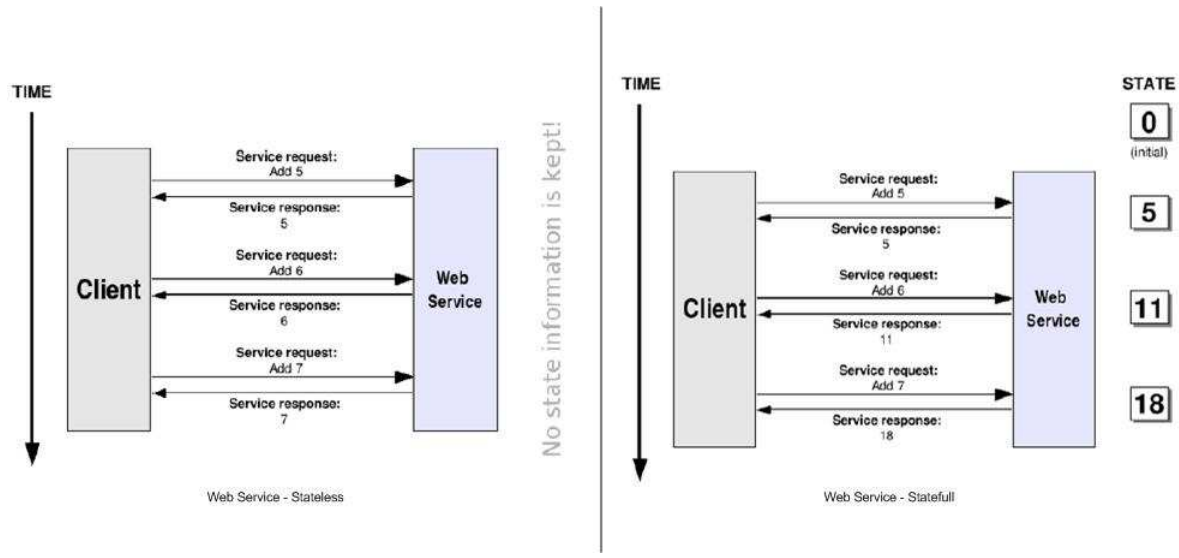
O *Reservation Services*, é um serviço de reserva de recursos que interage com outros serviços, como por exemplo o JM, que faz reservas para um grupo de trabalho que será gerenciado, ou pelo EPS que poderá solicitar reservas de recursos com a finalidade de garantir que o agendamento feito por ele possa ser executado posteriormente. A reserva de recursos é parecida com um documento de acordo que é assinado, podendo ser revogado pela parte solicitante.

2.6 WSRF - Web Services Resource Framework

Um dos elementos centrais da arquitetura OGSA é o serviço de grade (*Grid Service*), que nada mais é do que um serviço web adaptado às condições e ao ambiente de grades computacionais. A principal diferença entre um serviço web e um serviço de grade é que o serviço web não possui um "estado" associado (*stateless*), e serviços de grade podem ter um estado associado aos seus componentes[1] (*Statefull*). Serviços *statefull* podem "lembrar", ou seja, guardar informações entre uma chamada e outra. Um exemplo de estado é ilustrado na figura 7.

Para tratar desses aspectos foi criado o *WSRF*, que originalmente foi proposto pela Globus Aliance e IBM, em conjunto com HP, SAP, Akamai, TIBCO e Sonic, e submetido à *OASIS*. O padrão é atualmente mantido e desenvolvido pela OASIS, com suporte do GGF. Alguns dos principais objetivos são fornecer suporte à criação, endereçamento, inspeção e gerenciamento do ciclo de vida de recursos que possuem estado (*statefull resources*).

Figura 7: Comparativo Serviços WEB e Serviços de Grade



Fonte: (SOTOMAYOR; CHILDERS, 2006)

[1]Recursos capazes de armazenar informações pertencentes a um serviço web.

2.6.1 Resource Properties

Descreve a associação do estado dos recursos vinculados a um serviço web. Essa associação produz um *WS-Resources*. As propriedades dos recursos são utilizadas geralmente para armazenar três tipos de informações, valores para os serviços de dados, valores sobre metadados e informações necessárias para gerenciamento do estado. As propriedades dos recurso são definidas em uma descrição de interface de serviços WSDL.

2.6.2 WS-Resources

Um WS-Resource é um serviço web que representa um recurso dotado de um estado. Esse recurso pode ser acessado através de uma referência remota denominada *endpoint reference* (EPR). Um EPR é um documento que segue as especificações XML capaz de identificar um serviço web e um ou mais recursos associado a ele.

2.6.3 Especificações WSRF

O WSRF é uma coleção de quatro diferentes especificações. Todas relacionadas ou não, com a finalidade de gerenciar *WS-Resources*. A seguir será descrito cada uma delas.

- **WS-ResourceProperties:** Essa especificação possui um conjunto de interfaces que permitirão o acesso, modificações e consulta de propriedades dos recursos.
- **WS-ResourceLifeTime:** Cada recurso possui um ciclo de vida não trivial, ou seja, não é uma entidade estática que é criada quando o servidor inicia e destruída quando o servidor pára. A criação e eliminação de recursos pode ocorrer a qualquer momento. O *WS-ResourceLifeTime* possui mecanismos para gerenciamento do ciclo de vida desses recursos.
- **WS-Service Group:** O *WS-Service Group* é uma coleção heterogênea de serviços web, que pode ser usado para formar uma grande variedade de serviços ou *WS-Resouces*, incluindo registro de serviços e *WS-Resources* associados (MAGUIRE; SNELLING;

BANKS, 2006). Os membros de um *WS-ServiceGroup* são formados por componentes denominados de entradas. A entrada de um grupo de serviço (*WS-ServiceGroup*) é um *WS-Resource*. O serviço web associado a essa entrada do grupo de serviços pode ser composto por uma variedade de padrões de serviços web, incluindo o *WS-ResourceLifeTime*, *WS-BaseNotification* que define como terceiros podem ser informados sobre mudanças no grupo de serviços e o *WS-ResourceProperties*. É possível citar como exemplo o *IndexService* do Globus.

- **WS-BaseFaults:** Define um esquema básico XML para falhas, junto com regras de como essa "falha" deve ser usada em serviços web (LIU; MEDER, 2006). Sua finalidade é de representar falhas ocorridas durante uma chamada a um serviço web. O *WS-BaseFaults* é formado pelos seguintes elementos: **Timestamp**, que indica a hora que a falha ocorreu; **OriginatorReference** que indica a EPR do recurso que gerou a falha; **ErrorCode** é um elemento opcional que pode definir a URI de um código de erro padronizado; **Description** que indica a descrição da falha; **FaultCause** especifica um elemento do tipo *WS-BaseFaults* que indique a causa da falha e **ANY**, que permite a extensão do tipo de falha *WS-BaseFaults*.

2.6.4 A Especificação WS-Notification

WS-Notification é formado por uma coleção de especificações. Embora ela não faça parte das especificações WSRF, está relacionada. Essa especificação permite que um serviço web possa ser configurado como um produtor de notificações, e certos clientes podem ser os consumidores dessas notificações. Isso demonstra que caso um serviço sofra alguma alteração em seu estado, todos os clientes que o utilizam são notificados sobre essa alteração.

2.6.5 WS-Addressing

A especificação *WS-Addressing* provê um mecanismo capaz de endereçar serviços web de uma maneira muito mais versátil se comparado ao URL. Com o *WS-Addressing* é possível endereçar um serviço web juntamente com um recurso (WS-Resource).

2.7 A Plataforma GLOBUS

O Globus Toolkit 4 (GT4) é uma ferramenta composta por conjuntos de serviços, bibliotecas de programação e ferramentas de desenvolvimento que possibilitam a implementação e execução de aplicações para grades computacionais. É comum dizer que Globus é uma implementação de referência da arquitetura OGSA, pois o projeto responsável pelo seu desenvolvimento tem funcionado como um grande laboratório de desenvolvimento de ferramentas e tecnologias para computação em grade. É provável que Globus seja o *middleware* para computação em grade mais utilizado no mundo.

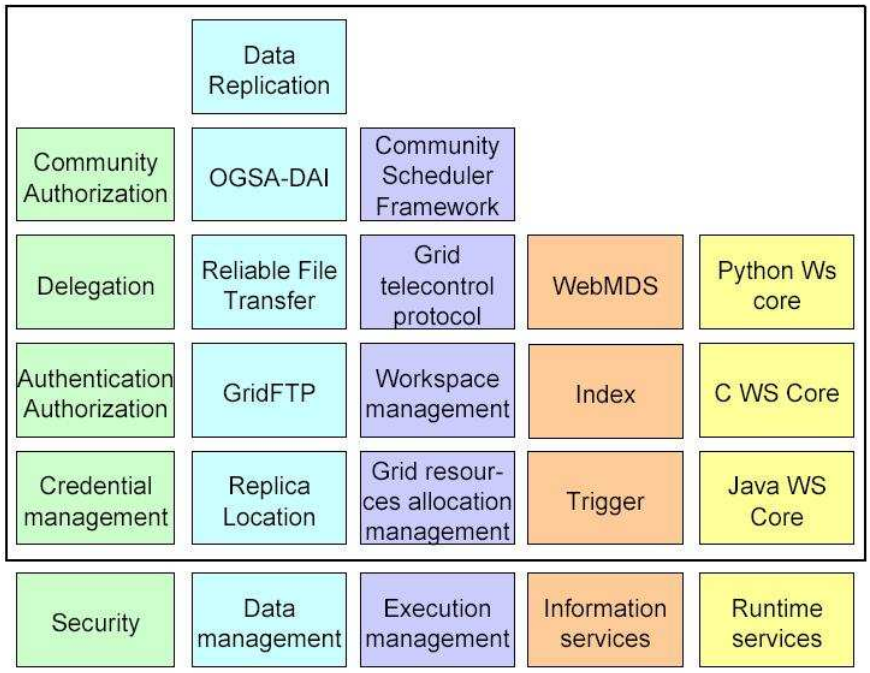
Atualmente a plataforma Globus encontra-se na versão 4 (GT4), que é baseada no padrão WSRF que substituiu o anterior OGSI. Contudo, as versões anteriores do Globus (2.x e 3.x) não são baseadas no padrão atual, e não possuem mais suporte. Tais versões ainda são utilizadas em diversos projetos.

Os componentes que implementam o GT4 podem ser agrupados em cinco grandes áreas que são segurança, movimentação de dados, gerenciamento de execução, serviços de informações e contêineres, conforme ilustra a Figura 8.

2.7.1 Infra-Estrutura de Segurança

Os componentes de segurança do GT4, são conhecidos como *Grid Security Infrastructure* GSI, que facilitam a comunicação segura e criam políticas uniformes entre diversos tipos de sistemas. O GSI implementa também um acesso único na grade. Esse acesso utiliza criptografia de chaves públicas, certificados X.509 e comunicação SSL. O GSI permite estabelecer uma identidade Globus do usuário na grade.

Figura 8: Componentes do GT4



Fonte: (SOTOMAYOR; CHILDERS, 2006)

Os componentes de Autenticação e Autorização *(Authentication and Authorization)* incluem bibliotecas e ferramentas para controlar o acesso a serviços e recursos. É possível contar ainda com um *framework* que possibilita o uso de diferentes métodos de autorização.

O componente *Community Authorization* (CAS), faz o gerenciamento das políticas de autorização de um usuário, para que os recursos de uma organização virtual possam ser utilizados.

O componente *Credential Management* inclui o SimpleCA, que implementa uma autoridade certificadora simplificada capaz de gerar certificados de usuários. Existe também o MyProxy, que é um repositório on-line de credenciais, que possibilita aos usuários a delegação de suas credenciais em um repositório, e a qualquer momento recuperá-la.

2.7.2 Gerenciamento de Dados

Aplicações para grades computacionais que manipulam grandes volumes de informações, necessitam de um componente que possa prover a transferência e acesso desses volumes de informações. Os seguintes componentes e protocolo provêem esse suporte:

- **GridFTP**: Esse protocolo conta com funcionalidade para servidores GridFTP e vários utilitários do lado do cliente. O protocolo GridFTP foi especialmente otimizado para grandes transferências de dados entre servidores.

- **RFT** (*Reliable File Transfer*): É um serviço que utiliza o protocolo GridFTP internamente e pode mover grandes volumes de informações. Ele implementa a transferência confiável de dados, ou seja, caso ocorra alguma falha na transferência esse dados serão retransmitidos. O RFT conta também com a possibilidade de continuação de uma transferência dos dados a partir do ponto em que ocorreu algum tipo de falha.

- **RLS**: O serviço *Replica Location Service* possibilita que os usuários descubram onde diferentes réplicas de um banco de dados estão disponíveis.

- **DRS**: O serviço *Data Replication Service* utiliza os serviços RLS e RFT a fim de garantir a replicação dos dados, para que estejam disponíveis nos locais onde algum usuário necessitar acessá-los.

- **OGSA-DAI**: OGSA *Data Access and Integration* provê o acesso e integração de conjuntos de dados em uma grade computacional. Tais conjuntos podem estar disponíveis em diversos formatos (arquivos textos, base de dados, arquivos XML, etc.).

2.7.3 Gerenciamento de Execução

Os componentes de Gerenciamento de Execução disponibilizam funcionalidades tais como inicialização, agendamento, monitoramento e execução de programas. Essas funcionalidades são descritas a seguir:

- **Grid Resource Allocation & Management (GRAM)**, o GRAM é um dos principais componentes do GT4. Com ele é possível submeter tarefas para execução em uma grade computacional e fazer o monitoramento. O GRAM também faz uso do componente *RFT* para transferência de dados.

- **Community Scheduler Framework (CSF)**, esse componente possui interfaces e ferramentas para que os usuários da grade possam submeter tarefas, criar reservas de recursos avançadas e definir diferentes políticas para agendamento. Utilizando o *CSF* os usuários podem acessar diferentes gerenciadores de recursos, tais como LSF, PBS, Condor e SGE, através de uma única interface.

- **Workspace Management**, um novo componente no GT4 permite que usuários possam criar e gerenciar dinamicamente *workspaces* nos nós remotos.

- **Grid TeleControl Protocol (GTCP)**, esse componente provê uma interface de serviço WSRF para telecontrole (controle remoto de instrumentos).

2.7.4 Serviço de Monitoramento e Descoberta

O serviço MDS *(Monitoring and Discovery Service)* inclui um conjunto de componentes para monitorar e descobrir recursos em uma organização virtual. Esses componentes são descritos a seguir:

- **Index Service**: Esse componente possibilita o agrupamento de diversos recursos de diversas instituições com a finalidade de se formar uma organização virtual (OV), onde esse componente pode ser consultado através de uma consulta do tipo *XPath*.

- **Trigger Service**, muito semelhante ao *Index Service*, o *Trigger Service* coleta dados dos recursos. Contudo, é configurado para executar certas ações baseadas em valores desses recursos, ou seja, ele funciona como um gatilho, que é disparado a partir de um determinado evento.

- **Aggregator Framework**, o *Index Service* e o *Trigger Service* são especializações de um *Framework* genérico. A finalidade do *Aggregator Framework* é agrupar dados coletados de uma ou mais partes, que podem ter como fonte de informações: *WS-Resource Property*, *WS-Notification* ou a execução de um programa fonte (JACOB et al., 2005).

- **WebMDS**, provê uma interface web para visualizar dados coletados pelo *Aggregator Services* do GT4.

2.7.5 Common Runtime

Os componentes *Common Runtime* proporcionam um conjunto fundamental de bibliotecas e ferramentas para armazenamento e disponibilização de serviços já existentes, bem como o desenvolvimento de novos serviços. Esses componentes possibilitam o desenvolvimento de serviços baseados em linguagens como Java (Java WS Core), C (C WS Core) e Python (Python WS Core).

2.7.6 Contêineres

A plataforma Globus disponibiliza contêineres para execução de serviços, cujas interfaces estão definidas em termos de especificações de serviços web. É possível também fazer uso de mecanismos WSRF. Os contêineres Globus permitem a troca de mensagens por meio do protocolo SOAP encapsulado sobre HTTP usando segurança em nível de mensagem e transporte.

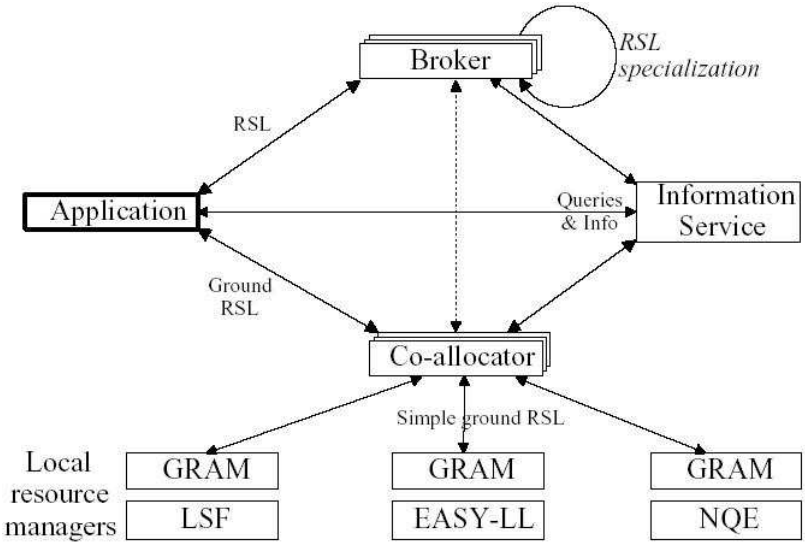
2.8 Submissão de Jobs

Aplicações em grade computacional necessitam de mecanismos que possibilitem a execução de tarefas remotas. Com a execução dessas tarefas, é possível o surgimento de algumas necessidades, tais como: propagação de mensagens de erro, notificação de falhas e transferência de dados de ou para computadores remotos (FELLER; FOSTER; MARTIN, 2007). No GT4 para executar tarefas remotas é necessário o uso do componente GRAM, que define mecanismos para submissão de tarefas (definidas em uma linguagem de descrição de tarefas) e para monitoramento e controle dos resultados dessas execuções (FELLER; FOSTER; MARTIN, 2007). O GT4 possuiu dois mecanismos, que são o *pré WS-GRAM*, proveniente da versão GT2, e o *WS-GRAM*, baseado em serviços web.

2.8.1 Arquitetura de Gerenciamento de Recursos

Um exemplo de arquitetura de gerenciamento de recursos que pode ser utilizada em conjunto com o GT4, pode ser ilustrada na figura 9. Nessa arquitetura, a linguagem RSL, (vide seção 2.9), é usada para comunicação de requisições de recursos entre componentes: da aplicação para os componentes *resource broker*, *co-allocators* e *resources managers*. As informações contidas no RSL são refinadas a cada etapa para serem passadas para outros componentes. Os *brokers* são responsáveis por transformar o RSL de alto-nível em especificações mais concretas, que são chamados de especializações. Os *co-allocators* são responsáveis por repartir a múltipla requisição, isto é, uma requisição de recursos que requer a alocação conjunta de recursos em um ou mais locais. Os elementos são passados para cada componente apropriado o *resource manager*. O *resource managers* recebe as requisições em formato RSL e são responsáveis por traduzi-los para operações locais segundo as especificações de cada gerenciador local de recursos. O *Information Service* é responsável por obter informações de modo eficiente sobre a corrente disponibilidade e capacidade dos recursos. Essa informação é utilizada por exemplo para localizar recursos com uma característica em particular, identificar o gerenciador de recursos associado ao recurso, entre outros.

Figura 9: Arquitetura de Gerenciamento de Recursos



Fonte: (CZAJKOWSKI et al., 1998)

2.9 Ferramentas para Submissão de *Jobs*

A submissão de *Jobs* para grades computacionais pode ser feita através de diversas linguagens ou padrões. Essas linguagens ou padrões podem ser utilizadas diretamente no GT4 ou indiretamente através de um *resource broker*. A seguir serão descritas algumas dessas linguagens.

2.9.1 RSL Resource Specification Language

A especificação da linguagem RSL pode ser resumida conforme a gramática apresentada figura 10. A especificação RSL é constituída pela combinação de especificações de parâmetros simples e condições utilizando os operadores "&" para especificar conjunção de especificação de parâmetros, "|" para especificar a disjunção de especificação de parâmetros ou "+" para combinar duas ou mais requisições em uma simples requisição ou multi-requisição.

O conjunto *parameter-name* é utilizado para especificar o nome dos parâmetros que serão reconhecidos por *brokers*, *co-allocators* e *resource managers* específicos, visto que a linguagem RSL pode ser utilizada por diversos tipos de componentes.

O *Resource Manager*, é um sistema de componentes que interage com o escalonador local, e reconhece dois tipos de *parameters-name*:

- *MDS attribute names*: São usados para expressar as necessidades que uma tarefa possui para a alocação de um determinado recurso. Por exemplo, podem ser especificados *memory>=64*, *network=ATM*. Nesse exemplo é informado que o recurso deve possuir uma memória maior ou igual a 64MB, e que a rede dever ser do tipo ATM. O *parameter-name* é um campo definido na entrada do MDS para que os recursos sejam alocados. A especificação feita através do *parameter-name* será comparada com o valor fornecido pelo MDS.

- *Scheduler parameters*: Usado para comunicação de informações relativas à tarefa, tais como número de computadores requisitados (*count*), tempo máximo requerido (*max_time*), programa a ser executado (*executable*), argumentos (*arguments*), diretório (*directory*) e variáveis de ambiente (*environment*). Os parâmetros de escalonamento são interpretados diretamente pelo *resource manger*.

Figura 10: Gramática para sintaxe de requisição RSL

```
specifications    := request
request           := multirequest | conjunction | disconjunction | parameter
multirequest      := + request-list
conjunction       := & request-list
disconjunction    := | request-list
request-list      := ( request ) request-list | ( request )
parameter         := parameter-name op value
op                := = | > | < | >= | <= | !=
value             := ([a..Z][0..9][_])+
```

Fonte: (CZAJKOWSKI et al., 1998)

Por exemplo, a seguinte especificação requisita 5 máquinas com pelo menos 1GB de memória, ou 10 máquinas com pelo menos 512MB de memória. Nesta requisição *executable* e *count* são nomes de atributos do escalonador, enquanto *memory* é um nome de atributo do MDS:

```
&(executable=myprog)
   (|(&(count=5)(memory>=1024))(&(count=10)(memory=512)))
```

2.9.2 JSDL - Job Submission Description Language

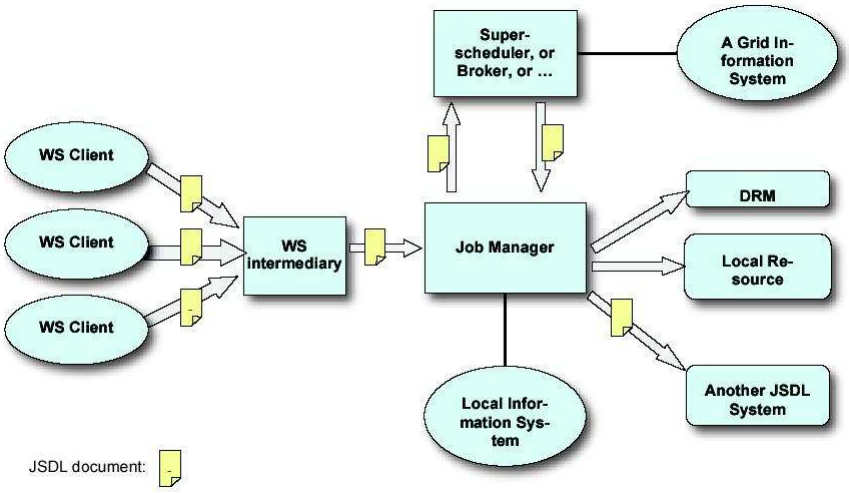
O JSDL é uma linguagem de descrição dos requisitos para a submissão de tarefas em um ambiente de grade computacional. O JSDL contém um vocabulário descrito por esquemas XML para facilitar o detalhamento de cada requisito.

A especificação JSDL é uma recomendação do Open Grid Forum (OGF), com a finalidade de criar uma padronização entre diversos tipos de linguagens de submissão de jobs. Atualmente, diversos sistemas possuem uma linguagem de submissão própria, causando dificuldades de interoperabilidade. Para utilizar diversos sistemas sem qualquer padronização em sua linguagem de submissão de jobs, é necessário que o desenvolvedor crie diversos documentos, um para cada sistema. Através da padronização de linguagens de submissão de tarefas será possível utilizar uma mesma especificação de tarefas em diversos sistemas.

A figura 11 ilustra uma possível visão de componentes que utilizam documentos JSDL para a execução de jobs. A descrição de uma submissão de tarefa representada pelo documento JSDL pode ser transformada por intermediários ou refinada durante sua passagem nas diversas etapas.

O JSDL 1.0 possui um vocabulário para descrição de submissão de tarefas para ambientes de grade. A composição desse vocabulário é destinada para um número existente de sistemas: Condor, Globus Toolkit, Load Sharing Facility (LSF), Portable Batch System (PBS), (Sun) GridEngine(SGE) e Uniform Interface to Computing Resources (UNICORE). O JSDL 1.0 restringe-se somente a possuir informações para a submissão das tarefas. Não existe nenhum elemento definido no JSDL 1.0 que informe qualquer situação da tarefa após sua submissão.

Figura 11: JSDL consumidores em um ambiente de Grade Computacional



Fonte: (ANJOMSHOAA et al., 2005)

2.9.3 APST - A Parameter Sweep Template

APST é uma ferramenta que possibilita a utilização de grades computacionais para aplicações do tipo PSA *(Parameters Sweep Applications)*. Esse tipo de aplicação é tipicamente estruturado como um conjunto de experimentos, onde cada experimento é executado utilizando um conjunto de parâmetros (CASANOVA et al., 2000). Freqüentemente, os termos *Parameters Sweep* e Bag-of-Tasks são considerados sinônimos. O APST permite a utilização de diversos recursos disponíveis na grade computacional e utiliza algoritmos de escalonamento como o *MinMin*, *MaxMin*, *XSufferage*, *Sufferage*, *Workqueue* ou *Round Robin* que serão discutidos no capítulo 3.

A especificação da aplicação no APST é feita através de arquivos que informam os parâmetros de execução, arquivos a serem transferidos, recursos computacionais a serem utilizados entre outros parâmetros. É possível observar um exemplo do arquivo XML para uma aplicação, conforme demonstra a figura 12. Nesse exemplo é possível descrever algumas partes do XML.

O elemento <*storage*> define um apelido para acesso a um diretório especificado pelo elemento <*scp server*>. O elemento <*compute*> define um conjunto de computadores que serão acessados via ssh que poderão acessar os discos definidos pelo elemento

<*storage*>. O elemento <*files*> indica os arquivos de deverão ser copiados para os discos informados pelo <*storage*>. Nesse mesmo elemento <*files*> é especificado também um tamanho aproximado dos arquivos, possibilitando que o APST tome decisões sobre o escalonamento das tarefas. O elemento <*tasks*> informa as tarefas que deverão ser executadas, juntamente com os arquivos de entrada e saída. O elemento <*gridinfo*> informa quais grades computacionais deverão ser utilizadas para a execução das tarefas especificadas no elemento <*tasks*>.

2.10 OurGrid

O OurGrid é uma ferramenta que tem por finalidade formar uma grade computacional, constituída principalmente por laboratórios de universidades de pequeno ou médio porte, espalhados pelo mundo. A agregação de recursos à grade computacional é simples, sendo necessário apenas uma intervenção mínima humana, podendo qualquer um disponibilizar recursos para a grade (OURGRID, 2007). O OurGrid funciona como uma grade cooperativa onde cada laboratório doa o tempo ocioso de suas máquinas para a grade e, em contrapartida, pode utilizar os recursos disponíveis na grade computacional (CIRNE et al., 2006).

A execução de tarefas em um laboratório pertencente ao OurGrid geralmente executa tarefas que foram disparadas por outros laboratórios, também pertencentes ao OurGrid. Esse cenário necessita de mecanismos de segurança para que os recursos locais fiquem protegidos. Essa proteção é fornecida por um componente *Sandbox* denominado *Sandboxing Without a Name* (SWAN). Em geral o OurGrid possui três componentes principais:

- **OurGrid peer**: Cada *peer* é um site integrado ao OurGrid. Cada *peer* se comunica com outros, para formar uma rede de favores. Essa rede de favores funciona de maneira colaborativa, ou seja, para poder utilizar recursos da grade é necessário que haja também uma doação de recursos para a grade, garantindo que o local que doa mais recursos consiga obter mais recursos quando necessitar.

Figura 12: Exemplo de uma Aplicação para APST

```
<apst>
  <storage>
    <disk id="common" datadir="/home/et/work_space">
      <scp server="diskserver.ufo.edu"/>
    </disk>
    <disk id="redDisk">
      <gass server="https://red.ufo.edu:4499"/>
    </disk>
    <disk id="myDisk">
      <local/>
    </disk>
  </storage>
  <compute>
    <host id="blue.ufo.edu" disk="common"><ssh/></host>
    <host id="orange.ufo.edu" disk="common"><ssh/></host>
    <host id="redHost" disk="redDisk">
      <globus server="/O=Grid/O=Globus/CN=red.ufo.edu"/>
    </host>
    <host id="myMachine"><local/></host>
  </compute>
  <files>
    <file id="big_input">
      <copy disk="common"/>
      <copy disk="redDisk"/>
      <copy disk="myDisk" path="/scratch/et/big_input"/>
    </file>
    <file id="big_output" size="16M"/>
  </files>
  <tasks>
    <task executable="myProg"
          input="big_input another_input"
          output="big_output"
          cost="10"/>
    <task executable="myProg"
          input="small_input"
          output="small_output"
          cost="1"/>
  </tasks>
  <gridinfo>
    <infosource id="myMds">
      <mds server="orange.ufo.edu:2135"/>
    </infosource>
  </gridinfo>
</apst>
```

Fonte: (APST, 2005)

- **MyGrid broker**: O usuário interage com o OurGrid através do MyGrid. O MyGrid é um broker que proporciona o escalonamento da aplicação provendo um grau de abstração para o usuário de tal forma que ele esconde a heterogeneidade da grade. O MyGrid interage com o *peer* OurGrid para ter acesso a outros computadores capazes de executar uma determinada tarefa. O grande desafio do escalonador é assegurar um bom desempenho na execução da aplicação. Esse bom desempenho é atingido através da replicação das tarefas em diversos computadores, sendo que, quando qualquer um dos computadores terminar a tarefa, as replicas deverão ser destruídas.

- **SWAN**: Ele isola o código desconhecido em um ambiente de execução denominado *Sandbox* onde ele não consegue ter acesso aos dados locais nem tão pouco a usar rede.

Para o escalonamento das tarefas, é possível trabalhar somente com duas heurísticas *workqueue* ou *storage affinity* (SANTOS-NETO et al., 2004). Essas heurísticas são informadas no momento em que a grade computacional é iniciada. A submissão de tarefas é feita através de um arquivo JDF (Job Description File). O JDF é um arquivo de texto onde são colocados informações de execução para uma determinada aplicação. Essa aplicação é dividida em três partes *Init*, *Remote* e *Final*. O *Init* define a fase de transferência de arquivos para as tarefas. Nessa fase será transferido tanto os executáveis quanto os arquivos necessários para a execução das tarefas para as diversas máquinas que o escalonador escolher. O *Remote* define o que deve ser executado em cada tarefa. Nessa fase, é disparada a execução de cada tarefa nas máquinas remotas escolhidas pelo escalonador. A fase *Final* define o transferência dos resultados gerados nas máquinas remota para o *broker* que deu início a aplicação. Um exemplo de arquivo JDF é ilustrado na figura 13.

Figura 13: Exemplo de uma Aplicação para OurGrid

```
job :
label  : MultiplicacaoDeMatrizes

task :
init : put exemplos.jar exemplos.jar
remote  : java -cp .:exemplos.jar
          br.edu.ufcg.exemplos.MultiplicacaoDeMatrizes 50
final   : get MultiplicacaoDeMatrizes.resultado
          ./saida/MultiplicacaoDeMatrizes.resultado50

task :
init : put exemplos.jar exemplos.jar
remote  : java -cp .:exemplos.jar
          br.edu.ufcg.exemplos.MultiplicacaoDeMatrizes 10
final   : get MultiplicacaoDeMatrizes.resultado
          ./saida/MultiplicacaoDeMatrizes.resultado10

task :
init : put exemplos.jar exemplos.jar
remote  : java -cp .:exemplos.jar
          br.edu.ufcg.exemplos.MultiplicacaoDeMatrizes 20
final   : get MultiplicacaoDeMatrizes.resultado
          ./saida/MultiplicacaoDeMatrizes.resultado20
```

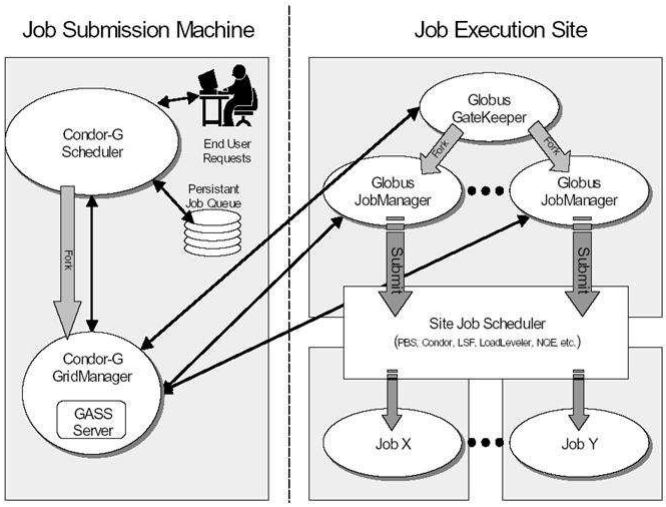
2.11 Condor-G

O Condor-G é um gerenciador de tarefas que faz parte do Condor. Condor-G recebe esse nome pois ele é capaz de interagir com os gerenciadores de recursos, que podem ser o próprio Condor ou GT4. Ele possibilita a submissão de tarefas em uma fila, possui um descritivo sobre a execução de cada tarefa submetida e gerencia os arquivos de entrada e saída. Com isso através do Condor-G é possível submeter tarefas para qualquer grade computacional que possua o Condor instalado ou que essa grade possua uma interface de submissão de tarefas Globus (FREY et al., 2002). Um exemplo dessa funcionalidade é a submissão de tarefas para o Origin2000 situado na universidade de Boston que possui o LSF, e possui um interface Globus.

Vale destacar que o Condor-G não é um sistema gerenciador de grades computacionais, e sim um gerenciador de tarefas. Portanto, para a utilização de uma grade computacional utilizando o Condor-G é necessário que a descoberta e acesso remoto a recursos utilizem padrões. Esses padrões são fornecidos pelos componentes existentes no Globus Toolkit.

Os componentes utilizados pelo Condor-G e fornecidos pelo Globus Toolkit são GRAM, GASS, MDS-2 e GSI. Uma vez que esses componentes são do tipo *Open Source* o Condor-G pode utilizá-los sem problema algum. Um exemplo da execução remota do Condor-G pode ser visualizado na figura 14.

Figura 14: Execução remota via Condor-G em Recursos Globus



Fonte: (FREY et al., 2002)

2.12 GridBus

Com a heterogeneidade dos sistemas de grades computacionais, a utilização de recursos ou serviços não é uma tarefa simples para os usuários, tendo em vista que eles teriam que utilizar diversas formas de acesso a esses recursos ou serviços. Um *broker* de recursos é uma entidade que provê acesso a recursos ou serviços escondendo do usuário a complexidade para acessá-los. O *broker* oferece uma camada de abstração em que o usuário não necessita saber como estes recursos ou serviços serão alcançados. Um *broker*

de recursos pode ser criado pelo próprio usuário, a fim de prover acesso a recursos ou serviços de grades computacionais.

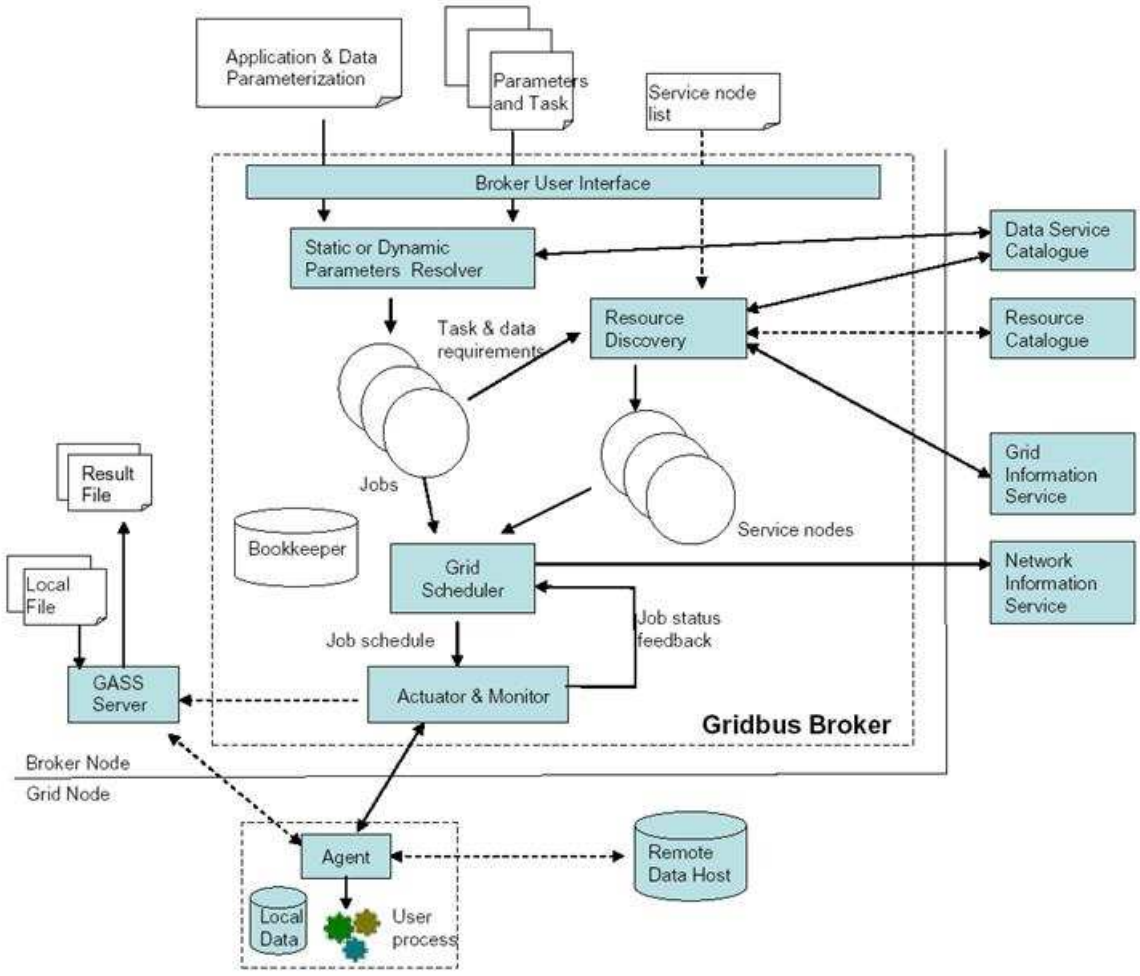
Um grupo de pesquisa da Universidade de Melbourne, na Austrália, desenvolveu uma ferramenta chamada GridBus, que implementa um *broker* e é capaz de trabalhar com o conceito de economias grid. Essa ferramenta foi implementada em JAVA e provê transparência no acesso de nós da grade computacional.

A versão do GridBus analisada neste trabalho foi a 2.4.4. O GridBus broker foi desenvolvido para suportar tanto grades computacionais quanto grade de dados. Sua implementação provê um acesso transparente a vários tipos de *middlewares* e sua utilização pode ser comandada por meio de arquivos do tipo XML que são interpretados e executados conforme seu conteúdo ou através da utilização de API´s que são fornecidas junto com a ferramenta. A figura 15 ilustra a arquitetura do GridBus broker.

A utilização do GridBus se inicia com entrada da descrição da aplicação, o qual consiste na descrição das tarefas e seus parâmetros, e uma descrição dos recursos que serão utilizados para a execução dessas tarefas. Os parâmetros das tarefas podem ser estáticos ou dinâmicos, ou seja, parâmetros dinâmicos são aqueles em que por exemplo são fornecidos intervalos de valores, e o GridBus interpreta esses parâmetros e gera diversas tarefas, porém, cada uma com parâmetros diferentes. A interpretação da descrição dos recursos é feita validando-se cada recursos descrito com a sua disponibilidade na grade e características necessárias para a execução das tarefas. Com o conhecimento das tarefas à serem executadas e os recursos disponíveis, o GridBus inicia o processo de escalonamento dessas tarefas, distribuindo as tarefas em seus recursos disponíveis. O algoritmo de escalonamento é configurado através do arquivo "broker.properties", e pode ser escolhido dentre os seguintes algoritmos: cost, time, costtime, costdata, timedata e round-robin.

Após o início da execução do escalonamento, o GridBus permanece monitorando a execução de cada tarefa nos diferentes nós da grade até o término da execução de todas as tarefas da aplicação. A utilização de um gerenciador de banco de dados MySQL (versão 3.x ou superior) permite que o GridBus armazene informações sobre a tarefa, tais como

Figura 15: Arquitetura do GridBus Broker



Fonte: (VENUGOPAL; BUYYA; WINTON, 2004)

o seu estado, o nó em que esta sendo executada, os arquivos de entrada e saída, entre outras, isso permite maior robustez no caso de falha na execução de alguma tarefa. O tratamento no caso de falha de uma tarefa também pode ser configurado pelo usuário, que pode desejar que a mesma seja reiniciada ou abortada.

O GridBus é uma ferramenta que oferece a possibilidade de se trabalhar ou não com o conceito de economia grid. A idéia de economias grid surgiu com a necessidade de possuir um maior poder computacional por parte de diversos usuários, com a finalidade de executar suas aplicações em um menor tempo. Com a utilização de grades computacionais isso tornou-se mais fácil e conseqüentemente a oportunidade de um negócio. As instituições que possuem um grande poder computacional podem "vender" tempo de pro-

cessamento de suas máquinas. O GridBus implementa essa negociação de valores e tempo de processamento.

2.13 Conclusão

Para a implementação de grades computacionais, é preciso a criação de padrões para estimular a sua disseminação e principalmente a interoperabilidade, visto que uma grade computacional pode ser formada por diversas instituições. Um padrão emergente para arquiteturas de grade é o padrão OGSA. Esse padrão trata de diversos aspectos relativos à construção de grades computacionais. Um aspecto é o gerenciamento de execução de tarefas. A execução de tarefas em uma grade computacional deve proporcionar para o usuário uma facilidade com relação à descoberta de recursos disponíveis, reserva de recursos e mecanismos capazes de monitorar a execução de uma determinada tarefa. Além disso, é preciso gerenciar a alocação de recursos de forma eficiente.

Com o amplo acesso à Internet, uma maneira de agregar poder computacional e proporcionar o acesso a grades computacionais é através da utilização de serviços web. Para a utilização de serviços web em sistemas de grade, existe a necessidade de uma padronização de tal forma que diversos serviços possam ser utilizados por diversas instituições. Essa padronização é definida pelo WSRF, que provê uma série de especificações para serviços web.

O GT4 pode ser utilizado em conjunto com diversas outras ferramentas tais como Condor-G, LSF, PBS, entre outros. A utilização de outras ferramentas em conjunto com o GT4 proporciona facilidades, como por exemplo o uso de um *broker* especializado para escalonamento de aplicações. Uma das funcionalidades mais utilizadas do GT4 é a submissão de *jobs* para serem executados nessa grade.

Existem diversas linguagens que disponibilizam maneiras de especificar uma lista de tarefas para serem executadas. Uma linguagem que tende a tornar-se um padrão para submissão de tarefas para diversas grades computacionais é a linguagem JSDL. Com

a versão 4.1.1 do Globus Toolkit é possível fazer a submissão de *jobs* utilizando-se da linguagem JSDL. Essa versão não suporta todos os elementos disponíveis no JSDL, e uma lista detalhada desses elementos pode ser encontrada em (GLOBUS, 2007). Além disso, o programa de submissão de *jobs* "globusrun-ws" não aceita esse tipo de linguagem, sendo necessário o uso de um programa não oficial (por enquanto) denominado "GlobusRun" que aceita o JSDL.

É importante destacar que para este trabalho algumas ferramentas foram descartadas, pois apresentarem características que dificultam a sua utilização. Sendo que:

- O *broker* GridBus foi descartado porque exigia para a submissão dos *jobs* um conhecimento prévio sobre quais máquinas da grade podem executar as tarefas. Obter tais informações em uma grade computacional é muito difícil, pois a quantidade de recursos pode ser muito grande, além de haver flutuações. Outro ponto importante é que uma vez feito o escalonamento, o mesmo torna-se estático, não permitindo alterações durante sua execução.

- O *MyGrid* possui uma arquitetura basicamente mestre/escravo, de modo que para a implementação da arquitetura hierárquica exigiria alterações consideráveis na sua implementação. Outro ponto importante é que a replicação constante de tarefas entre as máquinas da grade poderia prejudicar o desempenho da aplicação.

- O *Condor-G* não possui um conjunto de API´s desenvolvido em linguagem Java que possibilite fazer uma interface com o Condor-G.

Pelos motivos já mencionados, após uma avaliação das ferramentas, optou-se por utilizar o próprio GRAM, que é um componente do GT4. Essa ferramenta é interessante porque possui vários componentes do tipo *Web Service*, possibilitando a comunicação com diversas linguagens de programação e sistemas operacionais, além de disponibilizar um conjunto de API´s desenvolvidas em linguagem Java, facilitando ainda mais a interação do escalonador proposto neste trabalho com esse conjunto de API´s.

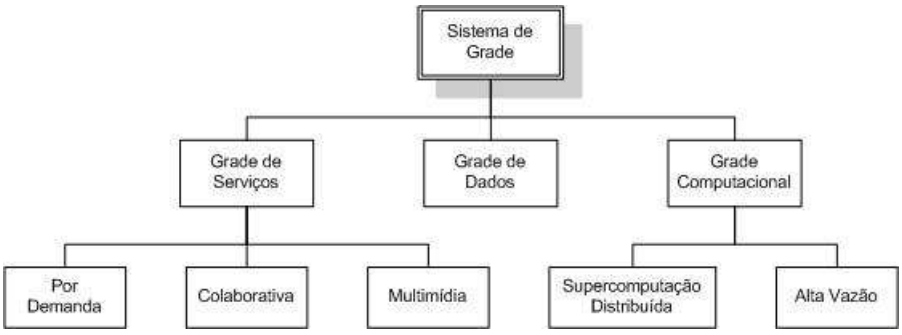
3 Escalonamento de Tarefas em Grades Computacionais

Este capítulo discute o escalonamento de aplicações em arquiteturas como aglomerados de computadores e grades computacionais, com especial atenção ao escalonamento de tarefas independentes que compartilham arquivos (TCA).

Com a popularização da *Internet*, como também a disponibilidade de computadores poderosos e redes de alta velocidade, emerge uma nova maneira de se utilizar computadores. Essa nova maneira consiste na utilização de diversos computadores geograficamente distribuídos como se fossem um único computador, conduzindo ao que se chama computação de grade (BAKER; BUYYA; LAFORENZA, 2002).

A computação em grade pode ser classificada em diversas categorias (KRAUTER; BUYYA; MAHESWARAN, 2002), conforme ilustrado na figura 16.

Figura 16: Taxonomia de Grades Computacionais



Fonte: (KRAUTER; BUYYA; MAHESWARAN, 2002)

A categoria **Grade Computacional** denota sistemas que possuem uma alta capacidade de disponibilizar poder computacional para uma única aplicação agregando diversas máquinas. Dependendo de como essa capacidade é utilizada, esse sistema pode

ser dividido nas categorias Supercomputação Distribuída e Alta Vazão. As grades de *supercomputação* distribuída executam aplicações em paralelo em diversas máquinas, para reduzir o tempo de conclusão da aplicação. Tipicamente, aplicações que utilizam essa categoria são aplicações que necessitam de alto processamento tais como previsões metereológicas e simulações nucleares. Grades de alta vazão aumentam a taxa de conclusão das tarefas como, por exemplo, simulações utilizando o método de Monte Carlo.

A categoria **Grade de Dados** são sistemas que possuem uma infra-estrutura para extrair novas informações a partir de repositórios, tais como bibliotecas digitais ou *data warehouses*, que estão largamente distribuídas pela rede. Grade computacional também necessita prover acesso a dados, mas a maior diferença entre grade computacional e grade de dados é a infra-estrutura especializada para administração e armazenamento de dados desta última.

A categoria **Grade de Serviços** engloba sistemas que provêem serviços que não podem ser executados com o uso de uma única máquina. Essa categoria é dividida em grade sob demanda, grade colaborativa, e grade multimídia. A categoria grade colaborativa conecta usuários e aplicações em grupos de trabalho colaborativos. Esse sistema habilita em tempo real a interação entre pessoas e aplicações, por meio de um espaço de trabalho virtual. A categoria de grade por demanda agrega diferentes recursos para prover novos serviços. A visualização de dados dos trabalhos permite a um cientista aumentar dinamicamente a fidelidade de uma simulação, adicionando mais máquinas para essa simulação de acordo com a necessidade. Isso seria um exemplo de grade sob demanda. A grade multimídia provê um infra-estrutura para aplicações multimídia em tempo real. Esse tipo de grade visa atender a execução de aplicações multimídia que possuem requisitos de qualidade de serviço (QoS), uma vez que máquinas isoladas podem executar aplicações desse tipo sem QoS. Deste ponto em diante quando houver a referência sobre grades entende-se a utilização de grades computacionais.

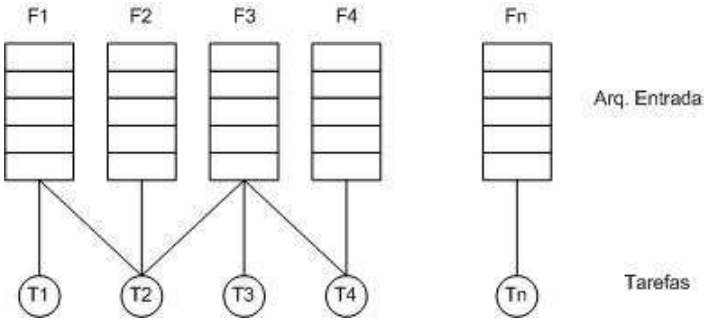
A utilização de grades computacionais possibilita o compartilhamento, seleção, e agregação de uma grande variedade de recursos, incluindo supercomputadores, sistemas

de armazenamento, fontes de dados, e componentes especializados que estão geograficamente distribuídos e gerenciados por diferentes organizações, para resolver problemas de processamento em larga escala nas áreas da ciência, engenharia e negócios (BAKER; BUYYA; LAFORENZA, 2002). Grades computacionais são capazes de executar diversos tipos de aplicações em diversas áreas.

Um tipo de aplicação abordado por este trabalho são as aplicações BoT. Aplicações BoT são compostas por conjunto de tarefas independentes que não se comunicam, e podem ser executadas em qualquer ordem. Pelo fato dessas aplicações não se comunicarem, isso proporciona que o ambiente onde serão executadas não necessite de respostas rápidas com relação à comunicação, portanto, essas aplicações são adequadas para serem executadas em arquiteturas tais como aglomerados de computadores e grades computacionais (CIRNE et al., 2003).

Um tipo particular de tarefas BoT é caracterizada pela relação de compartilhamento de arquivos entre as tarefas (CASANOVA et al., 2000). Nesse tipo de aplicação, cada tarefa pode depender de um ou mais arquivos de entrada, e pode produzir um ou mais arquivos de saída como resultado. Em tais aplicações, cada arquivo de entrada pode ser compartilhado por duas ou mais tarefas, conforme mostra a figura 17. Um exemplo desse tipo de aplicação inclui várias aplicações de mineração de dados. A mineração de dados tem por objetivo descobrir conhecimento em bancos de dados utilizando um processo não-trivial de identificação de padrões (FAYYAD; SHAPIRO; SMYTH, 1996).

Figura 17: Modelo de Compartilhamento de Arquivos



Fonte: (SENGER; SILVA; NASCIMENTO, 2006)

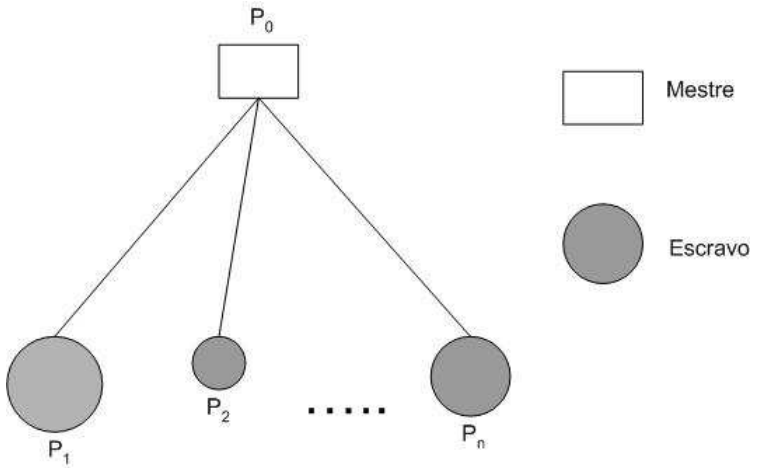
Aplicações BoT também aparecem na literatura sob outras denominações, como por exemplo *Parameters Sweep* (CASANOVA et al., 2000). A inspiração para esse nome vem do fato das aplicações serem estruturadas de maneira a formar conjuntos de múltiplas tarefas, onde cada tarefa se diferencia das outras pelos parâmetros de execução.

A seguir serão discutidas as principais questões relativas ao escalonamento de TCA, suas principais limitações em termos de desempenho, e soluções adotadas.

3.1 Modelo Mestre-Escravo

O modelo mestre-escravo (GIERSCH; ROBERT; VIVIEN, 2006), conforme ilustrado na figura 18, é comumente utilizado em ambientes voltados à execução de aplicações do tipo BoT. Esse modelo é composto por um computador denominado mestre e vários computadores denominados escravos.

Figura 18: Arquitetura Mestre-Escravo



Fonte: (GIERSCH; ROBERT; VIVIEN, 2006)

O nó mestre tem a finalidade de distribuir as tarefas de uma aplicação aos processadores escravos, enviar arquivos necessários, e coordenar a execução da aplicação. Os nós escravos executam tais tarefas e devolvem os resultados ao mestre. Esse modelo possui algumas limitações intrínsecas, principalmente ao que se refere a escalabilidade. Essa limitação se verifica quando o nó mestre ultrapassa o limite de escravos que o mesmo tem a capacidade de gerenciar, tornando-se um gargalo do sistema, e fazendo com que

o desempenho total da aplicação seja prejudicado. Os problemas de escalabilidade serão abordados mais adiante na seção 3.4.

3.2 Escalonamento de Tarefas Independentes

Um dos primeiros trabalhos que tratam do escalonamento de tarefas independentes em aglomerados heterogêneos foi apresentado por Maheswaran (MAHESWARAN et al., 1999). Nesse artigo, os autores propõem seis heurísticas divididas em duas categorias, denominadas *direta* e *em lote*. Esse trabalho teve como base o escalonamento de tarefas independentes, mas desconsiderou o compartilhamento de arquivos. As heurísticas propostas pelos autores faziam com que os arquivos de entrada fossem transmitidos diversas vezes quanto forem necessários para executar as tarefas. Não considerava por exemplo, o fato de um arquivo anteriormente transmitido poder permanecer em um determinado processador para as próximas execuções. As duas categorias de heurísticas denominadas *direta* e *em lote* serão discutidas a seguir.

3.2.1 Heurísticas de Mapeamento Direto

As heurísticas que utilizam técnicas de mapeamento direto, isto é, que identifica em qual processador uma determinada tarefa deverá ser executada, fazem o mapeamento dessa tarefa no exato momento em que é submetida ao escalonador. Uma vez mapeada a tarefa, esse mapeamento não poderá ser modificado.

Para avaliar o funcionamento dessas heurísticas, os autores definiram algumas métricas. A métrica Tempo Esperado de Execução (*Expected Execution Time*) é definida pelo tempo decorrido da execução de uma única tarefa em uma determinada máquina. A métrica Tempo Esperado de Conclusão (*Expected Completion Time*) é definida pelo tempo decorrido da execução da tarefa em uma determinada máquina após o término de todas as tarefas previamente mapeadas para ela.

Maheswaran e colegas criaram as heurísticas: MCT (*Minimum Completion Times*), que mapeia as tarefas para as máquinas com capacidade de completar as tarefas em um menor tempo, MET (*Minumum Execution Time*), que mapeia as tarefas para as máquinas com capacidade de executar as tarefas em um menor tempo, e a heurística SA (*Switching Algorithm*) que é motivada pela seguintes observações: a heurística MET pode provocar um desbalanceamento de carga entre as diversas máquinas, porque máquinas com maior capacidade de processar tarefas podem receber mais tarefas do que outras máquinas com menor capacidade. Considerando que a heurística MCT tenta balancear a carga do sistema mapeando tarefas para máquinas com o menor tempo de conclusão, se nas tarefas que estão sendo mapeadas for utilizado uma combinação dos dois critérios, então será possível utilizar o MET até um ponto em que o sistema fique com sua carga desbalanceada entre as máquinas, para então utilizar o MCT, que fará com que as cargas entre as máquinas fiquem balanceadas novamente. A heurística KPB (*K-Percent Best*), utiliza somente um subconjunto de máquinas para realizar o mapeamento e a OLB (*Opportunistic Load Balancing*), que mapeia tarefas para máquinas que, no menor espaço de tempo estejam prontas para executar outra tarefa.

As heurísticas de mapeamento acima descritas estão fora do escopo desta dissertação, e portanto, não serão mais discutidas.

3.2.2 Heurísticas de Mapeamento em Lote

Em (MAHESWARAN et al., 1999), os autores propõem três heurísticas denominadas *Min-min, Max-min e Sufferage*, para escalonamento de tarefas independentes. Todas essas heurísticas consideram que o sistema é dedicado, ou seja, não é compartilhado por demais usuários ou aplicações.

Essas heurísticas utilizam um algoritmo com uma estrutura lógica padrão, conforme ilustrado na figura 19. Esse algoritmo continua sua execução enquanto existirem tarefas a serem executadas. O laço existente entre as linhas 3 e 7 formam pares de tarefa/processador, avaliando-os com uma função *Objetivo*. Na linha 8 o melhor par de

Figura 19: Algoritmo Genérico para as Heurísticas Min-Min, Max-Min e Sufferage

```
(1) S recebe T (S é o conjunto de Tarefas que ainda não foram Escalonadas)
(2) Enquanto S # 0 faça
(3)   Para cada Processador Pi faça
(4)     Para cada Tarefa Tj pertencente a S faça
(5)       Avalie Objetivo(Tj, Pi)
(6)     Fim Para
(7)   Fim Para
(8)   Pegue a melhor dupla de tarefa Tj pertencente a S e o processador Pi
        de acordo com o Objetivo(Tj, Pi)
(9)   Mapeie Tj para Pi o mais rápido possível
(10)  Remova Tj de S
(11)Fim Enquanto
```

Fonte: (GIERSCH; ROBERT; VIVIEN, 2006)

tarefa/processador é selecionado para o mapeamento, de acordo com o valor resultante da função *Objetivo*. Após o mapeamento, a tarefa mapeada é retirada do conjunto de tarefas S e o algoritmo prossegue executando o laço principal.

Para todas as heurísticas que utilizam esse algoritmo, a função *Objetivo*(T_j, P_i) é realmente o MCT da tarefa T_j se mapeada para o processador P_i. É importante destacar que o modo de avaliar o resultado da função objetivo difere de uma heurística para outra. Com base no valor de retorno da função *Objetivo*(T_j, P_i), as heurísticas propostas pelos autores funcionam da seguinte maneira:

- *Min-min.* A "melhor" tarefa T_j é aquela que possui o menor valor da função *Objetivo*. Isso determina que tarefas menores sejam escalonadas primeiro, evitando que logo no início da execução das aplicações os processadores fiquem ociosos, aguardando por novas tarefas a serem executadas.

- *Max-min.* A "melhor" tarefa T_j é aquela que possui o maior valor da função *Objetivo*. Isso determina que tarefas maiores sejam escalonadas primeiro, fazendo que tarefas longas não sejam deixadas para o final do escalonamento, afim de não atrasar o término de toda a execução.

- *Sufferage*. A "melhor" tarefa T_j é aquela que será menos penalizada, caso a mesma não seja escalonada para o melhor processador, de acordo com a função *Objetivo*. Essa penalização é definida pela diferença do menor valor da função *Objetivo* e o segundo menor valor da função *Objetivo* (isto é, para o melhor processador e o segundo melhor processador disponíveis).

3.3 Tarefas que Compartilham Arquivos

Casanova (CASANOVA et al., 2000) adaptou as heurísticas propostas por Maheswaran, introduzindo a possibilidade de compartilhamento de arquivos de entrada, e propôs uma nova heurística variante do *Sufferage*, denominada *XSufferage*. Essa nova heurística é uma adaptação do algoritmo *Sufferage* para o caso em que a arquitetura não é um mestre-escravo puro. Em vez disso, considerou uma arquitetura composta por múltiplos aglomerados conectados a um nó mestre. Adicionou também a esse algoritmo a possibilidade de compartilhamento de arquivos entre as tarefas, evitando retransmissões de arquivos que já estejam presentes nos nós.

O algoritmo *XSufferage*, diferentemente do algoritmo original, utiliza o valor do MCT de todas as máquinas que compõem os aglomerados. Com a intenção melhorar o tempo total de execução das tarefas, foi considerada a localização física dos arquivos de entrada. Isso possibilita que a tarefa seja direcionada para o aglomerado que já possui seu arquivo de entrada. Contudo, esse tipo de métrica não se mostrou ideal, pois caso o aglomerado possua duas (ou mais) máquinas com desempenho semelhante (o que é bastante comum), essas máquinas teriam o MCT bem próximos, resultando em um baixo valor de *Sufferage*. Isto fará com que a tarefa receba uma baixa prioridade, e conseqüentemente, dê lugar para a execução de outras tarefas.

Esse problema foi solucionado usando uma modificação no cálculo do *Sufferage*. O *Nível de Sufferage do Aglomerado* é calculado pela diferença entre o melhor MCT alcançado em um determinado aglomerado e do segundo melhor MCT dos outros aglomerados. A tarefa com o maior *Sufferage do Aglomerado* recebe prioridade, e será mapeada

no aglomerado que atingiu o menor MCT, sendo executada na máquina que oferecer o melhor MCT desse aglomerado. Maiores detalhes sobre as heurísticas e seu desempenho poderão ser vistos em (MAHESWARAN et al., 1999).

Posteriormente, Giersch et al. (GIERSCH; ROBERT; VIVIEN, 2006) mostraram que a complexidade inerente ao problema de escalonamento de tarefas que compartilham arquivos é $NP - Completo$, propondo novas heurísticas de menor complexidade. Suas heurísticas são baseadas na avaliação de uma função *Objetivo* para classificar previamente uma lista de tarefas, e posteriormente fazer o escalonamento. Uma vez criada essa lista, os pares processador/tarefa são escolhidos de acordo com o MCT calculado (GIERSCH; ROBERT; VIVIEN, 2006). O algoritmo utilizado por essas heurísticas é ilustrado pela figura 20.

As heurísticas propostas observam dois pontos importantes:

1. Alguns arquivos podem ser transmitidos para uma ou várias máquinas escravas, para que essas máquinas possam processar várias tarefas que dependem desses arquivos.

2. Arquivos enviados para certo processador, devem lá permanecer até o final da execução de todas as tarefas, evitando assim a retransmissão. Dessa forma, será possível executar nesse mesmo processador várias tarefas que dependam de arquivos já enviados. Tal procedimento deve proporcionar um melhor desempenho no que diz respeito ao tempo total de transferência de dados, e conseqüentemente, no término aplicação.

A função objetivo proposta pelos autores, é calculada com base em diferentes critérios ou políticas. Inicialmente, será construída uma lista de tarefas, para cada processador. Cada lista será ordenada segundo um dos seguintes critérios, indicado pela função *Objetivo* (T_j,P_i), nas linhas (3) e (5) do algoritmo:

Figura 20: Estrutura para as Heurísticas

```
(1) Para todos os processadores P faça
(2)   Para todas as tarefas T pertencentes ao conjunto de tarefas faça
(3)     Avalie a função Objetivo(T,P)
(4)   Fim Para
(5)   Crie uma lista L(P) com as tarefas ordenadas de acordo com o valor
          da função Objetivo(T,P)
(6) Fim Para
(7) Enquanto restarem tarefas para escalonar faça
(8)   Para todos os processadores P faça
(9)     Pegue a primeira tarefa não mapeada da lista L(P)
(10)    Avalie o MCT(T,P)
(11)  Fim Para
(12)  Pegue o par tarefa/processador T/P minimizando o MCT(T, P)
(13)  Mapeie T para P o mais cedo possível
(14)  Assinale T como Mapeada
(15) Fim Enquanto
```

- A política *Duration* considera somente o tempo de execução global da tarefa, entende-se o tempo global da tarefa como sendo o tempo de transmissão dos arquivos e o tempo de execução da tarefa.

- Na política *Payoff*, enquanto uma tarefa é transmitida para a execução em uma determinada máquina escrava as outras máquinas escravas aguardam o término dessa transmissão. Portanto, essa política é baseada numa relação de benefício/custo, sendo que o benefício é o tamanho da tarefa concluída e o custo é a somatória dos tempos de envio dos arquivos. Essa política pega as tarefas que obtiverem um maior benefício/custo. Como o cálculo dessa política não depende do processador, a ordem de execução das tarefas também não depende do processador.

- Na política *Advance*, para deixar um processador ocupado, é preciso enviar os arquivos da próxima tarefa antes que a tarefa atual seja concluída. Conseqüentemente, o tempo de execução da tarefa atual deve ser maior que o tempo de transferência dos arquivos dessa tarefa. Isso é calculado entre a diferença do tempo de computação e a somatória dos tempos de transmissão dos arquivos. Essa política usa as tarefas em ordem decrescente desse calculo.

- Na política *Johnson*, as tarefas são ordenadas de modo que o tempo de comunicação, ou seja, a somatória dos tempos de envio de cada arquivo, seja menor ou igual ao tempo de execução da tarefa e então essas tarefas são ordenadas em ordem crescente de tempo de comunicação. As tarefas restantes são ordenadas de maneira decrescente em relação ao seu tempo de execução.

- A política *Communication* considera que a comunicação pode ser o gargalo com relação ao tempo necessário de transferência de arquivos para tarefas. Essa política utiliza esses tempos em ordem crescente, para escolher a próxima tarefa a ser mapeada.

- A política *Computation* considera apenas o tempo de execução da tarefa sem levar em conta o tempo de transferência dos arquivos. Essa política utiliza esses tempos em ordem crescente.

Na linha (12) do algoritmo, a escolha do melhor *par* pode utilizar também, os seguintes critérios adicionais:

- *Shared*: indica que quando um arquivo for enviado a um processador ele pode beneficiar todas as tarefas que dependam dele, e que venham a ser executadas nesse mesmo processador futuramente. Essa política considera a relação custo/benefício, onde o custo de transmitir um arquivo é recompensado pelo número de tarefas que dependem dele.

- *Readiness*: quando um processador P_i está pronto para executar uma nova tarefa, o escalonador escolhe, do conjunto de tarefas não escalonadas, aquela em que o arquivo de entrada já se encontra no processador P_i. O escalonador apanha a primeira tarefa da fila que satisfaça esse critério.

- *Locality*: sua finalidade é reduzir a quantidade de replicação dos arquivos. Para isso o escalonador verifica se a próxima tarefa possui os arquivos necessários para sua execução em algum outro processador, então essa tarefa deverá ser escalonada nesse processador.

Como as políticas *Duration, Payoff, Advance, Johnson, Communication* e *Computation* não consideram o compartilhamento de arquivos entre as tarefas, os autores combinam junto a essas políticas os critérios *Shared, Readiness* e *Locality* que tratam do compartilhamento de arquivos. Isso resulta em 44 heurísticas distintas para o escalonamento das tarefas. De acordo com o trabalho descrito em (GIERSCH; ROBERT; VIVIEN, 2006) as 10 melhores heurísticas dentre as 44 resutantes foram: Sufferage, Min-min, Computation + Readiness, Duration + Locality + Readiness, Duration + Readiness, Payoff + Shared + Readiness, Payoff + Readiness, Payoff + Shared + Locality + Readiness, Payoff + Locality + Readiness e Computation + Locality + Readiness.

A seção seguinte discuti alguns aspectos relacionados com a escalabilidade de aplicações.

3.4 Escalabilidade

Para medir o desempenho de uma máquina, permitindo a análise do ganho obtido com o aumento do total de processadores utilizados, são geralmente utilizadas duas medidas usuais: o *speedup* (ganho de desempenho) e a eficiência (SATO; MIDORIKAWA; SENGER, 1996).

O *speedup* (S) obtido por um algoritmo paralelo executando com p processadores é a razão entre o tempo levado por aquele computador executando o algoritmo serial mais rápido (t_s) e o tempo levado executando o algoritmo paralelo usando p processadores (t_p).

$$S = \frac{t_s}{t_p}$$

A eficiência (E) é a razão entre o *speedup* obtido com a execução com p processadores e p. Esta medida mostra o quanto o paralelismo foi explorado no algoritmo. Quanto maior a fração inerentemente seqüencial menor será a eficiência.

$$E = \frac{S}{p}$$

Escalabilidade pode ser definido como a "habilidade do sistema em aumentar seu *speedup* à medida que aumentamos o número de processadores" (GRAMA; GUPTA; KUMAR, 1993). Outra definição que não é baseada no conceito de *speedup* é a seguinte: "A combinação entre máquina-algoritmo é escalável, caso a velocidade média para o algoritmo em uma determinada máquina permaneça constante caso o número de processadores aumente, contanto que o tamanho do problema cresça junto com o tamanho do sistema" (SUN; ROVER, 1994).

O problema de escalabilidade ocorre a partir do momento em que máquinas são adicionadas à grade, e o tempo total de execução dessas aplicações não diminui na mesma proporção. Isso ocorre na arquitetura mestre-escravo.

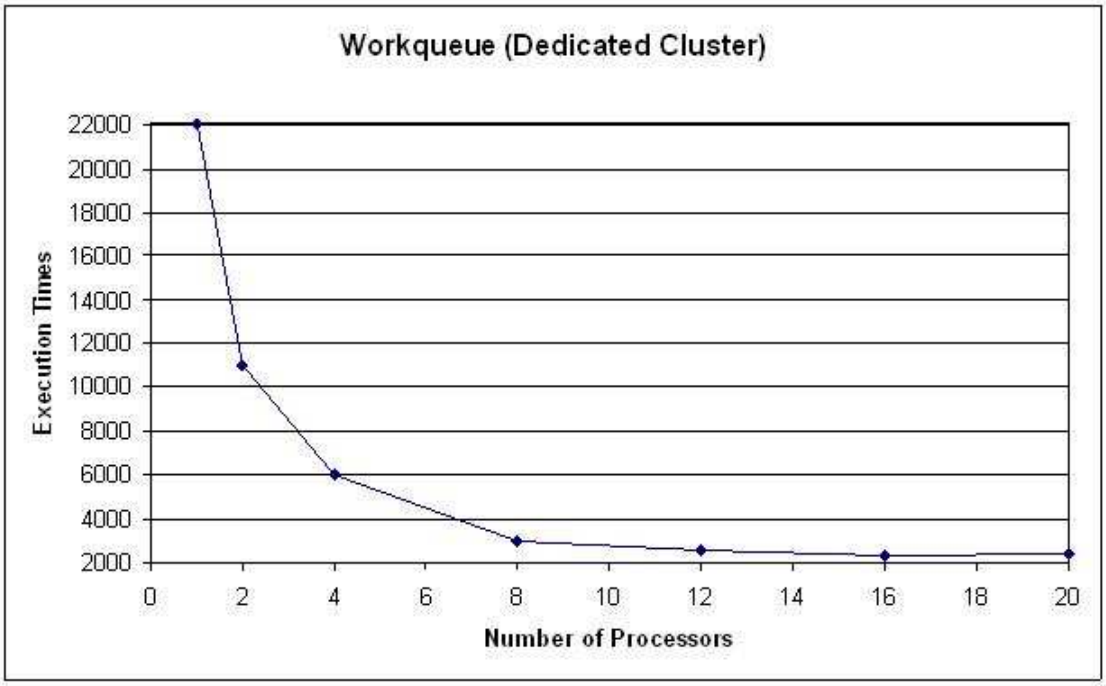
Para ilustrar as limitações de escalabilidade da arquitetura mestre-escravo, é apresentado o comportamento de uma aplicação real que envolve a execução de um algoritmo de agrupamento de dados denominado Cluster Genetic Algorithm (CGA), descrito em (HRUSCHKA; EBECKEN, 2003). O objetivo do CGA é identificar um conjunto finito de categorias para descrever um determinado conjunto de dados, maximizando a homogeneidade dentro de cada categoria e a heterogeneidade entre as diferentes categorias. Assim, objetos que pertencem a mesma categoria deverão ser mais similares entre si do que objetos que pertencem a diferentes categorias. O experimento utiliza um conjunto de dados de referência para aplicações de mineração de dados (o Registro de Votação do Congresso Norte-Americano), disponível no repositório de bases de dados da Universidade da Califórnia em Irvine - *UCI - Machine Learning Repository* (ASUNCION; NEWMAN, 2007). Esse conjunto de dados é composto por 435 ocorrências (267 democratas, 168 republicanos) e 16 atributos com valores lógicos, que representam os votos de cada um dos representantes do congresso americano. Nesse conjunto de dados, 203 ocorrências apresentam falta de valores, e serão removidas do experimento.

Para esse experimento foi utilizada a plataforma MyGrid para sua execução, sendo utilizado como algoritmo de escalonamento o *Workqueue* onde as tarefas são escolhidas para execução arbitrárriamente (CIRNE et al., 2003). Inicialmente um conjunto de tarefas

foi executado seqüencialmente em uma única máquina com um processador Pentium IV (1.8 GHz) com 1 GB de memória principal. Então, o mesmo conjunto de tarefas foi executado em 2, 4, 8, 12, 16 e 20 máquinas dedicadas com características de hardware similares. A figura 21 apresenta o tempo de execução dessa aplicação.

Nesse mesmo exemplo, é possível observar que o tempo total de execução diminui somente até 12 processadores. A partir daí, a adição de processadores não mais reduziu o tempo total de execução dessa aplicação, formando-se um gargalo no mestre. Esse gargalo aparecerá quando o número de escravos exceder a capacidade de entrega de arquivos do mestre, o que irá provocar ociosidade nos escravos. Essa situação pode ser agravada nos casos em que a relação entre o tempo de processamento das tarefas e os tempos de transferências dos arquivos de entrada for desfavorável, isto é, baixa (SENGER; SILVA; NASCIMENTO, 2006). Esses problemas serão discutidos em detalhe mais adiante.

Figura 21: Tempos de execução para um experimento real em um aglomerado dedicado



Fonte: (SENGER; SILVA; NASCIMENTO, 2006)

Para entender as limitações de escalabilidade referente a execução das aplicações, é interessante ver como se dá a execução de tarefas em ambiente como o MyGrid.

1. A fase da inicialização consiste na transferência dos arquivos de entrada, que serão enviados do mestre aos escravos, onde será iniciada a execução das tarefas. A duração dessa fase é denominada t_{init}.

2. A fase da computação é o momento em que a tarefa recebe os parâmetros de execução, processa e gera um ou mais arquivos de saída. A duração desta fase é denominada t_{comp}.

3. A fase da conclusão consiste na transferência dos arquivos de saída localizados nos nós escravos para o nó mestre. A duração desta fase é denominada t_{end}.

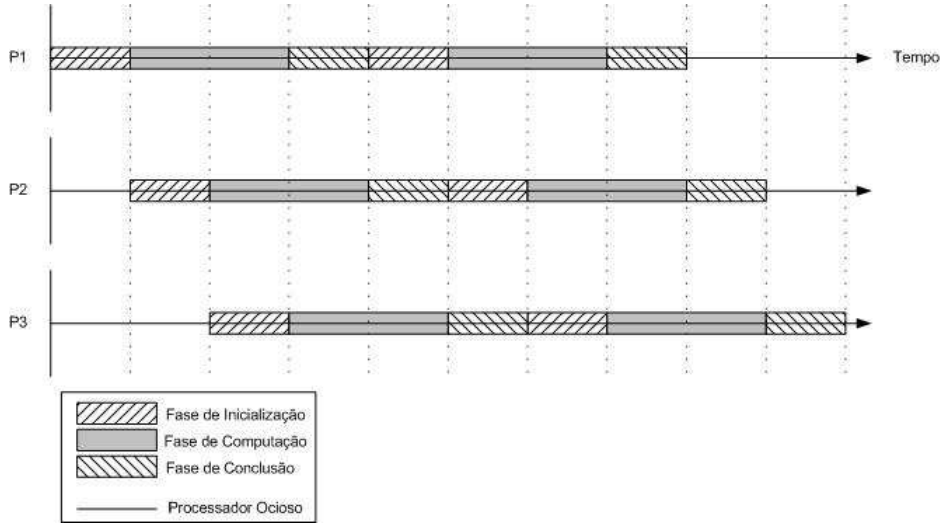
O tempo total de conclusão denominado t_{total} de uma determinada tarefa pode ser obtido com a somatório dos tempos de execução dessas três fases.

$$t_{total} = t_{init} + t_{comp} + t_{end} \tag{3.1}$$

Em situações em que as tarefas produzem resultados de tamanho bastante reduzido é possível unificar as fases de computação e conclusão sem que os resultados sejam afetados, resultando em t'_{comp}, que será utilizado mais adiante. A figura 22 ilustra a execução de tarefas em uma arquitetura do tipo mestre-escravo.

É possível observar na figura 22 que o mestre consegue iniciar uma nova transmissão de arquivos para um determinado escravo somente após o término de uma transmissão já iniciada. Isso ocasiona certo período de ociosidade nos demais processadores. Portando, é preciso definir um número eficiente de processadores escravos para compor a arquitetura de tal maneira que o tempo de ociosidade dos escravos seja diminuído. Em (SILVA; SENGER, 2008), é apresentado o conceito do "Número Efetivo de Processadores" (P_{eff}) como sendo o número máximo de processadores necessários para a execução de uma aplicação sem a existência de períodos de ociosidade nos processadores escravos. A ociosidade de um processador pode ocorrer quando:

Figura 22: Execução de tarefas em arquitetura mestre-escravo utilizando algoritmo *Round Robin*



Fonte: (SILVA; SENGER, 2008)

$$t'_{comp} < (P-1)t_{init} \tag{3.2}$$

O P_{eff} é dado pela seguinte equação:

$$P_{eff} = \left\lfloor \frac{t'_{comp}}{t_{init}} + 1 \right\rfloor \tag{3.3}$$

O número máximo de tarefas que devem ser transmitidas para serem executadas em um determinado processador e dada pela seguinte equação:

$$M = \left\lceil \frac{T}{P} \right\rceil \tag{3.4}$$

Para uma plataforma com P_{eff} processadores, o maior valor do tempo total de execução *(makespan)* será:

$$|t_{makespan}| = M\left(t_{init} + t'_{comp}\right) + (P-1)\, t_{init} \tag{3.5}$$

O segundo termo do lado direito da equação 3.5 mosta o tempo necessário para iniciar as primeiras $P-1$ tarefas nos $P-1$ processadores restantes. Caso seja utilizada uma plataforma onde o número total de processadores é maior do que P_{eff}, o tempo total de execução da aplicação será denominado pelo tempo de comunicação entre o mestre e o escravo. Então a equação seria:

$$\lceil t_{makespan} \rceil = MPt_{init} + t'_{comp} \tag{3.6}$$

Analisando as equações 3.2, 3.3 e 3.6, é possível chegar a duas conclusões:

1. A escalabilidade de uma aplicação é garantida em uma plataforma onde existe uma quantidade de processadores menor do que P_{eff}.

2. Além de P_{eff}, o tempo de ociosidade dos processadores aumenta proporcionalmente com o número de processadores.

3.4.1 Agrupamento de Tarefas

Para melhorar a escalabilidade do sistema, Silva et al. (SILVA; CARVALHO; HRUSCHKA, 2004) propuseram uma estratégia de escalonamento baseada no agrupamento das tarefas que possuam arquivos de entrada em comum. Esse agrupamento torna possível que cada arquivo seja transferido uma única vez para cada processador escravo, reduzindo o gargalo imposto sobre o mestre.

Para desenvolver a heurística de agrupamento, foi proposta por Silva em (SILVA; SENGER, 2008) uma métrica denominada *Input File Affinit* (IFA). O propósito dessa métrica é medir o grau de compartilhamento dos arquivos de um determinado grupo de tarefas, e utilizar essa informação para melhorar a escalabilidade do sistema.

Para ilustrar, considere um conjunto G de tarefas, composto de k tarefas, $G = \{T_1, T_2, \ldots, T_k\}$, e um conjunto F de y arquivos de entrada necessários, pertencente ao

grupo de tarefas G, $F = \{f_1, f_2, \ldots, f_y\}$. I_{aff} é definido como segue:

$$I_{aff}(G) = \frac{\sum_{i=1}^{y}(N_i - 1)|f_i|}{\sum_{i=1}^{y} N_i|f_i|}, \tag{3.7}$$

onde $|f_i|$ é o tamanho do arquivo f_i em bytes e $N_i(1 \leq N_i \leq k)$ é o número de tarefas no grupo G que possui como entrada o arquivo f_i. O termo $(N_i - 1)$ localizado no numerador da equação 3.7 pode ser explicado como segue: se N_i tarefas compartilham o arquivo de entrada f_i, então o arquivo pode ser enviado uma única vez (em vez de N_i vezes) quando o grupo de tarefas for executado em um escravo. A redução do número de bytes transferidos do mestre para os escravos considerando somente o arquivo de entrada f_i é então $(N_i - 1)|f_i|$. Assim, o *IFA* indica uma redução geral do montante de dados que precisa ser transferido para os escravos, quando todas as tarefas do grupo forem enviadas para o nó. Observe que $0 \leq I_{aff} < 1$. Mais precisamente, $0 \leq I_{aff} \leq \frac{k-1}{k}$Um valor de *IFA* igual a zero indica ausência de compartilhamento de arquivos entre as tarefas do grupo. Se o valor de *IFA* for igual a $\frac{k-1}{k}$ indica que todas as tarefas do grupo compartilham o mesmo arquivo, tendo um alto grau de compartilhamento de arquivo.

3.4.2 A Heurística de Agrupamento

Esta seção descreve a estratégia denominada Agrupamento Dinâmico, que utiliza o IFA (SILVA; SENGER, 2008). O objetivo desse algoritmo é criar um agrupamento inicial baseado somente em informações estáticas, e então as tarefas serão distribuídas entre os nós levando em conta a sua taxa de conclusão, que por sua vez, dependerá da carga individual de cada processador. A descrição desse algoritmo pode ser observada na figura 23 e explicada a seguir:

- (1) Cria um grupo de tarefas utilizando a métrica IFA. Conforme exemplificado na figura 24
- (2) Quando tarefas são enviadas aos processadores, os arquivos de entrada do grupo de tarefas são enviados de uma maneira enfileirada, sempre sobrepondo as fases de

Figura 23: Algoritmo de Agrupamento Dinâmico

```
(1)  Agrupar as Tarefas de acordo com o Input File Affinity
(2)  Escalonar os grupos de tarefas no processadores escravos
(3)  Esperar Pi executar as tarefas;
(4)  Quando Pi retornar resultados após completadas x tarefas
     {
(5)    Calcule o Tempo de Execução no processador Pi
(6)    Atualize a lista de tarefas
(7)    Cancele a execução de réplicas que ainda estão sendo executadas
(8)    Se o processador Pi estiver ocioso
(9)      Se existe ainda grupos não replicados em processadores mais lentos
(10)       Replique grupos que não terminaram para o processador Pi
     }
```

comunicação e computação quando possível. O nó mestre somente envia o arquivo para o escravo caso o arquivo não estiver armazenado no nó escravo, com a finalidade de diminuir o uso da banda da rede. A tarefa executada primeiro será aquela em que a soma dos *bytes* do arquivo de entrada for menor, com a finalidade de iniciar a execução no escravo o mais cedo possível. Veja que isso também reduz o t_{init} quando o arquivo é transmitido seqüencialmente.

- (4) Para cada x tarefas completadas (onde x é um parâmetro ajustável), a máquina escrava deverá enviar de volta os resultados correspondentes para o nó mestre. Esse mecanismo é parecido como um *checkpoint*. Se a máquina escrava falhar, as únicas tarefas que deverão ser executadas serão aquelas cujos resultados ainda não foram recebidos. Com isso, também é possível obter informações atualizadas sobre a situação da carga de uma máquina, medindo o número de tarefas que restam para serem executadas.

- (8) Se uma máquina ficar ociosa, envie uma réplica do restante das tarefas (ainda não replicadas), da máquina que possuir o maior número de tarefas ainda não concluídas para a máquina que estiver ociosa. Exemplos sobre o uso de replicações para escalonamento de tarefas do tipo BoT são descritos em (SANTOS-NETO et al., 2004) e (SILVA; CIRNE; BRASILEIRO, 2003).

- (10) Se o processador P_i estiver ocioso, o mestre identificará o processador P_s como sendo aquele que processou a menor parte de suas tarefas, e replicará algumas das tarefas não iniciadas para esse processador P_i. As tarefas a serem replicadas serão escolhidas de uma maneira que o IFA seja maximizado.

No mesmo artigo é proposto uma heurística de agrupamento:

Figura 24: Heurística para Agrupamento de Tarefas Utilizando a Métrica IFA

```
(1)  Para todos os processadores defina o número de tarefas a ser mapeado
(2)  Para todas as tarefas
(3)    Calcule o total de bytes para os arquivos de entrada (Ifile-sum)
(4)  Classifique os resultados do passo (3) de acordo com o (Ifile-sum)
na lista L
(5)  P = número de processadores
(6)  Para todos os grupos definidos no passo (1)(na ordem decrescente do
número de tarefas, começando pelo maior grupo)
(7)    Mapeie a tarefa com o menor (Ifile-sum) ainda não mapeada para
o grupo
(8)    Posição = 1
(9)    Enquanto o grupo é completado faça
(10)     Se Iaff(L[Posição],L[Posição+1]) < Y então
    Posição = (Posição + P) MOD(tamanho de L)
(11)     Se não Posição = (Posição + 1) MOD(tamanho de L)
(12)     Mapeie para o grupo a tarefa localizada na Posição na lista L
(13)   Fim Enquanto
(14)   Remova as tarefas mapeadas da lista L
(15)   P = P - 1
```

- (1) Defina o número de tarefas que serão atribuídas para cada processador dependendo de sua velocidade relativa, baseada na velocidade média dos processadores do aglomerado. Por exemplo, se a velocidade relativa de um processador for igual a 1.0 (igual à velocidade média dos processadores escravos do aglomerado), o número de tarefas que será atribuída para cada processador será no máximo $\left\lceil \frac{T}{P} \right\rceil$. Vale notar que isso é somente uma primeira aproximação, que será ajustada posteriormente pela heurística.

- (3) Calcule a soma total dos bytes de todos os arquivos necessário para a execução de cada tarefa no processador escravo ($Ifile_{sum}$).

- (4) Classifique todos os resultados calculados por $Ifile_{sum}$. Valores $Ifile_{sum}$ menores devem aparecer no topo da lista.

- (6) As tarefas são agrupadas neste laço. Nessa heurística há no máximo um grupo por processador em qualquer momento durante a execução. Logo no início da execução o número de grupos é igual ao número de processadores escravos, pois inicialmente existe um grupo por processador. A primeira tarefa com o menor $Ifile_{sum}$ não mapeada para um grupo é selecionada. Isso é feito de maneira a minimizar o tempo necessário para iniciar a execução da primeira tarefa no processador escravo.

- (10) Se o conjunto de arquivos associado para a próxima tarefa na lista criada no passo (4) possuir o IFA maior que β para o conjunto de arquivos pertencente à tarefa recém mapeada, então a próxima tarefa na lista é mapeada para o mesmo processador. Veja que β é um parâmetro variável, mas normalmente ele deverá ser maior que $0,5$. A razão para se fazer isso é maximizar o IFA (I_{aff}) dentro do grupo. Então é evitado o envio de conjuntos similares de arquivos para múltiplos processadores. Se o conjunto de arquivos pertencentes à próxima tarefa for diferente o bastante (por exemplo o IFA (I_{aff}) entre os dois conjuntos de tarefas for menor que $0,5$), então a tarefa localizada P posições a frente da tarefa recém mapeada será selecionada. Isso é feito por duas razões: primeira, garantir que as P tarefas com $Ifile_{sum}$ pequenos sejam disparadas primeiro para P processadores. Segundo, para garantir que os grupos sejam o mais uniforme possível relativo ao número de bytes que deverão ser enviados do nó mestre para os escravos. Vale lembrar que a transferência dos arquivos de entrada é feita de maneira enfileirada, sobrepondo computação em comunicação quando possível.

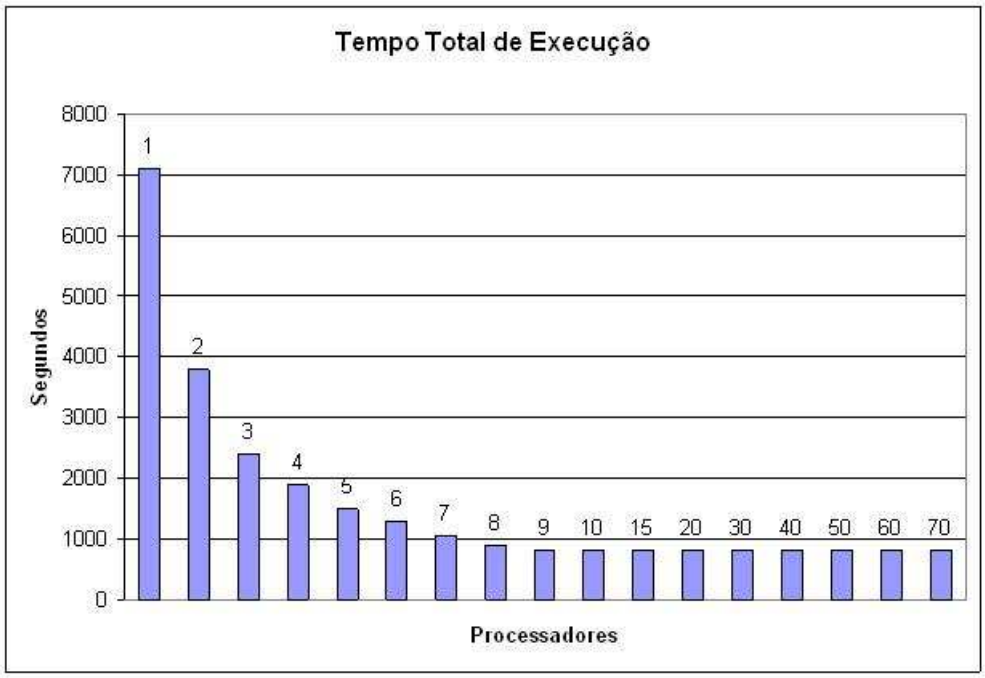
Note que a complexidade da heurística ilustrada na figura 24 é determinada pelos passos (3), (4) e (10). O laço representado pelo passo (3) calcula o $Ifile_{sum}$ para todas

as tarefas. Essa operação pode ser implementada com baixa sobrecarga se for mantido, para cada tarefa, uma lista que represente o conjunto de arquivos que essa tarefa depende. Assim, a execução do passo (3) pode ser implementada processando-se todas as T listas (uma única vez) em tempo $O(T+D)$, onde D é o número total de relações de dependência entre arquivos e tarefas da aplicação e T é o número de tarefas. Como o passo (4) classifica as tarefas de acordo com o $Ifile_{sum}$ sua complexidade é $O(T.logT)$. A complexidade dos passos (6) até (13) é dominada pelo passo (10), com o qual calcula o I_{aff} para todos os pares de tarefas adjacentes na lista. Se ΔT denota o número máximo de arquivos que uma tarefa depende, então o cálculo do I_{aff} para um único par de tarefas pode ser executado em tempo $O(\Delta T)$, e o tempo para todos os pares pode ser executado em tempo $O(T.\Delta T)$. Então, a complexidade da heurística acima é $O(T+D+T.logT+T.\Delta T)$. Além disso, é notado que $T.\Delta T \geq D$ e $T.\Delta T \geq T$. Então, a complexidade da heurística pode ser simplificada como $O(T.logT + T.\Delta T)$. Desde que a complexidade do algoritmo de Agrupamento Dinâmico é denominado pela função responsável por agrupar as tarefas, a complexidade do algoritmo de Agrupamento Dinâmico é também $O(T.logT + T.\Delta T)$. É preciso enfatizar que a heurística ilustrada na figura 24 é apenas uma possível heurística para agrupamento de tarefas utilizando a métrica IFA. Outras heurísticas são possíveis, mas o objetivo global de qualquer heurística deverá ser a criação de grupos com o máximo IFA e a menor complexidade.

A seguir, a figura 25, ilustra a execução de uma aplicação sem o uso do agrupamento, onde os autores utilizaram o simulador SimGrid com os seguintes valores, $t_{init} = 1$, $t'_{comp} = 8$ e $T = 800$. A execução dessa aplicação confirma os resultados obtidos através das equações 3.2, 3.3, 3.4, 3.5 e 3.6, que $P_{eff} = 9$. Realmente, para $P \geq 9$ o tempo global de execução é constante e igual a 808 segundos.

A figura 26, representa a simulação de uma aplicação onde as tarefas compartilham os mesmos arquivos de entrada, considerando os mesmos parâmetros de entrada ilustrados na figura 25. Contudo, é utilizado o agrupamento de tarefas.

Figura 25: Simulação do tempo total de execução em uma plataforma homogênea



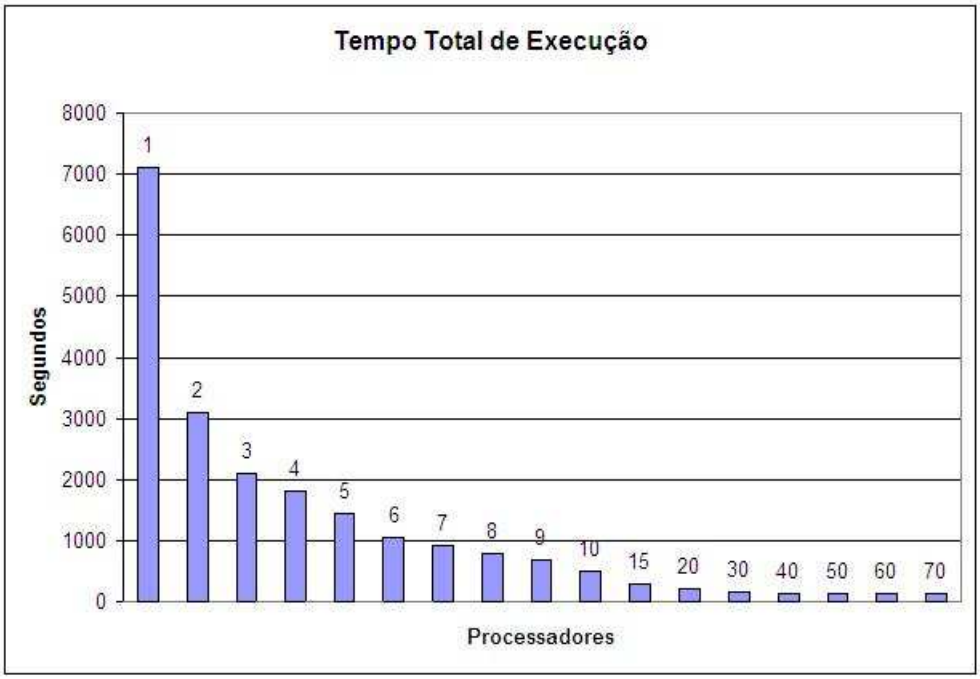
Fonte: (SILVA; SENGER, 2008)

3.4.3 Modelo Hierárquico

O escalonamento hierárquico (SENGER; SILVA; NASCIMENTO, 2006), foi proposto com a finalidade de melhorar a escalabilidade na arquitetura mestre-escravo, uma vez que para efetuar o controle das tarefas e transferência dos arquivos para os escravos o mestre torna-se um gargalo. A estratégia é dividida em três partes: inicialmente é feito um agrupamento das tarefas que compartilham arquivos, esse agrupamento utiliza a métrica IFA proposta em (SILVA; SENGER, 2008); em seguida, os processadores disponíveis são organizados de acordo com uma hierarquia; e por último é feito o mapeamento dos grupos de tarefas para a hierarquia dos processadores. A figura 27 ilustra uma visão da hierarquia de processadores.

O modelo apresentado a seguir ajuda a compreender melhor o funcionamento e os ganhos que podem ser obtidos com o escalonamento hierárquico. Seja RT o tempo de resposta para a execução de uma tarefa, que é dado pela soma do tempo médio de execução (ET) das tarefas, o tempo médio necessário para transmissão dos arquivos de

Figura 26: Simulação do tempo total de execução com agrupamento em uma plataforma homogênea



Fonte: (SILVA; SENGER, 2008)

entrada ($TTIF$), e o tempo médio necessário para a transferência dos arquivos de saída ($TTOF$):

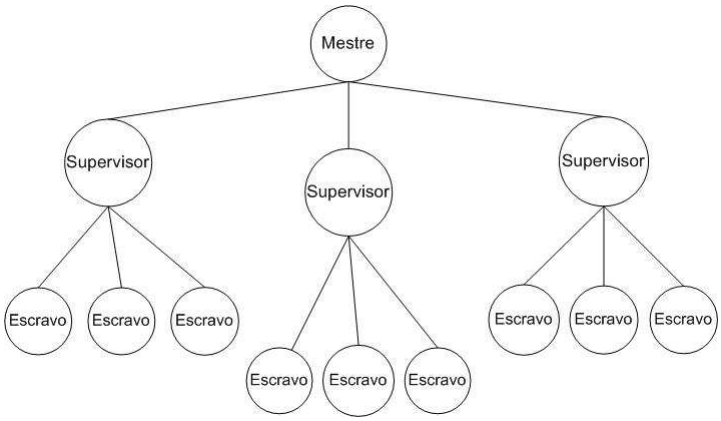
$$RT = TTIF + ET + TTOF \tag{3.8}$$

Nesse modelo, considera-se que os arquivos são transmitidos seqüencialmente para os processadores escravos. Nesse cenário, o número máximo de processadores escravos sem a perda de desempenho do sistema (S_{eff}), pode ser estimado como:

$$S_{eff} = \frac{RT}{TTIF + TTOF} \tag{3.9}$$

Utilizando a estratégia de agrupamento, é possível melhorar a escalabilidade de um sistema como visto anteriormente. Assim, adaptando a equação 3.9 para a utilização de agrupamento, uma vez que essa equação não leva em conta o compartilhamento de

Figura 27: Modelo Hierárquico



Fonte: (SENGER; SILVA; NASCIMENTO, 2006)

arquivos, o tempo médio de resposta para um grupo de tarefas que compartilham arquivos (RT_{job}) é calculado por:

$$RT_{job} = TTIF + NT_{job} * (ET + TTOF) \tag{3.10}$$

onde, (NT_{job}) é o número de tarefas agrupadas para um *job*, ($TTIF$) é o tempo necessário para transmissão dos arquivos de entrada (o qual é o mesmo para todas as tarefas desse *job*), e ($TTOF$) é o tempo necessário para transmissão dos arquivos de saída produzidos por cada tarefa do *job*. Assim, o número efetivo de processadores escravos que pode ser efetivamente controlado pelo mestre é:

$$S'_{eff} = \frac{RT_{job}}{TTIF + NT_{job} * TTOF}. \tag{3.11}$$

Com a arquitetura hierárquica, pode-se aliviar o gargalo que se forma no mestre quando se utiliza a arquitetura mestre-escravo. Com a arquitetura hierárquica, a redução do gargalo é obtida através da redução do número de transferências e ações de controle.

Para reduzir a carga de trabalho no mestre, utiliza-se um número adicional de máquinas supervisoras para a arquitetura mestre-escravo. O supervisor é responsável pelo controle da execução da aplicação bem como a transmissão de arquivos para os nós escravos. O mestre agrupa as tarefas, formando unidades de execução maiores, compostas

por tarefas que compartilham arquivos, denominada *jobs*, que serão distribuídos entre os diversos supervisores. Em troca, cada supervisor é responsável pelo controle da execução desse *job* em um subconjunto de maquinas escravas. Nesse modelo, não existe interação entre os escravos e o mestre. Em vez disso, o mestre delega *jobs* para os supervisores, que são processadores responsáveis pela comunicação com os escravos e pelo gerenciamento da execução das tarefas.

3.5 Estratégia para Escalonamento Hierárquico

Primeiramente, considere uma arquitetura distribuída composta por um nó mestre, uma coleção de P processadores pertencentes a C aglomerados inter-conectados por links de comunicação. A aplicação é composta por T tarefas, onde T é pelo menos uma ordem de grandeza maior que $P(T >> P)$. Essa coleção é dividida em dois conjuntos. O primeiro é composto por S computadores escravos, e o outro é composto por N computadores supervisores. Dentro dessas condições, propõem-se os seguintes passos para ser cuidado pelo nó mestre, para o início da execução da aplicação.

1. Inicialmente o mestre obtém informações estáticas sobre todos os recursos disponíveis, como por exemplo, o número e identificação de computadores disponíveis no sistema, velocidade dos processadores, e memória. Ao final desse passo o parâmetro P é conhecido.

2. Para cada aglomerado conhecido é calculado um número N_i de supervisores, de modo que cada aglomerado deverá possuir pelos menos um supervisor. Caso se verifique a existência de um número elevado de processadores em um aglomerado é possível definir mais do que um supervisor no mesmo aglomerado, a fim de manter a eficiência. Mais precisamente, o número N_i de supervisores para o *i-ésimo* aglomerado é calculado como:

$$N_i = \left\lceil \frac{P_i}{S'_{eff}} \right\rceil, \tag{3.12}$$

onde, P_i é o número de processadores no *i-ésimo* aglomerado, e S'_{eff} é o número máximo de processadores que um supervisor pode controlar eficientemente quando o agrupamento de tarefas é aplicado. Então se obtém N, como o número total de supervisores no sistema como:

$$N = \sum_{i=1}^{C} N_i. \tag{3.13}$$

3. Calcular o número de escravos $S = P - N$.

4. Em seguida, agrupam-se as tarefas em S unidades de execução, tal como proposto em (SILVA; SENGER, 2008), e então cada *job* pode ser mapeado para um supervisor e seus escravos. Cada *job* agrupará um número (no intervalo $[\lfloor T/S \rfloor, \lceil T/S \rceil]$) de tarefas que compartilham arquivos em comum. O mapeamento da tarefa para os processadores pode ser feito aleatoriamente, ou por meio de alguma heurística. Por exemplo, a granularidade (isto é, número de tarefas) de cada *job* pode ser ajustado de acordo com a capacidade dos processadores escravos. O resultado desse passo é uma lista contendo a identificação da tarefa e o processador escravo no qual ela foi mapeada.

5. Decompor a lista de *jobs* em N sub-listas, em que cada sub-lista corresponda a um supervisor e contenha os *jobs* mapeados para os processadores escravos de seu controle. Esse passo pode ser realizado em tempo $O(N)$, selecionando a sub-lista para qual cada tarefa que deve ser movida. Para cada supervisor, é enviado uma sub-lista contendo informações dos *jobs* que acabaram de ser mapeados para ele, e transmitido os arquivos necessários (somente uma vez).

6. Esperar até que as tarefas sejam executadas e os arquivos de saída retornados.

Tão logo o supervisor receba uma lista de *jobs* e os arquivos de entrada, serão distribuídas as tarefas para os processadores escravos. Cada vez que o supervisor for notificado de que uma tarefa foi concluída, deverá executar os seguintes passos:

1. Alterar o estado da tarefa para Terminado.

2. Encaminhar a notificação recebida e os resultados para seu respectivo mestre.

3. Enquanto houver tarefas a serem executadas, isto é, no estado Pronto, cujos arquivos de entrada já tenham sido transmitidos para o escravo ocioso, o supervisor mapeia a próxima tarefa para o processador que estiver ocioso.

4. Quando não restarem mais tarefas a serem executadas, o supervisor deve procurar por tarefas que estejam Executando, e cujos arquivos de entrada já tenham sido transmitidos para o escravo ocioso, e cria uma réplica dessa tarefa no processador ocioso. A replicação pode melhorar a probabilidade dessa tarefa ser concluída mais cedo, bem como algum nível de tolerância a falhas

5. Quando os passos 3 e 4 estiverem completos (todas as tarefas foram concluídas), o supervisor solicita para o nó mestre outras tarefas incompletas, para outros processadores.

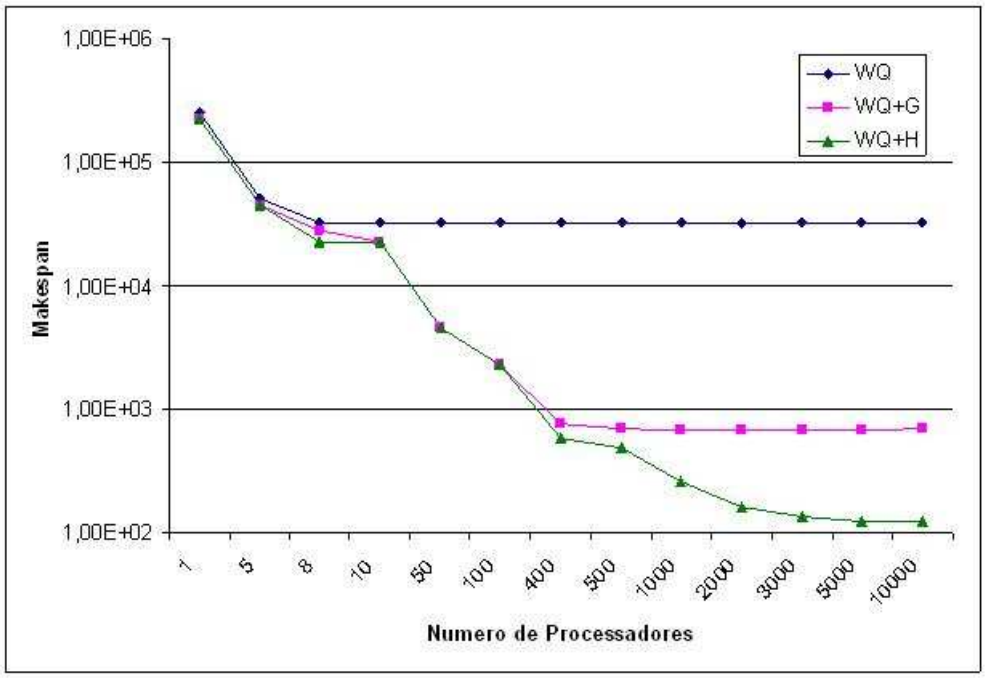
6. Se o mestre não tiver mais tarefas a serem executadas, então o procedimento termina.

Cada supervisor mantém uma lista dos *jobs* e tarefas sobre seu controle, e o mestre mantém uma lista para todos os *jobs* e tarefas da aplicação. Para cada tarefa, existe uma informação de estado (pronto, executando, terminado), o qual é atualizado por mensagens trocadas entre os processadores, em seus eventos correspondentes.

O modelo hierárquico foi avaliado por meio de simulação utilizando o simulador SimGrid, a aplicação e arquitetura utilizadas foram as mesmas descritas na seção 3.4, e ilustradas na figura 21. Os resultados obtidos são comentados abaixo:

Para a execução dessa simulação, foram considerados computadores e *links* de comunicação dedicados. Também a aplicação simulada é composta por tarefas homogêneas. O tempo total de *makespan*, é calculado pelo tempo entre a transmissão do primeiro arquivo de entrada, e o recebimento do último arquivo de saída.

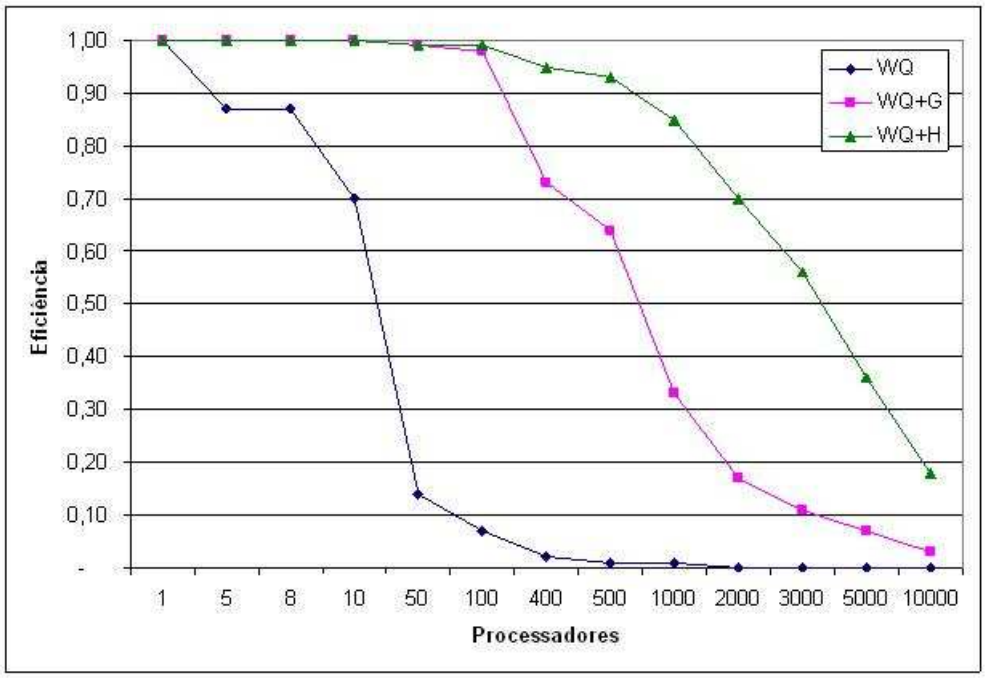
Figura 28: Makespan para simulação de escalonamento de experimentos usando Workqueue(WQ), Workqueue com Agrupamento(WQ+G), e Workqueue com Hierarquia e Agrupamento(WQ+H)



Fonte: (SENGER; SILVA; NASCIMENTO, 2006)

Conforme ilustrado na figura 28, com o algoritmo *Workqueue* puro, o *makespan* foi reduzido até um mínimo de 32.007 unidades de tempo, utilizando oito processadores. Após esse limiar, o *makespan* não é reduzido mesmo adicionando mais processadores. Com o algoritmo *Workqueue* com agrupamento (WQ+G), o *makespan* foi reduzido até um mínimo de 673 unidades de tempo, empregando 1.000 nós. Neste experimento a execução com 500 nós é concluída em 702 unidades de tempo e sua eficiência computacional por volta de 0,64, conforme mostra a figura 29. Contudo, após esse limiar, adicionar mais processadores não conduzirá a uma redução significativa no tempo de execução, e a eficiência cai muito rápido. Finalmente, com a combinação do algoritmo *Workqueue* mais o modelo hierárquico (WQ+H), o tempo total de *makespan* pode ser reduzido até um mínimo de 123 unidades de tempo, com o qual pode ser alcançado utilizando-se 5.000 processadores. Neste experimento, a execução com 3.000 nós é concluída em 132 unidades de tempo e sua eficiência computacional é 0,56. Contudo, o valor notado após esse limiar

Figura 29: Eficiência para simulação de escalonamento de experimentos usando Workqueue(WQ), Workqueue com Agrupamento(WQ+G), e Workqueue com Hierarquia e Agrupamento(WQ+H)



Fonte: (SENGER; SILVA; NASCIMENTO, 2006)

adicionando novos processadores não possibilita uma redução significante no tempo total de *makespan* e a eficiência vai diminuindo gradativamente.

3.6 Conclusão

Conforme discutido neste capítulo, o escalonamento de tarefas do tipo BoT em grades computacionais é um assunto relevante e de grande importância em várias aplicações, é possível citar alguns exemplos como simulações que utilizam o método de Monte Carlo, buscas massívas para quebramento de chaves criptográficas, física de altas energias e mineração de dados. Um escalonamento deve visar a execução das tarefas da melhor maneira possível, ou seja, executar o maior número de tarefas em um menor espaço de tempo possível.

Uma arquitetura bastante utilizada para a execução de tarefas do tipo BoT é a arquitetura mestre-escravo. O escalonamento de tarefas BoT tem sido temo de di-

versos trabalhos, sendo os mais relevantes apresentados em (MAHESWARAN et al., 1999), (CASANOVA et al., 2000) e (GIERSCH; ROBERT; VIVIEN, 2006). Contudo, para a execução dessas heurísticas é necessário conhecer previamente a duração das tarefas que serão executadas. Esse conhecimento dificilmente pode ser obtido com precisão para que seja usado de forma eficiente.

Outra alternativa para o escalonamento dessas tarefas seria eliminar a necessidade do conhecimento prévio da duração das tarefas. Existem vários trabalhos que estudam essa alternativa, podemos citar como exemplo o *MyGrid* em (CIRNE et al., 2003) e *GridBus* em (VENUGOPAL; BUYYA; WINTON, 2004). A presente dissertação tem como objetivo dar continuidade aos trabalhos (SENGER; SILVA; NASCIMENTO, 2006), (SILVA; CARVALHO; HRUSCHKA, 2004) e (SILVA; SENGER, 2008). Esses estudos dão um enfoque em melhorar a utilização da arquitetura mestre-escravo. Essa melhor utilização começa com estudos para a escalabilidade na arquitetura mestre-escravo, e é introduzido o conceito de agrupamento de tarefas a fim de evitar a retransmissão de arquivos do mestre para os escravos. Outro conceito introduzido foi a mudança na arquitetura, que passou de mestre-escravo para mestre-supervisor-escravo, criando um novo modelo, denominado modelo hierárquico. Segundo estudos feitos por meio de simulação, esse modelo em conjunto com o agrupamento de tarefas, proporcionam uma melhor escalabilidade.

4 Proposta de um Escalonador Hierárquico Com Agrupamento

O objetivo principal deste trabalho é propor e implementar um escalonador para dar suporte a execução de aplicações baseadas em tarefas independentes que compartilham arquivos. Esse escalonador deve ser capaz de receber as especificações da aplicação fornecidas pelo usuário. As especificações devem conter informações sobre as tarefas a serem executadas, arquivos de entrada com sua localização, uma lista contendo as relações de dependências entre tarefas e arquivos e os parâmetros para a execução de cada tarefa.

A principal motivação para a implementação dessa ferramenta é a possibilidade de utilização de uma grade computacional para a execução de tarefas do tipo BoT, de tal maneira que os detalhes de implementações fiquem transparentes aos usuários, tornando possível que os usuário não necessitem de um conhecimento profundo sobre grades computacionais.

4.1 Requisitos do Sistema

Nesta seção serão discutidos alguns requisitos importantes para o desenvolvimento de um escalonador.

1. Arquitetura escalável. Para a execução de tarefas do tipo BoT em um ambiente de grande computacional, é desejável que o escalonador de tarefas possua uma arquitetura escalável. Conforme discutido na seção 3.4, a arquitetura hierárquica possui a capacidade de aliviar o gargalo que se forma no computador mestre, portanto, permite a utilização de um número maior de máquinas que possam executar as tarefas

de uma determinada aplicação e conseqüentemente a obtenção de um resultado em um menor tempo.

2. Redução da Comunicação. A retransmissão desnecessária de arquivos aumenta o tempo total de execução da aplicação, e por isso deve ser evitada ao máximo. Conforme discutido na seção 3.4.1, o agrupamento de tarefas possibilita que diversas tarefas que possuam arquivos de entrada em comum possam ser executados em uma mesma máquina. Assim, é possível diminuir o gargalo de comunicação do nó mestre, pois um mesmo arquivo não precisará ser enviado para diversas máquinas. O agrupamento deverá ser realizado com base nas informações das tarefas e seus arquivos de entrada.

3. Configuração. O escalonador deve ser capaz de ler as especificações da aplicação, e então processá-las. Um padrão emergente para especificações é o JSDL. O padrão JSDL proposto é mantido pela OGF, conforme mencionado na seção 2.9.2. A partir da leitura do documento JSDL, o escalonador pode assimilar as relações de dependências entre tarefas e arquivos e tirar proveito disso.

4. Autenticação do Usuário. A autenticação do usuário deverá ser feita no nó mestre, através da criação de um *proxy*, validado por uma autoridade certificadora capaz de confirmar a identidade do usuário.

5. Autorização do Usuário. A autorização ocorre a todo o momento em que existir a necessidade de utilização de qualquer recurso disponível na grade computacional. O escalonador deve permitir que as políticas de autorização existentes sejam consideradas de forma transparente.

6. Delegação. Uma vez que o escalonador coordenará a execução de tarefas em diversos nós, transferências de arquivos bem como diversas ações na grade, será preciso "*delegar*"credenciais aos supervisores e escravos.

7. Descoberta Dinamica de Recursos. Os recursos necessários para a execução da aplicação são obtidos dinamicamente. O sistema proposto deverá descobrir os recur-

sos dinamicamente. Esse mecanismo fará consultas ao serviço de informações para descobrir os recursos existentes nos diversos domínios administrativos que compõem a organização virtual.

8. Controle e Gerenciamento da Execução das Tarefas. As tarefas deverão ser submetidas aos recursos solicitados, caso os mesmos estejam disponíveis. Esse gerenciamento deverá ocorrer durante todo o tempo que durar a execução, tratando do provisionamento dinâmico dos recursos computacionais. Esse provisionamento poderá ser obtido através do armazenamento de informações em um repositório de dados.

9. Gerenciamento de Falhas. O gerenciamento de falhas é responsável por detectar e tratar ocorrências de falhas. As tarefas cuja execução for interrompida por motivo de falha deverão ser reiniciadas.

4.2 Arquitetura

A arquitetura utilizada para esse escalonador é do tipo mestre/supervisor/escravo. Conforme descrito nas seções 3.1 e 3.4.3, essa arquitetura possibilita uma maior escalabilidade na execução das tarefas, e é composta por três níveis:

- **Mestre**. O nó mestre é composto pelas entidades agente usuário, gerenciador de aplicação, serviço gerenciador do estado da tarefa, que desse ponto em diante será chamado simplesmente de Serviço GET e o serviço MDS. O gerenciador de aplicação é responsável pela organização dos recursos descobertos, interpretação dos documentos JSDL da aplicação e criação do plano de execução para a aplicação, bem como seu armazenamento no repositório de dados. Para a descoberta desses recursos o gerenciador invoca um serviço MDS do GT4. Esse serviço é capaz de descobrir dinamicamente os recursos computacionais distribuídos geograficamente e disponibilizados na organização virtual. O plano de execução da aplicação será gerado com base nos recursos disponíveis e nas tarefas que devem ser executadas. O mestre

disponibiliza os arquivos necessários para a execução das tarefas, e o gerenciador solicita ao serviço RFT a transferência deles para os supervisores.

- **Supervisor**. O nó supervisor é composto pelas entidades serviço supervisor, serviço MDS e serviço RFT. A qualificação de um nó como supervisor, consiste em o nó possuir o registro do serviço supervisor no MDS da própria máquina. Para o descobrimento dos recursos disponíveis na organização virtual, o serviço supervisor utiliza o serviço de informações MDS do GT4, que retorna as informações sobre os recursos em um arquivo do tipo XML. Esse arquivo pode ser consultado através da linguagem *XPath* (XML..., 1999).

  Outra funcionalidade do supervisor é a utilização do serviço RFT para a transferência dos arquivos de entrada para as máquinas escravas. Após o término da execução das tarefas, esse nível deverá solicitar o envio dos arquivos de resultado para o nível mestre.

- **Escravo**. O escravo é responsável pela execução das tarefas que foram escalonadas para ele. Para que essa execução ocorra é utilizado o serviço de execução GRAM. Após o término da execução da tarefa, o escravo é responsável pela solicitação ao serviço RFT da transferência dos arquivos de resultados para o nó supervisor.

Conforme comentado anteriormente, o nó mestre é responsável pela interpretação dos documentos JSDL da aplicação. A utilização de uma padronização nesse arquivo auxilia o usuário no momento de sua criação. A seção 4.4 tratará melhor a questão do tipo de documento utilizado para a submissão das tarefas.

A seguir, alguns casos de uso serão apresentados para ilustrar os principais aspectos do funcionamento do escalonador.

4.3 Casos de Uso

Na execução de uma aplicação é possível ocorrer diversas situações. Nesta seção alguns casos de uso serão detalhados.

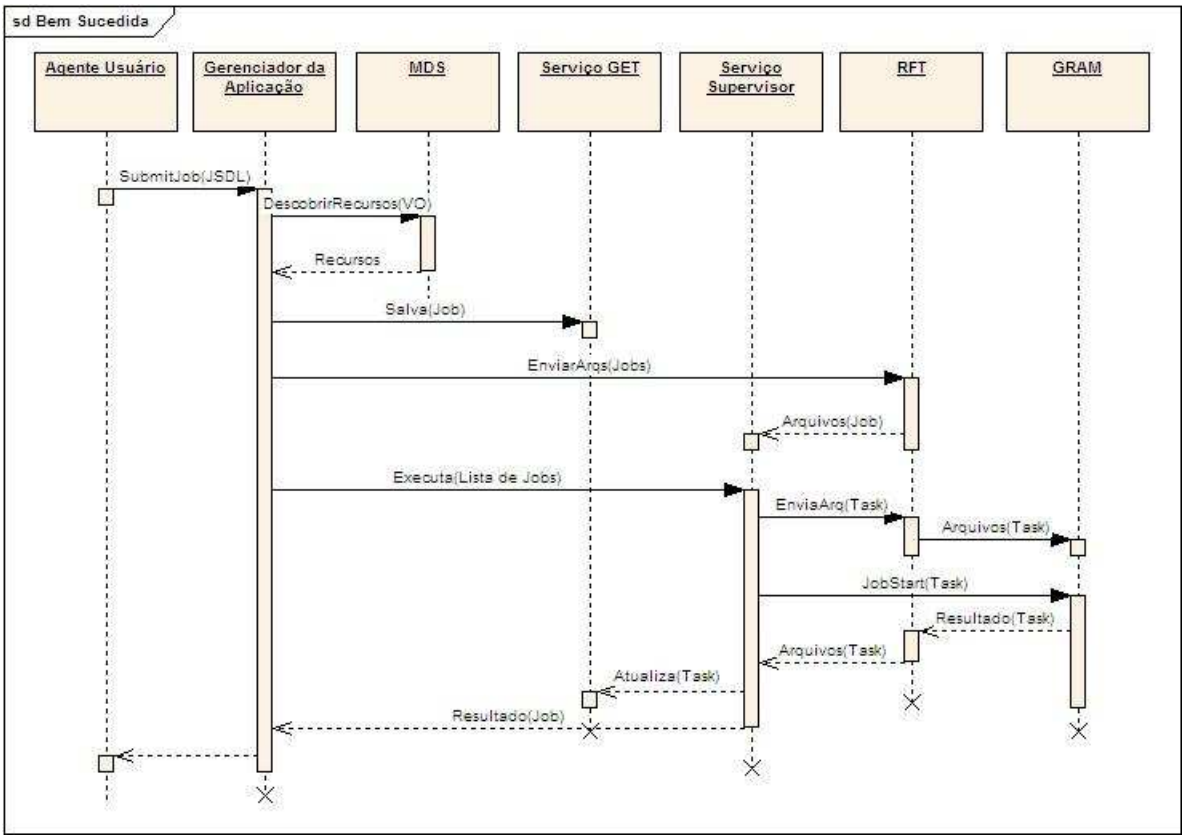
4.3.1 Execução Bem Sucedida

- **Pré-condições:** A grade computacional já deve estar configurada e o usuário autenticado pela grade.
- **Entidades:** As entidades para esse caso de uso são Agente Usuário, Gerenciador da Aplicação, Serviço GET, Serviço Supervisor e GRAM
- **Descrição do cenário:** Este caso ilustra o funcionamento do escalonador em caso de ausência de falhas. O cenário para este caso é a execução de uma aplicação composta por T tarefas e F arquivos compartilhados entre as tarefas. A quantidade total de máquinas disponíveis na grade computacional é P máquinas.
- **Execução da aplicação:** A execução dessa aplicação é ilustrada pela figura 30 e descrita a seguir:

  1. O agente usuário envia para o gerenciador da aplicação um documento JSDL contendo a descrição das tarefas.
  2. O gerenciador da aplicação interpreta o documento JSDL, e solicita ao serviço MDS a descoberta dos recursos disponíveis.
  3. Um agrupamento é feito com as tarefas, e logo após é gerado uma lista com um pré-escalonamento, que é armazenado no em um meio persistente através do serviço GET.
  4. Essa lista é enviada para o serviço supervisor juntamente com os arquivos de entrada necessários para sua execução utilizando o serviço RFT.
  5. O serviço supervisor efetua a transferência dos arquivos de entrada através do serviço RFT para os diversos componentes GRAM localizado nos escravos.
  6. O serviço supervisor invoca o serviço GRAM passando para ele as tarefas que devem ser executadas.
  7. As tarefas são executadas nas máquinas escravas de acordo com os parâmetros de execução de cada uma.

8. Os resultados das tarefas são devolvidos para o supervisor através do serviço RFT.

Figura 30: Execução bem sucedida



Fonte: Próprio Autor

4.3.2 Execução com Falha no Escravo

- **Pré-condições:** A grade computacional já deve estar configurada e o usuário autenticado pela grade. A aplicação já foi escalonada e algumas tarefas encontram-se em execução.

- **Entidades:** As entidades para esse caso de uso são Agente Usuário, Gerenciador da Aplicação, Serviço GET, Serviço Supervisor e GRAM

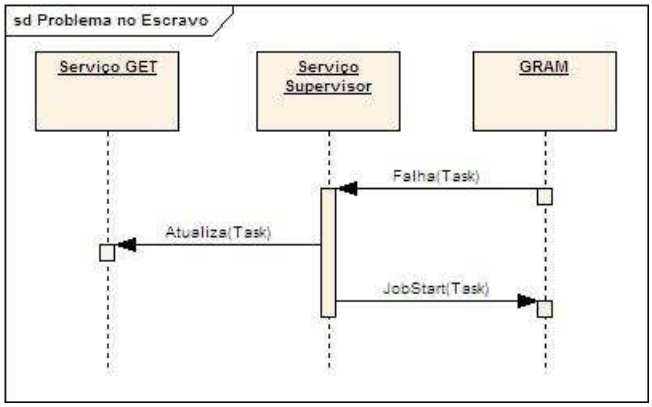
- **Descrição do cenário:** Este caso é uma variante do caso apresentado na seção 4.3.1, e ilustra somente o gerenciamento de uma falha no nó escravo. Ao ser detectado essa falha o supervisor deverá escalonar a tarefa em que a falha ocorreu

para outro escravo, sendo que todo o processamento já executado dessa tarefa será descartado.

- **Execução da aplicação:** A execução dessa aplicação é ilustrada pela figura 31 e descrita a seguir:

  1. O serviço supervisor é notificado através de um evento gerado pelo serviço GRAM que uma falha ocorreu na execução de uma determinada tarefa.
  2. O serviço supervisor então solicita a atualização do repositório de dados através do serviço GET, e faz um re-escalonamento dessa tarefa para outra máquina, de preferência uma máquina que já possua os arquivos de entrada, afim de evitar sua retransmissão.
  3. O serviço supervisor invoca o serviço GRAM de outro escravo, passando para ele a tarefa em que ocorreu a falha para que ela possa ser executada.

Figura 31: Execução com Problemas no Escravo



Fonte: Próprio Autor

4.3.3 Execução com Falha no Supervisor

- **Pré-condições:** A grade computacional já deve estar configurada e o usuário autenticado pela grade. A aplicação já foi escalonada e algumas tarefas encontram-se em execução.

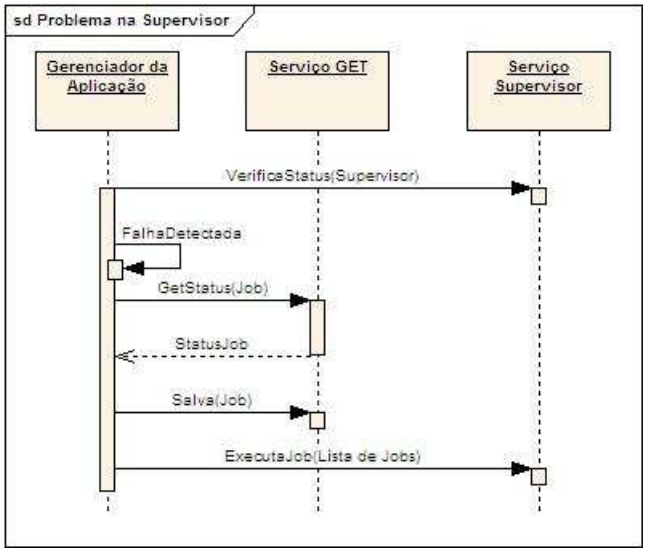
- **Entidades:** As entidades para esse caso de uso são Agente Usuário, Gerenciador da Aplicação, Serviço GET, Serviço Supervisor e GRAM.

- **Descrição do cenário:** Este caso é uma variante do caso apresentado na seção 4.3.1, e ilustra funcionamento do escalonador com falha no serviço supervisor. Ao ser detectado essa falha, o gerenciador da aplicação deverá buscar no repositório de dados quais foram as tarefas que pertenciam àquele supervisor, e então será feito então uma nova submissão somente das tarefas que não foram completadas para outro supervisor.

- **Execução da aplicação:** A execução dessa aplicação é ilustrada pela figura 32 e descrita a seguir:

    1. O gerenciador da aplicação detecta a falha ocorrida no serviço supervisor, através de um monitoramento que ele faz em cada supervisor.
    2. O gerenciador da aplicação então solicita ao serviço GET as informações sobre os estados de todas as tarefas supervisionadas pelo serviço supervisor que ocorreu a falha.
    3. É escolhido um novo supervisor, tendo como critério para a escolha do novo supervisor aquele que possuir o menor número de tarefas incompletas.
    4. O repositório de dados é atualizado através do serviço GET, sendo informado para ele o novo supervisor em que as tarefas incompletas (do supervisor que ocorreu falha) devem ser enviadas. Formando uma lista de tarefas a serem executadas nesse novo supervisor.
    5. Essa lista é enviada para o serviço supervisor escolhido, para que seja providenciado sua execu cão. Caso seja necessário os arquivos de entrada dessas tarefas também serão enviados.

Figura 32: Execução com Problemas no Supervisor



Fonte: Próprio Autor

4.4 Linguagem para Submissão das Tarefas

É possível verificar algumas formas de descrição de tarefas tais como APST, JSDL, RSL e JDF, conforme visto nas seções 2.9 e 2.10 respectivamente. Dentre essas formas de descrição de tarefas apresentadas, destaca-se a linguagem JSDL. Por ser uma recomendação da OGF, essa linguagem é uma tendência. Com a possibilidade de o JSDL vir a se tornar um padrão, torna-se interessante sua utilização frente à ampla utilização em diversas plataformas. As outras formas de descrição de tarefas apresentadas na seção 2.9 não serão utilizadas neste trabalho.

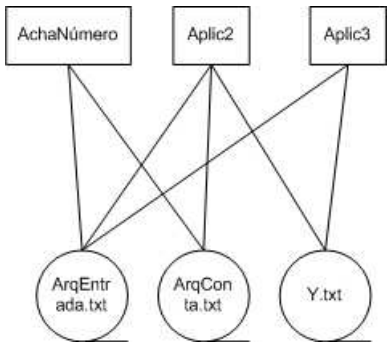
Para a ferramenta proposta neste trabalho, será utilizado um subconjunto das especificações JSDL 1.0.

4.4.1 Adaptando o JSDL 1.0

A utilização de um subconjunto das especificações JSDL ocorre porque nem todos os itens disponíveis nessa especificação foram utilizados, visto que tamanho físico da memória, quantidade de processadores em uma máquina, tamanho máximo de arquivos, total de espaço em disco livre, entre outras, não são informações necessárias para a ex-

ecução da ferramenta proposta neste trabalho. A especificação completa do JSDL pode ser encontrada em (ANJOMSHOAA et al., 2005). Um subconjunto é ilustrado na figura 34, que utiliza como exemplo a aplicação descrita pelo grafo na figura 33. O subconjunto utilizado é descrito da seguinte maneira:

Figura 33: Grafo de Aplicação tipo TICA



Fonte: Próprio Autor

- <jsdl:JobName>. É utilizado para identificar a tarefa que está sendo descrita.
- <jsdl:Description>. Especifica uma descrição sobre a tarefa.
- <jsdl:ApplicationName>. Especifica o nome da aplicação.
- <jsdl-posix:POSIXApplication>
  - <jsdl-posix:WorkingDirectory>. Diretório de trabalho na máquina executora da tarefa. Caso não seja informado nenhum parâmetro é utilizado o diretório *home* do usuário.
  - <jsdl-posix:Executable>. Especifica o nome do programa que deverá ser executado.
  - <jsdl-posix:Argument>. Especifica os argumentos a serem utilizados como parâmetros para o programa executável.
  - <jsdl-posix:Input>. Especifica os arquivos de entrada para a tarefa.
  - <jsdl-posix:Output>. Especifica os arquivos que serão gerados pela tarefa como resultados.

Figura 34: Exemplo de Descrição de uma Tarefa

```
<?xml version="1.0" encoding="UTF-8"?>
<jsdl:JobDefinition xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
  xmlns:jsdl-posix="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix">
  <jsdl:JobDescription>
    <jsdl:JobIdentification>
      <jsdl:JobName>Procurar Número</jsdl:JobName>
      <jsdl:Description>Este programa procura um número dentro de um arquivo</jsdl:
    </jsdl:JobIdentification>
    <jsdl:Application>
      <jsdl:ApplicationName>AchaNumero</jsdl:ApplicationName>
      <jsdl-posix:POSIXApplication>
        <jsdl-posix:WorkingDirectory>/tmp</jsdl-posix:WorkingDirectory>
          <jsdl-posix:Executable>/home/flavio/apc1001
          </jsdl-posix:Executable>
          <jsdl-posix:Argument>arqEntrada</jsdl-posix:Argument>
          <jsdl-posix:Argument>9087634</jsdl-posix:Argument>
          <jsdl-posix:Input>arqEntrada.txt</jsdl-posix:Input>
          <jsdl-posix:Output>stdout.job01</jsdl-posix:Output>
      </jsdl-posix:POSIXApplication>
    </jsdl:Application>
    <jsdl:DataStaging>
        <jsdl:FileName>arqEntrada.txt</jsdl:FileName>
        <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
        <jsdl:DeleteOnTermination>false</jsdl:DeleteOnTermination>
        <jsdl:Source>
            <jsdl:URI>/home/flavio</jsdl:URI>
        </jsdl:Source>
    </jsdl:DataStaging>
    <jsdl:DataStaging>
        <jsdl:FileName>arqConta.txt</jsdl:FileName>
        <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
        <jsdl:DeleteOnTermination>false</jsdl:DeleteOnTermination>
        <jsdl:Source>
            <jsdl:URI>/home/flavio</jsdl:URI>
        </jsdl:Source>
    </jsdl:DataStaging>
    <jsdl:DataStaging>
        <jsdl:FileName>stdout.job01</jsdl:FileName>
        <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
        <jsdl:DeleteOnTermination>true</jsdl:DeleteOnTermination>
        <jsdl:Target>
            <jsdl:URI>/home/flavio/resultado/job01</jsdl:URI>
        </jsdl:Target>
   </jsdl:DataStaging>
  </jsdl:JobDescription>
</jsdl:JobDefinition>
```

Exemplo.jsdl

  - <jsdl-posix:Error>. Especifica o arquivo de erro, caso ocorra algum.

- <jsdl:DataStaging>

  - <jsdl:FileName>. Especifica o caminho e o nome do arquivo que deverá ser transferido de/para máquina executora.
  - <jsdl:CreationFlag>. Especifica se o arquivo poderá ou não ser sobrescrito.
  - <jsdl:DeleteOnTermination>. Informa que o arquivo de entrada ou saída, após a execução da tarefa deverá ou não ser excluído.
  - <jsdl:Source>. Indica o diretório da máquina remota onde o arquivo deverá ser colocado no momento do *Stage-in*.
  - <jsdl:Target>. Indica o diretório da máquina servidora onde o arquivo deverá ser colocado no momento do *Stage-out*.

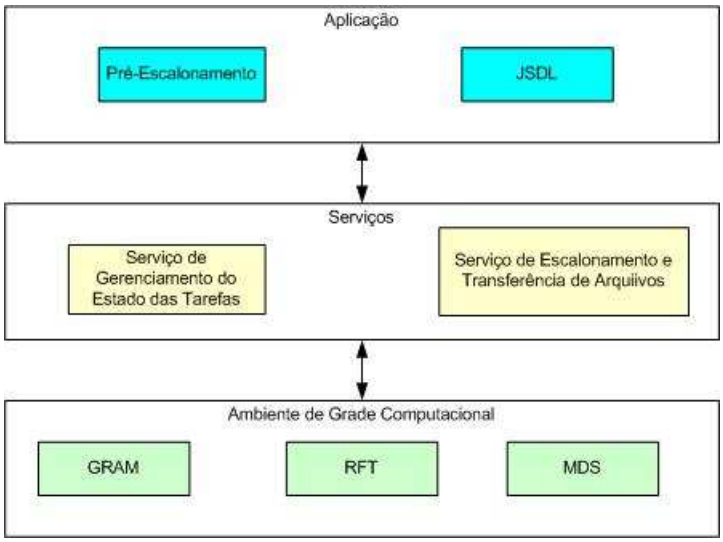
4.5 Gerenciamento da Execução

Além dos níveis que formam a hierarquia do escalonador, são utilizados serviços da infra-estrutura do Globus para prover o descobrimento de recursos, transferência de dados e gerenciamento de execução de tarefas. A figura 35 mostra o diagrama desse sistema.

Os componentes utilizados para o gerenciamento da aplicação são disponibilizados na forma de serviços web. Esses serviços são distribuídos entre os diversos níveis. Cada serviço é responsável por uma determinada funcionalidade do sistema. Os serviços GET e Supervisor são detalhados a seguir:

- *Serviço GET*. Serviço que disponibiliza o acesso à um repositório de dados. Esse serviço será utilizado para disponibilizar um local onde todas as informações para controle da aplicação tais como o estado de cada tarefa, qual supervisor é responsável por uma determinada tarefa, qual escravo está executando ou executou uma determinada tarefa, e quais arquivos são utilizados por quais tarefas. Esse serviço

Figura 35: Diagrama do Sistema



Fonte: Próprio Autor

encontra-se no nó mestre, e poderá ser acessado somente por ele e pelo supervisor. A utilização de um serviço que guarda informações em um meio persistente é extremamente útil, pois caso o nó mestre falhe é possível identificar quais tarefas já foram completadas, e reiniciar a aplicação somente para as outras tarefas.

- *Serviço Supervisor*. Esse serviço recebe informações sobre os *Jobs*, que são compostos por tarefas já agrupadas e que foram escalonadas para o aglomerado que está sob o seu controle, efetua as transferências necessárias dos arquivos de entrada ou de retorno, descobre recursos disponíveis na organização virtual, dispara todas as tarefas para as máquinas escravas para sua execução, e elimina arquivos desnecessários ao final da execução. Esse serviço encontra-se no nível supervisor e deve ser instalado previamente em uma ou mais máquinas locais, com a finalidade da construção de um modelo hierárquico.

4.5.1 Especificação do Sistema

Por simplicidade, decidiu-se utilizar uma notação baseada em componentes para especificar tais serviços. A seguir será detalhado cada nível existente na hierarquia.

4.5.2 Mestre

O nó mestre foi desenvolvido para fazer a interface com o Agente Usuário através de um arquivo de descrição da aplicação, utilizando a linguagem JSDL 1.0. Após o recebimento desse arquivo, o mestre interpreta a descrição da aplicação, e invoca o serviço de informações para descobrir os recursos disponíveis nos aglomerados, que sejam supervisores. Com o conhecimento dos supervisores disponíveis na organização virtual, o mestre solicita para esses supervisores que descubram os recursos disponíveis no aglomerado a que ele pertence.

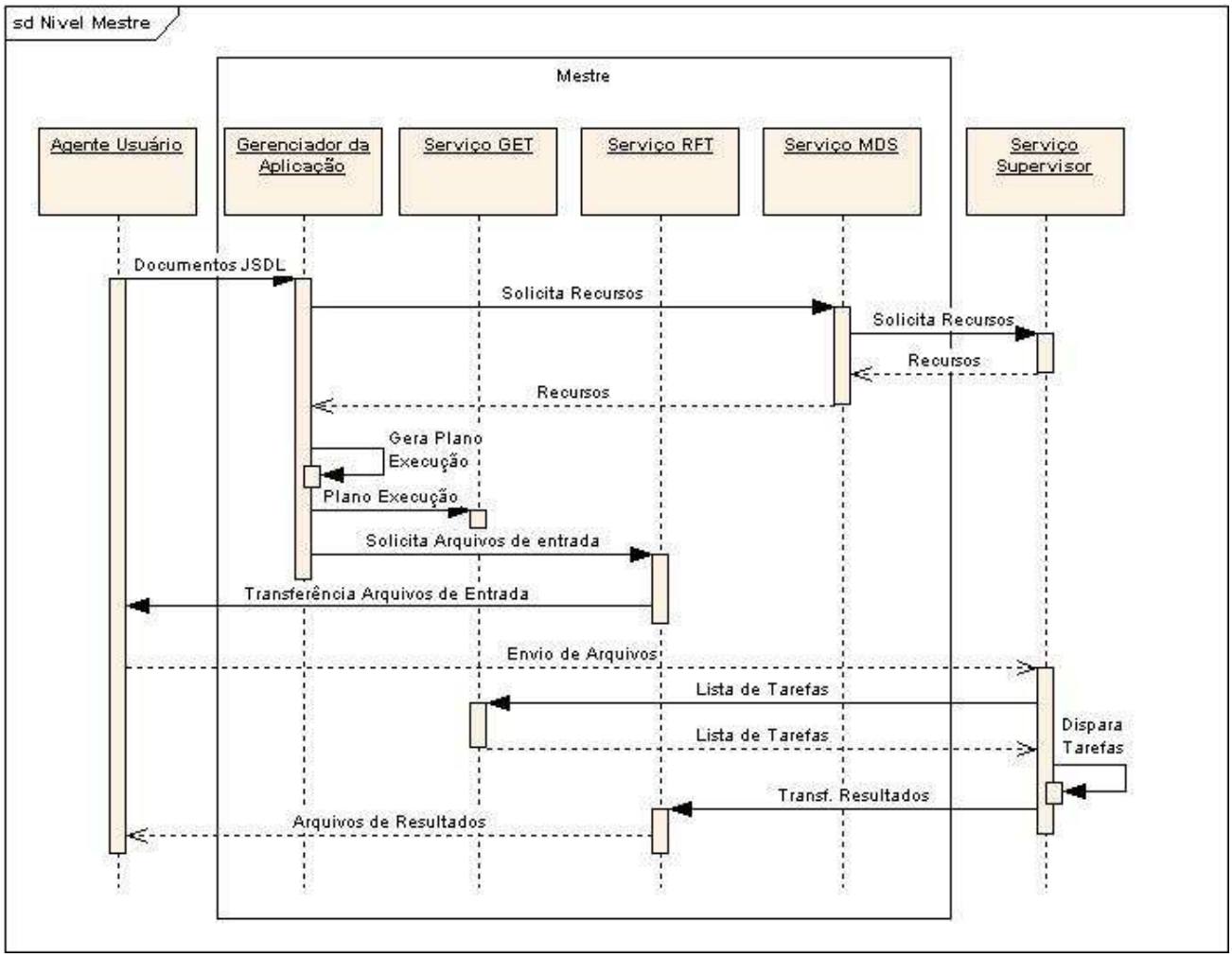
Com a informação dos recursos disponíveis e a descrição da aplicação, o mestre inicia o processo de escalonamento das tarefas, gerando um plano de execução. Esse escalonamento agrupa tarefas que utilizam os mesmos arquivos de entrada. Esse agrupamento utiliza a métrica IFA, que indica o grau de compartilhamento de arquivos entre tarefas, descrita na seção 3.4.1. O propósito é melhorar o escalonamento, tentando evitar a retransmissão desnecessária de arquivos. O plano de execução será armazenado em um meio persistente através do serviço GET. Após o armazenamento, o mestre invoca o serviço na camada supervisor para que as tarefas que foram escalonadas sejam executadas.

Após a execução de todas as tarefas, o mestre dispara uma solicitação para a camada supervisor informando quais arquivos devem ser excluídos dos escravos e dos supervisores, eliminando arquivos de entrada, de trabalho e executáveis que não mais serão necessários. A figura 36 representa o diagrama de seqüencia do nó mestre.

4.5.3 Supervisor

O supervisor serve como um intermediário entre o mestre e as máquinas escravas. Sua finalidade é possibilitar um melhor escalonamento na execução de tarefas, diminuindo o gargalo que se formaria no nó mestre com relação ao controle das tarefas e transmissão dos arquivos.

Figura 36: Diagrama de seqüência do Nível Mestre



Fonte: Próprio Autor

Esse nó é composto pelo serviço denominado *SupervisorService*, MDS e RFT. Sua funcionalidade consiste em efetuar o descobrimento de recursos disponíveis na grade através do serviço de informações MDS, enviar informações sobre esses recursos descobertos para o mestre, acessar o plano de execução criado pelo mestre e armazenado no repositório de dados, transferir para os escravos todos os arquivos necessários para a execução de cada tarefa utilizando o serviço de transferência de arquivos RFT, e gerenciar a execução do plano de execução nos nós escravos subordinados a esse supervisor

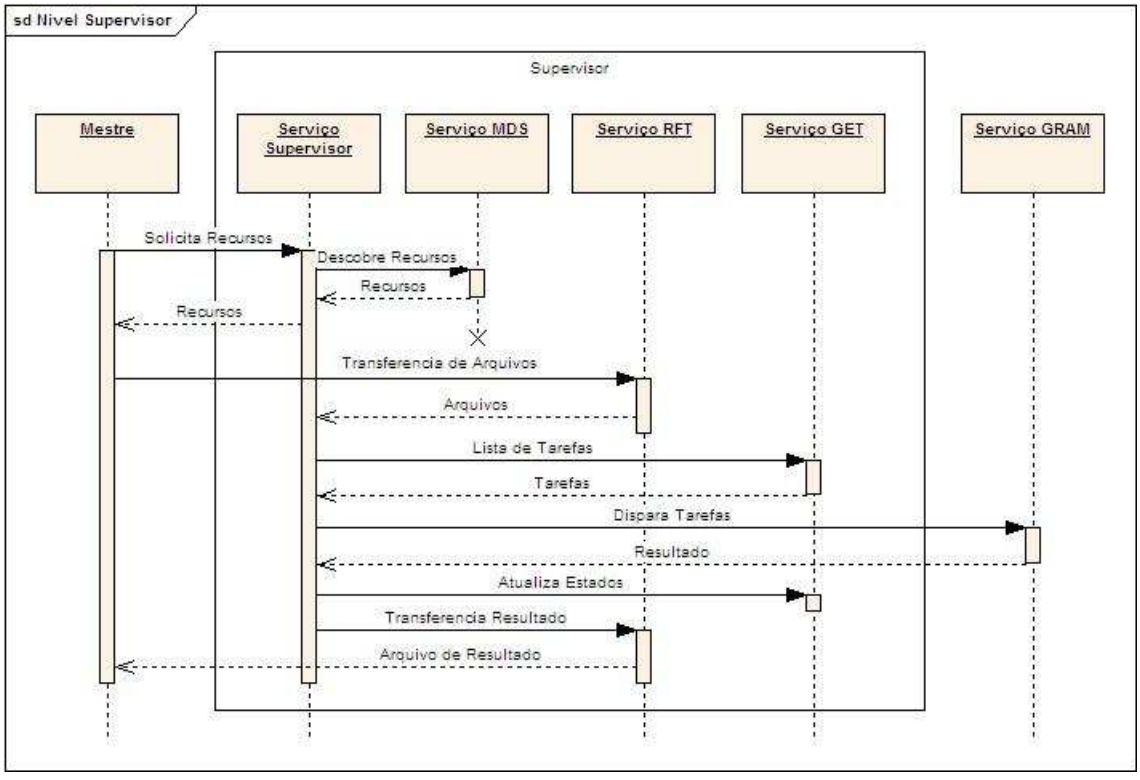
Durante a execução das tarefas, o supervisor permanece monitorando as informações sobre o estado de cada tarefa. É possível que um determinado escravo execute

todas suas tarefas e torne-se ocioso. Caso isso ocorra, significa que as tarefas delegadas para esse supervisor já foram distribuídas entre seus escravos, não restando nenhuma pendente. Portanto, nesta situação o mestre vai redistribuir as tarefas pendentes de outros supervisores para este. Essa redistribuição consiste em transferir uma tarefa que ainda não foi iniciada, de um nó supervisor para o nó supervisor que possui escravos ociosos. Essa transferência de tarefas pode proporcionar um término antecipado da aplicação.

Após o término da execução de cada tarefa, o supervisor solicita ao serviço GET uma atualização no repositório de dados e transfere os arquivos de resultado gerado no escravo que terminou a tarefa para o repositório de arquivos localizados na nível mestre.

Ao final da execução de todas as tarefas que compõem a aplicação, o supervisor recebe ainda uma solicitação do mestre, para que todos os arquivos utilizados na execução das tarefas sejam excluídos dos escravos e do repositório de arquivos local. A figura 37 representa o diagrama de seqüência da camada supervisor.

Figura 37: Diagrama de seqüência do Nível Supervisor



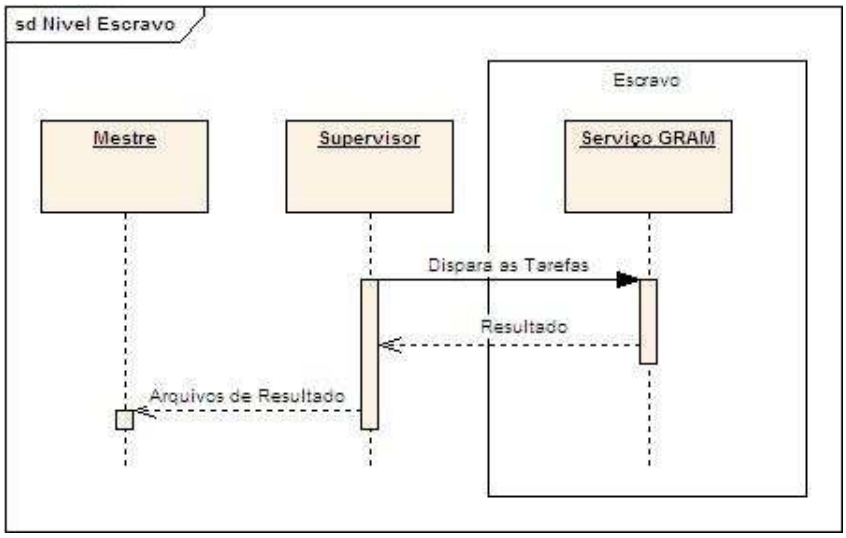
Fonte: Próprio Autor

4.5.4 Escravo

O nó escravo, é responsável pela execução da tarefa. Utilizando os serviços disponíveis na infra-estrutura de execução do Globus, o escravo faz uso do GRAM. A tarefa é submetida para o GRAM que a executa, e através de notificações informa o estado de cada uma.

As máquinas escravas não possuem nenhum serviço criado por este trabalho. Portando, para que a tarefa seja executada os arquivos de entrada já devem ter sido transmitidos para o repositório local, e após a execução da tarefa é informado para o supervisor que a tarefa foi concluída. A figura 38 representa o diagrama de seqüência das atividades dos nós escravos.

Figura 38: Diagrama de Seqüência das Atividades dos Nós Escravos



Fonte: Próprio Autor

4.5.5 Descrição dos Serviços

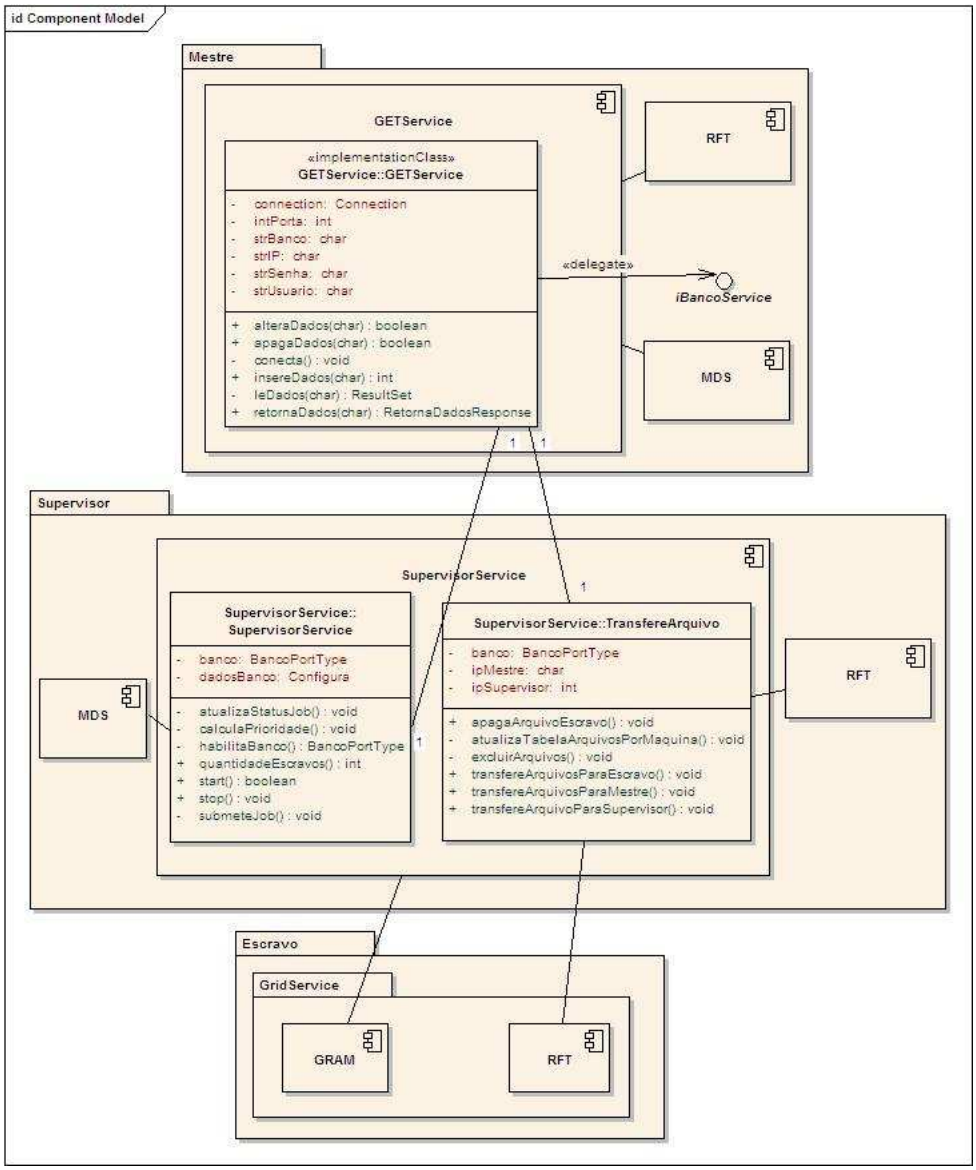
Os serviços GET e SupervisorService podem ser descritos de acordo com o diagrama em UML representado na figura 39.

4.6 Conclusão

A ferramenta proposta neste capítulo implementa uma arquitetura hierárquica do tipo mestre/supervisor/escravo. O nível supervisor é um componente do tipo *Web Service*. Esse tipo de componente proporciona uma maior interação com diversos outros componentes, *Web Services* ou não. O supervisor tem a finalidade de receber uma lista de tarefas do mestre e encaminha-las para a execução nos diversos escravos que se encontram hierarquicamente abaixo do supervisor.

Para controle da aplicação submetida ao escalonador, foi necessário a utilização de um banco de dados, que tem como finalidade disponibilizar informações ao escalonador sobre os estados das tarefas para que um monitoramento e controle da aplicação possa ser feito.

Figura 39: Diagrama UML dos Serviços



Fonte: Próprio Autor

5 Avaliação e Desempenho

5.1 Introdução

Este capítulo contém uma demonstração dos resultados obtidos com a utilização do escalonador proposto. Será descrito na seção 5.3 o ambiente de grade computacional, configurado com o auxilio da ferramenta *Globus Tool Kit 4.0*, para que o escalonador possa ser avaliado.

A aplicação utilizada para testar a ferramenta desenvolvida é a ferramenta de mineração de dados denominada WEKA (WITTEN; FRANK, 2005). Essa ferramenta implementa uma série de algoritmos, tais como de classificação (Id3, C45, J48 e ADTree), geradores de regras de classificação (Prisma, Part e OneR), entre outros (SILVA, 2004)(WITTEN; FRANK, 2005). O *WEKA* foi desenvolvido na Universidade de Waikato na Nova Zelândia, sendo escrito em linguagem Java e possuindo código aberto disponível na Web (atual versão 3.4 - versão Java mínima 1.4).

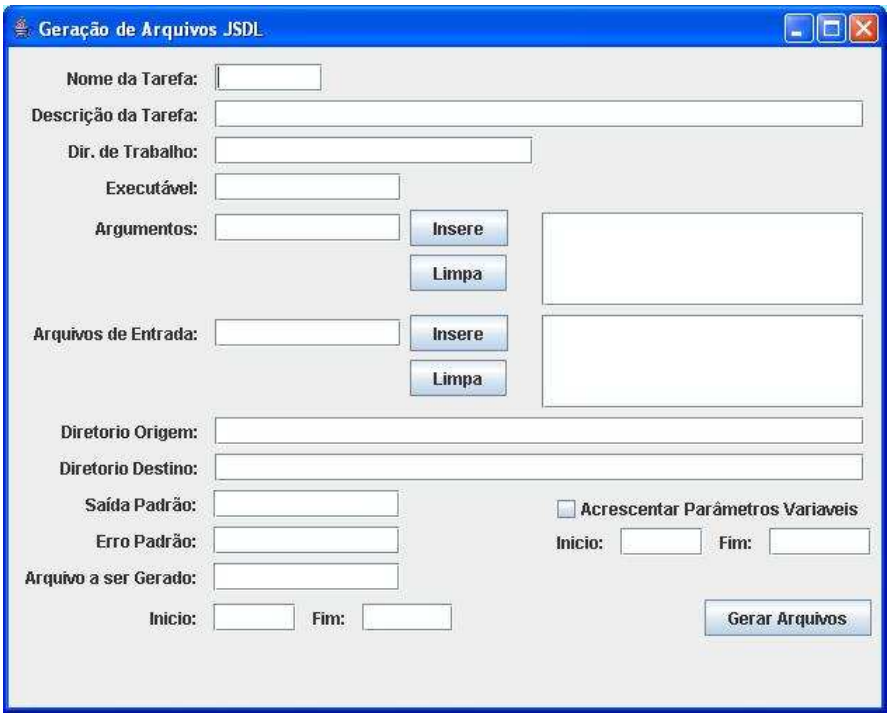
A ferramenta WEKA é capaz de analisar grandes bases de dados, o que manualmente não seria possível. A escolha dessa ferramenta para testes no escalonador é motivada pela possibilidade da criação de tarefas independentes (BoT), que podem ou não compartilhar arquivos.

Com o auxílio do utilitário **Gerador de Tarefas** que será descrito na seção 5.2, é possível criar diversas tarefas que utilizarão um dos algoritmos disponíveis no WEKA, com a finalidade de analisar através de diferentes aspectos uma mesma base de dados já conhecida.

5.2 Gerador de Tarefas

O Gerador de Tarefas ilustrado na figura 40, é um utilitário capaz de gerar uma seqüencia de documentos JSDL segundo as especificações descritas na seção 4.4.1. A utilização dessa ferramenta pode proporcionar uma gama de variações de parâmetros para aplicações que podem ser executadas por escalonadores, nesse caso o escalonador proposto neste trabalho. O conjunto de tarefas geradas pode formar uma aplicação de mineração de dados, renderização de imagens, ou outro tipo qualquer de aplicação que necessite de um grande número de tarefas do tipo BoT e que cada tarefa possa possuir parâmetros diferentes.

Figura 40: Gerador de Tarefas .JSDL



Fonte: Próprio Autor

Para a execução desse utilitário é preciso fornecer alguns parâmetros:

- *Nome da Tarefa*, especifica o nome da tarefa dentro da aplicação.
- *Descrição da Tarefa*, uma breve descrição sobre qual a finalidade dessa tarefa.

- *Diretório de Trabalho*, deve ser especificado em qual diretório, local ou remoto, essa tarefa assumirá como diretório padrão.

- *Executável*, nome de um programa do tipo executável, arquivo de lote (Windows) ou arquivo *shell script* (Linux) a ser executado em cada tarefa.

- *Argumentos*, especifica cada argumento estático necessário para a execução da tarefa. Cada argumento deve ser inserido na lista de argumentos através do botão "INSERE".

- *Arquivos de Entrada*, especifica um ou mais arquivos de entrada necessários para as tarefas. Cada um dos arquivos deverá ser inserido na lista de arquivos através do botão "INSERE".

- *Diretório Origem*, informa o nome do diretório onde o executável e os arquivos de entrada encontram-se na máquina local, para que os mesmos possam ser copiados para a máquina que irá executar essa tarefa.

- *Diretório Destino*, informa o nome do diretório onde o executável e os arquivos de entrada deverão ser copiados na máquina remota.

- *Saída Padrão*, nome do arquivo que será gerado quando a tarefa estiver em execução e ocorrer qualquer tipo de saída de informação no console (vídeo), estas serão redirecionadas para este arquivo.

- *Erro Padrão*, quando uma tarefa é executada pelo GRAM, um mecanismo é executado juntamente com a tarefa, ficando esse mecanismo encarregado de, no caso de uma mensagem de erro ser emitida pelo sistema operacional, essa mensagem será automaticamente redirecionada para esse arquivo de erro padrão, que será devolvido para o usuário.

- *Acrescentar Parâmetros Variáveis*, caso essa tarefa em questão necessite de algum tipo de parâmetro variável, esta opção deverá ser assinalada e, informado um valor numérico inicial e final desse parâmetro. Um parâmetro é incluído na lista de

argumentos de cada tarefa. Esse argumento assume um valor, que varia entre o número inicial e o final que foram especificados logo abaixo da opção selecionada. Com isso o conjunto de tarefas geradas possuirá um valor em sua lista de argumento diferente em cada uma das tarefas.

- *Arquivo a ser Gerado*, nome do documento *.JSDL* que será gerado.

- *Início e Fim*; esses atributos especificam o intervalo numérico em que os documentos *.JSDL* serão gerados, seguindo o formato <nome-do-documento-*xxx*.JSDL>, onde *xxx* é o intervalo especificado nesse parâmetro.

5.3 Ambiente Computacional Utilizado

O ambiente computacional utilizado para este experimento é composto por máquinas homogêneas, com sistema operacional Windows XP Professional. O escalonador proposto nesse trabalho foi desenvolvido em linguagem Java 1.5 para ambiente Linux. Para que esse escalonador seja executado nas máquinas que compõem o ambiente computacional, houve a necessidade da criação de uma virtualização do sistema operacional Linux em máquinas com sistema operacional Windows. Essa virtualização foi obtida com o auxilio do software *VMWare Workstation 6.0* (VMWARE, 2008b) em conjunto com o *VMWare Player 2.0* (VMWARE, 2008a).

O software *VMWare* proporciona uma virtualização de máquinas e sistemas operacionais sobre um sistema operacional distinto. Tendo como base essa característica, foi utilizado o software *VMWare Workstation* para a criação de uma máquina virtual que utilize como sistema operacional virtualizado o Linux. Uma vez criada uma máquina virtual, que é composta por arquivos que são interpretados pelo *VMWare*, é possível que essa máquina virtual seja copiada em quantas máquinas forem necessárias. Após efetuado as cópias, inicia-se o processo de configuração dessas cópias em cada uma das máquinas que as possuem. Essa configuração caracteriza-se por definir o nome da máquina bem

como seu endereço da rede. Para colocar cada cópia configurada/instalada em execução é utilizado o *VMWare Player*.

Para a criação de uma máquina virtual, é preciso especificar a quantidade de memória *RAM* que essa nova máquina virtualizada terá, sempre respeitando o limite que o *VMWare* impõe. Como por exemplo, caso a máquina física possua 256 MB de memória *RAM* a máquina virtual poderá ter no máximo 50% de memória *RAM* disponível. Isso ocorre pelo fato de que existe uma porcentagem da memória que deve ser reservada para que o sistema operacional nativo possa trabalhar, no caso o Windows.

Para este experimento foram utilizadas: uma máquina denominada mestre, até duas máquinas denominadas supervisores e até 17 máquinas denominadas escravas, sendo utilizado uma rede ethernet com *switch* de 10/100 Mbps de velocidade para interligação de todas as máquinas. A tabela 1 ilustra as configurações de cada uma.

Quantidade	**Processador**	**MF**	**MV**	**S.O.**	**Virtual**
16	Pentium IV 3.00Ghz	512Mb	256Mb	Linux	Sim
1	Celeron M 1.46Ghz	512Mb		Linux	Não

Tabela 1: Configurações das máquinas utilizadas
Legenda MF:Memória Física - MV:Memória da Máquina Virtualizada

O conjunto de máquinas utilizadas foi organizado segundo diferentes configurações. Cada configuração será utilizada para a execução e análise de um mesmo conjunto de tarefas. De acordo com as configurações criadas, é possível representam uma arquitetura do tipo mestre/supervisor/escravo ou mestre/escravo. Os 17 nós que compõem conjunto de máquinas foram arranjados de tal forma a compor *clusters* cujo tamanho variou de 3 a 17 nós.

A criação de configurações de computadores com 1 e 2 supervisores mais a variação do número de escravos para cada supervisor, tem como finalidade criar um ambiente onde a escalabilidade da ferramenta possa ser testada e avaliada. Essa escalabilidade é avaliada adicionando mais escravos a cada supervisor a cada nova iteração de testes. A execução do escalonador em configurações com 1 ou 2 supervisores possibilita uma comparação de desempenho e escalabilidade que será discutido na seção 5.5.

Foram geradas 12 configurações distintas afim de proporcionar vários ambientes para testes do escalonador. Essas configurações foram identificadas seguindo o padrão *XX-YY*, onde o *XX* representa a quantidade de escravos, e *YY* representa o número de supervisores dessa configuração.

A seguir é demonstrado como as configurações foram formadas.

- Com um supervisor: 02-01, 03-01, 06-01, 10-01, 14-01 e 17-01.
- Com dois supervisores: 02-02, 04-02, 06-02, 10-02, 14-02 e 17-02.

Quando necessário em algumas configurações tanto o nó mestre quanto os nós supervisores também são utilizados como escravos. Esse tipo de configuração auxilia no desempenho total da aplicação pois, a carga de processamento que o escalonador utiliza do mestre e dos supervisores é pequena, possibilitando que os mesmos possam ajudar na aplicação fazendo papel de escravo também.

Uma vez formada a grade computacional com as configurações acima descritas, é importante destacar que a escolha de uma aplicação adequada para ser executada no escalonador pode ajudar no desempenho do mesmo, a próxima seção detalha explica essa escolha.

5.3.1 Escolha da Carga de Trabalho

Para composição da carga de trabalho das tarefas foram escolhidas aplicações cuja razão entre o tempo de transferência de arquivos (T) por tempo de execução (E) maiores ou iguais a 50%. Essa proporção indica que essas aplicações são do tipo *I/O Bound*, ou seja, é utilizado mais tempo para a transferência de arquivos do que processamento. Com essa proporção é esperado um melhor resultado para a execução do escalonador proposto neste trabalho, visto que, a formação de um gargalo no mestre é mais iminente.

Existem diversos algoritmos para mineração de dados, um desses algoritmos é o algoritmo de classificação *J48* que foi utilizado para compor as tarefas do experimento,

também conhecido na literatura como *C4.5*, que é implementado na ferramenta *WEKA*. O modo padrão de predizer uma taxa de erro do classificador *J48* dado uma amostra simples, fixa de dados, é utilizar a validação cruzada. No processo de validação cruzada o conjunto de dados é dividido em n partes, e o classificador é treinado em $n-1$ partes e testado na parte restante. Este procedimento é repetido em n conjuntos de treinamentos diferentes e a taxa de erro calculada é a média dos conjuntos de testes e, ao final uma árvore de decisão é gerada.

A base de dados que será utilizada nesse experimento e a base de dados *Adult* (ASUNCION; NEWMAN, 2007) que pode ser encontrada no repositório de bases de dados da UCI. O conjunto de dados contido nessa base foi extraído da base de dados do Censo de 1994 *(http://www.census.org)* usando as condições seguintes: $((AGE > 16)and(AGI > 100)and(FNLWGT > 1)and(HRSWK > 0)$, onde AGE indica a idade da pessoa, AGI indica renda bruta ajustada, $AFNLWGT$ indica peso da ocorrência e $HRSWK$ indica horas trabalhadas na semana.

A predição consiste em determinar se uma pessoa ganha mais que 50.000 US$ ao ano, analisando seus outros atributos. Essa base de dados é composta por 14 atributos: idade, classe de funcionalismo, peso da ocorrência, escolaridade, código da escolaridade, estado civil, ocupação, relacionamento, raça, sexo, ganho de capital, perda de capital, horas trabalhadas por semana e cidade natal. Foram utilizados para testes no escalonador 24000 ocorrências dessa base de dados. Um exemplo dessa amostragem pode ser visualizado no Anexo A.

Uma vez definida a aplicação de teste para o experimento, é possível iniciar a medição e comparação dos tempos de execução do experimento.

5.4 Obtenção dos Tempos de Transferência e Execução

A medição dos tempos de transferência e execução das tarefas demonstradas nos gráficos da seção 5.5, foram obtidos seguindo os passos abaixo:

1. Ao iniciar tanto uma operação de transferência, quanto execução de uma tarefa, o supervisor solicita ao mestre (nó que possui o serviço *GET*), que armazene o registro do instante em que a transferência ou execução iniciou no repositório de dados para a operação em questão.

2. Ao finalizar a operação, o serviço *GRAM* da máquina escrava notifica seu supervisor informando-o que a operação está concluída, podendo assim obter o tempo gasto, tanto para a transferência de arquivos, quanto para a execução de tarefas.

3. O supervisor por sua vez, solicita ao mestre que armazene um registro daquele instante no repositório de dados para a operação que foi concluída.

O tempo de duração de cada operação é determinado pela soma dos tempos de transmissão, tempo de execução, e o tempo de retorno do resultado, conforme descrito na equação 3.1 na página 55.

5.5 Análise de Desempenho

Para cada configuração do cluster foi disparada a execução de uma aplicação composta por 170 tarefas que compartilham um único arquivo de entrada. O objetivo de utilizar um único arquivo de entrada para todas as tarefas é analisar uma aplicação com o compartilhamento máximo de arquivos. Além disso, esse tipo de compartilhamento pode ser encontrado em aplicações reais, como por exemplo mineração de dados. As 170 tarefas são homogêneas, utilizam um mesmo arquivo de dados de 1,5MB, e executam o algoritmo de classificação *J48* descrito na seção 5.3.1.

A tabela 2 ilustra todos os tempos médios de transferência de arquivos denominado *Stage-In*, os tempos médios de retorno do resultado denominado *Stage-Out*, bem como os tempos médios de *Makespan* (tempo entre o início da aplicação e seu término), que foram obtidos variando o número de escravos de 1 a 17, e 1 ou 2 supervisores. Os valores que aparecem na tabela representam a média de 8 execuções de cada experimento. Portanto, foram executadas 2380 tarefas para cada configuração de computadores

definidas na seção 5.3. É possível observar ainda na tabela 2, que nas colunas *Stage-In* e *Stage-Out* à medida que aumenta o número de máquinas de cada configuração os tempos médios aumentam também, isso ocorre porque o supervisor é a máquina responsável por enviar os arquivos para cada escravo, e gerenciar as operações de *Stage-In* e *Stage-Out* dos escravos. Com o aumento do número de escravos trabalhando concorrentemente, aumenta a divisão do tempo do supervisor entre os escravos, tornando a operação mais lenta. Já o tempo execução da tarefa medido no próprio escravo desconsiderando os tempos de *Stage-In* e *Stage-Out*, não é alterado.

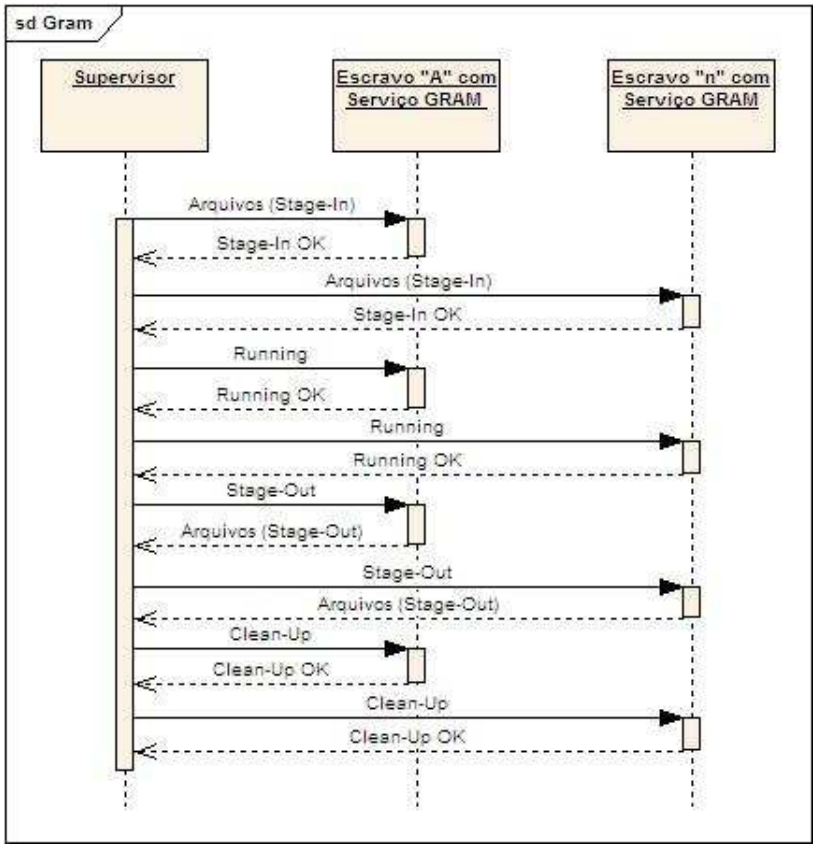
Para que os tempos obtidos possam ser registrados no repositório de dados controlado pelo supervisor, antes de enviar a tarefa para o escravo, o supervisor registra o instante em que a tarefa foi enviada, e tão logo o escravo termine de executá-la, informa ao seu supervisor que a tarefa foi concluída. Como o supervisor mantém o controle de diversos escravos, ao receber as notificações dos escravos informando que as tarefas foram concluídas ele leva um tempo para poder processar todas essas notificações. Com isso é gerado um atraso entre o real tempo de término da tarefa e o efetivo registro desse tempo no repositório de dados, já que o escravo envia para o supervisor somente um aviso que a tarefa foi concluída, e não propriamente o horário em que a tarefa foi concluída. A seção 5.4 descreve o funcionamento do envio e recebimento de tarefas e os devidos registros de tempos no banco de dados.

É possível observar na tabela 2 que comparando-se os valores entre a utilização de 1 supervisor e 2 supervisores, tanto da coluna *Stage-In* quanto *Stage-Out*, quando a configuração utiliza 2 supervisores os valores são menores do que quando é utilizado somente 1 supervisor. Essa diferença pode ser justificada da seguinte maneira: com a elevação do número de escravos e conseqüentemente do número de requisições para um único supervisor, este tende a ficar saturado mais rapidamente, passando a demorar a processar todas as requisições que lhe são feitas e, por conseqüência, piorando o makespan da aplicação. Ao adicionar mais um supervisor na configuração, as requisições serão

divididas entre eles, proporcionando uma resposta mais rápida para todas as requisições feitas a eles.

A figura 41 representa os estágios para a execução de uma tarefa qualquer utilizando-se a ferramenta *GRAM* do GT4. A seqüência dos estágios são:

Figura 41: Estágios da Execução de uma Tarefa Utilizando o Serviço GRAM



Fonte: Próprio Autor

1. Stage-In, consiste na transferência dos arquivos do supervisor para o escravo necessários para a execução de uma determinada tarefa no escravo;

2. Running, consiste na execução da tarefa propriamente dita no escravo;

3. Stage-Out, neste estágio o escravo envia para seu supervisor os arquivos com os resultados da execução da tarefa, quando os mesmos existirem;

4. Clean-Up, consiste na exclusão dos arquivos enviados para o escravo e/ou os arquivos que foram gerados no escravo através da execução da tarefa.

De acordo com a figura 41 o supervisor pode gerenciar uma ou mais execuções em paralelo em diversos escravos, e como a cada término de um estágio cada escravo envia algum tipo de resposta para o supervisor, podendo gerar um *overhead* e conseqüentemente atrasos no supervisor. Nas configurações do *cluster* que utilizam 2 supervisores o *overhead* formado nesses supervisores diminui, pois a tarefa de gerenciar os diversos escravos fica distribuída entre eles, e conseqüentemente os supervisores passam a ter um tempo de resposta mais rápido

A figura 42 demonstra que o *makespan* da aplicação utilizando 17 máquinas escravas, em uma hierarquia do tipo mestre/supervisor/escravo com 2 supervisores que utiliza agrupamento de arquivos é $20,29\%$ mais rápida do que o mesmo conjunto de tarefas sendo executadas em uma arquitetura puramente mestre/escravo com agrupamento de arquivos e, $35,45\%$ mais rápida se comparado com a arquitetura mestre/supervisor/escravo com 2 supervisores, contudo, sem o agrupamento de arquivos.

	Médio Stage-In		**Médio Stage-Out**		**Makespan**		
	1 Sup.	**2 Sup.**	**1 Sup.**	**2 Sup.**	**1 Sup.**	**2 Sup.**	
Qtd.Maq.	**CA**	**CA**	**CA**	**CA**	**CA**	**CA**	**SA**
2	80,90	69,01	15,80	14,50	2780,00	2696,00	4209,00
3	88,49		17,92		2100,00		
4		70,59		18,20		1650,00	2652,00
6	106,30	72,50	23,22	20,74	1298,33	1211,33	2077,00
10	163,71	92,16	37,17	27,50	1066,33	931,00	1554,00
14	200,78	114,34	51,98	33,42	1024,25	840,00	1307,00
17	218,64	135,00	57,82	39,94	979,00	780,33	1209,00

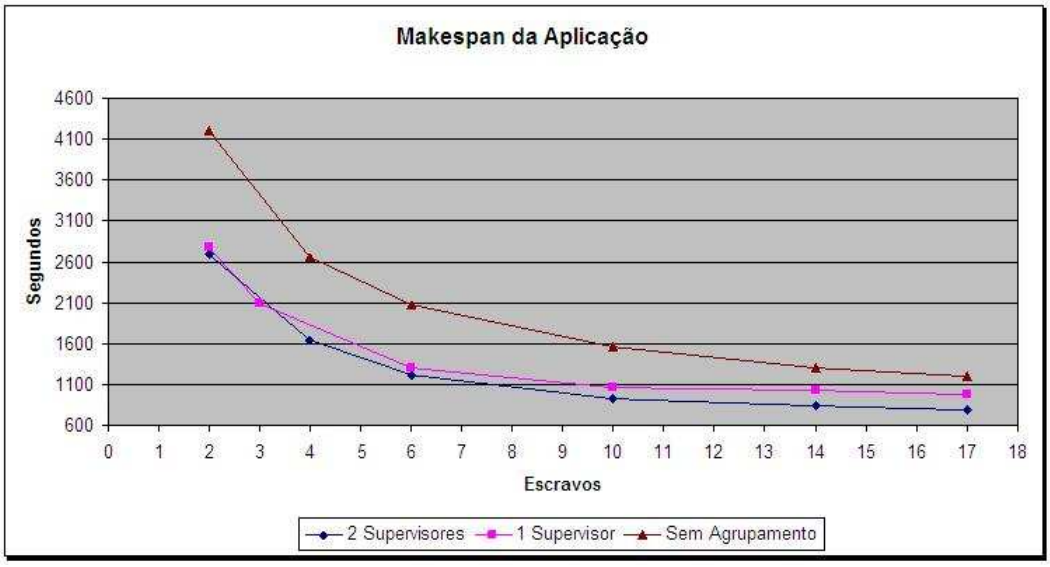
Tabela 2: Medições

(Valores expressos em segundos)

Legenda CA:Com Agrupamento - SA:Sem Agrupamento

A figura 43 representa os tempos médios obtidos na fase de *stage-in* dos arquivos que cada tarefa necessita no ato de sua execução. Essa média é calculada através da somatória do tempo de *stage-in* dos arquivos para cada escravo. É importante lembrar que, quando é utilizada a técnica do agrupamento, cada arquivo é transmitido uma única vez para cada escravo que irá executar a tarefa que necessita desse arquivo. Observando a mesma figura, fica nítido que na utilização da arquitetura mestre/escravo, quanto maior

Figura 42: Makespan da Aplicação



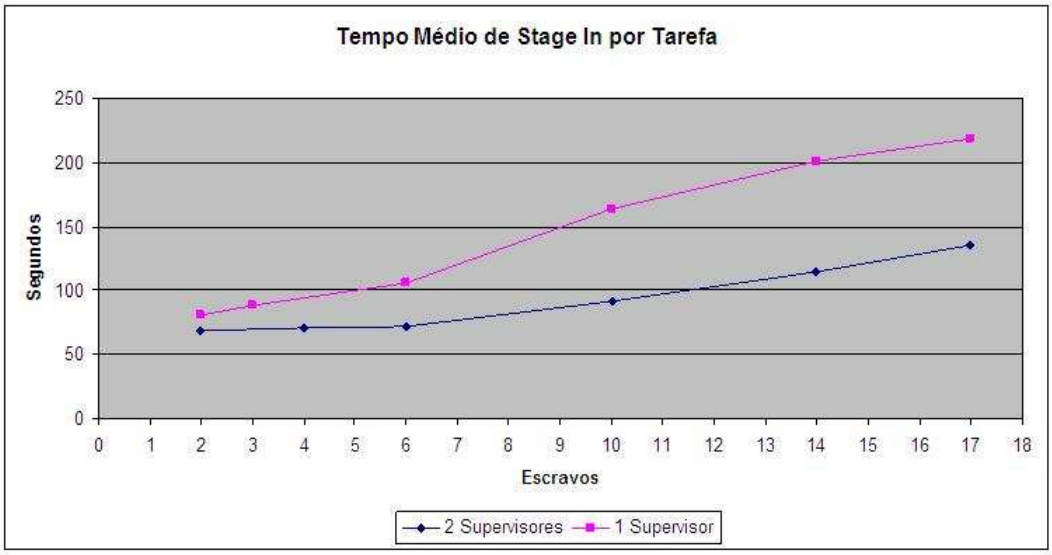
Fonte: Próprio Autor

for o número de escravos que compõem a grade, pior será seu desempenho no quesito tempo de *Stage-In*. Isso ocorre porque o *overhead* criado na fase de *Stage-In* no mestre aumenta. Já com a utilização de uma arquitetura mestre / supervisor / escravo o tempo de *stage-in* dos arquivos tende a ficar mais estável. Isto ocorre porque o *overhead* que existia no mestre na arquitetura anterior foi distribuído em duas máquinas, nesse caso os dois supervisores da configuração desse *cluster*. Se for comparado os tempos de cada um com a utilização de 17 escravos, é possível concluir que a segunda arquitetura é $38,25\%$ melhor do que a primeira, refletindo diretamente no *makespan* da aplicação.

É possível concluir que, colocando-se mais supervisores na configuração do *cluster* não só o tempo de *stage-in* diminui quanto o tempo de *stage-out* diminui também, conforme visto na figura 44. Contudo, é prudente que a inclusão de mais supervisores no *cluster* somente ocorra quando os supervisores já existentes estiverem saturados. Para determinar um número de supervisores ideal para uma determinada aplicação sem que os mesmos fiquem saturados pode ser obtido através da fórmula 3.12 existente na seção 3.5.

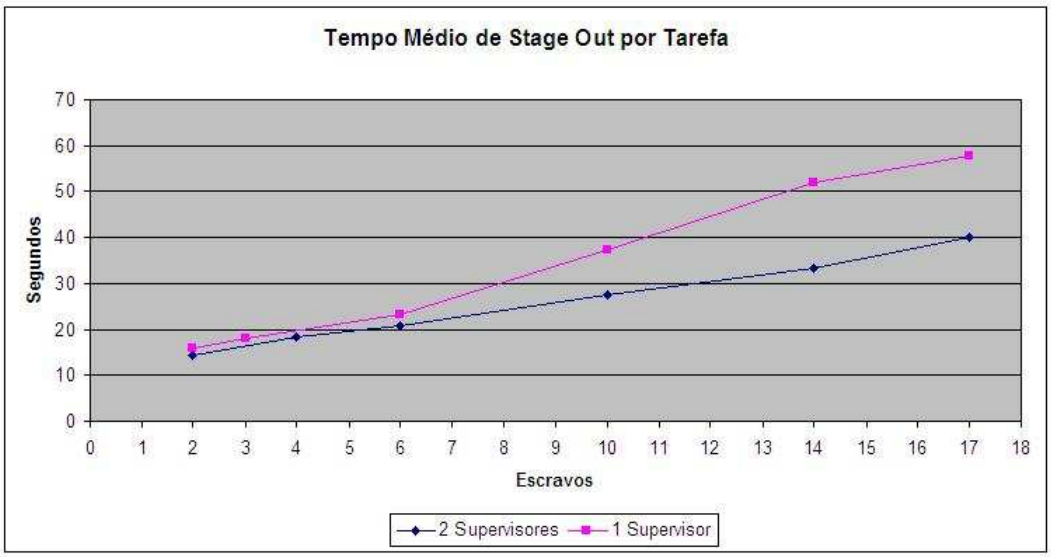
A figura 44 ilustra que na execução com a utilização de 1 ou 2 supervisores, ambos com agrupamento de arquivos, com até 6 máquinas escravas o tempo de stage-

Figura 43: Tempo de Stage-In



Fonte: Próprio Autor

Figura 44: Tempo de Stage-Out



Fonte: Próprio Autor

out praticamente não é alterado. Contudo, a partir desse ponto o supervisor passa a ficar cada vez mais sobrecarregado, influenciando diretamente no processo de registro no repositório de dados e conseqüentemente no tempo médio de makespan da aplicação como foi explicado anteriormente.

Conforme se observa na tabela 2, quando a execução ocorre sem o agrupamento de arquivos, o tempo médio total de *makespan* é praticamente o dobro, o que mostra um ganho significativo em favor do agrupamento. Isso se deve ao fato de que os supervisores que trabalham sem o agrupamento tem maior processamento a ser feito, pois, além de monitorarem a execução das tarefas em cada escravo, eles também necessitam fazer uma nova transferência de arquivo para cada nova tarefa que é executada em cada escravo, gerando mais processamento.

A tabela 3 utiliza os dados da tabela 2 para ilustrar os valores do *speedup* e da eficiência.

		Speedup			**Eficiência**		
	Tempo	**1 Sup.**	**2 Sup.**		**1 Sup.**	**2 Sup.**	
Qtd. Maq.	**Seqüencial**	**CA**	**CA**	**SA**	**CA**	**CA**	**SA**
2	2234	0,80	0,83	0,53	0,40	0,41	0,27
3	2234	1,06			0,35		
4	2234		1,35	0,84		0,34	0,21
6	2234	1,72	1,84	1,08	0,29	0,31	0,18
10	2234	2,10	2,40	1,44	0,21	0,24	0,14
14	2234	2,18	2,66	1,71	0,16	0,19	0,12
17	2234	2,28	2,86	1,85	0,13	0,17	0,11

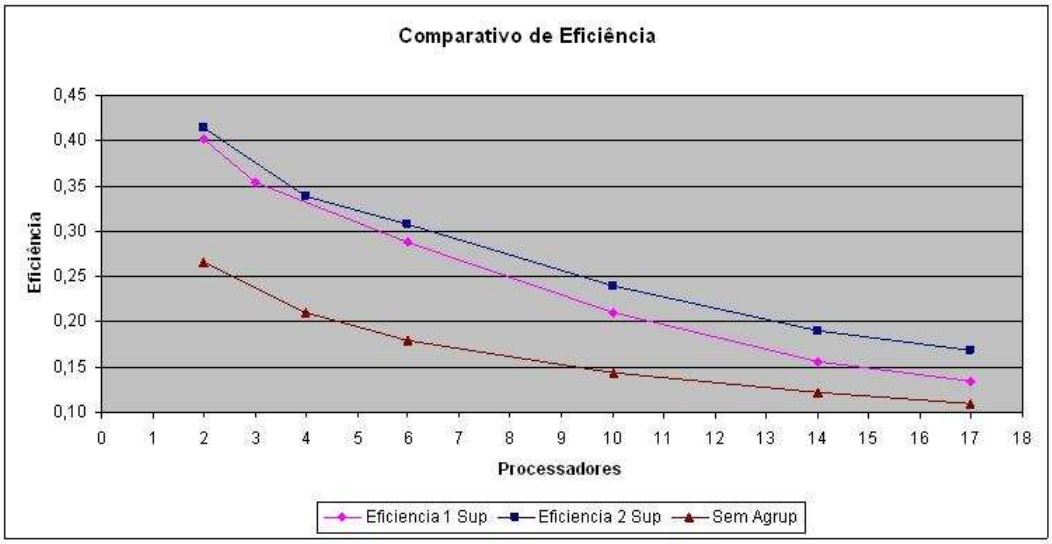
Tabela 3: Valores de Speedup e Eficiência

Legenda CA:Com Agrupamento - SA:Sem Agrupamento

A eficiência obtida nos testes, demonstrou, conforme ilustra a figura 45, que a execução da aplicação mais eficiente obtida foi com a utilização de 2 supervisores com agrupamento, seguido da utilização de um único supervisor e por final a utilização de 2 supervisores sem o agrupamento. Isso confirma que nesse experimento ao ser utilizado mais que 6 máquinas escravas com um único supervisor existe a necessidade da colocação de mais um supervisor.

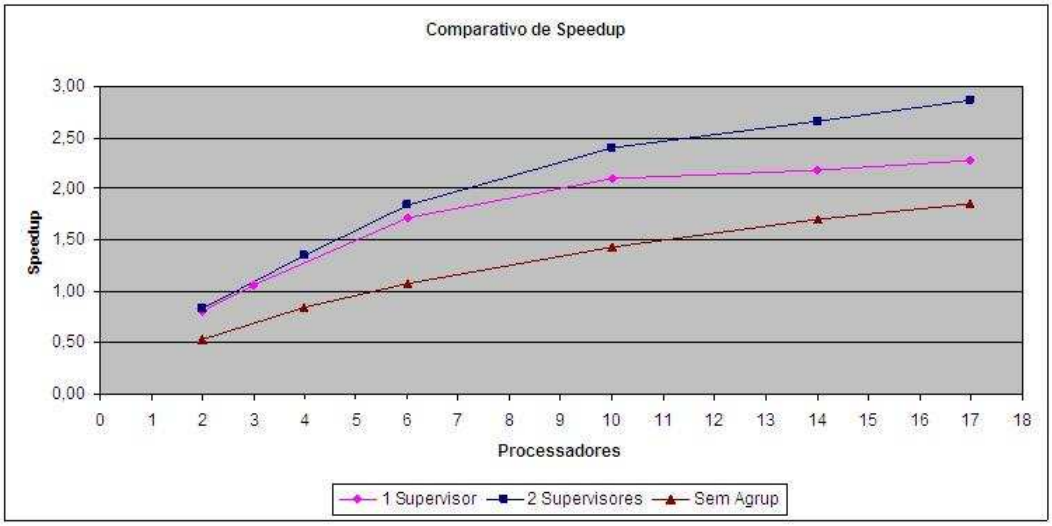
É possível fazer um comparação entre os três últimos resultados na utilização de 17 escravos. Essa comparação indica que a utilização de 2 supervisores com agrupamento foi $30,76\%$ melhor que a utilização com 1 supervisor com agrupamento e $54,54\%$ se comparado com a utilização de 2 supervisores sem agrupamento.

Figura 45: Eficiência



Fonte: Próprio Autor

Figura 46: Speedup



Fonte: Próprio Autor

O *speedup* obtido com o experimento demonstra que, com a utilização de um único supervisor a partir da colocação do décimo escravo seu *speedup* não melhora muito. Já com a utilização de 2 supervisores é possível observar que mesmo com 17 escravos na estrutura seu *speedup* tende a aumentar, e o pior caso foi com a utilização de 2 supervisores sem agrupamento. É possível observar pelo *speedup* que na configuração com 17 escravos e 1 supervisor com a utilização da técnica de agrupamento de arquivos apresentou-se 20%

pior que o melhor *speedup* obtido de todas as configurações. Observa-se também que o *speedup* na configuração que utiliza 17 escravos e 2 supervisores sem a utilização da técnica de agrupamento de arquivos apresentou-se 35% pior que o melhor *speedup* obtido dentre todas as configurações.

5.6 Transferência de Arquivos

Para a transferência de arquivos nesse trabalho foram implementadas as técnicas de agrupamento descrito na seção 3.4.1 e o modelo hierárquico descrito na seção 3.4.3. Além disso, através de experimentos que procuram analisar somente a transmissão de arquivos para a execução de uma determinada tarefa, constatou-se que a transmissão de arquivos agrupados é mais vantajosa do que transmitir os mesmos arquivos individualmente, ou seja, caso uma determinada tarefa A necessite dos arquivos f_1, f_2 e f_3 e uma tarefa B dos arquivos f_1, f_2 e f_4, sendo que ambas as tarefas serão executadas no computador C_1, é mais vantajoso fazer uma única transferência dos arquivos f_1, f_2, f_3 e f_4 do que efetuar a uma transferência para os arquivos f_1, f_2 e f_3 e posteriormente efetuar outra transferência somente do arquivo f_4.

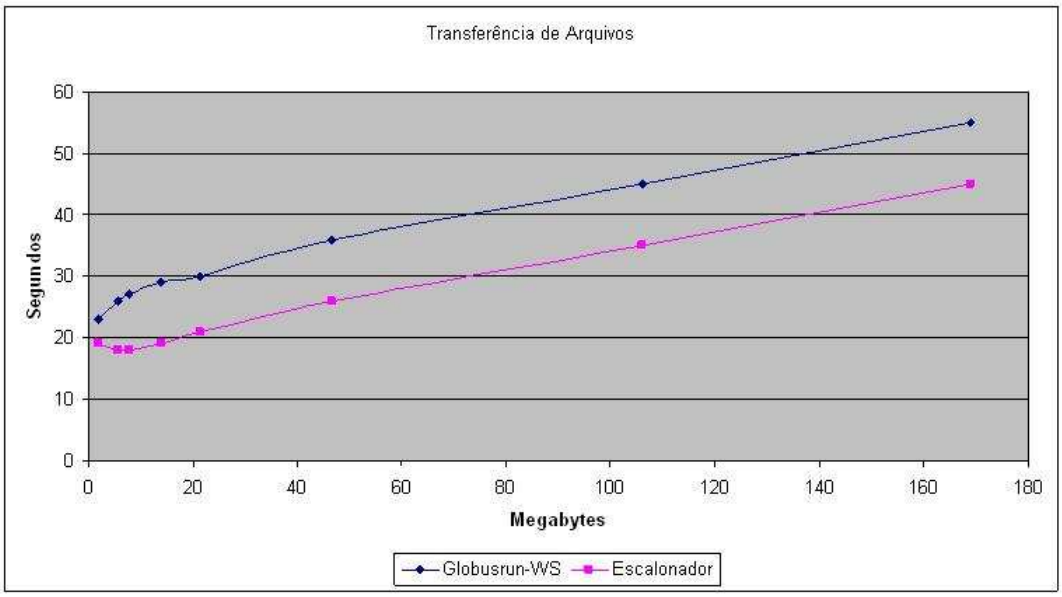
O experimento referenciado acima consiste em efetuar a transferência de arquivos com aproximadamente $1.94MB$, $5.53MB$, $7.65MB$, $13.87MB$, $21.49MB$, $46.71MB$, $106.03MB$ e $168.93MB$ cada. O experimento demonstrou que o mecanismo *GridFTP*, responsável pela transferência de arquivos, existente na ferramenta GT4 trabalha melhor quando é solicitado transferências de arquivos grandes do que pequenos. As figuras 47 e 48 demonstram tal ganho.

A ferramenta GT4 possui um conjunto de classes escritas em linguagem Java que proporcionam a criação de utilitários que possam interagir com os diversos mecanismos internos existentes na ferramenta. Afim de justificar o uso dessas classes do GT4, foram executados testes, visando apresentar uma comparação entre o uso dessas classes e o utilitário *GlobusRun-WS* escrito em linguagem *C*, nativo do GT4. Esse utilitário tem por finalidade prover a execução de tarefas, e quando essas tarefas necessitam de arquivos

o mecanismo *GridFTP* é invocado para fazer tais transferências. A figura 47 ilustra a comparação entre os tempos de transferência de arquivos com a utilização do utilitário nativo do GT4 e um utilitário de transferência de arquivos criado em Java que utiliza as classes disponibilizadas pelo GT4.

Os valores foram obtidos através da execução de um mesmo experimento 20 vezes. Para essa execução foram utilizados um notebook com processador *Celeron-M* de 1.46 GHZ e memória de 512 MB, e um computador desktop com processador *Sempron 2600+* de 1.83 GHZ com 512 MB de memória, ambos interligados por uma rede de comunicação de velocidade igual a 100 Mbits por segundo.

Figura 47: Comparativo do mecanismo de transmissão de arquivos

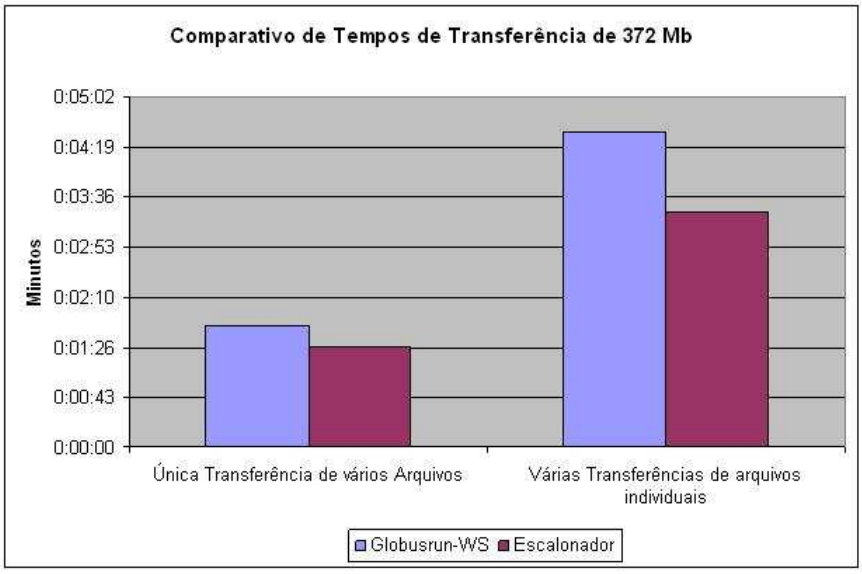


Fonte: Próprio Autor

De acordo com a figura 48, é possível observar que o melhor desempenho obtido na transferência foi agrupando todos os arquivos necessários para o conjunto de tarefas. O total de bytes transferidos é o mesmo. Contudo, a grande diferença no desempenho ocorre pelo fato que o GT4 necessita sempre iniciar os mecanismos de verificação antes de efetuar a transmissão, de modo a evitar problemas. Portanto, quando são executadas várias transferências é necessário chamar diversas vezes esse mecanismo de verificação, o que prejudica o desempenho geral da transmissão.

A figura 47 ilustra que o mecanismo de transferência do GT4 trabalha melhor quando o mesmo efetua grandes transferências de dados, como por exemplo, quase dobrando-se o tamanho do arquivo para transferência de $\cong 7MB$ para $\cong 13MB$ o tempo total de transferência subiu somente $\cong 6\%$.

Figura 48: Comparativo de Tempos de Transferência



Fonte: Próprio Autor

5.7 Conclusão

Foram apresentados a implementação e os teste de um protótipo com o objetivo de avaliar a técnica de agrupamento e o modelo hierárquico. Foram implementados a técnica de agrupamento e o modelo hierárquico. Foram criados cenários com variações de 2 a 17 computadores com a finalidade de formar uma grade computacional. Foi também criado um aplicativo capaz de produzir documentos do tipo *JSDL* com a finalidade de gerar carga para processamento nessa grade computacional.

Foram executados 3 tipos de testes, sendo um deles com apenas 1 supervisor representando a arquitetura mestre/escravo, outro com 2 supervisores aplicando a técnica de agrupamento para representar a arquitetura mestre/supervisor/escravo e o último também com 2 supervisores sem a utilização da técnica de agrupamento.

Após a realização dos testes foi possível observar que em todas as comparações a não utilização da técnica de agrupamento compromete o desempenho. Com isso é possível concluir que para todas as arquiteturas avaliadas, a técnica de agrupamento mostrou-se vantajosa, levando a uma melhoria do desempenho, que tem como finalidade reduzir o total de *bytes* transferidos pela rede.

A adição de mais supervisores em uma arquitetura mestre/supervisor/escravo se faz necessária a partir de um determinado ponto, sendo que esse ponto pode variar de acordo com as configurações das máquinas e da infra-estrutura utilizada bem como da própria aplicação. Nesse experimento ficou demonstrado que o número ideal de escravos para cada supervisor está entre 7 e 10 escravos para cada um. Não foi possível realizar testes com 3 supervisores devido a limitações de recursos computacionais.

Observa-se que as técnicas de agrupamento e uso de hierarquias em grades computacionais propiciam melhor desempenho no caso de aplicações que envolvem transferências de dados significativas. O desempenho obtido com o escalonador proposto foi bom, conforme o esperado, por reduzir o total de *bytes* transferidos pela rede e proporcionar uma redução no gargalo que existia no nó mestre, e conseqüentemente possibilitando a adição de novos recursos computacionais na arquitetura.

6 Conclusão e Trabalhos Futuros

O objetivo do presente trabalho foi propor uma ferramenta que implemente o escalonamento de tarefas do tipo BoT com compartilhamento de arquivos sobre uma grade computacional. Foram estudados os problemas de escalabilidade sobre a arquitetura mestre/escravo que pode apresentar problemas de escalabilidade quando o nó mestre ultrapassar o limite de gerenciamento de escravos.

A arquitetura hierárquica realmente leva a uma melhoria de desempenho por aliviar o gargalo que antes era formado no nó mestre, distribuindo parte da carga de serviço com os nós supervisores.

A técnica de agrupamento de tarefas também mostrou-se eficiente na redução dos gargalos, porque os arquivos que eram antes transferidos diversas vezes pela rede, agora são transferidos em um número bem menor de vezes, diminuindo o trafego pela rede.

Os resultados obtidos nos testes apresentados no capítulo 5 demonstram que a arquitetura proposta no capítulo 4 proporciona ganhos de desempenho, conforme esperado, pois o uso do modelo hierárquico fez com que fosse diminuido o gargalo formado no mestre quando a arquitetura é mestre/escravo. Além disso, foi verificado também que com a utilização do agrupamento de tarefas e arquivos de entrada, a utilização da banda de rede pode ser reduzida, tornando mais rápido o início das execuções das diversas tarefas e conseqüentemente o makespan da aplicação pode ser reduzido.

A implementação dessa nova arquitetura foi feita com a utilização linguagem Java, sendo executada sobre a grade computacional Globus Tool Kit 4.0.

É possível concluir também que a implementação da arquitetura hierarquia e da técnica de agrupamento é viável permitindo obter ganhos de desempenho.

Esse trabalho teve como principal contribuição propor e avaliar uma arquitetura de implementação das técnicas de agrupamento e hierárquicas, em uma grade computacional real. Outra contribuição deste trabalho é a disponibilização de uma ferramenta implementada em Java que pode ser utilizada ou atualizada em novos trabalhos.

6.1 Trabalhos Futuros

O presente trabalho apresenta diversas possibilidades de estudos e trabalhos futuros:

- Definir mecanismo para um cálculo mais preciso do número de escravos suportados por um determinado supervisor. Esse mecanismo pode ser auxiliado através de um agente instalado no supervisor que calcule a carga de trabalho executada por essa máquina e possa determinar o número de escravos suportados por ela;
- Proporcionar uma auto-instalação do serviço de supervisor em algumas máquinas da grade, fazendo a escolha da melhor máquina, e que esse serviço possa ser migrado para outras máquinas sem a necessidade de reiniciar a execução do aplicativo;
- Uso das técnicas aplicadas neste trabalho em outras plataformas de *clustes*, tais como *RMI*, pois desta maneira é possível formar um grade computacional de menor complexidade do que o Globus GT4.

Referências

ANJOMSHOAA, A. et al. Job submission description language (jsdl) specification version 1.0. *GGF - Global Grid Forum*, (Relatorio Tecnico), n. GFD-R.056, November 2005. Disponível em: <http://www.gridforum.org/documents/GFD.56.pdf>. Acesso em: 17 abr. 2007.

APST. *The Apples Parameter Sweep Template - Project Document*. 2005. Disponível em: <http://grail.sdsc.edu/projects/apst/apst_xml_man.html>. Acesso em: 12 jul. 2007.

ASUNCION, A.; NEWMAN, D. *UCI Machine Learning Repository*. 2007. Disponível em: <http://www.ics.uci.edu/∼mlearn/MLRepository.html>.

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, n. 15, p. 1437–1466, 2002. ISSN 0038-0644.

CASANOVA, H. et al. The apples parameter sweep template: User-level middleware for the grid. In: PROCEEDINGS OF THE SUPER COMPUTING CONFERENCE (SC'2000). Dallas, Texas, USA, 2000. p. 75–76.

CHINNICI, R. et al. *Web Services Description Language (WSDL) Version 2.0*. (W3C - Relatório Técnico), June 2007. Disponível em: <http://www.w3.org/TR/wsdl20/>.

CIRNE, W. et al. Labs of the world, unite!!! *Journal of Grid Computing*, v. 4, n. 3, p. 225–246, 2006. Disponível em: <http://dx.doi.org/10.1007/s10723-006-9040-x>.

CIRNE, W. et al. Running bag-of-tasks applications on computational grids: the mygrid approach. In: *Parallel Processing, 2003. Proceedings. 2003 International Conference on*. [s.n.], 2003. p. 407–416. Disponível em: <http://dx.doi.org/10.1109/ICPP.2003.1240605>.

CZAJKOWSKI, K. et al. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, v. 1459, p. 62+, January 1998. Disponível em: <http://www.metapress.com/link.asp?id=M7LL8Y4CHHNMFJ8U>.

FALLSIDE, D. C.; WALMSLEY, P. *XML Schema Part 0: Primer Second Edition*. (W3C - Relatorio Tecnico), October 2004. Disponível em: <http://www.w3c.org/TR/xmlschema-0/>.

FAYYAD, U.; SHAPIRO, P. G.; SMYTH, P. From data mining to knowledge discovery in databases. *Ai Magazine*, Univ Calif Irvine, Dept Comp & Informat Sci, Irvine, Ca, 92717 Gte Labs Inc, Knowledge Discovery Databases Kdd Project, Tech Staff, Waltham, Ma, 02254, v. 17, p. 37–54, 1996.

FELLER, M.; FOSTER, I.; MARTIN, S. Gt4 gram: A functionality and performance study. In: *Proceedings of the Teragrid 2007 Conference*. Madison, WI, USA: [s.n.], 2007.

FOSTER, I. et al. Open grid services architecture use cases. *GGF - Global Grid Forum*, (Relatorio Tecnico), n. GFD-I.029, 2004. Disponível em: <http://www.ggf.org/documents/GWD-I-E/GFD-I.029v2.pdf>.

FOSTER, I. et al. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. 2002.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The anatomy of the grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, v. 2150, 2001.

FOSTER, I. et al. The open grid services architecture, version 1.0. *OGF - Open Grid Forum*, (Documento Tecnico), n. GFD-I.030, January 2005. Disponível em: <http://www.ogf.org/documents/GFD.30.pdf>.

FREY, J. et al. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, v. 5, n. 3, p. 237–246, July 2002. Disponível em: <http://dx.doi.org/10.1023/A:1015617019423 ou http://www.cs.wisc.edu/condor/doc/condorg-hpdc10.pdf>.

GIERSCH, A.; ROBERT, Y.; VIVIEN, F. Scheduling tasks sharing files on heterogeneous master-slave platforms. *J. Syst. Archit.*, Elsevier North-Holland, Inc., New York, NY, USA, v. 52, n. 2, p. 88–104, 2006. ISSN 1383-7621.

GLOBUS. *GT - The Globus Toolkit, GT 4.1.1 GRAM4: User Guide*. 2007. Disponível em: <http://www.globus.org/toolkit/docs/development/4.1.1/execution/gram4/user/index.html#gram4-user-usagescenarios-jsdl>. Acesso em: 24 jul. 2007.

GRAMA, A. Y.; GUPTA, A.; KUMAR, V. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, IEEE Computer Society, Los Alamitos, CA, USA, v. 01, n. 3, p. 12–21, 1993. ISSN 1063-6552.

GUDGIN, M. et al. *SOAP Version 1.2 Part 1: Messaging Framework*. (W3C - Relatorio Tecnico), April 2007. Disponível em: <http://www.w3.org/TR/soap12-part1/>.

CASANOVA, H. et al. (Ed.). *Heuristics for Scheduling Parameter Sweep Applications in Grid Environments*. Heterogeneous Computing Workshop (HCW 2000) Proceedings 9th, Cancun, Mexico: IEEE Computer Society, 2000. 349–363 p.

HRUSCHKA, E. R.; EBECKEN, N. F. f. A genetic algorithm for cluster analysis. *Intell. Data Anal.*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 7, n. 1, p. 15–25, 2003. ISSN 1088-467X.

JACOB, B. et al. *Introdution to Grid Computing*. IBM - RedBooks, 2005. Disponível em: <www.ibm.com/redbooks>.

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, n. 2, p. 135–164, February 2002. ISSN 0038-0644. Disponível em: <http://portal.acm.org/citation.cfm?id=565296>.

LIU, L.; MEDER, S. Web services base faults 1.2. *OASIS Standard*, (Standard Document), April 2006. Disponível em: <http://docs.oasis-open.org/wsrf/wsrf-ws_base_faults-1.2-spec-os.pdf>.

MAGUIRE, T.; SNELLING, D.; BANKS, T. Web services service group 1.2. *OASIS Standard*, (Standard Document), April 2006. Disponível em: <http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-os.pdf>.

MAHESWARAN, M. et al. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In: *Heterogeneous Computing Workshop*. [S.l.: s.n.], 1999. p. 30+.

OURGRID. *OurGrid*: Ourgrid. 2007. Disponível em: <http://www.ourgrid.org/twiki-public/bin/view/OG/OurVision>. Acesso em: 17 abr. 2007.

SANTOS-NETO, E. et al. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. *10th Workshop on Job Scheduling Strategies for Parallel Processing*, Lecture Notes on Computer Science. Berlin : Springer-Verlag, Piscataway, NJ, USA, v. 3277, p. 210–232, 2004.

SATO, L. M.; MIDORIKAWA, E. T.; SENGER, H. Introdução a programação paralela e distribuída. In: *XVI Congresso da SBC*. Recife, PE: Anais da XV Jornada Atualização de Informática, 1996. p. 1–56.

SENGER, H.; SILVA, F. A. B.; NASCIMENTO, W. M. Hierarchical scheduling of independent tasks with shared files. In: *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2006. p. 51. ISBN 0-7695-2585-7.

SILVA, D. P. da; CIRNE, W.; BRASILEIRO, F. V. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. *LECTURE NOTES IN COMPUTER SCIENCE*, Springer-Verlag, Germany, p. 169–180, 2003. ISSN 0302-9743.

SILVA, F. A. B. da; CARVALHO, S.; HRUSCHKA, E. R. A scheduling algorithm for running bag-of-tasks data mining application on the grid. *Euro-Par 2004 Parallel Processing*, Springer Berlin / Heidelberg, Pisa, Italy, v. 3149/2004, p. 254–262, 2004. ISSN 0302-9743 (Print) 1611-3349 (Online).

SILVA, F. A. B. da; SENGER, H. Improving scalability of bag-of-tasks applications running on master-slave platforms. *(Relatório Técnico)*, 2008.

SILVA, M. P. dos S. Mineração de dados - conceitos, aplicações e experimentos com weka. *Livro da Escola Regional de Informática Rio de Janeiro*, SBC - Sociedade Brasileira de Computação, Porto Alegre, Espirito Santo, BR, v. 1, p. 1–20, 2004.

SOTOMAYOR, B.; CHILDERS, L. *Globus Toolkit 4 - Programming JAVA Services*. [S.l.]: Morgan Kaufmann, 2006.

SUN, X. H.; ROVER, D. T. Scalability of parallel algorithm-machine combinations. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 5, n. 6, p. 599–613, 1994. ISSN 1045-9219.

VENUGOPAL, S.; BUYYA, R.; WINTON, L. A grid service broker for scheduling distributed data-oriented applications on global grids. In: *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*. Toronto, Ontario, Canada: ACM, 2004. p. 75–80. ISBN 1-5... Acesso em: 20 mai. 2007.

VMWARE. *VMWare Player 2.0*: Vmware. http://www.vmware.com/product/player/: [s.n.], 2008. Acesso em: 05 mar. 2008.

VMWARE. *VMWare Workstation 6.0*: Vmware. http://www.vmware.com/product/ws/: [s.n.], 2008. Acesso em: 05 mar. 2008.

WITTEN, I. H.; FRANK, E. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. 2nd edition. ed. Morgan Kaufmann, 2005. Paperback. ISBN 1558605525. Disponível em: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/1558605525>.

XML Path Language Version 1.0. (W3C - Relatorio Tecnico), November 1999. Disponível em: <http://www.w3.org/TR/xpath>. Acesso em: 18 abr. 2007.

ANEXO A – Exemplo da Base de Dados do Censo de 1994 do Estados Unidos

Dados retirados do arquivo *(adult.arff)* utilizado para processamento nos tarefas submetidas ao escalonador.

39, State-gov, 77516, Bachelors, 13, Never-married, Adm-clerical, Not-in-family, White, Male, 2174, 0, 40, United-States, $<= 50K$

50, Self-emp-not-inc, 83311, Bachelors, 13, Married-civ-spouse, Exec-managerial, Husband, White, Male, 0, 0, 13, United-States, $<= 50K$

38, Private, 215646, HS-grad, 9, Divorced, Handlers-cleaners, Not-in-family, White, Male, 0, 0, 40, United-States, $<= 50K$

53, Private, 234721, 11th, 7, Married-civ-spouse, Handlers-cleaners, Husband, Black, Male, 0, 0, 40, United-States, $<= 50K$

28, Private, 338409, Bachelors, 13, Married-civ-spouse, Prof-specialty, Wife, Black, Female, 0, 0, 40, Cuba, $<= 50K$

37, Private, 284582, Masters, 14, Married-civ-spouse, Exec-managerial, Wife, White, Female, 0, 0, 40, United-States, $<= 50K$

49, Private, 160187, 9th, 5, Married-spouse-absent, Other-service, Not-in-family, Black, Female, 0, 0, 16, Jamaica, $<= 50K$

52, Self-emp-not-inc, 209642, HS-grad, 9, Married-civ-spouse, Exec-managerial, Husband, White, Male, 0, 0, 45, United-States, $> 50K$

31, Private, 45781, Masters, 14, Never-married, Prof-specialty, Not-in-family, White, Female, 14084, 0, 50, United-States, $> 50K$

42, Private, 159449, Bachelors, 13, Married-civ-spouse, Exec-managerial, Husband, White, Male, 5178, 0, 40, United-States, $> 50K$

37, Private, 280464, Some-college, 10, Married-civ-spouse, Exec-managerial, Husband, Black, Male, 0, 0, 80, United-States, $> 50K$

30, State-gov, 141297, Bachelors, 13, Married-civ-spouse, Prof-specialty, Husband, Asian-Pac-Islander, Male, 0, 0, 40, India, $> 50K$

23, Private, 122272, Bachelors, 13, Never-married, Adm-clerical, Own-child, White, Female, 0, 0, 30, United-States, $<= 50K$

32, Private, 205019, Assoc-acdm, 12, Never-married, Sales, Not-in-family, Black, Male, 0, 0, 50, United-States, $<= 50K$

40, Private, 121772, Assoc-voc, 11, Married-civ-spouse, Craft-repair, Husband, Asian-Pac-Islander, Male, 0, 0, 40, ?, $> 50K$

34, Private, 245487, 7th-8th, 4, Married-civ-spouse, Transport-moving, Husband, Amer-Indian-Eskimo, Male, 0, 0, 45, Mexico, $<= 50K$

25, Self-emp-not-inc, 176756, HS-grad, 9, Never-married, Farming-fishing, Own-child, White, Male, 0, 0, 35, United-States, $<= 50K$

32, Private, 186824, HS-grad, 9, Never-married, Machine-op-inspct, Unmarried, White, Male, 0, 0, 40, United-States, $<= 50K$

38, Private, 28887, 11th, 7, Married-civ-spouse, Sales, Husband, White, Male, 0, 0, 50, United-States, $<= 50K$

43, Self-emp-not-inc, 292175, Masters, 14, Divorced, Exec-managerial, Unmarried, White, Female, 0, 0, 45, United-States, $> 50K$

40, Private, 193524, Doctorate, 16, Married-civ-spouse, Prof-specialty, Husband, White, Male, 0, 0, 60, United-States, $> 50K$

54, Private, 302146, HS-grad, 9, Separated, Other-service, Unmarried, Black, Female, 0, 0, 20, United-States, $<= 50K$

35, Federal-gov, 76845, 9th, 5, Married-civ-spouse, Farming-fishing, Husband, Black, Male, 0, 0, 40, United-States, $<= 50K$