UNIVERSIDADE ESTADUAL PAULISTA
“Júlio de Mesquita Filho”
Pós-Graduação em Ciência da Computação

Patrícia Batista Franco

Escalonamento de Tarefas em Ambiente de Simulação de *Grid* Computacional

UNESP
2011

Patrícia Batista Franco

Escalonamento de Tarefas em Ambiente de Simulação de *Grid* Computacional

Orientadora: Profª. Adj. Roberta Spolon

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração - Sistemas de Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

UNESP
2011

Franco, Patrícia Batista.
Escalonamento de tarefas em ambiente de simulação de *Grid* computacional / Patrícia Batista Franco – São José do Rio Preto: [s.n.], 2011.
100 f. : il. ; 30 cm.

Orientador: Roberta Spolon
Dissertação (mestrado) – Universidade Estadual Paulista, Instituto de Biociências, Letras e Ciências Exatas

1. Computação. 2. Simulação (Computadores). 3. Computação em grade (Sistemas de computadores). I. Spolon, Roberta. II. Universidade Estadual Paulista, Instituto de Biociências, Letras e Ciências Exatas. III. Título.

CDU – 004.94

Patrícia Batista Franco

Escalonamento de Tarefas em Ambiente de Simulação de *Grid* Computacional

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração - Sistemas de Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

BANCA EXAMINADORA

Profª. Drª. Roberta Spolon
Professora Adjunto Doutora
UNESP – Bauru
Orientadora

Profª. Drª. Sarita Mazzini Brushi
Professora Doutora
Universidade de São Paulo – São Carlos

Prof. Dr. Marcos Antônio Cavenaghi
Professor Adjunto Doutor
UNESP – Bauru

Bauru, 01 de julho de 2011

Dedico este trabalho aos meus familiares e
amigos queridos que sempre me apoiaram.

AGRADECIMENTOS

Esse é um momento muito especial, pois tenho a oportunidade de agradecer a todos aqueles que me ensinaram, ajudaram, apoiaram e incentivaram durante os anos de mestrado e neste trabalho.

Aproveito a oportunidade, para agradecer também a todas as pessoas que estão sempre ao meu lado em todas as horas de minha vida; mesmo aquelas que estão fisicamente distantes, sei que posso contar sempre com vocês.

Durante esta etapa, passei por muitos momentos bons e ruins. Enfrentei diversos desafios, tropecei, caí muitas vezes, mas o mais importante é que sempre me levantei... e a cada tombo um novo aprendizado, uma nova conquista e um crescimento interior.

Sou Grata a Deus e a Jesus, por me concederem a oportunidade de aprender e crescer com os obstáculos e ter ao meu lado pessoas maravilhosas em todos os momentos para me ajudar, escutar, aconselhar, apoiar, ensinar e incentivar.

Eu sei que "*Só podemos dar aquilo que temos em nós mesmos*" (Wayne Dyer), por isso agradeço a colaboração de cada um.

Finalizo esta etapa com a certeza de ter dado o melhor de mim... Dever cumprido!!!!

Agradeço:

... aos meus pais Edison e Lúcia, minha irmã Graziela e meu cunhado Marco;

... aos meus tios, tias, primos e primas;

... aos meus amigos: Família Cruz (Juliana, Álvaro e Arthur); Família Victorino (Ari e Regina); Família Alves (Nilton, Sueli, Karine e Thais); Família Shindo (Naomi, Marcelo e Ricardo); Família Aguiar (Gilson, Irani, Inara e Guilherme); Família Mietto (Clóvis, Cláudia e Diego); Família Ishicava (Tadao, Marie, Rodrigo e Juliana); Família Bertonha (Luiz, Regina, Igor e Vitor); Weendely Pereira e família (Toninho e Gabriel); Andréa Toti, Igor Custódio, José Luiz Correa Jr., Dr. Marcelo Girão, Guilherme Trentini Duque e Jane Aquino;

... a minha orientadora Roberta Spolon;

... os professores do mestrado: Nilceu, Hilda, Cavenaghi e Marar;

... os professores: Humberto Ferasoli, Renê Pegoraro e M. Henrique Salgado;

... os funcionários do Depto. de Computação de Bauru e do LEPEC;

... aos colegas do mestrado: Júnior Rosante e Ives Bertoli;

... os professores: Jesus Andreo, Silvia Peres e Heitor Honório (FOB-USP);

... aos amigos e colegas da MSTECH: Rose e Pedro Vidor, Juan e Fernanda Falguera, Lilian Martins, Alexsander Moreira, Sérgio Lima, Washington Rodrigues, Isabella Fontes, Fábio dos Santos, Barbara De Franco, Tatiana Crepaldi, Bruno Penteado, Jorge Seki, Nicholas Assis, Simone Montalvão, Tania Vieira, Nazaré Leite e Rafael Arrivabene;

... aos colegas: Marcelo Esteves, Gustavo Dobkowoski, Alex Branti e Thiago Ferrarini.

Em agradecimento, ofereço a vocês a *Prece de Cáritas*:

Deus, nosso Pai, que sois todo Poder e Bondade, dai força àquele que passa pela provação, daí luz àquele que procura a verdade, ponde no coração do homem a compaixão e a caridade.

Deus! Dai ao viajor a estrela guia, ao aflito a consolação, ao doente o repouso.

Pai! Dai ao culpado o arrependimento, ao Espírito a verdade, à criança o guia, ao órfão o pai.

Senhor! Que Vossa bondade se estenda sobre tudo que criastes.

Piedade, Senhor, para aqueles que Vos não conhecem, esperança para aqueles que sofrem.

Que Vossa bondade permita aos Espíritos consoladores derramarem por toda parte a paz, a esperança e a fé.

Deus, um raio de luz, uma centelha do Vosso amor pode iluminar a Terra; deixai-nos beber nas fontes dessa bondade fecunda e infinita, e todas as lágrimas secarão, todas as dores acalmar-se-ão; um só coração, um só pensamento subirão até vós, como um grito de reconhecimento e amor.

Como Moisés sobre a montanha, nós Vos esperamos com os braços abertos, oh! Poder, oh! Bondade, oh! Beleza, oh! Perfeição, e queremos de algum modo alcançar a Vossa misericórdia.

Deus! Dai-nos a força de ajudar o progresso a fim de subirmos até Vós;
Dai-nos a caridade pura, dai-nos a fé e a razão;
Dai-nos a simplicidade que fará das nossas almas o espelho onde se refletirá a Vossa Imagem.

Que assim seja!

"O conhecimento é orgulhoso por ter aprendido tanto;
a sabedoria é humilde por não saber mais."
William Cowper

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS viii
LISTA DE FIGURAS ix
LISTA DE TABELAS x
RESUMO xi
ABSTRACT xii
1. INTRODUÇÃO 1
1.1. Motivação 3
1.2. Objetivos 3
1.3. Estrutura da dissertação 4
2. COMPUTAÇÃO EM *GRID* 5
2.1. Conceitos Básicos 5
2.2. Distinção entre *Grid* Computacional e Cluster 8
2.3. Arquitetura de *Grid* Computacional 9
2.4. Utilização Geral 11
2.5. Simulação 15
2.6. Ferramentas de Simulação em *Grid* Computacional 17
2.7. Considerações Finais 28
3. ESCALONAMENTO DE TAREFAS EM *GRID* COMPUTACIONAL 29
3.1. Considerações Iniciais 29
3.2. Escalonamento em Sistemas Computacionais Paralelos e Distribuídos 29
3.3. Escalonamento de Tarefas em *Grid* Computacional 33
3.4. Políticas de Escalonamento em *Grid* Computacional 36
3.5. Considerações Finais 50
4. DESENVOLVIMENTO DA BIBLIOTECA DE ESCALONAMENTO DE TAREFAS 51
4.1. Apresentando a LIBTS 51
4.2. Arquitetura da LIBTS 52
4.3. Simulação Utilizando o Módulo MSG 53
4.4. Considerações Finais 58
5. EXPERIMENTOS REALIZADOS 59
5.1. Validação da LIBTS 59
5.2. Estudo de Caso 64
5.2.1. Cenários com Plataforma de 5 Hosts 65
5.2.2. Cenários com Plataforma de 90 Hosts 69
5.3. Análises Estatísticas dos Cenários 74
5.4. Considerações Finais 77
6. CONCLUSÕES E TRABALHOS FUTUROS 78
6.1. Contribuições do Trabalho 79
6.2. Trabalhos Futuros 80
REFERÊNCIAS BIBLIOGRÁFICAS 81
ANEXO 87
Procedimentos para instalação da LIBTS 87
Menu inicial da LIBTS 87
Masterslave_bypass.c 88
Masterslave.c 93
Escalonamento.c 96
Escalona_suff.c 97

LISTA DE ABREVIATURAS E SIGLAS

APIs	*Application Programming Interface.*
BoT	*Bag-of-Task.*
FIFO	*First In First Out.*
MFLOPS	*Mega Floating point Operations Per Second.*
FPLTF	*Fastest Processor to Largest Task First.*
LIBTS	*Library Tasks Scheduling.*
LIFO	*Last In First Out.*
QoS	Qualidade de serviço.
RR	*Round Robin.*
SDKs	*Software Development Kits.*
SJF	*Shortest Job First.*
VO	Organização Virtual.
WQ	*Workqueue.*
WQR	*Workqueue with Replication.*
WQR	*Workqueue with Replication.*
XML	*eXtensible Markup Language.*
CT	*Completion Time.*
TBA	*Time to Become Available.*

LISTA DE FIGURAS

Figura 1 – Arquitetura de *Grid* dividida em camadas e sua relação com a arquitetura de protocolos da Internet (FOSTER; KESSELMAN; TUECKE, 2001) 10

Figura 2 – Visão geral das camadas do SimGrid. (SimGrid, 2010) 23

Figura 3 – A taxonomia hierárquica para algoritmos de escalonamento (CASAVANT, KUHL 1988) 31

Figura 4 – Esboço do algoritmo WQ 37

Figura 5 – Diagrama do algoritmo WQ integrado ao SimGrid 38

Figura 6 – Esboço do algoritmo WQR 39

Figura 7 – Diagrama do algoritmo WQR integrado ao SimGrid 40

Figura 8 – Esboço do algoritmo *Sufferage* 41

Figura 9 – Diagrama do algoritmo *Sufferage* integrado ao SimGrid 43

Figura 10 – Esboço do algoritmo *XSufferage* 44

Figura 11 – Diagrama do algoritmo *XSufferage* integrado ao SimGrid 45

Figura 12 – Esboço do algoritmo *Dynamic* FPLTF 47

Figura 13 – Diagrama do algoritmo *Dynamic* FPLTF integrado ao SimGrid 48

Figura 14 – SimGrid com a Biblioteca LIBTS 52

Figura 15 – Trecho do código-fonte “Masterslave_bypass.c” 55

Figura 16 – Trecho do código-fonte “Masterslave.c” 56

Figura 17 – Estrutura de uma aplicação no SimGrid 57
(a) Estrutura do módulo MSG do SimGrid
(b) Estrutura da LIBTS

Figura 18 – Execução do algoritmo *Sufferage* na LIBTS 61

Figura 19 – Média dos tempos de simulação de acordo com a Tabela 6 66

Figura 20 – Média dos tempos de simulação de acordo com a Tabela 7 67

Figura 21 – Média dos tempos de simulação de acordo com a Tabela 8 68

Figura 22 – Média dos tempos de simulação de acordo com a Tabela 9 69

Figura 23 – Média dos tempos de simulação de acordo com a Tabela 10 70

Figura 24 – Média dos tempos de simulação de acordo com a Tabela 11 71

Figura 25 – Média dos tempos de simulação de acordo com a Tabela 12 72

Figura 26 – Média dos tempos de simulação de acordo com a Tabela 13 73

LISTA DE TABELAS

Tabela 1 – Principais características das ferramentas de acordo com os estudos apresentados por Sulistio; Yeo; Buyya (2004), Quetier; Cappello (2005); Martins et al (2006) e Oliveira (2007).. 27

Tabela 2 – Principais características dos algoritmos WQ, WQR, *Sufferage*, *XSufferage*, *Dynamic* FPLTF.. 49

Tabela 3 – Especificação da aplicação do cenário de validação 59

Tabela 4 – Especificação da plataforma do cenário de validação... 60

Tabela 5 – Teste de mesa do cenário de validação 63

Tabela 6 – Média dos tempos de simulação em segundos com 100, 400 e 700 tarefas com 5 *hosts* 65

Tabela 7 – Média dos tempos normais de simulação em segundos com 1000, 2000 e 3000 tarefas com 5 *hosts* 66

Tabela 8 – Média dos tempos normais de simulação em segundos com 4000, 5000 e 6000 tarefas com 5 *hosts* 67

Tabela 9 – Média dos tempos normais de simulação em segundos com 7000, 8000, 9000 e 10000 tarefas com 5 *hosts* 68

Tabela 10 – Média dos tempos normais de simulação em segundos com 100, 400 e 700 tarefas com 90 *hosts* 70

Tabela 11 – Média dos tempos normais de simulação em segundos com 1000, 2000 e 3000 tarefas com 90 *hosts* 71

Tabela 12 – Média dos tempos normais de simulação em segundos com 4000, 5000 e 6000 tarefas com 90 *hosts* 72

Tabela 13 – Média dos tempos normais de simulação em segundos com 7000, 8000, 9000 e 10000 tarefas com 90 *hosts* 73

Tabela 14 – Média dos tempos de simulação em segundos com 5 hosts 74

Tabela 15 – Média dos tempos de simulação em segundos com 90 hosts 75

Tabela 16 – Análises estatísticas dos cenários com plataforma de 5 hosts 76

Tabela 17 – Análises estatísticas dos cenários com plataforma de 90 hosts 76

RESUMO

Diversos são os esforços para o desenvolvimento de políticas de escalonamento em *grid* computacional. O uso de simuladores de *grid* computacional é de especial importância para o estudo de algoritmos de escalonamento de tarefas. Através dos simuladores, é possível avaliar e comparar o desempenho de diferentes algoritmos em diferentes cenários. Apesar das ferramentas de simulação fornecerem funcionalidades básicas para simulação de ambientes distribuídos, elas não disponibilizam políticas internas de escalonamento de tarefas; além disso, a implementação dos algoritmos deve ser feita pelo próprio usuário. Portanto, o objetivo deste trabalho é desenvolver a biblioteca de escalonamento de tarefas LIBTS (*Library Tasks Scheduling*) e adaptá-la ao simulador SimGrid para oferecer aos usuários uma ferramenta que possibilite o estudo de algoritmos de escalonamento de tarefas em *grid* computacional. Através da LIBTS os usuários podem comparar os algoritmos implementados (*Workqueue* (WQ), *Workqueue with Replication* (WQR), *Sufferage*, *XSufferage*, *Dynamic* FPLTF) em diversos cenários, como também desenvolver e implementar novos algoritmos de escalonamento de tarefas. Este trabalho também proporciona uma revisão de literatura sobre *grid* computacional, apresentando as características e metodologias dos algoritmos implementados na LIBTS e as principais características das ferramentas de simulação. Além disso, os cenários de testes criados para comparar os algoritmos validaram o funcionamento da biblioteca e o funcionamento correto dos algoritmos na LIBTS.

Palavras-Chave: *Grid* computacional, escalonamento de tarefas em *grid* computacional, simulação.

ABSTRACT

Too much has been done to develop scheduling policies in computational grid. The use of computational grid simulators is particularly important for studying the algorithms of task scheduling. Through the simulators it's possible to assess and compare the performance of different algorithms in various scenarios. Despite the simulation tools provide basic features for simulation in distributed environments, they don't offer internal policies of task scheduling, so that the implementation of the algorithms must be realized by the user himself. Therefore, this study aims to develop the library of task scheduling LIBTS (Library Tasks Scheduling) and adapt it to the SimGrid simulator to provide the users with a tool to analyze the algorithms of task scheduling in the computational grid. Through the LIBTS, the users can compare the implemented algorithms (Workqueue (WQ), Workqueue with Replication (WQR), Sufferage, XSufferage, Dynamic FPLTF) in several scenarios, as well as to develop and implement new algorithms of task scheduling. This work also provides a literature review about the computational grid, presenting the characteristics and methodologies of the implemented algorithms in the LIBTS and the most important features of the simulation tools. Furthermore, the test scenarios created to compare the algorithms validate the library operation and the correct operation of the algorithms in LIBTS.

Key words: Computational Grid, task scheduling in computational grid, simulation.

1. INTRODUÇÃO

Grid computacional é uma infraestrutura de computação que conecta múltiplos recursos computacionais de diversos computadores, para permitir a execução de aplicações com alta demanda de recursos computacionais, memória e espaço em disco. A ideia do *grid* é utilizar todo recurso computacional ocioso para permitir que novos tipos de aplicações sejam criados, ou ainda para estender aplicações existentes utilizando recursos de sistemas distribuídos altamente conectados. De forma simplificada, um *grid* é um ambiente computacional de alto desempenho, o qual é caracterizado por prover o compartilhamento de serviços geograficamente distribuídos através de organizações distribuídas geograficamente (FOSTER, KESSELMAN, TUECKE, 2001; DANTAS, 2005; MAGOULÈS, 2009; TANENBAUM, 2010).

A computação em *grid* foi criada a partir de uma proposta computacional que se baseia na similaridade às malhas de energia elétrica. Deste modo, os usuários podem ter acesso aos recursos e serviços sem a necessidade de saber onde os mesmos estão localizados (BUYYA, 2002; DANTAS, 2005).

As características próprias do ambiente de *grid* computacional, como a heterogeneidade, comportamento dinâmico, distribuição em larga escala e compartilhamento de recursos, demonstram a complexidade desse ambiente. Devido a esses fatores, o escalonamento de tarefas é uma das principais técnicas estudadas em *grid* computacional. A atividade de escalonamento de aplicações consiste em alocar tarefas de uma aplicação paralela a recursos do *grid*. O problema de escalonamento é uma questão importante em um ambiente de *grid* computacional, pois um algoritmo de escalonamento eficiente deve fazer a distribuição das tarefas para os recursos apropriados, melhorando assim o desempenho da aplicação (CIRNE, 2002; ZHIHONG XU, XIANGDAN HOU, JIZHOU SUN, 2003; YU LIANG, ZHOU JILIU, 2007; CHO-CHIN LIN, CHUN-WEI SHIH, 2008; HAO TIAN, 2008; KUN-MING YU, CHENG-KWAN CHEN, 2008; NIANMING, PEIYU HONG, 2010; SHIHONG, HONG LUO, 2010).

Desde o surgimento do conceito de *grid* computacional, pesquisadores buscam a otimização dos problemas de escalonamento de tarefas neste ambiente. Estudos recentes

como os de Munir et al (2007); Cho-Chin Lin, Chun-Wei Shih (2008); Hao Tian (2008); Hong Jiang, Tianwei Ni (2009); Youchan, Xueying, Yanyan (2009); Ang, Nianming, Peiyu Hong (2010); Ku-Mahamud, Nasir (2010); Shihong, Hong Luo (2010); Sha (2010) demonstram os esforços contínuos dos pesquisadores no desenvolvimento de novas políticas de escalonamento de tarefas.

A análise do comportamento deste tipo de sistema pode ser feita através de modelos de simulação por meio de modelos analíticos, ou ainda por meio de experimentos em um sistema real. Por ocorrer complexas interações em sistemas distribuídos, como *grid* computacional, muitos parâmetros devem ser considerados, o que torna a modelagem analítica complicada (QUETIER, CAPPELLO, 2005).

Sendo assim, a simulação é uma maneira adequada para analisar algoritmos de sistemas distribuídos em larga escala de recursos heterogêneos e tem sido pesquisada e aplicada com sucesso para modelar processos, aplicações e objetos do mundo real (SULISTIO, YEO, BUYYA, 2004; SimGrid, 2010).

Consequentemente, o uso de simuladores para *grid* computacional é de especial importância para o estudo de algoritmos de escalonamento de tarefas, pois, através das ferramentas de simulação, torna-se possível analisar e comparar em diferentes cenários, o desempenho de diferentes algoritmos, como por exemplo: *Sufferage* (MAHESWARAN et al., 1999), *XSufferage* (CASANOVA et al, 2000), *Workqueue with Replication* (WQR) (SILVA, 2003), *Dynamic* FPLTF (PARANHOS, CIRNE, BRASILEIRO, 2003). Desta maneira, diversas ferramentas de simulação foram desenvolvidas para esse propósito, como por exemplo: Bricks (TAKEFUSA et al, 1999), GridSim (BUYYA, 2002), MicroGrid (SONG et. al., 2000), OptorSim (CAMERON, 2002), SimGrid (CASANOVA, 2001).

Apesar das ferramentas de simulação fornecerem funcionalidades básicas para simulação de ambientes distribuídos, não disponibilizam nenhum algoritmo de escalonamento de tarefas implementado.

1.1. Motivação

O gerenciamento de recursos e escalonamento de tarefas são problemas complexos e importantes em um ambiente de *grid* computacional devido à heterogeneidade dos recursos computacionais. Desta maneira, são diversos os esforços para o desenvolvimento de políticas de escalonamento, ambientes de negociação de recursos e escalonadores integrados com os serviços existentes em um sistema para computação em *grid* (MAHESWARAN et al., 1999; BUYYA, 2002; ZHIHONG, XIANGDAN, JIZHOU, 2003; SILVA, 2003; YU LIANG, ZHOU JILIU, 2007; KUN-MING, CHENG-KWAN, 2008; NIANMING, PEIYU HONG, 2010; SHIHONG, HONG LUO, 2010).

Normalmente, uma infraestrutura de teste real possui dificuldades como: custo elevado, limitação de recursos intensivos, causado na maior parte por alguns ambientes de área local, não fornecendo um ambiente configurável e de repetição para experimentação e avaliação de estratégias de escalonamento. Uma infraestrutura baseada em simulação contorna esses problemas (MURSHED, BUYYA, 2002; QUETIER, CAPPELLO, 2005; BUYYA, SULISTIO, 2008).

As ferramentas de simulação oferecem funções básicas e abstrações para a simulação de aplicações em ambientes distribuídos heterogêneos. Apesar do SimGrid ser uma ferramenta de simulação de aplicações em ambientes distribuídos heterogêneos, não disponibiliza políticas internas de escalonamento de tarefas, a implementação dos algoritmos deve ser feita pelo próprio usuário, desta maneira motiva-se o desenvolvimento da biblioteca LIBTS (*Library Tasks Scheduling*) para o SimGrid.

1.2. Objetivos

O objetivo deste trabalho é desenvolver a biblioteca de escalonamento de tarefas LIBTS (*Library Tasks Scheduling*) e integrá-la ao simulador SimGrid para auxiliar no estudo dos algoritmos de escalonamento de tarefas em *grid* computacional. A LIBTS é uma ferramenta que permitirá aos usuários dedicar-se ao estudo das políticas de escalonamento existentes através da comparação dos algoritmos implementados (*Workqueue* (WQ), *Workqueue with Replication* (WQR), *Sufferage*, *XSufferage*,

Dynamic FPLTF) em diversos cenários, como também desenvolver e implementar novos algoritmos.

1.3. Estrutura da dissertação

Este trabalho está dividido em 6 capítulos que estão descritos a seguir:

Capítulo 1 – Apresenta uma introdução do assunto tratado, com as motivações e objetivos desse trabalho.

Capítulo 2 – Descreve os conceitos básicos de *grid* computacional, sua arquitetura e utilização geral. Apresenta também a utilização da simulação em ambiente de *grid* computacional e as principais ferramentas de simulação.

Capítulo 3 – Refere-se ao escalonamento de tarefas em *grid* computacional e as políticas de escalonamento de tarefas utilizadas em *grid*.

Capítulo 4 – Apresenta a biblioteca LIBTS, sua arquitetura e utilização.

Capítulo 5 – Mostra os experimentos realizados através de cenários para validar a LIBTS.

Capítulo 6 – Finaliza a dissertação abordando as contribuições deste projeto e trabalhos futuros.

2. COMPUTAÇÃO EM *GRID*

Grid computacional é uma infraestrutura de hardware e software que compartilham recursos computacionais geograficamente distribuídos entre diversos usuários.

Este capítulo fornece uma revisão de *grid* computacional com a apresentação de conceitos básicos, arquitetura e utilização de forma geral. São apresentadas também a utilização de simulação em *grid* e ferramentas de simulação.

2.1. Conceitos Básicos

Em meados dos anos 90 foi proposta uma infraestrutura de hardware e software com suporte à execução de aplicações paralelas e acopladas a recursos distribuídos e heterogêneos. Essa infraestrutura foi denominada computação em *grid* ou *grid* computacional (*grid computing*) (FOSTER, KESSELMAN, 1999).

Alessandro Volta inventou a bateria elétrica em 1800. Posteriormente, Thomas Edison e Nikola Tesla difundiram a eletricidade inventando a lâmpada elétrica e a corrente alternada, que evoluiu para um malha de energia elétrica mundial. Por sua vez, essa malha de energia forneceu acesso confiável, consistente e intenso de energia para utilidade pública, tornando uma parte integral da sociedade moderna. Inspirados na expansão da rede de energia elétrica, facilidade de uso e confiabilidade, cientistas da computação, em meados da década de 90, começaram a explorar o modelo e o desenvolvimento de uma infraestrutura análoga chamada *grid* computacional para ampla área de computação paralela e distribuída (BUYYA, 2002).

A motivação por *grid* computacional foi inicialmente direcionada para larga escala, pois os recursos de aplicações científicas intensivas (computacional e de dados) que requerem mais recursos do que um único computador (PC, estação de trabalho, supercomputador ou cluster) pode ser oferecido em um único domínio administrativo. *Grid* computacional permite compartilhamento, seleção e agregação de uma ampla

variedade de recursos geograficamente distribuídos, incluindo supercomputadores, sistemas de armazenamento, fontes de dados (*data sources*) e dispositivos especializados de propriedade de diferentes organizações para solucionar problemas de recursos intensivos em larga escala na ciência, engenharia e comércio (BUYYA, 2002).

Para construir um *grid* computacional, é necessário o desenvolvimento e a distribuição de uma série de serviços. Isso inclui serviços de baixo nível como segurança, informação, gerenciamento de recursos (troca de recurso, alocação de recursos, qualidade de serviços) e serviços de alto nível. Além de ferramentas para o desenvolvimento de aplicações, gerenciamento de recursos e escalonamento (descoberta de recurso, negociação de custos de acesso, seleção de recursos, estratégias de escalonamento, qualidade de serviços e gerenciamento de execução). Dentre eles, os dois aspectos mais desafiantes da computação em *grid* são: gerenciamento de recursos e escalonamento (BUYYA, 2002; BUYYA, SULISTIO, 2008).

Grid computacional é uma nova classe de infraestrutura. Baseada na Internet e na World Wide Web, fornece escalabilidade, segurança, mecanismos de alto desempenho para descobrir e negociar acesso a recursos remotos. Essa nova classe promete ser útil às colaborações científicas para compartilhar recursos em uma escala sem precedentes para grupos distribuídos geograficamente. Antes do *grid* computacional esse compartilhamento de recursos era impossível (FOSTER, 2002).

O conceito de compartilhamento de recursos distribuídos não é novo. Em 1965, Fernando Corbató do MIT e outros projetistas do sistema operacional Multics previram uma facilidade computacional operando "como uma companhia de energia ou de água". Em 1968, Licklider e Taylor anteciparam cenários como de *grid*. Desde o final da década de 60, muito trabalho tem sido dedicado para o desenvolvimento de sistemas distribuídos. Uma combinação de tendências tecnológicas e avanços de pesquisa possibilitaram a compreensão da visão de *grid* computacional para colocar em cena uma nova infraestrutura internacional: a utilização de ferramentas que, juntas, pudessem encontrar os desafios da ciência do século XXI. De fato, a maioria das comunidades científicas aceitou que a tecnologia de *grid* computacional era importante para o futuro (FOSTER, 2002).

Com o intuito de esclarecer a definição de *grid* computacional, já que constantemente se confunde o real significado do que é *grid*, Foster (2002a) elaborou um *grid Checklist*, no qual são definidas três características básicas para que um sistema computacional possa ser chamado de *grid*:

1) Recursos coordenados que não se sujeitam a um controle centralizado: sistemas em *grid* podem englobar recursos entre os mais variados tipos, desde o desktop de um usuário até um supercomputador. Pode haver um controle local em uma empresa, mas não existe um controle central para todo o *grid*;
2) Padrões abertos, interfaces e protocolos de propósitos gerais: a utilização de protocolos e padrões abertos é essencial para que os sistemas em *grid* possam realizar funções fundamentais como autenticação, autorização, descoberta de recursos e acesso a eles, sem perder a capacidade de escalonar e interagir com diferentes plataformas de hardware e software;
3) Qualidade mínima de serviços: sistemas em *grid* permitem que os recursos sejam utilizados de forma coordenada com o objetivo de alcançar qualidades de serviço como: tempo de resposta, *throughput* (vazão), disponibilidade, segurança e/ou a co-alocação de recursos para se adequar às exigências do usuário. Assim, a utilidade de um sistema combinado é significativamente maior que a soma das partes.

Dantas (2005), Buyya e Sulistio (2008) consideram um *grid* como um ambiente computacional de alto desempenho, o qual é caracterizado por prover o compartilhamento geograficamente distribuído de serviços por organizações distribuídas geograficamente. Consideram também o *grid* sob o paradigma de melhoria de alguns aspectos físicos nas redes de comunicação e computadores, tais como:

- Melhor utilização de largura de banda;
- Utilização agregada de um grande poder computacional;
- Acesso rápido a dados, pacotes de software e dispositivos remotos com qualidade de serviço (QoS);
- Melhor utilização de processadores remotos, memórias e espaço em discos.

Em 2005, Ramalho escreveu que muitos dos conceitos de *grid* computacional estavam no mercado há anos, e a partir daquele ano foi possível fazer o uso integrado e funcional de todos esses conceitos e tecnologias. O *grid* computacional integra um número variável de máquinas capazes de atuar como um só equipamento. Isso acontece porque o *grid* computacional consolida virtualmente — por meio de uma camada de software — os servidores e os sistemas de armazenamento em um único *pool* de recursos. Uma das conseqüências dessa característica é que o processamento de aplicativos passa a ser distribuído por toda a rede de maneira dinâmica. A carga de trabalho é balanceada automaticamente em função da ociosidade e da capacidade de recursos existentes, evitando o desperdício e a subutilização dos equipamentos. Com isso, o novo modelo elimina as "ilhas de computação", pois permite que se monitore o uso de muitos servidores pequenos. Esses servidores, por sua vez, passam a atuar como uma única máquina de grande porte para executar aplicativos padrão.

Uma configuração de *grid* computacional pode ser entendida como uma plataforma de computadores geograficamente dispersos, na qual os usuários fazem acesso ao ambiente através de uma interface única. A diferença fundamental entre uma configuração de *grid* computacional e outra caracterizada como ambiente distribuído convencional é referente à grande quantidade de serviços e recursos que os ambientes de *grid* focam no seu compartilhamento (DANTAS, 2005; HAO TIAN, 2008).

2.2. Distinção entre *Grid* Computacional e Cluster

A principal diferença entre *grid* e cluster, está relacionada ao modo como é feito o gerenciamento dos recursos, ou seja, se o compartilhamento de recursos é gerenciado por um único sistema global, sincronizado e centralizado, então é um cluster. Em um cluster, todos os nós trabalham cooperativamente em um objetivo comum, e a alocação de recursos é executada por um gerente centralizado e global. Em um *grid*, os recursos estão distribuídos geograficamente, sendo que os donos desses recursos possuem autonomia para fazer o gerenciamento local. A alocação dos recursos é feita pelos usuários do *grid*, e os nós executam diferentes tarefas relacionadas a objetivos distintos. Um *grid* pode ser encarado como sendo uma evolução do cluster, dado que os recursos

deste podem ser compartilhados pelo *grid*. Assim, um *grid* é mais heterogêneo, complexo e distribuído (BUYYA, 2002; KU-MAHAMUD, NASIR, 2010).

De acordo com Ramalho (2005) embora o conceito de *grid* computacional possa se confundir com o conceito de cluster existem diferenças entre eles. Um cluster é uma das tecnologias utilizadas para criar uma infraestrutura de *grid*. Clusters simples possuem recursos estáticos usados por aplicativos predefinidos por alguns usuários. *Grids* computacionais, ao contrário, vão muito além: podem ser constituídas por vários clusters. Na prática, *grids* são *pools* dinâmicos, nos quais os recursos diversos de configurações distintas podem ser compartilhados entre vários usuários no processamento de muitos aplicativos.

De maneira semelhante, Dantas (2005) descreveu que a principal diferença entre clusters e *grids* computacionais é baseada na maneira pela qual os recursos e serviços são gerenciados. Em um cluster, os recursos são gerenciados por uma autoridade central e os computadores trabalham juntos como uma única entidade física. Nas configurações de um *grid*, cada organização virtual faz o gerenciamento de seus computadores não objetivando uma visão única do sistema. O usuário sabe da existência de inúmeros recursos e serviços disponíveis e que de alguma forma deverá fazer uma requisição de utilização dos mesmos.

2.3. Arquitetura de *Grid* Computacional

De maneira análoga a Internet e principalmente devido à experiência com a mesma, pesquisadores e técnicos propuseram modelos de arquitetura com o objetivo de uma padronização que permitisse a interoperabilidade entre diferentes organizações virtuais (FOSTER, KESSELMAN, TUECKE, 2001).

A arquitetura de um *grid* é baseada em protocolos, sendo tais protocolos responsáveis pelo estabelecimento da comunicação, gerenciamento e negociação de recursos, e demais ações relacionadas com a interação entre elementos do *grid*. A portabilidade do *grid* é beneficiada quando os protocolos utilizados são baseados em padrões abertos. Além disso, o uso de tais padrões facilita a escalabilidade, a portabilidade e o compartilhamento de código entre os envolvidos com o ambiente de

computação em *grid* (FOSTER, KESSELMAN, TUECKE, 2001; BUYYA, SULISTIO, 2008).

Além dos protocolos, as APIs (*Application Programming Interfaces*) e os SDKs (*Software Development Kits*) são utilizados para oferecer abstrações na programação dos *grids*. Os serviços são utilizados para oferecer aos usuários ações como acesso a capacidade de computação, descoberta de recursos e replicação de dados (FOSTER, KESSELMAN, TUECKE, 2001; BUYYA, SULISTIO, 2008).

A Figura 1 mostra uma arquitetura baseada em camadas (não são apresentados detalhes referentes à APIs, SDKs e serviços, apenas são endereçados aspectos gerais de um *grid*). Os elementos dessa arquitetura são descritos a seguir:

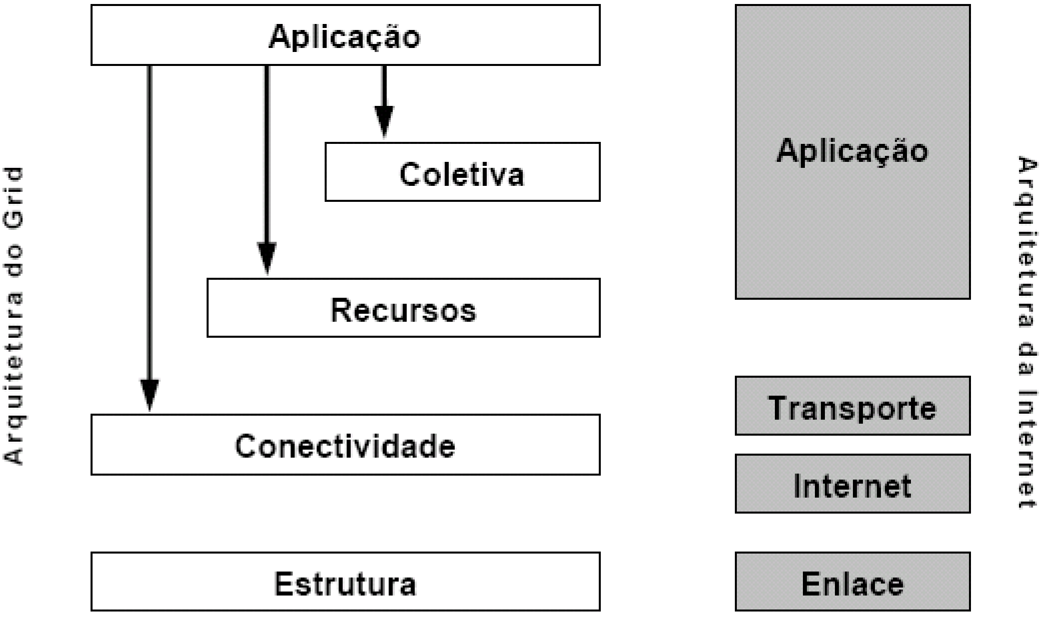

Figura 1 – Arquitetura de *grid* dividida em camadas e sua relação com a arquitetura de protocolos da Internet (FOSTER, KESSELMAN, TUECKE, 2001).

- Estrutura: os componentes desta camada implementam operações específicas locais que ocorrem em cada recurso como resultado das operações de compartilhamento nos níveis superiores. De um lado, devem ser implementados mecanismos de negociação que façam uma solicitação para obtenção de informações sobre a estrutura, o estado e as possibilidades dos recursos. Do outro lado, mecanismos de gerenciamento de recursos

devem fornecer formas de monitorar a qualidade de serviço (QoS – *Quality of Service*);

- Conectividade: esta camada define os protocolos básicos de comunicação e autenticação necessários para as transações de rede específicas do *grid*. Os protocolos de comunicação permitem a troca de dados entre os níveis de ambiente e recursos. Entre os requisitos de comunicação estão o transporte, o roteamento e o serviço de nomes. Os protocolos de autenticação constroem os serviços de comunicação de modo a prover mecanismos seguros e criptográficos para a verificação da identidade de usuários e recursos;
- Recursos: nesta camada são encontrados os protocolos de autenticação e comunicação do nível conectividade para definir protocolos (APIs e SDKs) que forneçam segurança na negociação, iniciação, monitoramento, controle, geração de relatórios e outros detalhes envolvidos nas operações com recursos individuais. As implementações destes protocolos chamam as funções da camada de estrutura para acessar e controlar os recursos locais;
- Coletiva: enquanto na camada de recursos são tratadas as operações de cada recurso individualmente, nesta camada os componentes atuam nas interações entre coleções e recursos. Os componentes dessa camada baseiam-se nos níveis recursos e aplicação e implementam uma grande variedade de serviços, como: Serviços de diretório: que permitem aos membros de uma organização virtual descobrir quais são os recursos desta organização; Servidores de autorização comunitários: que reforçam a política de acesso aos recursos;
- Aplicação: esta camada compreende as aplicações dos usuários que operam no ambiente da organização virtual. Os níveis anteriores proveem serviços úteis às aplicações desenvolvidas que as invocam.

2.4. Utilização Geral

O desenvolvimento de *grid* computacional foi motivado por requisitos de comunidades de usuários profissionais que necessitavam de acesso a recursos como grandes conjuntos de computadores que eram utilizados em simulações e análise de

dados em larga escala. A disseminação desta tecnologia possibilitou o surgimento de projetos com diferentes propósitos, variando no tamanho, escopo, duração, estrutura, comunidade e sociologia (FOSTER, KESSELMAN, TUECKE, 2001).

De acordo com Foster, Kesselman, Tuecke (2001); Berstis (2002) e Buyya, Sulistio (2008) as grandes capacidades do *grid* computacional são:

- Explorar recursos subutilizados e recursos adicionais: uma aplicação pode ser executada em qualquer uma das máquinas participantes, desde que estas possuam acesso a determinados recursos solicitados pela aplicação. Pode-se escolher a máquina utilizando-se diversos parâmetros, como a que estiver com menor carga de processamento, a que possuir disponível certo tipo de dispositivo ou determinados dados. Aplicações que exigem grande processamento de dados e pouca interação com o usuário podem ser melhor escalonadas através do *grid*.
  Além dos recursos de processamento muitas máquinas também possuem seus discos rígidos sendo subutilizados. Assim, o *grid* pode ser utilizado como um *DataGrid* alocando o espaço disponível como se fosse um disco apenas. Outra forma de alocar o espaço seria dividir os dados, de forma que as aplicações possam ser executadas em uma máquina mais próxima de onde se encontram os dados em processamento, ou para garantir uma maior disponibilidade caso alguma máquina falhe.
  Diversos outros recursos podem ser compartilhados em um *grid*. Para uma aplicação que demande um maior acesso à Internet, pode-se dividir o trabalho entre outras máquinas que também possuam acesso à rede, acelerando os resultados.
  Outros exemplos podem abranger uma impressora remota com maior qualidade, um gravador de DVD ou equipamentos médicos e científicos avançados como um microscópio eletrônico ou um robô. Com isso, os recursos de uma instituição ou empresa podem ser melhor utilizados, diminuindo despesas e aumentando a eficiência e a competitividade;
- Realizar processamento paralelo: outra característica interessante é a possibilidade de melhor utilizar o processamento paralelo através de *grid*. Em alguns tipos de aplicações como nas científicas, financeiras,

processamento de imagens e simulações, a utilização de processamento paralelo pode agilizar e muito o trabalho.

Uma aplicação escrita utilizando-se de algoritmos e técnicas de programação paralela pode ser dividida em partes menores e estas podem ser separadas e processadas independentemente. Cada uma destas partes de código pode ser executada em uma máquina distinta no *grid*, melhorando o desempenho.

Existem algumas barreiras que podem impedir que uma aplicação utilize todo este potencial. Se a aplicação pode ser dividida em um número fixo de partes independentes, isso se torna uma barreira que impede sua escalabilidade. Outra forma de problema encontrado é quando as partes não podem ser completamente independentes e precisam comunicar-se entre si. Essa situação causa uma possível espera para que as comunicações sejam sincronizadas ou para que o tempo necessário para as mensagens sejam transmitidas.

Além disso, as partes podem precisar acessar uma base de dados ou outro tipo de recurso. Essas características devem ser levadas em conta no momento de se utilizar à funcionalidade de processamento paralelo, mas isso não impede a grande utilização do *grid* como uma excelente arquitetura para o processamento paralelo;

- <u>Potencializar a utilização de dispositivos e organizações virtuais</u>: a colaboração entre os mais diversos tipos de usuários e aplicações é outra capacidade que pode ser desenvolvida com o advento do *grid*. Recursos e máquinas podem ser agrupados para trabalharem juntos, formando assim uma Organização Virtual (VO).

  Uma organização virtual é uma entidade que compartilha recursos sob uma determinada política em uma configuração de *grid*. Exemplos de VO são: empresas, centros de pesquisas e universidades que provêm facilidades de armazenamento de dados, poder de processamento e o uso de equipamentos (ex.: telescópios) e aplicações (ex.: pacotes de software de simulação que podem executar com dados fornecidos pelo próprio usuário);

- <u>Disponibilizar confiança</u>: existem diversas maneiras de aumentar a confiabilidade em um sistema computacional, como por exemplo: processadores e discos duplicados (caso um falhe o outro assuma seu lugar),

fontes de energia e circuitos redundantes, geradores elétricos, entre outros. Todas estas formas comprovadamente aumentam a disponibilidade e confiança em um sistema, mas seus altos custos podem torná-las impraticáveis.

Utilizando-se uma abordagem baseada em *grid* (com máquinas espalhadas em diversos lugares diferentes), quando uma máquina falha atinge uma parte do *grid*, as demais podem continuar sua operação normalmente. Sistemas de gerenciamento podem executar novamente processos importantes caso seja detectada alguma falha. Esses sistemas podem ser executados redundantemente para garantir sua consistência. Dados podem ser duplicados ou separados em diversas partes através do *grid*, aumentando sua disponibilidade. Um grande avanço nessa área são sistemas que podem automaticamente detectar uma falha e tomar as medidas necessárias para contornar o problema.

Grid computacional permite as organizações geograficamente dispersas funcionarem como organizações virtuais (VO) que partilham recursos computacionais de armazenamento e aplicações, assim como também partilham o aproveitamento de recursos computacionais usuais e/ou pouco utilizados na resolução de problemas computacionais que requerem alto desempenho. Deste modo, revolucionam a forma de trabalhar em ciência e tecnologia e em diversas áreas. Em uma era em que o trabalho científico de qualquer domínio do conhecimento produz cada vez mais informação, tais como: a decodificação do genoma humano, a cura de doenças terminais, ou o estudo do comportamento da matéria e da energia, em experiências com aceleradores de partículas cujos dados são analisados e partilhados por investigadores dispersos por todo o mundo (INGRID, 2006; GHANEM, SALEH, ALI, 2010; NUKARAPU et al, 2011).

A computação em *grid* vem oferecer respostas às enormes exigências relacionadas à capacidade computacional e ao armazenamento que o processamento de grandes quantidades de dados coloca. Também abre novas perspectivas à criação de novos produtos e serviços, afetando a forma como as empresas podem conduzir os seus negócios (INGRID, 2006; NUKARAPU et al, 2011).

Grandes empresas financeiras recorrem a *grid* computacional para efetuar análises de risco cada vez mais detalhadas, que por sua vez, levam à criação de novos serviços financeiros para os clientes. Entre as empresas pioneiras na adoção de sistemas de computação em *grid* encontram-se vários bancos (Bank of America, Bank of Montreal, BNP Paribas, Bowne, Citigroup, Genworth, HSBC, JPMorgan Chase, Markit, MassMutual, Nationwide, Royal Bank of Scotland, Societe Generale, TD Bank Financial Group, UBS, Wachovia and WestLB) e relatórios estimam uma utilização crescente de sistemas de *grid* pelas empresas do setor financeiro (INGRID, 2006).

Outro exemplo é o das empresas farmacêuticas a quem esta tecnologia facilita o acesso a uma capacidade computacional cada vez maior. Essa facilidade pode permitir a descoberta de novos compostos e analisar, cada vez com mais detalhes, os riscos associados aos medicamentos antes destes serem comercializados (INGRID, 2006).

2.5. Simulação

A simulação é uma ferramenta adequada para analisar algoritmos de sistemas distribuídos em larga escala de recursos heterogêneos. Ao contrário de usar sistema real no tempo real, a simulação evita a sobrecarga de coordenação de recursos reais. A simulação é igualmente eficaz no trabalho com grandes problemas hipotéticos, que exigem envolvimento de grande número de usuários ativos e de recursos (GridSim, 2002).

A motivação para a utilização de simulação ao invés de utilizar diretamente uma plataforma de teste real de *grid* (especialmente nos modelos de análise e algoritmos em fases iniciais), pode ser traçada a partir dos seguintes fatores (MURSHED, BUYYA, 2002; BUYYA, SULISTIO, 2008):

- A criação de uma plataforma de teste real de *grid* é demorada, de custo elevado, depende de recursos intensivos e é limitada na maior parte a alguns ambientes de área local;
- Uma plataforma de teste real não fornece um ambiente controlável e de repetição para experimentação e avaliação de estratégias de escalonamento;

- A simulação permite analisar tanto os atuais, como os novos modelos econômicos e algoritmos de escalonamento;
- A análise de novos modelos e algoritmos requer um grande número de testes dependentes de recursos disponíveis, que os custos dos testes dificultariam o desenvolvimento de projetos de novos modelos e algoritmos. Usar a simulação ao invés de ambientes de testes reais, certamente eliminaria uma grande parte dos custos.

A simulação tem sido pesquisada e aplicada com sucesso para modelar processos, aplicações e objetos do mundo real. Ela permite o estudo de vários assuntos, tais como: a viabilidade, o comportamento e o desempenho, sem construir o sistema real, poupando assim um tempo precioso, custo e esforço. Uma simulação pode ser ajustada de acordo com vários cenários possíveis. Os resultados obtidos a partir da simulação indicam como o sistema real se comporta, permitindo, assim, que os pesquisadores compreendam e melhorem seus projetos sem a implementação verdadeira (SULISTIO, YEO, BUYYA, 2004; BUYYA, SULISTIO, 2008).

Em detrimento da segurança, os simuladores de *grid* preocupam-se especialmente com os aspectos de escalonamento. Na maioria das vezes, eles implementam métodos que permitem imitar a distribuição de processos segundo estratégias, como FIFO (*First In First Out*) e RR (*Round-Robin*), além de permitir que os usuários construam seus próprios algoritmos de escalonamento. Como os aspectos de segurança não são tratados nos simuladores de *grid*, compete ao usuário incorporar novos métodos que traduzam as características e comportamentos desejados para a plataforma de simulação de sua preferência. No entanto, é preciso certificar-se que o simulador atenda aos requisitos e que suas limitações não impactem no trabalho a ser desenvolvido (MARTINS et al., 2006).

2.6. Ferramentas de Simulação em *Grid* Computacional

Diversas ferramentas de simulação foram desenvolvidas com o propósito de estudar estratégias de escalonamento, como Bricks, GridSim, MicroGrid, OptorSim, SimGrid.

As seções a seguir apresentam a descrição das ferramentas de simulação. O SimGrid é apresentado com mais detalhes por ser a ferramenta utilizada neste trabalho.

2.6.1. Bricks

É uma ferramenta de avaliação de desempenho que permite a análise e comparação de diferentes estratégias de escalonamento de tarefas em sistemas computacionais distribuídos de alto desempenho como *grids*. Normalmente é utilizada para simulação de algoritmos de escalonamento, de topologia de sistemas cliente-servidor e de estratégias de processamento para redes e servidores.

As especificações da simulação são feitas através de arquivos de configurações escrito em linguagem definida pela ferramenta. A modelagem da plataforma do *grid* é realizada de forma indireta pela configuração e especificação das tarefas, os atributos e suas quantidades são indicados para cada componente da arquitetura do *grid*. A topologia de rede é modelada pela especificação da interconexão dos nós. A modelagem das tarefas é feita pelos usuários do *grid*, através da definição da taxa de submissão das tarefas e do tamanho computacional das mesmas. Esse sistema permite a incorporação de ferramentas ou informações externas através de ferramentas de monitoração, benchmarking e predição (TAKEFUSA et al, 1999; AIDA et al, 2000).

Oliveira (2007) também elucidou outras características, como:

<u>Funcionalidades</u>:

- Voltado ao estudo de algoritmos de escalonamento de tarefas;
- Permite a integração de ferramentas externas ao simulador;

- Estabelece um modelo de *grid* que facilita a análise das interações cliente-servidor.

Pontos Fortes:

- Facilidade de uso;
- Extensibilidade da API;
- Flexibilidade na integração com ferramentas externas;
- Portabilidade.

Deficiências:

- Interface pouco amigável;
- Formatação dos resultados dificulta a análise de saída;
- Modelagem da plataforma do *grid* limitada;
- Falta de suporte e desenvolvimento do simulador.

2.6.2. GridSim

É uma ferramenta para modelagem e simulação de escalonamento e gerenciamento de recursos distribuídos para *grid* computacional. As especificações da simulação são feitas através da codificação do programa de simulação em Java. A modelagem da plataforma do *grid* é efetuada através da especificação dos componentes básicos da arquitetura do *grid*, apresentando a interação entre os recursos criados, os usuários que gerarão as tarefas, a submissão das tarefas nos recursos e a estratégia de escalonamento utilizada. Para a modelagem das tarefas, utiliza-se uma classe própria, chamada *Gridlet*. As informações sobre o tamanho computacional, operações de E/S, tamanho dos dados de entrada/saída e da origem da tarefa podem ser gerados aleatoriamente pela classe *GridsimRandom*. O GridSim não permite a incorporação de ferramentas ou informações externas (GridSim, 2002; BUYYA, 2002; BUYYA, SULISTIO, 2008).

Oliveira (2007) também elucidou outras características, como:

Funcionalidades:

- Possibilita o estudo de diversos aspectos relacionados a um *grid* ou sistemas paralelos como cluster;
- Permite a modelagem de aplicações baseadas em diferentes modelos de paralelismo;
- Possui extensões que facilitam o estudo de *grid* de dados e aspectos específicos da rede do *grid*, como provisão de QoS.

Pontos Fortes:

- Grande flexibilidade;
- Permite uma modelagem detalhada dos componentes do sistema;
- Escalabilidade;
- Portabilidade.

Deficiências:

- Ausência de interface amigável;
- Necessidade de codificação do programa de simulação;
- Modelagens da plataforma do *grid* e das tarefas devem ser feitas no próprio programa de simulação.

2.6.3. MicroGrid

É uma ferramenta que implementa uma infraestutura de *grid* virtual para executar aplicações baseada no Globus. O MicroGrid emula um ambiente do *Grid* Globus para gerenciamento de recursos, por isso é considerado um emulador. Um emulador é uma ferramenta que funciona como um sistema real. A emulação executa

como se fosse o próprio sistema real, é útil para testes precisos e de confiança quando não se tem um sistema real. Um simulador é uma ferramenta que pode representar e modelar um sistema real (SONG, et. al., 2000; SULISTIO, YEO, BUYYA, 2004).

No MicroGrid, a modelagem da plataforma do *grid* é feita através do mapeamento entre recursos reais e virtuais. Por esse mapeamento, é feita a modelagem da topologia de rede, indicando os links e os seus atributos, as sub-redes e os roteamentos entre elas. Na modelagem das tarefas é efetuada a identificação do *host* virtual que será submetido cada aplicação, assim como seus parâmetros de entrada e os horários de submissão (HUAXIA et al, 2004).

Oliveira (2007) também elucidou outras características, como:

Funcionalidades:

- Permite a criação de um ambiente computacional distribuído virtual, para o estudo da escalabilidade, resposta a falhas e outros comportamentos;
- Viabiliza o estudo de aplicações reais em um ambiente virtual.
- Permite um alto grau de detalhamento dos recursos da rede.

Pontos Fortes:

- Maior grau de realismo;
- A virtualização dos recursos e serviços é feita de forma transparente ao usuário;
- Possibilidade de avaliação de aplicações reais.

Deficiências:

- Pouco escalável;
- Necessidade de uso de diversos recursos reais;
- Não portável.

2.6.4. OptorSim

É uma ferramenta de simulação de *grid*, projetada para testar estratégias de replicação dinâmica usadas na otimização de localização de dados dentro de um *grid*. Deste modo, é um simulador voltado para *grid* de dados, no qual as transferências de dados estabelecem uma importante limitação do desempenho das tarefas executadas. As especificações da simulação são feitas através de arquivos de configurações escritos em linguagem definida pela ferramenta, com a apresentação da política de escalonamento adotada, o tipo de usuário do *grid*, a estratégia de otimização de réplicas, e a existência ou não de tráfego de fundo. Na plataforma do *grid,* cada site é modelado em função dos seus elementos de computação e de armazenamento, e uma matriz é utilizada para representar a conectividade e largura de banda entre os sites. Possibilita a incorporação de ferramentas ou informações externas através da utilização de *traces* que são modelados a partir de ferramentas de monitoração (CAMERON, 2002; OMII-UK, 2006; BELALEM, SLIMANI, 2007; DataGRID, 2010).

Oliveira (2007) também elucidou outras características, como:

<u>Funcionalidades</u>:

- Voltado ao estudo de algoritmos de escalonamento de tarefas que empregam o conceito de replicação;
- Adota um modelo de *grid* que facilita a avaliação de *grid* de dados;
- Provê interface gráfica para facilitar a construção de simulações e sua análise;

<u>Pontos Fortes:</u>

- Facilidade de uso;
- Utilidades para a análise dos resultados;
- GUI bem projetada;
- Modelagem de tráfego de fundo;

- Portabilidade.

Deficiências:

- Aplicação restrita;
- Pouca flexibilidade na modelagem da plataforma do *grid*;
- Modelagem da rede dentro dos sites é limitada;
- Modelagem do padrão de submissão das tarefas é limitada.

2.6.5. SimGrid

É uma ferramenta de simulação de aplicações distribuídas em ambientes distribuídos heterogêneos para redes simples, desde estações de trabalho a *grid* computacional. As especificações da simulação são feitas pela codificação em C e pela configuração de arquivos XML (*eXtensible Markup Language*). A modelagem da plataforma do *grid* pode ser efetuada no programa em C ou na configuração de um arquivo XML, por exemplo: platform.xml, que contém a especificação dos recursos de computação, os links de comunicação e o roteamento entre os nós. De forma semelhante, a modelagem das tarefas pode ser efetuada no programa em C ou na configuração de um arquivo XML, por exemplo: deployment.xml, com a especificação da quantidade e atributos das tarefas. O SimGrid permite a incorporação de ferramentas ou informações externas através da utilização de *traces* que são modelados a partir de ferramentas de monitoração (CASANOVA, LEGRAND, QUINSON, 2008; SimGrid, 2010).

O SimGrid surgiu de um projeto comum entre a universidade de Hawai em Manoa, LIG Laboratory (Grenoble, França) e universidade de Nancy (França), com o objetivo de prover funcionalidades para a simulação de aplicações distribuídas em ambientes distribuídos heterogêneos (SimGrid, 2010). Proporciona outras características, como:

- Interface de usuário de alto nível para pesquisadores de computação distribuída desenvolverem simulações tanto em C quanto em Java

(introduzido na versão 3.3 ao MSG API (jMSG)) (CASANOVA, LEGRAND, QUINSON, 2008);

- Mecanismo de simulação escalável e extensível que executa vários modelos de simulação válidos e que torna possível simular topologias de redes arbitrárias, computação dinâmica e disponibilidades de recursos de rede, bem como falhas de recursos;

- APIs para desenvolvedores de computação distribuída desenvolverem aplicações distribuídas que podem simultaneamente executar no modo de simulação ou no modo real.

O SimGrid é constituído de três camadas: *Programmation environments layer*, *Simulation kernel layer* e *Base layer*, como apresentado na Figura 2.

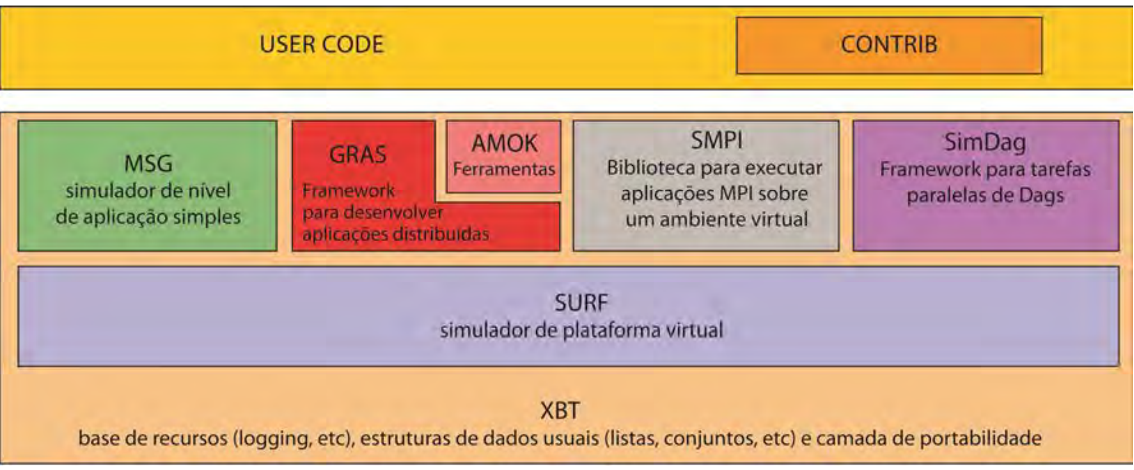

Figura 2 – Visão geral das camadas do SimGrid (SimGrid, 2010).

A camada dos ambientes de programação (*Programmation environments layer*) fornece diversos ambientes de programação construído sobre um único *kernel* de simulação. Cada ambiente objetiva um alvo específico e constituí um paradigma diferente. Os componentes dessa camada são:

- MSG (*Simple programming environment*): Foi o primeiro ambiente de programação disponibilizado e é o de uso mais difundido. Usado para modelar aplicações como processos sequenciais concorrentes (*Concurrent*

Sequential Processes), é útil para modelar problemas teóricos e para comparar diferentes heurísticas.

- SMPI (*Simulated* MPI): Para simulações de códigos MPI. Simulação do comportamento de uma aplicação MPI usando técnicas de emulação.

- GRAS (*Grid Reality And Simulation*): Possibilita a execução de aplicações reais para estudos e testes.

- AMOK (*Advanced Metacomputing Overlat Kit*): Conjunto de ferramentas construída sobre o GRAS, que implementa em alto nível diversos serviços necessários a várias aplicações distribuídas.

- SimDag: Ambiente dedicado à simulação de aplicações paralelas, por meio do modelo DAG (*Direct Acyclic Graphs*). Com este modelo é possível especificar relações de dependência entre tarefas de um programa paralelo.

A camada de simulação do *kernel* (*Simulation kernel layer*) possui o módulo chamado SURF que fornece todas as funcionalidades essenciais para simular uma plataforma virtual. É utilizada como base para a camada de nível superior, e não se destina a usuários finais por ser considerada de muito baixo nível. Uma das principais características do SURF é a capacidade de mudar de forma transparente o modelo utilizado para descrever a plataforma, facilitando muito a comparação dos vários modelos existentes na literatura.

A camada de base (*Base layer*) é constituída pelo XBT (*eXtended Bundle of Tools*). O XBT é uma biblioteca portável que fornece alguns recursos como suporte de registro, suporte de exceção e de configuração. O XBT também abrange as estruturas de dados convenientes: Dynar: *generic Dynamic array*; Fifo: *generic workqueue*; Dict: *generic dictionary*; Heap: *generic heap data structure*; Set: *generic set datatype*; Swag: O(1) *set datatype*.

No SimGrid o poder computacional é definido pelo número de unidades de trabalho por unidade de tempo. Não existe nenhuma distinção entre transferência de dados e computação, ambos são vistos como tarefas. É o usuário que tem a

responsabilidade de garantir que as tarefas computacionais sejam escalonadas em processadores e as transferências de dados para conexões de rede. Todas as tarefas computacionais são *CPU-bound*, ou seja, tarefas de processamento intensivo. A transferência de dados são *bandwidth-bound*, portanto diferentes tarefas computacionais e transferência de dados devem ser implementadas no próprio algoritmo ou de alguma outra forma, porém separadas do simulador.

A implementação das políticas de escalonamento é feita através da programação com a utilização da API em C fornecida pelo simulador. A API permite manipular tipos de dados para recursos e para tarefas. Um recurso é descrito pelo nome, um conjunto de métricas relacionadas a desempenho, traços e constantes. Uma tarefa é descrita pelo nome, custo e estado. Além de funções básicas como criação, inspeção e destruição, são fornecidas funções que descrevem possíveis dependências entre tarefas e funções, para designar tarefas para os recursos (SimGrid, 2010).

Oliveira (2007) elucidou outras características, como:

Funcionalidades:

- Facilita o estudo de estratégias de escalonamento de tarefas;
- Provê componente que auxilia o desenvolvimento de aplicações reais para *grid* (GRAS);
- Provê componente que auxilia o desenvolvimento de programas MPI (SMPI);
- Provê componente para a avaliação de tarefas paralelas em sistemas distribuídos (SIMDAG).

Pontos Fortes:

- Flexibilidade;
- Facilidade de depuração;
- Reutilização de código;

- Modelagem de tráfego e computação de fundo (é o tráfego não proveniente das aplicações do *grid*, que conferem um tratamento mais realista ao ambiente computacional do *grid*);

Deficiências:

- Dificuldade de uso;
- Interface pouco amigável;
- Ausência de funcionalidades que auxiliem na modelagem da plataforma;
- Pouco escalável.

2.6.6. Comparação entre as ferramentas de simulação

As principais características das ferramentas são apresentadas na Tabela 1 e para uma melhor compreensão das características: ambiente de projeto e simulação, elas são detalhadas em seguida.

Tabela 1 – Principais características das ferramentas de acordo com os estudos apresentados por Sulistio, Yeo, Buyya (2004), Quetier, Cappello (2005); Martins et al (2006), DataGRID (2010), SimGrid (2010).

	Brincks	**GridSim**	**MicroGrid**	**OptorSim**	**SimGrid**
Tipo de ferramenta	Simulador	Simulador	Emulador	Simulador	Simulador
Linguagem de implementação	Java	Java	C	Java	C / Java
Plataformas suportadas	Qualquer S.O. com suporte a JVM	Qualquer S.O. com suporte a JVM	Linux/Alpha em um *Grid* Globus	Qualquer S.O. com suporte a JVM	Linux, MacOs, Windows
Interface	Modo texto	Modo texto e GUI (limitada)	Modo texto	Modo texto e GUI	Modo texto
Framework de programação	Orientada a objeto	Orientada a objeto	Estruturada	Orientada a objeto	Estruturada
Ambiente de projeto	Linguagem	Biblioteca	Linguagem	Biblioteca	Biblioteca
Simulação	Estática, discreta, determinística	Estática, discreta, determinística	Dinâmica, contínua, determinística	Dinâmica	Estática, discreta, determinística
Mecanismo de simulação	Sequencial	Sequencial	Paralelo	Multitarefa	Sequencial
Arquitetura do *grid*	Definida pela ferramenta	Definida parcialmente pelo usuário	Definida pelo usuário de acordo com o Globus *Grid*	Definida pela ferramenta	Definida pelo usuário
Licença	Opensource	Opensource	Opensource	Opensource	Opensource

A característica ambiente de projeto: determina como o usuário utiliza a ferramenta para desenvolver os modelos de simulação (SULISTIO, YEO, BUYYA, 2004).

Uma ferramenta de simulação baseada em linguagem fornece um conjunto de construtores definidos para o usuário desenvolver os modelos de simulação. Já uma ferramenta que utiliza biblioteca proporciona um conjunto de rotinas para ser usado como uma linguagem de programação de suporte.

Uma ferramenta de simulação baseada em biblioteca normalmente dá ao usuário maior flexibilidade na criação e no controle da simulação. Um usuário experiente em programação pode aperfeiçoar e otimizar a simulação através da exploração de algumas bibliotecas. A ferramenta de simulação baseada em linguagem, geralmente oculta do usuário detalhes de implementação de baixo nível e, portanto, oferece menos flexibilidade. No entanto, por ser de nível mais elevado se comparado a uma ferramenta baseada em bibliotecas, é mais fácil de aprender e de usar.

Em relação ao tipo de simulação uma ferramenta pode ser classificada em três propriedades: presença do tempo, base de valores e comportamento (SULISTIO, YEO, BUYYA, 2004).

- Presença do tempo: indica se a simulação de um sistema engloba o fator tempo. É dividida em simulação estática e dinâmica. A simulação estática não tem tempo como parte da simulação, em contraste com uma simulação dinâmica.
- Base de valor: especifica os valores que as entidades simuladas podem conter. É dividida em simulação discreta e contínua. A simulação discreta tem entidades que possuem apenas um dos muitos valores dentro de um intervalo finito, enquanto que uma simulação contínua tem entidades que possuem um dos muitos valores dentro de um intervalo infinito.
- Comportamento: define como são os recursos da simulação. É dividida em simulação determinística e probabilística. A simulação determinística, não tem eventos aleatórios ocorrendo, desta forma repetindo a mesma simulação sempre retornará os mesmos resultados. Em contraste, uma simulação probabilística tem eventos aleatórios ocorrendo, então repetindo a mesma simulação, muitas vezes retorna diferentes resultados.

2.7. Considerações Finais

Este capítulo apresentou os fundamentos básicos de *grid* computacional como forma de prover o compartilhamento geograficamente distribuído de serviços e recursos, evidenciando suas características e arquitetura, com a intenção de distinguir sua definição, que constantemente é confundida com outras tecnologias. Foram destacadas as vantagens para a utilização de simulação ao invés da utilização de plataformas reais de *grid*, especialmente nos novos modelos de análise e de algoritmos. Também foram apresentadas as descrições das ferramentas de simulação para *grid* computacional, sendo elas: Brincks, GridSim, MicroGrid, OptorSim, SimGrid. Ao final do capítulo, suas principais características foram elucidadas em uma tabela comparativa.

3. ESCALONAMENTO DE TAREFAS EM *GRID* COMPUTACIONAL

3.1. Considerações Iniciais

Escalonamento de tarefas em *grid* é a atividade de escalonar um conjunto de aplicações originadas de usuários diferentes para um conjunto de recursos computacionais espalhados por locais distintos. Essa atividade tem o intuito de maximizar a utilização do sistema (HE, SUN, LASZEWSKI, 2002; HONG JIANG, TIANWEI NI, 2009).

Este capítulo apresenta a taxonomia hierárquica para algoritmos de escalonamento em sistemas computacionais paralelos e distribuídos, como também para *grid* computacional. Além disso, apresenta as políticas de escalonamentos de tarefas em *grid*.

3.2. Escalonamento em Sistemas Computacionais Paralelos e Distribuídos

Com o surgimento dos sistemas distribuídos, houve a necessidade de criar novos escalonadores denominados globais. Receberam esse nome, porque, em um ambiente distribuído, existem diversos recursos disponíveis; sendo assim, sua tarefa passa a ser escalonar processos entre um conjunto acoplado de máquinas. Em um sistema convencional, existe apenas um recurso disponível. Desta forma, os escalonadores locais utilizam critérios para selecionar qual processo será escolhido para fazer uso do processador (MAHESWARAN et al., 1999).

Um escalonamento é feito objetivando diversas metas de desempenho, por isso os escalonadores globais agregaram a função de escolher quando e quais processos têm acesso a determinados recursos do sistema (JAIN, 1991; MAHESWARAN et al., 1999;

SILBERSCHATZ et al., 2001; TANENBAUM, 2010). Entre as metas existentes as principais são:

- Aumentar o *throughput* do sistema: também chamado de vazão do sistema, é a medida feita a partir do número de processos finalizados por unidade de tempo.

- Diminuir o tempo de resposta: tal medida é definida pela diferença entre o momento de término da execução da tarefa e seu instante de chegada na fila de processos, ou seja, essa medida é a soma dos tempos gastos em fila de espera por recursos e na execução propriamente dita dos processos.

- Aumentar a utilização de recursos: o escalonador pode fazer com que os recursos do sistema – tais como: CPU, memória ou rede – sejam utilizados ao máximo, mesmo que para atingir tal meta seja necessário esquecer outros critérios.

- Balancear a carga do sistema: consiste em não subutilizar recursos enquanto outros estão trabalhando em sua capacidade máxima. A intenção é distribuir os processos para os recursos de acordo com a capacidade dos mesmos.

Com a intenção de padronizar a utilização dos algoritmos de escalonamento em sistemas distribuídos, Casavant, Kuhl (1988) propuseram uma taxonomia hierárquica, mostrada na Figura 3. Os elementos dessa taxonomia são descritos a seguir:

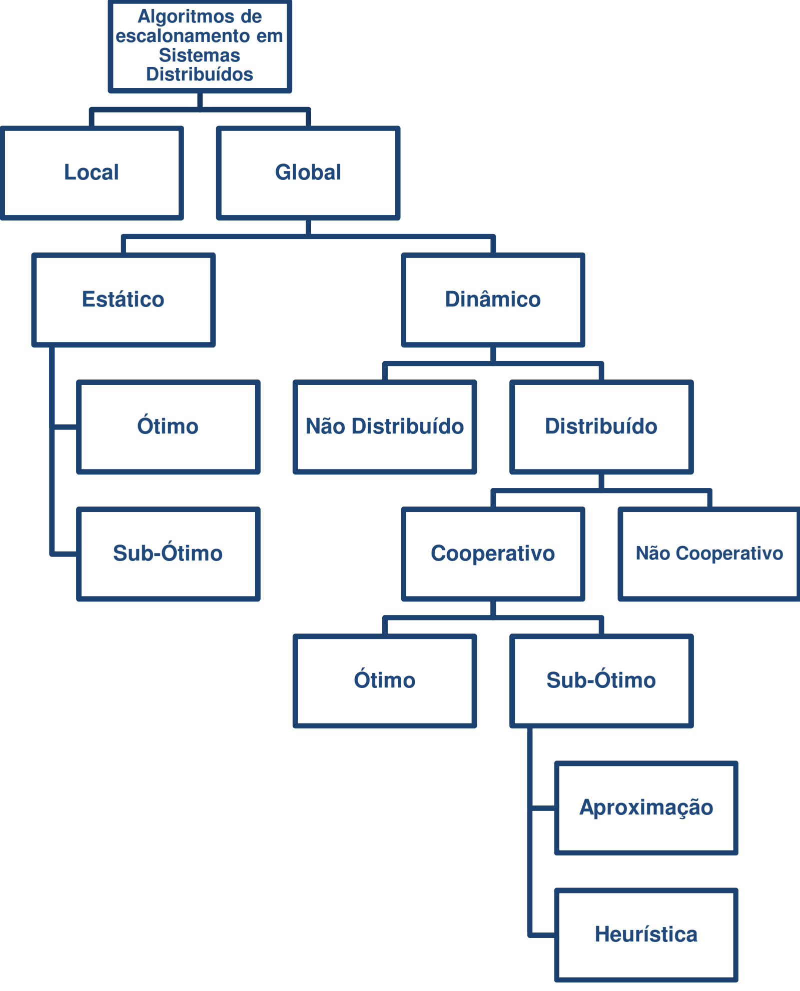

Figura 3 – A taxonomia hierárquica para algoritmos de escalonamento (CASAVANT, KUHL 1988).

- Local e Global: o escalonamento local determina como os processos residentes em um único CPU (Processador) são alocados e executados. Um escalonamento global utiliza-se de informações sobre o sistema para alocar processos para múltiplos processadores e para otimizar um sistema que objetiva mais desempenho. *Grid* computacional utiliza escalonamento global.

- Estático e Dinâmico: no escalonamento estático, as informações são obtidas antes do escalonamento da aplicação, pois não obtém informações sobre as mudanças dinâmicas de estado do sistema que ocorrem durante o processo. No escalonamento dinâmico, a ideia básica é que se ofereça alocação de tarefas durante a execução da aplicação. É de responsabilidade do sistema decidir onde o processo será executado.

O escalonamento dinâmico é usualmente aplicado quando se tem dificuldade em estimar o custo da aplicação e, também quando as tarefas da aplicação estão dinamicamente em tempo real.

- Ótimo e Sub-ótimo: em alguns casos, tem-se um escalonamento ótimo quando toda a informação do estado dos recursos e da aplicação é conhecida. Na maioria das vezes não é possível ter todas as informações, como por exemplo, a dificuldade em estimar com precisão o tempo de execução de uma aplicação; estimar a carga dos recursos selecionados que em muitos casos podem sofrer grandes variações. Por estas dificuldades os algoritmos de escalonamento, geralmente, variam entre resultados sub-ótimos que podem ser obtidos através de Aproximação ou de Heurística.

- Aproximação e Heurística: um algoritmo de aproximação consiste em seguir as mesmas regras de um algoritmo ótimo, com a diferença de que não é necessário encontrar uma solução ótima. Desta forma, uma solução que seja suficientemente boa em relação a que foi previamente definida como ótima é satisfatória.

  Uma heurística consiste em um algoritmo que leva em consideração alguns parâmetros que afetam o sistema de uma maneira indireta. Esses parâmetros, por exemplo, são mais simples de se calcular do que os parâmetros verdadeiros para análise de desempenho. É nesse ponto que a heurística representa boa alternativa para o escalonamento de processos.

- Distribuído e Não-distribuído: nos escalonadores dinâmicos, as tomadas de decisões de um escalonamento global podem ser de um escalonador centralizado ou pode ser compartilhada por múltiplos escalonadores distribuídos. Em *grid* computacional, muitas aplicações podem ser enviadas ou requeridas para serem escalonadas simultaneamente, o que pode inviabilizar o uso de um escalonador centralizado. Uma estratégia centralizada tem a vantagem de ser simples de implementar, mas tem a possibilidade de ser um gargalo no desempenho.

- Cooperativo e Não-Cooperativo: na utilização de um algoritmo de escalonamento distribuído, deve-se considerar se todos os escalonadores

envolvidos com o escalonamento da aplicação estão trabalhando de forma cooperativa ou independente (não-cooperativa). No modo não-cooperativo, os escalonadores individuais atuam sozinhos, como entidades independentes que não se preocupam diretamente com nenhum objetivo que resulte na melhoria do desempenho do resto do sistema. As decisões afetam somente o desempenho local de cada escalonador. No modo cooperativo, cada escalonador responsabiliza-se por sua própria porção do escalonamento da tarefa, contudo, todos os escalonadores estão trabalhando em comum para um amplo sistema global.

3.3. Escalonamento de Tarefas em *Grid* Computacional

Escalonamento em *grid* computacional é o processo de tomar decisões de escalonamento envolvendo recursos sobre múltiplos domínios administrativos. A realização de um escalonamento é feita através da ativação de um conjunto de regras que ditam como e quando determinadas informações do sistema devem ser colhidas, de que maneira essas informações influenciam na distribuição de tarefas e quais serão os recursos utilizados para a execução das aplicações. Assim, os algoritmos de escalonamento são utilizados para implementar as regras de uma política de escalonamento. Enquanto as políticas de escalonamento ditam as regras gerais de como lidar com processos e administrar recursos do sistema, os algoritmos de escalonamento estão preocupados com a implementação dessas regras que podem ser feitas de diversas formas (SOUZA, 2000; SCHOPF, 2002; REIS, 2005; CIRNE, et al, 2007; WEIFENG SUN, 2010).

Segundo Schopf (2002) e Falavinha Jr (2009) o processo de escalonamento em *grid* abrange três fases principais:

1) <u>Descoberta de recursos</u>: na qual uma lista de recursos disponíveis é gerada;

2) <u>Seleção do sistema</u>: coleta de informações dos recursos e seleção do melhor grupo;

3) Execução do trabalho: que inclui exibição do arquivo e coleta dos resultados.

O escalonamento de processos em *grid* computacional é uma grande área de pesquisa pelo desafio que a própria natureza desse sistema representa (BERMAN, 1998; CIRNE, 2002; HONG JIANG, TIANWEI NI, 2009; GHANEM, SALEH, ALI, 2010). As características de *grid* que representam desafios no momento da atribuição das tarefas são:

- Grande quantidade de recursos: a grande quantidade de recursos torna-se um problema para o escalonador, que pode se tornar um gargalo do sistema, pois ele deve escolher de forma apropriada, qual recurso irá executar cada processo.

- Grande heterogeneidade de recursos: máquinas pertencentes ao *grid* podem apresentar configurações heterogêneas. Entre as configurações, as principais são: poder de processamento, interconexões e sistemas operacionais.

- Alto compartilhamento de recursos: a variação de carga nas máquinas causada pela submissão de novos processos ao sistema é proporcional ao número de usuários do *grid*, isto é, quanto mais usuários, maior será a variação de carga do sistema. Isso pode fazer com que políticas de escalonamento que não presumem tal fato atinjam um resultado negativo.

- Movimentação e consistência de dados: em *grid* deve-se evitar a submissão de aplicações que realizem muita comunicação, pois a baixa latência da rede de interconexão dos recursos pode causar prejuízos ao escalonamento.

Aspectos que não são abrangidos pela taxonomia hierárquica e que estão vinculados às novas mudanças no cenário de *grid* computacional são apresentados por Dong e Akl (2006):

- Objetivos Funcionais: os algoritmos de escalonamento também podem ser classificados de acordo com o objetivo funcional do escalonamento e podem ser classificados em duas categorias: centralizado nas aplicações e centralizado nos recursos. Os algoritmos centralizados nas aplicações têm por finalidade otimizar o desempenho de cada aplicação individualmente, enquanto que os centralizados nos recursos visam otimizar o desempenho e a utilização destes.

- Adaptativo e Probabilístico: em um algoritmo adaptativo, os parâmetros utilizados em sua implementação mudam dinamicamente de acordo com o comportamento do sistema, em resposta à política de escalonamento adotada. Um algoritmo probabilístico gera aleatoriamente, a partir do dado inicial, diversas soluções de um problema, e, dentre elas, seleciona a melhor. Neste método, o número de soluções geradas deve ser suficiente para possibilitar que, entre o conjunto apresentado, tenha pelo menos uma solução que se aproxime da solução ótima.

- Dependência entre as Tarefas: quando existe uma dependência entre as tarefas, uma tarefa não pode iniciar antes que termine a outra com a qual tem dependência. Assim, as tarefas seguem uma determinada ordem de execução. As tarefas independentes não possuem relação de dependência e podem ser executadas em qualquer ordem. As aplicações denominadas *Bag-of-Task* são do tipo independente. Este tipo de aplicação facilita a utilização em ambientes distribuídos geograficamente dispersos, como em *grid* computacional (IOSUP, EPEMA, 2011).

- Requisitos de QoS: em ambientes distribuídos heterogêneos não dedicados, requisitos de QoS (*Quality of Service*), se configuram como uma grande preocupação de muitas aplicações. QoS pode variar de acordo com a preocupação de cada usuário, que pode ser um requisito de velocidade de CPU, tamanho de memória, largura de banda, versão de software. Normalmente, QoS não é o último objetivo de uma aplicação, mas um conjunto de condições para executar uma aplicação bem sucedida (SUCHANG GUO et al, 2011).

3.4. Políticas de Escalonamento em *Grid* Computacional

Reis (2005) relata em seus estudos que uma das características das políticas de escalonamento é que elas devem ser focadas em um conjunto de aplicações específicas. Uma política deve conhecer os detalhes das aplicações as quais irá escalonar, pois a adoção de uma política genérica pode influenciar de maneira negativa nos resultados das execuções. Muitas vezes é mais vantajoso utilizar políticas de escalonamento simples em vez de uma política altamente eficaz que não atenda ao perfil do processo ao qual se deve supervisionar.

Um tipo de aplicação paralela utilizada em *grid* computacional são as aplicações BoT (*Bag-of-Task*). Essas aplicações são compostas por tarefas independentes, por isso, não é necessário qualquer tipo de comunicação entre as tarefas durante o processamento (a execução das tarefas não depende uma das outras). Desta forma, permite o uso de políticas baseadas em apenas alguns dados do sistema, raramente necessitando de informações sobre a infraestrutura do *grid*, como latência da rede e largura de banda existente entre os recursos (SILVA, 2003; CIRNE et al, 2007; GHANEM, SALEH, ALI, 2010).

As características das aplicações BoT implicam em uma maior simplicidade para escalonar as tarefas, o que permite o uso de políticas tradicionais como *Workqueue* (WQ) e *Round-Robin* (RR) (utilizadas localmente em sistemas operacionais) a serem utilizadas para o escalonamento em *grid* computacional. Essas políticas, apesar de serem simples, representam uma grande base para o desenvolvimento de outras mais robustas e adaptadas com as características do ambiente e das aplicações do *grid* (REIS, 2005).

A seguir serão apresentados os principais algoritmos de escalonamento para tarefas em *grid* computacional, descrevendo suas funcionalidades e metodologias para que fiquem claras as diferenças entre cada um. Vale ressaltar que, cada algoritmo tem suas próprias características e, assim, atuam de maneira diferente em determinadas fases do escalonamento.

3.4.1. *Workqueue*

Workqueue (WQ) é uma política de escalonamento que atende a um *pool* de processos. A escolha de qual processo será submetido para execução é feita de maneira aleatória sempre que um recurso se encontrar disponível. Como na *Round-Robin* (RR), a política WQ também não necessita de informações do sistema ou da aplicação, no entanto, ela apresenta melhor resultado que a RR, pois atribui dinamicamente as tarefas aos processadores que ao passar do tempo se tornam livres. Processadores mais velozes tendem a se tornar disponíveis mais rapidamente, portanto, recebem mais tarefas. Contudo, o problema de desbalanceamento ainda persiste caso uma grande tarefa seja atribuída a um processador lento perto do final da execução da aplicação (SILVA, 2003).

A Figura 4 apresenta um esboço do algoritmo WQ.

```
01.   Enquanto existir tarefa na fila
02.           Verificar se tem host disponível
03.                 Enviar tarefa para o host
04.           Senão, aguardar liberar host para enviar a tarefa
05.   Finalizar aplicação após a execução de todas as tarefas
```

Figura 4 – Esboço do algoritmo WQ.

Como apresentado na Figura 4, nas linhas 01 e 02 é verificada a existência de tarefas na fila e a disponibilidade das máquinas. Enquanto houver tarefas na fila, as mesmas são enviadas para execução nas máquinas disponíveis. Na linha 05 as máquinas que já concluíram a execução de sua tarefa, aguardam a conclusão de todas as tarefas em execução para finalizar a aplicação.

A Figura 5 apresenta um diagrama do algoritmo WQ integrado ao SimGrid.

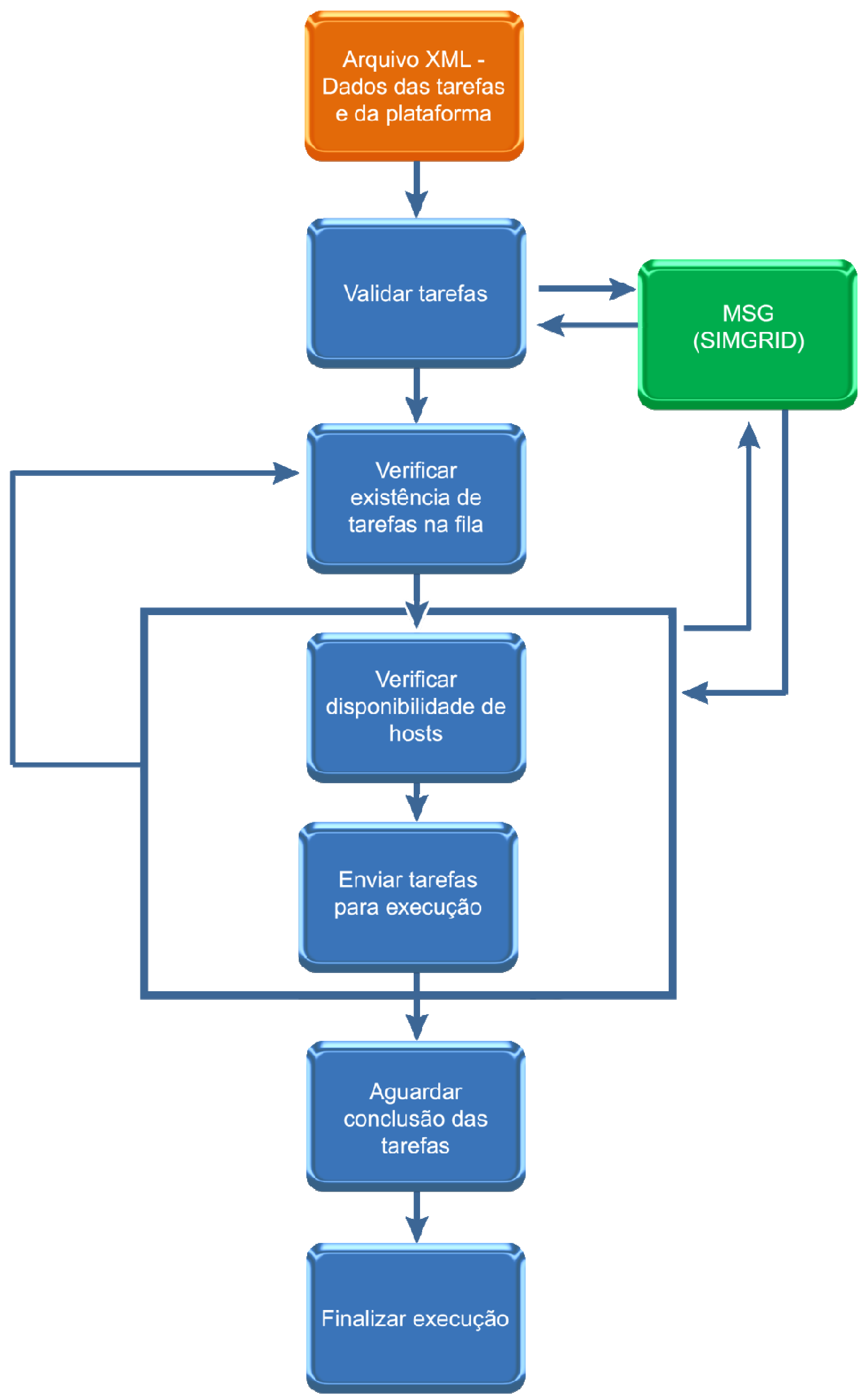

Figura 5 – Diagrama do algoritmo WQ integrado ao SimGrid.

3.4.2. *Workqueue with Replication*

O *Workqueue with Replication* (WQR) foi desenvolvido para solucionar o problema da obtenção de informações sobre a aplicação e a carga de utilização dos recursos do *grid*. Em sua fase inicial, o WQR é similar a um WQ tradicional, as tarefas são enviadas para execução nas máquinas que se encontram disponíveis. Quando uma máquina finaliza a execução de uma tarefa, ela recebe uma nova tarefa para processar. Os algoritmos WQR e WQ passam a diferir no momento em que uma máquina se torna disponível e não há mais nenhuma tarefa pendente para executar. Neste momento, o WQ já terminou seu trabalho e apenas aguarda a finalização de todas as tarefas. Porém, o WQR inicia sua fase de replicação para tarefas que ainda estão em execução, e assim que a tarefa original ou uma de suas réplicas finalizarem, as outras são interrompidas. Vale ressaltar que o WQR assume que as tarefas são idempotentes, isto é, não geram efeitos colaterais, como por exemplo: incrementação de valores, de modo a prevenir a inconsistência de dados que as réplicas poderiam gerar (SILVA, 2003).

A Figura 6 apresenta um esboço do algoritmo WQR.

```
01.   Enquanto existir tarefa na fila
02.         Verificar se tem host disponível
03.               Enviar tarefa para o host disponível
04.         Senão aguardar liberar host para enviar a tarefa
05.         Enviar tarefa para replicação no host disponível
06.         Após finalizar a execução da tarefa original ou de uma de suas réplicas,
            interromper as outras.
07.   Finalizar aplicação após execução de todas as tarefas
```

Figura 6 – Esboço do algoritmo WQR.

Como apresentado na Figura 6, inicialmente das linhas 01 a 04 o funcionamento do WQR é semelhante ao WQ. Na linha 05 começa a replicação das tarefas que continuam em execução para as máquinas disponíveis. As tarefas são replicadas até que um número máximo de réplicas predefinido seja atingido. Quando a tarefa original ou uma das réplicas completarem a execução, as outras são interrompidas. Na linha 07 a aplicação é finalizada após a conclusão de todas as tarefas.

A Figura 7 apresenta um diagrama do algoritmo WQR integrado ao SimGrid.

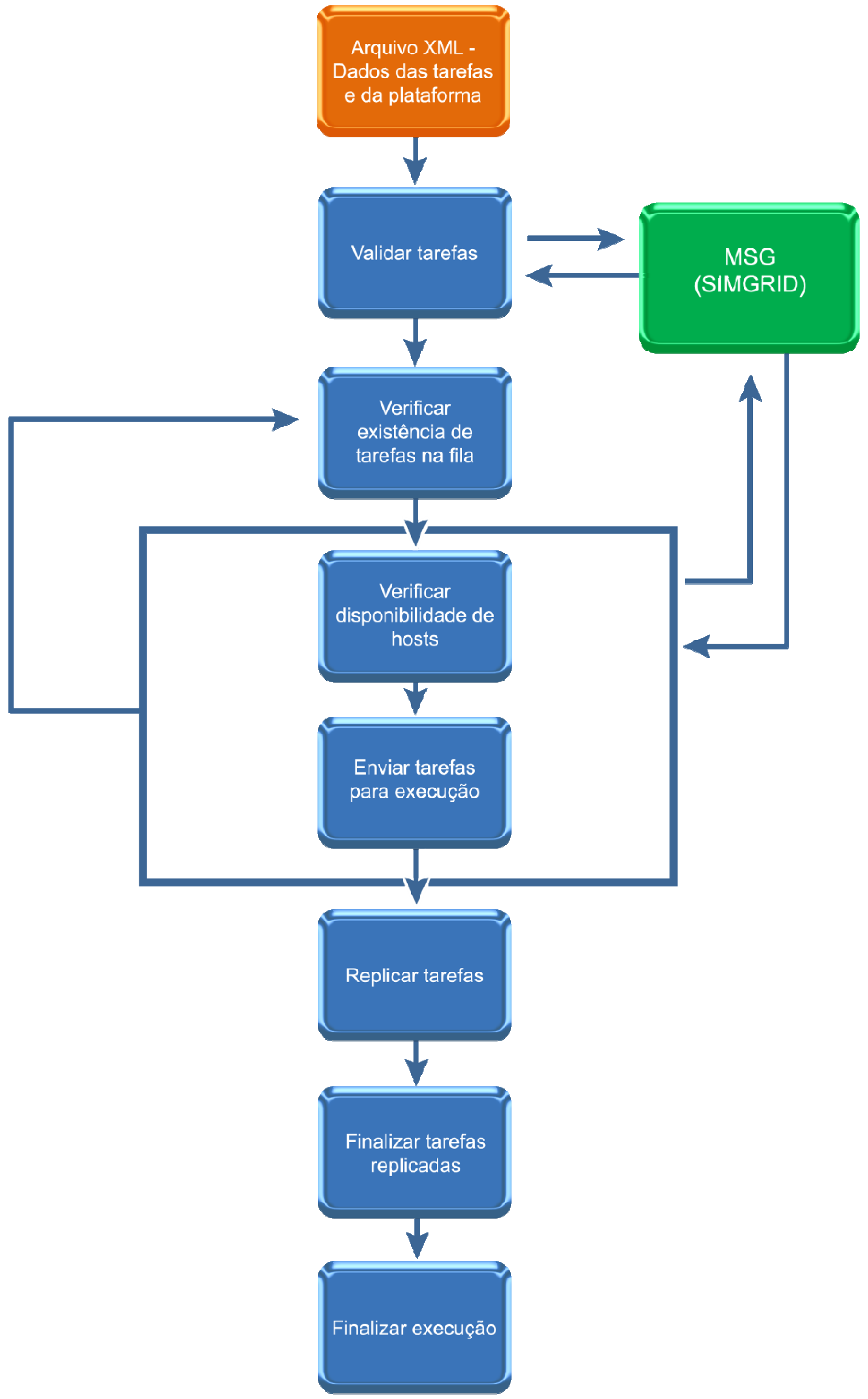

Figura 7 – Diagrama do algoritmo WQR integrado ao SimGrid.

3.4.3. *Sufferage*

A ideia básica do *Sufferage* (MAHESWARAN et al., 1999) é determinar o quanto cada tarefa seria prejudicada se não fosse escalonada no processador que a executaria de forma mais eficiente. Portanto, o *Sufferage* prioriza as tarefas de acordo com o valor que mede o prejuízo de cada tarefa. O valor *Sufferage* de cada tarefa é definido pela diferença entre o melhor e o segundo melhor CT (*Completion Time*), considerando todos os processadores do *grid*. Onde CT = TBA + *Task Cost*. O TBA (*Time to Become Available*) é o tempo para o *host* tornar-se disponível. O *Task Cost* = (*Task size* / *Host speed*) / (1 - *Host load*).

- <u>*Host Speed*</u>: representa a velocidade relativa da máquina. Uma máquina que tem *Host Speed* = 2 executa uma tarefa duas vezes mais rapidamente que uma máquina com *Host Speed* = 1.
- <u>*Host Load*</u>: representa a fração de CPU da máquina que não está disponível para a aplicação (fração de CPU que está sendo usada por outros usuários e aplicações). Deve-se lembrar que *Host Load* varia com o tempo, dependendo da carga que é imposta à máquina por outros usuários e aplicações.
- <u>*Task Size*</u>: é o tempo necessário para uma máquina com *Host Speed* = 1 completar a tarefa quando *Host Load* = 0.

A Figura 8 apresenta um esboço do algoritmo *Sufferage*.

01.	Enquanto existir tarefa na fila
02.	No início cada *host* é marcado como “disponível”
03.	Para cada tarefa da fila calcular o melhor e o segundo melhor CT
04.	Calcular valor de *sufferage* para a tarefa atual
05.	Se o *host* que tem o melhor CT para a tarefa atual estiver disponível, enviar a tarefa atual para o *host* e incrementar TBA do *host* com o CT da tarefa atual.
06.	Senão, se o valor de *sufferage* da tarefa atual for maior que o valor de *sufferage* da tarefa atribuída no *host*, desalocar a tarefa atribuída no *host* e devolver para a fila. Enviar tarefa atual p/ *host*.
07.	Senão, pular para a próxima tarefa
08.	Finalizar aplicação após execução de todas as tarefas

Figura 8 – Esboço do algoritmo *Sufferage*.

Como apresentado na Figura 8, na linha 01 inicia-se a verificação das tarefas na fila, na primeira execução todas as máquinas serão marcadas como disponível (linha 02). Nas linhas 03 e 04 são efetuados os cálculos do melhor e segundo melhor CT e o valor de *Sufferage*. Na linha 05 é feita a verificação para saber se a máquina que tem o melhor CT está disponível para receber a tarefa atual. Se a máquina não estiver disponível, é comparado o valor de *Sufferage* da tarefa que está atribuída à máquina com a tarefa atual. Fica na máquina a tarefa que tiver o maior valor de *Sufferage*, após a atribuição da tarefa o TBA da máquina que recebeu a tarefa é incrementado. Se a tarefa atual tiver o maior valor de *Sufferage*, a tarefa atribuída à máquina é desalocada e devolvida para a fila de tarefas, caso contrário quem volta para a fila é a tarefa atual (linha 06). A tarefa que foi devolvida para a fila só é considerada na próxima iteração (linha 07). A aplicação é finalizada na linha 08 após a conclusão de todas as tarefas.

A Figura 9 apresenta um diagrama do algoritmo *Sufferage* integrado ao SimGrid.

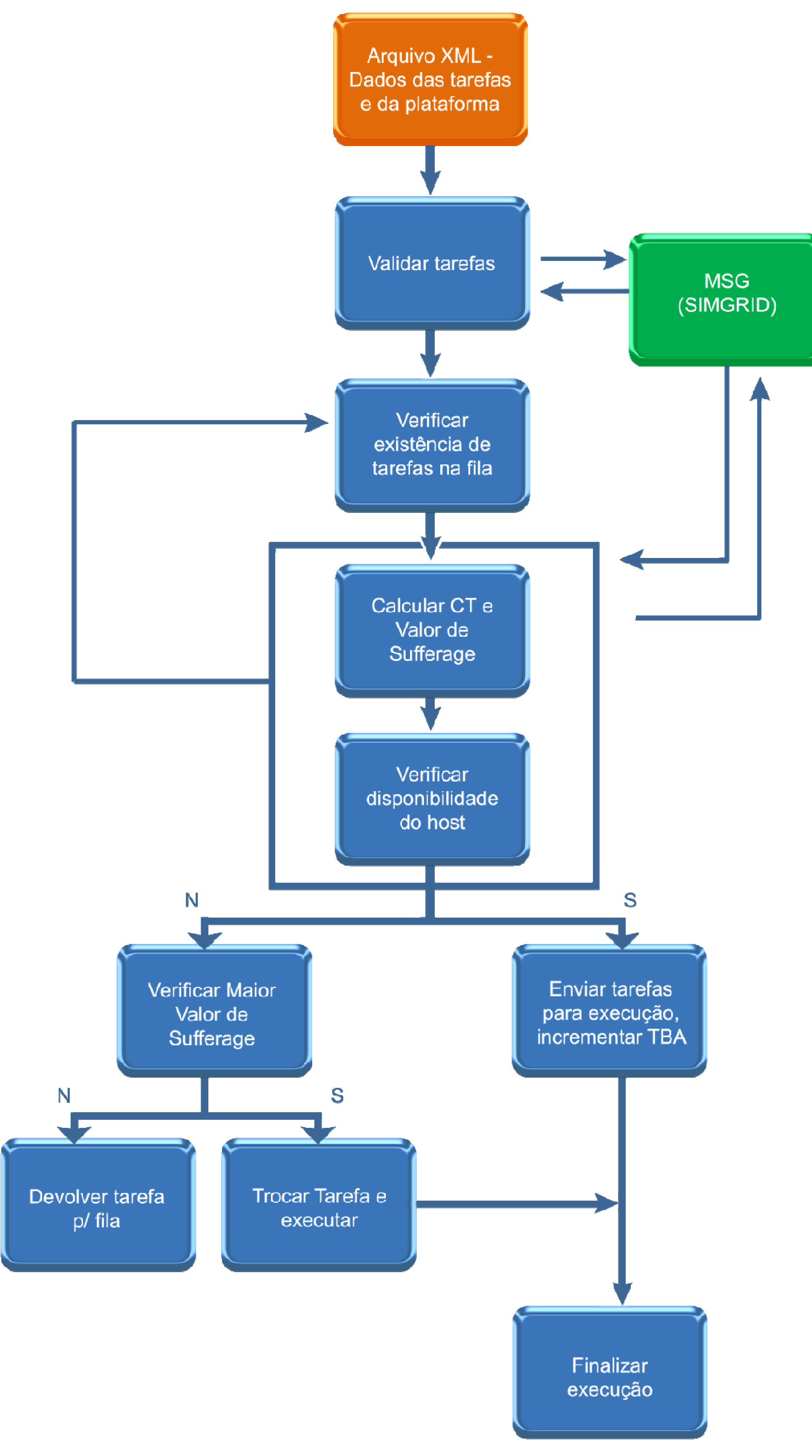

Figura 9 – Diagrama do algoritmo *Sufferage* integrado ao SimGrid.

3.4.4. *XSufferage*

O *XSufferage* (CASANOVA et al, 2000) é uma política de escalonamento baseada nas informações sobre o desempenho dos recursos para associar tarefas aos processadores. Aborda o impacto das grandes transferências de dados em aplicações que usam grandes quantidades de dados. O *XSufferage* é uma extensão modificada do *Sufferage* (MAHESWARAN et al, 1999).

A principal diferença entre o *Sufferage* e *XSufferage* é o método usado para calcular o valor do *sufferage*. O *XSufferage* leva em consideração a transferência dos dados de entrada da tarefa durante o cálculo dos tempos de execução. Isso implica na utilização das informações relacionadas à CPU e o tempo estimado de execução da tarefa usado pelo *Sufferage* mais a largura de banda disponível na rede que conecta os recursos. Um ponto a ser observado é que o *XSufferage* considera somente os recursos livres no momento em que vai escalonar uma tarefa, pois caso contrário sempre o recurso mais rápido e com melhor conexão de rede receberia todas as tarefas.

A Figura 10 apresenta um esboço do algoritmo *XSufferage*.

```
01.  Enquanto existir tarefa na fila
02.       No início cada host é marcado como "disponível"
03.       Para cada tarefa da fila calcular o melhor e o segundo melhor CT
04.       Calcular valor de Sufferage para a tarefa atual
05.       Se o host que tem o melhor CT para a tarefa atual estiver disponível, enviar
          a tarefa para o host e incrementar TBA do host com o CT da tarefa atual
06.       Senão, passar para a próxima tarefa
07.  Finalizar aplicação após execução de todas as tarefas
```

Figura 10 – Esboço do algoritmo *XSufferage*.

Como apresentado na Figura 10, inicialmente, das linhas 01 a 03 o funcionamento do *XSufferage* é semelhante ao *Sufferage*. A principal diferença está no cálculo do valor de *Sufferage*, que é considerado a taxa de transferência de dados (linha 04). Na linha 05 a tarefa é enviada para a máquina que tem o melhor CT se a mesma estiver disponível para recebê-la, e da mesma forma que no *Sufferage* o TBA da máquina que recebeu a tarefa é incrementado. Se a máquina não estiver disponível, a tarefa atual continua na fila e só será considerada na próxima iteração (linha 06). A aplicação é finalizada na linha 07, após a conclusão de todas as tarefas.

A Figura 11 apresenta um diagrama do algoritmo *XSufferage* integrado ao SimGrid.

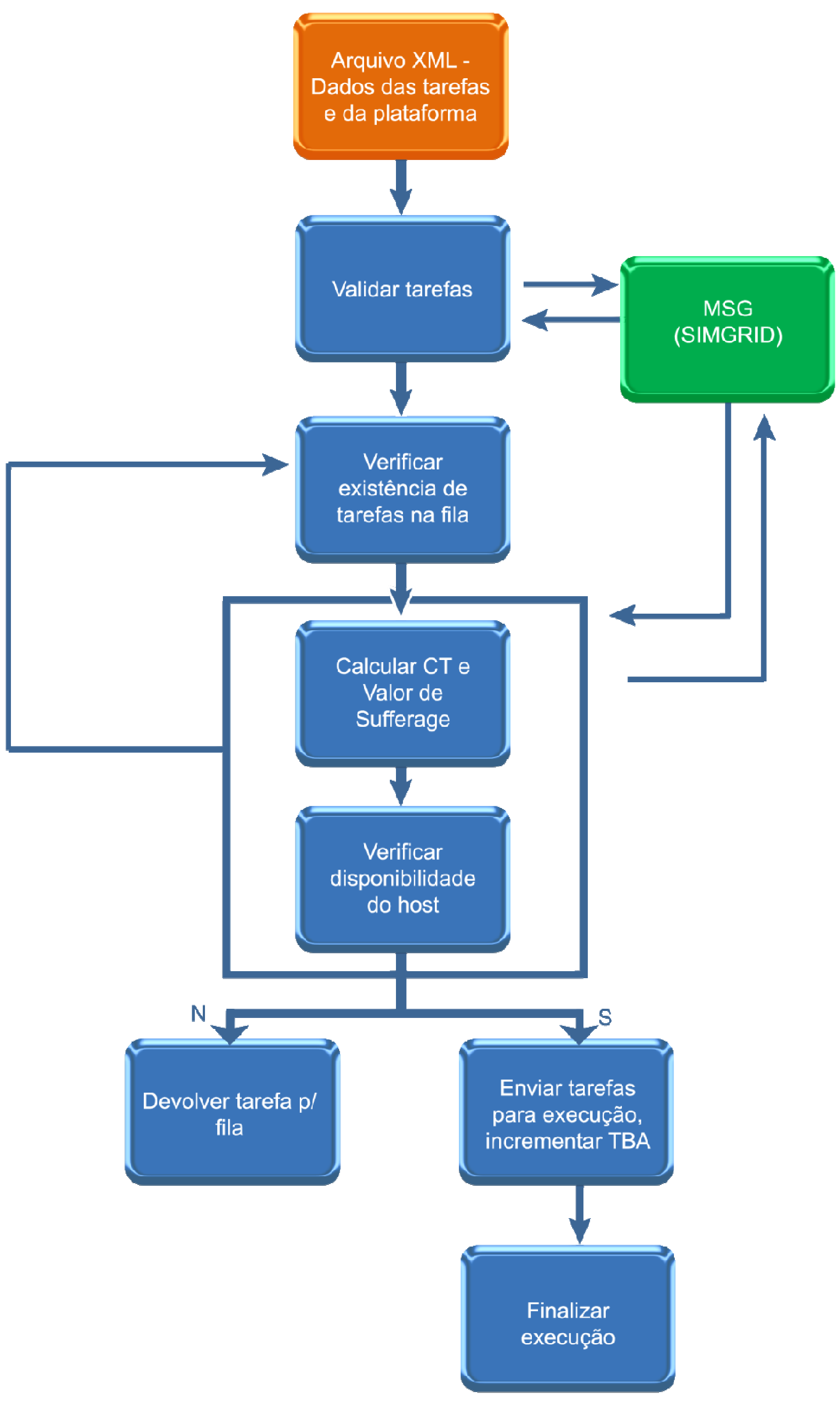

Figura 11 – Diagrama do algoritmo *XSufferage* integrado ao SimGrid.

3.4.5. *Dynamic* FPLTF

O *Dynamic* FPLTF é o resultado da modificação do FPLTF (*Fastest Processor to Largest Task First*) (MENASCÉ et al, 1995) feita por Paranhos, Cirne, Brasileiro (2003). O *Dynamic* FPLTF necessita de três tipos de informações para escalonar as tarefas devidamente: *Task Size, Host Load e Host Speed*.

- *Host Speed*: representa a velocidade relativa da máquina. Uma máquina que tem *Host Speed* = 2 executa uma tarefa duas vezes mais rapidamente que uma máquina com *Host Speed* = 1.
- *Host Load*: representa a fração de CPU da máquina que não está disponível para a aplicação (fração de CPU que está sendo usada por outros usuários e aplicações). Deve-se lembrar que *Host Load* varia com o tempo, dependendo da carga que é imposta à máquina por outros usuários e aplicações.
- *Task Size*: é o tempo necessário para uma máquina com *Host Speed* = 1 completar a tarefa quando *Host Load* = 0.

No início do algoritmo, o tempo para se tornar disponível TBA (*Time to Become Available*) de cada *host* é iniciado com 0 e as tarefas ordenadas por tamanho em ordem decrescente. Desta maneira, a maior tarefa é a primeira a ser alocada. Cada tarefa é alocada para o *host* que provê o menor tempo de execução CT (*Completion Time*) para ela, onde CT = TBA + *Task Cost* e o *Task Cost* = (*Task size* / *Host speed*) / (1 - *Host load*).

Quando uma tarefa é alocada para uma máquina, o valor do TBA correspondente a este *host* é incrementado com *Task Cost*. As tarefas são alocadas até que todas as máquinas do *grid* estejam em uso. Após isso, a execução da aplicação é iniciada. Quando uma tarefa é completada, todas as tarefas que não estão executando são escalonadas novamente até que todas as máquinas fiquem em uso. Isso continua até que todas as tarefas sejam completadas.

A Figura 12 apresenta um esboço do algoritmo *Dynamic* FPLTF.

```
01.   Organizar as tarefas da fila em ordem decrescente de tamanho
02.   No início atribuir zero no valor de TBA para cada host
03.   Enquanto tiver tarefa na fila
04.       Enquanto pelo menos um host estiver disponível
05.           Calcular TaskCost e CT da tarefa atual para cada host
06.           Alocar a tarefa atual para o host que possuir o melhor CT
07.           Incrementar TBA do host com o CT da tarefa atual
08.   Finalizar aplicação após execução de todas as tarefas
```

Figura 12 – Esboço do algoritmo *Dynamic* FPLTF.

Como apresentado na Figura 12, na linha 01, as tarefas da fila são organizadas em ordem decrescente de tamanho. No início da execução (linha 02), é atribuído zero para o TBA de todas as máquinas. Enquanto existir tarefas na fila e máquinas disponíveis (linhas 03 e 04), é calculado o *Task Cost* e o CT da tarefa atual (linha 05). A tarefa é alocada para a máquina com o melhor CT (linha 06) e o TBA da máquina que recebeu a tarefa é incrementado (linha 07). A aplicação é finalizada na linha 07, após a conclusão de todas as tarefas.

A Figura 13 apresenta um diagrama do algoritmo *Dynamic* FPLTF integrado ao SimGrid.

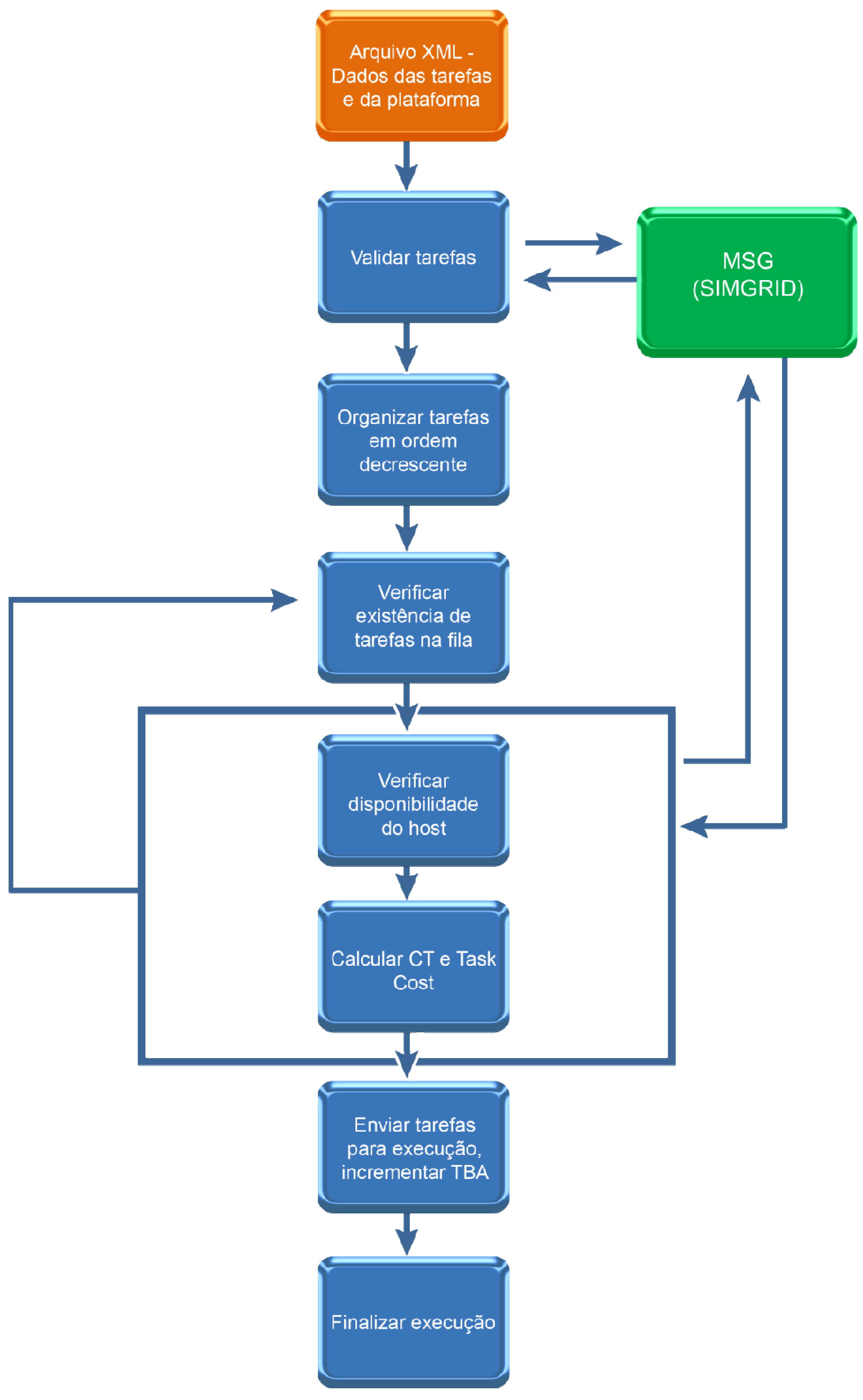

Figura 13 – Diagrama do algoritmo *Dynamic* FPLTF integrado ao SimGrid.

3.4.6. Principais características dos algoritmos

A Tabela 2 apresenta, de uma maneira sintetizada, as principais características dos algoritmos estudados neste trabalho.

Tabela 2 – Principais características dos algoritmos WQ, WQR, *Sufferage*, *XSufferage*, *Dynamic* FPLTF.

Algoritmo	Características
WQ	• Implementação simples; • Não necessita de informações da plataforma ou aplicação; • Tarefas são atribuídas aos *hosts* disponíveis; • Problema de desbalanceamento quando uma tarefa grande é atribuída um *host* lento;
WQR	• Implementação simples; • Não necessita de informações da plataforma ou aplicação; • Tarefas são atribuídas aos *hosts* disponíveis; • Utiliza replicação para diminuir o problema de desbalanceamento; • Consumo adicional de ciclos de CPU devido à replicação; • Limitar a quantidade de replicação controla o consumo adicional de ciclos de CPU, com um pouco de perda no desempenho;
Sufferage	• Implementação complexa; • Necessita de informações da plataforma e da aplicação; • Dificuldade em obter essas informações em um ambiente de *grid*; • Informações imprecisas prejudicam o desempenho; • As tarefas são atribuídas aos *hosts* de acordo com o desempenho dos recursos;
XSufferage	• Implementação complexa; • Necessita de informações da plataforma e da aplicação; • Dificuldade em obter essas informações em um ambiente de *grid*; • Informações imprecisas prejudicam o desempenho; • As tarefas são atribuídas aos *hosts* de acordo com o desempenho dos recursos; • Para aplicações que utilizam grandes quantidades de dados; • Considera a transferência dos dados de entrada da tarefa para calcular os tempos de execução;
***Dynamic* FPLTF**	• Implementação complexa; • Necessita de informações da plataforma e da aplicação; • Dificuldade em obter essas informações em um ambiente de *grid*; • Informações imprecisas prejudicam o desempenho; • As tarefas são organizadas em uma fila em ordem decrescente de tamanho; • As tarefas são atribuídas aos *hosts* de acordo com o desempenho dos recursos;

3.5. Considerações Finais

Este capítulo apresentou a taxonomia hierárquica para algoritmos de escalonamento em sistemas computacionais paralelos e distribuídos, assim como aspectos que não são abrangidos por essa taxonomia e que estão vinculados às novas mudanças no cenário de *grid* computacional. Além disso, foram apresentadas as políticas de escalonamento de tarefas WQ, WQR, *Sufferage*, *XSufferage* e *Dynamic* FPLTF que foram implementadas na LIBTS. Em cada política de escalonamento foi apresentado um diagrama para mostrar o fluxo das informações.

O WQ e o WQR são algoritmos que não utilizam nenhuma informação sobre as aplicações e os recursos. O WQ escalona as tarefas submetidas aos recursos em uma ordem arbitrária. O WQR faz a replicação de tarefas quando um processador se torna disponível e não há mais tarefas pendentes para serem colocadas em execução. Em comparação ao WQ o WQR tende a ter um melhor desempenho, pois a replicação tenta diminuir o tempo de término das tarefas em processamento (SILVA, 2003).

Os algoritmos *Sufferage*, *XSufferage* e *Dynamic* FPLTF necessitam de informações sobre as aplicações e o ambiente, como por exemplo: tamanho das tarefas, velocidade e carga das máquinas. Apesar das dificuldades em obter informações corretas sobre os recursos do *grid* e sobre as tarefas, a maioria dos estudos de escalonamento de aplicações compostas por tarefas independentes em ambientes heterogêneos e dinâmicos assumem que as informações disponíveis são confiáveis (CASANOVA et al, 1999; MAHESWARAN et al, 1999; CASANOVA et al, 2000).

Quando as informações sobre as aplicações e o ambiente estão disponíveis os estudos de James, Hawick, Coddington (1999) e Silva (2003) mostraram que os algoritmos que necessitam dessas informações apresentam melhores resultados, mas quando as informações não são fáceis de serem previstas, é melhor usar um algoritmo que não necessita das informações como o WQR e WQ.

4. DESENVOLVIMENTO DA BIBLIOTECA DE ESCALONAMENTO DE TAREFAS

Este capítulo apresenta a biblioteca de escalonamento de tarefas LIBTS (*Library Tasks Scheduling*) desenvolvida neste trabalho. A LIBTS disponibiliza os algoritmos: WQ, WQR, *Sufferage*, *XSufferage* e *Dynamic* FPLTF, como também os algoritmos básicos: FIFO, LIFO, RR, SJF para utilização no SimGrid.

4.1. Apresentando a LIBTS

A LIBTS foi desenvolvida para disponibilizar a implementação dos algoritmos de escalonamento de tarefas: WQ, WQR, Sufferage, XSufferage e Dynamic FPLTF, no SimGrid. O SimGrid é uma ferramenta de simulação de aplicações em ambientes distribuídos heterogêneos utilizada para o estudo dos algoritmos de escalonamento de tarefas em *grid* computacional, que não disponibiliza políticas internas de escalonamento de tarefas, além disso, a implementação dos algoritmos deve ser feita pelo próprio usuário.

A biblioteca LIBTS utiliza as funções do módulo MSG disponibilizadas pelo SimGrid e funciona com aplicações do tipo *Master-Slave*. Em uma aplicação *Master-Slave* um computador é definido como *Master* (mestre) e os outros como *Slaves* (escravos). A função do *Master* é controlar o envio das tarefas para os *Slaves*.

O escalonamento de tarefas tem sido uma questão importante para melhorar a execução paralela em sistemas distribuídos. O modelo *Master-Slave* tem sido aplicado com sucesso em muitas aplicações paralelizadas em diferentes domínios de aplicações. Alguns pesquisadores e desenvolvedores de aplicações, como Yuanqiang et al (2008) têm se dedicado a explorar os métodos para melhorar o desempenho de uma aplicação *Master-Slave* em sistemas heterogêneos.

A Figura 14 apresenta a visão geral do SimGrid e mostra onde a LIBTS foi adicionada para interagir com os módulos do SimGrid.

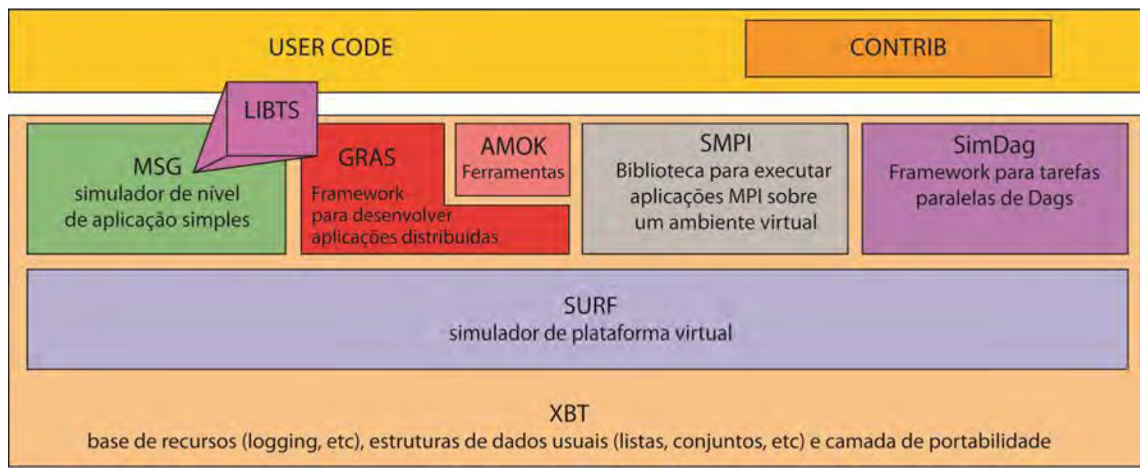

Figura 14 – SimGrid com a Biblioteca LIBTS

A biblioteca LIBTS foi desenvolvida em linguagem C, que é a linguagem utilizada pelo SimGrid. Para o desenvolvimento da LIBTS, foi utilizado um notebook com 4GB de memória RAM, processador Intel Core 2 Duo 2.10GHz com Linux Ubuntu v.10.04. Inicialmente, utilizou-se o SimGrid versão 3.3.4, depois, foi necessário atualizar para a versão 3.4.1 e para a versão 3.5, para correções de *bugs*. Para os testes das comparações entre os algoritmos implementados na biblioteca, foi utilizado o mesmo notebook do desenvolvimento da LIBTS e também um desktop com 8GB de memória RAM, processador Intel Core I7 2.93GHz com Linux Ubuntu v.10.10 e posteriormente com Linux Ubuntu v.11.04.

4.2. Arquitetura da LIBTS

A biblioteca é composta pelo módulo principal “Escalonamento”, que contém os algoritmos: WQ, WQR, *Sufferage*, *XSufferage*, *Dynamic* FPLTF, como também os algoritmos básicos: FIFO, LIFO, RR, SJF. Os algoritmos WQ, WQR, *Sufferage*, *XSufferage*, *Dynamic* FPLTF foram desenvolvidos de acordo com os conceitos explicados na seção 3.4. No entanto, para os algoritmos funcionarem no SimGrid, foram feitas adaptações seguindo os padrões dos componentes do módulo MSG Nativo (seção

4.3). Uma das alterações efetuadas foi a utilização da estrutura dos dados de acordo com os *MSG datatypes* do simulador. A arquitetura da biblioteca LIBTS é apresentada na Figura 17 (b).

O usuário que desejar desenvolver seu próprio exemplo pode utilizar a LIBTS, desde que crie a aplicação seguindo os padrões dos componentes do módulo MSG Nativo. Após a criação do exemplo, o usuário deve acrescentar em seu código a linha que chama a função "*escalonamento.c*" passando os parâmetros necessários. Exemplo:

escalonamento (todo, number_of_tasks, slaves, slaves_count);

4.3. Simulação Utilizando o Módulo MSG

O módulo MSG é um ambiente de programação simples que permite construções rápidas e flexíveis de simulação. Foi o primeiro ambiente de programação oferecido pelo SimGrid e permite a simulação de plataformas heterogêneas sem a necessidade de implementar uma aplicação. É usado para modelar aplicações como processos sequenciais concorrentes (*Concurrent Sequential Processes*) e é útil para modelar problemas teóricos e para comparar diferentes heurísticas.

A arquitetura do módulo MSG é apresentada na Figura 17 (a). O MSG Nativo é composto por:

- MSG *Data Types* (Tipos de dados MSG): descreve os diferentes tipos de dados fornecidos pelo MSG.
- *Management Functions of Agents* (Funções de Gerenciamento de Agentes): descreve a estrutura do agente de MSG e as funções de gerenciamento.
- *Management Functions of Hosts* (Funções de Gerenciamento dos *Hosts*): descreve a estrutura do *host* do MSG.
- *Managing Functions of Tasks* (Funções de Gerenciamento das Tarefas): descreve a estrutura da tarefa de MSG e as funções de gerenciamento.

- MSG *Operating System Functions* (Funções do Sistema de Operações de MSG): descreve as funções que podem ser usadas por um agente para o tratamento de alguma tarefa.

- *Understanding Channels* (Entendimento de Canais): descreve brevemente a noção de canal de MSG.

- *Platform and Application Management* (Gerenciamento da Plataforma e Aplicação): descreve as funções para gerenciar a criação da plataforma das aplicações.

- MSG *Simulation Functions* (Funções de Simulação do MSG): descreve as funções necessárias para configurar uma simulação.

Em uma simulação utilizando o módulo MSG, três itens são necessários: o arquivo binário, a descrição da plataforma e a descrição da aplicação.

O arquivo binário é obtido através da compilação do código-fonte da aplicação que deve ser escrito em C, usando as funções disponibilizadas na API do simulador. No código-fonte da aplicação é necessário fazer a codificação de cada agente (descrição da aplicação) e a criação dos recursos (descrição da plataforma física) onde são especificados os hosts, os links de comunicação e a tabela de roteamento. Após isso, é necessário criar e alocar os agentes nas localidades para a simulação ser realizada. Um agente representa uma entidade que adota decisões de escalonamento, isto é, executa em uma localidade determinada e interage através do envio, recebimento e processamento das tarefas. Uma localidade é definida como um recurso computacional localizado dentro da topologia do grid. Uma tarefa é uma atividade, no qual pode ser uma computação ou uma transferência de dados.

Para descrever a plataforma e a descrição da aplicação, pode-se utilizar arquivos XML (eXtensible Markup Language) que devem conter informações como: o nome do nó, as funções que os nós devem exercer (*server* ou *slave*), o poder computacional do nó (em MFLOPS – *Mega Floating point Operations Per Second*), a largura de banda (em *Bytes*), a latência (em segundos) e a tabela de roteamento entre os nós (SimGrid, 2010).

O exemplo de aplicação *Master-Slave* disponibilizado pelo SimGrid contém todas as informações (declaração das bibliotecas do SimGrid, dados das aplicações e plataforma, criação e organização das tarefas, entre outras) dentro do código-fonte “Masterslave_bypass.c”, desta forma não é utilizado arquivos XML para descrever as aplicações e a plataforma, como também não tem opções de escalonamento das tarefas.

A Figura 15 apresenta um trecho do código-fonte “Masterslave_bypass.c” com declaração das bibliotecas do SimGrid e das aplicações e plataforma. O código-fonte completo do “Masterslave_bypass.c” encontra-se no Anexo.

```
#include <stdio.h>
#include "msg/msg.h"
#include "surf/surfxml_parse.h"
#include "xbt/log.h"

XBT_LOG_NEW_DEFAULT_CATEGORY(msg_test, "Messages specific for
this msg example");
#define FINALIZE ((void*)221297)

{
   ⋮
}

static int surf_parse_bypass_platform(void)
{
  static int AX_ptr = 0;
  static int surfxml_bufferstack_size = 2048;

/* allocating memory for the buffer, I think 2kB should be enough
*/
  surfxml_bufferstack = xbt_new0(char, surfxml_bufferstack_size);

/* <platform> */
  SURFXML_BUFFER_SET(platform_version, "2");

  SURFXML_START_TAG(platform);

/*   <host id="host A" power="100000000.00"/> */
  SURFXML_BUFFER_SET(host_id, "host A");
  SURFXML_BUFFER_SET(host_power, "100000000.00");
  SURFXML_BUFFER_SET(host_availability, "1.0");
  SURFXML_BUFFER_SET(host_availability_file, "");
  A_surfxml_host_state = A_surfxml_host_state_ON;
```

Figura 15 – Trecho do código-fonte “Masterslave_bypass.c”.

A aplicação *Master-Slave* “Masterslave.c” utilizada neste trabalho foi criada com base no “Masterslave_bypass.c”. O “Masterslave.c” também contém as informações necessárias para o funcionamento da aplicação, com algumas diferenças, como por exemplo: para as aplicações e a plataforma são utilizados arquivos XML e o

escalonamento de tarefas é efetuado pela LIBTS, onde o usuário pode escolher qual algoritmo será utilizado.

A Figura 16 apresenta um trecho do código-fonte do “Masterslave.c” com a declaração das bibliotecas do SimGrid, da LIBTS e das aplicações e plataforma. O código-fonte completo do “Masterslave.c” encontra-se no Anexo.

```
#include <stdio.h>
#include "escalonamento.h"

XBT_LOG_NEW_DEFAULT_CATEGORY(msg_test,"Messages specific for this
msg example");
int flag_task[TCOUNT];
int atoi_task=9;
int s_flag;
int *p_flag_task=flag_task;

#include "masterslave.h"

{
  ⋮
}

  /*  Simulation setting */
MSG_set_channel_number(MAX_CHANNEL);
MSG_create_environment ("plataform_test_2.xml");

/*   Application deployment */
MSG_function_register("master", master);
MSG_function_register("slave", slave);
MSG_launch_application("application_test_2.xml");

{
  ⋮
}
INFO1("Got %d slave(s) :", slaves_count);

  for (i = 0; i < slaves_count; i++)
    INFO1("\t %s", slaves[i]->name);

  INFO1("Tem %d tarefas para processar:", number_of_tasks);

/* LIBTS */
  escalonamento (todo, number_of_tasks, slaves, slaves_count);

  INFO0("Goodbye now!");
  free(slaves);
  free(todo);
  return 0;
```

Figura 16 – Trecho do código-fonte “Masterslave.c”.

.

A Figura 17 apresenta a arquitetura de uma aplicação no SimGrid.

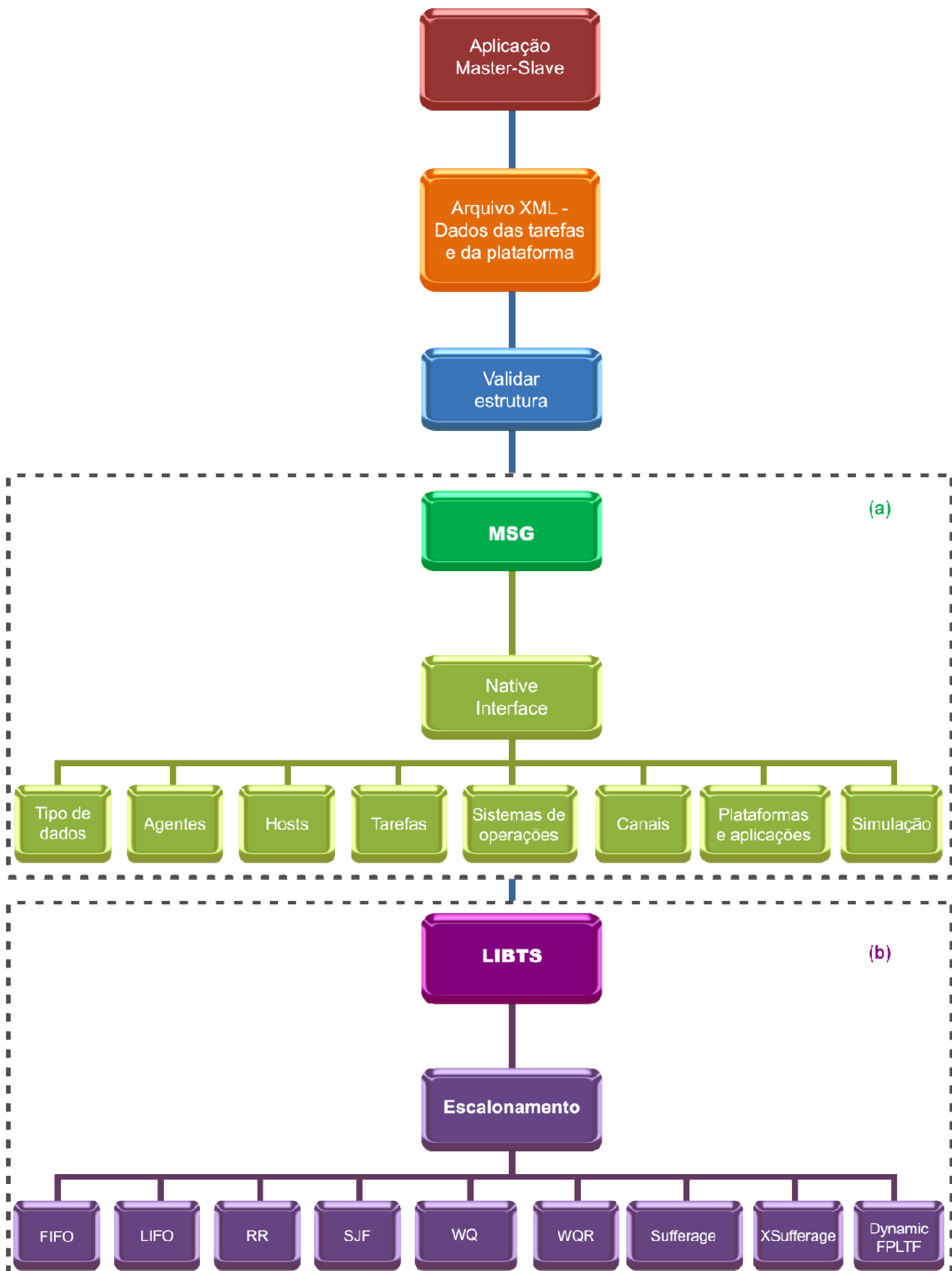

Figura 17 – Arquiterura de uma aplicação no SimGrid.

(a) Arquitetura do módulo MSG do SimGrid. (b) Arquitetura da LIBTS.

4.4. Considerações Finais

Este capítulo apresentou como a LIBTS foi desenvolvida, mostrando sua arquitetura e a estrutura de uma aplicação utilizando a LIBTS. Foi apresentada também uma síntese do módulo MSG do SimGrid informando os principais itens necessários para utilizar uma simulação neste módulo.

5. EXPERIMENTOS REALIZADOS

De acordo com MacDougall (1987) o processo de modelagem e testes no desenvolvimento de uma simulação é constituído por três fases principais: desenvolvimento, testes e análises. Deste modo, o cenário de validação apresentado na seção 5.1, foi utilizado como parte dos testes para a verificação e validação do funcionamento da LIBTS. Na seção 5.3, é apresentada a etapa de análise dos resultados dos experimentos realizados.

5.1. Validação da LIBTS

Para o cenário de validação foi utilizado 7 tarefas e 5 *hosts* com a finalidade de proporcionar o teste de mesa da execução dos algoritmos WQ, WQR, *Sufferage*, *XSufferage*, *Dynamic* FPLTF.

O algoritmo *Sufferage* é um dos algoritmos mais citados na literatura, por isso foi o algoritmo escolhido para apresentar os resultados da execução do cenário de validação. As especificações da plataforma e das aplicações são apresentadas nas Tabelas 3 e 4.

Tabela 3 – Especificação da aplicação do cenário de validação.

Tarefas	Tamanho (MFLOPS)
Tarefa 0	45000
Tarefa 1	60000
Tarefa 2	37000
Tarefa 3	35000
Tarefa 4	50000
Tarefa 5	57000
Tarefa 6	46000

Tabela 4 – Especificação da plataforma do cenário de validação.

Hosts	**Poder computacional (MFLOPS)**	**Função**
Tremblay	98095	*Master/slave*
Jupiter	70296	*Slave*
Fafard	76296	*Slave*
Ginette	40492	*Slave*
Bourassa	48492	*Slave*

Com o propósito de continuar os testes do processo de modelagem e testes no desenvolvimento de uma simulação (MACDOUGALL, 1987), o teste de mesa do cenário de validação foi efetuado para os algoritmos FIFO, LIFO, RR, SJF, WQ, WQR, *Sufferage*, *XSufferage*, *Dynamic* FPLTF para verificar o funcionamento da LIBTS. As informações da Tabela 5 confirmaram os resultados apresentados na execução do cenário de validação com o algoritmo *Sufferage* mostrados na Figura 16.

A Figura 18 apresenta a execução do algoritmo *Sufferage* na LIBTS, utilizando as especificações das Tabelas 3 e 4.

[Tremblay:master:(1) 0.000000] [msg_suff/INFO] **Enviando "0" para "Tremblay"**
[Tremblay:master:(1) 0.000000] [msg_suff/INFO] Ah! Sou eu! :)
[Tremblay:slave:(2) 0.000000] [msg_test/INFO] Eu sou um slave
[Jupiter:slave:(3) 0.000000] [msg_test/INFO] Eu sou um slave
[Fafard:slave:(4) 0.000000] [msg_test/INFO] Eu sou um slave
[Ginette:slave:(5) 0.000000] [msg_test/INFO] Eu sou um slave
[Bourassa:slave:(6) 0.000000] [msg_test/INFO] Eu sou um slave
[Tremblay:master:(1) 0.002339] [msg_suff/INFO] Completado o envio
[Tremblay:master:(1) 0.002339] [msg_suff/INFO] **Enviando "1" para "Fafard"**
[Tremblay:slave:(2) 0.002339] [msg_test/INFO] Recebido "0"
[Tremblay:slave:(2) 0.002339] [msg_test/INFO] Processando "0"
[Fafard:slave:(4) 0.220492] [msg_test/INFO] Recebido "1"
[Fafard:slave:(4) 0.220492] [msg_test/INFO] Processando "1"
[Tremblay:master:(1) 0.220492] [msg_suff/INFO] Completado o envio
[Tremblay:master:(1) 0.220492] [msg_suff/INFO] **Enviando "2" para "Jupiter"**
[Jupiter:slave:(3) 0.386454] [msg_test/INFO] Recebido "2"
[Jupiter:slave:(3) 0.386454] [msg_test/INFO] Processando "2"
[Tremblay:master:(1) 0.386454] [msg_suff/INFO] Completado o envio
[Tremblay:master:(1) 0.386454] [msg_suff/INFO] **Enviando "3" para "Bourassa"**
[Tremblay:slave:(2) 0.461078] [msg_test/INFO] "0" concluída
[Bourassa:slave:(6) 0.602326] [msg_test/INFO] Recebido "3"
[Bourassa:slave:(6) 0.602326] [msg_test/INFO] Processando "3"
[Tremblay:master:(1) 0.602326] [msg_suff/INFO] Completado o envio
[Tremblay:master:(1) 0.602326] [msg_suff/INFO] **Enviando "4" para "Tremblay"**
[Tremblay:master:(1) 0.602326] [msg_suff/INFO] Ah! Sou eu! :)
[Tremblay:slave:(2) 0.604665] [msg_test/INFO] Recebido "4"
[Tremblay:slave:(2) 0.604665] [msg_test/INFO] Processando "4"
[Tremblay:master:(1) 0.604665] [msg_suff/INFO] Completado o envio
[Tremblay:master:(1) 0.604665] [msg_suff/INFO] **Enviando "6" para "Ginette"**
[Ginette:slave:(5) 0.745124] [msg_test/INFO] Recebido "6"
[Ginette:slave:(5) 0.745124] [msg_test/INFO] Processando "6"
[Tremblay:master:(1) 0.745124] [msg_suff/INFO] Completado o envio
[Tremblay:master:(1) 0.745124] [msg_suff/INFO] **Enviando "5" para "Jupiter"**
[Jupiter:slave:(3) 0.912799] [msg_test/INFO] "2" concluída
[Fafard:slave:(4) 1.006903] [msg_test/INFO] "1" concluída
[Tremblay:master:(1) 1.078761] [msg_suff/INFO] Completado o envio
[Tremblay:master:(1) 1.078761] [msg_suff/INFO] Todas as tarefas foram despachadas.
[Jupiter:slave:(3) 1.078761] [msg_test/INFO] Recebido "5"
[Jupiter:slave:(3) 1.078761] [msg_test/INFO] Processando "5"
[Tremblay:slave:(2) 1.114375] [msg_test/INFO] "4" concluída
[Bourassa:slave:(6) 1.324094] [msg_test/INFO] "3" concluída
[Ginette:slave:(5) 1.881151] [msg_test/INFO] "6" concluída
[Jupiter:slave:(3) 1.889618] [msg_test/INFO] "5" concluída
[Jupiter:slave:(3) 1.904818] [msg_test/INFO] Recebido "Finaliza"
[Jupiter:slave:(3) 1.904818] [msg_test/INFO] Tudo OK. Até mais!
[Fafard:slave:(4) 1.925369] [msg_test/INFO] Recebido "Finaliza"
[Fafard:slave:(4) 1.925369] [msg_test/INFO] Tudo OK. Até mais!
[Ginette:slave:(5) 1.938600] [msg_test/INFO] Recebido "Finaliza"
[Ginette:slave:(5) 1.938600] [msg_test/INFO] Tudo OK. Até mais!
[Bourassa:slave:(6) 1.958936] [msg_test/INFO] Recebido "Finaliza"
[Bourassa:slave:(6) 1.958936] [msg_test/INFO] Tudo OK. Até mais!
[Tremblay:slave:(2) 1.959092] [msg_test/INFO] Recebido "Finaliza"
[Tremblay:slave:(2) 1.959092] [msg_test/INFO] Tudo OK. Até mais!
Fim
[Tremblay:master:(1) 1.959092] [msg_test/INFO] Tchau!

Figura 18 – Execução do algoritmo *Sufferage* na LIBTS

Conforme comprovado no teste de mesa (Tabela 5), a tarefa 0 foi atribuída ao *host* Tremblay por ter o melhor CT. O cálculo para encontrar o melhor CT de cada tarefa foi efetuado para todas as tarefas. A atribuição das tarefas para os *hosts* foram

feitas de acordo com o melhor CT de cada uma. No caso da tarefa 5, o melhor *host* para executá-la seria o Jupiter, mas, como nele ainda estava sendo executada a tarefa 2 e ela tem o maior valor de *Sufferage* não foi efetuada a troca. Isso acontece porque deve permanecer na máquina a tarefa que tem o maior valor de *Sufferage*. Portanto, a tarefa 5 volta para a fila para ser executada posteriormente.

A Tabela 5 apresenta o teste de mesa do algoritmo *Sufferage* que utiliza as especificações das Tabelas 3 e 4.

Tabela 5 – Teste de mesa do cenário de validação.

CT = TBA + *Task Cost*	*Task Cost* = (*Task size* / *Host speed*) / (1 - *Host load*)			
Tarefa 0	***Task Cost***	***Host***	**CT 1 execução**	**Valor *sufferage***
	0,458738978	Tremblay	0,458738978	0,131069138
	0,640150222	Jupiter	0,640150222	
	0,589808116	Fafard	0,589808116	
	1,111330633	Ginette	1,111330633	
	0,927988122	Bourassa	0,927988122	
Tarefa 1	***Task Cost***	***Host***	**CT 2 execução**	**Valor *sufferage***
	0,61165197	Tremblay	1,070390948	0,067122808
	0,853533629	Jupiter	0,853533629	
	0,786410821	Fafard	0,786410821	
	1,481774178	Ginette	1,481774178	
	1,237317496	Bourassa	1,237317496	
Tarefa 2	***Task Cost***	***Host***	**CT 3 execução**	**Valor *sufferage***
	0,377185382	Tremblay	0,835924359	0,236666718
	0,526345738	Jupiter	0,526345738	
	0,48495334	Fafard	1,271364161	
	0,913760743	Ginette	0,913760743	
	0,763012456	Bourassa	0,763012456	
Tarefa 3	***Task Cost***	***Host***	**CT 4 execução**	**Valor *sufferage***
	0,356796983	Tremblay	0,81553596	0,093767421
	0,497894617	Jupiter	1,024240355	
	0,458739646	Fafard	1,245150467	
	0,86436827	Ginette	0,86436827	
	0,721768539	Bourassa	0,721768539	
Tarefa 4	***Task Cost***	***Host***	**CT 5 execução**	**Valor *sufferage***
	0,509709975	Tremblay	0,968448953	0,7844175
	0,711278024	Jupiter	1,237623762	
	0,655342351	Fafard	1,441753172	
	1,234811815	Ginette	1,234811815	
	1,031097913	Bourassa	1,752866452	
Tarefa 5	***Task Cost***	***Host***	**CT 6 execução**	**Valor *sufferage***
	0,581069372	Tremblay	1,549518324	0,070482783
	0,810856948	Jupiter	1,337202686	
	0,74709028	Fafard	1,533501101	
	1,407685469	Ginette	1,407685469	
	1,175451621	Bourassa	1,89722016	
Tarefa 6	***Task Cost***	***Host***	**CT 7 execução**	**Valor *sufferage***
	0,468933177	Tremblay	1,43738213	0,044694651
	0,654375782	Jupiter	1,18072152	
	0,602914963	Fafard	1,389325784	
	1,13602687	Ginette	1,13602687	
	0,94861008	Bourassa	1,670378619	
Tarefa 5	***Task Cost***	***Host***	**CT 8 execução**	**Valor *sufferage***
	0,581069372	Tremblay	1,549518324	0,196298415
	0,810856948	Jupiter	1,337202686	
	0,74709028	Fafard	1,533501101	
	1,407685469	Ginette	2,543712338	
	1,175451621	Bourassa	1,89722016	

5.2. Estudo de Caso

Na fase de análise do processo de modelagem e testes no desenvolvimento de uma simulação (MACDOUGALL, 1987), outros experimentos foram realizados em diferentes cenários com o propósito de mostrar as análises estatísticas dos comparativos entre os algoritmos nos cenários das seções 5.2.1 e 5.2.2. Para possibilitar a comparação das amostras, os dados foram padronizados considerando como base o total de 100 tarefas.

Os cenários foram criados utilizando os arquivos com plataformas de 5 e 90 *hosts*, que são disponibilizados nos exemplos do módulo MSG do SimGrid. É complexo gerar manualmente um arquivo de plataforma, pois os arquivos de plataformas do SimGrid devem conter informações, como por exemplo: todos os *hosts* e *links* de roteamentos entre eles, utilizando todos os algoritmos de geração clássica do simulador (SimGrid, 2010).

Os cenários foram executados 5 vezes em cada algoritmo para obter a média dos tempos de simulação. As execuções não apresentaram resultados diferentes a cada repetição no mesmo cenário com o mesmo algoritmo, pois como apresentado na Tabela 1, na seção 2.6, uma das características de simulação do SimGrid é a simulação determinística. Desta forma, a repetição da mesma simulação (mesmo cenário, nas mesmas condições), sempre retornará os mesmos resultados.

Um ponto a ser destacado nos experimentos realizados é que os algoritmos baseados em informações sobre o ambiente e as aplicações (*Sufferage*, *XSufferage* e *Dynamic* FPLTF) foram alimentados com as informações necessárias (por exemplo: tamanho das tarefas, velocidade e carga das máquinas. Devido a complexidade de um ambiente de *grid*, nem sempre é possível obter essa situação em um ambiente real.

5.2.1. Cenários com Plataforma de 5 Hosts

Os cenários de 1 a 4 utilizaram a plataforma de 5 hosts, com aplicações de 100 a 10000 tarefas, que variam de 42000 a 65000 MFlops, e o poder computacional dos hosts de 40492 a 98095 MFlops.

Cenário 1

O Cenário 1 apresenta os valores da média do tempo de simulação dos algoritmos da plataforma com 5 *hosts* e aplicações com 100, 400 e 700 tarefas.

Tabela 6 – Média dos tempos de simulação em segundos com 100, 400 e 700 tarefas com 5 *hosts*.

Algoritmos	**Quantidade de tarefas**					
	Tempos normais			**Tempos padronizados (base de 100 tarefas)**		
	100	**400**	**700**	**100**	**400**	**700**
WQ	29,943936	116,974202	204,825801	29,943936	29,243551	29,260829
WQR	30,328051	117,378868	204,230467	30,328051	29,344717	29,175781
Sufferage	27,302542	105,755858	183,961846	27,302542	26,438965	26,280264
XSufferage	35,222690	144,119039	248,608632	35,222690	36,029760	35,515519
***Dynamic* FPLTF**	29,273109	116,778649	203,489388	29,273109	29,194662	29,069913

A Figura 19 apresenta o gráfico de desempenho dos algoritmos utilizando os tempos padronizados de acordo com a Tabela 6.

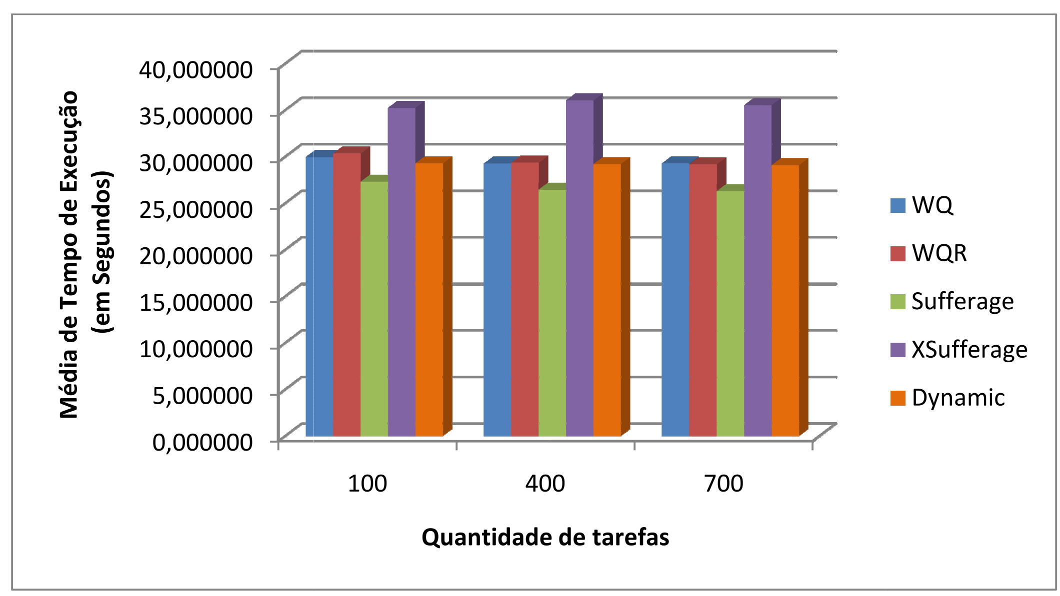

Figura 19 – Média dos tempos de simulação de acordo com a Tabela 6.

Cenário 2

O Cenário 2 apresenta os valores da média do tempo de simulação dos algoritmos da plataforma com 5 *hosts* e aplicações com 1000, 2000 e 3000 tarefas.

Tabela 7 – Média dos tempos de simulação em segundos com 1000, 2000 e 3000 tarefas com 5 *hosts*.

Algoritmos	Quantidade de tarefas					
	Tempos normais			Tempos padronizados (base de 100 tarefas)		
	1000	2000	3000	1000	2000	3000
WQ	291,762412	580,027643	869,584130	29,176241	29,001382	28,986138
WQR	291,307538	580,243516	869,752586	29,130754	29,012176	28,991753
Sufferage	265,150215	528,346873	795,747911	26,515022	26,417344	26,524930
XSufferage	372,297010	722,654899	1061,411388	37,229701	36,132745	35,380380
***Dynamic* FPLTF**	290,730349	583,226777	874,018134	29,073035	29,161339	29,133938

A Figura 20 apresenta o gráfico de desempenho dos algoritmos utilizando os tempos padronizados de acordo com a Tabela 7.

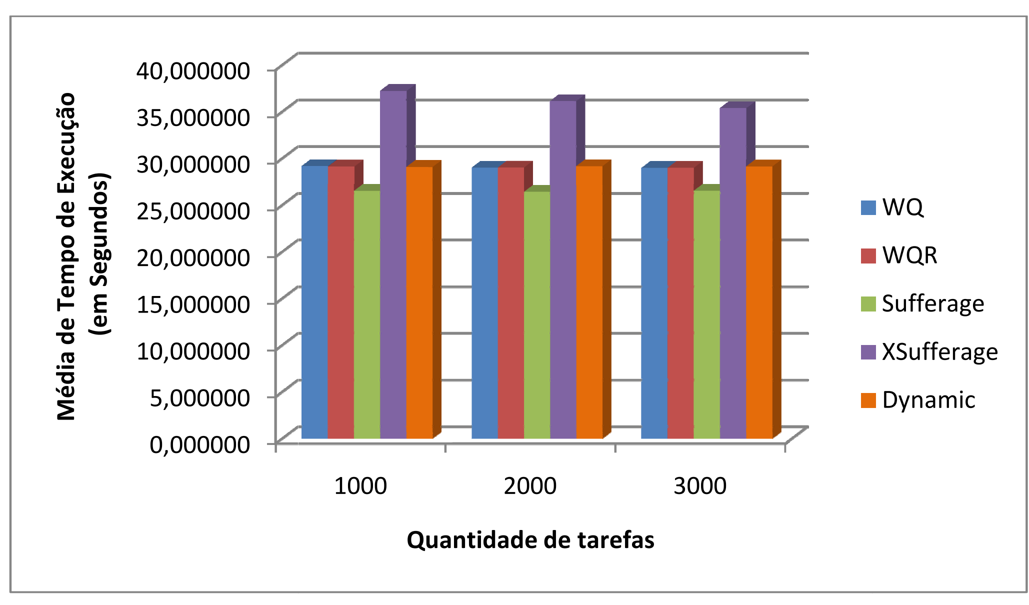

Figura 20 – Média dos tempos de simulação de acordo com a Tabela 7.

Cenário 3

O Cenário 3 apresenta os valores da média do tempo de simulação dos algoritmos da plataforma com 5 *hosts* e aplicações com 4000, 5000 e 6000 tarefas.

Tabela 8 – Média dos tempos de simulação em segundos com 4000, 5000 e 6000 tarefas com 5 *hosts*.

Algoritmos	**Quantidade de tarefas**					
	Tempos normais			**Tempos padronizados (base de 100 tarefas)**		
	4000	**5000**	**6000**	**4000**	**5000**	**6000**
WQ	1157,904407	1447,224684	1736,723628	28,947610	28,944494	28,945394
WQR	1158,072707	1447,608799	1737,128294	28,951818	28,952176	28,952138
Sufferage	1048,997326	1321,361292	1583,318298	26,224933	26,427226	26,388638
XSufferage	1438,014401	1799,814764	2130,590557	35,950360	35,996295	35,509843
***Dynamic* FPLTF**	1164,203167	1455,947507	1742,583632	29,105079	29,118950	29,043061

A Figura 21 apresenta o gráfico de desempenho dos algoritmos utilizando os tempos padronizados de acordo com a Tabela 8.

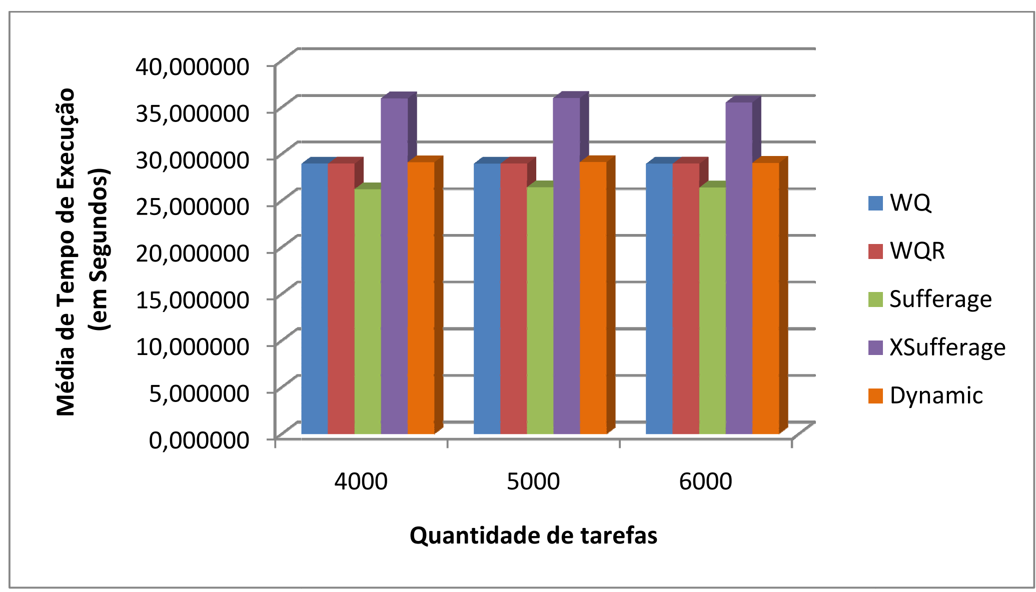

Figura 21 – Média dos tempos de simulação de acordo com a Tabela 8.

Cenário 4

O Cenário 4 apresenta os valores da média do tempo de simulação dos algoritmos da plataforma com 5 *hosts* e aplicações com 7000, 8000, 9000 e 10000 tarefas.

Tabela 9 – Média dos tempos de simulação em segundos com 7000, 8000, 9000 e 10000 tarefas com 5 *hosts*.

Algoritmos	Quantidade de tarefas							
	Tempos normais				Tempos padronizados (base de 100 tarefas)			
	7000	8000	9000	10000	7000	8000	9000	10000
WQ	2027,043906	2316,449195	2604,714426	2894,270912	28,957770	28,955615	28,941271	28,942709
WQR	2026,448571	2315,994321	2604,930298	2894,439369	28,949265	28,949929	28,943670	28,944394
Sufferage	1841,401632	2094,369995	2342,113766	2629,442032	26,305738	26,179625	26,023486	26,294420
XSufferage	2501,963435	2862,384212	3212,578968	3583,079679	35,742335	35,779803	35,695322	35,830797

Não foi possível efetuar a execução do algoritmo *Dynamic* FPLTF com aplicações acima de 6000 tarefas, devido a um erro de uma biblioteca do SimGrid. O SimGrid utiliza por padrão uma função chamada "*ucontexts*" da libc (biblioteca padrão do C). Algumas versões da glibc (biblioteca padrão do C) não implementam esta

função, isso engana o mecanismo de detecção do SimGrid que leva ao erro de falha de segmentação e, limita em alguns casos, a quantidade de tarefas simuladas (SimGrid, 2010).

A Figura 22 apresenta o gráfico de desempenho dos algoritmos utilizando os tempos padronizados de acordo com a Tabela 9.

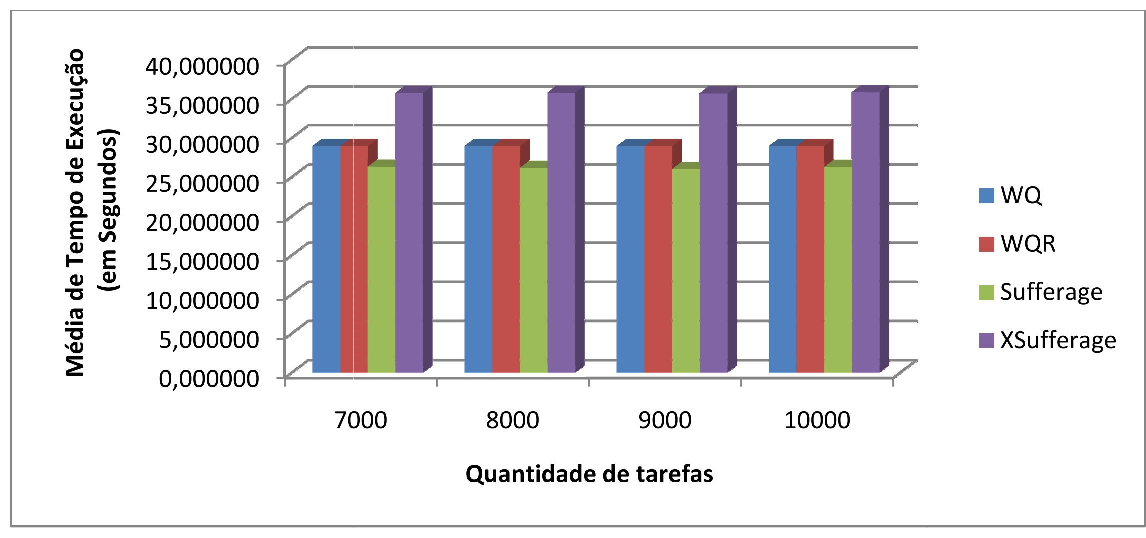

Figura 22 – Média dos tempos de simulação de acordo com a Tabela 9.

5.2.2. Cenários com Plataforma de 90 Hosts

Os cenários de 5 a 8 utilizaram a plataforma de 90 hosts, com aplicações de 100 a 10000 tarefas que variam de 42000 a 65000 MFlops e o poder computacional dos *hosts* de 22151 a 171667 MFlops.

Cenário 5

O Cenário 5 apresenta os valores da média do tempo de simulação dos algoritmos da plataforma com 90 *hosts* e aplicações com 100, 400, 700 tarefas.

Tabela 10 – Média dos tempos de simulação em segundos com 100, 400, 700 tarefas com 90 *hosts*.

Algoritmos	**Quantidade de tarefas**					
	Tempos normais			**Tempos padronizados (base de 100 tarefas)**		
	100	**400**	**700**	**100**	**400**	**700**
WQ	1500,766980	5106,902381	8713,119065	1500,766980	1276,725595	1244,731295
WQR	1632,681606	5624,934562	9577,436732	1632,681606	1406,233641	1368,205247
Sufferage	1115,904604	4299,388454	7466,441068	1115,904604	1074,847114	1066,634438
XSufferage	1210,712280	4507,314939	7804,207665	1210,712280	1126,828735	1114,886809
***Dynamic* FPLTF**	1152,869397	4412,589880	7741,316423	1152,869397	1103,147470	1105,902346

A Figura 23 apresenta o gráfico de desempenho dos algoritmos utilizando os tempos padronizados de acordo com a Tabela 10.

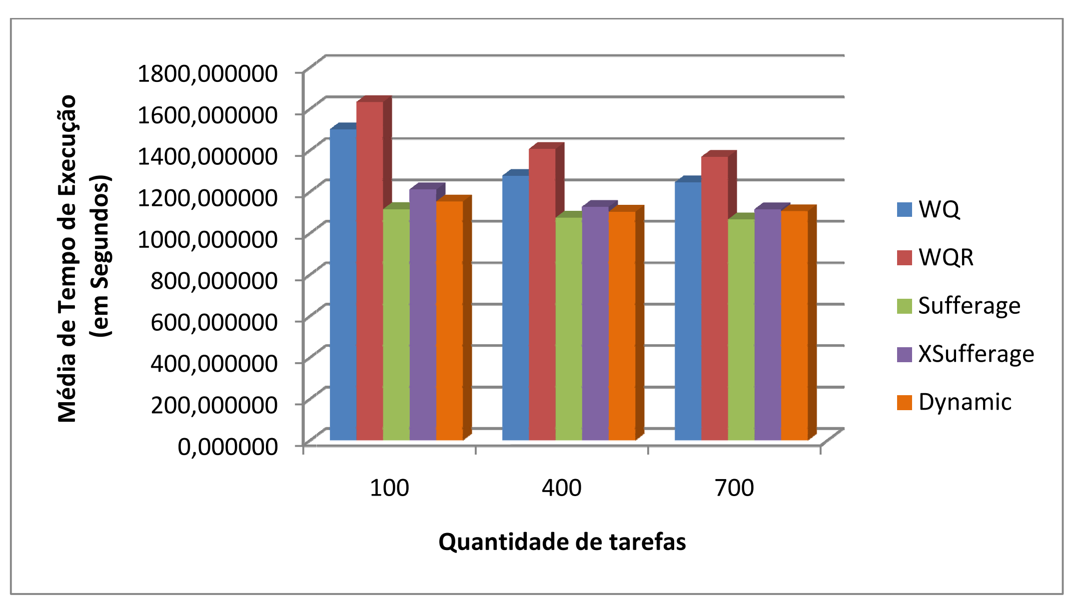

Figura 23 – Média dos tempos de simulação de acordo com a Tabela 10.

Cenário 6

O Cenário 6 apresenta os valores da média do tempo de simulação dos algoritmos da plataforma com 90 *hosts* e aplicações com 1000, 2000 e 3000 tarefas.

Tabela 11 – Média dos tempos de simulação em segundos com 1000, 2000, 3000 tarefas com 90 *hosts*.

Algoritmos	**Quantidade de tarefas**					
	Tempos normais			**Tempos padronizados (base de 100 tarefas)**		
	1000	**2000**	**3000**	**1000**	**2000**	**3000**
WQ	12282,964864	24266,074567	36270,764433	1228,296486	1213,303728	1209,025481
WQR	12413,741203	24529,847712	36679,571119	1241,374120	1226,492386	1222,652371
Sufferage	10625,858935	21192,046531	31760,608382	1062,585894	1059,602327	1058,686946
XSufferage	11067,665088	22022,442316	32996,941813	1106,766509	1101,122116	1099,898060
***Dynamic* FPLTF**	11069,317052	22095,438923	33180,097127	1106,931705	1104,771946	1106,003238

A Figura 24 apresenta o gráfico de desempenho dos algoritmos utilizando os tempos padronizados de acordo com a Tabela 11.

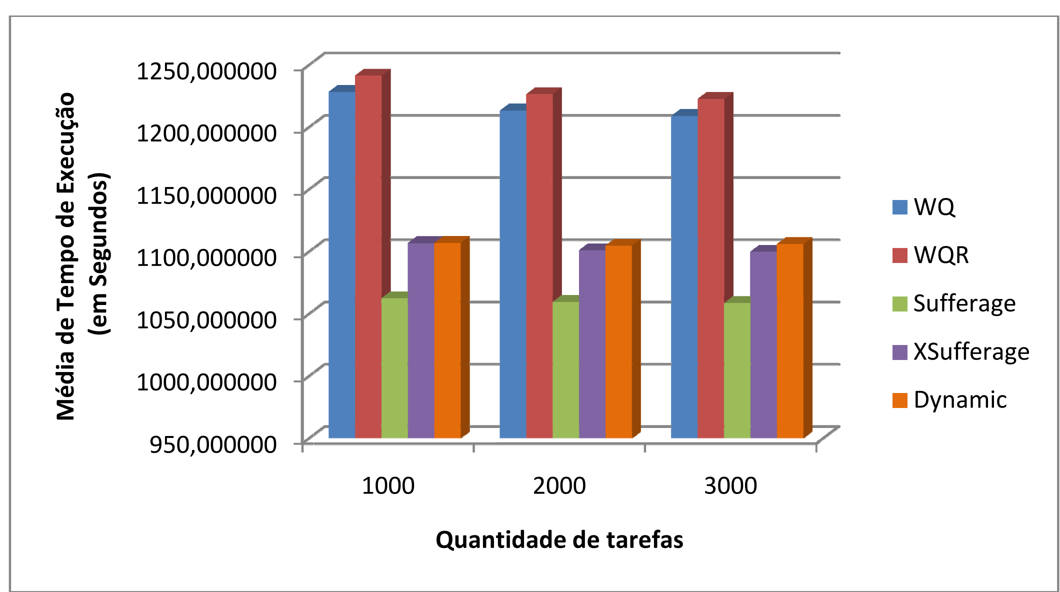

Figura 24 – Média dos tempos de simulação de acordo com a Tabela 11.

Cenários 7

O Cenário 7 apresenta os valores da média do tempo de simulação dos algoritmos da plataforma com 90 *hosts* e aplicações com 4000, 5000 e 6000 tarefas.

Tabela 12 – Média dos tempos de simulação em segundos com 4000, 5000, 6000 tarefas com 90 *hosts*.

Algoritmos	Quantidade de tarefas					
	Tempos normais			Tempos padronizados (base de 100 tarefas)		
	4000	**5000**	**6000**	**4000**	**5000**	**6000**
WQ	48235,693920	60229,246778	72217,266087	1205,892348	1204,584936	1203,621101
WQR	48731,923120	60864,790181	72974,897617	1218,298078	1217,295804	1216,248294
Sufferage	42432,393404	53011,992969	63542,294543	1060,809835	1060,239859	1059,038242
XSufferage	43935,126170	54913,861070	65868,251668	1098,378154	1098,277221	1097,804194
***Dynamic* FPLTF**	44249,559444	55314,704439	66413,089655	1106,238986	1106,294089	1106,884828

A Figura 25 apresenta o gráfico de desempenho dos algoritmos utilizando os tempos padronizados de acordo com a Tabela 12.

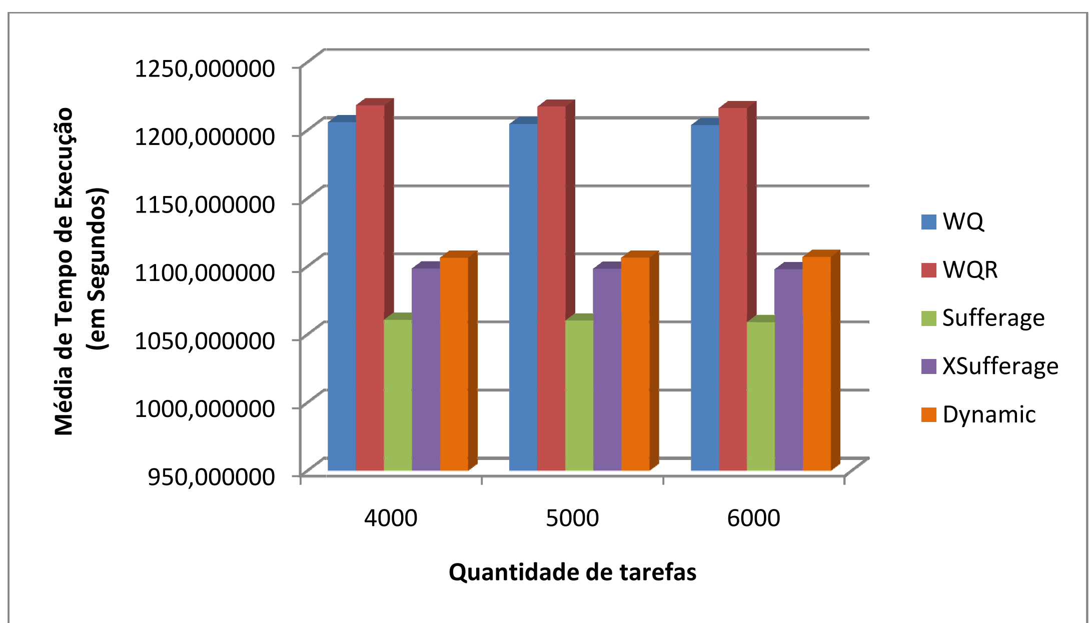

Figura 25 – Média dos tempos de simulação de acordo com a Tabela 12.

Cenários 8

O Cenário 8 apresenta os valores da média do tempo de simulação dos algoritmos da plataforma com 90 *hosts* e aplicações com 7000, 8000, 9000 e 10000 tarefas.

Tabela 13 – Média dos tempos de simulação em segundos com 7000, 8000, 9000 e 10000 tarefas com 90 *hosts*.

Algoritmos	Quantidade de tarefas							
	Tempos normais				Tempos padronizados (base de 100 tarefas)			
	7000	8000	9000	10000	7000	8000	9000	10000
WQ	84187,609536	96128,292989	108088,233256	120089,905981	1202,680136	1201,603662	1200,980370	1200,899060
WQR	85030,363686	97078,983878	109121,367960	120221,871414	1214,719481	1213,487298	1212,459644	1202,218714
Sufferage	74181,850449	84764,872832	95367,651148	105991,412554	1059,740721	1059,560910	1059,640568	1059,914126
XSufferage	76802,877319	87731,322065	98662,052585	109637,193165	1097,183962	1096,641526	1096,245029	1096,371932

Não foi possível efetuar a execução do algoritmo *Dynamic* FPLTF com aplicações acima de 6000 tarefas, devido a um erro de uma biblioteca do SimGrid. O SimGrid utiliza por padrão uma função chamada "ucontexts" da libc (biblioteca padrão do C). Algumas versões da glibc (biblioteca padrão do C) não implementam esta função, isso engana o mecanismo de detecção do SimGrid que leva ao erro de falha de segmentação e, limita em alguns casos, a quantidade de tarefas simuladas (SimGrid, 2010).

A Figura 26 apresenta o gráfico de desempenho dos algoritmos utilizando os tempos padronizados de acordo com a Tabela 13.

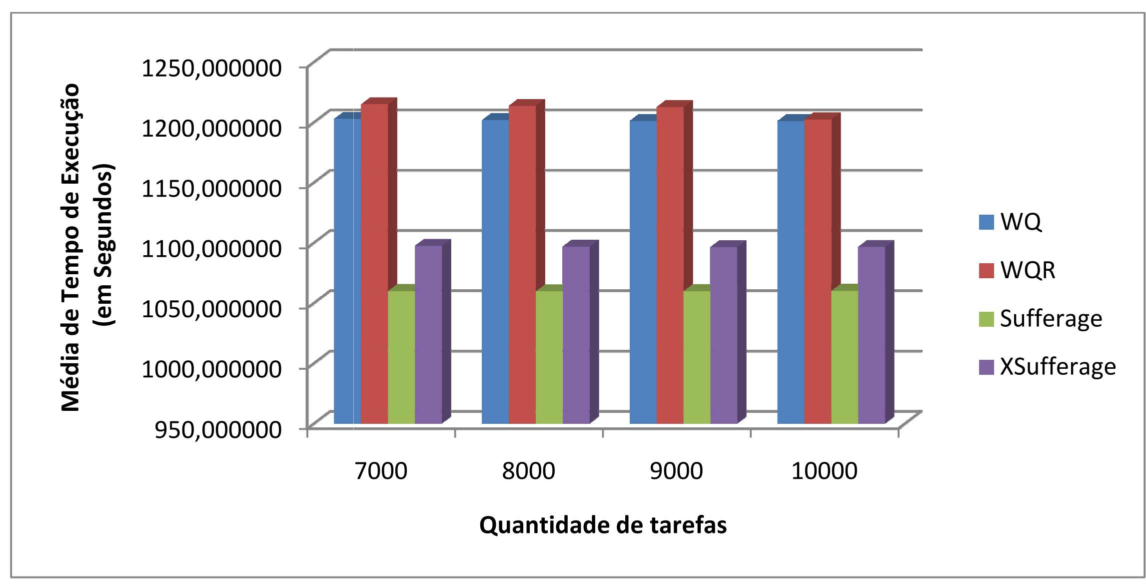

Figura 26 – Média dos tempos de simulação de acordo com a Tabela 13.

5.3. Análises Estatísticas dos Cenários

As análises estatísticas foram efetuadas utilizando a distribuição *t* de *Student* para estimar os valores médios e nível de confiança do tempo de simulação em segundos do universo correspondente as amostras (COSTA NETO, 2002). Para possibilitar a comparação das amostras, os dados foram padronizados considerando como base o total de 100 tarefas.

As Tabelas 14 e 15 apresentam as médias dos tempos de execução de 100 a 10000 tarefas das plataformas de 5 *hosts* e 90 *hosts* utilizados nos cenários das seções 5.2.1 e 5.2.2. Estas tabelas foram utilizadas para efetuar as análises estatísticas dos resultados.

Tabela 14 – Média dos tempos de simulação em segundos com plataforma de 5 *hosts*.

Tarefas	**Algoritmos**				
	WQ	**WQR**	***Sufferage***	***XSufferage***	***Dynamic* FPLTF**
100	29,943936	30,328051	27,302542	35,222690	29,273109
400	29,243551	29,344717	26,438965	36,029760	29,194662
700	29,260829	29,175781	26,280264	35,515519	29,069913
1000	29,176241	29,130754	26,515022	37,229701	29,073035
2000	29,001382	29,012176	26,417344	36,132745	29,161339
3000	28,986138	28,991753	26,524930	35,380380	29,133938
4000	28,947610	28,951818	26,224933	35,950360	29,105079
5000	28,944494	28,952176	26,427226	35,996295	29,118950
6000	28,945394	28,952138	26,388638	35,509843	29,043061
7000	28,957770	28,949265	26,305738	35,742335	*
8000	28,955615	28,949929	26,179625	35,779803	*
9000	28,941271	28,943670	26,023486	35,695322	*
10000	28,942709	28,944394	26,294420	35,830797	*

* Não foi possível efetuar a execução do algoritmo *Dynamic* FPLTF com aplicações acima de 6000 tarefas, devido a um erro de uma biblioteca do SimGrid. O SimGrid utiliza por padrão uma função chamada "ucontexts" da libc (biblioteca padrão do C). Algumas versões da glibc (biblioteca padrão do C) não implementam esta função, isso engana o mecanismo de detecção do SimGrid que leva ao erro de falha de segmentação e, limita em alguns casos, a quantidade de tarefas simuladas (SimGrid, 2010).

Tabela 15 – Média dos tempos de simulação em segundos com plataforma de 90 *hosts*.

Tarefas	**Algoritmos**				
	WQ	**WQR**	***Sufferage***	***XSufferage***	***Dynamic* FPLTF**
100	1500,766980	1632,681606	1115,904604	1210,712280	1152,869397
400	1276,725595	1406,233641	1074,847114	1126,828735	1103,147470
700	1244,731295	1368,205247	1066,634438	1114,886809	1105,902346
1000	1228,296486	1241,374120	1062,585894	1106,766509	1106,931705
2000	1213,303728	1226,492386	1059,602327	1101,122116	1104,771946
3000	1209,025481	1222,652371	1058,686946	1099,898060	1106,003238
4000	1205,892348	1218,298078	1060,809835	1098,378154	1106,238986
5000	1204,584936	1217,295804	1060,239859	1098,277221	1106,294089
6000	1203,621101	1216,248294	1059,038242	1097,804194	1106,884828
7000	1202,680136	1214,719481	1059,740721	1097,183962	*
8000	1201,603662	1213,487298	1059,560910	1096,641526	*
9000	1200,980370	1212,459644	1059,640568	1096,245029	*
10000	1200,899060	1202,218714	1059,914126	1096,371932	*

* Não foi possível efetuar a execução do algoritmo *Dynamic* FPLTF com aplicações acima de 6000 tarefas, devido a um erro de uma biblioteca do SimGrid. O SimGrid utiliza por padrão uma função chamada “ucontexts” da libc (biblioteca padrão do C). Algumas versões da glibc (biblioteca padrão do C) não implementam esta função, isso engana o mecanismo de detecção do SimGrid que leva ao erro de falha de segmentação e, limita em alguns casos, a quantidade de tarefas simuladas (SimGrid, 2010).

As Tabelas 16 e 17 apresentam os resultados das análises estatísticas dos experimentos realizados nas seções 5.2.1 e 5.2.2.

O objetivo básico da estatística descritiva é sintetizar um universo de valores de uma mesma natureza, permitindo assim uma visão global da variação desses valores. As análises estatísticas efetuadas neste trabalho foram realizadas em função da distribuição *t* de *Student*, com nível de confiança de 95% (COSTA NETO, 2002).

Tabela 16 – Análises estatísticas dos cenários com plataforma de 5 *hosts*.

Algoritmos	**Média**	**Desvio Padrão**	**Margem de erro (95%)**	**Intervalo de confiança**
WQ	29,0959	0,2813	0,1700	28,9259 - 29,2659
WQR	29,1251	0,3814	0,2305	28,8946 - 29,3556
Sufferage	26,4095	0,3028	0,1830	26,2265 - 26,5925
XSufferage	35,8473	0,4941	0,2986	35,5487 - 36,1459
Dynamic FPLTF	29,1303	0,0714	0,0549	29,0754 - 29,1852

Tabela 17 – Análises estatísticas dos cenários com plataforma de 90 *hosts*.

Algoritmos	**Média**	**Desvio Padrão**	**Margem de erro (95%)**	**Intervalo de confiança**
WQ	1237,9316	82,0485	49,5814	1188,3502 - 1287,5130
WQR	1276,3360	124,7227	75,3692	1200,9668 - 1351,7052
Sufferage	1065,9389	15,6574	9,4617	1056,4772 - 1075,4006
XSufferage	1110,8551	31,3192	18,9260	1091,9291 - 1129,7811
Dynamic FPLTF	1111,0049	15,7431	12,1012	1098,9037 - 1123,1061

As análises estatísticas apresentam uma síntese dos resultados dos experimentos realizados com os cenários. Através da média foi possível constatar que os algoritmos que necessitam de informações sobre as plataformas e aplicações (*Sufferage*, *XSufferage* e *Dynamic* FPLTF) na maioria dos experimentos realizados tiveram melhor desempenho em comparação aos algoritmos WQ e WQR que não necessitam de informações das plataformas e aplicações para escalonar as tarefas.

Os experimentos que proporcionaram um desvio padrão maior indicam que as médias dos tempos de execução de suas tarefas tiveram uma variabilidade maior nos resultados. Essa variabilidade pode ser confirmada pelos valores apresentados na Tabela 15. Nessa Tabela foi possível verificar a dispersão dos valores do WQR para 100, 400 e 700 tarefas em relação aos valores de 1000 a 10000 tarefas que não apresentaram grandes diferenças, sendo assim, a dispersão dos valores influencia nos resultados do desvio padrão apresentado.

5.4. Considerações Finais

Este capítulo apresentou os testes efetuados através do teste de mesa do cenário de validação (seção 5.1), como também dos cenários do estudo de caso (seção 5.2) para validar o funcionamento da LIBTS.

O teste de mesa confirmou os resultados apresentados na execução do cenário de validação da LIBTS. Os experimentos realizados com os cenários da seção 5.2.1 e 5.2.2 também validaram o funcionamento da LIBTS por meio das comparações dos cenários. Os resultados obtidos foram apresentados em tabelas e gráficos.

6. CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou uma revisão de literatura sobre *grid* computacional, abordando o conceito, a arquitetura e as características próprias desse ambiente que o tornam complexo. Devido à complexidade desse ambiente, foi evidenciada a importância do estudo de escalonamento de tarefas que continua sendo uma das principais técnicas estudadas em *grid* computacional. Além disso, foram apresentadas as políticas de escalonamento de tarefas WQ, WQR, *Sufferage*, *XSufferage* e *Dynamic* FPLTF que foram implementadas na LIBTS. Em cada política de escalonamento foi apresentado um diagrama com o fluxo das informações e as características principais dos algoritmos foram apresentadas em uma tabela.

Foram destacadas as vantagens da utilização de simulação para analisar e comparar os algoritmos de escalonamento de tarefas ao invés de utilizar um ambiente de *grid* real. Desta forma, foram descritas as ferramentas de simulação para *grid* computacional, sendo elas: Brincks, GridSim, MicroGrid, OptorSim, SimGrid, apresentando seus pontos fortes, deficientes e suas principais características em uma tabela comparativa.

O SimGrid é uma ferramenta de simulação que oferece funcionalidades básicas para simulação em ambientes distribuídos, entretanto, não disponibiliza políticas internas de escalonamento de tarefas e a implementação dos algoritmos deve ser feita pelo próprio usuário. Toda vez que um usuário quer estudar sobre escalonamento de tarefas em *grid* computacional, como também comparar as políticas de escalonamento de tarefas existentes com novas políticas, necessitam implementá-las. Desta maneira, o usuário perde muito tempo, pois além de precisar fazer um estudo aprofundado das políticas de escalonamento de tarefas e do funcionamento do simulador, necessita fazer as implementações dos algoritmos integrando ao simulador.

Portanto, a biblioteca LIBTS foi desenvolvida e integrada ao simulador SimGrid para auxiliar no estudo dos algoritmos de escalonamento de tarefas em *grid* computacional, permitindo aos usuários dedicar-se ao estudo das políticas de escalonamento existentes através da comparação dos algoritmos implementados, da análise dos resultados, assim como a implementação de novos algoritmos.

Vários cenários foram utilizados para testar a LIBTS. O cenário de validação foi criado e executado na LIBTS e seus resultados foram confirmados pelo teste de mesa, validando assim o funcionamento da LIBTS. Outros testes de comparação dos algoritmos foram realizados, o desempenho dos tempos de simulação das aplicações foram apresentados em tabelas e gráficos. As análises estatísticas foram efetuadas utilizando a distribuição *t* de *Student* para estimar os valores médios e nível de confiança dos tempos de simulação.

6.1. Contribuições do Trabalho

Este trabalho apresenta como contribuições:

- Um estudo bibliográfico sobre as políticas de escalonamento em *grid* computacional.
- As características e metodologias dos algoritmos WQ, WQR, *Sufferage*, *XSufferage* e *Dynamic* FPLTF implementados na LIBTS.
- Os caminhos para novas implementações na LIBTS, disponibilizando um ambiente amigável e de simples utilização para o estudo dos algoritmos de escalonamentos em *grid*.
- Um modelo de como implementar os algoritmos de escalonamento de tarefas no módulo MSG, permitindo aos pesquisadores uma melhor compreensão da integração dos algoritmos com o SimGrid.
- Os resultados das análises estatísticas das comparações entre os algoritmos implementados na LIBTS.
- A otimização para os testes de comparações entre diferentes cenários e algoritmos, pois após a implementação dos algoritmos na LIBTS, não é necessário cada usuário refazer as implementações dos algoritmos, somente precisam implementar os novos algoritmos que desejam utilizar.

6.2. Trabalhos Futuros

Com o desenvolvimento deste trabalho surgiram algumas necessidades que podem ser úteis para complementar este estudo e acrescentar melhorias em novos trabalhos:

- Flexibilizar o desenvolvimento de novas plataformas e aplicações dos arquivos XML no simulador SimGrid.
- Utilizar *traces* gerados por softwares de monitoração para obter as informações das aplicações e plataformas em tempo real, como por exemplo: carga de CPU.
- Implementar outras políticas existentes de escalonamento de tarefas na LIBTS para realizar novos estudos e comparações, e com os estudos, propor novas políticas de escalonamento de tarefas.

REFERÊNCIAS BIBLIOGRÁFICAS

AIDA, K.; et al. **Performance evaluation model for scheduling in a global computing system**. The International Journal of High Performance Computing Applications; 14(3), 2000.

ANG LI; NIANMING YAO; PEIYU HONG. **A cost and time balancing algorithm for scheduling parallel tasks on Computing Grid.** Computer, Mechatronics, Control and Electronic Engineering (CMCE), p. 185-188, 2010.

BELALEM, G.; SLIMANI, Y. **Consistency Management for Data Grid in OptorSim Simulator**. Multimedia and Ubiquitous Engineering (MUE'07), p.554-560, 2007.

BERMAN, F. **The Grid: Blueprint for a New Computing Infrastructure, chapter High-performance schedulers**. Ed. Morgan Kaufmann, San Francisco, USA, p. 279-309, 1998.

BERSTIS, V. **Fundamentals of Grid Computing**. IBM Redbooks Paper, 2002. Disponível em: <http://www.redbooks.ibm.com/redpapers/pdfs/redp3613.pdf>. Acesso em: março de 2010.

BUYYA, R. **Economic-based Distributed Resource Management and Scheduling for Grid Computing**. (Doutorado). 2002. Disponível em: <http://www.buyya.com/thesis/>. Acesso em: março de 2010.

BUYYA, R.; SULISTIO, A. **Service and Utility Oriented Distributed Computing Systems: Challenges and Opportunities for Modeling and Simulation Communities**. Simulation Symposium (ANSS 2008), p.68-81, 2008.

CAMERON, D.G.; et. al. **OptorSim: a simulation tool for scheduling and replica optimization in data grids**. 2002. Disponível em: <http://www.gridpp.ac.uk/papers/chep04_OptorSim.pdf>. Acesso em: março de 2010.

CASANOVA, H. **SimGrid: A Toolkit for the Simulation of Application Scheduling**. Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01), p. 430-437, May 2001.

CASANOVA, H.; et al. **Heuristics for Scheduling Parameter Sweep Applications in Grid environments**. In 9th Heterogeneous Computing Systems Workshop (HCW 2000), 2000.

CASANOVA, H.; et al. **Using Simulation to Evaluate Scheduling Heuristics for a Class of Applications in Grid Environments**. Ecole Normale Superieure de Lyon, 1999.

CASANOVA, H.; LEGRAND, A.; QUINSON, M. **SimGrid: a Generic Framework for Large-Scale Distributed Experimentations**. 10th IEEE International Conference on Computer Modelling and Simulation. (EUROSIM/UKSim2008), p. 126-131, 2008.

CASAVANT, T.; KUHL, J. **A Taxonomy of Scheduling in General-purpose Distributed Computing Systems**. IEEE Trans. on Software Engineering, v. 14, n .2, p. 141-154, Feb. 1988.

CHO-CHIN LIN; CHUN-WEI SHIH; **An efficient scheduling algorithm for grid computing with periodical resource reallocation.** Computer and Information Technology, 2008. 8th IEEE International Conference, p. 295 – 300, 2008.

CIRNE, W. **Grids Computacionais: Arquiteturas, Tecnologias e Aplicações**. Terceiro Workshop em Sistemas Computacionais de Alto Desempenho, 2002.

CIRNE, W; et. al. **On the efficacy, efficiency and emergent behavior of task replication in large distributed systems**. Journal of Parallel Computing, v.33, n.3, 2007.

COSTA NETO, P.L.O. **Estatística**. 3ª. ed., Ed. Blücher, ISBN: 8521203004, 280p, 2002.

DANTAS, M. **Computação Distribuídas de Alto Desempenho: Redes, Clusters e Grids Computacionais**. Ed. Axcel Books do Brasil, ISBN: 85-7323-240-4, 278p, 2005.

DataGRID. **OptorSim**. Disponível em: <http://edg-wp2.web.cern.ch/edg-wp2/optimization/optorsim.html>. Acesso em: março de 2010.

DONG, F.; AKL, S. G. **Scheduling Algorithms for Grid Computing: State of the Art and Open Problems**. Technical Report List for 2006 - Queen's University School of Computing, 2006.

FALAVINHA Jr, J.N. **Escalonamento de tarefas em Sistemas Distribuídos baseado no Conceito de Propriedade Distribuída**. (Doutorado). UNESP, São José do Rio Preto-SP, 2009.

FOSTER, I. **The Grid: A New Infrastructure for 21st Century Science**. Disponível em: <http://scitation.aip.org/journals/doc/PHTOAD-ft/vol_55/iss_2/42_1.shtml>. 2002. Acesso em: março de 2010.

FOSTER, I. **What is the Grid? A Three Point Checklist**. jul 2002a. Disponível em:<http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf>. Acesso em: março de 2010.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. **The anatomy of the Grid: Enabling scalable virtual organizations**. Intl. J. Supercomputer Applications, 2001. Disponível

em: <http://www.globus.org/alliance/publications/papers/anatomy.pdf>. Acesso em: março de 2010.

FOSTER, I; KESSELMAN, C. **The Grid: Blueprint for a New Computing Infrastructure**. San Francisco, USA: Morgan Kaufmann Publishers, ISBN: 1-55860-475-8, 677 p., 1999.

GHANEM, A.M.A.; SALEH, A.I.; ALI, H.A. **High performance adaptive framework for scheduling Grid Workflow applications**. Computer Engineering and Systems (ICCES), p. 52-57, 2010.

GridSim. **A Grid Simulation Toolkit for Resource Modelling and Application Scheduling for Parallel and Distributed Computing**. 2002. Disponível em:<http://www.gridbus.org/GridSim/>. Acesso em: março de 2010.

HAO TIAN; **A New Resource Management and Scheduling Model in Grid Computing Based on a Hybrid Genetic Algorithm**. CCCM '08. ISECS International Colloquium on Computing, Communication, Control, and Management, v.3, p.113-117, 2008.

HE, X.; SUN, X.-H.; LASZEWSKI, G. **A QoS Guided Scheduling Algorithm for the Computational Grid**. In the Proc. of the International Workshop on Grid and Cooperative Computing (GCC02), Hainan, Chian, Dec. 2002.

HONG JIANG; TIANWEI NI; **PB-FCFS-a task scheduling algorithm based on FCFS and backfilling strategy for grid computing.** Pervasive Computing (JCPC), p. 507-510, 2009.

HUAXIA XIA et al. **The MicroGrid: using online simulation to predict application performance in diverse grid network environments**. Challenges of Large Applications in Distributed Environments (CLADE 2004), p. 52-61, Jun. 2004.

INGRID – Iniciativa Nacional Grid. **Partilha de recursos computacionais de elevado desempenho.** 2006. Disponível em:<http://www.gridcomputing.pt/>. Acesso em: março de 2010.

IOSUP A.; EPEMA D. **Grid Computing Workloads**. IEEE Internet Computing, p. 19-26, March/April 2011.

JAIN, H. **The Art of Computer Systems Performance Analysis: Tecniques for Experimental Design Measurement, Simulation, and Modeling**. Wiley, 1991.

JAMES, H.; HAWICK, K.; CODDINGTON, P. **Scheduling Independent Tasks on Metacomputing Systems**. The University of Adelaide. DHPC-066, 1999.

KU-MAHAMUD, KU RUHANA; NASIR, HUSNA JAMAL ABDUL. **Ant Colony Algorithm for Job Scheduling in Grid Computing.** Mathematical/Analytical Modelling and Computer Simulation (AMS), p. 40-45, 2010.

KUN-MING YU; CHENG-KWAN CHEN. **An Evolution-Based Dynamic Scheduling Algorithm in Grid Computing Environment.** Intelligent Systems Design and Applications (ISDA'08), p. 450-455, 2008.

MACDOUGALL, M.H. **Simulating computer system, techniques and tools**. The MIT Press, 1987.

MAGOULÈS, F; **Fundamentals of grid computing: theory, algorithms and technologies.** ISBN 978-1-4398-0367-7. 2009.

MAHESWARAN, M.; et. al. **Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems**. Journal of Parallel and Distributed Computing - Special issue on software support for distributed computing. Issue 2, v.59, Nov. 1999.

MARTINS, F.; et al. **Desafios para Provisão de Integridade de Processamento em Grades Computacionais**. WCGA, 2006.

MENASCÉ, D.; et al. **Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures**. Journal of Parallel and Distributed Computing, p. 1-18, 1995.

MUNIR, E.U. et al. **Performance Analysis of Task Scheduling Heuristics in Grid. Machine Learning and Cybernetics**. ISBN: 978-1-4244-0973-0, 2007.

MURSHED, M.; BUYYA, R. **Using the GridSim toolkit for enabling grid computing education**. International Conference on Communication Networks and Distributed Systems Modeling and Simulation (CNDS 2002), 2002.

NUKARAPU, D.; et al. **Data Replication in Data Intensive Scientific Applications With Performance Guarantee**. Parallel and Distributed Systems, IEEE Transactions, v. 22, n. 99, 2011.

OLIVEIRA, L.J. **Comparação de ferramentas de simulação de grades computacionais**. 2007. Disponível em: <http://www.dcce.ibilce.unesp.br/spd/pubs/gridsimulationtools.pdf>. Acesso em: março de 2010.

OMII-UK. **Project: OptoSim**. 2006. Disponível em: <http://www.omii.ac.uk/repository/project.jhtml?pid=99>. Acesso em: março de 2010.

PARANHOS, D.; CIRNE, W.; BRASILEIRO, F. V. **Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids**.

Euro-Par 2003 Parallel Processing. Lecture Notes in Computer Science, v. 2790/2003, p.169-180, 2003.

QUETIER, B.; CAPPELLO, F. **A survey of grid research tools: simulators, emulators and real life platforms**. 17th IMACS World Congress (IMACS 2005). 2005.

RAMALHO, J.A.A. **Oracle 10G**. Ed. Cengage Learning. ISBN: 8522104565, 396 p., 2005.

REIS, V.Q. **Escalonamento em grids computacionais: estudo de caso**. (Mestrado). ICMC-USP, São Carlos-SP, 2005.

SCHOPF, J.M. **A General Architecture for Scheduling on the Grid**. Special Issue on Grid Computing, J. Parallel and Distributed Computing, April 2002.

SHA FAN; **Session Scheduling Algorithm of Grid Computing.** Knowledge Discovery and Data Mining, 2010 (WKDD'10), p. 3-6, 2010.

SHIHONG FANG; HONG LUO; **Research on task scheduling algorithms of grid computing based on multipled QoS constrain and genetic algorithms.** Information Science and Engineering (ICISE), p. 5354-5357, 2010.

SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. **Sistemas Operacionais: Conceitos e Aplicações**. Ed. Campus, 2001.

SILVA, D.P. **Usando Replicações para Escalonar Tarefas Bag-of-Tasks em Grids Computacionais**. (Mestrado), Universidade Federal de Campina Grande (UFCG), 2003.

SimGrid. **SimGrid Project. Toolkit for simulation of distributed applications in heterogeneous distributed environments**. Disponível em: <http://SimGrid.gforge.inria.fr/>. Acesso em: março de 2010.

SONG, H.J.; et. al. **The MicroGrid: a scientific tool for modeling computacional grids**. Supercomputing '00 Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), n.53, 2000.

SOUZA, P.S.L. **AMIGO: Uma Contribuição para a Convergência na Área de Escalonamento de Processos**. (Doutorado). ICMC-USP, São Carlos, Brasil. 2000.

SUCHANG GUO; et al. **Grid Service Reliability Modeling and Optimal Task Scheduling Considering Fault Recovery**. IEEE Transactions on Reliability, v. 60, n. 1, March 2011.

SULISTIO, A.; YEO, C.S.; BUYYA, R. **A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools**. Journal Software – Practice & Experience. Issue 7, v.34, Jun. 2004.

TAKEFUSA, A. et al. **Overview of a performance evaluation system for global computing scheduling algorithms**. HPDC '99 Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, p. 11, 1999.

TANENBAUM, A.S. **Sistemas operacionais modernos**. Ed. Pearson Prentice Hall. ISBN: 978-85-7605-237-1, 653 p., 2010.

WEIFENG SUN; et al. **A Priority-Based Task Scheduling Algorithm in Grid**. Parallel Architectures, Algorithms and Programming (PAAP), p.311-315, Dec. 2010.

YOUCHAN ZHU; XUEYING GUO; YANYAN CHEN;**A Heuristic Scheduling Algorithm for Computing of Power Grid.** E-Business and Information System Security (EBISS '09), p. 1-5, 2009.

YU LIANG; ZHOU JILIU; **The Improvement of a Task Scheduling Algorithm in Grid Computing.** Data, Privacy, and E-Commerce (ISDPE 2007), p. 292-297, 2007.

YUANQIANG HUANG; et al. **EOMT: A Master-Slave Task Scheduling Strategy for Grid Environment**. High Performance Computing and Communications (HPCC'08), pp.226-233, Sept. 2008.

ZHIHONG XU; XIANGDAN HOU; JIZHOU SUN;**Ant algorithm-based task scheduling in grid computing.** Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference Digital Object Identifier: 10.1109/CCECE.2003.1226090. 2003 , Page(s): 1107 - 1110 vol.2.

ANEXO

Procedimentos para instalação da LIBTS

Para que o usuário utilize a LIBTS, é necessário efetuar os seguintes passos:

1. Instalar o SimGrid conforme instruções no site.
2. Descompactar e copiar a pasta LIBTS para o local desejado.
3. Configurar o arquivo Makefile de acordo com o caminho da LIBTS. No Makefile, as linhas que deverão ser alteradas estarão indicadas com um comentário.

Menu inicial da LIBTS

O menu da biblioteca LIBTS é apresentado como na Figura A.

```
Digite um número para escolher o algoritmo de escalonamento:

  1 -> First In First Out (FIFO)
  2 -> Last In First Out (LIFO)
  3 -> Round Robin (RR)
  4 -> Short Job First (SJF)
  5 -> Workqueue (WQ)
  6 -> Workqueue with Replication (WQR)
  7 -> Sufferage
  8 -> XSufferage
  9 -> Dynamic FPLTF
  0 -> finalizar
```

Figura A – Menu da LIBTS

Masterslave_bypass.c

```
/*     $Id$    */
/* Copyright (c) 2002,2003,2004 Arnaud Legrand. All rights reserved.        */
/* This program is free software; you can redistribute it and/or modify it
 * under the terms of the license (GNU LGPL) which comes with this package. */

#include <stdio.h>
#include "msg/msg.h"            /* Yeah! If you want to use msg, you need to
include msg/msg.h */
#include "surf/surfxml_parse.h" /* to override surf_parse and bypass the
parser */

/* Create a log channel to have nice outputs. */
#include "xbt/log.h"
XBT_LOG_NEW_DEFAULT_CATEGORY(msg_test,
                             "Messages specific for this msg example");
#define FINALIZE ((void*)221297)        /* a magic number to tell people to
stop working */

static int surf_parse_bypass_platform(void)
{
  static int AX_ptr = 0;
  static int surfxml_bufferstack_size = 2048;

  /* allocating memory for the buffer, I think 2kB should be enough */
  surfxml_bufferstack = xbt_new0(char, surfxml_bufferstack_size);

  /* <platform> */
  SURFXML_BUFFER_SET(platform_version, "2");

  SURFXML_START_TAG(platform);

/*   <host id="host A" power="100000000.00"/> */
  SURFXML_BUFFER_SET(host_id, "host A");
  SURFXML_BUFFER_SET(host_power, "100000000.00");
  SURFXML_BUFFER_SET(host_availability, "1.0");
  SURFXML_BUFFER_SET(host_availability_file, "");
  A_surfxml_host_state = A_surfxml_host_state_ON;
  SURFXML_BUFFER_SET(host_state_file, "");
  SURFXML_BUFFER_SET(host_interference_send, "1.0");
  SURFXML_BUFFER_SET(host_interference_recv, "1.0");
  SURFXML_BUFFER_SET(host_interference_send_recv, "1.0");
  SURFXML_BUFFER_SET(host_max_outgoing_rate, "-1.0");

  SURFXML_START_TAG(host);
  SURFXML_END_TAG(host);

/*   <host id="host B" power="100000000.00"/> */
  SURFXML_BUFFER_SET(host_id, "host B");
  SURFXML_BUFFER_SET(host_power, "100000000.00");
  SURFXML_BUFFER_SET(host_availability, "1.0");
  SURFXML_BUFFER_SET(host_availability_file, "");
  A_surfxml_host_state = A_surfxml_host_state_ON;
  SURFXML_BUFFER_SET(host_state_file, "");
  SURFXML_BUFFER_SET(host_interference_send, "1.0");
  SURFXML_BUFFER_SET(host_interference_recv, "1.0");
  SURFXML_BUFFER_SET(host_interference_send_recv, "1.0");
  SURFXML_BUFFER_SET(host_max_outgoing_rate, "-1.0");

  SURFXML_START_TAG(host);
  SURFXML_END_TAG(host);

/*   <link id="LinkA" bandwidth="10000000.0" latency="0.2"/> */
  SURFXML_BUFFER_SET(link_id, "LinkA");
  SURFXML_BUFFER_SET(link_bandwidth, "10000000.0");
```

```
  SURFXML_BUFFER_SET(link_bandwidth_file, "");
  SURFXML_BUFFER_SET(link_latency, "0.2");
  SURFXML_BUFFER_SET(link_latency_file, "");
  A_surfxml_link_state = A_surfxml_link_state_ON;
  SURFXML_BUFFER_SET(link_state_file, "");
  A_surfxml_link_sharing_policy = A_surfxml_link_sharing_policy_SHARED;
  SURFXML_START_TAG(link);
  SURFXML_END_TAG(link);

/*   <route src="host A" dst="host B"><link:ctn id="LinkA"/></route> */
  SURFXML_BUFFER_SET(route_src, "host A");
  SURFXML_BUFFER_SET(route_dst, "host B");
  SURFXML_BUFFER_SET(route_impact_on_src, "0.0");
  SURFXML_BUFFER_SET(route_impact_on_dst, "0.0");
  SURFXML_BUFFER_SET(route_impact_on_src_with_other_recv, "0.0");
  SURFXML_BUFFER_SET(route_impact_on_dst_with_other_send, "0.0");

  SURFXML_START_TAG(route);

  SURFXML_BUFFER_SET(link_c_ctn_id, "LinkA");
  SURFXML_START_TAG(link_c_ctn);
  SURFXML_END_TAG(link_c_ctn);

  SURFXML_END_TAG(route);

/*   <route src="host B" dst="host A"><link:ctn id="LinkA"/></route> */
  SURFXML_BUFFER_SET(route_src, "host B");
  SURFXML_BUFFER_SET(route_dst, "host A");
  SURFXML_BUFFER_SET(route_impact_on_src, "0.0");
  SURFXML_BUFFER_SET(route_impact_on_dst, "0.0");
  SURFXML_BUFFER_SET(route_impact_on_src_with_other_recv, "0.0");
  SURFXML_BUFFER_SET(route_impact_on_dst_with_other_send, "0.0");

  SURFXML_START_TAG(route);

  SURFXML_BUFFER_SET(link_c_ctn_id, "LinkA");
  SURFXML_START_TAG(link_c_ctn);
  SURFXML_END_TAG(link_c_ctn);

  SURFXML_END_TAG(route);
/* </platform> */
  SURFXML_END_TAG(platform);

  free(surfxml_bufferstack);
  return 0;
}

static int surf_parse_bypass_application(void)
{
  static int AX_ptr;
  static int surfxml_bufferstack_size = 2048;

  /* allocating memory to the buffer, I think 2MB should be enough */
  surfxml_bufferstack = xbt_new0(char, surfxml_bufferstack_size);

  /* <platform> */
  SURFXML_BUFFER_SET(platform_version, "2");

  SURFXML_START_TAG(platform);

/*   <process host="host A" function="master"> */
  SURFXML_BUFFER_SET(process_host, "host A");
  SURFXML_BUFFER_SET(process_function, "master");
  SURFXML_BUFFER_SET(process_start_time, "-1.0");
  SURFXML_BUFFER_SET(process_kill_time, "-1.0");
  SURFXML_START_TAG(process);

/*      <argument value="20"/> */
```

```
  SURFXML_BUFFER_SET(argument_value, "20");
  SURFXML_START_TAG(argument);
  SURFXML_END_TAG(argument);

/*      <argument value="5000000"/> */
  SURFXML_BUFFER_SET(argument_value, "5000000");
  SURFXML_START_TAG(argument);
  SURFXML_END_TAG(argument);

/*      <argument value="100000"/> */
  SURFXML_BUFFER_SET(argument_value, "100000");
  SURFXML_START_TAG(argument);
  SURFXML_END_TAG(argument);

/*      <argument value="host B"/> */
  SURFXML_BUFFER_SET(argument_value, "host B");
  SURFXML_START_TAG(argument);
  SURFXML_END_TAG(argument);

/*   </process> */
  SURFXML_END_TAG(process);

/*   <process host="host B" function="slave"/> */
  SURFXML_BUFFER_SET(process_host, "host B");
  SURFXML_BUFFER_SET(process_function, "slave");
  SURFXML_BUFFER_SET(process_start_time, "-1.0");
  SURFXML_BUFFER_SET(process_kill_time, "-1.0");
  SURFXML_START_TAG(process);
  SURFXML_END_TAG(process);

/* </platform> */
  SURFXML_END_TAG(platform);

  free(surfxml_bufferstack);
  return 0;
}

int master(int argc, char *argv[]);
int slave(int argc, char *argv[]);
MSG_error_t test_all(void);

typedef enum {
  PORT_22 = 0,
  MAX_CHANNEL
} channel_t;

/** Emitter function  */
int master(int argc, char *argv[])
{
  int slaves_count = 0;
  m_host_t *slaves = NULL;
  m_task_t *todo = NULL;
  int number_of_tasks = 0;
  double task_comp_size = 0;
  double task_comm_size = 0;

  int i;

  xbt_assert1(sscanf(argv[1], "%d", &number_of_tasks),
              "Invalid argument %s\n", argv[1]);
  xbt_assert1(sscanf(argv[2], "%lg", &task_comp_size),
              "Invalid argument %s\n", argv[2]);
  xbt_assert1(sscanf(argv[3], "%lg", &task_comm_size),
              "Invalid argument %s\n", argv[3]);

  {           /*  Task creation */
    char sprintf_buffer[64];
```

```
    todo = xbt_new0(m_task_t, number_of_tasks);

    for (i = 0; i < number_of_tasks; i++) {
      sprintf(sprintf_buffer, "Task_%d", i);
      todo[i] =
        MSG_task_create(sprintf_buffer, task_comp_size, task_comm_size, NULL);
    }
  }

  {                              /* Process organisation */
    slaves_count = argc - 4;
    slaves = xbt_new0(m_host_t, slaves_count);

    for (i = 4; i < argc; i++) {
      slaves[i - 4] = MSG_get_host_by_name(argv[i]);
      if (slaves[i - 4] == NULL) {
        INFO1("Unknown host %s. Stopping Now! ", argv[i]);
        abort();
      }
    }
  }

  INFO1("Got %d slave(s) :", slaves_count);
  for (i = 0; i < slaves_count; i++)
    INFO1("\t %s", slaves[i]->name);

  INFO1("Got %d task to process :", number_of_tasks);

  for (i = 0; i < number_of_tasks; i++)
    INFO1("\t\"%s\"", todo[i]->name);

  for (i = 0; i < number_of_tasks; i++) {
    INFO2("Sending \"%s\" to \"%s\"",
          todo[i]->name, slaves[i % slaves_count]->name);
    if (MSG_host_self() == slaves[i % slaves_count]) {
      INFO0("Hey ! It's me ! :)");
    }
    MSG_task_put(todo[i], slaves[i % slaves_count], PORT_22);
    INFO0("Send completed");
  }

  INFO0
    ("All tasks have been dispatched. Let's tell everybody the computation is
over.");
  for (i = 0; i < slaves_count; i++)
    MSG_task_put(MSG_task_create("finalize", 0, 0, FINALIZE),
                 slaves[i], PORT_22);

  INFO0("Goodbye now!");
  free(slaves);
  free(todo);
  return 0;
}                                /* end_of_master */

/** Receiver function  */
int slave(int argc, char *argv[])
{
  INFO0("I'm a slave");
  while (1) {
    m_task_t task = NULL;
    int a;
    a = MSG_task_get(&(task), PORT_22);
    if (a == MSG_OK) {
      INFO1("Received \"%s\" ", MSG_task_get_name(task));
      if (MSG_task_get_data(task) == FINALIZE) {
        MSG_task_destroy(task);
        break;
      }
```

```
      INFO1("Processing \"%s\" ", MSG_task_get_name(task));
      MSG_task_execute(task);
      INFO1("\"%s\" done ", MSG_task_get_name(task));
      MSG_task_destroy(task);
    } else {
      INFO0("Hey ?! What's up ? ");
      xbt_assert0(0, "Unexpected behavior");
    }
  }
  INFO0("I'm done. See you!");
  return 0;
}                /* end_of_slave */

/** Test function */
MSG_error_t test_all(void)
{
  MSG_error_t res = MSG_OK;

  /*  Simulation setting */
  MSG_set_channel_number(MAX_CHANNEL);
  MSG_paje_output("msg_test.trace");
  surf_parse = surf_parse_bypass_platform;
  MSG_create_environment(NULL);

  /*   Application deployment */
  MSG_function_register("master", master);
  MSG_function_register("slave", slave);
  surf_parse = surf_parse_bypass_application;
  printf("teste");
  MSG_launch_application(NULL);

  res = MSG_main();

  INFO1("Simulation time %g", MSG_get_clock());
  return res;
}                                /* end_of_test_all */

/** Main function */
int main(int argc, char *argv[])
{
  MSG_error_t res = MSG_OK;

  MSG_global_init(&argc, argv);
  res = test_all();
  MSG_clean();

  if (res == MSG_OK)
    return 0;
  else
    return 1;
}                     /* end_of_main */
```

Masterslave.c

```
/*     Baseado no masterslave_bypass.c   */
#include <stdio.h>
#include "escalonamento.h" /* LIBTS */

XBT_LOG_NEW_DEFAULT_CATEGORY(msg_test,"Messages specific for this msg
example");
int flag_task[TCOUNT];
int atoi_task=9;
int s_flag;
int *p_flag_task=flag_task;

#include "masterslave.h"

int master(int argc, char *argv[]);
int slave(int argc, char *argv[]);
MSG_error_t test_all(void);
int escolhealg;

struct host_task{
       int flag[5];
       m_host_t my_slave[5];
       m_task_t my_tasks[5];
       } host_task;

/** Emitter function  */
// Gera as tarefas
int master(int argc, char *argv[])
{

  int slaves_count = 0;
  m_host_t *slaves = NULL;
  m_task_t *todo = NULL;
  int number_of_tasks = 0;
  double task_comp_size = 0;
  double task_comm_size = 0;
  double w_task_comp_size = 0;

double r_number[7]={0,
                                 15000000,
                                 -8000000,
                                 -10000000,
                                 5000000,
                                 12000000,
                                 1000000};

printf("\n\n\n\n\n\n");

int indice_tcs=0;
int i;

  xbt_assert1(sscanf(argv[1], "%d", &number_of_tasks),
              "Parametro invalido %s\n", argv[1]);
  xbt_assert1(sscanf(argv[2], "%lg", &task_comp_size),
              "Parametro invalido %s\n", argv[2]);
  xbt_assert1(sscanf(argv[3], "%lg", &task_comm_size),
              "Parametro invalido %s\n", argv[3]);

for (s_flag=0;s_flag<number_of_tasks;s_flag++)
  {
  flag_task[s_flag]=atoi_task;
  }

  {              /*  Task creation */
    char sprintf_buffer[64];
```

```
    todo = xbt_new0(m_task_t, number_of_tasks);

    for (i = 0; i < number_of_tasks; i++) {
      sprintf(sprintf_buffer, "%d", i);

      w_task_comp_size = task_comp_size + r_number[indice_tcs];

      indice_tcs++;
      if (indice_tcs > 6) indice_tcs=0;

      todo[i] = MSG_task_create(sprintf_buffer, w_task_comp_size,
task_comm_size, NULL);
    }
  }
  {
    slaves_count = argc - 4;
    slaves = xbt_new0(m_host_t, slaves_count);

    for (i = 4; i < argc; i++) {
      slaves[i - 4] = MSG_get_host_by_name(argv[i]);
      if (slaves[i - 4] == NULL) {
        INFO1("Host desconhecido %s. Stopping Now! ", argv[i]);
        abort();
      }
    }
  }

  INFO1("Got %d slave(s) :", slaves_count);

  for (i = 0; i < slaves_count; i++)
    INFO1("\t %s", slaves[i]->name);

  INFO1("Tem %d tarefas para processar:", number_of_tasks);

  escalonamento (todo, number_of_tasks, slaves, slaves_count); /* LIBTS */

  INFO0("Goodbye now!");
  free(slaves);
  free(todo);
  return 0;
}                               /* end_of_master */

/** Receiver function  */
int slave(int argc, char *argv[])
{
  m_host_t my_self = MSG_host_self();
  m_host_t *master = NULL;
  int cancelado = 0;
  INFO0("\tEu sou um slave");
  while (1) {
    m_task_t task = NULL;
    int a;

    a = MSG_task_get(&(task), PORT_22);
    if (a == MSG_OK) {
      INFO1("\tRecebido \"%s\" ", MSG_task_get_name(task));
         // set indicador de tarefas para slaves
         atoi_task=atoi(MSG_task_get_name(task));
         flag_task[atoi_task]=1;

         if (MSG_task_get_data(task) == FINALIZE) {
        MSG_task_destroy(task);
        break;

      } else if (!strcmp(MSG_task_get_name(task), "ask")) {
        INFO0("\tConfigurando Master");
        master = (m_host_t*)MSG_task_get_data(task);
```

```
      }
      INFO1("\tProcessando \"%s\" ", MSG_task_get_name(task));
      MSG_task_execute(task);
      if (MSG_task_get_remaining_computation(task) > 0) {
        INFO1("\t\"%s\" parada ", MSG_task_get_name(task));
        cancelado = 1;
      } else {
      int t_cancel=atoi_task=atoi(MSG_task_get_name(task));
        if (flag_task[t_cancel]==2)
         INFO1("\t\"%s\" cancelada ", MSG_task_get_name(task));
         else
                INFO1("\t\"%s\" concluida ", MSG_task_get_name(task));
        cancelado = 0;

         // reset indicador de tarefas para slaves
         atoi_task=atoi(MSG_task_get_name(task));
         flag_task[atoi_task]=0;
      }
      MSG_task_destroy(task);
      if (master != NULL && !cancelado) {
        task = MSG_task_create("idle", 0, 0, &my_self);
        INFO2("\tSending \"%s\" to \"%s\"",
          task->name, (*master)->name);
        MSG_task_put(task, *master, PORT_23);
      }
    } else {
      INFO0("Hey ?! What's up ? ");
      xbt_assert0(0, "Unexpected behavior");
    }
  }
  INFO0("\tTudo OK. See you!");
  return 0;
}                               /* end_of_slave */

/** Test function */
MSG_error_t test_all(void)
{
  MSG_error_t res = MSG_OK;

  /*  Simulation setting */
  MSG_set_channel_number(MAX_CHANNEL);
  MSG_create_environment ("plataform_test_2.xml");

  /*   Application deployment */
  MSG_function_register("master", master);
  MSG_function_register("slave", slave);
  MSG_launch_application("application_test_2.xml");

  res = MSG_main();

  INFO1("Tempo de Simulacao %g ", MSG_get_clock());
  return res;
}                               /* end_of_test_all */

/** Main function */
int main(int argc, char *argv[])
{

  MSG_error_t res = MSG_OK;
  MSG_global_init(&argc, argv);
  res = test_all();
  MSG_clean();

 if (res == MSG_OK)
    return 0;
 else
    return 1;
}                /* end_of_main */
```

Escalonamento.c

```
/* LIBTS - Biblioteca de escalonamento de tarefas */
/* Patrícia Batista Franco - aluna de mestrado UNESP */
#include <stdio.h>
#include <stdlib.h>
#include "msg/msg.h"
#include "surf/surfxml_parse.h"
#include "escalonamento.h"
#include "escalona_lifo.h"
#include "escalona_fifo.h"
#include "escalona_rr.h"
#include "escalona_sjf.h"
#include "escalona_wq.h"
#include "escalona_wqr.h"
#include "escalona_suff.h"
#include "escalona_xsuff.h"
#include "escalona_dyn.h"

void escalonamento (m_task_t *task, int number_of_tasks, m_host_t *slaves, int
slaves_count)
{
    int escolhealg = -1;

    printf ("\n \n \n \n \n");
    printf ("Digite um número para escolher o algoritmo de escalonamento: \n
\n");
    printf ("  1 -> First In First Out (FIFO) \n");
    printf ("  2 -> Last In First Out (LIFO) \n");
    printf ("  3 -> Round Robin (RR) \n");
    printf ("  4 -> Short Job First (SJF) \n");
    printf ("  5 -> Workqueue (WQ) \n");
    printf ("  6 -> Workqueue with Replication (WQR) \n");
    printf ("  7 -> Sufferage \n");
    printf ("  8 -> XSufferage \n");
    printf ("  9 -> Dynamic FPLTF \n");
    printf ("  0 -> finalizar \n \n");
    printf ("O número é: ");
    scanf ("%d", &escolhealg);
    switch (escolhealg) {
        case 0:
            printf ("\n FIM - SAIR \n");
             int i;
             for (i = 0; i < slaves_count; i++)
                    MSG_task_put(MSG_task_create("finaliza", 0, 0, FINALIZE),
                slaves[i], PORT_22);
            break;
        case 1:
            escalona_fifo (task, number_of_tasks, slaves, slaves_count);
            break;
        case 2:
            escalona_lifo (task, number_of_tasks, slaves, slaves_count);
            break;
        case 3:
            escalona_rr (task, number_of_tasks, slaves, slaves_count);
            break;
        case 4:
            escalona_sjf (task, number_of_tasks, slaves, slaves_count);
            break;
        case 5:
            escalona_wq (task, number_of_tasks, slaves, slaves_count);
            break;
        case 6:
            escalona_wqr (task, number_of_tasks, slaves, slaves_count);
            break;
        case 7:
```

```
            escalona_suff (task, number_of_tasks, slaves, slaves_count);
            break;
        case 8:
            escalona_xsuff (task, number_of_tasks, slaves, slaves_count);
            break;
        case 9:
            escalona_dyn (task, number_of_tasks, slaves, slaves_count);
            break;
    }
    printf("fim\n");
}
```

Escalona_suff.c

```
/* LIBTS - Biblioteca de escalonamento de tarefas */
/* Patrícia Batista Franco - aluna de mestrado UNESP */
/* Algoritmo Sufferage adaptado para o SimGrid */

#include <stdio.h>
#include <stdlib.h>
#include "msg/msg.h"
#include "surf/surfxml_parse.h"
#include "escalona_suff.h"

XBT_LOG_NEW_DEFAULT_CATEGORY(msg_suff,"Messages specific for this msg
example");

void escalona_suff (m_task_t *task, int number_of_tasks, m_host_t *slaves, int
slaves_count)
{
int i = 0;
int j = 0;
int k = 0;
m_task_t t;
m_task_t todo;

alloc_queue(number_of_tasks);
for(i=0;i<number_of_tasks;i++)
  push_queue(task[i]);
i=0;
while ((t=pop_queue())!=NULL){
  task[i++]=t;
  }
free_queue();

m_task_t task_slave[slaves_count];
for (i=0; i < slaves_count; i++)
  {
  task_slave[i] = NULL;
  }

float task_size;
float task_cost;
float host_speed;
//float host_load;
float w_ct;

struct w_slave
    {
    int s_flag      [slaves_count]; // maquina em uso
    int s_task      [slaves_count]; // qual tarefa esta na maquina
    float s_tba     [slaves_count]; // tba
    int s_num[slaves_count];
```

```
    float s_ct  [slaves_count];
    float s_tc      [slaves_count]; // task cost
    } w_slave;

struct t_task
    {
      int t_flag[number_of_tasks];
      float t_suff[number_of_tasks];
    } t_task;

for (i=0; i<slaves_count; i++)
      {
      w_slave.s_tba[i]=0;
      w_slave.s_flag[i]=0;
      w_slave.s_task[i]=0;
      w_slave.s_num[i]=i;
    }

for (i=0; i<number_of_tasks; i++)
    {
      t_task.t_flag[i] = 1;
      t_task.t_suff[i] = 0;
    }

float w_sufferage;
int h_task = 0;
int task_x;
extern int flag_task[14];

j=-1;
i=0;

int host_1;
float ct_1;
int host_2;
float ct_2;

h_task=1;

do
  {

  // Calcular ct da tarefa para cada host
  for (j=0;j<slaves_count; j++)
    {
    //host_load=rand()%2;
      task_size = MSG_task_get_compute_duration(task[i]);
      host_speed = MSG_get_host_speed(slaves[j]);
      task_cost=(task_size/host_speed)+(1-host_load);
      w_slave.s_tc[j]=task_cost;
      w_ct = w_slave.s_tba[j]+task_cost;
      w_slave.s_ct[j]=w_ct;
    }

// Encontra host com menor ct

host_1=0;
ct_1=w_slave.s_ct[0];
host_2=0;
ct_2=ct_1++;

for (j=0; j < slaves_count; j++)
      {
      if (w_slave.s_ct[j] < ct_1)
        {
        ct_2=ct_1;
        host_2=host_1;
        ct_1=w_slave.s_ct[j];
```

```
        host_1=j;
        }
      else
      if ((w_slave.s_ct[j] < ct_2) && (w_slave.s_ct[j] > ct_1))
        {
        ct_2=w_slave.s_ct[j];
        host_2=j;
        }
      }

  w_sufferage = w_slave.s_ct[host_2] - w_slave.s_ct[host_1];
  t_task.t_suff[i] = w_sufferage;

  //todo = suff_task_copy(task[i]);

  if (w_slave.s_flag[host_1] == 0) // Maquina livre
    {
    INFO2("\tEnviando \"%s\" para \"%s\"", task[i]->name, slaves[host_1]-
>name);
    if (MSG_host_self() == slaves[host_1 % slaves_count])
        {
        INFO0("\tHey ! It's me ! :)");
        }
      todo = suff_task_copy(task[i]);
      MSG_task_put(todo, slaves[host_1 % slaves_count], PORT_22);
      INFO0("\tCompletado o envio");

      w_slave.s_flag[host_1] = 1;
      t_task.t_flag[i] = 0;
      w_slave.s_task[host_1] = i;
      w_slave.s_tba[host_1] = w_slave.s_tba[host_1] + w_slave.s_tc[host_1];
      k = number_of_tasks;
      //free(todo);
    }
  else // Máquina não está livre
      {
    task_x=w_slave.s_task[host_1];
    //INFO3("\tComparando tarefas %d e %d em %s",
      //       task_x, i, slaves[host_1]->name);
      if (t_task.t_suff[task_x] <  t_task.t_suff[i]) // Tarefa em
processamento maior q a corrente
        {
        int t_cancel=w_slave.s_task[host_1];

        INFO2("\tCancelando tarefa %d em %s",t_cancel, slaves[host_1]->name);
        todo=NULL;
        t_task.t_flag[task_x]=1;
        w_slave.s_flag[host_1] = 0; //libera maquina
        flag_task[t_cancel]=2;
        i--;

      }
        else
      {
      k--;
      if (k==0)
       {
       w_slave.s_flag[host_1]=0;
       }
      }
       }

for (j=0; j<slaves_count; j++)

  {
  int sf = w_slave.s_task[j];
  if (flag_task[sf]==0)   // tarefa terminada
    {
```

```
       t_task.t_flag[sf]=0;
       w_slave.s_flag[j]=0;
       }
  }

         h_task=0;
         int repete = 2;
         do
              {
              i++;   //vai para a proxima tarefa
              if (i>(number_of_tasks-1))
                {
                i=0;
                repete--;
                }
              if (t_task.t_flag[i]==1) // tarefa existe
              h_task = 1;
              }
         while ((h_task == 0) & (repete != 0));
  } while (h_task == 1); // Continua enquanto houver tarefa

  INFO0
    ("\tAll tasks have been dispatched.Let's tell everybody the computation is
over.");
  for (i = 0; i < slaves_count; i++)
    MSG_task_put(MSG_task_create("Finaliza", 0, 0, FINALIZE),
                 slaves[i], PORT_22);
} // Fim

m_task_t suff_task_copy(m_task_t src) {
    m_task_t dest;
    dest = MSG_task_create(MSG_task_get_name(src),
       MSG_task_get_compute_duration(src),
        MSG_task_get_data_size(src), NULL);
    if (MSG_task_get_data(src) != NULL)
    {
        void *p = malloc((int)MSG_task_get_data_size(src));
        memcpy(p, MSG_task_get_data(dest), (int)MSG_task_get_data_size(src));
        MSG_task_set_data(dest, p);
    }
    return dest;
}
```