UNIVERSIDADE FEDERAL FLUMINENSE

JACQUES ALVES DA SILVA

Tolerância a Falhas para Aplicações Autônomas em Grades Computacionais

NITERÓI

2010

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

UNIVERSIDADE FEDERAL FLUMINENSE

JACQUES ALVES DA SILVA

Tolerância a Falhas para Aplicações Autônomas em Grades Computacionais

Tese de Doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Doutor. Área de concentração: Redes e Sistemas Distribuídos e Paralelos.

Orientador:
Eugene Francis Vinod Rebello

NITERÓI

2010

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

S586 Silva, Jacques Alves da.
Tolerância a falhas para aplicações autônomas em grades computacionais / Jacques Alves da Silva. – Niterói, RJ : [s.n.], 2010.
108 f.

Tese (Doutorado em Computação) - Universidade Federal Fluminense, 2010.
Orientador: Eugene Francis Vinod Rebello.

1. Tolerância a falha (Computação). 2. Sistema de computação em grade. 3. Ciência da computação. I. Título.

CDD 004.2

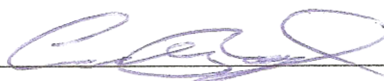
Tolerância a Falhas para Aplicações Autônomas em Grades Computacionais

Jacques Alves da Silva

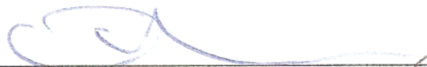
Tese de Doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Doutor.

Aprovada por:

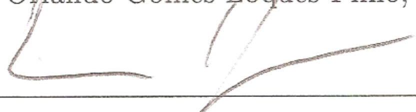
Prof. Eugene Francis Vinod Rebello, UFF (Presidente)



Prof. Célio Vinicius Neves de Albuquerque, UFF



Prof. Orlando Gomes Loques Filho, UFF



Prof. Bruno Richard Schulze, LNCC

Prof^{a}. Taisy Silva Weber, UFRGS

Niterói, 11 de junho de 2010.

Aos meus pais, Raimundo e Maurina.

À memória da minha irmã Jaqueline, que jamais será esquecida.

Agradecimentos

À Deus por me dar a força e a sabedoria necessárias para enfrentar todas as dificuldades surgidas durante este trabalho.

Aos meus pais, Raimundo e Maurina, pela paciência em compreender os momentos de ausência, em função do meu envolvimento com este trabalho, e por sempre me apoiarem e me ajudarem nos momentos mais difíceis.

Aos meus familiares e padrinhos pelo apoio e orações.

Ao meu orientador Vinod Rebello pelo incentivo, dedicação e ajuda para a realização deste trabalho.

Aos membros da banca pelas contribuições e sugestões fornecidas para melhorar a qualidade deste trabalho.

Aos amigos que sempre me ajudaram e me deram força para a realização deste trabalho. Os amigos do SGCLab: Alexandre, Aline, Ariel, Carlos Henrique, Carolina Cunha, Daniel Marques, Daniel Souza, Eduardo, Felipe, Fernanda, Henrique, Idalmis e Sean. Os amigos de sala: Diego, Juliana, Juliano e Luciene. Os amigos da sala vizinha: Adria, Anand, Janine, Luciana Brugiolo, Mário, Renatha e Puca. E os amigos: Aletéia, Augusto, Carlão, Clayton, Copetti, Cristiane, Daniela Vianna, Deolinda, Edgar, Flávio, Johnny, Jonivan, Kennedy, Leonardo Motta, Luciana Pessoa, Maurício Guedes, Nuba, Rodrigo Toso, Sandoval, Sérgio, Stênio, Synara, Tiago Neves, Vinícius, Viviane Thomé e tantos outros que contribuíram de alguma forma com este trabalho.

Aos funcionários Carlos e Rafael do suporte e as secretárias Angela, Izabela, Maria, Teresa e Viviane pelos auxílios prestados.

À CAPES por financiar parte do meu doutorado.

Resumo

Desenvolver aplicações capazes de executar eficientemente em ambientes de Grades Computacionais é extremamente difícil para usuários inexperientes. Os recursos distribuídos são tipicamente heterogêneos, compartilhados e são oferecidos sem qualquer garantia de desempenho ou disponibilidade. Sistemas capazes de adaptar a execução de uma aplicação para as características dinâmicas do ambiente grade são essenciais. O Sistema de Gerenciamento de Aplicações (SGA) EasyGrid transforma as aplicações MPI baseadas em *clusters* (desenvolvidas para ambientes homogêneos e estáveis) em aplicações autônomas capazes de executar de forma eficiente e robusta no ambiente grade. O SGA EasyGrid transforma as aplicações paralelas tradicionais em aplicações *system-aware*, onde a própria aplicação se torna responsável pela sua gerência e busca se auto-adaptar às mudanças dinâmicas ocorridas no ambiente. Através do SGA EasyGrid, esta tese propõe uma estratégia para dotar as aplicações MPI autônomas com a propriedade de auto-recuperação (*self-healing*) e, assim, serem capazes de resistir a múltiplas falhas simultâneas de colapso (*crash*) de processos e/ou processadores. Este trabalho apresenta um modelo de tolerância a falhas; seus mecanismos; a integração destes mecanismos com as outras funcionalidades do SGA EasyGrid; e uma avaliação da eficácia da proposta. O custo de intrusão extremamente baixo da solução proposta agora pode facilitar a aceitação de técnicas de tolerância a falhas em aplicações de alto desempenho de larga escala.

Palavras-chave: Tolerância a Falhas, Aplicações MPI Autônomas, Grades Computacionais

Abstract

Writing applications capable of executing efficiently in Grids is extremely difficult and tedious for inexperienced users. The distributed resources are typically heterogeneous, non-dedicated, and are offered without any performance or availability guarantees. Systems capable of adapting the execution of an application to the dynamic characteristics of the Grid are essential. The EasyGrid Application Management System (AMS) transforms cluster-based MPI applications (designed for homogeneous, stable environments) into autonomic ones capable of executing robustly and efficiently in Grids. The EasyGrid AMS transforms traditional parallel applications in system-aware applications, in which this application itself becomes responsible for its own management and attempts to self-adapt according to the dynamic changes of the environment. Through the EasyGrid AMS, this thesis proposes a strategy to endow autonomic MPI applications with the property of self-healing and thus capable of withstanding multiple simultaneous crash faults of processes and/or processors. This work presents a fault tolerance model; its mechanisms; the integration of these mechanisms with other EasyGrid AMS functionalities; and an evaluation of the effectiveness of the proposal. The extremely low intrusion cost of the proposed solution might now facilitate acceptance of fault tolerance techniques in large scale high performance application.

Keywords: Fault tolerance, Autonomic MPI Applications, Computational Grids

Palavras-chave

1. Tolerância a Falhas
2. Aplicações MPI autônomas
3. Grades Computacionais

Lista de Siglas e Acrônimos

BLCR : Berkeley Laboratory Checkpoint/Restart;
GAD : Grafo Acíclico Direcionado;
GG : Gerenciador Global;
GM : Gerenciador da Máquina;
GS : Gerenciador do Site;
LAM : *Local Area Multicomputer*;
MPI : *Message Passing Interface*;
MTBF : *Mean Time Between Failure*;
SGA : Sistema de Gerenciamento de Aplicações;
TF : Tolerância a Falhas;
UFF : Universidade Federal Fluminense;

Sumário

Lista de Figuras xi

Lista de Tabelas xiii

1 Introdução 1

1.1 Contexto 2

1.2 Objetivos 4

1.3 Contribuições 5

1.4 Organização 7

2 Conceitos 8

2.1 Dependabilidade 9

2.2 Tolerância a Falhas 12

2.2.1 Definição de Falha, Erro e Defeito 12

2.2.2 Características dos Modelos de Falhas 13

2.2.3 Mecanismos de Tolerância a Falhas 16

2.3 Computação Autônoma 24

2.4 Resumo 26

3 Trabalhos Relacionados 27

3.1 Versões de MPI Tolerante a Falhas 29

3.2 Sistema de Gerenciamento de Aplicações EasyGrid 35

3.2.1 Modelo de Execução 36

3.2.2 Modelo da Aplicação 37
3.2.3 Modelo Arquitetural 38
3.2.4 Gerenciamento dos processos da aplicação no SGA 39
3.3 Resumo 42

4 Tolerância a Falhas no SGA EasyGrid 43

4.1 Modelo de Falhas Utilizado 43
4.2 Mecanismos de Tolerância a Falhas no SGA EasyGrid 45
4.2.1 Tolerância a Falhas para Processos da Aplicação 46
4.2.2 Tolerância a Falhas para Gerenciadores da Máquina 49
4.2.3 Tolerância a Falhas para Gerenciadores do *Site* 54
4.2.4 Tolerância a Falhas para o Gerenciador Global 57
4.3 Estudo de Caso: Tolerância a Falhas em Aplicações MPI 59
4.3.1 Tolerância a Falhas para Processos da Aplicação no MPI 60
4.3.2 Tolerância a Falhas para Gerenciadores da Máquina no MPI 61
4.3.3 Tolerância a Falhas para Gerenciadores do *Site* no MPI 64
4.3.4 Tolerância a Falhas para o Gerenciador Global no MPI 64
4.4 Resumo 65

5 Experimentos Computacionais 66

5.1 Falhas nos Processos da Aplicação 69
5.2 Falhas nos Processos Gerenciadores GS e GM 74
5.2.1 SGA EasyGrid com Tolerância a Falhas nos Gerenciadores GS e GM 74
5.2.2 SGA EasyGrid com Tolerância a Falhas em todos os Gerenciadores 81
5.3 Falhas no Processo GG 87
5.4 Resumo 94

6 Conclusões e Trabalhos Futuros 95

6.1 Trabalhos Futuros 98

Referências **100**

Lista de Figuras

2.1 Falha, erro e defeito no modelo de 3 universos (WEBER, 2002). 12

3.1 Modelo de execução 1PProc (SENA, 2008). 29

3.2 Modelo de execução 1PTask (SENA, 2008). 36

3.3 Exemplo de tarefa do modelo 1PTask (SENA, 2008). 37

3.4 Exemplo de GAD (SENA, 2008). 38

3.5 A hierarquia do SGA EasyGrid. 40

3.6 Arquitetura em camadas do SGA EasyGrid. 41

4.1 Comunicação de um processo da aplicação com o seu GM. 47

4.2 Comunicação de um processo GM com o seu GS. 49

4.3 Comunicação de um processo GS com o GG. 55

4.4 Respostas da *thread* com o comando *mpitask*. 63

5.1 Iterações do algoritmo *ring* em 4 processadores (SENA, 2008). 68

5.2 Iterações do novo algoritmo *ring* com $W = 4$ e $P = 4$ (SENA, 2008). 68

5.3 Sobrecarga do SGA no caso de falha irrecuperável na máquina do GS do Site 1. 78

5.4 Sobrecarga do SGA no caso de falhas irrecuperáveis em 6 máquinas com GM. 79

5.5 Sobrecarga do SGA no caso de falha recuperável no processo GS do Site 1. 79

5.6 Sobrecarga do SGA no caso de falhas recuperáveis em 6 processos GM's. . 80

5.7 Sobrecarga do SGA no caso de falha irrecuperável na máquina do GS do Site 1. 85

5.8 Sobrecarga do SGA no caso de falhas irrecuperáveis em 6 máquinas com GM. 85

5.9 Sobrecarga do SGA no caso de falha recuperável no processo GS do Site 1. 86

5.10 Sobrecarga do SGA no caso de falhas recuperáveis em 6 processos GM's. . 86

5.11 AMS ring com N=250000 e intervalos entre *checkpoints* iguais a 60 e 30 segundos. 90

5.12 AMS ring com N=500000 e intervalos entre *checkpoints* iguais a 60 e 30 segundos. 91

5.13 AMS ring com N=1000000 e intervalos entre *checkpoints* iguais a 60 e 30 segundos. 91

Lista de Tabelas

5.1 Comparação entre os tempos de execução real (R) e o limite inferior estimado (E(pf)), em segundos. A terceira coluna mostra a sobrecarga relativa a execução ideal (S). 71

5.2 Tempos de execução realizados pelo MTrab com o SGA EasyGrid (em segundos) através dos cenários de falhas e a sobrecarga (S) produzida na falha depois de 100 segundos em relação a de 50 segundos 73

5.3 Tempo de execução médio para a aplicação MTrab variando o número de processos e granularidades sobre diferentes cenários: Caso 0 - LAM/MPI puro; Caso 1 - SGA sem código de TF. 75

5.4 Tempo de execução médio para a aplicação MTrab variando o número de processos e a granularidade sobre diferentes cenários: Caso 1 - SGA sem código de TF; Caso 2 - SGA com código de TF executado sem falhas; Caso 3 - Falhas Irrecuperáveis em 6 máquinas com GM; Caso 4 - Falhas Recuperáveis em 6 processos GM's; Caso 5 - Falha Irrecuperável na máquina do GS do *Site* 1; Caso 6 - Falha Recuperável no processo GS do *Site* 1. 76

5.5 Tempo de execução médio para a aplicação MTrab variando o número de processos e granularidades sobre diferentes cenários: Caso 0 - LAM/MPI puro; Caso 1 - SGA sem código de TF. 82

5.6 Tempo de execução médio para a aplicação MTrab variando o número de processos e a granularidade sobre diferentes cenários: Caso 1 - SGA sem código de TF; Caso 2 - SGA com código de TF executado sem falhas; Caso 3 - Falhas Irrecuperáveis em 6 máquinas com GM; Caso 4 - Falhas Recuperáveis em 6 processos GM's; Caso 5 - Falha Irrecuperável na máquina do GS do *Site* 1; Caso 6 - Falha Recuperável no processo GS do *Site* 1. 83

5.7 Tempo de execução médio em segundos para a aplicação N-corpos variando o número de processadores e o intervalo dos *checkpoints*. 89

5.8 Tempo de execução médio em segundos para a recuperação da aplicação N-corpos variando o número de processadores. 92

5.9 Sobrecarga total dos mecanismos de TF incluindo a recuperação da aplicação N-corpos variando o número de processadores. 94

Capítulo 1

Introdução

A popularidade da Internet aliada a disponibilidade de computadores poderosos e redes de alta velocidade propiciaram o surgimento de uma nova forma de compartilhamento de recursos, conhecido popularmente como **Grades Computacionais** (FOSTER; KESSELMAN, 1999; FOSTER; KESSELMAN; TUECKE, 2001). As grades computacionais visam agregar um número significante de recursos geograficamente distribuídos para fornecer poder computacional suficiente e de baixo custo para uma grande variedade de aplicações, como por exemplo, estudo de genomas, modelagem climática, pesquisa sobre terremotos, entre outras. Elas podem ser vistas como supercomputadores baseados em múltiplos *clusters* de computadores independentes compostos de recursos especializados e/ou comuns (*off-the-shelf* - por exemplo, os computadores pessoais) localizados em diferentes domínios administrativos. O uso das grades pode ser na forma dedicada, onde os recursos são utilizados exclusivamente pelos seus *jobs*, ou na forma compartilhada, onde os *jobs* da grade só aproveitam a capacidade ociosa dos recursos. A política de uso das grades deste último tipo, que são conhecidas como **Grades Computacionais Oportunistas** (CARDOSO; COSTA, 2010), permitem agregar ainda mais recursos e oferecer as aplicações maior poder computacional. Nas duas formas de utilização, as melhorias nos tempos de execução das aplicações são conseguidas por uma utilização mais eficiente dos recursos disponíveis.

As grades computacionais então permitem os pesquisadores desenvolverem aplicações com demandas computacionais maiores do que as oferecidas pelo seu próprio sistema local. Mas, ao contrário dos supercomputadores, nos ambientes grade os custos de aquisição, operação e manutenção são divididos entre os colaboradores, o que torna este tipo de computação mais acessível para a maioria dos cientistas e companhias. Entretanto, o desenvolvimento de aplicações que executem de forma eficiente neste ambiente é uma

tarefa mais difícil. Ao contrário dos *clusters* de computadores, que são na sua maioria estáveis e dedicados, as grades computacionais são ambientes dinâmicos, compartilhados e instáveis, onde o poder computacional disponível para um usuário se modifica com a entrada e saída de recursos, de *jobs* locais ou outros *jobs* da grade.

Dada estas características e o tamanho dos ambientes grades, falhas podem ocorrer com mais probabilidade durante a execução das aplicações neste ambiente. Consequentemente, caso não existam mecanismos de tolerância a falhas, a aplicação paralela pode falhar por completo devido a falha de um único recurso. Desta maneira, o processamento realizado nos outros recursos computacionais alocados para a aplicação seriam desperdiçados. Assim, torna-se necessário que o sistema ou a aplicação seja tolerante a falhas, a fim de que se utilize os recursos disponíveis de forma mais eficiente.

Embora o tempo médio entre defeitos (*Mean Time Between Failure* - MTBF) de cada componente (ou seja, processadores, discos, memórias, redes, entre outros) possa ser alto, um sistema composto por um grande número de componentes pode efetivamente falhar mais frequentemente. Segundo Reed et al. em (REED; LU; MENDES, 2006), um sistema composto por 64000 nós, cada nó com um MTBF de mais de 100 anos (10^6h), possui um tempo médio sem falhas menor do que 24h para o sistema como um todo. Muitas aplicações de *e-Science* requerem dias ou mesmo meses para completar sua execução, o que pode ser significativamente maior do que o MTBF de um ambiente grade de grande porte. Assim, a falta de um sistema tolerante a falhas não somente degrada o desempenho da aplicação, visto que essas deverão ser reiniciadas, como também provoca a perda significativa de poder computacional.

1.1 Contexto

Com o objetivo de simplificar a implementação de aplicações para grades computacionais, torna-se necessário o desenvolvimento de uma camada de *software*, chamada *middleware*, para esconder a complexidade deste ambiente. O **Globus Toolkit** (FOSTER; KESSELMAN, 1997; Globus Toolkit, 2010) é um conjunto de ferramentas básicas de *software* de código aberto comumente usado para construir um ambiente grade. Este conjunto de ferramentas inclui serviços e bibliotecas que permitem a autenticação e autorização; a descoberta, monitoramento e alocação de recursos; a transferência de dados; e a submissão de aplicações nos recursos da grade (FERREIRA et al., 2003). O Globus Toolkit fornece os meios necessários para os usuários finais resolverem os problemas operacionais encontra-

dos quando utilizando os recursos distribuídos que não estão no seu controle direto (Globus Toolkit, 2010). Porém, é deixado ao próprio usuário final ou uma outra camada de *software* tratar as falhas da aplicação na grade.

Para aplicações científicas, a biblioteca MPI (*Message Passing Interface*) (MPI, 2010) é o mais popular modelo de programação paralela (REED; LU; MENDES, 2006). Por este motivo, para a execução eficiente de aplicações paralelas nas grades computacionais, especialmente as aplicações fortemente acopladas, foi escolhida nesta tese a utilização de aplicações desenvolvidas utilizando a biblioteca MPI, que se tornou o padrão *de facto* de troca de mensagens (FOSTER, 1995). A interface MPI é uma biblioteca padronizada e portável de rotinas para troca de mensagens. A biblioteca MPI foi inicialmente desenvolvida para ser utilizada em ambientes estáticos, homogêneos e menos propensos a falhas, tais como *clusters* computacionais dedicados. Por isso, a biblioteca MPI não tem suporte nativo de tolerância a falhas. Atualmente, existe uma grande variedade de aplicações de alto desempenho baseadas em MPI sendo executadas em *clusters*, que poderiam se beneficiar de ainda mais poder computacional quando executadas em uma grade. De forma ideal, do ponto de vista da comunidade, seria melhor se estas mesmas aplicações, que foram desenvolvidas sem considerações para escalonamento dinâmico e tolerância a falhas, pudessem ser capazes de executar com a mesma eficiência em um ambiente grade sem a necessidade de serem reescritas pelo programador ou de conhecimento especial da parte do usuário quando configurando a aplicação para a execução.

Em ambientes de grades computacionais é comum a utilização de um Sistema Gerenciador de Recursos (SGR) para controlar a execução das aplicações. Nesta visão, centrada no sistema (*system-centric*), os serviços fornecidos pelo *middleware* são pré-instalados nos recursos da grade para que a utilização destes recursos possa ser maximizada. O SGR deve considerar a autonomia dos *sites* que compõem a grade, a heterogeneidade desses sistemas, as diferentes políticas de segurança e escalonamento adotados. Consequentemente, o gerencimento neste ambiente torna-se mais complexo do que em ambientes dedicados, tal como *clusters*, e aumenta em proporção quando a própria grade cresce em tamanho. Os SGR's continuamente monitoram e analisam informações do sistema para selecionar os recursos mais adequados à execução de cada aplicação. Contudo, para aplicações paralelas, esta escolha de recursos com base somente em informações do sistema pode não ser suficiente. As tarefas da aplicação paralela devem ser escalonadas considerando as características de cada aplicação, tal como a dependência entre as tarefas.

Devido as características e variedade das aplicações paralelas e o dinamismo e hete-

rogeneidade das grades, torna-se clara a necessidade de sistemas capazes de se adaptar a execução destas aplicações. Em 2001, a IBM introduziu o conceito de computação autônoma (*autonomic computing*) (IBM Research, 2010) para descrever uma nova estratégia para o desenvolvimento de sistemas de computadores distribuídos complexos. A computação autônoma (STERRITT et al., 2005) é inspirado no sistema nervoso humano e foca no desenvolvimento de sistemas e aplicações com a capacidade de se autogerenciar (*self-management*), ou seja, o próprio sistema ou aplicação se regula de acordo com as mudanças do ambiente.

Contra o paradigma atualmente seguido na computação em grades, e para reduzir a complexidade de gerenciamento das grades computacionais, foi adotado em (BOERES; REBELLO, 2004) a filosofia de um Sistema de Gerenciamento da Aplicação (SGA). Nesta filosofia, centrada na aplicação (*application-centric*), a própria aplicação se torna responsável pela sua gerência e busca se autoadaptar às mudanças dinâmicas ocorridas no ambiente grade. O SGA EasyGrid (VIANNA, 2005; NASCIMENTO et al., 2005, 2007; SENA et al., 2007; SILVA; REBELLO, 2007; ARAÚJO, 2008; NASCIMENTO, 2008; SENA, 2008; NASCIMENTO et al., 2008; SENA et al., 2008; NASCIMENTO et al., 2009) é uma prova de conceito, que é capaz de transformar aplicações MPI (*Message Passing Interface*) (MPI, 2010), tanto fracamente quanto fortemente acopladas, desenvolvidas para *clusters* (ambientes homogêneos e estáveis) em aplicações autônomas eficientes e robustas nos ambientes grade. O SGA EasyGrid é um *middleware* de gerenciamento no nível da aplicação e específico para cada aplicação, sendo embutido automaticamente no programa MPI em tempo de compilação sem esforço de programação por conta do desenvolvedor da aplicação.

1.2 Objetivos

Esta tese tem como objetivo central aplicar e desenvolver mecanismos de tolerância a falhas que possibilitem as aplicações MPI, utilizando o *middleware* SGA EasyGrid, serem executadas com sucesso em ambientes de grades computacionais na presença de falhas. Desta forma, estas aplicações desenvolvidas para serem executadas em *clusters* passam a poder executar em grades, através do SGA EasyGrid, sem alteração do código fonte. Os mecanismos de tolerância a falhas visam permitir que as aplicações se recuperem de falhas de processos e/ou processadores, o que inclui a morte de um processo e a indisponibilidade de um recurso durante a execução da aplicação. Além disso, supõe-se que um ou mais processos e/ou recursos podem falhar a qualquer momento. Assim, em caso de falha de

um recurso ou de um processo, pretende-se evitar que a aplicação precise recomeçar a sua execução desde o início, o que contribui para uma utilização mais eficiente dos recursos disponíveis.

Para atingir seu objetivo central, este trabalho propõe estratégias de tolerância a falhas que melhor aproveitam as características do *middleware* SGA EasyGrid. Uma característica importante deste SGA é o seu modelo de execução para ambientes grades, apresentado em (SENA et al., 2007; SENA, 2008), onde os processos não são criados todos na inicialização. Por consequência, o *middleware* é baseado em implementações MPI que suportam criação dinâmica de processos, tal como LAM/MPI (LAM Team, 2010). Outra característica do SGA EasyGrid é que por razões de portabilidade, nenhuma modificação é feita na distribuição padrão ou instalação do LAM/MPI e nenhum *software* adicional, além do Globus Toolkit 2.x (Globus Toolkit, 2010) e certificados X.509 para recursos e usuários, é requerido para a execução de aplicações EasyGrid autônomas nas grades. Porém, as aplicações EasyGrid autônomas podem executar sem o Globus Toolkit e os certificados X.509 em qualquer ambiente *cluster* em que as aplicações LAM/MPI já executam. Assim, todos os mecanismos de tolerância a falhas desenvolvidos são incluídos exclusivamente dentro do SGA EasyGrid no nível da aplicação. Ou seja, nem o LAM/MPI e nem o Globus Toolkit 2.x foram modificados para interagir com os mecanismos de tolerância a falhas. Outro objetivo é integrar eficientemente as estratégias de escalonamento e tolerância a falhas dentro de aplicações de alto desempenho, de forma transparente ao programador e aproveitando da melhor forma possível o modelo de execução utilizado, e avaliar seu desempenho em ambientes grades.

1.3 Contribuições

A principal contribuição desta tese é propor e avaliar uma estratégia de tolerância a falhas que beneficie cientistas interessados em executar aplicações de alto desempenho em ambientes de grades computacionais. O ideal é que as mesmas aplicações desenvolvidas para executarem em *clusters*, que normalmente não possuem considerações para escalonamento dinâmico e tolerância a falhas, possam executar com igual eficiência em um ambiente grade sem a necessidade de serem reescritas pelo desenvolvedor. Entretanto, existem algumas restrições impostas pelos próprios ambientes grades, que interferem no desenvolvimento dos mecanismos de tolerância a falhas neste tipo de ambiente. Uma dificuldade é que os recursos remotos pertencem a diferentes domínios administrativos

e, consequentemente, estão sujeitos a diferentes políticas de utilização. Desta forma, se torna complicada a instalação, no nível do sistema e em todas as máquinas da grade, de *softwares* customizados pelo usuário. Outra dificuldade para os mecanismos de tolerância a falhas no ambiente grade é que neste tipo de ambiente geralmente não existe um sistema de arquivo global e seguro para a gravação de todos os arquivos de *checkpoint* de uma aplicação paralela, o que é um fator limitante para a utilização desta técnica.

Para a realização desta principal contribuição, é utilizado o *middleware* SGA EasyGrid para transformar as aplicações paralelas em aplicações autônomas. Este trabalho contribui com o *middleware* SGA EasyGrid ao adicionar a camada de tolerância a falhas na arquitetura do SGA. Com isso, foi acrescentada robustez aos processos da aplicação do usuário e aos próprios gerenciadores do SGA, o que é uma característica essencial para a execução das aplicações nos ambientes grades. Por ter sido incluído dentro do SGA EasyGrid, os mecanismos de tolerância a falhas ficaram restritos a sua atuação no nível da aplicação.

Em ambientes de grades computacionais existem diferentes tipos de falhas, tais como falhas de recurso e de processos. Um exemplo típico de falha de processo é quando o administrador de um dos domínios decide limitar a utilização dos seus recursos por abortando os processos em execução. Tipicamente, essas falhas são tratadas como o mesmo tipo de falha necessitando de uma única estratégia. Neste trabalho, as falhas de processos incluem não só as falhas dos processos da aplicação, mas também possíveis falhas dos gerenciadores do SGA. Para uma menor sobrecarga dos mecanismos de tolerância a falhas, o SGA EasyGrid adota diferentes estratégias para o tratamento de cada tipo de falha.

Aproveitando-se do modelo de execução utilizado neste trabalho, apresentado em (SENA et al., 2007; SENA, 2008) e explicado com mais detalhes na Seção 3.2.1, a estratégia de tolerância a falhas proposta usa a técnica de *log* de mensagens para os processos da aplicação e a técnica de *checkpoint* no nível da aplicação para os processos gerenciadores. O SGA EasyGrid possui uma hierarquia de gerenciadores e, na abordagem implementada, os gerenciadores de nível acima possuem as informações necessárias para a recuperação dos gerenciadores de nível inferior. Assim, o *checkpoint* em memória volátil é realizado pelo gerenciador de nível acima. Já para o caso do gerenciador no topo da hierarquia, o *checkpoint* no nível da aplicação é realizado pelo próprio gerenciador e gravado em um meio de armazenamento seguro. Vale destacar que a partir do arquivo gerado é possível recuperar o estado de todos os gerenciadores do SGA EasyGrid. Desta forma, no meca-

nismo proposto, não é necessário realizar *checkpoints* de todos os gerenciadores de forma a gravar um estado consistente entre eles e nem se preocupar em gargalos na gravação dos diferentes arquivos em um meio de armazenamento seguro. Além disso, a solução proposta é completa, pois os mecanismos de tolerância a falhas possuem os estágios de detecção, localização, isolamento e recuperação de múltiplas falhas simultâneas.

Para que os mecanismos de tolerância a falhas possam ser usados nas aplicações de alto desempenho é necessário que a solucão proposta tenha uma baixa intrusão, ou seja, os mecanismos devem ter pouco impacto na eficiência das aplicações. Desta forma, esta tese também realiza a integração dos mecanismos de tolerância a falhas propostos com a camada de escalonamento dinâmico pertencente ao SGA EasyGrid. O objetivo desta integração é reduzir da melhor forma possível a sobrecarga adicionada pelos mecanismos de tolerância a falhas, o que contribui com o desempenho da aplicação paralela robusta.

1.4 Organização

Esta tese está dividida da seguinte forma. Primeiramente, no Capítulo 2 serão explicados os conceitos e esclarecidos os termos utilizados neste trabalho no contexto de tolerância a falhas, dependabilidade e computação autônoma. No Capítulo 3 é descrita as características da biblioteca de troca de mensagens MPI e apresentada as estratégias adotadas por algumas versões de MPI tolerante a falhas. Além disso, é apresentada uma descrição do Sistema de Gerenciamento de Aplicações (SGA) EasyGrid. O Capítulo 4 descreve os mecanismos de tolerância a falhas propostos e implementados no SGA EasyGrid e algumas das dificuldades enfrentadas no processo. Em seguida, o Capítulo 5 apresenta uma avaliação baseada em experimentos realizados para verificar a habilidade de uma aplicação EasyGrid de terminar eficientemente sua execução mesmo na presença de falhas de processos e/ou processadores e para analisar as sobrecargas do SGA EasyGrid. Por fim, o Capítulo 6 apresenta de forma resumida as vantagens de se usar o SGA EasyGrid e indica alguns trabalhos que podem ser explorados a partir desta tese.

Capítulo 2

Conceitos

Este capítulo almeja introduzir alguns conceitos relacionados a tolerância a falhas, dependabilidade e computação autônoma. O objetivo é definir a terminologia usada neste trabalho, já que existem vários trabalhos na literatura e que utilizam os mesmos termos com mais de um significado ou termos diferentes para designar a mesma propriedade ou conceito. Por exemplo, alguns autores nacionais da área traduzem as palavras inglesas *failure* como falha e *fault* como falta, porém, como no caso desta tese, outros definem como defeito e falha, respectivamente.

Os sistemas computacionais são desenvolvidos para atender um conjunto de requisitos que satisfaçam uma necessidade do usuário do sistema. Um requisito importante em alguns sistemas é que eles sejam altamente confiáveis (*dependable*), principalmente em sistemas de tempo real críticos, tais como sistemas militares de defesa, controle de tráfego aéreo e ferroviário, entre outros. A técnica de tolerância a falhas é considerada um meio de se atingir dependabilidade (*dependability*) (HEIMERDINGER; WEINSTOCK, 1992).

O conceito de tolerância a falhas foi originalmente apresentado por Avizienis em 1967 (AVIZIENIS, 1998), que usou a seguinte definição: *Um sistema é tolerante a falhas se seus programas podem ser apropriadamente executados apesar da ocorrência de falhas lógicas.* Segundo Heimerdinger e Weinstock em (HEIMERDINGER; WEINSTOCK, 1992), *falhas lógicas* são aquelas que ocorrem quando os recursos adequados estão disponíveis, mas o sistema não se comporta de acordo com a sua especificação, por exemplo, quando o sistema não realiza uma função esperada. Estas falhas podem ser o resultado de uma modelagem ou implementação incorreta e podem ocorrer em *hardware* e *software*. Porém, as *falhas físicas*, que ocorrem quando o *hardware* quebra ou uma mutação ocorre no executável do *software*, também devem ser consideradas. Além disso, aplicações executando

em sistemas distribuídos como grades computacionais, por exemplo, são susceptíveis a uma outra condição de falha, as falhas por esgotamento de recursos (*resource depletion faults*), em que uma parte da aplicação é incapaz de obter os recursos necessários para executar suas tarefas. Por isso, existem muitas aplicações atualmente necessitando um tempo de execução maior do que o tempo médio entre defeitos (*Mean Time Between Failure* - MTBF) de vários ambientes.

Segundo (GÄRTNER, 1999), os modelos de sistemas distribuídos existentes diferem em relação a noção de tempo real, que é expressa de acordo com algumas suposições sobre velocidade de execução dos processos e retardo na entrega das mensagens. Em sistemas **síncronos** são assumidos limites em tempo real para transmissão das mensagens e tempo de resposta dos processos. Se nenhuma suposição de tempo é feita, então o sistema é **assíncrono**. Já os modelos intermediários que possuem limites de tempo com um grau de variação são normalmente chamados de **parcialmente síncronos**. Vale ressaltar que todo sistema é assíncrono. Dado que mesmo um sistema, em que os processos executam em iterações sincronizadas e a entrega das mensagens são instantâneas, satisfaz a definição de um sistema assíncrono (bem como a definição de sistema síncrono). Assim, um protocolo modelado para ser usado em um sistema assíncrono pode ser usado em qualquer sistema distribuído.

2.1 Dependabilidade

Dependabilidade de um sistema computacional é a capacidade de fornecer serviços que se justificam confiáveis (AVIZIENIS; LAPRIE; RANDELL, 2001). O serviço fornecido por um sistema é seu comportamento como percebido pelo seu usuário. O usuário pode ser visto como um outro sistema que interage com o primeiro através da interface do serviço. A função de um sistema é o que o sistema destina-se a ser e é descrito pela especificação do sistema. Um serviço é fornecido corretamente quando o serviço implementa a função especificada. Porém, existe a possibilidade de que um sistema possa falhar devido ao não cumprimento da sua especificação por alguma razão ou porque a especificação não descreve adequadamente as suas funções.

Segundo Avizienis, Laprie e Randell em (AVIZIENIS; LAPRIE; RANDELL, 2001), o conceito de dependabilidade integra os seguintes atributos:

- Disponibilidade: o sistema está pronto para executar um serviço corretamente;

- Confiabilidade: a continuidade do serviço fornecido de forma correta;
- Segurança (*safety*) de funcionamento: ausência de danos a outros sistemas e pessoas que dependam do sistema;
- Confidencialidade: ausência de acesso a informações não autorizadas;
- Integridade: ausência de alterações impróprias no estado do sistema;
- Manutenabilidade: capacidade de reparos e modificações serem submetidos no sistema.

O atributo de segurança (*security*) é a existência concorrente de a) disponibilidade somente para usuários autorizados, b) confidencialidade e c) integridade com 'alterações impróprias' significando 'alterações não autorizadas'.

Os atributos acima podem ser enfatizados para um maior ou menor grau dependendo da aplicação. Algum grau de disponibilidade é sempre requerida, enquanto confiabilidade, segurança de funcionamento e confidencialidade podem ou não ser necessários. Os atributos de dependabilidade devem ser interpretados em um sentido probabilístico e relativo, pois, devido a presença ou a ocorrência inevitável de falhas, sistemas nunca são totalmente disponíveis, confiáveis e seguros.

Os meios para atingir dependabilidade combinam a utilização de quatro técnicas:

- Prevenção de falhas: como prevenir a ocorrência ou a introdução de falhas. Este objetivo é atingido pelas técnicas de controle de qualidade empregadas durante a modelagem e a fabricação do *hardware* e do *software*;
- Tolerância a falhas: como fornecer o serviço de forma correta na presença de falhas ativas. Falhas ativas são aquelas que produzem erros. Normalmente, esta técnica é implementada pela detecção da falha através do erro (ver Seção 2.2.1) e a subsequente correção e/ou recuperação do sistema. Mais detalhes desta técnica serão vistos na Seção 2.2;
- Remoção de falhas: como reduzir o número ou severidade das falhas. Esta técnica é realizada durante a fase de desenvolvimento e durante a fase de vida operacional do sistema. Durante a fase de desenvolvimento do sistema, a remoção consiste de três passos: 1) verificação se o sistema atende a certas propriedades, nomeadas as condições de verificação; 2) em caso de falha, diagnóstico da(s) falha(s) que causou o

não atendimento das condições de verificação; e 3) realizar as correções necessárias. Durante a vida operacional de um sistema, a remoção é realizada por manutenção preventiva ou corretiva. A manutenção preventiva visa descobrir e remover possíveis falhas antes que elas causem erros. Esta manutenção inclui falhas físicas, que ocorreram depois da última manutenção preventiva, e falhas de modelagem, que provocaram erros em outros sistemas similares. Entretanto, nas aplicações tipicamente as falhas provocarão erros que serão tratados posteriormente. A manutenção corretiva busca remover falhas que produziram um ou mais erros detectados. Normalmente, nesta manutenção para falhas de modelagem, a falha primeiro é isolada, por exemplo através de *patches*, para depois a remoção ser completada. Estas formas de manutenção podem ser *on-line*, sem a interrupção do serviço fornecido, ou *off-line*, durante uma parada do serviço;

- Previsão de falhas: como estimar o número atual, as futuras incidências e as possíveis consequências das falhas. Esta técnica é realizada pela avaliação do comportamento do sistema em relação a ocorrência ou a ativação das falhas. A avaliação do sistema tem dois aspectos: 1) avaliação qualitativa ou ordinal, que visa identificar, classificar os modos de defeito ou a combinação de eventos que causam defeito no sistema; 2) avaliação quantitativa ou probabilística, que busca avaliar, em termos de probabilidade, o nível em que alguns dos atributos de dependabilidade são satisfeitos. A avaliação da cobertura fornecida pelos mecanismos de tratamento de falhas e erros pode ser realizada na modelagem ou através de testes, ou seja, na injeção de falhas para verificar o comportamento do sistema.

Este trabalho propõe introduzir mecanismos de tolerância a falhas em um SGA, embutido na aplicação do usuário, para permitir que as aplicações MPI terminem as suas execuções mesmo em caso de falhas. Neste trabalho serão usados os componentes físicos disponíveis, que não são do nosso controle e não é possível ter nenhuma garantia sobre o seu comportamento, e a aplicação desenvolvida pelo usuário, que é recebida pronta e dentro dela é embutida o SGA EasyGrid. Assim, supõe-se que a aplicação do usuário foi desenvolvida corretamente e que os componentes físicos podem falhar. Por essa razão, este trabalho não utiliza as técnicas de prevenção e remoção de falhas, já que a modelagem da aplicação e suas correções deverão ser feitas pelo usuário e a fabricação/manutenção do *hardware* está fora do escopo do trabalho. A tolerância a falhas é a principal técnica utilizada neste trabalho e será descrita com mais detalhes na próxima seção. Na previsão de falhas, a injeção de falhas foi utilizada para testar o comportamento do sistema na presença de falhas.

2.2 Tolerância a Falhas

Este trabalho estará focado na técnica de tolerância a falhas para alcançar dependabilidade. O objetivo é conseguir manter o sistema fornecendo o serviço de forma correta mesmo na presença de falhas.

Neste contexto, na próxima seção serão descritas as diferenças sutis entre os conceitos de falha, erro e defeito. Na Seção 2.2.2 são apresentadas algumas características importantes para os modelos de falhas. Logo em seguida, na Seção 2.2.3, são mostrados os mecanismos de tolerância a falhas normalmente adotados pelos sistemas computacionais.

2.2.1 Definição de Falha, Erro e Defeito

Utilizando a mesma nomenclatura apresentada por (ANDERSON; LEE, 1981; GÄRTNER, 1999; WEBER, 2002), um **defeito** (*failure*) no sistema é definido como um desvio da sua especificação. Um sistema está em estado errôneo, ou em **erro**, se este estado pode causar um defeito. Um defeito ocorre quando um erro atinge a interface do serviço e altera o serviço. Por fim, **falha** (*fault*) é a causa física ou algorítmica do erro. No modelo de 3 universos, mostrado na Figura 2.1, as falhas pertencem ao universo físico, erros ao universo da informação e defeitos ao universo do usuário (WEBER, 2002).

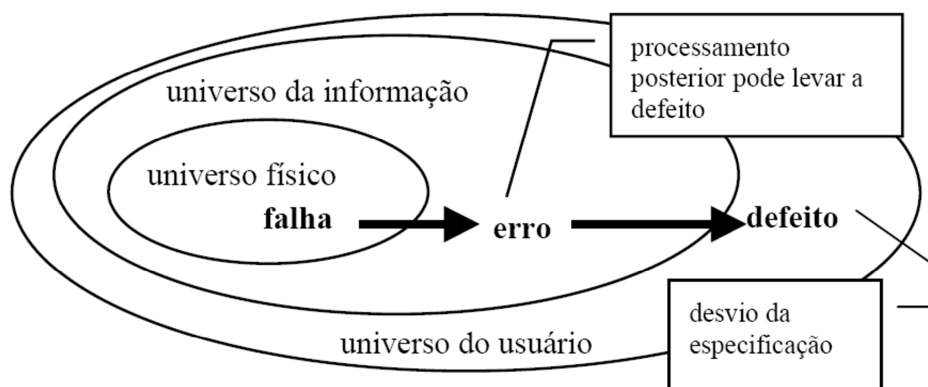


Figura 2.1: Falha, erro e defeito no modelo de 3 universos (WEBER, 2002).

Por exemplo, no modelo de 3 universos, um bit alterado seja pela transmissão de dados ou por interferência externa é uma falha no universo físico. Esta falha pode provocar uma interpretação errada do dado armazenado no sistema, que é o erro no universo da informação. Como consequência, esta informação errada pode acarretar um serviço incorreto do sistema, que é o defeito no universo do usuário. É importante observar que nem sempre uma falha leva a um erro. No exemplo acima, a área de memória alterada

poderia nunca ser acessada. Outra ressalva é que nem todo erro pode conduzir a um defeito. No exemplo utilizado, a informação requisitada poderia ser obtida a partir de outro dado redundante no sistema ou o erro ser detectado pela verificação do dado recebido e corrigido.

As falhas são inevitáveis, pois os *hardwares* envelhecem e sofrem interferências externas, e os *softwares* por serem complexos são atingidos por falhas humanas em trabalhar com grande volume de detalhes ou com deficiências de especificação. Uma falha normalmente causa um erro dentro do estado de um ou mais componentes do sistema. A presença de um erro no sistema deve ser verificada ativamente pelo próprio sistema ou por algum sistema externo. Assim, os defeitos podem ser evitados através das técnicas de tolerância a falhas.

Um dos princípios fundamentais para sistemas confiáveis (*dependable*) é que um modelo de falhas deve ser especificado para um sistema tolerante a falhas (KOOPMAN, 2003). Este modelo de falhas é que definirá quais falhas o sistema deve tratar. Sem um modelo de falhas, não existe modo de avaliar se um dado sistema é ou não tolerante a falhas.

2.2.2 Características dos Modelos de Falhas

Os sistemas computacionais e as aplicações podem falhar de diversos modos. As técnicas de tolerância a falhas assumem um determinado modelo na especificação dos tipos de falhas que serão tratadas. Koopman em (KOOPMAN, 2003) apresentou as seguintes características tipicamente consideradas nos modelos de falhas: manifestação da falha, duração, fonte da falha, perfil de ocorrência e granularidade.

Considerando a manifestação da falha, nem todas as falhas são tão severas quanto outras. Assim, as falhas podem ser classificadas por modelos abstratos que descrevem como o sistema se comportará na presença de falhas. Schneider, em (SCHNEIDER, 1993), apresentou o seguinte modelo de falhas:

- *Fail-stop*: Um processador falha por travamento (*halting*). Uma vez travado, o processador permanece no mesmo estado. Neste modelo assume-se que a parada de um processador é detectável por outros processadores. O processador pode até ser amigável para anunciar que está prestes a falhar (TANENBAUM; STEEN, 2001).
- Falhas por Colapso (*Crash*): Um processador falha por travamento. Uma vez travado, o processador permanece no mesmo estado. O fato que um processador falhou

pode não ser detectável por outros processadores. Segundo (CHANDRA; TOUEG, 1996), uma falha por colapso também ocorre quando um processo que está executando corretamente, pára prematuramente e uma vez travado não se recupera mais.

- Falhas por Colapso + *Link*: Um processador falha por travamento ou por perda de mensagens. Uma vez travado, o processador permanece no mesmo estado. Um *link* falha pela perda de algumas mensagens, mas nenhuma mensagem é atrasada, duplicada ou corrompida.
- Omissão-Recebimento (*Receive-Omission*): Um processador falha por travamento e permanece travado ou por não receber todas as mensagens que lhe foram enviadas.
- Omissão-Envio (*Send-Omission*): Um processador falha por travamento e permanece travado ou por não transmitir todas as mensagens que deveria enviar.
- Omissão em Geral: Um processador falha por receber somente um subconjunto de mensagens que lhe foram enviadas, por transmitir somente um subconjunto das mensagens que deveria enviar, e/ou por travamento e permanecer travado.
- Falhas Bizantinas: Um processador falha por exibir um comportamento arbitrário.

Segundo (SCHNEIDER, 1993), as falhas *fail-stop* são menos perturbadoras, porque os processadores nunca realizam ações erradas e as falhas são detectáveis. Entretanto, em sistemas assíncronos é impossível distinguir entre um processador que está executando muito lento e um outro que está travado devido a uma falha por colapso. Um processador que tem uma falha por colapso não executa mais nenhuma instrução, contudo um processador lento pode executar. Assim, outros processadores podem "seguramente" substituir um processador com falha por colapso, mas não um processador que só ficou lento, porque as ações subsequentes deste processador mais lento podem não ser consistentes com as ações realizadas pelos substitutos, por exemplo, gravação de dados em memória compartilhada. Desta forma, falhas por colapso em sistemas assíncronos são mais difíceis de tratar do que falhas do tipo *fail-stop*. Porém, em sistemas síncronos, os modelos de falhas dos tipos *fail-stop* e por colapso são equivalentes. Os modelos de falhas por Colapso + *Link*, Omissão-Recebimento, Omissão-Envio e Omissão Geral tratam falhas envolvendo perda de mensagens. As falhas bizantinas são mais perturbadoras e um sistema que tolera este tipo de falha pode tolerar falhas de qualquer tipo.

Já Treaster, em (TREASTER, 2005), incluiu a falha de desempenho e apresentou três modelos de falhas considerando o comportamento do sistema:

- Falhas *Fail-stop*: qualquer componente de *hardware* ou *software* pode falhar a qualquer momento. Entretanto, quando uma falha ocorre, este componente cessa de gerar resultados e de interagir com o restante do sistema;

- Falhas *Fail-stutter*: inclui todas as situações do modelo *fail-stop*, mas também permite falhas de desempenho. Uma falha de desempenho é uma situação na qual um componente fornece inesperadamente baixo desempenho, mas continua funcionando corretamente;

- Falhas Bizantinas: o componente de *hardware* ou *software* que falhou continua interagindo com o resto do sistema. O comportamento deste componente pode ser arbitrário e inconsistente, o que pode colaborar para o desenvolvimento de resultados errôneos. Este modelo pode representar falhas randômicas de sistema e ataques maliciosos de *hackers*. Então, segundo (LIMA; GREVE, 2009), os detectores de falhas bizantinas fornecem uma forma de tratar problemas de segurança, visto que este modelo considera a existência de processos corruptos que podem, entre outras ações, assumir a identidade de outros processos.

Além disso, as falhas também podem ser classificadas de acordo com a sua duração. De acordo com Nelson em (NELSON, 1990), a duração de uma falha pode ser:

- Transitória: A falha existe por um tempo limitado e não é repetitiva. Normalmente, esta falha é provocada por uma interferência externa;

- Intermitente: A falha provoca uma oscilação entre operações errôneas e livre de erros. Esta falha, na maioria dos casos, é resultado de operações em dispositivos instáveis ou sobrecarregados;

- Permanente ou "*hard*": A falha ocorre devido a uma condição do dispositivo que não é corrigida com o tempo. Esta falha é provocada por defeitos de componentes, danos físicos ou erros de projeto.

Uma característica importante é a fonte das falhas que serão consideradas no modelo. As falhas podem ocorrer devido a erros de requisitos, erros de implementação, erros operacionais, entre outros. Mudanças em ambientes operacionais e a investida de ataques maliciosos podem fazer um sistema que funcionava previamente parar de funcionar. Alguns sistemas são modelados para somente tratar falhas de *hardware*, tais como perda de memória ou capacidade de CPU, e não falhas de *software* (KOOPMAN, 2003).

Além da fonte das falhas, é necessário definir o perfil das ocorrências de falhas que são consideradas para serem tratadas. As falhas consideradas podem ser somente falhas esperadas, tais como falhas observadas historicamente ou exceções definidas; falhas com base na análise da modelagem; ou falhas inesperadas. Adicionalmente, as falhas podem ser randômicas e independentes, podem ser correlacionadas em espaço ou tempo, ou podem ser intencionais devido a ações maliciosas (KOOPMAN, 2003).

Outra característica que deve ser considerada é a granularidade de uma falha, que é o tamanho do componente comprometido pela falha. A noção do tamanho da região de contenção de uma falha é um parâmetro chave da modelagem em sistemas tolerantes a falhas (KOOPMAN, 2003). Uma falha pode atingir um módulo de *software*, uma tarefa, um conjunto de CPU's ou um *site* computacional inteiro. Diferentes mecanismos de tolerância a falhas são apropriados dependendo da granularidade da falha e da granularidade da ação de recuperação.

2.2.3 Mecanismos de Tolerância a Falhas

Segundo Ziv and Bruck em (ZIV; BRUCK, 1996), os mecanismos de tolerância a falhas normalmente possuem 4 estágios:

1. Detecção da falha/erro: o processo de reconhecimento da ocorrência de um erro no sistema;
2. Localização da falha: envolve identificar a parte do sistema que causou o erro;
3. Contenção da falha: visa prevenir uma possível propagação de erro para o resto do sistema por isolando o componente com falha;
4. Recuperação da falha/erro: o processo de restaurar o estado operacional do sistema para um estado consistente e livre de erros e falhas que possam ser ativadas novamente.

Independente dos 4 estágios citados acima, outra classe para a técnica de tolerância a falhas é o mascaramento de falhas. Segundo Weber em (WEBER, 2002), nesta classe as falhas não se manifestam como erros, pois são mascaradas na origem. O uso de redundância suficiente pode permitir a recuperação sem detecção de erro explícito. Assim, o sistema não entra em estado errôneo e não é necessário gastar tempo para realizar os 4 estágios. Este trabalho não usará a técnica de mascaramento de falhas, pois a biblioteca MPI possui mecanismos próprios de detecção de erros e estes erros devem ser tratados.

Detecção da Falha/Erro

O primeiro estágio dos mecanismos de tolerância a falhas consiste na detecção da falha através do erro, já que o erro é causado por uma falha. Existem duas classes de técnicas de detecção de erros (AVIZIENIS; LAPRIE; RANDELL, 2001): detecção concorrente de erro (*concurrent error detection*), que ocorre durante o fornecimento do serviço; e detecção preemptiva de erro (*preemptive error detection*), que ocorre enquanto o fornecimento do serviço está suspenso.

Como dito por (SCHNEIDER, 1993), em sistemas assíncronos, é impossível distinguir se um processador está executando muito lento ou se está travado devido a uma falha por colapso. Para sobrepor este problema, Chandra e Toueg (CHANDRA; TOUEG, 1991, 1996) propõem o conceito de detectores de falhas não-confiáveis para sistemas com falhas por colapso. Os autores definem o detector de falhas como um oráculo distribuído, onde cada processo tem seu próprio módulo detector de falhas local. Cada módulo local monitora um subconjunto de processos no sistema não necessariamente disjuntos e mantém uma lista daqueles que são atualmente suspeitos de falhas. O detector de falhas não é confiável no sentido que cada módulo detector pode cometer erros por adicionar indevidamente processos na sua lista de suspeitos, ou seja, o detector pode suspeitar de um processo mesmo que ele continue executando. Caso o módulo detector descubra que foi um erro suspeitar de um determinado processo, então ele pode remover o processo em questão da sua lista de suspeitos. Assim, cada módulo pode repetidamente adicionar e remover processos da sua lista de suspeitos. Em qualquer momento, dois módulos detectores de dois diferentes processos podem ter diferentes listas de suspeitos.

Os detectores de falhas não-confiáveis são caracterizados por duas propriedades que devem satisfazer: **completude** e **precisão**. Existem dois tipos de completude:

- **Completude Forte**: Eventualmente todo processo que falha por colapso é permanentemente suspeito por todos os processos corretos.
- **Completude Fraca**: Eventualmente todo processo que falha por colapso é permanentemente suspeito por algum processo correto.

Esta propriedade por si só não é útil, pois um detector de falhas pode satisfazê-la por estar sempre suspeitando de todos os processos do sistema. Para impedir este comportamento, um detector de falha também deve satisfazer um requisito de precisão que restringe os erros que um detector pode cometer. Foram definidos dois tipos de precisão:

- **Precisão Forte**: Os processos corretos nunca são suspeitos de falha.
- **Precisão Fraca**: Algum processo correto nunca é suspeito de falha.

Com base nos tipos de completude e precisão, (CHANDRA; TOUEG, 1991) definiram as seguintes três classes de detectores:

- P: o conjunto de **Detectores de Falhas Perfeito**, que satisfaz as propriedades de **Completude Forte** e **Precisão Forte**.
- S: o conjunto de **Detectores de Falhas Forte**, que satisfaz as propriedades de **Completude Forte** e **Precisão Fraca**.
- W: o conjunto de **Detectores de Falhas Fraco**, que satisfaz as propriedades de **Completude Fraca** e **Precisão Fraca**.

Mesmo os detectores da classe W, **detectores de falhas fraco**, devem garantir que pelo menos um processo correto nunca é suspeito. Visto que esta precisão é muito difícil de ser alcançada, os próprios autores em (CHANDRA; TOUEG, 1991) consideraram uma forma mais fraca de precisão. Eles consideram que os detectores podem suspeitar de todos os processos em um momento ou outro, ou seja, a propriedade de precisão é eventualmente satisfeita. Foram definidos dois tipos de precisão eventual:

- **Precisão Eventualmente Forte**: Existe um tempo depois que processos corretos não são suspeitos.
- **Precisão Eventualmente Fraca**: Existe um tempo depois que algum processo correto não é suspeito.

Para cada classe de detectores, pode-se substituir o requisito de **precisão** pelo correspondente requisito de **precisão eventual**, o que resulta nas seguintes três classes:

- $\diamond P$: o conjunto de **Detectores de Falhas Eventualmente Perfeito**, que satisfaz as propriedades de **Completude Forte** e **Precisão Eventualmente Forte**.
- $\diamond S$: o conjunto de **Detectores de Falhas Eventualmente Forte**, que satisfaz as propriedades de **Completude Forte** e **Precisão Eventualmente Fraca**.
- $\diamond W$: o conjunto de **Detectores de Falhas Eventualmente Fraco**, que satisfaz as propriedades de **Completude Fraca** e **Precisão Eventualmente Fraca**.

Os detectores da classe $\diamond W$, **detectores de falhas eventualmente fraco**, identificarão eventualmente todos os processos com falhas e pelo menos um processo correto. Segundo (STELLING et al., 1999), em sistemas reais, esta espera ilimitada não é aceitável, quando o custo de esperar por uma determinação absolutamente correta pode exceder o custo que incidiria se fosse assumido simplesmente que o detector de falhas estava correto e fosse realizada uma ação com base nesta suposição. Segundo os mesmos autores, a decisão de quando a informação fornecida por um detector de falhas deve ser acreditada é de responsabilidade da aplicação. O detector de falhas não pode interpretar os seus resultados. Uma aplicação deve usar a informação fornecida pelo detector de falhas para tomar uma decisão com base na probabilidade de uma detecção de falhas está errada. Esta decisão envolve o custo específico da aplicação de realizar alguma ação se a detecção for falsa e o custo de não realizar aquela ação se a detecção de fato fosse verdadeira.

Segundo (DÉFAGO; HAYASHIBARA; KATAYAMA, 2003), para a implementação de detectores de falhas é bastante comum a utilização da estratégia de *heartbeat*. Nesta estratégia, cada módulo detector de falhas periodicamente envia uma mensagem de *heartbeat* para os outros módulos, com o objetivo de informá-los de que continua vivo. Um processo p suspeita de falha de um processo q, quando o seu módulo detector de falhas local deixa de receber as mensagens de *heartbeat* vindas do módulo de q, por um período de tempo maior do que um tempo de espera (*timeout*) determinado. Se o tempo de espera for pequeno, as falhas por colapso serão detectadas rapidamente, mas a probabilidade de detectar suspeitos errados é alta. Por outro lado, se o tempo de espera for longo, a chance de suspeitos errados é baixa, mas demora mais para detectar as falhas. Neste mesmo artigo citado, os autores apresentam um detector de falhas chamado *φ-failure detector*. Este detector de falhas associa um valor φ_p para cada processo p conhecido. O valor φ_p muda dinamicamente e representa o grau de confiança na detecção de que um processo p tenha falhado por colapso. Cada aplicação pode configurar o seu próprio limite para φ_p e, caso o valor de φ_p ultrapasse este limite, decidir suspeitar da falha do processo p. Algumas aplicações podem configurar um limite baixo para obter uma rápida detecção de falha, mesmo que esta não seja muito precisa, ou seja, muitas suspeitas equivocadas. Enquanto outras aplicações com requisitos mais fortes podem configurar um limite mais alto e obter suspeitos com maior precisão. Na prática, o cálculo do valor φ_p é baseado no histórico do intervalo de chegada das mensagens de *heartbeat*.

Em (STELLING et al., 1999), os autores realizaram experimentos utilizando a estratégia de *heartbeat* e mostraram que a probabilidade de uma detecção de falhas equivocada é geralmente baixa dependendo do tempo de espera configurado. Foram usadas 9 máqui-

nas interconectadas através de diferentes redes de conexão LAN, MAN e a internet. O intervalo de envio das mensagens de *heartbeat* foi definido em 10 segundos. Os resultados mostraram que se uma falha é definida como correspondendo a nenhuma mensagem de *heartbeat* no intervalo de 240 segundos, então a taxa de falso positivo é menor do que 1 em 100000. Segundo estes mesmos autores, alguns destes falsos positivos são devido a uma perda temporária de conectividade da rede que durou alguns minutos e que este tipo de evento pode querer ser visto como uma falha. Os autores também mencionam que uma detecção mais rápida é possível com algum aumento na taxa de falso positivo. Por exemplo, usando o tempo limite de 35 segundos para o recebimento de uma mensagem de *heartbeat*, a taxa de falso positivo é de 1 em algumas centenas de casos.

Localização e Contenção da Falha

Depois da detecção do erro, a localização da falha visa identificar o componente que causou o erro. A contenção da falha realiza a exclusão lógica ou física do componente que falhou, ou seja, evita este componente de participar do fornecimento do serviço e de produzir novos erros. No caso de aplicações paralelas, a localização e a contenção da falha pode ser facilitada por uma hierarquia de processos, onde nem todos os processos se comunicam diretamente e o isolamento de um processo passa a ser responsabilidade do processo de nível superior.

Recuperação da Falha/Erro

O último estágio é a recuperação do sistema, que consiste do tratamento dos erros e das falhas. A recuperação do erro no estado do sistema pode ser realizada de duas formas (AVIZIENIS; LAPRIE; RANDELL, 2001): recuperação por retorno (*rollback*), onde a transformação do estado consiste em retornar o sistema de volta para um estado gravado que existiu antes da detecção do erro; ou recuperação por avanço (*rollforward*), onde o estado sem erros detectados é um novo estado.

De acordo com Elnozahy, Alvisi, Wang e Johnson em (ELNOZAHY et al., 2002), os protocolos de recuperação por retorno supõem as seguintes condições:

- A execução de um processo é uma sequência de intervalos de estado, onde cada intervalo é iniciado por um evento não determinístico. A execução durante cada intervalo de estado é determinístico, tal que se um processo inicia do mesmo estado e estiver sujeito aos mesmos eventos não determinísticos nos mesmos pontos de execução, o processo sempre produzirá os mesmos resultados;

- Quando um processo falha, este perde seu estado volátil e interrompe sua execução de acordo com o modelo *fail-stop*;
- Todos os processos possuem acesso a um dispositivo de armazenamento estável que sobrevive a falhas, tal que as informações de estado gravadas neste dispositivo podem ser usadas na recuperação por todos os processos.

Sistemas com base em troca de mensagens complicam a recuperação por retorno, pois as mensagens induzem dependências entre os processos durante a execução. Quando um ou mais processos falham, estas dependências podem provocar o retorno de alguns processos que não falharam, o que normalmente é denominado de **propagação de retorno** (*rollback propagation*). Para ilustrar este problema, considere que o emissor de uma mensagem tenha retornado para um estado anterior ao envio. O receptor da mensagem também deve retornar para um estado anterior ao recebimento, pois senão os dois processos teriam um **estado inconsistente**, já que a mensagem em questão já teria sido recebida pelo receptor antes de ter sido enviada pelo estado atual do emissor. Esta situação é impossível numa execução correta e livre de falhas. Em alguns casos, a propagação de retorno pode se estender até o estado inicial da execução, onde todo o trabalho realizado antes da falha é perdido. Esta propagação é conhecida como **efeito dominó**.

Para sistemas de troca de mensagens, um **estado global** é o conjunto dos estados individuais de cada processo participante e dos estados dos canais de comunicação. Já um **estado consistente** é aquele que se o estado de um processo indica a recepção de uma mensagem, então o estado do emissor indica o envio daquela mensagem. Assim, um **estado global consistente** é aquele que pode ocorrer durante a execução correta e livre de falhas de uma computação distribuída.

Os protocolos de recuperação por retorno podem ser classificados como baseados em *checkpointing* e baseados em *log* (ELNOZAHY et al., 2002). O protocolo baseado em *checkpointing* periodicamente grava o estado de execução dos processos em um meio de armazenamento estável durante a execução livre de falhas. Se um erro é detectado, o processo que falhou é reiniciado de um dos seus estados gravados (*checkpoints*), o que reduz a quantidade de computação perdida, de forma que o sistema volte para um **estado global consistente** e livre de erros. O efeito dominó pode ocorrer se cada processo gravar o seu *checkpoint* de forma independente, o que é conhecido como *checkpoint* não coordenado ou independente. Embora possa ocorrer o efeito dominó, esta técnica tem como principal vantagem a autonomia de cada processo para decidir quando realizar o seu próprio *checkpoint*. Entretanto, existem duas outras técnicas usando *checkpoint* que evitam o

efeito dominó. Uma técnica é o *checkpoint* coordenado, em que os processos coordenam o momento da gravação do *checkpoint* a fim de gravar um estado global consistente do sistema. A principal vantagem desta técnica é a recuperação mais simples, visto que cada processo sempre reinicia do *checkpoint* mais recente. Entretanto, existe um grande atraso na entrega das mensagens para a aplicação durante o processo do *checkpoint* global, visto que as mensagens precisam esperar a realização do *checkpoint* para serem processadas pela aplicação. A outra técnica que evita o efeito dominó é o *checkpoint* induzido por comunicação (*communication-induced checkpoint*). Esta técnica permite os processos realizarem seus *checkpoints* independentemente, mas para garantir um estado consistente os processos podem ser forçados a fazerem *checkpoints* adicionais. Estes *checkpoints* adicionais são baseados em informações embutidas nas mensagens da aplicação recebidas de outros processos. Estas informações embutidas estão relacionadas ao protocolo e sempre garantem a existência de um estado consistente do sistema. Os *checkpoints* forçados são realizados antes do conteúdo da mensagem da aplicação ser processado, assim causam uma alta latência na entrega das mensagens. Desta forma, nesta técnica é desejável reduzir o máximo possível o número de *checkpoints* forçados.

As técnicas de *checkpointing* podem ser complexas, já que uma aplicação paralela precisa ser retornada para um estado global consistente, e custosa, visto que em um ambiente grade geralmente não existe um sistema de arquivo global e seguro para a gravação de todos os arquivos de *checkpoint* gerados por uma aplicação paralela. Outra característica desta técnica é que o mesmo número de processos usados no momento da gravação do *checkpoint* é necessário na fase de recuperação, já que todos os arquivos precisam ser usados na restauração do estado global da aplicação. Caso o mesmo número de recursos não esteja disponível, então será preciso compartilhar um recurso entre dois ou mais processos, o que pode prejudicar o desempenho da aplicação.

Os protocolos baseados em *log* combinam *checkpoint* com um *log* de eventos não determinísticos. Em (ELNOZAHY et al., 2002), um evento não determinístico é modelado como o recebimento de uma mensagem. Neste modelo, que também é usado neste trabalho, *log* de eventos não determinísticos significa *log* de mensagens. A técnica de *log* de mensagens confia que todos os eventos não determinísticos podem ser identificados e que as informações necessárias para reproduzir cada evento durante a recuperação podem ser gravadas. Pelas informações gravadas e a reprodução dos eventos não determinísticos na mesma ordem original, um processo pode de forma determinística recriar seu estado antes da falha, mesmo que este estado não tenha sido gravado em um *checkpoint*. Os *checkpoints* podem ser usados em conjunto para reduzir o tamanho do *log* de mensagens

e evitar a reexecução inteira dos processos que falharam. Dependendo de como o *log* de mensagens é gravado, existem três protocolos de recuperação por retorno baseado em *log*:

- *Log* Pessimista (*pessimistic logging*): todas as informações dos eventos não determinísticos (mensagens recebidas) são gravadas no meio de armazenamento estável antes do evento afetar a computação. Este protocolo supõe que uma falha pode ocorrer depois de qualquer evento não determinístico, o que é uma visão pessimista, já que falhas são relativamente raras;

- *Log* Otimista (*otimistic logging*): os processos mantêm as informações em memória volátil e periodicamente elas são gravadas no meio de armazenamento estável. Este protocolo tem a visão otimista que a gravação no meio estável será realizada antes de uma falha ocorrer;

- *Log* Causal (*causal log*): as informações de cada evento não determinístico que precede de forma causal o estado de um processo será gravado no meio estável ou localmente para aquele processo. A precedência causal é baseada na relação **aconteceu antes** de Lamport (LAMPORT, 1978). Os *logs* mantidos por cada processo atuam como um seguro de proteção para falhas que ocorrem em outros processos.

As técnicas de tratamento de falhas no nível de processos têm sido muito estudadas em sistemas paralelos e distribuídos. De acordo com Yu e Buyya em (YU; BUYYA, 2005), estas técnicas podem ser classificadas em: **tentar novamente** (*retrying*), **recurso alternativo** (*alternative resource*), **checkpoint/reinício** (*checkpoint/restart*) e **replicação** (*replication*). A técnica **tentar novamente** é a mais simples de recuperação de falhas de processos, que consiste em executar novamente o mesmo processo no mesmo recurso depois da falha. A técnica de **recurso alternativo** submete o processo que falhou para outro recurso, consequentemente podendo tratar falhas de recursos. Já a técnica de **checkpoint/reinício** move os processos que falharam para outros recursos, caso necessário, de modo que os processos possam continuar a sua execução de um ponto anterior a falha, isto é, do estado gravado no *checkpoint*. Por fim, a técnica de **replicação** executa o mesmo processo em diferentes recursos para assegurar a execução dos processos, supondo que pelo menos uma das réplicas não falhe.

Avizienis em (AVIZIENIS, 1998) descreveu um modelo conceitual para um sistema de alta confiabilidade em que tolerância a falhas fosse um atributo integrante de todo elemento de *hardware*. Este modelo foi proposto com base no corpo humano e suas defesas, tal como o sistema de imunidade, o sentido de dor e o processo de cicatrização. As principais

analogias são: o corpo humano para *hardware*, os processos cognitivos que são suportados pelo corpo para *software*, cuja execução fornece os serviços de alta confiabilidade para qual o sistema é programado.

2.3 Computação Autônoma

Em 2001, a IBM introduziu o conceito de computação autônoma (IBM Research, 2010) para descrever uma nova estratégia para o desenvolvimento de sistemas de computadores distribuídos complexos. Computação autônoma (STERRITT et al., 2005), inspirado no sistema nervoso humano, foca no desenvolvimento de sistemas e aplicações com autogerenciamento (*self-management*), ou seja, o próprio sistema ou aplicação se regula às mudanças do ambiente. Esse tipo de sistema tomará suas próprias decisões a respeito do que é necessário ser feito para que o sistema continue satisfazendo seus objetivos. Para isso, o sistema deve constantemente verificar e otimizar seu estado, bem como, adaptar-se automaticamente às alterações de condições. Este conceito da própria aplicação se adaptar as mudanças de ambiente se enquadra com o princípio fim-a-fim (*end-to-end argument*), apresentado em (SALTZER; REED; CLARK, 1984), que defende a implementação das funções no nível da aplicação. Segundo os autores, muitas funções somente podem ser completamente e corretamente implementadas com o conhecimento e ajuda da aplicação. Um exemplo disto é apresentado neste trabalho na Seção 2.2.3 em **Detecção da Falha/Erro**, onde o detector de falhas não-confiável não pode interpretar os seus resultados, sendo da aplicação esta responsabilidade, segundo (STELLING et al., 1999).

Em (IBM Research, 2010; PARASHAR; HARIRI, 2005) são mostradas as seguintes oito características necessárias para um sistema ou uma aplicação autônoma:

1. Autoconhecimento (*Self Awareness*): Conhecer a si mesmo, o que implica conhecimento detalhado de seus componentes, estado e comportamento;

2. Autoconfiguração (*Self Configuring*): Capacidade de configurar e reconfigurar a si mesmo sob condições variadas e imprevisíveis. A configuração do sistema deve ocorrer de forma automática, bem como, ajustes dinâmicos para melhor tratar as mudanças do ambiente;

3. Auto-otimização (*Self Optimizing*): Capacidade de detectar comportamentos subótimos e otimizar a si mesmo para melhorar sua execução. Os componentes do

sistema serão monitorados e ajustes finos no fluxo de trabalho realizará as metas pré-determinadas do sistema;

4. Autorrecuperação (*Self-Healing*): Capacidade de detectar problemas ou potenciais problemas que podem causar alguma de suas partes ter um mau funcionamento, e então encontrar um modo alternativo de usar os recursos ou reconfigurar o sistema para continuar funcionando normalmente;

5. Autoproteção (*Self Protecting*): Capacidade de detectar e proteger seus recursos de ataques internos e externos mantendo a segurança e integridade do sistema inteiro;

6. Conhecimento de Contexto (*Context Aware*): Conhecer seu ambiente de execução e o contexto que envolve suas atividades para ser capaz de reagir às mudanças no ambiente. Regras serão encontradas e geradas para melhor interagir com os sistemas vizinhos. O sistema utilizará os seus recursos até eles serem negociados para outros sistemas, quando o próprio sistema e seu ambiente serão modificados;

7. Aberta (*Open*): Funcionar em um ambiente heterogêneo devendo ser portável através de múltiplas arquiteturas de *hardware* e *software*. Consequentemente, deve ser construído sobre padrões, interfaces e protocolos abertos;

8. Antecipação (*Anticipatory*): Capacidade de prever o máximo possível suas necessidades e comportamentos dentro do seu contexto. Além disso, ser capaz de gerenciar a si mesmo de forma pró-ativa. O sistema perceberá as necessidades otimizadas dos seus recursos, enquanto mantendo sua complexidade oculta.

O desenvolvimento de mecanismos para habilitar sistemas e aplicações autônomos tem como base módulos autocontidos (*self-contained*), que são responsáveis pela implementação das suas funções e comportamentos de acordo com o contexto e políticas definidas ou implantadas em tempo de execução. Embora **Autorrecuperação** pareça ser idêntico a tolerância a falhas, sua implementação eficiente é dependente de uma efetiva integração com outros *self*-* mecanismos, em particular autoconhecimento (*self awareness*), autoconfiguração (*self configuring*) e auto-otimização (*self-optimization*).

Então computação autônoma é uma aproximação holística emergente para o desenvolvimento de sistemas computacionais que visam trazer um novo nível de automação e dependabilidade para os sistemas através das funções de autorrecuperação (*self-healing*), auto-otimização (*self optimizing*), autoconfiguração (*self configuring*) e autoproteção (*self

protecting) (STERRITT; BUSTARD, 2003a, 2003b). Visto que qualquer sistema configurado incorretamente, e/ou otimizado ineficientemente, provavelmente conduzirá a falhas durante o seu uso. Similarmente, qualquer sistema não protegido adequadamente é vulnerável a falhas maliciosas, sejam elas oriundas de *hackers* ou um vírus. Assim, essencialmente todas as características da computação autônoma estão relacionadas com dependabilidade (STERRITT; BUSTARD, 2003a).

2.4 Resumo

Este capítulo descreveu a origem do conceito de tolerância a falhas, a relação entre tolerância a falhas, dependabilidade e computação autônoma. Além disso, foram discutidos os meios de atingir dependabilidade com ênfase na técnica de tolerância a falhas, pois é o foco deste trabalho. Na Seção 2.2.3 foram apresentadas as características tipicamente consideradas nos modelos de falhas e que serão usadas na definição do modelo utilizado neste trabalho. Adicionalmente, foram descritos os quatro estágios dos mecanismos de tolerância a falhas, que são detecção da falha/erro, localização da falha, contenção da falha e recuperação da falha/erro. Sendo apresentados em mais detalhes os estágios detecção da falha/erro, que inclui os detectores de falhas não-confiáveis para sistemas assíncronos com falhas por colapso, e a recuperação da falha/erro, inclusive os protocolos de recuperação por retorno. Os estágios da localização e da contenção da falha são facilitados pela hierarquia de processos, que é utilizado no SGA EasyGrid, como será visto no próximo capítulo. Por fim, foi introduzido o conceito de computação autônoma e as características necessárias para um sistema ou uma aplicação ser considerada autônoma. Este conceito é importante, pois o SGA EasyGrid tem o objetivo de transformar aplicações MPI em aplicações robustas capazes de executar eficientemente em ambientes distribuídos de larga escala, tais como as grades computacionais. No próximo capítulo, dentre outros assuntos, serão descritas com mais detalhes as características do SGA EasyGrid e algumas versões de MPI tolerante a falhas.

Capítulo 3

Trabalhos Relacionados

Um dos objetivos desta tese é desenvolver e avaliar mecanismos eficientes de tolerância a falhas para aplicações autônomas de alto desempenho que podem permitir executá-las em ambientes distribuídos abertos. A exploração de paralelismo e a adoção de bibliotecas que facilitem o desenvolvimento de programas paralelos são fundamentais ao avanço de *e-Science*.

Este capítulo apresenta algumas características da biblioteca MPI. Uma biblioteca amplamente utilizada em diversas áreas da computação, que se tornou o padrão *de facto* de troca de mensagens (FOSTER, 1995). Atualmente, existem centenas/milhares de aplicações paralelas implementadas em C/C++ e Fortran que foram desenvolvidas para se beneficiarem de sistemas distribuídos de larga escala.

A interface para troca de mensagens, MPI (*Message Passing Interface*), é uma biblioteca padronizada e portável de rotinas para troca de mensagens desenvolvida por um fórum aberto internacional com representantes da indústria, universidades e laboratórios governamentais (MPI, 2010). A interface MPI tornou-se popular não apenas por ter sido desenvolvida para executar em uma variedade de sistemas, mas também por ser baseada na troca de mensagens, um dos paradigmas mais utilizados em programação de sistemas paralelos. A biblioteca MPI fornece portabilidade tanto para computadores de processamento maciçamente paralelo quanto para *clusters* de estações de trabalho, o que é importante em ambientes de grades computacionais.

Existem 2 versões do padrão MPI e uma terceira está sendo discutida. O padrão MPI-2 possui várias implementações tanto comercial/proprietário quanto de código aberto. As implementações de código aberto mais conhecidas são: MPICH2 (MPI *Chameleon*) (GROPP et al., 1996; Argonne National Laboratory, 2010), LAM/MPI (LAM - *Local Area Multicom-*

puter) (LAM Team, 2010) e mais recentemente Open MPI (GABRIEL et al., 2004; Open MPI, 2010). As implementações Scali MPI, MPI da Intel e MPI da Bull são exemplos de implementações de fabricantes para sistemas específicos de arquiteturas de computadores, que são tipicamente derivadas de implementações abertas de MPI. Para a utilização em ambientes de grade foi desenvolvida uma versão inicial do MPICH chamada MPICH-G2 (MPICH-G2, 2010). O MPICH-G2 usa os serviços fornecidos pelo Globus Toolkit (Globus Toolkit, 2010) para estender a funcionalidade de MPICH para permitir a execução através de múltiplos domínios. O MPICH-G2 esconde a heterogeneidade da grade utilizando os serviços do Globus Toolkit para realizar as operações de autenticação, autorização, distribuição de executável, criação de processos, monitoramento e controle de processos, comunicação, redirecionamento da E/S padrão e acesso a arquivos remotos (KARONIS; TOONEN; FOSTER, 2003). Já a implementação LAM/MPI, a partir da versão 7.0 também passou a fornecer suporte para a execução de aplicações MPI em grades computacionais que utilizam o Globus Toolkit. Uma vantagem do LAM/MPI em relação ao MPICH-G2 é o suporte às funções de gerenciamento dinâmico de processos, que são essenciais para ambientes dinâmicos e mais sujeitos a falhas como as grades computacionais. O Open MPI ainda não possui suporte para a utilização dos serviços fornecidos pelo Globus Toolkit, apesar de ser uma implementação influenciada pelo código base dos projetos LAM/MPI, LA-MPI (GRAHAM et al., 2003) e FT-MPI (FAGG; DONGARRA, 2000).

As especificações do MPI determinam que o tratamento padrão de erro é que erros no MPI são fatais. Assim, se nenhuma ação for realizada pela própria aplicação em relação a tratamento de erro, quando um processo terminar a sua execução por causa de alguma falha e esta condição for detectada pela biblioteca, todos os outros processos também abortam (GROPP; LUSK, 2004). Essa característica se deve ao fato de o MPI ter sido inicialmente desenvolvido para ser executado em ambientes estáticos, homogêneos e menos propensos a falhas, tais como *clusters* computacionais dedicados. Com o avanço dos sistemas computacionais, estas características estão mudando e, cada vez mais, surge a necessidade dos pesquisadores investigarem diversas aproximações para fornecer tolerância a falhas no MPI. Só que as estratégias propostas são limitadas pelo modelo de execução adotado pela aplicação.

As aplicações MPI são tipicamente desenvolvidas para executar processos de longa duração, um por processador (FOSTER, 1995), como mostrado na Figura 3.1. Neste modelo de execução, chamado de "**um processo por processador (1PProc)**" em (SENA et al., 2007; SENA, 2008), cada processador utilizado pela aplicação paralela executa apenas um processo da aplicação do início até o final da sua execução. Isso se deve ao fato

das aplicações paralelas serem desenvolvidas com o objetivo de maximizar o desempenho e, normalmente, consideram os recursos computacionais homogêneos e dedicados a sua execução. Este modelo de execução é eficiente para ambientes homogêneos, dedicados e interconectados através de uma rede local. Entretanto, em ambientes como as grades computacionais, o modelo de execução 1PProc se torna inapropriado, visto que a granularidade de cada processo tende a requerer ajustes. O fato de que os recursos podem ser heterogêneos, compartilhados com *jobs* locais e mais propensos a falhas não somente faz este modelo ineficiente, mas também torna gerenciar a execução das aplicações extremamente complexa (SENA, 2008; MAGHRAOUI et al., 2009).

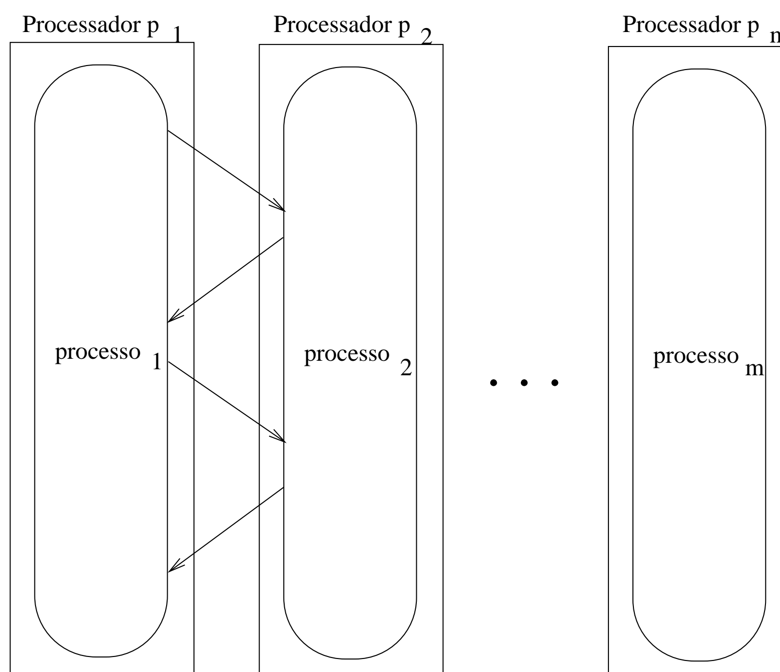


Figura 3.1: Modelo de execução 1PProc (SENA, 2008).

Neste capítulo serão apresentadas algumas versões de MPI tolerante a falhas na Seção 3.1. Visto que os mecanismos de tolerância a falhas desenvolvidos neste trabalho foram implementados no SGA EasyGrid, uma visão geral do Sistema de Gerenciamento de Aplicações (SGA) EasyGrid será mostrada na Seção 3.2, inclusive uma descrição dos modelos da aplicação, arquitetural e de execução adotados pelo SGA EasyGrid.

3.1 Versões de MPI Tolerante a Falhas

Segundo Neophytou (NEOPHYTOU, 2010), o framework MPI-FT é referenciado como uma das primeiras tentativas de abordar a questão de tolerância a falhas no MPI. Entretanto, segundo o próprio autor, este trabalho está atualmente desatualizado e existem

outras alternativas mais eficientes. O MPI-FT (LOUCA et al., 2000) propõe duas estratégias de *log* de mensagens para a recuperação de processos. Em uma estratégia, todas as comunicações são gravadas em um processo central chamado **Observador** (*Observer*). Se um processo falha, este processo é o responsável por reenviar as mensagens para o substituto. Na outra estratégia, cada processo é responsável por manter uma cópia de todas as suas mensagens enviadas. No caso de uma falha, o processo observador indica quem falhou e os processos reenviam as mensagens do processo morto para o substituto. Os processos substitutos podem ser criados dinamicamente ou na inicialização do programa ficando ociosos em recursos extras esperando uma ativação do processo observador. Apesar das funções desenvolvidas serem incluídas no código do usuário em tempo de compilação através da biblioteca MPI_FT, pontos de verificação de estado precisam ser inseridos explicitamente pelo usuário/programador no código. Os pontos de verificação de estado é que permitem ao processo observador avisar aos processos da aplicação sobre as falhas ocorridas e estes executarem as rotinas de recuperação. Uma suposição implícita é que o recurso onde o processo observador está executando nunca pode falhar.

CoCheck (STELLNER, 1996, 2010) é um ambiente que fornece *checkpointing* transparente para aplicações paralelas baseadas em sistemas de troca de mensagens. Existem implementações de CoCheck para as bibliotecas de troca de mensagens PVM (PVM, 2010) e tuMPI (Technische Universität München MPI) que é uma implementação MPI desenvolvida pelos próprios autores. Este sistema está situado em uma camada acima do PVM (ou MPI), que acrescenta funcionalidades (*wrapper*) à API original. As mensagens de controle dos processos são implementadas no mesmo nível das mensagens da aplicação. O CoCheck utiliza a técnica de *checkpointing* coordenado, onde um processo especial envia uma notificação para todos os processos da aplicação. De novo, para garantir a consistência do estado global, o *checkpoint* só é realizado quando não existe nenhuma mensagem sendo transmitida nos canais de comunicação. Para isso, cada processo da aplicação ao receber a notificação de *checkpoint* envia uma mensagem RM (*Ready Messages*) para todos os outros processos. Depois de receber todas as mensagens RM, um processo pode efetuar o seu *checkpoint*. Como não existem mensagens sendo transmitidas no momento do *checkpoint*, então os processos podem ser vistos de forma isolada na criação dos seus *checkpoints*. Além disso, no retorno dos processos para o último *checkpoint* armazenado não será necessário o reenvio de mensagens. Entretanto, o custo de se fazer o *checkpoint* fica maior, especialmente para muitos processos.

Egida (RAO; ALVISI; VIN, 1999, 2000; STRANDTMAN, 2010) é um conjunto de ferramentas orientada a objetos desenvolvida para fornecer, de forma transparente, recuperação

por retorno (*rollback*) baseado em *log* de mensagens. Egida é construída sobre uma biblioteca de objetos que implementam um conjunto de funcionalidades. Estas funcionalidades são o núcleo de todos os protocolos de recuperação, que são baseados em *log* pessimista, otimista ou causal. Além disso, os usuários ou programadores podem desenvolver o seu próprio protocolo de recuperação, pois foi definida uma linguagem de especificação para configurar protocolos a partir da biblioteca de objetos. Egida foi integrada pelos seus autores com a implementação MPICH da biblioteca MPI. Entretanto, a detecção e a recuperação de falhas ou erros foi implementada na camada mais baixa do MPICH. Assim, para as aplicações MPI existentes conseguirem suporte para a tolerância a falhas fornecidas por Egida é necessária a compilação da aplicação com Egida e sua biblioteca MPICH modificada.

Starfish (AGBARIA; FRIEDMAN, 2003; FRIEDMAN; AGBARIA, 2010) é um ambiente para execução de programas MPI que também fornece suporte para tolerância a falhas. Cada nó executa um *daemon* starfish. O conjunto destes *daemons* formam o ambiente paralelo Starfish, e eles são responsáveis pela criação de processos da aplicação, pela detecção de falhas da aplicação, pelo gerenciamento da configuração do sistema, pela interação com os usuários e por fornecer meios necessários para oferecer tolerância a falhas para a aplicação. Starfish apresenta uma arquitetura flexível e portável, permitindo diferentes implementações de protocolos de *checkpoint/restart*. O usuário pode escolher entre as estratégias de *checkpoint* coordenado e não coordenado. Entretanto, para o usuário usufruir de todos os benefícios de Starfish é necessário conhecer as novas funcionalidades adicionadas ao MPI padrão e decidir a melhor estratégia de tolerância a falhas a ser utilizada.

O MPI/FT (BATCHU et al., 2001, 2004) é um *middleware* baseado em MPI que fornece serviços adicionais para a detecção e a recuperação de processos que falharam. Este *middleware* fornece mecanismos para notificar falhas e reiniciar processos, mas deixa a tarefa de realizar o *checkpoint* e restaurar a execução do último *checkpoint* válido para a aplicação. O MPI/FT trata falhas para aplicações que seguem os estilos Mestre-Trabalhador e SPMD (*Single Program Multiple Data*). Nestes estilos, o fluxo do programa consiste de iterações de computação intercaladas com comunicação. O MPI/FT usa *threads* e mensagens de *heartbeat* para detectar falhas de processos MPI que não respondem. O processo mestre no estilo Mestre-Trabalhador e o rank 0 no estilo SPMD coordenam as funções de detecção e recuperação para a aplicação inteira e, por esta razão, são conhecidos como *threads* coordenadoras. O MPI/FT adota a noção de redundância de tarefas para fornecer tolerância a falhas. Assim, os processos que falharam são substituídos por

processos extras mantidos ociosos. Os próprios autores em (BATCHU et al., 2004) afirmam que os processos extras ociosos usam poucos ciclos de CPU e memória virtual, mas que esta aproximação parece viável somente para *clusters* de pequeno e médio porte. Uma vez que os processos extras tenham acabado, o MPI/FT não consegue mais recuperar uma falha de processo e, neste caso, a aplicação toda falha. A consistência de *checkpoints* é estabelecida pela participação de todos os processos em uma operação coletiva, ou seja, a operação de *checkpoint* se comporta como um ponto de sincronismo ou barreira.

O MPICH-GF (WOO; YEOM; PARK, 2004; MPICH-GF, 2010) é um sistema tolerante a falhas construído com base no MPICH-G2, implementação MPICH com um novo **dispositivo** desenvolvido para a execução em ambientes de grades. MPICH-GF é totalmente transparente para o desenvolvedor da aplicação e não requer nenhuma alteração no código fonte. Entretanto, um novo **dispositivo** foi implementado com base no existente da versão MPICH-G2. Este novo dispositivo é responsável pelo gerenciamento do grupo de processos dinâmicos, o conjunto de ferramentas de *checkpoint* e pelo gerenciamento da fila de mensagens. Além da modificação sobre MPICH-G2, uma outra desvantagem de MPICH-GF é a alteração de dois módulos do Globus, que são: o DUROC (*Dynamic Updated Request Online Co-allocator*) e os gerenciadores de *job* GRAM (*Globus Resource Allocation Management*). O DUROC distribui o pedido do usuário para módulos GRAM locais. Já os gerenciadores de *job* GRAM criam os processos requisitados. Estes módulos tiveram suas capacidades estendidas através de modificações no código fonte. Sendo o DUROC responsável pela detecção de falhas de *hardware* ou rede e os gerenciadores de *job* por falhas de processos. Eles também são responsáveis pelo *checkpointing* e pela recuperação automáticos. O MPICH-GF utiliza a técnica de *checkpointing* coordenado, onde o processo DUROC envia as requisições de *checkpointing* periodicamente para os gerenciadores de job GRAM. Então, estes gerenciadores sinalizam esta requisição para aos processos da aplicação. Por sua vez, os processos da aplicação realizam uma sincronização para evitar mensagens sendo transmitidas pelos canais de comunicação durante a realização do *checkpoint*. Assim, este sistema sofrerá, para cada *checkpoint* realizado, os custos de sincronizar os processos em ambientes de grades. No processo de recuperação, um novo processo é criado para substituir o que teve falha e os comunicadores são atualizados por uma nova função MPI desenvolvida pelos autores.

O FT-MPI (FAGG; DONGARRA, 2000, 2004; FAGG et al., 2005; ICL Team, 2010) é uma nova implementação de MPI tolerante a falhas que tem como meta oferecer ao usuário final uma biblioteca de comunicação através de uma API MPI. O FT-MPI é construído sobre o sistema HARNESS (Heterogeneous Adaptive Reconfigurable Networked SyStem) (FAGG;

BUKOVSKY; DONGARRA, 2001; ICL Team, 2010) e utiliza os benefícios da tolerância a falhas fornecidos por este sistema. O FT-MPI não realiza *checkpoint* ou *log* de mensagens. Ao invés disso, as funcionalidades dos comunicadores são estendidas para fornecer informações que permitirão a uma aplicação tomar a ação corretiva apropriada. A desvantagem é a falta de transparência para o programador, já que este é quem deve decidir como tratar a falha.

Reed, Lu e Mendes em (REED; LU; MENDES, 2006) estenderam a biblioteca LA-MPI (GRAHAM et al., 2003) pela adição de *checkpoints* em memória volátil (*diskless checkpoints*), que são gerenciados por uma biblioteca modificada. Para este tipo de *checkpoint*, os processadores (nós) utilizados pela aplicação foram particionados em grupos. Cada grupo contendo os nós que executam o código da aplicação e alguns nós extras. Quando o *checkpoint* da aplicação é realizado, os dados do *checkpoint* são gravados na memória local de cada nó com dados redundantes sendo gravados nas memórias dos nós extras atribuídos para cada grupo. Se um nó falha, a camada MPI modificada direciona a aplicação para executar a recuperação por retorno e reiniciar os processos que falharam nos nós extras. A recuperação da aplicação é realizada usando os dados recuperados dos *checkpoints* e das informações redundantes. Esta abordagem limita o número de falhas dentro de cada grupo para o número de nós extras e diminui o número de nós que poderia ser utilizado na execução da aplicação do usuário.

Adaptive MPI (AMPI) (HUANG et al., 2006; HUANG; ZHENG; KALÉ, 2007) implementa processos MPI virtuais migráveis utilizando *threads* no nível do usuário. Tipicamente, programas MPI dividem a computação dentro de P processos e executam cada processo em um dos P processadores. Ao contrário disso, no AMPI a computação é dividida em um número V de processadores virtuais e estes, em tempo de execução, são mapeados para os P processadores físicos. O número de processadores P e V são independentes, facilitando a implementação da aplicação. Em (ZHENG; HUANG; KALÉ, 2006) são apresentados dois protocolos de tolerância a falhas baseados na técnica de *checkpoint* coordenado em disco e memória volátil projetados para Charm++ (KALE; KRISHNAN, 1993) e AMPI. Esses esquemas de *checkpoint* da aplicação podem ser realizados em dois níveis: totalmente automatizados ou controlados pelo usuário através de funções auxiliares. O mecanismo de *checkpoint* sendo totalmente automatizado não requer esforço do programador ou usuário. Entretanto, uma maior eficiência é obtida com o envolvimento do usuário, já que o programador pode escolher quais os dados são utéis para serem gravados, pois este conhece que parte do processo precisa ser salvo.

O projeto MPICH-V (BOSILCA et al., 2002; BUNTINAS et al., 2008; MPICH-V, 2010) visa fornecer uma implementação MPI, baseada em MPICH, que ofereça múltiplos protocolos de tolerância a falhas para aplicações MPI. Atualmente, existem 5 protocolos desenvolvidos (MPICH-V, 2010): 2 protocolos de *log* de mensagens pessimista com *checkpointing* não coordenado, 1 protocolo de *log* de mensagens causal com *checkpointing* não coordenado e 2 protocolos de *checkpointing* coordenado com base no algoritmo de Chandy-Lamport (CHANDY; LAMPORT, 1985). Uma abstração usada pelo MPICH é a noção de um **dispositivo** (*device*), que implementa as rotinas básicas de comunicação para um *hardware* específico ou para novos protocolos de comunicação. O MPICH-V é composto por um conjunto de componentes criados em tempo de execução e um dispositivo implementado para a biblioteca MPICH. Este dispositivo confia na separação entre a aplicação MPI e o sistema de comunicação efetivo (BUNTINAS et al., 2008). A tolerância a falhas é realizada pela implementação de subrotinas nas rotinas relevantes de comunicação. Assim, os diferentes protocolos de tolerância a falhas são implementados entre a camada de gerenciamento de protocolo (gerenciamento de operações globais, protocolos ponto a ponto, entre outros) no nível mais alto do MPI e a camada de transporte no nível mais baixo (BOUTEILLER et al., 2006).

A implementação LAM/MPI (SANKARAN et al., 2003, 2005; LAM Team, 2010) incorporou o sistema de *checkpoint* de processo no nível do kernel do Linux chamado Berkeley Laboratory Checkpoint/Restart (BLCR) (HARGROVE; DUELL, 2006). Este sistema fornece *checkpoints* coordenados para as aplicações paralelas MPI sem nenhuma modificação no código da aplicação. O *checkpointing* de uma aplicação é iniciado por um usuário ou um escalonador através de um utilitário do BLCR que envia um pedido de *checkpoint* para o processo *mpirun*, que atua como um ponto de coordenação entre todos os processos de uma aplicação MPI. No recebimento desta requisição, o processo *mpirun* propaga este pedido para todos os processos MPI. Quando os processos recebem a solicitação de *checkpoint*, todos os processos MPI interagem com cada outro para garantir que os seus *checkpoints* locais resultarão em um estado global consistente. A aproximação adotada em LAM assegura que todos os canais de comunicação MPI entre os processos estão vazios quando um *checkpoint* é realizado. Assim, não é necessário gravar o estado dos canais de comunicação, mas somente o estado dos processos da aplicação. Durante a fase de reinício, um utilitário do BLCR é executado com a localização do arquivo de contexto que contém a informação de onde os arquivos de *checkpoints* foram armazenados, e todos os processos retornam sua execução a partir do seu estado guardado, sendo que os canais de comunicação são restaurados para seus estados vazios. O *middleware* SGA EasyGrid

utiliza a implementação LAM/MPI, mas esta solução não atende ao SGA EasyGrid, pois os processos da aplicação possuem granularidade fina e são criados dinamicamente.

Como dito anteriormente, o Open MPI é uma implementação da biblioteca MPI resultante da participação de projetos, tais como LAM/MPI (SANKARAN et al., 2005), LA-MPI (GRAHAM et al., 2003) e FT-MPI (FAGG; DONGARRA, 2000). Similar ao LAM/MPI, o Open MPI tem suporte ao sistema de *checkpoint* BLCR (HARGROVE; DUELL, 2006) e ao "self", que permite aplicações realizar sua própria funcionalidade de *checkpoint*/reinício. Para ambos os sistemas, o Open MPI fornece um protocolo de *checkpoint* coordenado e uma integração com uma variedade de interconexões de rede incluindo Ethernet, InfiniBand, Myrinet e memória compartilhada para processos que compartilham o mesmo recurso, não limitando o desempenho das aplicações em ambientes *multicore* (HURSEY et al., 2007; HURSEY; MATTOX; LUMSDAINE, 2009).

No caso do SGA EasyGrid, uma biblioteca `EGAMSmpi.h` foi desenvolvida para substituir o `mpi.h`. Esta biblioteca ativa um sistema de gerenciamento através de um código de abstração (*wrapper*) que estende a capacidade das funções MPI chamadas pela aplicação. As novas funcionalidades, que serão descritas com mais detalhes na próxima seção, são incluídas dentro do programa do usuário em tempo de compilação, sem a necessidade de alterações no código fonte. O SGA EasyGrid oferece mais do que somente autorrecuperação. A tolerância a falhas foi integrada com os mecanismos que fornecem suporte as outras propriedades autônomas (*self*-*) para prover um efetivo sistema de autogerenciamento de baixa intrusão.

3.2 Sistema de Gerenciamento de Aplicações EasyGrid

Aplicações autônomas são programas adaptativos, tolerante a falhas e autoescalonáveis capazes de reagir às mudanças que ocorrem em ambientes distribuídos, instáveis, dinâmicos e compartilhados, tal como grades computacionais. O *framework* EasyGrid (BOERES; REBELLO, 2004) é responsável pela transformação automática de uma aplicação paralela em uma aplicação autônoma. Para atingir esse objetivo, em cada aplicação é embutido um *middleware* de serviço específico para a aplicação na forma de um Sistema de Gerenciamento de Aplicações (SGA) (VIANNA, 2005). Assim, a aplicação passa a ser capaz de usar os recursos disponíveis na grade de forma eficiente. O SGA EasyGrid é um sistema gerenciador de aplicações MPI, que utiliza a biblioteca LAM/MPI. Esta biblioteca foi escolhida por fornecer suporte à criação dinâmica de processos e aos serviços forne-

cidos pelo Globus Toolkit para a execução no ambiente grade. Nenhuma modificação é realizada na distribuição padrão ou instalação do LAM/MPI por razões de portabilidade, o que é importante em ambientes de grades.

3.2.1 Modelo de Execução

Tipicamente, em uma aplicação MPI estática, espera-se que sejam criados um processo por processador ou núcleo e que estes processos executem uma função de inicialização do MPI antes de começarem o processamento do programa paralelo. Porém, o SGA Easy-Grid adota o modelo de execução "**um processo por tarefa da aplicação (1PTask)**", proposto em (SENA et al., 2007; SENA, 2008). Neste modelo, os programas consistem de um grande número de processos de curta duração que são determinados pelo paralelismo da aplicação e não pelo número de recursos. Processos de granularidade fina são favorecidos a fim de maximizar o paralelismo disponível na aplicação, consequentemente, uma aplicação pode ser formada por centenas ou milhares de tarefas. A Figura 3.2 mostra que cada tarefa da aplicação é executada em um processo distinto. Entretanto, nem todos os processos da aplicação são criados na inicialização do programa, onde e quando os processos são executados é determinado pelo SGA.

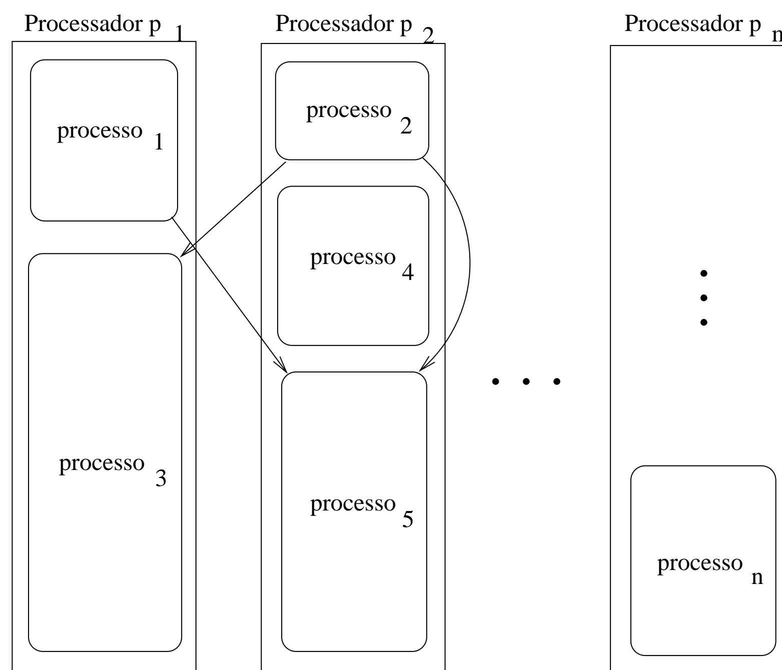


Figura 3.2: Modelo de execução 1PTask (SENA, 2008).

O SGA EasyGrid utiliza este modelo de execução e propõe adotar uma estratégia de tolerância a falhas baseada somente em *log* de mensagens para processos da aplicação.

Os benefícios obtidos de evitar a necessidade de implementar sofisticados esquemas de *checkpointing* e manter longos *log* de mensagens compensam o custo de gerenciar um maior número de processos. Além disso, este modelo permite que processos gerenciadores da grade reescalonem as tarefas da aplicação, que ainda não foram criadas, com o objetivo de adaptar as mudanças de carga de trabalho na aplicação e do poder computacional oferecidos pelos recursos durante a execução. Desta forma, o SGA EasyGrid visa tornar a execução da aplicação paralela do usuário eficiente e robusta.

3.2.2 Modelo da Aplicação

No modelo de execução 1PTask, uma aplicação paralela é constituída por um conjunto de (possivelmente milhares de) tarefas, que se comunicam através de troca de mensagens. O número de tarefas é independente do número de processadores, mas sim dependente do grau de paralelismo do programa. O termo tarefa é usado para referenciar um tipo específico de processo, como mostrado na Figura 3.3, que possui três fases: recebimento de dados de entrada, caso seja necessário; processamento dos dados; e o envio de dados para outras tarefas.

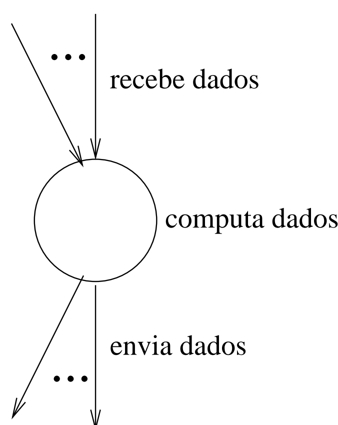


Figura 3.3: Exemplo de tarefa do modelo 1PTask (SENA, 2008).

O SGA EasyGrid adota o modelo GAD (Grafo Acíclico Direcionado) para representar as aplicações paralelas (SENA, 2008). Um GAD é denotado por G = (V, E, ε, ω), onde V é o conjunto de nós do grafo e E é o conjunto de arestas direcionadas. Cada nó do GAD representa uma tarefa da aplicação e cada aresta direcionada representa a relação de precedência e a comunicação entre duas tarefas. Para cada nó $v \in$ V, é associado um peso de execução, $\varepsilon(v)$, que indica a quantidade máxima de trabalho a ser realizada pela tarefa v. Enquanto para cada aresta direcionada $(u,v) \in$ E, é associado um custo de comunicação, $\omega(u, v)$, que representa a quantidade de dados a ser enviada de u para v.

Além disso, seguindo as três fases de uma tarefa no modelo 1PTask, a tarefa v somente pode começar a sua execução depois de receber os dados vindos da tarefa u. Um exemplo de GAD é mostrado na Figura 3.4, onde as tarefas possuem pesos de execução e as arestas direcionadas mostram os custos de comunicação. Como ilustrado neste exemplo, a tarefa 1 deve completar sua fase de processamento antes que a tarefa 2 comece sua execução (relação de precedência).

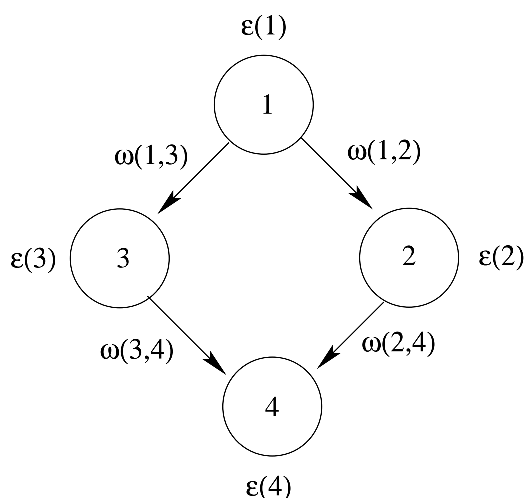


Figura 3.4: Exemplo de GAD (SENA, 2008).

As aplicações paralelas podem possuir tarefas determinísticas ou não-determinísticas. As tarefas determinísticas são aquelas que recebendo os mesmos dados de entrada sempre produzem os mesmos dados de saída. No caso do SGA EasyGrid, as tarefas determinísticas recebendo as mesmas mensagens de comunicação (dados de entrada), numa mesma ordem de chegada, produzem os mesmos dados de saída e estes são sempre enviados na mesma ordem. As tarefas determinísticas e suas mensagens são representadas no modelo GAD. Entretanto, para as tarefas não-determinísticas, onde seus pesos e comunicação podem não serem conhecidos a priori, o SGA EasyGrid gerenciará estas tarefas em tempo de execução.

3.2.3 Modelo Arquitetural

O modelo arquitetural especifica as características da arquiterura do ambiente computacional, onde o programa paralelo será executado. O SGA EasyGrid considera que uma grade computacional é formada por um conjunto limitado de processadores heterogêneos com sua própria memória local e interconectados segundo alguma topologia. Seja P o conjunto limitado de processadores heterogêneos. Para cada processador $p \in P$ é

determinado um índice de retardo computacional (*computational slowdown index*) $csi(p)$. O índice de retardo computacional de um processador é um valor que representa a sua capacidade de processamento, considerando o processador dedicado a aplicação do usuário. Para o cálculo estimado do tempo de execução (te) de uma tarefa $v \in V$ escalonado em um processador $p \in P$, multiplica-se o peso da tarefa v pelo csi do processador p, como mostrado na Equação 3.1 (SENA, 2008):

$$te(v, p) = \varepsilon(v) \times csi(p) \tag{3.1}$$

O csi de cada processador p é obtido através de um programa modelador, como mostrado em (MENDES, 2004; BOERES et al., 2006). O modelador executa uma tarefa com as mesmas características ou similar da tarefa da aplicação a ser executada e retorna o valor do csi para um determinado processador. Quanto menor for o valor do csi melhor será o desempenho do processador. Caso não seja possível usar uma tarefa com características similares, por alguma razão, o modelador desenvolvido em (MENDES, 2004) disponibiliza um código formado por um conjunto de instruções de diversos tipos para estimar o valor do csi.

A comunicação entre as tarefas da aplicação é realizada através de troca de mensagens. Dadas duas tarefas u e v que se comunicam, o custo total da comunicação é calculado considerando uma sobrecarga de envio no processador onde u foi alocado e uma sobrecarga de recebimento no processador em que v foi alocado. As sobrecargas de envio e recebimento variam em função do tamanho da mensagem e da capacidade de processamento do processador (SENA, 2008). Além destas sobrecargas, também é considerada a latência de comunicação $l(\omega(u,\ v))$ que depende da velocidade do canal de comunicação e varia de acordo com o tamanho da mensagem.

3.2.4 Gerenciamento dos processos da aplicação no SGA

O SGA controla a execução dos processos da aplicação MPI através de uma hierarquia distribuída de processos gerenciadores. Como mostra a Figura 3.5, uma grade computacional é formada por vários conjuntos de recursos geograficamente distribuídos. Cada conjunto de recursos é denominado de *site*. A estrutura hierárquica adotada no SGA é composta por três níveis de gerenciadores, como mostra a Figura 3.5. No topo da hierarquia (nível 0), um único **Gerenciador Global (GG)** é o responsável por supervisionar os *sites* da grade onde os processos da aplicação estão ou serão capazes de executar. O nível

1 corresponde aos processos **Gerenciadores do *Site* (GS)** que gerenciam a execução dentro do seu respectivo *site*, onde só existe um único GS em cada *site*. Finalmente, no nível mais baixo da hierarquia (nível 2), **Gerenciadores da Máquina (GM)**, um para cada recurso, possuem a responsabilidade por escalonar, criar e executar os processos da aplicação alocados para sua respectiva máquina.

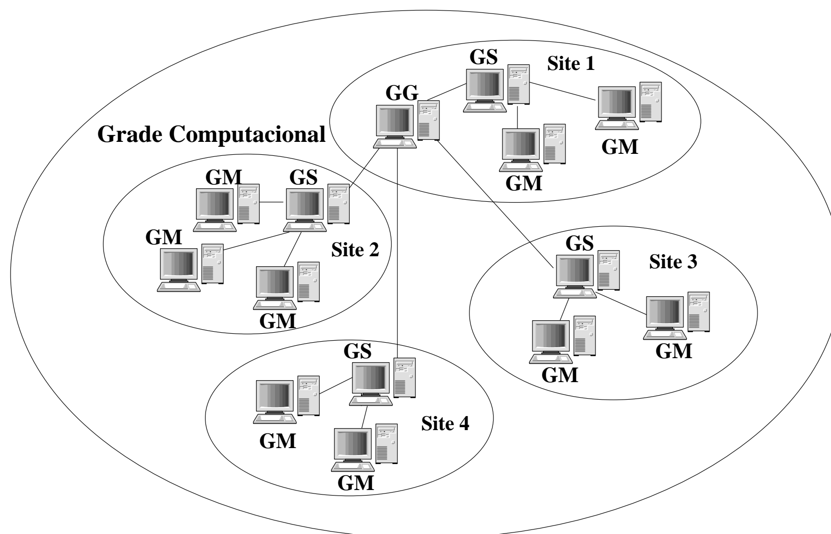


Figura 3.5: A hierarquia do SGA EasyGrid.

Como mostra a Figura 3.6, cada processo gerenciador é modelado seguindo uma arquitetura de camadas, denominada *subsumption architecture* (BROOKS, 1986), onde cada camada é responsável por um comportamento ou serviço específico. Nesta arquitetura, cada camada adiciona um novo nível de competência e os níveis mais altos influenciam as funções dos níveis mais baixos quando eles desejam mudar o comportamento da aplicação. Cada nível examina os dados fornecidos pela camada de monitoramento e retorna informações para modificar o comportamento normal do nível inferior (autoconfiguração). As camadas de nível inferior executam sem conhecer a existência das camadas mais altas. A funcionalidade de cada camada também depende do nível do processo gerenciador na hierarquia. Esta estrutura hierárquica permite a aplicação adaptar-se às mudanças do ambiente de forma independente, já que cada processo gerenciador pode adotar diferentes políticas dinâmicas (NASCIMENTO et al., 2005, 2007; NASCIMENTO, 2008).

A camada de gerenciamento de processos é responsável pela criação dinâmica de processos MPI, isto é, os processos gerenciadores e os da aplicação, e pelo redirecionamento de mensagens entre os processos. A camada de automonitoramento da aplicação coleta dados do sistema e fornece informações relevantes sobre o comportamento e desempenho da aplicação para as camadas de escalonamento dinâmico (NASCIMENTO et al., 2007; NASCIMENTO, 2008), responsável pela redistribuição dos processos da aplicação, e de

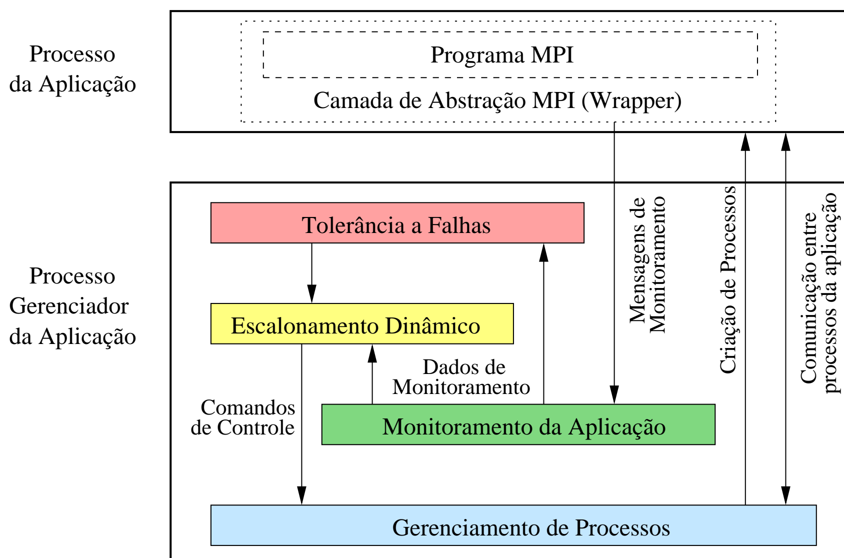


Figura 3.6: Arquitetura em camadas do SGA EasyGrid.

tolerância a falhas (SILVA; REBELLO, 2007), responsável pela identificação e recuperação dos processos que falharam. As mensagens de monitoramento são enviadas sempre que um processo da aplicação do usuário termina sua execução, indicando o seu término para os gerenciadores, e incluem o percentual de utilização da CPU, a quantidade de processos executando naquele recurso no momento e o tempo estimado para finalizar a execução da aplicação no recurso, entre outras informações. Já os processos da aplicação possuem a camada da aplicação MPI do usuário que estimula as funções dos gerenciadores de forma transparente através da camada de abstração MPI (*wrapper*).

Todos os processos MPI, exceto o GG e incluindo os da aplicação, são criados dinamicamente segundo a estrutura hierárquica de gerenciadores, onde cada processo filho possui um único comunicador individual com o processo pai. Visto que os processos da aplicação original não podem mais se comunicar diretamente com cada outro, os GM's, GS's e o GG realizam o trabalho de redirecionar as mensagens entre os processos da aplicação na mesma máquina, no mesmo site, e entre *sites*, respectivamente. Como nem todos os processos da aplicação são criados na inicialização, os gerenciadores também precisam guardar as mensagens de entrada e enviá-las só depois que o processo receptor tenha sido criado. Entretanto, por motivos de tolerância a falhas, todos os processos gerenciadores armazenam suas mensagens recebidas em um *log* de mensagens até terminarem sua execução com sucesso. Maiores detalhes do mecanismo de tolerância a falhas são apresentados na Seção 4.2 do Capítulo 4.

Os GS's também são responsáveis pela comunicação entre os GM's e o GG, redirecionando não somente mensagens da aplicação, mas também mensagens de controle do SGA,

tal como mensagens de monitoramento, escalonamento e erro. Note que os processos gerenciadores precisam rotear as mensagens dinamicamente, já que os processos destinos podem ser realocados pelo escalonador dinâmico em resposta para a autorrecuperação (falha de processos) ou mudanças no poder computacional disponível nos recursos.

3.3 Resumo

Este capítulo apresentou algumas características da biblioteca MPI (*Message Passing Interface*). Dentre elas, que o comportamento padrão do MPI é que falhas são fatais, ou seja, a biblioteca MPI não tem suporte nativo de tolerância a falhas. Este comportamento ocorre devido ao MPI ter sido inicialmente desenvolvido para ambientes estáticos, homogêneos e menos propensos a falhas. Outra característica é que as aplicações MPI são tipicamente desenvolvidas para executar processos de longa duração e o modelo de execução utilizado nesse trabalho é "**um processo por tarefa da aplicação (1PTask)**", onde os programas consistem de um grande número de processos de curta duração que são determinados pelo paralelismo da aplicação e não pelo número de recursos, que é um modelo mais eficiente do que o modelo 1PProc para ambientes heterogêneos, compartilhados e mais propensos a falhas, tal como os ambientes de grades. Também foram mostrados alguns trabalhos que visam oferecer tolerância a falhas para as aplicações MPI. Entretanto, os trabalhos mostrados fornecem tolerância a falhas sem transparência para o usuário/programador ou através de modificações nas implementações de MPI existentes. As exceções são: o LAM/MPI, que não atende ao SGA EasyGrid devido a granularidade fina dos processos da aplicação e a constante criação dinâmica; e o ambiente CoCheck, que foi implementado para uma nova versão de MPI, chamada tuMPI, mas segundo os autores pode ser adaptada para outras implementações de MPI. Entretanto, em (STELLNER, 2010) foi divulgado que não será mais oferecido suporte para a versão MPI. Por fim, foi apresentada uma descrição do SGA EasyGrid com sua hierarquia de processos e sua arquitetura em camadas dos processos gerenciadores. No próximo capítulo serão descritos os mecanismos de tolerância a falhas desenvolvidos no SGA EasyGrid.

Capítulo 4

Tolerância a Falhas no SGA EasyGrid

Este capítulo descreve os mecanismos de tolerância a falhas desenvolvidos no SGA EasyGrid. Os objetivos deste capítulo são definir na Seção 4.1 o modelo de falhas utilizado, apresentar na Seção 4.2 os mecanismos de tolerância a falhas empregados, e descrever um estudo de caso com a implementação dos mecanismos em aplicações MPI desenvolvidas na versão LAM/MPI, na Seção 4.3. Os mecanismos de tolerância a falhas visam garantir o funcionamento da aplicação paralela mesmo no caso das falhas definidas no modelo utilizado, assim tanto a aplicação quanto o próprio *middleware* EasyGrid precisam ser cobertos por estes mecanismos.

4.1 Modelo de Falhas Utilizado

Esta seção pretende deliminar os tipos de falhas que o *middleware* EasyGrid deve tratar. Sem este modelo de falhas, é impossível avaliar a eficácia dos mecanismos de tolerância a falhas implementados no SGA EasyGrid.

Como mencionado na Seção 3.2.3, que descreve o modelo arquitetural, o SGA EasyGrid considera que uma grade computacional é formada por um conjunto limitado de processadores heterogêneos com sua própria memória local e interconectados segundo alguma topologia. Já o modelo da aplicação, descrito na Seção 3.2.2, define uma aplicação paralela formada por um conjunto de tarefas, que se comunicam através do envio e o recebimento de mensagens. Segundo (GRAHAM et al., 2003), o padrão MPI assume que os dados chegam intactos ao seu destino e que a maioria das implementações usam o protocolo TCP/IP para garantir isto. Assim, cada par de processos é conectado por um canal de comunicação confiável, ou seja, nenhuma mensagem será perdida, corrompida ou

duplicada. Além disso, o sistema é assíncrono, ou seja, não existe um limite na velocidade dos processos e nem no retardo da transmissão das mensagens.

Na definição do modelo de falhas são consideradas as características apresentadas na Seção 2.2.2, que são elas: manifestação da falha, duração, fonte da falha, perfil de ocorrência e granularidade.

Considerando a manifestação da falha, que descreve como o sistema se comportará na presença de falhas, o SGA EasyGrid trata falhas do tipo *fail-stutter*, que inclui falhas de desempenho e do tipo colapso. Entretanto, este trabalho de doutorado só envolve as falhas do tipo colapso, que englobam as falhas do tipo *fail-stop*. As falhas de desempenho são tratadas pela camada de escalonamento dinâmico (NASCIMENTO et al., 2007; NASCIMENTO, 2008; RODRIGUES, 2009). Quando um determinado processador começa a diminuir o seu desempenho, ou seja, os processos executados no processador passam a gastar mais tempo para finalizar a sua execução, o escalonador dinâmico (NASCIMENTO et al., 2007; NASCIMENTO, 2008) detecta a mudança através da camada de monitoramento. Com isso, um novo escalonamento é gerado, levando em consideração o novo desempenho dos processadores, e os processos ainda não executados são reescalonados com o objetivo de minimizar o tempo de execução total da aplicação. Assim, as falhas de desempenho são tratadas pelo ajuste da carga de trabalho entre os processadores disponíveis. Os detalhes das heurísticas de escalonamento dinâmico adotadas no SGA EasyGrid e a integração entre a camada de escalonamento dinâmico e a camada de monitoramento podem ser obtidos em (NASCIMENTO, 2008).

Este trabalho foca nas situações onde os processos podem falhar por colapso. Segundo, a definição de falha por colapso, os processos com este tipo de falha nunca se recuperam. Assim, dada que uma falha seja detectada, a duração da falha sempre é considerada permanente.

A fonte da falha pode ser de infraestrutura ou da aplicação. A falha de infraestrutura, denominada neste trabalho de **falha irrecuperável**, inclui a perda de conexão com uma máquina utilizada pelo SGA EasyGrid, por exemplo, quando a máquina é desligada; e a perda do *daemon* responsável pelo gerenciamento dos processos MPI, pois a recriação de um novo *daemon* não é possível com o módulo Globus da implementação LAM/MPI. Já a falha da aplicação, denominada de **falha recuperável**, ocorre quando um processo gerenciador ou da aplicação tem uma falha, que no modelo utilizado corresponde ao processo ser abortado ou morto. Como o recurso continua disponível, o processo pode ser recriado e continuar sua execução no mesmo recurso.

O perfil de ocorrência das falhas é de considerar falhas de colapso que podem ocorrer a qualquer momento, tanto nos processadores quanto na aplicação, o que envolve os processos gerenciadores do SGA EasyGrid e os processos da aplicação. Supõe-se que a aplicação do usuário foi desenvolvida corretamente. No modelo deste trabalho, quando uma falha ocorre no processo, este é considerado abortado ou morto.

Com relação a granularidade, que é o tamanho do componente comprometido pela falha, uma falha pode atingir um processo da aplicação ou um processo gerenciador do SGA, que terá a falha tratada pelo gerenciador de nível acima responsável pelo processo que falhou. No caso do Gerenciador Global, que não tem um gerenciador em um nível acima dele, o portal da grade ou o sistema responsável por submeter o *job* é que deverá oferecer o suporte de tolerância a falha para o Gerenciador Global. Maiores detalhes sobre os mecanismos de tolerância a falhas serão descritos nas próximas seções.

4.2 Mecanismos de Tolerância a Falhas no SGA EasyGrid

Esta seção descreve os mecanismos de tolerância a falhas empregados neste trabalho. Os mecanismos de tolerância a falhas desenvolvidos no SGA EasyGrid seguiram os 4 estágios apresentados na Seção 2.2.3, que são eles: detecção da falha/erro, localização da falha, contenção da falha e recuperação da falha/erro. Para oferecer um serviço completo de tolerância a falhas (TF) para a aplicação do usuário foi necessário a elaboração de mecanismos de TF para os processos da aplicação, bem como, para os processos gerenciadores do SGA.

O SGA EasyGrid pretende usar uma combinação das quatro técnicas de tratamento de falhas no nível de tarefas, apresentadas em 2.2.3, para os processos da aplicação e seus gerenciadores. O SGA EasyGrid implementa a técnica de **tentar novamente** e **recurso alternativo** junto com um *log* de mensagens para todos os processos da aplicação, em execução ou esperando para executar, fornecendo um meio para a recuperação de falha de processos. Já para os gerenciadores do SGA é implementada a técnica de **checkpoint/reinício** no nível da aplicação, onde os gerenciadores de nível acima possuem as informações necessárias para a recuperação dos gerenciadores de nível inferior. O SGA EasyGrid possui uma versão em desenvolvimento com a utilização da técnica de **replicação** passiva (SARDIÑA; BOERES; DRUMMOND, 2006; SARDIÑA, 2010), que está fora do escopo deste trabalho, onde réplicas das tarefas são escalonadas pelo escalonador estático

de forma que se um processador falhe as tarefas réplicas são criadas e ativadas em um outro recurso pelo SGA. Entretanto, ainda não foi implementada a replicação ativa no SGA EasyGrid, em que as réplicas das tarefas são criadas e executadas junto com as tarefas originais independentemente de acontecer ou não uma falha. A replicação ativa deve ser usada junto com a camada de escalonamento dinâmico para escalonar as réplicas de forma a reduzir os custos de comunicação e melhorar o desempenho (SILVA, 2002; FREIRE, 2003). A replicação ativa também pode ser aproveitada pela camada de tolerância a falhas, onde a réplica substitui a tarefa original em caso de falha.

Este trabalho propõe uma maneira distribuída, através dos gerenciadores do SGA EasyGrid, de simultaneamente detectar, recuperar e se adaptar as falhas irrecuperáveis e recuperáveis em ambientes grades para aplicações MPI baseadas em LAM/MPI. A seguir serão descritos os mecanismos de tolerância a falhas desenvolvidos para cada nível de gerenciadores e as tarefas da aplicação. Como mencionado na Seção 3.2.2, o termo tarefa é usado para referenciar um tipo específico de processo, que possui três fases: recebimento de dados de entrada, caso seja necessário; processamento dos dados; e o envio de dados para outras tarefas. Os termos tarefas e processos serão usados indistintamente no restante desta tese, já que no modelo de execução 1PTask, os processos da aplicação sempre seguem o mesmo comportamento de uma tarefa.

4.2.1 Tolerância a Falhas para Processos da Aplicação

Seguindo a hierarquia de gerenciadores do SGA EasyGrid, a responsabilidade para a detecção e a recuperação de falhas ocorridas nos processos da aplicação é do Gerenciador da Máquina (GM) que criou o processo. As técnicas de detecção e recuperação de falhas no processo da aplicação foram apresentadas em (NASCIMENTO et al., 2005).

Para a detecção de falhas, o Gerenciador da Máquina utiliza as mensagens enviadas pelos processos da aplicação para saber se estes estão vivos, como ilustrado na Figura 4.1. Além disso, a camada de abstração MPI (*wrapper*), mostrada na Figura 3.6, garante o envio de uma mensagem para a camada de monitoramento informando a finalização de um processo. Assim, o Gerenciador da Máquina sempre receberá pelo menos uma mensagem de cada processo da aplicação. A falta de informação sobre um determinado processo indica sua falha para o GM. O tempo de espera máximo, que o GM aguarda por uma informação de um processo da aplicação, está relacionado com o peso da tarefa informado no GAD que representa a aplicação paralela. Desta forma, um GM consegue detectar uma falha no processo da aplicação de forma rápida, já que os processos no modelo de

execução 1PTask são, relativamente, de curta duração. Como nem sempre os pesos das tarefas são conhecidos a priori, transcorrido o tempo de espera máximo, o GM faz uma consulta local sobre o estado do processo (detalhes da implementação serão mostrados na Seção 4.3.1). Observando que o Gerenciador da Máquina responsável pelos processos da aplicação sempre estará na mesma máquina, o que elimina a necessidade de detectar as falhas de conexão de rede.

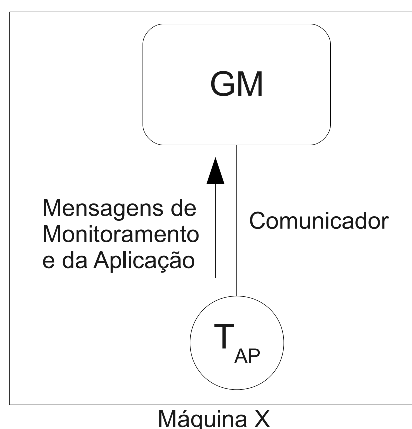


Figura 4.1: Comunicação de um processo da aplicação com o seu GM.

Após a detecção da falha, a localização e a contenção da falha são imediatos, pois pelo projeto do SGA EasyGrid o processo da aplicação se comunica diretamente e exclusivamente com o Gerenciador da Máquina que o criou. Assim, o GM consegue identificar o processo que falhou e evitar a propagação do erro isolando este processo do resto da aplicação.

O SGA EasyGrid utiliza o modelo de execução 1PTask, onde os programas consistem de um grande número de processos de curta duração que são determinados pelo paralelismo da aplicação e não pelo número de recursos. Os processos em execução não são migrados pela camada de escalonamento dinâmico e somente reexecutados em caso de falha. Somente os processos da aplicação que ainda não foram criados pelo SGA é que são reescalonados para se adaptar a mudança de desempenho dos processadores. Esta decisão foi tomada para evitar o custo de salvar e migrar o contexto ou o *checkpoint* dos processos. Por esta mesma razão, em princípio, no processo de recuperação da falha não é utilizada a técnica de *checkpoint*/reinício para os processos da aplicação, pois foi considerado que o custo de gravar e manter seguro o estado de uma grande quantidade de processos em um ambiente distribuído seria alto em comparação com o custo da reexecução dos processos.

Para o caso de tarefas determinísticas, o SGA implementa um esquema de gerenciamento de *log* de mensagens para manter as mensagens de entrada de um processo

da aplicação até que este termine sua execução com sucesso. Ou seja, para diminuir o consumo de memória, a cópia das mensagens é mantida somente para processos da aplicação em execução ou esperando para serem executados. Cada GM mantém e gerencia o seu próprio *log* de mensagens. Sendo as mensagens mantidas em ordem cronológica de chegada para cada processo da aplicação criado pelo próprio GM. Desta maneira, ao se comprovar a falha de um processo v, o subsistema de tolerância a falhas instruirá o subsistema de gerenciamento de processos para criar um novo processo $v\prime$ para substituir v. Todas as mensagens para $v\prime$ serão recuperadas do *log* de mensagens de v mantido localmente pelo GM. É importante destacar que na reexecução de v, através do processo $v\prime$, o reenvio de mensagens por $v\prime$ que já foram enviadas com sucesso por v antes da falha serão descartadas pelo GM. Se essa ação não fosse realizada, as mensagens duplicadas enviadas por $v\prime$ para o GM seriam redirecionadas para o processo destino, o que causaria uma inconsistência na aplicação paralela. Para evitar as mensagens duplicadas, o GM contabiliza o número de mensagens enviadas por cada processo e, no caso de falha, só serão redirecionadas as mensagens posteriores ao número registrado no GM. Este esquema só é possível para tarefas determinísticas, pois se faz necessário que os mesmos dados de entrada sempre produzam os mesmos resultados e as mesmas mensagens de saída. A ideia deste esquema é que os benefícios obtidos de evitar a necessidade de implementar sofisticados esquemas de *checkpointing* podem compensar o custo de manter uma cópia das mensagens enviadas e o gerenciamento de um maior número de processos.

Existe uma versão específica do SGA EasyGrid, chamada de SGA metaEasyGrid proposta em (ARAÚJO, 2008), que gerencia tarefas não-determinísticas. Nesta versão do SGA EasyGrid, desenvolvida para a execução de metaheurísticas paralelas, a quantidade de processos a serem criados não é conhecido inicialmente. O SGA gerencia os processos criados pela aplicação em tempo de execução e a aplicação paralela termina somente quando um determinado critério de parada definido pelo usuário é atingido. Como o comportamento das metaheurísticas variam dependendo da semente e dos números aleatórios gerados pela aplicação, as trocas de mensagens realizadas por um processo em uma execução podem não mais serem reproduzidas numa reexecução. Assim, o *log* de mensagens não pode ser usado na reexecução das tarefas. Em caso de falha, o SGA metaEasyGrid somente reinicializa o processo sem a necessidade de redirecionar as mensagens de entrada a ele (ARAÚJO, 2008), pois as mensagens são requisitadas pelo próprio processo da aplicação em tempo de execução. Efetivamente, um possível resultado do processo que falhou é perdido e um novo processo é executado em seu lugar.

4.2.2 Tolerância a Falhas para Gerenciadores da Máquina

O Gerenciador do *Site* (GS) é o responsável por garantir tolerância a falhas para os GM's e os recursos dentro do seu *site* (SILVA; REBELLO, 2007). Outra função do GS é o escalonamento dinâmico das tarefas da aplicação dentro do *site* (NASCIMENTO et al., 2007; NASCIMENTO, 2008). Como será mostrado a seguir, o subsistema de tolerância a falhas interage com as camadas de monitoramento e de escalonamento dinâmico para fornecer a autorrecuperação dentro do *site*.

Para cada GM, o escalonador dinâmico do *site* recebe informações da camada de monitoramento, tais como poder computacional oferecido pelo recurso de um GM e o tempo restante de execução das tarefas alocadas neste mesmo recurso. Estas informações são enviadas quando um processo da aplicação termina sua execução ou através de mensagens de *heartbeat*. As mensagens de *heartbeat* são enviadas somente quando o tempo de envio da última mensagem de monitoramento entre o GM e o GS ultrapassa um limite de tempo configurado no SGA. Logo, as mensagens de *heartbeat* são usadas no contexto do escalonamento dinâmico tanto para diminuir o intervalo entre o recebimento das informações atualizadas, quanto para a obtenção de informações no caso de não existir processos da aplicação alocados naquele recurso (NASCIMENTO, 2008).

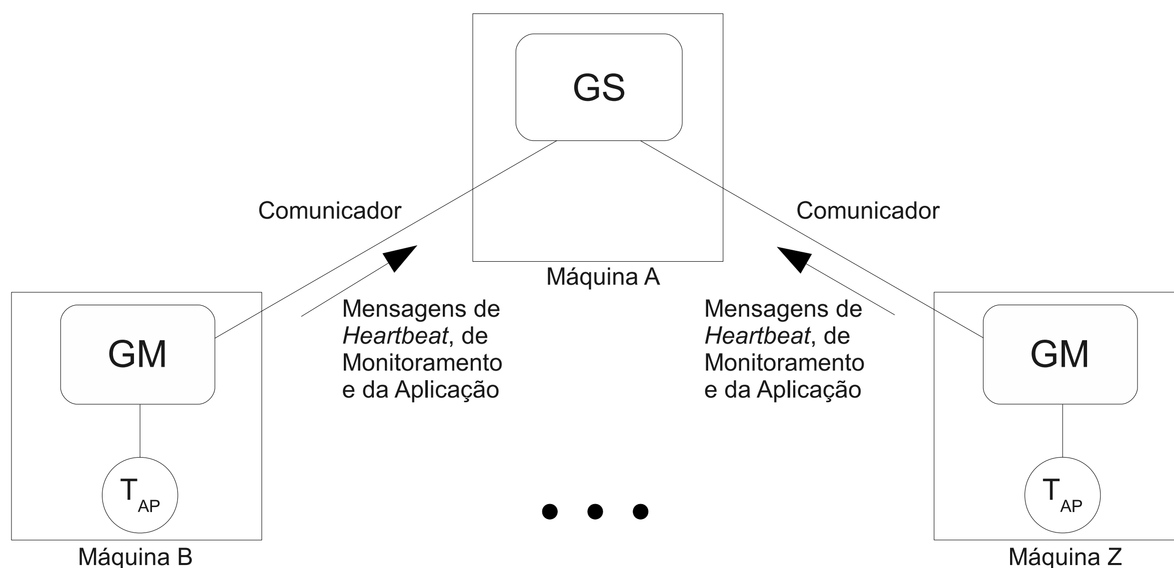


Figura 4.2: Comunicação de um processo GM com o seu GS.

No contexto de tolerância a falhas, o Gerenciador do *Site* utiliza as mesmas mensagens de *heartbeat*, de monitoramento e da aplicação para saber se um GM está vivo, como mostrado na Figura 4.2. Ou seja, a camada de tolerância a falhas também utiliza a estratégia de *heartbeat* (DÉFAGO; HAYASHIBARA; KATAYAMA, 2003) para detectar as falhas dos

GM's, onde cada GM através da camada de monitoramento envia mensagens periódicas para o GS de nível acima. Além disso, as mensagens da aplicação redirecionadas por um GM para o GS também são usadas para identificar que este continua vivo.

Diferente do GM e dos processos da aplicação que sempre estão alocados no mesmo processador, o GS e os GM's normalmente serão alocados em processadores diferentes. Como o sistema utilizado é assíncrono, não existe um limite na velocidade dos processos e nem no retardo da transmissão das mensagens. Desta forma, existe a dificuldade de determinar se um processo realmente falhou por colapso ou somente está muito lento. Para superar este problema, é utilizado neste trabalho o conceito de detectores de falhas não-confiáveis para sistemas com falhas por colapso, proposto por Chandra e Toueg (CHANDRA; TOUEG, 1991, 1996). Como explicado na Seção 2.2.3, o detector de falhas não é confiável, pois pode cometer erros. Os erros são suspeitar da falha de um processo correto ou não detectar a falha de um processo que realmente falhou. Esta tese usa detectores da classe $\diamond W$, denominados de **Detectores de Falhas Eventualmente Fracos** (*Eventually Weak Failure Detectors*)(CHANDRA; TOUEG, 1991), que satisfazem as seguintes duas propriedades:

- **Completude Fraca**: Eventualmente todo processo que falha por colapso é permanentemente suspeito por algum processo correto.
- **Precisão Eventualmente Fraca**: Existe um tempo depois que algum processo correto não é suspeito.

O mecanismo de detecção de falha consiste no GS receber mensagens, da aplicação ou de *heartbeat*, vindas de um GM indicando que este está vivo. Se o GS passar um tempo de espera máximo, configurado no SGA, sem receber nenhuma informação de um determinado GM, então o GS insere o GM em questão na sua lista de suspeitos de falha. Em seguida, o GS faz uma consulta remota na máquina do GM suspeito de falha para verificar o seu estado (detalhes da implementação são dados na Seção 4.3.2). Entretanto, esta consulta remota também estará sujeita a problemas de conexão de rede, lentidão e indisponibilidade do recurso, sendo necessário usar um tempo de espera máximo para que o GS não fique esperando indefinidamente uma resposta. Caso o GS não receba nenhuma resposta dentro do intervalo de tempo determinado no SGA ou a resposta indique a morte do processo GM, então o GM é considerado de ter falhado. Se em um tempo mais tarde, o GS receber uma mensagem de *heartbeat* vinda do GM considerado falho, então o GM em questão é retirado da lista de suspeitos e pode ser novamente usado

para receber tarefas da aplicação do usuário. Esta situação indica que o tempo de espera máximo configurado no GS para o GM específico foi menor do que o necessário. Logo, o tempo de espera pode ser acrescido para aquele GM na tentativa de diminuir o erro no futuro e, consequentemente, garantir que existirão processos corretos por tempo suficiente para a aplicação paralela terminar. O detector de falhas pode repetidamente adicionar ou remover processos corretos da sua lista de suspeitos, o que mostra a dificuldade de determinar se um processo está somente lento ou se teve uma falha. O número de erros do detector de falhas é limitado pelo tempo de execução da aplicação paralela e, com o acréscimo do tempo de espera, os processos corretos podem ser mantidos fora da lista de suspeitos de falha por tempo suficiente para a conclusão da aplicação. Entretanto, é necessário verificar se os custos envolvidos no SGA na adição e remoção dos GM's da lista de suspeitos de falha não é mais caro do que simplismente supor que o detector de falhas esteja correto. A opção escolhida, que depende da implementação, será mostrada no estudo de caso na Seção 4.3.2.

Quando o GS detecta que um GM falhou, a localização da falha é automática, já que o GS conhece o GM que falhou. A contenção da falha também é realizada pelo mesmo GS que identificou a falha. O GS isola o GM suspeito do resto da aplicação por não enviando mais nenhuma mensagem para ele e somente usando alguma possível mensagem recebida para identificar que o processo ainda está vivo.

O estágio de recuperação da falha é realizado de duas formas diferentes dependendo da fonte da falha. Para o caso da falha irrecuperável, onde o detector de falhas não obteve nenhuma resposta sobre o estado do GM, a máquina é considerada indisponível para ser utilizada pelo SGA e o subsistema de tolerância a falhas instrui o subsistema de escalonamento dinâmico para reescalonar todas as tarefas da aplicação que foram alocadas e ainda não finalizadas com sucesso naquele GM para os outros GM's do *site*. Normalmente, o escalonamento dinâmico é realizado seguindo algumas etapas bem definidas (NASCIMENTO, 2008). Primeiro, o GS detecta um desbalanceamento de carga dentro do *site*. O GS identifica o GM mais sobrecarregado e os GM's que precisam aumentar sua carga de trabalho, através do cálculo da carga média do *site*, que é calculada pelo próprio GS com base no desempenho das máquinas dos GM's e a quantidade de trabalho alocada para eles. Detectado o desbalanceamento de carga, o GS requisita um percentual de tarefas para o GM mais sobrecarregado. Ao receber o pedido de tarefas, o GM mais sobrecarregado seleciona as tarefas que serão cedidas e as envia para o GS. Quando o GS recebe as tarefas cedidas, ele as distribui entre os GM's com carga abaixo da carga média dentro do *site*, o que conclui o escalonamento dinâmico. Já no caso da falha irrecuperável, a camada

de tolerância a falhas muda o comportamento da camada de escalonamento dinâmico, fornecendo as tarefas e forçando o reescalonamento destas tarefas entre os GM's do *site*. No caso do detector de falhas cometer um erro e em um tempo mais tarde o GS receber uma mensagem do GM suspeito de falha, o GS envia uma mensagem para o GM solicitando a liberação de todas as tarefas atualmente mantidas por ele, já que estas tarefas já foram reescalonadas para os outros GM's. Depois do GM suspeito de falha enviar uma confirmação da liberação das tarefas, o subsistema de tolerância a falhas inclui o GM na lista de recursos utilizáveis no escalonamento dinâmico.

Para o caso da falha recuperável, onde o detector de falhas obtém uma resposta indicando a morte do processo GM, o GS recriará o GM e fará a restauração do seu estado anterior a falha. O GS mantém um *log* de mensagens para todas as tarefas da aplicação atualmente escalonadas e ainda não finalizadas dentro do seu *site*. Estas mensagens são armazenadas até o momento em que as tarefas terminam a sua execução com sucesso. Quando recriado, o GM recebe uma lista de tarefas da aplicação e as mensagens que elas receberam anteriormente. Observa-se que o GM recomeça a sua execução a partir das informações guardadas no GS e não desde o início como acontece com as tarefas da aplicação que estavam em execução no momento da falha.

Como mencionado na Seção 4.1, uma falha pode ocorrer a qualquer instante. Desta forma, é possível que uma falha ocorra no momento em que um GM está envolvido em um escalonamento dinâmico. Um GM pode está envolvido em dois tipos de escalonamento dinâmico: **local**, que é coordenado dentro do *site* pelo GS de nível acima; e **global**, que é determinado pelo GG e busca o balanceamento de carga entre os *sites* da grade (NASCIMENTO, 2008).

O escalonamento dinâmico local é realizado toda vez que o GS detecta a existência de um desbalanceamento de carga entre os GM's do seu *site*. Quando isto ocorre, o GS solicita ao GM mais sobrecarregado um percentual da sua carga e redistribui esta carga para os GM's com carga abaixo da média. Assim, existem duas situações de falhas que devem ser tratadas durante o escalonamento dinâmico: falha do GM mais sobrecarregado antes de ceder suas tarefas; e falha de um GM, com carga menor do que a média, antes de receber as tarefas cedidas pelo GM mais sobrecarregado.

No caso de falha do GM mais sobrecarregado, o escalonamento dinâmico é sempre abortado pelo subsistema de tolerância a falhas, visto que o GM não tem condições de selecionar as tarefas a serem cedidas e responder a solicitação do GS neste momento. Se a falha do GM for recuperável, o GM é recriado e as tarefas são reenviadas para ele,

sendo um escalonamento dinâmico realizado posteriormente para equilibrar a carga de trabalho entre os GM's do *site*. Já se a falha for irrecuperável, o mecanismo de tolerância a falhas instruirá o subsistema de escalonamento dinâmico para reescalonar todas as tarefas alocadas no GM mais sobrecarregado para os GM's restantes.

No caso de falha de um GM subcarregado, com carga menor do que a média, dependendo do momento que a falha for detectada e se a falha for recuperável ou irrecuperável, o escalonamento dinâmico pode ser executado com sucesso ou abortado. Durante o escalonamento dinâmico, o GS pode detectar a falha de um GM subcarregado antes de receber as tarefas cedidas pelo GM mais sobrecarregado ou no momento que receber as tarefas e for enviar estas tarefas para os GM's subcarregados. Se o GS detecta uma falha recuperável de um dos GM's subcarregados antes de receber as tarefas, o GM em questão é recriado e o procedimento do escalonamento dinâmico não é alterado. Para o caso de uma falha irrecuperável, o escalonamento dinâmico é abortado, visto que o GM ficará indisponível e suas tarefas serão redistribuídas entre os outros GM's do *site*, o que altera a carga média dos GM's restantes. Quando o GS receber a mensagem contendo as tarefas do GM mais sobrecarregado, este verificará que o escalonamento dinâmico foi abortado e retornará as tarefas para o próprio GM. Observe que o GM mais sobrecarregado cedeu as tarefas, mas para o GS as tarefas continuam sendo deste GM, visto que o escalonamento dinâmico foi abortado. Assim, nesta situação, no caso do GM mais sobrecarregado falhar depois de ter cedido as tarefas, o GS considerará que as tarefas ainda são dele no procedimento de recuperação. Outra situação a ser tratada pela camada de tolerância a falhas é o caso do GS detectar a falha de um ou mais GM's subcarregados no momento de enviar as tarefas cedidas. Nesse caso, o GS adiciona as novas tarefas para o GM que falhou e continua normalmente o procedimento de escalonamento dinâmico. Posteriormente, o estágio de recuperação será realizado e o tratamento da falha será feito considerando as novas tarefas adicionadas como pertencentes ao GM que falhou.

O escalonamento dinâmico global é realizado toda vez que o GG detecta a existência de um desbalanceamento de carga entre os *sites* da grade (NASCIMENTO, 2008). Os passos do escalonamento dinâmico global são parecidos ao do escalonamento local. A principal diferença está no fato de que o GG pede um percentual de tarefas para o GS mais sobrecarregado e este repassa o pedido para todos os seus GM's. Depois que todos os GM's respondem ao pedido, as tarefas cedidas pelos GM's são enviadas para o GG. Quando o GG recebe as tarefas cedidas, ele as distribui entre os GS's subcarregados. Os GS's ao receberem as tarefas executam o escalonamento dinâmico destas tarefas de acordo com a carga média de cada GM disponível. Assim, do ponto de vista do GM, existem

duas situações de falhas que devem ser tratadas durante o escalonamento global: falha de um dos GM's do *site* antes de ceder suas tarefas; e falha de um GM no momento de receber as tarefas cedidas pelo *site* mais sobrecarregado.

Diferente do escalonamento dinâmico local, a falha de um GM que cede tarefas não provoca a interrupção do escalonamento global, visto que os outros GM's do *site* continuam podendo responder a solicitação do GS. Vale ressaltar que um GM pode responder positivamente ao pedido de tarefas de um GS ou, caso não possua a carga requisitada disponível para ser reescalonada, negar o pedido. Quando a falha de um GM é detectada, o GS contabiliza como se a resposta do GM que falhou fosse de negação. O escalonamento dinâmico global somente será abortado se todos os GM's negarem a solicitação do GS.

No escalonamento dinâmico global, o GS somente realiza o escalonamento das tarefas entre os GM's disponíveis dentro do seu *site* no momento do recebimento das tarefas vindas do GG. Assim, a situação a ser tratada é a detecção de falha de um GM no momento do envio das tarefas pelo GS. Neste caso, o GS adiciona as tarefas para o GM que falhou e continua normalmente o envio das tarefas para os outros GM's. Desta forma, o escalonamento dinâmico global é concluído e, posteriormente, o tratamento da falha será feito considerando as tarefas adicionadas como pertencentes ao GM que falhou.

4.2.3 Tolerância a Falhas para Gerenciadores do *Site*

O Gerenciador Global (GG) é o responsável pela tolerância a falhas nos GS's (SILVA; REBELLO, 2007) e pelo escalonamento dinâmico de tarefas entre os *sites* (NASCIMENTO et al., 2007; NASCIMENTO, 2008). Semelhante ao GS, o subsistema de tolerância a falhas do GG também interage com as camadas de monitoramento e de escalonamento dinâmico para fornecer a autorrecuperação ao ambiente de execução no nível do *site*.

O esquema de detecção de falhas no GG é idêntico para aquele usado pelos GS's. Todos os GS's sempre enviam mensagens para o GG, seja para informar quando um processo da aplicação terminou sua execução dentro do *site*, mensagens de *heartbeat* ou mensagens da aplicação redirecionadas, como mostrado na Figura 4.3. O Gerenciador Global utiliza as mensagens enviadas pela camada de monitoramento e pela aplicação para saber se um GS está vivo. Assim, a estratégia de *heartbeat* (DÉFAGO; HAYASHIBARA; KATAYAMA, 2003) também é usada pelo GG para detectar as falhas dos GS's. Como o GG e os GS's normalmente serão alocados em processadores diferentes, o GG também usa detectores de falhas não-confiáveis. Logo, o detector de falhas no GG pode cometer

erros e repetidamente adicionar ou remover processos corretos da sua lista de suspeitos. O GG pode detectar falhas irrecuperáveis, quando não obtém nenhuma informação do GS, e falhas recuperáveis, quando obtém uma resposta indicando a morte do processo GS.

Figura 4.3: Comunicação de um processo GS com o GG.

A localização da falha de um GS é simultânea à detecção da falha, já que o GG conhece o GS que falhou. O GG realiza a contenção da falha isolando o GS suspeito do resto da aplicação. O GG pára de enviar mensagens para o GS e somente usa alguma possível mensagem recebida para identificar que o processo ainda está vivo.

Quando o GG detecta uma falha irrecuperável na máquina do GS, o SGA tentará encontrar um recurso alternativo entre as máquinas dos GM's para ser a nova máquina do GS. Esta característica visa continuar utilizando o poder computacional ainda disponível dentro de um *site*, apesar do recurso onde foi escalonado o GS estar indisponível. Determinada a nova máquina do GS, o GG criará o GS substituto e fará a restauração do seu estado anterior a falha. Para isso, o GG mantém um *log* de mensagens de todas as tarefas da aplicação atualmente escalonadas e ainda não finalizadas. Se todas as máquinas do *site* estiverem indisponíveis, o *site* é considerado inacessível e todas as tarefas da aplicação escalonadas e ainda não executadas naquele *site* serão redistribuídas entre os *sites* permanecentes através da camada de escalonamento dinâmico. A diferença entre a recuperação da falha irrecuperável e a recuperável é que no último caso o processo GS sempre será recriado no mesmo recurso. No caso de erro do detector de falhas, quando o GG depois de um tempo recebe alguma mensagem do GS suspeito de falha, o GG envia uma mensagem para o GS solicitando a interrupção da sua execução, já que todas as tarefas atualmente mantidas pelo GS suspeito já foram reescalonadas para os outros GS's.

Ao receber a mensagem do GG, o GS suspeito interrompe os processos da aplicação em execução e aborta a sua própria execução para liberar os recursos utilizados na máquina.

Quando um GS morre, os GM's criados por este GS se tornam órfãos, já que o GS era o único meio de comunicação desses processos com os demais. Por esta razão, quando o GM detecta que seu GS pai morreu, este termina a execução de seus processos da aplicação e aborta a sua execução. Este comportamento de remoção de processos é similar ao que é visto nas células humanas, onde também ocorrem mortes celulares programadas, que são conhecidas como apoptose (KERR; WYLLIE; CURRIE, 1972). A apoptose é um mecanismo de remoção controlada de células, que ajuda na regulação da população de células animais. A morte programada dos processos evita a reserva desnecessária de recursos pelos processos órfãos. Consequentemente, quando um novo GS é criado pelos mecanismos de tolerância a falhas, este deve recriar todos os GM's do *site*.

Assim como ocorre no GM, um GS pode falhar durante um escalonamento dinâmico **local** e **global**. No escalonamento dinâmico local, o GS é o responsável por coordenar o escalonamento e o GG não tem conhecimento de quando este escalonamento está sendo executado dentro de um *site*. Desta forma, no caso de uma falha do GS, o escalonamento é abortado, já que o GG não tem ciência do escalonamento local e mesmo porque todos os GM's daquele *site* serão recriados.

Já no caso do escalonamento dinâmico global, do ponto de vista do GS, existem duas situações de falhas que devem ser tratadas: falha do GS mais sobrecarregado antes de ceder suas tarefas; e falha de um GS subcarregado antes de receber as tarefas cedidas pelo GS mais sobrecarregado. O tratamento destas duas situações de falhas é muito similar ao realizado pelo mecanismo de tolerância a falhas no caso do GM no escalonamento local.

No caso de falha do GS mais sobrecarregado, o escalonamento dinâmico é sempre abortado pelo subsistema de tolerância a falhas, tal como acontece com o GM no escalonamento local. A diferença está no tratamento da falha irrecuperável do recurso onde um GS está alocado, neste caso o mecanismo de tolerância a falhas tentará encontrar um novo recurso para alocar o GS, antes de instruir o subsistema de escalonamento dinâmico para reescalonar as tarefas alocadas no GS mais sobrecarregado para os GS's restantes.

No caso de falha de um GS subcarregado, dependendo do momento que a falha for detectada e se a falha for recuperável ou irrecuperável, o escalonamento dinâmico pode ser executado com sucesso ou abortado. As situações de falhas são as mesmas do escalonamento local com falha no GM, o GG pode detectar a falha de um GS subcarregado antes de receber as tarefas cedidas pelo GS mais sobrecarregado ou no momento que receber

as tarefas e for enviar estas tarefas para os GS's subcarregados. A diferença no comportamento do escalonamento dinâmico local e global está na forma de tratamento da falha irrecuperável do recurso onde o GS está alocado. Se o GG detecta uma falha irrecuperável de um dos GS's antes de receber as tarefas, o GG tentará encontrar um novo recurso para alocar o GS. Conseguindo este recurso, o GS e os GM's do *site* serão recriados e o escalonamento dinâmico prossegue normalmente. O escalonamento global somente é abortado, no caso do GG não conseguir nenhum recurso disponível dentro do *site* para alocar o GS. No caso do GG detectar a falha de um ou mais GS's subcarregados no momento de enviar as tarefas cedidas, o GG adiciona as novas tarefas para o GS que falhou e continua o escalonamento global normalmente, tal como acontece no escalonamento dinâmico local com o GM.

4.2.4 Tolerância a Falhas para o Gerenciador Global

A existência de pelo menos um recurso persistente é um requisito comum entre as estratégias de tolerância a falhas. Entretanto, neste trabalho será usada a abordagem do portal da grade ou do escalonador de *jobs* para oferecer tolerância a falhas no GG (BOERES et al., 2006). O mecanismo para o tratamento de falhas no portal está fora do escopo do trabalho, mas uma solução para este problema é a criação de várias instâncias do portal baseado em técnicas de alta disponibilidade (MARCUS; STERN, 2000).

A responsabilidade para a detecção de falhas no GG é do portal da grade que dispara o processo GG, sendo a partir deste processo que toda a estrutura do SGA EasyGrid é criada. Quando o processo GG tem uma falha por colapso, o portal detecta o erro pela interrupção da execução do processo GG ou pelas informações de monitoramento que o portal faz nas máquinas da grade. O mecanismo de monitoramento do portal também não está incluído no escopo deste trabalho.

Após a detecção de que o GG falhou, a localização e a contenção da falha são automáticas. Isso se deve ao fato do portal saber qual aplicação apresentou a falha e esta falha não se propaga para outras aplicações, já que o SGA é por aplicação e as submissões das aplicações paralelas no portal são independentes uma das outras.

A recuperação do GG é realizada pela utilização da técnica de *checkpointing* no nível da aplicação, pois o GG é um processo de longa duração e não deve ser reexecutado desde o início. Os dados gravados no *checkpoint* são obtidos no nível da aplicação, pois as informações necessárias para a restauração do estado do GG são conhecidas previamente

no SGA EasyGrid. Existem duas possibilidades para a recuperação do GG: ser executada pela própria aplicação ou pelo portal EasyGrid. A recuperação sendo realizada pela aplicação seguiria o conceito de aplicação autônoma, onde a própria aplicação se recuperaria. Entretanto, existe o problema de qual processo reiniciaria o GG depois da falha. No caso de um processo GG *backup*, que recriasse o GG ou assumisse a função do GG, o sistema continuaria vulnerável a uma falha deste processo *backup*.

Diferente de outras abordagens de *checkpoints* para aplicações paralelas, a abordagem utilizada não requer que o mesmo número de recursos seja disponibilizado para restaurar a aplicação. Entretanto, quando o GG morre, a estrutura de processos gerenciadores precisa ser recriada, visto que os GS's criados pelo GG se tornam órfãos e o GG era o único meio de comunicação dos *sites* com os demais recursos da grade computacional. Assim, esse momento é oportuno para substituir os recursos indisponíveis e, por consequência, melhorar o desempenho da aplicação paralela restaurando o poder computacional perdido provocado por falhas nos recursos. Para isso, é necessária uma interação com o portal Easygrid, que gerencia os recursos da grade, para verificar o estado das máquinas e readquirir os recursos. Por estas razões, o mecanismo de tolerância a falhas foi desenvolvido esperando que o portal EasyGrid realize a recriação do GG, como se fosse a resubmissão do *job*.

A criação e a recuperação do processo GG possui as etapas descritas a seguir. Inicialmente, o portal EasyGrid cria o processo GG. Durante a sua execução, este processo cria de tempos em tempos um arquivo com o seu estado atual. O arquivo gerado é armazenado na máquina do portal EasyGrid, já que o portal terá a função de recriar o processo GG no caso de falha. O estado atual do GG contém o *log* de mensagens de todos os processos da aplicação atualmente não finalizados, ou seja, os processos que ainda não foram executados ou estão em execução. Com base na hierarquia de gerenciadores, o GG é o único processo que possui todas estas informações. O intervalo de tempo entre as realizações dos *checkpoints* no GG pode ser determinado pelo próprio SGA EasyGrid, no caso dos pesos das tarefas serem conhecidos a priori, ou pelo usuário. Esta opção visa oferecer um melhor ajuste do mecanismo ao tempo de execução da aplicação. Após a detecção da falha, o portal recria o GG e este busca o arquivo de *checkpoint* gerado anteriormente. Quando o GG inicia sua execução os dados gravados são usados na restauração do processo com o último estado armazenado antes da falha. Caso não exista o arquivo de *checkpoint*, o GG é reiniciado como se fosse a primeira execução.

4.3 Estudo de Caso: Tolerância a Falhas em Aplicações MPI

Esta seção descreve os detalhes da implementação dos mecanismos de tolerância a falhas no SGA EasyGrid em aplicações MPI desenvolvidas na versão LAM/MPI. Como já dito anteriormente, os mecanismos de tolerância a falhas desenvolvidos no SGA EasyGrid seguem 4 estágios, que são eles: detecção da falha/erro, localização da falha, contenção da falha e recuperação da falha/erro.

A biblioteca MPI oferece **detecção** da falha/erro através de *error handlers*, que são associados aos comunicadores. Por *default*, o *error handler* associado aos comunicadores é `MPI_ERRORS_ARE_FATAL`, ou seja, se uma função MPI retorna um erro, então todos os processos neste comunicador serão abortados. Para evitar que isso aconteça, o *error handler* `MPI_ERRORS_RETURN` pode ser associado a cada comunicador, de modo que o código de erro é retornado e, por consequência, erros possam ser identificados. O SGA EasyGrid usa intercomunicadores diferentes entre cada par de processos gerenciadores e da aplicação de modo que as falhas detectadas possam ser facilmente **localizadas** e **isoladas**.

O subsistema de tolerância a falhas do SGA, utilizando o *error handler* da biblioteca MPI, detecta falhas através de erros de comunicação. Estes erros ocorrem quando mensagens são enviadas ou recebidas entre processos da aplicação e GM e entre os processos gerenciadores. Entretanto, a detecção de falhas em processos MPI se torna difícil, especialmente para funções MPI não-bloqueantes que são operações locais e nem sempre identificam que um processo remoto está morto, mesmo com o *erro handler* configurado para `MPI_ERRORS_RETURN`. Mas, visto que a ordem, o tamanho e o tempo de chegada das mensagens gerenciadas pelo SGA são desconhecidos, os processos gerenciadores usam tais funções para minimizar a intrusão. Esse mesmo problema foi identificado na função `MPI_Send` usando os protocolos de comunicação *short* e *eager*, já que neste caso a função é não-bloqueante. Caso a mensagem já esteja disponível no *buffer* do sistema quando a função `MPI_Recv` é invocada, a morte de um processo remoto também nem sempre é detectada.

Outro problema observado através de experimentos é que o envio de duas mensagens consecutivas usando o protocolo *eager* (isto é, mensagens com tamanho de 52105 à 65536 bytes) para um processo morto pode causar a morte do processo origem. Por causa desse problema, mecanismos de tolerância a falhas foram desenvolvidos neste trabalho para

verificar o estado dos processos remotos antes de cada comunicação, como será visto em mais detalhes nas próximas subseções.

4.3.1 Tolerância a Falhas para Processos da Aplicação no MPI

Como dito na Seção 4.2.1, o GM é o responsável pela detecção de falhas dos processos da aplicação. Para isso, utiliza as mensagens recebidas pelos próprios processos da aplicação ou da camada de monitoramento para saber se os processos estão vivos.

A implementação do mecanismo de tolerância a falhas consiste em, caso o tempo decorrido desde a última mensagem recebida de um processo v ultrapasse um limite pré-determinado (configurado no SGA com o peso da tarefa informado no GAD), enviar um sinal[1] somente para testar o comunicador do processo da aplicação utilizando a função `MPIL_Signal`. Se a função retorna um erro, o processo gerenciador detecta que uma falha ocorreu.

Quando uma falha é detectada, o subsistema de tolerância a falhas libera o comunicador entre o gerenciador e a aplicação para prevenir uma possível propagação de falhas, o que é feito pela função `MPI_Comm_free`. Além disso, instrui o subsistema de gerenciamento de processos para criar um novo processo $v\prime$ para substituir v. Para a criação dinâmica de processos no MPI, o SGA usa a função `MPI_Comm_spawn`. Todas as mensagens para $v\prime$ serão reenviadas do *log* de mensagens de v mantido localmente pelo GM. Já no caso das mensagens enviadas por $v\prime$, o GM só redireciona as mensagens, quando o número de mensagens for maior do que o número de mensagens enviados por v. O controle do número de mensagens é feito pelo GM e é realizado como explicado a seguir. Para cada processo da aplicação, o GM possui dois contadores: $nMsgEnv(v)$ e $nMsgEnvF(v)$. O $nMsgEnv(v)$ representa o maior número de mensagens já enviadas pelo processo v, que é inicializado com zero somente na primeira execução. O $nMsgEnvF(v)$ representa o número de mensagens enviadas pela instância atual do processo v naquela execução e é zerado toda vez que o processo é reinicializado devido a ocorrência de uma falha. Toda vez que um processo da aplicação envia uma mensagem, esta é contabilizada em $nMsgEnvF(v)$ e comparada com o valor de $nMsgEnv(v)$. Se $nMsgEnvF(v) > nMsgEnv(v)$, então $nMsgEnv(v)$ é atualizada e a mensagem é redirecionada pelo GM. Observa-se que na primeira execução, os dois contadores são inicializados com zero e, a cada mensagem enviada, o valor de $nMsgEnvF(v)$ é incrementado e se torna maior do que o valor de

[1] Um sinal é uma forma de comunicação entre processos. Normalmente, é uma notificação assíncrona enviada para um processo a fim de notificá-lo de um evento que ocorreu.

$nMsgEnv(v)$. Assim, caso não ocorra falhas, as mensagens de um processo sempre são redirecionadas pelo GM. Caso ocorra uma falha, o processo v é reiniciado e o contador $nMsgEnvF(v)$ é zerado. Enquanto $nMsgEnvF(v) \leqslant nMsgEnv(v)$, as mensagens enviadas pelo processo reiniciado serão descartadas. Desta forma, as mensagens duplicadas para tarefas determinísticas são impedidas de propagar pelo GM. O *log* de mensagens e os contadores $nMsgEnv(v)$ e $nMsgEnvF(v)$ são liberados da memória para cada processo da aplicação que termina normalmente a sua execução.

4.3.2 Tolerância a Falhas para Gerenciadores da Máquina no MPI

Como dito na Seção 4.2.2, o GS é o responsável por oferecer tolerância a falhas para os recursos dentro do seu *site*, o que inclui os GM's criados no *site* (SILVA; REBELLO, 2007). O GS utiliza a estratégia de *heartbeat* para detectar as falhas dos GM's. Entretanto, para sistemas assíncronos, nenhuma suposição de tempo de chegada das mensagens de *heartbeat* pode ser feita. Assim, o GS usa um detector de falhas não-confiável, proposto por Chandra e Toueg (CHANDRA; TOUEG, 1991, 1996). Segundo (CHANDRA; TOUEG, 1996), sistemas práticos podem usar tempos de espera (*time-outs*) para implementar detectores da classe $\diamond W$, denominados de **Detectores de Falhas Eventualmente Fracos** (*Eventually Weak Failure Detectors*). Visto que existirão processos corretos fora da lista de suspeitos de falha por tempo suficiente para a aplicação progredir e terminar.

A implementação do mecanismo de detecção de falhas é realizada da forma a seguir. Quando o GS passa um tempo de espera máximo, configurado no SGA como duas vezes o período da mensagem de *Heartbeat*, sem receber nenhuma informação de um determinado GM, então o GS insere o GM em questão na sua lista de suspeitos de falha. Depois disso, o GS realiza uma consulta remota na máquina do GM suspeito para verificar se a falha é irrecuperável ou recuperável. Entretanto, a utilização da mesma técnica de envio de sinal da Seção 4.3.1 não pode mais ser usada na verificação da falha do GM. Este problema ocorre porque a função `MPIL_Signal` fica bloqueada quando o *daemon* remoto está inacessível. Com isso, um GS poderia ficar travado e não mais continuar suas funções por causa de um GM inatingível (*Host Unreachable*). Observe que no caso da Seção 4.3.1, o GM e as tarefas da aplicação sempre estão em uma mesma máquina. Já no caso entre um GS e um GM, os processos gerenciadores normalmente estarão em máquinas diferentes.

Uma tentativa para solucionar este problema foi criar uma *thread* somente para executar a função `MPIL_Signal`. Assim, a *thread* poderia ser cancelada caso ficasse travada

e o GS continuaria realizando as suas funções normalmente. Entretanto, os testes realizados com o SGA mostraram que a utilização de mais de uma *thread* ao mesmo tempo provoca o GS perder conectividade com outros *daemons*. O provável motivo para este comportamento é que o LAM/MPI ainda não tem suporte para múltiplas *threads* executando funções MPI, ou seja, somente uma operação MPI pode ser executada por vez de forma segura. Ao utilizar *threads*, a função `MPIL_Signal` era executada mais de uma vez ao mesmo tempo, já que mais de um processo GM poderia estar como suspeito de falha.

A solução adotada usa *threads* para executar uma chamada de sistema com o comando `mpitask` para testar os GM's. O comando `mpitask` pode ser usado para monitorar todos ou um processo MPI específico no ambiente LAM. As informações fornecidas para se identificar um processo específico é a identificação do nó e o *pid* do processo, que podem ser obtidos através da função `MPIL_Comm_gps`. Embora o comando `mpitask` possa ficar bloqueado no caso de uma falha irrecuperável, a *thread* pode ser cancelada e as funções MPI continuarão funcionando normalmente. Entretanto, foi observado experimentalmente que depois de serem identificados um determinado número de falhas irrecuperáveis, o *daemon* do LAM executando no recurso do GS perde sua conectividade com os outros *daemons*. Isso significa que depois de descobrir algumas falhas irrecuperáveis nos gerenciadores de nível inferior, o próprio gerenciador do *site* apresentaria uma falha irrecuperável. Para o processo GG, este *daemon* pareceria estar morto. Como uma tentativa de evitar esta situação, o comando `lamshrink` foi usado para remover os nós com falhas irrecuperáveis do universo LAM. Além disso, o comando `mpitask` bloqueado é morto pelo GS que o criou. Assim, não foi mais observada esta limitação de somente poder descobrir algumas falhas irrecuperáveis. Contudo, o GS é obrigado a excluir um recurso do universo LAM, quando este não recebe nenhuma informação de um determinado GM. Isto faz com que um GM erradamente suspeito de falha e que não tenha passado no teste do `mpitask`, não possa retornar a sua execução em um tempo mais tarde. Caso todos os GM's de um *site* tenham falhas irrecuperáveis, o GS em questão aborta a sua execução por falta de recursos computacionais.

Como mencionado no início desta seção de estudo de caso, um processo pode morrer se este envia duas mensagens para um processo morto. Consequentemente, o SGA foi obrigado a verificar o estado de cada processo destino antes de realizar uma comunicação. Entretanto, para ocultar esta sobrecarga, o estado de um processo é somente requisitado depois de cada comunicação e verificado o resultado somente antes da próxima comunicação para aquele processo. Embora uma falha possa ocorrer depois do teste e antes da comunicação, a falha será detectada antes da segunda mensagem ser enviada.

O mecanismo para detectar as falhas consiste de criar uma *thread* para testar se o *daemon* e o GM na máquina remota estão vivos. Uma variável compartilhada entre esta *thread* e a *thread* principal (o GS) recebe o resultado da chamada de sistema que usa o comando `mpitask`. Se este resultado não está disponível quando o GS precisa enviar outra mensagem, o GS continuará testando a variável numa série de tempos de espera que serão aumentados exponencialmente até um limite máximo configurável. Entretanto, foi identificado experimentalmente que os tempos de espera acrescidos exponencialmente prejudicam o desempenho do gerenciador e foi realizada uma mudança em que o gerenciador verifica a resposta da *thread* em pequenos intervalos de tempo até o limite máximo configurável. Se a *thread* não está bloqueada, isto é, o comando `mpitask` terminou, então o *daemon* remoto está vivo. A variável compartilhada identificará se uma falha tem ocorrido no GM. Depois do limite máximo de espera, se a thread continua bloqueada, então o GS assume que o *daemon* na máquina remota está morto e o GS executa uma chamada de sistema com o comando `lamshrink`. O tempo de espera máximo pode ser configurado levando em consideração que o GS e os GM's estão em uma rede local. Em resumo, a Figura 4.4 mostra as possíveis respostas da *thread* executando o comando `mpitask` e o seu significado para o mecanismo de tolerância a falhas.

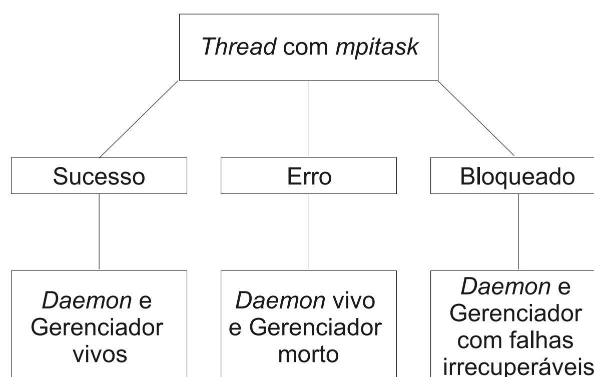


Figura 4.4: Respostas da *thread* com o comando *mpitask*.

Existem duas formas diferentes para a recuperação das falhas envolvendo os GM's. No caso de falha irrecuperável, onde a máquina não é mais utilizável, os processos da aplicação que foram alocados para aquele recurso serão redistribuídos entre os demais GM's do *site*. Esta operação é realizada pelo escalonador dinâmico, que recebe a identificação das tarefas do mecanismo de tolerância a falhas. No caso de falha recuperável, onde o GM morre e o *daemon* do LAM permanece vivo, o GS recriará o GM através da função `MPI_Comm_spawn` e enviará a lista de processos da aplicação e as mensagens que eles receberam antes da falha. Observa-se que o GM recomeça a sua execução a partir

das informações guardadas no GS. Caso algum processo da aplicação tenha terminado sua execução, mas esta informação não foi recebida pelo GS antes da falha, então este processo será reexecutado.

4.3.3 Tolerância a Falhas para Gerenciadores do *Site* no MPI

Como dito na Seção 4.2.3, o GG é o responsável por oferecer tolerância a falhas entre os *sites* (SILVA; REBELLO, 2007). O esquema de detecção de falhas no GG é idêntico para aquele usado pelos GS's, o que inclui a estratégia de *heartbeat* para detectar as falhas dos GS's. As falhas podem ser do tipo irrecuperável ou recuperável.

A recuperação da falha realizada pelo GG também é muito similar a usada pelo GS's, o que também inclui a utilização de *threads* para executar o comando `mpitask` com o objetivo de testar os GS's. A diferença é que caso o recurso de um GS sofra uma falha irrecuperável, o SGA tentará encontrar um recurso alternativo entre as máquinas dos GM's para ser a nova máquina do GS. O critério utilizado para a escolha do novo GS é seguir a ordem na qual as máquinas de um *site* foram usadas para iniciar o ambiente de execução LAM. O mecanismo de tolerância a falhas está preparado para receber esta informação e não remover o gerenciador daquele *site* da estrutura de gerenciadores. Já no caso de falha recuperável, o GS será recriado no mesmo recurso. Um GS recriado inicia a sua execução recriando os GM's do seu *site*.

4.3.4 Tolerância a Falhas para o Gerenciador Global no MPI

Neste trabalho será usada a abordagem do portal da grade ou do escalonador de *jobs* para oferecer tolerância a falhas no GG. O responsável pela submissão do *job* é que terá a função da detecção de falhas no GG. O processo GG é o único processo que é criado estaticamente no início da execução do SGA. Desta forma, o portal detectará a falha de colapso através do retorno de erro do comando `mpirun` ou através do mecanismo de monitoramento implementado no portal EasyGrid (BOERES et al., 2006).

No estágio de recuperação, o portal recria o GG e este busca o arquivo de *checkpoint* gravado pelo próprio GG antes da falha. Quando o GG inicia sua execução, os dados gravados são usados na restauração do processo com o último estado armazenado antes da falha. Caso não exista o arquivo de *checkpoint*, o GG é reiniciado como se fosse a primeira execução.

Como explicado na Seção 4.2.4, o processo GG de tempos em tempos cria um arquivo binário com o seu estado atual. Neste arquivo é gravado, para cada GS, o número de GM's pertencentes ao *site*, o número de tarefas já executadas por cada GM e o número de tarefas a serem executadas dentro do *site*. Além disso, para cada tarefa a ser executada são gravados o número de mensagens e as próprias mensagens recebidas pela tarefa. Vale ressaltar que no momento da recriação do GG não é necessário existir o mesmo número de GS's disponíveis antes da falha. No caso de um número menor de GS's, o GG detectará a falta do GS como uma falha e redistribuirá as tarefas entre os GS's restantes. O mesmo procedimento é realizado pelo GS no caso de existir um número menor de GM's disponíveis dentro do *site*.

Outro fato importante a ser considerado no mecanismo de recuperação é que uma falha no processo GG pode ocorrer no meio da gravação do arquivo de *checkpoint*. Desta forma, se faz necessário garantir uma memória estável atômica na gravação do estado do GG para que o arquivo gerado não fique incompleto e este não sirva para a sua recuperação. Para evitar a perda do único e último estado consistente gravado, uma cópia do último arquivo de *checkpoint* é mantida toda vez que uma nova gravação de estado ocorre. Para a verificação do término da gravação de um estado do GG, foi adicionado um sinal no final do arquivo para indicar a conclusão da gravação. Dessa forma, no procedimento de recuperação, o GG antes de começar a leitura dos dados do arquivo, verifica a marca inserida no final do arquivo. Caso o arquivo esteja incompleto, a leitura é feita a partir da cópia do arquivo de *checkpoint* com o último estado gravado. Para a cópia do arquivo de *backup* foi utilizada uma *thread*, que inicia a cópia logo em seguida que o arquivo de *checkpoint* teve sua gravação concluída.

4.4 Resumo

Este capítulo apresentou o modelo de falhas utilizado para fornecer tolerância a falhas nas aplicações MPI que incorporam o SGA EasyGrid. Foi descrito o mecanismo de recuperação para processos da aplicação. Uma descrição detalhada dos mecanismos de tolerância a falhas desenvolvidos para os processos gerenciadores GG, GS e GM. Além disso, foram mostradas as dificuldades de implementação dos mecanismos no LAM/MPI. O Capítulo 5 realiza uma avaliação dos mecanismos de tolerância a falhas aqui apresentados.

Capítulo 5

Experimentos Computacionais

Este capítulo descreve os experimentos realizados para examinar a capacidade do SGA EasyGrid para terminar eficientemente a execução de sua aplicação mesmo na presença de falhas de recursos ou processos. O foco é determinar as sobrecargas associadas com a implementação de uma aplicação autônoma baseada na implementação padrão do LAM/MPI habilitado para o Globus Toolkit. Nos experimentos são simuladas falhas irrecuperáveis e recuperáveis para testar o SGA EasyGrid.

Para a análise de desempenho foram utilizadas três aplicações, uma sintética e duas reais, que pertencem a classes comumente executadas em ambientes de grades computacionais. As aplicações são uma sintética e uma real do tipo Mestre-Trabalhador (ou *Bag-of-Tasks*) e uma aplicação iterativa com dependência entre tarefas.

A aplicação Mestre-Trabalhador, denominada **MTrab**, possui um processo mestre e os demais processos são trabalhadores. Cada processo trabalhador recebe e envia uma ou mais mensagens para um processo mestre. A aplicação MTrab é sintética, onde a quantidade de trabalho executado por cada processo trabalhador pode ser controlada a fim de investigar como a granularidade da aplicação afeta o desempenho do SGA EasyGrid com relação a tolerância a falhas (TF). A quantidade mínima de trabalho realizada por cada trabalhador é denominada aqui de uma **unidade de trabalho**.

A aplicação **MTrab** foi implementada em duas versões com base nos modelos de execução 1PProc e 1PTask. Na versão MPI tradicional no modelo 1PProc, um processo MPI por processador é criado e o processo mestre distribui uma **unidade de trabalho** entre os processos trabalhadores sob demanda. Na versão do modelo 1PTask, que será gerenciada pelo SGA EasyGrid, cada processo trabalhador executa somente uma **unidade de trabalho**. O número de processos trabalhadores é predefinido pela carga de trabalho

da aplicação. A comunicação do processo mestre com cada processo escravo é realizada indiretamente via os processos gerenciadores do SGA, o que é feito de forma transparente para a aplicação.

A segunda aplicação paralela do tipo Mestre-Trabalhador não é sintética. Esta aplicação, chamada **Térmions**, é utilizada para calcular a dispersão térmica macroscópica em um meio poroso (SOUTO et al., 2002). Dado um meio poroso composto de elementos sólidos e fluidos, a dispersão térmica é avaliada pela movimentação de um grande número de partículas hipotéticas, denominadas Térmions, a partir de um ponto fixo no meio. A distância percorrida, dentro de um período de tempo, por cada térmion individual é determinada pela sua posição, energia, um componente randômico e pelas propriedades térmicas dos sólidos ou pela velocidade do fluxo do fluido. O custo computacional para calcular o caminho percorrido por cada térmion varia e não pode ser precisamente determinado *a priori*, devido à natureza randômica do caminho dos térmions. Para esta aplicação, a **unidade de trabalho** corresponde a determinação do caminho de um único térmion. Cada tarefa da aplicação executa somente uma unidade de trabalho. Não foi necessária a implementação de uma outra versão para o SGA, pois o programa desenvolvido no modelo 1PProc divide os térmions entre os processos em execução e para os experimentos computacionais realizados foram utilizados o número de processos igual ao número de térmions e a aplicação compilada com a biblioteca EasyGrid.

A terceira aplicação também não é sintética e é do tipo iterativa. A aplicação, chamada de **N-corpos**, é utilizada para resolver o problema astrofísico N-corpos (*N-body*). Este problema simula a evolução de um sistema composto de N corpos ou partículas, onde as forças gravitacionais são exercidas em cada partícula pela interação com as outras partículas do sistema. Para o cálculo das forças exercidas entre todas as partículas será usado o algoritmo paralelo *ring* ou também conhecido de sistólico (DORBAND; HEMSENDORF; MERRITT, 2003), que é um dos melhores algoritmos para a execução em ambientes dedicados e homogêneos, como *clusters* de computadores. Este algoritmo para o modelo de execução 1PProc, chamado de **MPI ring** em (SENA, 2008), as N partículas são divididas entre os P processadores disponíveis. Cada processo calcula as forças que interagem nas suas N/P partículas. Este cálculo é feito de forma iterativa e são necessárias P iterações para a conclusão do algoritmo, como mostrado na Figura 5.1.

Na primeira iteração, cada processo possui a posição das suas partículas, calcula a força gravitacional exercida entre elas e envia a posição das suas partículas para o processo vizinho da direita. Nas próximas $P - 1$ iterações, cada processo recebe a posição das

partículas do vizinho da esquerda, calcula a força gravitacional exercida pelas partículas recebidas sobre as suas partículas e repassa a posição das partículas recebidas para o vizinho da direita. Ao final das P iterações, cada processo recebe a posição de todas as N partículas e para cada partícula é somada a força gravitacional exercida pelas outras $N-1$ partículas.

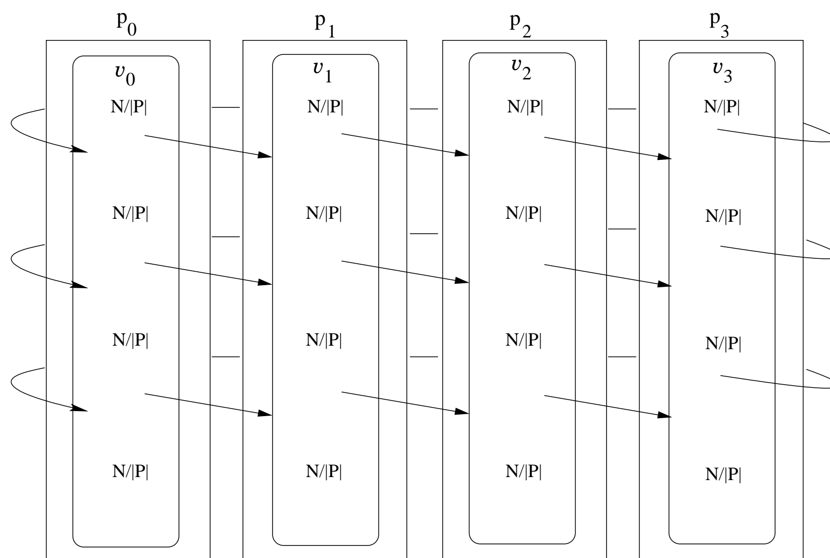


Figura 5.1: Iterações do algoritmo *ring* em 4 processadores (SENA, 2008).

Uma versão do algoritmo *ring* para o modelo de execução 1PTask, chamada de **AMS ring**, foi desenvolvida em (SENA, 2008). Nesta versão, ao invés de se executar um processo por processador, um conjunto de processos menores com relação de precedência é definido, como ilustrado na Figura 5.2. A aplicação *ring* é modelada por um Grafo Acíclico Direcionado (GAD) com largura e altura iguais a W, e então um grau de paralelismo W.

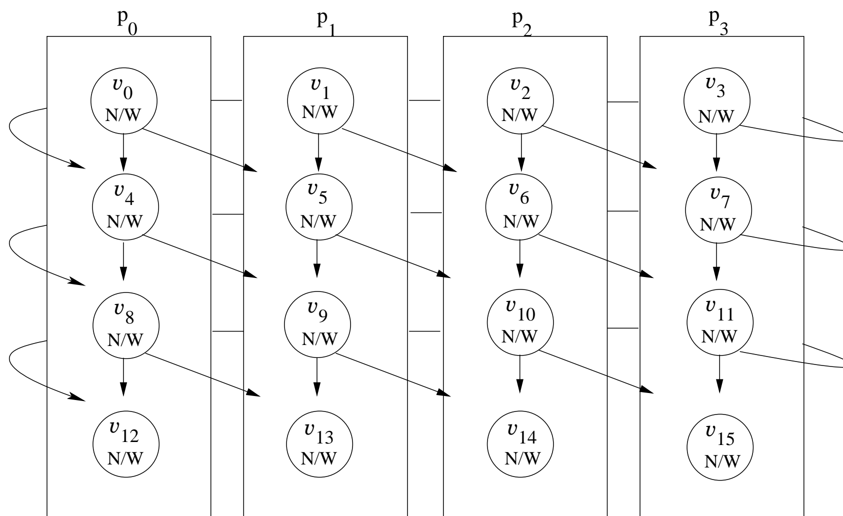


Figura 5.2: Iterações do novo algoritmo *ring* com $W = 4$ e $P = 4$ (SENA, 2008).

Como mostrado na Figura 5.2, se $W = P$, cada processo v_i calcula a força exercida para N/W partículas em uma iteração do algoritmo *ring* original. Após a computação, cada processo v_i envia uma mensagem para cada um dos seus dois sucessores. Para o processo vizinho da direita é enviada a posição das suas partículas, tal como no algoritmo *ring* original. A outra mensagem, adicionada neste algoritmo, é para o vizinho abaixo, que recebe a posição das partículas e a soma das forças exercidas nas partículas até o momento. Assim, a versão do algoritmo *ring* para o modelo de execução 1PTask possui um custo maior de comunicação. A vantagem deste algoritmo está na utilização de processadores heterogêneos, onde o escalonador dinâmico pode escalonar os processos menores para aproveitar melhor o desempenho dos processadores disponíveis.

A seguir serão mostrados os experimentos realizados para demonstrar a habilidade do SGA EasyGrid em completar a execução da aplicação mesmo na presença de falhas de processos e processadores. O SGA é composto pela aplicação do usuário e por três níveis de gerenciadores, consequentemente, todos os níveis precisam de tolerância a falhas. Para melhor avaliar a sobrecarga produzida pelos mecanismos de tolerância a falhas, os experimentos foram divididos em três seções. A Seção 5.1 apresenta os experimentos envolvendo somente as falhas de processos da aplicação, quando o SGA EasyGrid só tinha suporte para este tipo de falha. Na Seção 5.2 serão tratadas as falhas dos processos GM's e GS's. Esta seção foi dividida em duas partes. A primeira parte apresenta a sobrecarga dos mecanimos de tolerância a falhas, quando o SGA EasyGrid só tratava as falhas de processadores e em dois níveis de gerenciadores (GM e GS). A segunda parte possui os mesmos experimentos da primeira parte, mas utiliza a versão com tolerância a falhas em todos os níveis de gerenciadores (GM, GS e GG). Além disso, é feita uma comparação entre os resultados obtidos na primeira e segunda parte. Por fim, a Seção 5.3 avalia o tratamento da falha do processo GG. O objetivo desta divisão é mostrar a sobrecarga dos mecanismos por nível e de acordo com a sua evolução.

5.1 Falhas nos Processos da Aplicação

Em (NASCIMENTO et al., 2005, 2008; SENA et al., 2007) foram apresentados os experimentos realizados para demonstrar a habilidade do SGA EasyGrid em tratar as falhas do tipo *fail-stutter* nos processos MPI da aplicação.

Este primeiro experimento foi executado em um ambiente grade semi-controlado com 4 *sites*. Um ambiente semi-controlado significa que os experimentos foram realizados com

acesso exclusivo para os recursos computacionais, mas estes estavam conectados em uma rede pública, o que pode ter afetado o desempenho das aplicações pelo tráfego de rede externo. Os *sites* são interconectados por um *switch* Fast Ethernet e cada *site* é composto por um *cluster* de idênticos processadores Pentium IV de 2.6 GHz e com 512 Mb de RAM executando Linux Fedora Core 2, Globus Toolkit 2.4 e LAM/MPI 7.0.6. O primeiro *site* contendo 13 máquinas interconectadas em uma rede Gigabit, o segundo com 8 conectadas por um *switch* Fast Ethernet, o terceiro possuindo 8 processadores conectados em outro switch Gigabit e, por fim, o último *site* com 2 máquinas ligadas por um *hub* Ethernet. Os 4 *sites* estão localizados no Instituto de Computação da UFF.

A fim de simplificar este experimento em relação ao escalonamento dinâmico, a configuração utilizada foi de um GG e um GS com 28 GM's em um total de 30 processadores. Observe que tipicamente se configura um GS para cada *site* do ambiente, mas para focar na recuperação dos processos da aplicação foi utilizada esta configuração. Assim, o GG não precisa realizar nenhum escalonamento dinâmico global, pois só existe um GS em seu controle. Entretanto, o GS continua realizando o escalonamento dinâmico dentro do *site*. Além disso, somente os 28 processadores com os GM's são usados para executar as unidades de trabalho.

No experimento foram utilizadas as aplicações MTrab na versão 1PTask e os Térmions, ambos com o SGA embutido. A duração de cada tarefa na aplicação MTrab foi de 5 segundos e na aplicação Térmions varia entre 4 e 5 segundos, sendo na média 4,4 segundos. Em ambos os casos esta medição de tempo foi feita em processadores sem carga.

Para realizar o experimento, os seguintes cenários de falhas (F) foram considerados: (1) nenhuma tarefa falhou durante a execução da aplicação; (2) 10% das tarefas da aplicação falharam; e (3) 20% das tarefas da aplicação falharam. Seja **pf** definido como a percentagem de falha.

A fim de simular a falha de processos e testar a integração das camadas de tolerância a falhas e escalonamento dinâmico, os processos da aplicação foram mortos em só quatro máquinas distribuídas através de três *sites*: 2 máquinas no primeiro site, 1 no segundo e 1 no terceiro. O comando **kill** do Linux/Unix foi usado para abortar um número fixo total de processos da aplicação. Este comando foi executado em períodos de 6,5 segundos para evitar a possibilidade de tentar abortar o mesmo processo, antes mesmo da falha ser detectada. Assim, não existiriam processos executando no processador, visto que esses ainda não teriam sido recriados pelo GM, o que causaria nenhum processo ser morto e interferiria nos resultados dos testes. O período de tempo utilizado no teste é mais do

que suficiente para os mecanismos de tolerância a falhas do SGA EasyGrid detectarem a falha e reiniciarem o processo que falhou, já que o tempo de espera máximo para o GM obter informações do processo da aplicação é de 5 segundos neste experimento.

A Tabela 5.1 mostra o tempo de execução real (R) da aplicação com o SGA EasyGrid tendo **N** unidades de trabalho; um limite inferior de tempo de execução estimado (E(pf)) da aplicação com pf = 0%, 10% e 20% de tarefas que falharam, que são os cenários (1), (2) e (3), respectivamente; e o percentual de sobrecarga (S) do SGA com os mecanismos de tolerância a falhas para ambas as aplicações. Os tempos de execução mostrados na Tabela 5.1 foram obtidos a partir da média de três execuções, onde o coeficiente de variação[1] da média calculada foi menor do que 1,5% para estes experimentos.

Tabela 5.1: Comparação entre os tempos de execução real (R) e o limite inferior estimado (E(pf)), em segundos. A terceira coluna mostra a sobrecarga relativa a execução ideal (S).

F	MTrab						Térmions		
	N = 10.000			N = 1.000			N = 1.000		
	R	E(pf)	S(%)	R	E(pf)	S(%)	R	E(pf)	S(%)
(1)	1808,19	1790,00	1,02	183,17	180,00	1,76	161,93	158,40	2,23
(2)	1929,19	1875,00	2,89	197,21	187,50	5,18	172,53	165,00	4,56
(3)	2043,28	1964,29	4,02	209,20	196,43	6,50	184,96	172,86	7,00

Apesar da regularidade na execução do comando **kill**, é difícil garantir que os processos serão mortos exatamente no mesmo momento. Com o objetivo de estimar o tempo de execução ótimo, é assumido para o cálculo do limite inferior estimado (E(pf)) que, em média, as tarefas foram mortas na metade do seu tempo computacional e que a carga de trabalho foi perfeitamente balanceada, ou seja, a carga de trabalho adicional criada pela reexecução foi igualmente distribuída entre os processadores disponíveis durante a execução.

Considerando as suposições apresentadas para o cálculo do limite inferior, seja **N** o número de tarefas da aplicação, **m** o número de processadores alocados com o GM, **e** o tempo de execução em média estimado para cada tarefa e **pf** a percentagem de tarefas que falharam. Relembrando que para MTrab, e = 5 segundos, e para os Térmions, e = 4,4 segundos. O limite inferior estimado é calculado da seguinte forma:

$$E(pf) = \begin{cases} \lceil \frac{N}{m} \rceil \times e & \text{se } pf = 0 \\ \frac{(N \times e) + ((pf \times N) \times \frac{e}{2})}{m} & \text{se } pf > 0 \end{cases} \quad (5.1)$$

[1] O coeficiente de variação é igual ao desvio-padrão dividido pela média.

A Equação 5.1 mostra que existem dois casos para o cálculo do limite inferior estimado. O primeiro, no caso de pf igual a 0, é utilizado o teto da divisão entre o número de tarefas da aplicação e o número de processadores, pois uma tarefa não pode ser dividida. No segundo caso, para valores de pf maior que 0, o cálculo é feito pela soma dos tempos gastos na execução das N tarefas e mais a metade do tempo de execução das tarefas que falharam, supondo que as tarefas foram mortas na metade do seu tempo computacional, dividido pelo número de processadores, supondo que a carga de trabalho foi perfeitamente balanceada.

A sobrecarga apresentada na Tabela 5.1 é calculada da seguinte forma:

$$S = \frac{R - E(pf)}{E(pf)} \tag{5.2}$$

A Tabela 5.1 mostra que a sobrecarga não foi maior do que 7% do limite inferior estimado. Este valor pode ser considerado baixo, já que os custos de detecção da falha, tal como o tempo de espera para determinar se uma tarefa falhou, realocação das tarefas (escalonamento dinâmico) e recuperação junto com a reexecução estão incluídos. Observe que também estão incluídos os custos de execução dos gerenciadores do SGA EasyGrid, que neste experimento são 1 GG, 1 GS e 28 GM's. Vale ressaltar que o escalonamento dinâmico ajuda a diminuir a sobrecarga, já que contribui para o balanceamento da carga de trabalho adicional.

O segundo experimento também utilizou um ambiente grade semi-controlado com 4 *sites*. Os *sites* são interconectados por um *switch* Fast Ethernet. Todos os processadores disponíveis executam Globus Toolkit 2 e LAM/MPI 7.0.6. Os *Sites* 1, 2 e 3 são compostos de 28 processadores Pentium IV de 2.6 GHz com 512 Mb de RAM executando Linux Fedora Core 2 e 3 processadores Pentium IV de 3.2 GHz com 512 Mb de RAM executando Debian 1 (2 processadores no *Site* 2 e 1 no *Site* 3). Os *Sites* 1 e 3 são compostos de 8 processadores cada e o *Site* 2 com 15 processadores. O *Site* 4 é composto de 22 Pentium II de 400 MHz com 256 Mb de RAM executando Red Hat Linux 3. As máquinas dentro dos *Sites* 1 e 4 são interconectadas por um *switch* Fast Ethernet para cada *site* e as máquinas dentro dos *Sites* 2 e 3 são ligadas através de um *switch* Gigabit para cada *site*. Os *Sites* 1, 2 e 3 estão localizados no Instituto de Computação da UFF e o *Site* 4 no Departamento de Informática da PUC-Rio. O *site* da PUC-Rio se conecta ao *site* da UFF através da rede giga. Este ambiente foi utilizado para mostrar a integração entre as camadas de escalonamento dinâmico e tolerância a falhas para o caso de *sites* heterogêneos, onde as cargas de trabalho escalonadas nas máquinas precisam ser ajustadas de acordo com o seu poder computacional.

A aplicação MTrab na versão 1PTask com o SGA embutido foi utilizada neste experimento. Nesta configuração, 53 recursos estão disponíveis para a execução, mas somente 52 executam processos trabalhadores. Um dos recursos localizados no Site 2 foi escolhido para executar o processo mestre da aplicação e o GG do SGA EasyGrid. Os demais recursos executam um processo GM que gerencia a criação e a execução de cada processo trabalhador alocado para sua respectiva máquina. Adicionalmente, um recurso dentro de cada *site* foi escolhido para executar um GS para gerenciar a execução da aplicação dentro do seu *site*. Assim, estes processadores possuem uma sobrecarga extra produzida pela execução dos dois processos gerenciadores (GS e GM).

O tempo de execução da aplicação MTrab, com 1000 unidades de trabalho de 5 segundos cada, foi de 164,00 segundos para um cenário sem falhas no ambiente apresentado, mas com algumas cargas extras sendo executadas por programas CPU-bound externos para provocar maior heterogeneidade. Três cenários de falhas (F) foram avaliados: 1% das tarefas da aplicação falhando (10 processos); 3% das tarefas da aplicação falhando (30 processos); e 5% das tarefas da aplicação falhando (50 processos). Para simular as falhas de processos, um *script* com o comando **kill** matou o número respectivo de processos em cinco recursos diferentes distribuídos através dos *Sites* 1, 2 e 3. Além disso, o número total de processos mortos foi igualmente distribuído entre estes 5 recursos. Duas situações diferentes foram consideradas com relação ao momento em que os processos começaram a ser mortos: 50 segundos depois da aplicação iniciar; e 100 segundos depois. Em cada um dos recursos escolhidos, os processos foram mortos em intervalos de 6 segundos. Assim, por exemplo, no caso de 50 processos mortos, 10 por processador, a aplicação teria falhas de processos durante aproximadamente 60 segundos nos cinco recursos.

Tabela 5.2: Tempos de execução realizados pelo MTrab com o SGA EasyGrid (em segundos) através dos cenários de falhas e a sobrecarga (S) produzida na falha depois de 100 segundos em relação a de 50 segundos

(F)	Depois de 50s	Depois de 100s	S(%)
1%	168,22	168,07	-0,09
3%	172,25	172,55	0,17
5%	176,53	178,62	1,18

A Tabela 5.2 apresenta o desempenho da aplicação MTrab com o SGA embutido nos três cenários de falhas e a sobrecarga do resultado obtido na falha depois de 100 segundos em relação a de 50 segundos. Duas características importantes dos mecanismos de tolerância a falhas embutidos no SGA podem ser destacadas. A primeira é que a penalidade da reexecução destes processos é baixa, sendo proporcional a quantidade de falhas e a perda

média da unidade de trabalho (2,5 segundos). Embora uma sobrecarga seja causada nos recursos onde as falhas ocorreram, o custo do tempo extra gasto com a reexecução dos processos foi eficientemente assimilado pelo escalonador dinâmico. Outra conclusão observada dos resultados é que o desempenho da aplicação não depende do momento da falha, desde que o escalonador dinâmico tenha tempo para balancear a carga de trabalho igualmente. Este fato pode ser comprovado pelas baixas sobrecargas apresentadas nos cenários com 1% e 3% de falhas, onde os tempos de execução da aplicação foram muito próximos independentemente do momento das falhas. Mesmo em uma situação relativamente extrema, onde 5% de falhas ocorreram depois de 100 segundos e a última falha de processo ocorreu quase no final da execução, o aumento no tempo de execução foi menos do que 1,2% em relação ao teste com falhas depois de 50 segundos.

5.2 Falhas nos Processos Gerenciadores GS e GM

Devido a hierarquia dos processos gerenciadores no SGA EasyGrid é necessário tratar as falhas nos três níveis da estrutura. Caso contrário, uma falha em um dos níveis causa a perda de todos os processos nos níveis inferiores ao ponto da falha. Neste capítulo é a primeira vez que serão tratadas falhas de processadores. Esta seção é dividida em duas partes: a Seção 5.2.1, que utiliza o SGA EasyGrid com tolerância a falhas em dois níveis de gerenciadores (GS e GM); e a Seção 5.2.2, que usa a versão com tolerância a falhas em todos os níveis da estrutura hierárquica do SGA EasyGrid e compara os seus resultados com os obtidos anteriormente.

5.2.1 SGA EasyGrid com Tolerância a Falhas nos Gerenciadores GS e GM

Em (SILVA; REBELLO, 2007) foram apresentados os experimentos realizados para demonstrar a habilidade do SGA EasyGrid em tratar as falhas do tipo colapso nos seus processos gerenciadores GS e GM.

Os experimentos foram executados em um ambiente grade semi-controlado com 3 *sites* interconectados entre si por um *switch* Fast Ethernet. Todos os recursos disponíveis possuem instalados Linux Fedora Core 2, Globus Toolkit 2.4 e LAM/MPI 7.0.6. Os *sites* 1, 2 e 3 são compostos de processadores Pentium IV 2.6 GHz com 512 Mb de RAM, onde o Site 1 contém 13 processadores e os Sites 2 e 3 tem 7 e 5 processadores, respectivamente.

As máquinas dentro dos *Sites* 1 e 2 são ligadas através de um *switch* Gigabit para cada *site* e as máquinas dentro do *Site* 3 são interconectadas por um *switch* Fast Ethernet. Os três *sites* estão localizados no Instituto de Computação da UFF. A seguir, os tempos mostrados nos resultados foram obtidos a partir da média de três execuções, onde o coeficiente de variação da média calculada foi menor do que 3,5% para todos os experimentos.

A aplicação MTrab, nas duas versões 1PProc e 1PTask, foi utilizada neste experimento. Nesta configuração, 25 recursos estão disponíveis para a execução, sendo uma máquina do *Site* 1 dedicada para o processo mestre e 24 para executarem os processos trabalhadores. No caso da aplicação com o SGA EasyGrid, o mesmo recurso escolhido para executar o processo mestre da aplicação foi usado para executar o GG. Os demais recursos executam um processo GM cada e os processos trabalhadores. Adicionalmente, um recurso dentro de cada *site* foi escolhido para executar um GS para gerenciar a execução da aplicação dentro do seu *site*. Assim, estes recursos possuem uma sobrecarga extra produzida pela execução dos dois processos gerenciadores (GS e GM).

Tabela 5.3: Tempo de execução médio para a aplicação MTrab variando o número de processos e granularidades sobre diferentes cenários: Caso 0 - LAM/MPI puro; Caso 1 - SGA sem código de TF.

G	P	Caso 0	Caso 1	S(%)
5	500	105,25	107,96	2,57
	1000	209,59	215,71	2,92
	2000	418,47	425,20	1,61
	4000	831,25	846,04	1,78
	8000	1661,69	1686,79	1,51
10	500	209,79	218,16	3,99
	1000	418,49	426,99	2,03
	2000	836,16	846,35	1,22
	4000	1661,70	1689,58	1,68
	8000	3322,50	3372,26	1,50
20	500	418,88	439,27	4,87
	1000	836,17	853,37	2,06
	2000	1678,49	1695,52	1,01
	4000	3322,64	3380,69	1,75
	8000	6690,61	6744,91	0,81
40	500	836,99	871,62	4,14
	1000	1671,58	1722,36	3,04
	2000	3342,40	3383,75	1,24
	4000	6644,59	6754,70	1,66
	8000	13287,94	13486,71	1,50

A primeira questão a ser investigada é a sobrecarga da aplicação MTrab, implementada no modelo de execução 1PTask, com o SGA EasyGrid embutido em comparação com a versão tradicional equivalente, que é no modelo 1PProc usado no LAM/MPI. A Tabela 5.3 apresenta o percentual de sobrecarga (S) para a aplicação MTrab SGA EasyGrid sem o

código de TF com 500, 1000, 2000, 4000 e 8000 processos (P) e granularidades (G) de 5, 10, 20 e 40 segundos. Na versão mestre-trabalhador de LAM/MPI, o processo mestre distribui as unidades de carga de trabalho, que são equivalentes as de um processo P da versão 1PTask, para 24 trabalhadores sob demanda. Assim, 500 processos com 5 segundos de granularidade na versão do SGA corresponde a 500 unidades de carga de trabalho com granularidade de 5 segundos na versão LAM/MPI.

Como é esperado, o resultado da Tabela 5.3 mostra que a execução do SGA é um pouco mais lenta do que a do LAM/MPI puro para um ambiente homogêneo. Este cenário é revertido em grades heterogêneas compartilhadas (SENA et al., 2007).

Tabela 5.4: Tempo de execução médio para a aplicação MTrab variando o número de processos e a granularidade sobre diferentes cenários: Caso 1 - SGA sem código de TF; Caso 2 - SGA com código de TF executado sem falhas; Caso 3 - Falhas Irrecuperáveis em 6 máquinas com GM; Caso 4 - Falhas Recuperáveis em 6 processos GM's; Caso 5 - Falha Irrecuperável na máquina do GS do *Site* 1; Caso 6 - Falha Recuperável no processo GS do *Site* 1.

P	Caso 1	Caso 2	S(%)	Caso 3	S(%)	Caso 4	S(%)	Caso 5	S(%)	Caso 6	S(%)
Granularidade 5s											
500	107,96	127,18	17,80	168,48	35,89	131,89	22,16	189,52	19,21	135,65	25,65
1000	215,71	229,31	6,31	295,24	19,12	255,29	18,35	350,22	10,18	257,06	19,17
2000	425,20	449,80	5,78	550,08	11,67	456,73	7,42	672,68	6,34	474,47	11,59
4000	846,04	892,33	5,47	1034,97	5,28	905,42	7,02	1325,32	5,35	910,41	7,61
8000	1686,79	1833,19	8,68	2063,06	5,34	1872,60	11,02	2654,99	5,63	2056,87	21,94
Granularidade 10s											
500	218,16	231,22	5,99	291,89	17,19	241,83	10,85	372,59	16,77	248,62	13,96
1000	426,99	440,51	3,17	539,13	9,25	454,53	6,45	691,11	9,10	469,29	9,91
2000	846,35	863,18	1,99	1020,59	3,81	879,47	3,91	1327,94	5,13	883,06	4,34
4000	1689,58	1710,47	1,24	1997,45	1,66	1726,39	2,18	2604,79	3,58	1729,75	2,38
8000	3372,26	3411,71	1,17	3982,15	1,69	3426,19	1,60	5120,59	1,88	3428,95	1,68
Granularidade 20s											
500	439,27	454,00	3,35	544,16	8,91	473,91	7,88	705,20	10,25	482,46	9,83
1000	853,37	878,86	2,99	1029,30	4,32	898,02	5,23	1369,80	8,14	906,49	6,22
2000	1695,52	1719,37	1,41	2024,58	2,89	1746,47	3,00	2600,77	2,89	1763,39	4,00
4000	3380,69	3411,79	0,92	3993,79	1,61	3436,36	1,65	5129,99	1,98	3445,70	1,92
8000	6744,91	6791,23	0,69	7945,10	1,44	6819,75	1,11	10169,30	1,16	6821,11	1,13
Granularidade 40s											
500	871,62	897,55	2,97	1047,49	5,19	919,78	5,52	1380,10	8,17	951,87	9,21
1000	1722,36	1748,57	1,52	2053,40	3,64	1777,21	3,18	2736,37	7,68	1806,06	4,86
2000	3383,75	3429,97	1,37	4025,92	2,39	3480,69	2,86	5190,76	2,75	3496,43	3,33
4000	6754,70	6814,09	0,88	7963,06	1,35	6861,38	1,58	10239,25	1,81	6883,20	1,90
8000	13486,71	13561,43	0,55	15748,25	0,54	13594,67	0,80	20659,57	2,77	13622,61	1,01

O próximo passo é medir o grau de intrusão causado pelo próprio mecanismo de autorrecuperação. As colunas rotuladas Caso 1 e Caso 2 na Tabela 5.4 referem-se ao tempo de execução médio, em segundos, da aplicação MTrab com as duas versões do SGA EasyGrid (sem e com o código de tolerância a falhas para os GS's, GM's e processos da aplicação) em um ambiente sem falhas. A coluna adjacente ao Caso 2 apresenta o percentual de sobrecarga (S) para o SGA com tolerância a falhas. Para um dado número de processos, a sobrecarga de monitoramento da tolerância a falhas diminui com o aumento

da granularidade dos processos, visto que o tempo de execução é maior, mas o número de mensagens permanece o mesmo. Entretanto, para uma dada granularidade, quando o número de processos na aplicação é aumentado, a sobrecarga também diminui ao invés de permanecer constante como esperado. Esse fato ocorre devido a habilidade do SGA para realocar os processos da aplicação superando o desbalanceamento de carga causado pelo subsistema de tolerância a falhas.

Dois aspectos precisam ser avaliados considerando o grau de intrusão quando recuperando de falhas do tipo colapso. O custo de redistribuir processos da aplicação entre outros processos gerenciadores quando uma falha irrecuperável ocorre e o custo de recriar o gerenciador e/ou processos da aplicação no caso de uma falha recuperável. No Caso 3 da Tabela 5.4, 25% (6 máquinas) dos recursos executando GM's falharam, já no Caso 4, 25% dos processos GM's falharam (o *daemon* do LAM/MPI permaneceu ativo permitindo a recriação dos GM's). Nos Casos 5 e 6, os cenários anteriores são repetidos, mas desta vez o GS com metade dos recursos computacionais (12 máquinas) sofre uma falha irrecuperável e recuperável, respectivamente. A falha ocorre aproximadamente na metade do tempo de execução da aplicação medido no Caso 2.

As sobrecargas mostradas na Tabela 5.4 são relativas ao Caso 1 e nos Casos 3 e 5 são levados em consideração a perda de poder computacional disponível no ambiente. Com exceção das execuções de MTrab para 500 processos e para 5s de granularidade, as sobrecargas para a recuperação são pequenas quando comparadas ao programa equivalente sem o código de TF. Como anteriormente, estas sobrecargas diminuem para menos do que 2% quando o número de processos e granularidade crescem, sendo a exceção o teste de 8000 processos com granularidade 40 segundos que obteve sobrecarga de 2,77% no caso de falha irrecuperável no GS do *site* 1. Uma justificativa para esta exceção é a grande quantidade de tarefas da aplicação que precisam ser reescalonadas e pelo alto grau de granularidade que dificulta o melhor balanceamento de carga.

Uma observação interessante pode ser feita com relação ao modelo de execução. No Caso 1, SGA sem o código de TF, os melhores tempos de execução para uma dada carga total de trabalho foram realizados pelos programas com processos de granularidade 5s. Este fato ocorre, porque o SGA EasyGrid não trabalha com migração de processos, ou seja, depois de um processo começar sua execução em um recurso este não pode ser transferido para outro recurso. Assim, o escalonamento dinâmico realiza um melhor balanceamento de carga com os processos de menor granularidade. Nos casos restantes, os melhores tempos são geralmente realizados pelos programas com pequenas granularidades

(10s), embora as menores sobrecargas de TF são obtidas para maiores granularidades. O motivo do melhor desempenho para granularidades de 10 segundos é o resultado de um compromisso entre as demandas dos mecanismos de escalonamento dinâmico e de tolerância a falhas. O subsistema de escalonamento dinâmico tem o seu melhor desempenho para a granularidade de 5s, mas a sobrecarga de monitoramento da tolerância a falhas diminui com o aumento da granularidade dos processos. Lembrando que a sobrecarga do monitoramento da tolerância a falhas está relacionada ao número de mensagens trocadas pela aplicação. Dessa forma, quanto maior a granularidade para um mesmo número de mensagens menor será a sobrecarga.

Com base na Tabela 5.4, as Figuras 5.3 e 5.4 mostram a sobrecarga dos mecanismos de tolerância a falhas adicionados ao SGA EasyGrid, respectivamente, no caso de falhas irrecuperáveis na máquina do GS do Site 1 e em 6 máquinas com GM. Este tipo de falha é comum nos ambientes de grades computacionais, pois o administrador do *site* pode decidir pela exclusividade dos recursos a qualquer momento. Como pode-se observar nas figuras, existe a tendência da sobrecarga diminuir com o aumento da granularidade e do número de tarefas. No caso de 8000 tarefas de 5 segundos, a sobrecarga é menor do que 6% mesmo quando a perda é de metade do poder computacional disponível (falha no Site 1) e o subsistema de escalonamento dinâmico precisa reescalonar uma grande quantidade de tarefas.

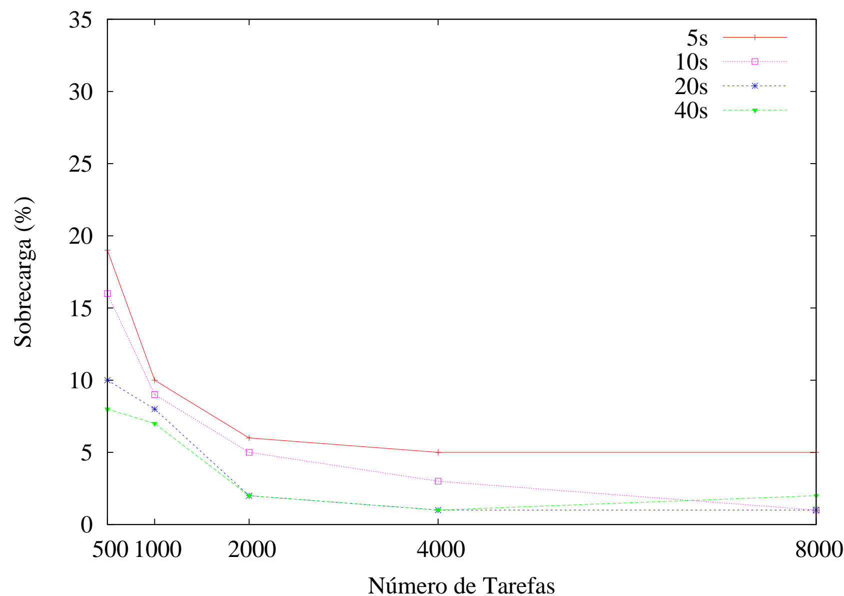


Figura 5.3: Sobrecarga do SGA no caso de falha irrecuperável na máquina do GS do Site 1.

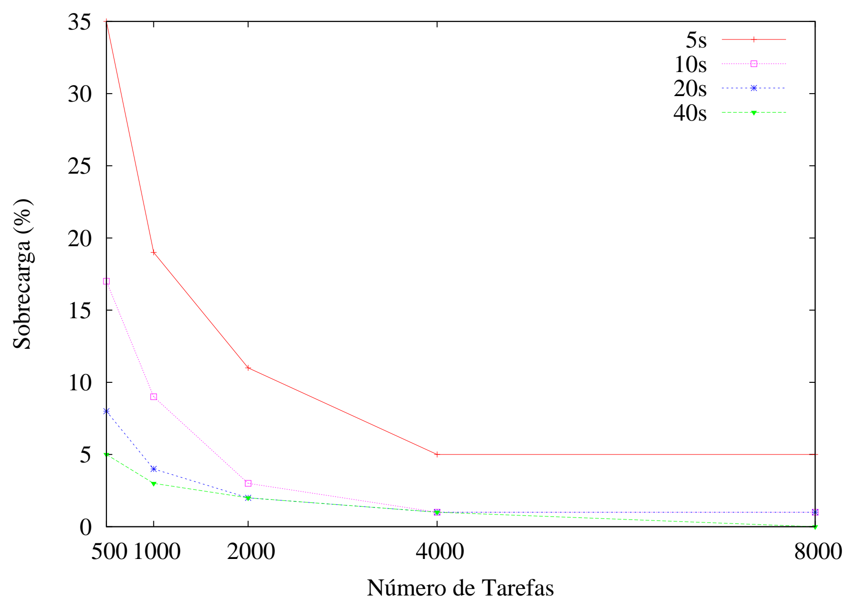


Figura 5.4: Sobrecarga do SGA no caso de falhas irrecuperáveis em 6 máquinas com GM.

Com base na Tabela 5.4, as Figuras 5.5 e 5.6 mostram a sobrecarga dos mecanismos de tolerância a falhas adicionados ao SGA EasyGrid, respectivamente, no caso de falhas recuperáveis na máquina do GS do Site 1 e em 6 máquinas com GM. As figuras mostram que também existe a tendência da sobrecarga diminuir com o aumento da granularidade e do número de tarefas. Entretanto, no caso de 8000 tarefas de 5 segundos, a sobrecarga aumentou para os dois casos.

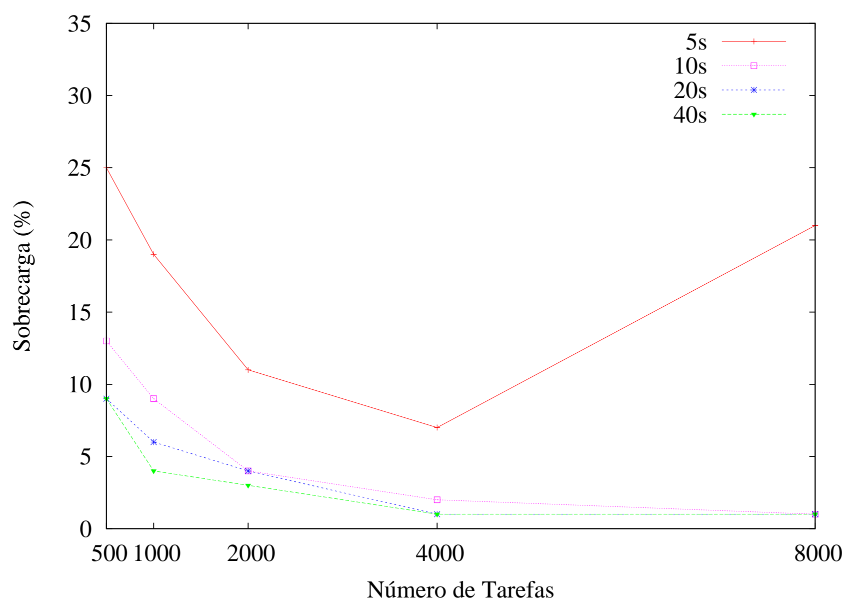


Figura 5.5: Sobrecarga do SGA no caso de falha recuperável no processo GS do Site 1.

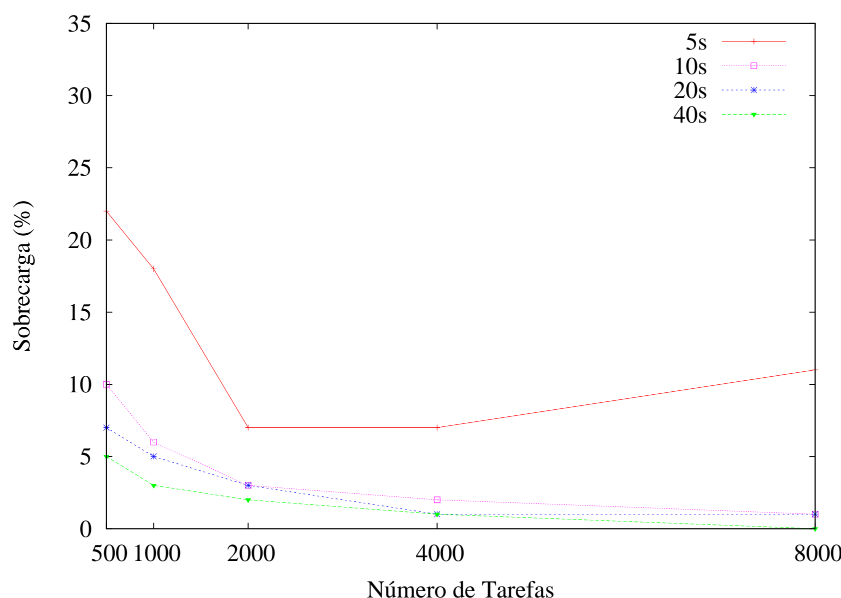


Figura 5.6: Sobrecarga do SGA no caso de falhas recuperáveis em 6 processos GM's.

A explicação para este fato é que a concorrência do recebimento das mensagens de entrada para o gerenciador recriado e a mensagem de resposta dos processos trabalhadores causam uma maior sobrecarga nos processos gerenciadores envolvidos, já que estes devem receber e gerenciar estes dois tipos de mensagens ao mesmo tempo. Este problema tem sua origem no gerenciamento das mensagens pela implementação LAM/MPI, já que pelo protocolo de comunicação utilizado as mensagens são enviadas para o processo destino independente da aplicação estar pronta para o recebimento e, no caso da aplicação não estiver preparada, o sistema tem que armazenar temporariamente as mensagens em *buffers* do sistema. Assim, será necessário um mecanismo de controle do fluxo de mensagens para tratar o problema (FONSECA, 2010). Este mesmo problema não ocorre nas granularidades maiores do que 5 segundos, pois o processo trabalhador demora mais tempo para executar e o processo gerenciador recriado consegue receber de forma mais rápida as mensagens de entrada para os processos da aplicação. Para o caso de 8000 tarefas de 5 segundos, a sobrecarga mostrada na Figura 5.5 é maior do que a da Figura 5.6, pois uma maior quantidade de recursos falham ao mesmo tempo quando a falha é no GS do *Site* 1.

O mesmo problema descrito para falhas recuperáveis de 8000 tarefas de 5 segundos não é tão grave no caso de falhas irrecuperáveis. Isto é devido ao fato de que nas falhas irrecuperáveis as tarefas não podem ser reescalonadas para os mesmos recursos e são divididas entre os recursos restantes pelo escalonador dinâmico. Assim, a sobrecarga do recebimento das mensagens de entrada é dividida entre os processos gerenciadores que ficarem responsáveis pela execução das tarefas reescalonadas.

5.2.2 SGA EasyGrid com Tolerância a Falhas em todos os Gerenciadores

Nesta seção serão utilizados os mesmos experimentos da Seção 5.2.1 para mostrar os custos dos mecanismos de tolerância a falhas do SGA EasyGrid no tratamento de falhas do processos gerenciadores GS e GM. A diferença é que nesta seção o SGA EasyGrid possui os mecanismos para tratar as falhas no processo gerenciador global (GG). Sendo esta versão a mais completa do SGA EasyGrid em relação aos mecanismos de tolerância a falhas, já que todos os níveis da estrutura hierárquica do SGA estão sendo tratados.

Os experimentos foram executados em um ambiente grade semi-controlado com 3 *sites* interconectados entre si por um *switch* Gigabit. Todos os recursos disponíveis possuem instalados Linux CentOS 5.3, Globus Toolkit 4.2 (Versão Pre-WS) e LAM/MPI 7.1.4. Os *sites* 1, 2 e 3 são compostos de processadores Pentium IV 2.6 GHz com 512 Mb de RAM, onde o Site 1 contém 13 processadores e os Sites 2 e 3 tem 7 e 5 processadores, respectivamente. As máquinas dentro dos *Sites* 1, 2 e 3 são ligadas através de um *switch* Gigabit para cada *site*. Os três *sites* estão localizados no Instituto de Computação da UFF. As máquinas usadas nesta subseção são as mesmas da subseção anterior, porém com uma atualização de SO, compiladores, entre outros. Os resultados a seguir foram obtidos a partir da média de três execuções para os casos onde o coeficiente de variação da média calculada foi menor do que 2% e de seis execuções para os demais casos, cujo o coeficiente de variação da média calculada foi menor do que 5%.

Como na subseção anterior, a aplicação MTrab, nas duas versões 1PProc e 1PTask, foi utilizada neste experimento e a mesma configuração foi usada. Ou seja, 25 recursos estão disponíveis para a execução, sendo uma máquina do *Site* 1 dedicada para o processo mestre e 24 para executarem os processos trabalhadores. No caso da aplicação com o SGA EasyGrid, o mesmo recurso escolhido para executar o processo mestre da aplicação foi usado para executar o GG. Os demais recursos executam um processo GM cada e os processos trabalhadores. Adicionalmente, um recurso dentro de cada *site* foi escolhido para executar um GS para gerenciar a execução da aplicação dentro do seu *site*. Assim, estes recursos possuem uma sobrecarga extra produzida pela execução dos dois processos gerenciadores (GS e GM).

Apesar das mudanças desta versão do SGA no mecanismo de tolerância a falhas não influenciar nos resultados da Tabela 5.3 e os recursos computacionais serem os mesmos da subseção anterior, os experimentos tiveram que ser refeitos. Um dos motivos da reexecução

se deve ao fato da atualização do sistema operacional nas máquinas, o que influenciou na aplicação sintética MTrab e mudou a quantidade de trabalho a ser executado por cada processo trabalhador. Outro motivo são algumas alterações no código base do SGA com o objetivo de melhorar a sua escalabilidade e o seu desempenho(FONSECA, 2010).

Como mostrado na Seção 5.2.1, a primeira questão a ser investigada é a sobrecarga da aplicação MTrab, implementada no modelo de execução 1PTask, com o SGA EasyGrid embutido em comparação com a versão tradicional equivalente, que é no modelo 1PProc usado no LAM/MPI. Comparando os resultados das Tabelas 5.3 e 5.5 podemos observar que a maioria das sobrecargas obtidas nesta subseção foram menores do que da subseção anterior. Este fato é explicado por uma diminuição nos custos de monitoramento, onde o usuário pode escolher uma maior ou menor quantidade de informações a serem coletadas pela camada de monitoramento. Entretanto, algumas modificações em determinadas estruturas de dados tornaram o código mais escalável, mas aumentaram o tempo de acesso, por exemplo, a mudança de vetor para tabela *hash*. Como justificado na subseção anterior, o resultado mostra que a execução do SGA é um pouco mais lenta do que a do LAM/MPI puro para um ambiente homogêneo.

Tabela 5.5: Tempo de execução médio para a aplicação MTrab variando o número de processos e granularidades sobre diferentes cenários: Caso 0 - LAM/MPI puro; Caso 1 - SGA sem código de TF.

G	P	Caso 0	Caso 1	S(%)
5	500	106,34	110,12	3,56
	1000	211,38	217,88	3,07
	2000	421,47	426,95	1,30
	4000	841,11	848,95	0,93
	8000	1677,18	1695,32	1,08
10	500	212,07	219,10	3,32
	1000	422,20	435,79	3,22
	2000	841,78	855,19	1,59
	4000	1681,65	1695,34	0,81
	8000	3353,05	3386,18	0,99
20	500	423,08	437,87	3,49
	1000	843,24	871,09	3,30
	2000	1682,37	1703,56	1,26
	4000	3363,18	3396,07	0,98
	8000	6711,95	6773,53	0,92
40	500	845,39	874,30	3,42
	1000	1686,05	1740,47	3,23
	2000	3364,58	3403,10	1,14
	4000	6717,16	6794,29	1,15
	8000	13412,21	13530,92	0,89

A seguir, será medido o grau de intrusão causado pelo próprio mecanismo de autorrecuperação. As colunas rotuladas Caso 1 e Caso 2 na Tabela 5.6 refere-se ao tempo de

execução médio, em segundos, da aplicação MTrab nas versões sem e com o código de tolerância a falhas em um ambiente sem falhas. A coluna adjacente ao Caso 2 apresenta o percentual de sobrecarga (S) para o SGA com tolerância a falhas.

Tabela 5.6: Tempo de execução médio para a aplicação MTrab variando o número de processos e a granularidade sobre diferentes cenários: Caso 1 - SGA sem código de TF; Caso 2 - SGA com código de TF executado sem falhas; Caso 3 - Falhas Irrecuperáveis em 6 máquinas com GM; Caso 4 - Falhas Recuperáveis em 6 processos GM's; Caso 5 - Falha Irrecuperável na máquina do GS do *Site* 1; Caso 6 - Falha Recuperável no processo GS do *Site* 1.

P	Caso 1	Caso 2	S(%)	Caso 3	S(%)	Caso 4	S(%)	Caso 5	S(%)	Caso 6	S(%)
Granularidade 5s											
500	110,12	112,99	2,61	148,25	18,54	117,84	7,01	140,75	27,89	122,09	10,87
1000	217,88	223,57	2,61	278,11	11,72	228,26	4,76	254,33	16,17	231,15	6,09
2000	426,95	439,07	2,84	528,37	7,07	442,33	3,60	470,89	8,63	447,05	4,71
4000	848,95	872,99	2,83	1032,68	4,90	874,44	3,00	921,75	7,25	883,39	4,06
8000	1695,32	1736,90	2,45	2032,94	3,58	1741,94	2,75	1798,40	4,70	1744,53	2,90
Granularidade 10s											
500	219,10	224,96	2,67	266,31	6,71	231,94	5,86	265,66	21,00	239,00	9,08
1000	435,79	445,60	2,25	525,97	5,64	451,63	3,63	483,68	10,46	457,58	5,00
2000	855,19	875,11	2,33	1040,21	5,33	884,84	3,47	935,44	7,82	886,29	3,64
4000	1695,34	1736,04	2,40	2048,53	4,11	1737,15	2,47	1802,87	4,96	1748,80	3,15
8000	3386,18	3463,85	2,29	4063,88	3,59	3462,59	2,26	3563,89	3,81	3471,70	2,53
Granularidade 20s											
500	437,87	449,22	2,59	523,15	4,85	455,80	4,09	493,19	12,36	466,03	6,43
1000	871,09	876,54	0,63	1034,37	3,90	882,99	1,37	926,46	5,82	897,66	3,05
2000	1703,56	1712,64	0,53	2023,99	2,65	1740,07	2,14	1802,77	4,10	1742,78	2,30
4000	3396,07	3423,61	0,81	4000,60	1,59	3445,00	1,44	3548,05	3,20	3449,38	1,57
8000	6773,53	6824,65	0,75	7968,76	1,55	6838,27	0,96	7006,27	2,03	6864,63	1,34
Granularidade 40s											
500	874,30	895,53	2,43	1032,88	3,58	898,92	2,82	978,95	11,61	922,64	5,53
1000	1740,47	1732,28	-0,47	2051,65	3,09	1755,44	0,86	1851,02	5,76	1797,30	3,27
2000	3403,10	3434,66	0,93	4023,71	2,08	3466,55	1,86	3577,50	3,35	3486,87	2,46
4000	6794,29	6831,41	0,55	7988,16	1,41	6891,54	1,43	7060,91	2,67	6895,30	1,49
8000	13530,92	13633,93	0,76	15904,90	1,40	13649,12	0,87	13958,42	1,70	13686,48	1,15

Comparando os resultados das Tabelas 5.4 e 5.6 podemos observar que a maioria das sobrecargas obtidas nesta subseção foram menores do que da subseção anterior. Este fato é explicado por uma melhoria na implementação do mecanismo de tolerância a falhas. Como explicado na Seção 4.3.2, um gerenciador que suspeita de falha de outro gerenciador cria uma *thread* para verificar o estado do gerenciador remoto. Caso não tenha uma resposta imediata, este gerenciador aguarda uma resposta da *thread* em uma série de tempos de espera que são aumentados exponencialmente até um limite máximo configurável. Estes tempos crescidos exponencialmente prejudicam o desempenho do gerenciador, pois este pode ficar bloqueado desnecessariamente por uma grande quantidade de tempo, o que impede o gerenciador de desenvolver suas atividades a favor da aplicação do usuário, tal como o redirecionamento das mensagens para os processos da aplicação. A mudança foi que o gerenciador verifica a resposta da *thread* em pequenos intervalos de tempo até o limite máximo. Com isso, um gerenciador com retardo no canal de comunicação é identificado como sem falhas em menos tempo. Esta mudança é melhor vista na menor

sobrecarga medida para a granularidade de 5s, onde o tempo de espera menor teve um impacto significativo na sobrecarga para todas as quantidades de tarefas. Além disso, vale ressaltar que a sobrecarga do mecanismo de tolerância a falhas foi menor do que 3% para todas as granularidades e quantidades de tarefas experimentadas em um ambiente sem falhas. O grau de intrusão negativo para o caso de 1000 tarefas e granularidade de 40s mostra que neste caso a intrusão é tão baixa que a própria variação dos tempos de execução do programa paralelo foi maior que a intrusão provocada pelos mecanismos de tolerância a falhas. A sobrecarga dos mecanismos de tolerância a falhas, tanto para o caso sem falhas quanto para os cenários de falhas, foram obtidos com o Gerenciador Global realizando o seu *checkpoint* no nível da aplicação a cada minuto de execução. Por exemplo, no experimento com 8000 tarefas de 40 segundos foram realizados 225 *checkpoints* no SGA com código de TF sem falhas (caso 2), já que o tempo de execução foi em média de 13633,93 segundos e neste tempo está incluído o custo dos *checkpoints* realizados.

Com base na Tabela 5.6, as Figuras 5.7 e 5.8 mostram a sobrecarga dos mecanismos de tolerância a falhas adicionados ao SGA EasyGrid, respectivamente, no caso de falhas irrecuperáveis na máquina do GS do Site 1 (Caso 5) e em 6 máquinas com GM (Caso 3). Para a maioria dos resultados, a sobrecarga mostrada na Figura 5.7 é maior do que a apresentada na Figura 5.3, que mostra os resultados obtidos na Seção 5.2.1 para o mesmo caso de falha. Este fato pode ser explicado pelas estratégias diferentes de recuperação usadas nas duas versões do SGA EasyGrid. Na seção anterior, quando o recurso de um GS tinha uma falha irrecuperável, todas as máquinas do *site* eram perdidas e esta perda foi considerada no cálculo da sobrecarga. Já na versão atual, somente a máquina do GS é perdida, já que uma nova máquina é escolhida para executar o novo GS. Assim, a sobrecarga calculada nesta seção inclui o tempo gasto na recriação do novo GS e GM's e não considera o tempo em que as máquinas do *site* não podiam ser utilizadas até a criação do novo GM da máquina e o recebimento das tarefas a serem executadas. Consequentemente, apesar da maior sobrecarga, os tempos de execução na versão mais completa do SGA são menores do que os mostrados na seção anterior, já que somente uma máquina do *site* é perdida e não todos os recursos do *site*. Este fato pode ser comprovado pelos tempos mostrados para o Caso 5 nas Tabelas 5.6 e 5.4, respectivamente, versão com TF em todos os gerenciadores e versão com TF nos gerenciadores GS e GM. Já para o Caso 3, onde ocorrem falhas em 6 máquinas com GM, a sobrecarga mostrada na Figura 5.8 é menor, na maioria dos testes, do que a sobrecarga mostrada na Figura 5.4. Para os testes com sobrecarga maior, a justificativa é que a mudança feita nos gerenciadores, para verificar a resposta da *thread* em pequenos intervalos de tempo até o limite máximo, provoca

uma sobrecarga maior no caso das falhas irrecuperáveis e o escalonamento dinâmico não conseguiu esconder esta sobrecarga. Visto que a verificação da resposta da *thread* será feita até o limite máximo, já que nenhuma resposta será retornada no caso das falhas irrecuperáveis.

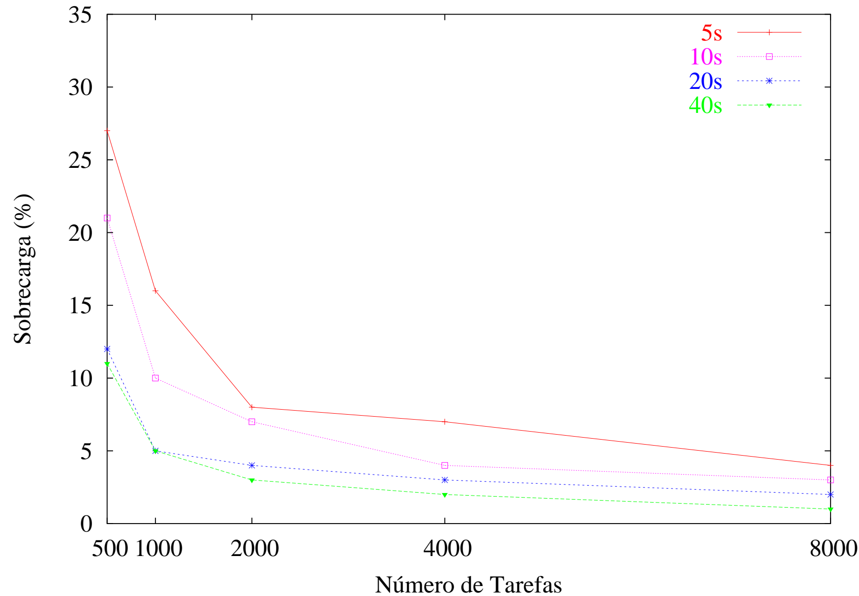


Figura 5.7: Sobrecarga do SGA no caso de falha irrecuperável na máquina do GS do Site 1.

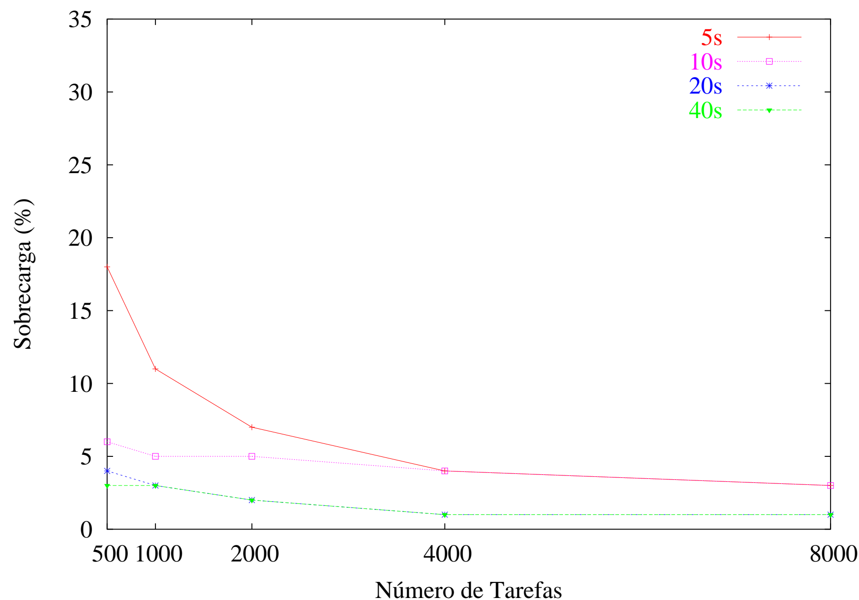


Figura 5.8: Sobrecarga do SGA no caso de falhas irrecuperáveis em 6 máquinas com GM.

Com base na Tabela 5.6, as Figuras 5.9 e 5.10 mostram a sobrecarga dos mecanismos de tolerância a falhas adicionados ao SGA EasyGrid, respectivamente, no caso de falhas recuperáveis na máquina do GS do Site 1 (Caso 6) e em 6 máquinas com GM (Caso 4). As

figuras mostram que também existe a tendência da sobrecarga diminuir com o aumento da granularidade e do número de tarefas.

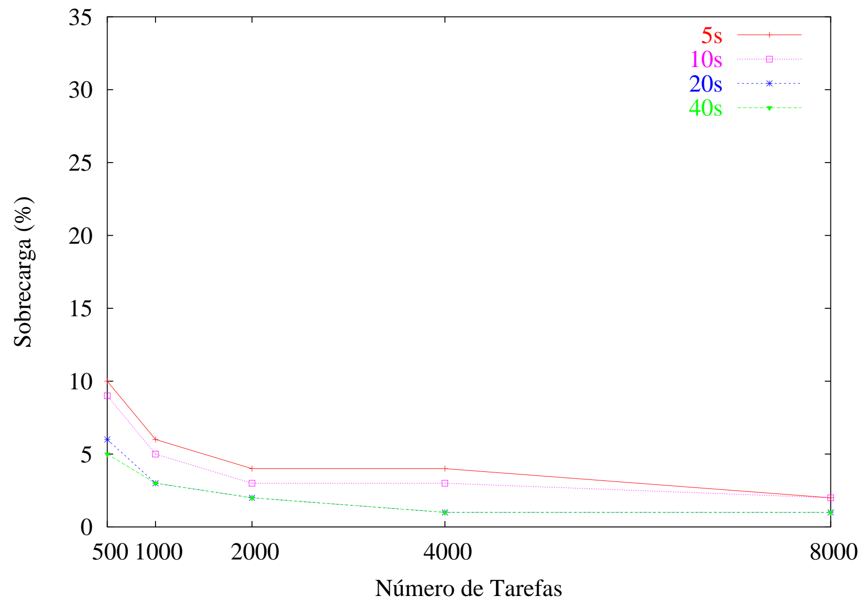


Figura 5.9: Sobrecarga do SGA no caso de falha recuperável no processo GS do Site 1.

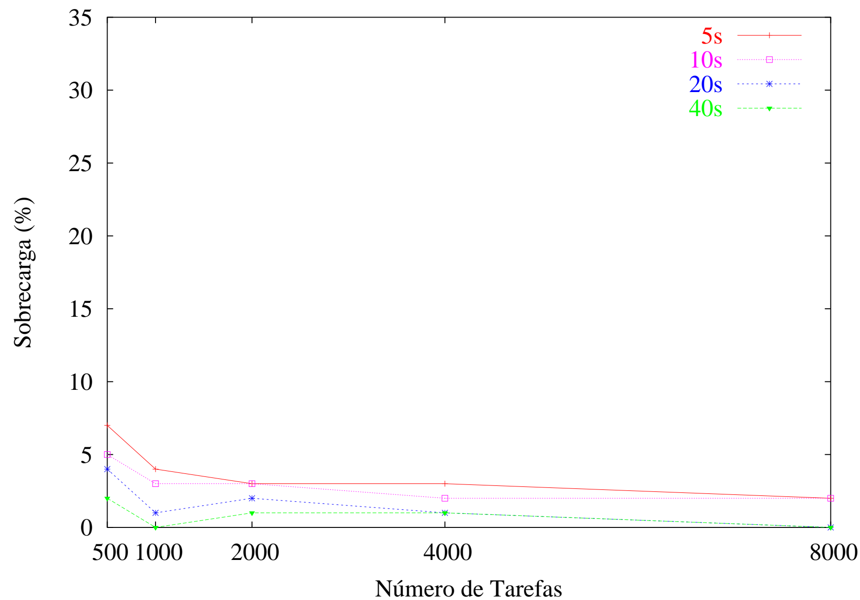


Figura 5.10: Sobrecarga do SGA no caso de falhas recuperáveis em 6 processos GM's.

Diferente do apresentado nas Figuras 5.5 e 5.6, no caso de 8000 tarefas de 5 segundos, a sobrecarga não aumentou para os dois casos. A explicação da melhora de desempenho do SGA está na diminuição dos intervalos de espera para a verificação de falha do processo gerenciador remoto. Com isso, houve uma melhora no desempenho dos gerenciadores e uma maior quantidade de mensagens puderam ser redirecionadas para a aplicação.

Desta forma, o problema de sobrecarga de mensagens foi sobreposto sem a necessidade de um mecanismo para o controle do fluxo de mensagens. Entretanto, com o crescimento do número de mensagens sendo reenviadas, devido a uma falha de um gerenciador do *site* com um grande número de máquinas, em certas situações ainda se faz necessário o desenvolvimento de um controle de fluxo de mensagens.

5.3 Falhas no Processo GG

Esta seção apresenta alguns experimentos para avaliar os custos do tratamento de falhas no Gerenciador Global. Como explicado na Seção 4.2.4, nesta tese foi usada a técnica de *checkpoint* no nível da aplicação para a recuperação do GG.

Os experimentos foram executados em um ambiente grade semi-controlado com 3 *sites* interconectados entre si por um *switch* Gigabit. Todos os recursos disponíveis possuem instalados Linux CentOS 5.3, Globus Toolkit 4.2 (Versão Pre-WS) e LAM/MPI 7.1.4. Os *sites* 1, 2 e 3 são compostos de processadores Pentium IV 2.6 GHz com 512 Mb de RAM, onde o Site 1 contém 13 processadores e os Sites 2 e 3 tem 7 e 5 processadores, respectivamente. As máquinas dentro dos *Sites* 1, 2 e 3 são ligadas através de um *switch* Gigabit para cada *site*. Os três *sites* estão localizados no Instituto de Computação da UFF. A seguir, os tempos mostrados nos resultados foram obtidos a partir da média de três execuções, onde o coeficiente de variação da média calculada foi menor do que 1,5% para todos os experimentos.

Os experimentos a seguir utilizaram a aplicação N-corpos com o algoritmo *ring* nas duas versões, que são **MPI ring** e **AMS ring**. Os objetivos destes experimentos são medir a sobrecarga gerada na aplicação *MPI ring* pela utilização do sistema de *checkpoint* coordenado do Berkeley Laboratory Checkpoint/Restart (BLCR), que é bastante utilizado em aplicações de alto desempenho e já possui uma integração com a implementação LAM/MPI, em comparação com a sobrecarga do mecanismo de tolerância a falhas implementado no SGA EasyGrid, que é utilizado pelo *AMS ring*. Além disso, os experimentos visam avaliar a sobrecarga gerada pela frequência dos *checkpoints* e o tempo para a recuperação da aplicação. Lembrando que na recuperação, o SGA EasyGrid tem a vantagem de se recuperar automaticamente de falhas envolvendo os processos gerenciadores do *site* e da máquina, bem como, as falhas dos processos da aplicação do usuário sem a necessidade de parar a execução da aplicação. Só no caso de falha do Gerenciador Global é que se faz necessário a utilização do arquivo de *checkpoint* com o estado do global. Já no caso

da aplicação MPI, sem o *middleware* SGA EasyGrid e usando o sistema de *checkpoint* do BLCR, sempre é necessário a interrupção da aplicação e a disponibilidade da mesma quantidade de recursos utilizada antes da falha para a recuperação da aplicação.

Para a medição das sobrecargas foram usados 12, 18 e 24 processadores homogêneos. Entretanto, como explicado anteriormente, o algoritmo *AMS ring* possui um custo maior de comunicação e a vantagem deste algoritmo está na utilização de processadores heterogêneos em ambientes compartilhados e dinâmicos, onde o escalonador dinâmico pode escalonar os processos menores para aproveitar melhor o desempenho dos processadores disponíveis (SENA et al., 2008; NASCIMENTO et al., 2009). Nos experimentos a seguir só estão sendo usados processadores homogêneos dedicados para uma medição mais precisa da sobrecarga dos mecanismos de tolerância a falhas, o que beneficia o algoritmo *MPI ring*. Contudo, nesta situação o algoritmo *ring* necessita de um certo sincronismo entre os processos em cada iteração do algoritmo para que o seu desempenho não seja prejudicado. Desta forma, a fim de não prejudicar o desempenho da aplicação *AMS ring*, para cada processador utilizado na execução da aplicação N-corpos foram alocados um GM, junto com um processo da aplicação, e as máquinas foram configuradas como pertencentes a um único *site*. Além disso, foi utilizado um mesmo processador adicional para alocar os processos GG e GS, o que prejudica o algoritmo *AMS ring*. Observe que usando somente um GS, este fará o redirecionamento de todas as mensagens entre as máquinas e o envio de uma cópia destas mensagens para o *log* do GG. Portanto, uma alta sobrecarga do mecanismo de *checkpoint* do GG acarretará em um retardo da aplicação, já que o GG e o GS compartilham o mesmo recurso. Em resumo, o experimento foi montado para favorecer a execução tradicional.

A Tabela 5.7 mostra os tempos de execução para a aplicação N-corpos com o algoritmo *ring* nas suas duas versões. As colunas **MPI** e **AMS** referenciam respectivamente as versões **MPI ring**, usando somente a biblioteca LAM/MPI, e **AMS ring**, usando o SGA EasyGrid. Ambas as versões executando sem qualquer mecanismo de tolerância a falhas. As colunas **Ckpt-60s** e **Ckpt-30s** mostram respectivamente os tempos de execução das duas versões com intervalos entre *checkpoints* de 60 e 30 segundos. Para isso, a versão *MPI ring* utiliza o sistema de *checkpoint* coordenado do BLCR e a versão *AMS ring* usa os mecanismos de tolerância a falhas incluídos no *middleware* SGA EasyGrid. Os valores entre parêntesis mostram o número de *checkpoints* realizados pelos respectivos mecanismos. As colunas **S** apresentam o percentual de sobrecarga dos mecanismos de tolerância a falhas sobre as versões sem estes mecanismos. Para comparar as versões utilizadas, o *AMS ring* usa o valor de W, largura do grafo, igual ao número de processadores

homogêneos dedicados utilizados.

Tabela 5.7: Tempo de execução médio em segundos para a aplicação N-corpos variando o número de processadores e o intervalo dos *checkpoints*.

P	MPI	Ckpt-60s	S(%)	Ckpt-30s	S(%)	AMS	Ckpt-60s	S(%)	Ckpt-30s	S(%)
					$N = 250000$					
12	208,61	225,40 (3)	8,05	237,91 (6)	14,05	207,04	211,19 (3)	2,00	212,22 (6)	2,50
18	137,76	148,80 (2)	8,01	160,77 (4)	16,70	135,51	140,04 (2)	3,35	141,20 (4)	4,20
24	103,42	109,58 (1)	5,95	118,66 (3)	14,73	101,50	107,50 (1)	5,91	108,36 (3)	6,76
					$N = 500000$					
12	840,73	926,09 (13)	10,15	1028,30 (25)	22,31	835,53	850,13 (13)	1,75	849,80 (26)	1,71
18	559,98	619,29 (8)	10,59	671,29 (16)	19,88	557,91	565,02 (9)	1,27	586,69 (18)	5,16
24	417,61	456,60 (6)	9,34	493,42 (12)	18,15	413,41	427,26 (6)	3,35	429,33 (13)	3,85
					$N = 1000000$					
12	3360,49	3808,30 (46)	13,33	4119,82 (78)	22,60	3339,95	3372,36 (51)	0,97	3380,01 (95)	1,20
18	2242,44	2628,86 (29)	17,23	2868,71 (47)	27,93	2227,26	2261,82 (34)	1,55	2269,38 (64)	1,89
24	1683,01	1970,88 (22)	17,10	2167,15 (37)	28,77	1671,89	1695,37 (26)	1,40	1698,93 (48)	1,62

A aplicação *MPI ring*, usando a biblioteca LAM/MPI e o BLCR *checkpoint*, não executou para o caso de 1000000 partículas usando 12, 18 e 24 processadores com intervalos entre *checkpoints* de 60 e 30 segundos. Durante a execução da aplicação, alguns *checkpoints* são realizados até que um pedido de *checkpoint* fique travado, o que pára toda a aplicação. Os valores apresentados na Tabela 5.7, para estes casos, foram uma estimativa calculada sobre os custos obtidos pela realização de um *checkpoint*.

Os resultados da Tabela 5.7 mostram que as sobrecargas causadas pelos mecanismos de tolerância a falhas do SGA EasyGrid são menores do que aqueles provocados pelo sistema de *checkpoint* do BLCR. A maior sobrecarga do SGA foi de 6,76% no caso de 250000 partículas para 24 processadores com intervalos entre *checkpoints* de 30 segundos. Isto se deve ao fato da sobrecarga do mecanismo de tolerância a falhas está relacionado ao número de mensagens enviadas pela aplicação, já que para cada mensagem redirecionada pelo GS este deve verificar o estado do processo GM destino. Além disso, a versão *AMS ring* com 250000 partículas divididas entre 24 processadores possui um tempo de execução total relativamente pequeno, igual a 101,50 segundos. Assim, apesar da sobrecarga ter sido de apenas 6,86 segundos, o impacto do mecanismo de tolerância a falhas causado pela maior comunicação gerou a maior sobrecarga. Entretanto, com um aumento no número de partículas, a granularidade de cada processo aumenta e o custo de monitoramento do estado dos processos passam a ter um impacto menor no tempo de execução total da aplicação. Por exemplo, os mecanismos de tolerância a falhas para 1000000 de partículas utilizando 24 processadores e realizando *checkpoints* com intervalos de 60 e 30 segundos produziram, respectivamente, uma sobrecarga de 1,40% e 1,62%.

Pelos dados da Tabela 5.7, outra característica importante do SGA EasyGrid é que

com a diminuição do intervalo entre *checkpoints* de 60 para 30 segundos, os tempos de execução da aplicação nas duas situações ficaram muito próximos, como pode ser visto nas Figuras 5.11, 5.12 e 5.13. A exceção foi o caso de 500000 partículas executando em 18 processadores, onde o intervalo de 30 segundos fica bem próximo do tempo de execução de cada processo da versão *AMS ring*. Desta forma, o momento da realização do *checkpoint* no global coincide com a fase de troca de mensagens entre os processos da aplicação. Por isso, houve um aumento na sobrecarga de 3,89%, já que o *checkpoint* no GG interferiu no desempenho do GS em cada um dos 18 *checkpoints* realizados. Isto foi comprovado diminuindo o intervalo entre *checkpoints* de 30 para 29 segundos e o tempo de execução da aplicação diminuiu de 586,69 segundos para 571,46 segundos, o que fez a sobrecarga reduzir de 5,16% para 2,43%. Ainda diminuindo o intervalo do *checkpoint* para 28 segundos, o tempo de execução caiu para 564,61 segundos, o que resultou em uma sobrecarga de 1,20%.

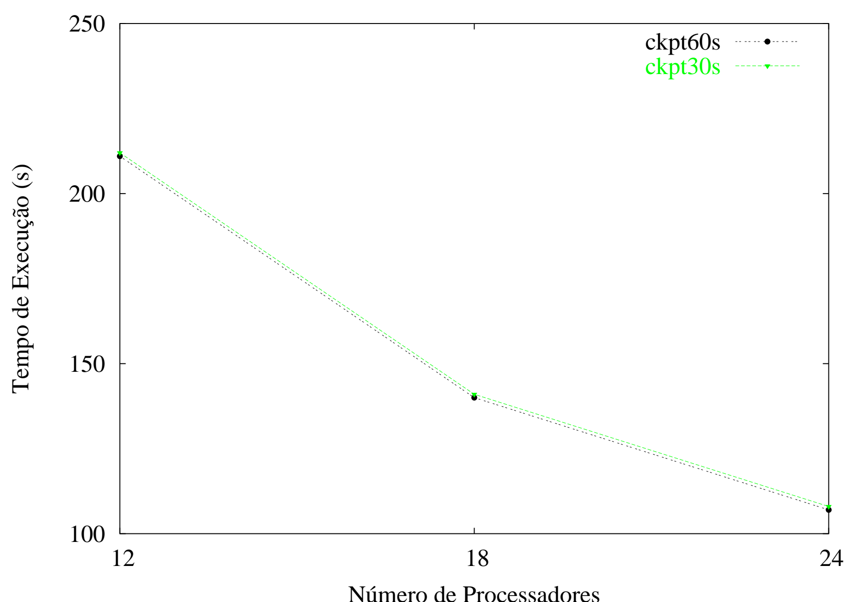


Figura 5.11: AMS ring com N=250000 e intervalos entre *checkpoints* iguais a 60 e 30 segundos.

Uma das vantagens do mecanismo de tolerância a falhas do SGA EasyGrid, em relação ao sistema de *checkpoint* coordenado do BLCR, é que somente se faz necessário realizar o *checkpoint* no processo GG em um meio de armazenamento estável. Ao contrário disso, o BLCR precisa coordenar e gravar o estado de todos os processos da aplicação em um meio estável. No experimento realizado, os arquivos de *checkpoint* de ambos os mecanismos foram gravados em um servidor NFS (*Network File System*). Outra vantagem é que o SGA EasyGrid realiza o *checkpoint* no nível da aplicação, onde somente os dados necessários

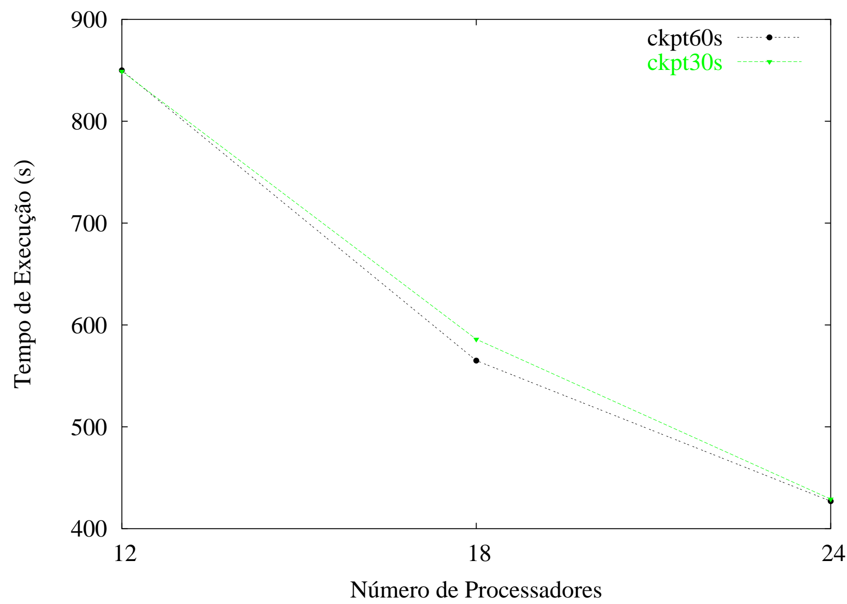


Figura 5.12: AMS ring com N=500000 e intervalos entre *checkpoints* iguais a 60 e 30 segundos.

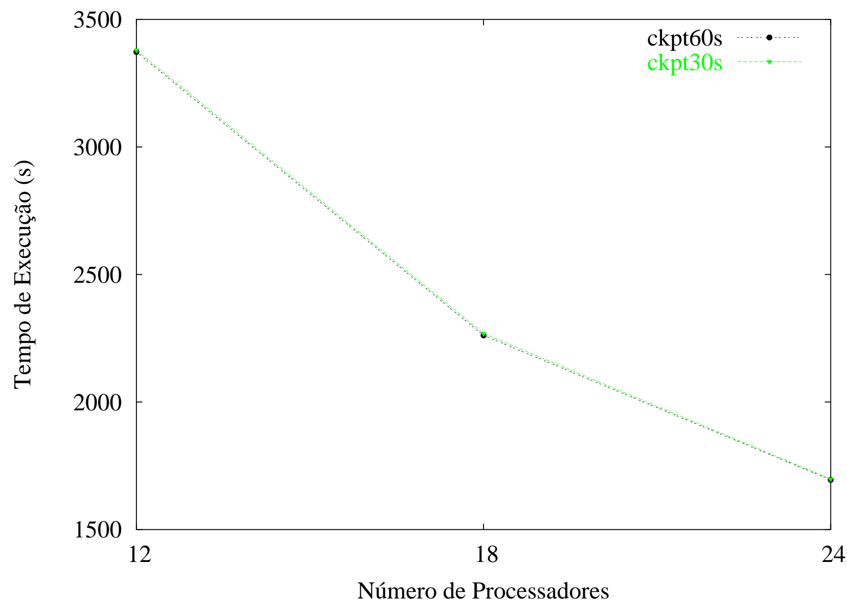


Figura 5.13: AMS ring com N=1000000 e intervalos entre *checkpoints* iguais a 60 e 30 segundos.

para a continuação da aplicação são gravados. Já o BLCR é um sistema de *checkpoint* de processo no nível do kernel e grava o estado completo de cada processo, mesmo que existam informações que não serão mais usadas pela aplicação. Para estes experimentos, o arquivo de *checkpoint* do GG teve um tamanho de aproximadamente dois terços da soma dos arquivos de *checkpoint* do BLCR. Por exemplo, para o caso de 1000000 partículas executando em 24 processadores, o GG gerou um arquivo de 70 Mb, enquanto o BLCR

gerou um total de 102 Mb. Por estas razões, a sobrecarga causada pelo sistema de *checkpoint* do BLCR aumenta consideravelmente com a diminuição do intervalo entre *checkpoints* de 60 para 30 segundos.

A Tabela 5.8 mostra os tempos de execução para a recuperação, depois de uma falha, da aplicação N-corpos com o algoritmo *ring* nas suas duas versões. As colunas **MPI_Ckpt** e **AMS_Ckpt** referenciam respectivamente as versões **MPI ring**, usando a biblioteca LAM/MPI e o BLCR *checkpoint*, e **AMS ring**, usando o SGA EasyGrid com o mecanismo de tolerância a falhas. Os valores entre parêntesis mostram o número de *checkpoints* realizados pelos respectivos mecanismos. A coluna **Intervalo Ckpt** mostra o intervalo de tempo, em segundos, em que os *checkpoints* foram realizados. A coluna **Recuperação** apresenta o tempo de execução total da aplicação, em segundos, somando-se o tempo de execução até o último *checkpoint* antes da falha e o tempo de execução deste a iniciação da recuperação a partir do arquivo de *checkpoint* gerado até a finalização. O objetivo principal deste experimento é medir a sobrecarga gerada pelos mecanismos ao buscar as informações dos arquivos de *checkpoint* e reestabelecer a execução da aplicação paralela, por isso não será considerado na sobrecarga o tempo adicional entre o momento da falha e o último *checkpoint* gerado. As colunas **S** apresentam o percentual de sobrecarga da recuperação da aplicação sobre as versões realizando *checkpoints*. Para comparar as versões utilizadas, o *AMS ring* usa o valor de W, largura do grafo, igual ao número de processadores homogêneos dedicados utilizados.

Tabela 5.8: Tempo de execução médio em segundos para a recuperação da aplicação N-corpos variando o número de processadores.

P	MPI_Ckpt	Intervalo Ckpt	Recuperação	S(%)	AMS_Ckpt	Intervalo Ckpt	Recuperação	S(%)
$N = 250000$								
12	225,40 (3)	60	230,23	2,14	212,45 (9)	21	215,96	1,65
18	148,80 (2)	60	155,32	4,39	141,95 (6)	20	145,39	2,42
24	109,58 (1)	60	114,22	4,24	108,05 (2)	39	111,75	3,43
$N = 500000$								
12	906,39 (10)	75	933,49	2,99	850,92 (10)	75	860,25	1,10
18	619,29 (8)	60	631,27	1,93	569,83 (15)	35	576,87	1,24
24	456,60 (6)	60	466,53	2,17	430,65 (17)	22	436,44	1,34
$N = 1000000$								
12	3457,84 (10)	298	3466,27	0,24	3372,76 (11)	297	3379,24	0,19
18	2428,99 (14)	138	2436,63	0,31	2255,78 (15)	138	2262,76	0,31
24	1892,37 (17)	84	1908,78	0,87	1694,95 (19)	79	1707,21	0,72

Para medir somente a sobrecarga da recuperação usando o SGA EasyGrid foi necessário ajustar o intervalo entre *checkpoints* de acordo com a granularidade dos processos da aplicação *AMS ring*. Ou seja, o *checkpoint* é realizado após uma ou mais iterações da aplicação terminar e as mensagens serem trocadas entre os processos, visto que o *check-*

point no GG é realizado sobre o *log* das mensagens geradas pelos processos da aplicação. Caso o *checkpoint* fosse realizado no meio da execução dos processos em uma iteração, todos os processos daquela iteração seriam reiniciados e esse tempo de reexecução adicional seria incluído na sobrecarga do mecanismo de recuperação da aplicação, o que faria a sobrecarga variar bastante de acordo com o momento da falha e da granularidade dos processos. Para o BLCR *checkpoint*, foi utilizado o intervalo de *checkpoints* igual a 60 segundos ou o mesmo intervalo usado no SGA EasyGrid nos casos de intervalos maiores do que 60 segundos.

Como acontecido anteriormente, a aplicação *MPI ring*, usando a biblioteca LAM/MPI e o BLCR *checkpoint*, não executou até o final para o caso de 1000000 partículas usando 12, 18 e 24 processadores com intervalos entre *checkpoints* iguais aos usados pelo SGA EasyGrid. Para o cálculo da sobrecarga, foi realizado apenas um *checkpoint* e a partir deste foi feita a recuperação da aplicação. Depois, foi adicionado o custo dos demais *checkpoints*, que deveriam ter sido realizados. Assim, os valores apresentados na Tabela 5.8, para estes casos, foram uma estimativa.

Os resultados da Tabela 5.8 mostram que as sobrecargas de recuperação da aplicação N-corpos usando o SGA EasyGrid não são maiores do que aqueles provocados pelo sistema de *checkpoint* do BLCR. Isso demonstra que os custos de leitura dos arquivos de *checkpoint* do servidor NFS, para cada processo da aplicação, no sistema BLCR foram maiores ou iguais a sobrecarga gerada pelo SGA EasyGrid em ler um único arquivo do mesmo servidor NFS, recriar toda a estrutura de gerenciadores durante a recuperação e reenviar as mensagens para os processos da aplicação recriados.

Mesmo o SGA EasyGrid realizando um maior número de *checkpoints*, a sobrecarga total gerada pelos mecanismos de tolerância a falhas do SGA EasyGrid foram menores do que a sobrecarga total gerada pelo sistema de *checkpoint* do BLCR, como mostrado na Tabela 5.9. As colunas **MPI** e **AMS** referenciam respectivamente as versões **MPI ring**, usando somente a biblioteca LAM/MPI, e **AMS ring**, usando o SGA EasyGrid. Ambas as versões executando sem qualquer mecanismo de tolerância a falhas. A coluna **Recuperação** apresenta o tempo de execução total da aplicação, em segundos, somando-se o tempo de execução até o último *checkpoint* antes da falha e o tempo de execução deste a iniciação da recuperação a partir do arquivo de *checkpoint* gerado até a finalização. Os valores entre parêntesis mostram o número de *checkpoints* realizados pelos respectivos mecanismos. As colunas **S** apresentam o percentual de sobrecarga total dos mecanismos de tolerância a falhas sobre as versões sem estes mecanismos, o que inclui na sobrecarga

os custos da recuperação.

Tabela 5.9: Sobrecarga total dos mecanismos de TF incluindo a recuperação da aplicação N-corpos variando o número de processadores.

P	MPI	Recuperação	S(%)	AMS	Recuperação	S(%)
$N = 250000$						
12	208,61	230,23 (3)	10,37	207,04	215,96 (9)	4,31
18	137,76	155,32 (2)	12,75	135,51	145,39 (6)	7,30
24	103,42	114,22 (1)	10,44	101,50	111,75 (2)	10,10
$N = 500000$						
12	840,73	933,49 (10)	11,03	835,53	860,25 (10)	2,96
18	559,98	631,27 (8)	12,73	557,91	576,87 (15)	3,40
24	417,61	466,53 (6)	11,71	413,41	436,44 (17)	5,57
$N = 1000000$						
12	3360,49	3466,27 (10)	3,15	3339,95	3379,24 (11)	1,18
18	2242,44	2436,63 (14)	8,66	2227,26	2262,76 (15)	1,59
24	1683,01	1908,78 (17)	13,41	1671,89	1707,21 (19)	2,11

5.4 Resumo

Este capítulo mostrou e analisou os resultados obtidos nos experimentos computacionais realizados para três aplicações: uma sintética e uma real do tipo Mestre-Trabalhador e uma aplicação iterativa. Os primeiros experimentos tiveram como objetivo mostrar a sobrecarga do SGA EasyGrid em tratar as falhas nos processos da aplicação. Quando estes testes foram realizados, o SGA EasyGrid só tinha suporte para tolerar falhas nestes processos. A segunda parte dos experimentos é voltada para a análise da sobrecarga dos mecanismos de tolerância a falhas na versão mais completa que inclui os processos gerenciadores GM e GS. A primeira análise realizada teve como objetivo mostrar a sobrecarga do SGA EasyGrid usando o modelo de execução 1PTask, mas sem os mecanismos de TF, em comparação com a versão tradicional equivalente, que é no modelo 1PProc usado no LAM/MPI. A segunda análise mostrou a sobrecarga causada pelos mecanismos de TF, no caso de não ocorrer falhas e no caso de falhas irrecuperáveis e recuperáveis de processos GM e GS. Os resultados mostraram que para maiores granularidades e números de tarefas, as sobrecargas para a recuperação são baixas quando comparadas ao programa equivalente sem o código de TF. Esta baixa sobrecarga é conseguida pela integração dos mecanismos de TF com a camada de escalonamento dinâmico, que esconde a sobrecarga gerada. Por fim, a terceira parte dos experimentos mostram uma baixa sobrecarga do mecanismo de TF para a realização do *checkpoint*, no nível da aplicação, do processo GG e a sua recuperação, em caso de falha. Além disso, foi mostrado que a sobrecarga total do mecanismo de TF do SGA EasyGrid é menor do que a produzida pelo sistema de *checkpoint* do BLCR.

Capítulo 6

Conclusões e Trabalhos Futuros

As grades computacionais são uma importante e poderosa plataforma computacional para aplicações científicas e comerciais de larga escala. Uma das metas das grades computacionais é agrupar conjuntos de recursos distribuídos, heterogêneos e compartilhados para fornecer poder computacional necessário para estas aplicações distribuídas e/ou paralelas. Contudo, existem desafios na exploração do desempenho de tais recursos, devido principalmente as características dinâmicas e instáveis de tal ambiente. Projetar aplicações eficientes para serem executadas em máquinas paralelas tradicionais já é difícil, consequentemente, para serem executadas em ambientes grades computacionais é uma tarefa ainda mais árdua e altamente exposta a erros.

As aplicações já existentes e as desenvolvidas atualmente não conseguem sozinhas tratar todos os desafios oriundos de um ambiente grade. O Sistema de Gerenciamento de Aplicações EasyGrid é um *middleware* projetado para habilitar de forma transparente aplicações MPI para executar em grades computacionais. Este trabalho mostra um estudo de caso usando LAM/MPI padrão e apresenta uma estratégia para fornecer dependabilidade e autorrecuperação para aplicações MPI desenvolvidas para *clusters*, onde heterogeneidade e falhas geralmente não são consideradas. A estratégia de tolerância a falhas apresentada é integrada com outros *self*-* mecanismos do SGA EasyGrid, em particular automonitoramento (*self-monitoring*), autoajuste (*self-adjusting*) e auto-otimização (*self-optimization*) para tornar as aplicações MPI autônomas.

O SGA EasyGrid possui dois tipos de processos: gerenciadores e da aplicação, por esta razão existem duas estratégias diferenciadas de recuperação. Para os processos da aplicação é utilizado o protocolo de recuperação por retorno baseado em *log* de mensagens. Nesta proposta não foi utilizada a estratégia comum de *checkpoints* em combinação com

o *log* de mensagens, devido a adoção do modelo de execução "**um processo por tarefa da aplicação (1PTask)**" empregado no SGA EasyGrid. Visto que os processos que falharem podem ser reiniciados desde o início da execução, já que são de curta duração e o custo da reexecução é baixo. O objetivo é evitar a necessidade de implementar sofisticados esquemas distribuídos de *checkpointing*. Além disso, a não utilização de *checkpoints* evita a preocupação de coordenar quando eles devem ser realizados e de determinar onde armazenar os *checkpoints*, dado que precisam ser gravados em um meio de armazenamento estável e este pode ser difícil de encontrar em um ambiente grade. Sendo que dependendo da quantidade de *checkpoints*, esse armazenamento precisa ser feito de forma distribuída para evitar congestionamentos quando escrevendo ou recuperando os *checkpoints*. Por todas essas razões, só foi utilizado o *log* de mensagens no protocolo de recuperação. O armazenamento das mensagens é realizado pelos gerenciadores e o reinício dos processos da aplicação que falharam se mostrou efetivo para realizar a recuperação da aplicação paralela.

Para a tolerância a falhas nos processos gerenciadores é utilizado *checkpoint* no nível da aplicação, onde as informações necessárias para restaurar o estado dos processos são armazenadas pelos gerenciadores de nível acima. As informações são atualizadas toda vez que o gerenciador envia informações para os gerenciadores de nível inferior e os processos da aplicação terminam. O SGA possui três níveis de gerenciadores: Gerenciador Global, Gerenciador do *Site* e Gerenciador da Máquina. O Gerenciador do *Site* é o responsável pela detecção, recriação e envio das informações necessárias para a recuperação dos Gerenciadores da Máquina do seu *site*. Já o Gerenciador Global é o responsável pela recuperação dos Gerenciadores do *Site* pertencentes à grade computacional. Para o caso de falha do Gerenciador Global, o portal da grade ou o *broker* que submeteu o *job* é o responsável para detectar e recriar o GG. As informações com o último estado do GG são gravadas em um arquivo pré-determinado e o GG, ao ser recriado, restaura o seu estado a partir das informações deste arquivo. Observa-se que nessa estratégia somente o GG gera um arquivo que precisa ser armazenado em um meio estável. A implementação para o tratamento de falhas no portal/*broker* está fora do escopo do trabalho, mas uma solução para este problema pode ser a criação de várias instâncias baseada em serviços de alta disponibilidade (MARCUS; STERN, 2000).

Os experimentos computacionais foram divididos em três partes: falhas nos processos da aplicação, falhas nos seus processos gerenciadores GM e GS, e por fim, falha no processo GG. Os resultados da Seção 5.2.2 mostraram uma sobrecarga na inclusão dos mecanismos de tolerância a falhas no SGA EasyGrid menor do que 3% para todas as granularidades e

quantidades de tarefas experimentadas em um ambiente sem falhas. Uma característica é que a sobrecarga diminui com o crescimento da granularidade e do número de tarefas. Vale destacar que o grau de intrusão foi menor do que 1% para os casos com mais de 1000 tarefas e granularidade de 40 segundos, onde os tempos de execução usando ou não os mecanismos de tolerância a falhas ficaram muito próximos. Esse comportamento pode ser visto inclusive nos cenários de falhas experimentados. A sobrecarga dos mecanismos de TF foi menor do que 2% em todos os cenários de falhas para o caso de 8000 tarefas com granularidade de 40s. Os experimentos realizados na Seção 5.3 mostraram que a sobrecarga total do mecanismo de TF do SGA EasyGrid é menor do que a produzida pelo sistema de *checkpoint* do BLCR, mesmo quando o SGA EasyGrid realiza um número maior de *checkpoints*. Como nos experimentos da Seção 5.2.2, a sobrecarga diminui com o crescimento da granularidade dos processos. O algoritmo *AMS ring* obteve uma sobrecarga menor do que 3% na recuperação de toda a estrutura de processos gerenciadores do SGA, falha no processo GG, para o caso de 1000000 partículas utilizando 12, 18 e 24 processadores. Em resumo, todos os resultados apresentados mostram que as estratégias dos mecanismos de tolerância a falhas adotadas tiveram um baixo impacto no desempenho da aplicação do usuário, o que se deve ao fato da integração dos mecanismos de TF com a camada de escalonamento dinâmico.

Os experimentos computacionais realizados nas Seções 5.2.2 e 5.3 utilizaram no máximo 25 máquinas. Esta configuração foi utilizada para garantir uma execução com acesso exclusivo para os recursos computacionais e, por consequência, medir com maior precisão a sobrecarga dos mecanismos de tolerância a falhas. Por esta razão e por falta de acesso a um ambiente maior, não foi utilizada uma grade com diferentes domínios administrativos. Porém, para uma análise completa seria necessária a utilização de tal ambiente, em particular, para avaliar as questões de escalabilidade. Entretanto, pela característica de diminuição da sobrecarga dos mecanismos de tolerância a falhas com o aumento do tamanho da aplicação, espera-se que com aplicações maiores e um maior número de máquinas a sobrecarga se mantenha em queda. Um fator a ser considerado para testes maiores é o consumo de memória para guardar os logs de mensagens, especialmente no caso de aplicações com comunicação intensiva e com grandes mensagens. Vale ressaltar que as mensagens guardadas são liberadas assim que seu respectivo processo da aplicação termina a sua execução. Contudo, o SGA EasyGrid possui um controle de memória que pode ser usado nestes testes para limitar a quantidade de memória usada pelos gerenciadores GG e GS. Adicionalmente, pode até ser considerado o armazenamento de parte do *log* de mensagens em arquivo. Porém, esta abordagem provavelmente acarretará em

alguma sobrecarga que deverá ser avaliada.

6.1 Trabalhos Futuros

Uma estratégia de tolerância a falhas não adotada nesta tese e que pode ser utilizada é a replicação de tarefas, onde as réplicas das tarefas são criadas e executadas junto com as tarefas originais. Esta forma de replicação pode ser empregada pela camada de escalonamento para evitar custos de comunicação e então reduzir o tamanho do escalonamento (*makespan*) da aplicação. Nesta estratégia, tarefas da aplicação são replicadas e, no caso de falha de uma tarefa, a réplica já em execução produziria os resultados esperados. Caso não aconteça nenhuma falha no ambiente, a tarefa que gerasse o resultado mais rápido enviaria os dados para as outras tarefas. Para o uso desta forma de replicação, um controle no SGA EasyGrid teria que ser desenvolvido para o gerenciamento dinâmico das tarefas, suas réplicas e suas mensagens, como em (SILVA, 2002; FREIRE, 2003). Observa-se que é necessário um gerenciamento das mensagens para evitar uma inconsistência na aplicação do usuário, onde uma tarefa poderia receber a mesma mensagem mais de uma vez pelo envio de dados duplicados das tarefas original e suas réplicas. Apesar de consumir mais recursos computacionais, a replicação de tarefas pode ser viável em um ambiente de grade computacional, já que o desempenho da aplicação com dependência entre tarefas pode se beneficiar bastante (NASCIMENTO, 1999) e se espera um número significativo de recursos disponíveis neste tipo de ambiente.

Uma outra funcionalidade que pode ser desenvolvida a partir desta tese é a detecção e o tratamento de falhas bizantinas. Segundo (LIMA; GREVE, 2009), para a identificação de falhas bizantinas, os processos corretos devem ter uma visão coerente das mensagens enviadas por cada processo e devem poder verificar se estas mensagens estão consistentes com os requisitos do algoritmo ou protocolo executado. Para a verificação dos dados gerados pelos processos da aplicação, o SGA EasyGrid pode usar a replicação ativa, onde uma determinada tarefa pode ser executada em vários recursos com diferentes níveis de confiança e os dados gerados são comparados para verificar a sua corretude. Outra abordagem é a utilização de *jobs* de testes, em que os resultados já são conhecidos, para testar os recursos remotos, como mostrado em (SARMENTA, 2001; ZHAO; LO; GAUTHIERDICKEY, 2005).

Um outro trabalho que pode ser explorado a partir desta tese está relacionado com a detecção de falhas. Nesta tese, os gerenciadores, para suspeitarem da falha de um

outro gerenciador remoto, utilizam um tempo de espera com um limite configurável. Este tempo pode ser diferente para o gerenciador Global e os gerenciadores do Site, já que o GG detecta as falhas de processos em *sites* geograficamente distribuídos, enquanto os GS's detectam as falhas de processos dentro de um *site*. Entretanto, este valor não é alterado durante a execução da aplicação do usuário. Um possível estudo é verificar formas de adaptar esse valor as condições atuais da rede, como por exemplo, usando um histórico recente da latência das mensagens oriundas daquele recurso.

Um outro trabalho futuro são os critérios para a escolha de um novo Gerenciador do *Site*, quando este tem uma falha irrecuperável. Atualmente, quando um GS não pode ser recriado no mesmo recurso, ele é reescalonado em um novo recurso seguindo a ordem na qual as máquinas de um *site* foram usadas para iniciar o ambiente de execução LAM. Um novo critério poderia ser estudado para definir uma melhor forma para a escolha do novo recurso a ser utilizado, por exemplo, a utilização de CPU do novo recurso.

Um outro ponto a ser investigado é a utilização de *checkpoints* não coordenados para as tarefas da aplicação com granularidades maiores, onde cada GM produziria o *checkpoint* independente dos outros. Entretanto, será necessário determinar o limite da granularidade dos processos para seguir a estratégia de reexecução ou utilizar a estratégia de *checkpoint*, levando em consideração, os custos de gravar os arquivos de *checkpoints* no recurso do gerenciador de nível acima ou em um meio de armazenamento estável. Caso a gravação dos arquivos de *checkpoint* seja muito custosa, uma opção é manter estes arquivos seguros somente dentro de cada *site*.

Referências

AGBARIA, A.; FRIEDMAN, R. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, v. 6, n. 3, p. 227–236, 2003.

ANDERSON, T.; LEE, P. A. *Fault Tolerance: Principles and Practice.* Englewood Cliffs, N.J.: Prentice-Hall, 1981.

ARAÚJO, A. P. F. *Paralelização Autonômica de Metaheurísticas em Ambientes de Grid.* Tese (Doutorado) — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brasil, Abril 2008.

Argonne National Laboratory. *MPICH2: High-performance and Widely Portable MPI.* 2010. Disponível em http://www.mcs.anl.gov/research/projects/mpich2. Última visita em 09 de Março de 2010.

AVIZIENIS, A. Infrastructure-based design of fault-tolerant systems: How to get high-confidence computing for all. *In the Electronic Proceedings of the IFIP International Workshop on Dependable Computing and its Applications (DCIA 1998)*, Johannesburg, South Africa, p. 51–69, January 1998.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B. *Fundamental Concepts of Dependability.* Technical Report N01145, LAAS-CNRS. Toulouse, France, April 2001.

BATCHU, R.; DANDASS, Y.; SKJELLUM, A.; BEDDHU, M. MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware. *Cluster Computing*, v. 7, n. 4, p. 303–315, 2004.

BATCHU, R.; NEELAMEGAM, J. P.; CUI, Z.; BEDDHU, M.; SKJELLUM, A.; DANDASS, Y.; APTE, M. MPI/FT^{TM}: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing. In: *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid.* Washington, DC, USA: IEEE Computer Society, 2001. p. 26–33. ISBN 0-7695-1010-8.

BOERES, C.; FONSECA, A. A.; MENDES, H. A.; MENEZES, L. T.; MOURA, N. T.; SILVA, J. A.; VIANNA, B. A.; REBELLO, V. E. F. An easygrid portal for scheduling system-aware applications on computational grids. *Concurrency and Computation: Practice and Experience*, John Wiley and Sons Ltd., Chichester, UK, v. 18, n. 6, p. 553–566, May 2006. ISSN 1532-0626.

BOERES, C.; REBELLO, V. E. F. EasyGrid: Towards a framework for the automatic grid enabling of legacy MPI applications. *Concurrency and Computation: Practice and Experience*, John Wiley and Sons Ltd, v. 16, n. 5, p. 425–432, April 2004.

BOSILCA, G.; BOUTEILLER, A.; CAPPELLO, F.; DJILALI, S.; FEDAK, G.; GERMAIN, C.; HERAULT, T.; LEMARINIER, P.; LODYGENSKY, O.; MAGNIETTE, F.; NERI, V.; SELIKHOV, A. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In: *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002. p. 1–18.

BOUTEILLER, A.; HERAULT, T.; KRAWEZIK, G.; LEMARINIER, P.; CAPPELLO, F. MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI. *International Journal of High Performance Computing Applications*, v. 20, n. 3, p. 319–333, 2006.

BROOKS, R. A. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, IEEE Press, v. 2, n. 1, p. 14–23, March 1986.

BUNTINAS, D.; COTI, C.; HERAULT, T.; LEMARINIER, P.; PILARD, L.; REZMERITA, A.; RODRIGUEZ, E.; CAPPELLO, F. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols. *Future Generation Computer Systems*, v. 24, n. 1, p. 73–84, January 2008.

CARDOSO, M. C.; COSTA, F. M. MPI support on opportunistic grids based on the InteGrade middleware. *Concurrency and Computation: Practice and Experience*, John Wiley and Sons Ltd., Chichester, UK, v. 22, n. 3, p. 343–357, 2010. ISSN 1532-0626.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for asynchronous systems (preliminary version). In: *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1991. p. 325–340. ISBN 0-89791-439-2.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, ACM, New York, NY, USA, v. 43, n. 2, p. 225–267, 1996. ISSN 0004-5411.

CHANDY, K. M.; LAMPORT, L. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, ACM, New York, NY, USA, v. 3, n. 1, p. 63–75, February 1985. ISSN 0734-2071.

DÉFAGO, X.; HAYASHIBARA, N.; KATAYAMA, T. On the design of a failure detection service for large-scale distributed systems. *Proceedings of International Symposium on Towards Peta-Bit Ultra-Networks (PBit 2003)*, Ishikawa, Japan, p. 88–95, September 2003.

DORBAND, E. N.; HEMSENDORF, M.; MERRITT, D. Systolic and hyper-systolic algorithms for the gravitational n-body problem, with an application to brownian motion. *Journal of Computational Physics*, Academic Press Professional, Inc., San Diego, CA, USA, v. 185, n. 2, p. 484–511, 2003. ISSN 0021-9991.

ELNOZAHY, M.; ALVISI, L.; WANG, Y.-M.; JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, ACM Press, New York, NY, USA, v. 34, n. 3, p. 375–408, 2002. ISSN 0360-0300.

FAGG, G. E.; BUKOVSKY, A.; DONGARRA, J. J. HARNESS and fault tolerant MPI. *Parallel Computing*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 27, n. 11, p. 1479–1495, 2001. ISSN 0167-8191.

FAGG, G. E.; DONGARRA, J. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In: Jack Dongarra and Péter Kacsuk and Norbert Podhorszki (Ed.). *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 7th European PVM/MPI Users' Group Meeting, Balatonfüred, Hungary, September 2000*. Berlin, Heidelberg: Springer-Verlag, 2000. (Lecture Notes in Computer Science, v. 1908), p. 346–353. ISBN 3-540-41010-4.

FAGG, G. E.; DONGARRA, J. J. Building and using a fault-tolerant MPI implementation. *International Journal of High Performance Computer Applications*, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 18, n. 3, p. 353–361, 2004. ISSN 1094-3420.

FAGG, G. E.; GABRIEL, E.; CHEN, Z.; ANGSKUN, T.; BOSILCA, G.; PJESIVAC-GRBOVIC, J.; DONGARRA, J. Process fault tolerance: Semantics, design and applications for high performance computing. *International Journal of High Performance Computing Applications*, v. 19, n. 4, p. 465–477, November 2005.

FERREIRA, L.; BERSTIS, V.; ARMSTRONG, J.; KENDZIERSKI, M.; NEUKOETTER, A.; TAKAGI, M.; BING, R.; AMIR, A.; MURAKAWA, R.; HERNANDEZ, O.; MAGOWAN, J.; BIEBERSTEIN, N. *Introduction to grid computing with globus*. Riverton, NJ, USA: IBM Corporation, 2003. ISBN 0-7384-9988-9.

FONSECA, A. A. da. *Análise Experimental do Controle de Fluxo de Mensagens na Execução Paralela de Aplicações em Grades Computacionais*. Dissertação (Mestrado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Abril 2010.

FOSTER, I. *Designing and Building Parallel Programs*. An Online Publishing Project of Addison-Wesley Inc., Argonne National Laboratory, and the NSF Center for Research on Parallel Computation, 1995.

FOSTER, I.; KESSELMAN, C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, v. 11, n. 2, p. 115–128, 1997.

FOSTER, I.; KESSELMAN, C. (Ed.). *The grid: blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN 1-55860-475-8.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 15, n. 3, p. 200–222, 2001. ISSN 1094-3420.

FREIRE, P. M. S. *Monitoramento para Aplicações MPI System-Aware*. Dissertação (Mestrado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Junho 2003.

FRIEDMAN, R.; AGBARIA, A. *Starfish Project*. 2010. Disponível em http://dsl.cs.technion.ac.il/projects/starfish/default.htm. Última visita em 09 de Março de 2010.

GABRIEL, E.; FAGG, G. E.; BOSILCA, G.; ANGSKUN, T.; DONGARRA, J. J.; SQUYRES, J. M.; SAHAY, V.; KAMBADUR, P.; BARRETT, B.; LUMSDAINE, A.; CASTAIN, R. H.; DANIEL, D. J.; GRAHAM, R. L.; WOODALL, T. S. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Dieter Kranzlmüller and Péter Kacsuk and Jack Dongarra (Ed.). *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004*. Berlin, Heidelberg: Springer-Verlag, 2004. (Lecture Notes in Computer Science, v. 3241), p. 97–104. ISBN 3-540-23163-3.

GÄRTNER, F. C. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 31, n. 1, p. 1–26, 1999. ISSN 0360-0300.

Globus Toolkit. *The Globus Alliance*. 2010. Disponível em http://www.globus.org. Última visita em 14 de Fevereiro de 2010.

GRAHAM, R. L.; CHOI, S.-E.; DANIEL, D. J.; DESAI, N. N.; MINNICH, R. G.; RASMUSSEN, C. E.; RISINGER, L. D.; SUKALSKI, M. W. A network-failure-tolerant message-passing system for terascale clusters. *International Journal of Parallel Programming*, Kluwer Academic Publishers, Norwell, MA, USA, v. 31, n. 4, p. 285–303, August 2003. ISSN 0885-7458.

GROPP, W.; LUSK, E. Fault tolerance in message passing interface programs. *International Journal of High Performance Computer Applications*, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 18, n. 3, p. 363–372, 2004. ISSN 1094-3420.

GROPP, W.; LUSK, E.; DOSS, N.; SKJELLUM, A. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, v. 22, n. 6, p. 789–828, September 1996.

HARGROVE, P. H.; DUELL, J. C. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series*, v. 46, p. 494–499, 2006.

HEIMERDINGER, W. L.; WEINSTOCK, C. B. *A conceptual framework for system fault tolerance*. Technical Report CMU/SEI-92-TR-33, Carnegie Mellon University. Pittsburgh, PA, USA, 1992.

HUANG, C.; ZHENG, G.; KALÉ, L. V. *Supporting Adaptivity in MPI for Dynamic Parallel Applications*. Urbana-Champaign, IL, USA, 2007.

HUANG, C.; ZHENG, G.; KUMAR, S.; KALÉ, L. V. Performance evaluation of adaptive MPI. In: *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2006. p. 12–21. ISBN 1-59593-189-9.

HURSEY, J.; MATTOX, T. I.; LUMSDAINE, A. Interconnect agnostic checkpoint/restart in Open MPI. In: *HPDC '09: Proceedings of the 18th ACM international symposium on High Performance Distributed Computing*. New York, NY, USA: ACM, 2009. p. 49–58. ISBN 978-1-60558-587-1.

HURSEY, J.; SQUYRES, J. M.; MATTOX, T. I.; LUMSDAINE, A. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In: *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2007. p. 1–8.

IBM Research. *Autonomic computing*. 2010. Disponível em http://www.research.ibm.com/autonomic. Última visita em 14 de Fevereiro de 2010.

ICL Team. *FT-MPI*. 2010. Disponível em http://icl.cs.utk.edu/ftmpi. Última visita em 07 de Março de 2010.

KALE, L. V.; KRISHNAN, S. CHARM++: a portable concurrent object oriented system based on C++. *SIGPLAN Notices*, ACM, New York, NY, USA, v. 28, n. 10, p. 91–108, 1993. ISSN 0362-1340.

KARONIS, N. T.; TOONEN, B.; FOSTER, I. MPICH-G2: a Grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, Academic Press, Inc., Orlando, FL, USA, v. 63, n. 5, p. 551–563, May 2003. ISSN 0743-7315.

KERR, J. F. R.; WYLLIE, A. H.; CURRIE, A. R. Apoptosis: a basic biological phenomenon with wide-ranging implications in tissue kinetics. *Br. J. Cancer*, v. 26, n. 4, p. 239–257, August 1972.

KOOPMAN, P. Elements of the self-healing system problem space. *Proceeding of the Workshop on Software Architecture for Dependable Systems (WADS2003), ICSE'03 International Conference on Software Engineering*, Portland, Oregon, USA, p. 31–36, May 2003.

LAM Team. *LAM/MPI Parallel Computing*. 2010. Disponível em http://www.lam-mpi.org. Última visita em 09 de Março de 2010.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, v. 21, n. 7, p. 558–565, July 1978.

LIMA, M. S. de; GREVE, F. G. P. Detectando falhas bizantinas em sistemas distribuídos dinâmicos. *Revista Brasileira de Redes de Computadores e Sistemas Distribuídos*, v. 2, n. 1, p. 9–21, Junho 2009. ISSN 1983-4217.

LOUCA, S.; NEOPHYTOU, N.; LACHANAS, A.; EVRIPIDOU, P. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters*, v. 10, n. 4, p. 371–382, 2000.

MAGHRAOUI, K. E.; DESELL, T. J.; SZYMANSKI, B. K.; VARELA, C. A. Malleable iterative MPI applications. *Concurrency and Computation: Practice and Experience*, John Wiley and Sons Ltd., Chichester, UK, v. 21, n. 3, p. 393–413, 2009. ISSN 1532-0626.

MARCUS, E.; STERN, H. *Blueprints for high availability: designing resilient distributed systems*. New York, NY, USA: John Wiley and Sons, Inc., 2000. ISBN 0-471-35601-8.

MENDES, H. de A. *HLogP: Um Modelo de Escalonamento para a Execução de Aplicações MPI em Grades Computacionais*. Dissertação (Mestrado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Novembro 2004.

MPI. *Message Passing Interface Forum: MPI 2*. 2010. Disponível em http://www.mpi-forum.org. Última visita em 09 de Março de 2010.

MPICH-G2. *MPICH-G2*. 2010. Disponível em http://www3.niu.edu/mpi. Última visita em 09 de Março de 2010.

MPICH-GF. *MPICH-GF: User Transparent, Fault Tolerant, Grid-enabled MPICH*. 2010. Disponível em http://dcslab.snu.ac.kr/projects/mpichgf/main.html. Última visita em 07 de Março de 2010.

MPICH-V. *MPICH-V: MPI Implementation for Volatile resources*. 2010. Disponível em http://mpich-v.lri.fr. Última visita em 07 de Março de 2010.

NASCIMENTO, A. P. *Aglomeração de Tarefas em Arquiteturas Paralelas com Memória Distribuída*. Dissertação (Mestrado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Outubro 1999.

NASCIMENTO, A. P. *Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais*. Tese (Doutorado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Maio 2008.

NASCIMENTO, A. P.; SENA, A. C.; BOERES, C.; REBELLO, V. E. F. Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation: Practice and Experience*, John Wiley and Sons Ltd, Chichester, UK, v. 19, n. 14, p. 1955–1974, September 2007. ISSN 1532-0626. Published Online: 14 Nov 2006.

NASCIMENTO, A. P.; SENA, A. C.; BOERES, C.; REBELLO, V. E. F. On the Feasibility of Dynamically Scheduling DAG Applications on Shared Heterogeneous Systems. In: *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009. p. 191–202. ISBN 978-3-642-03868-6.

NASCIMENTO, A. P.; SENA, A. C.; SILVA, J. A.; VIANNA, D. Q. C.; BOERES, C.; REBELLO, V. E. F. Managing the execution of large scale MPI applications on computational grids. In: *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)*. Washington, DC, USA: IEEE Computer Society, 2005. p. 69–76. ISBN 0-7695-2446-X.

NASCIMENTO, A. P.; SENA, A. C.; SILVA, J. A.; VIANNA, D. Q. C.; BOERES, C.; REBELLO, V. E. F. Autonomic application management for large scale MPI programs. *International Journal Of High Performance Computing And Networking*, Inderscience Publishers, December 2008.

NELSON, V. P. Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, IEEE Computer Society, Los Alamitos, CA, USA, v. 23, n. 7, p. 19–25, July 1990.

NEOPHYTOU, N. *MPI-FT: PORTABLE FAULT TOLERANCE SCHEME FOR MPI*. 2010. Disponível em http://fytos.com/index.php?pagename=MPI-FT. Última visita em 14 de Fevereiro de 2010.

Open MPI. *Open MPI: Open Source High Performance Computing*. 2010. Disponível em http://www.open-mpi.org. Última visita em 02 de Abril de 2010.

PARASHAR, M.; HARIRI, S. Autonomic Computing: An Overview. In: Jean-Pierre Banâtre and Pascal Fradet and Jean-Louis Giavitto and Olivier Michel (Ed.). *Unconventional Programming Paradigms, International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers.* Berlin, Heidelberg: Springer-Verlag, 2005. (Lecture Notes in Computer Science, v. 3566), p. 257–269. ISBN 3-540-27884-2.

PVM. *PVM: Parallel Virtual Machine.* 2010. Disponível em http://www.csm.ornl.gov/pvm. Última visita em 02 de Abril de 2010.

RAO, S.; ALVISI, L.; VIN, H. M. Egida: An Extensible Toolkit for Low-Overhead Fault-Tolerance. In: *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing.* Washington, DC, USA: IEEE Computer Society, 1999. p. 48–55. ISBN 0-7695-0213-X.

RAO, S.; ALVISI, L.; VIN, H. M. The Cost of Recovery in Message Logging Protocols. *IEEE Transactions on Knowledge and Data Engineering*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 12, n. 2, p. 160–173, 2000. ISSN 1041-4347.

REED, D. A.; LU, C. da; MENDES, C. L. Reliability challenges in large systems. *Future Generation Computer Systems*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, v. 22, n. 3, p. 293–302, February 2006. ISSN 0167-739X.

RODRIGUES, H. B. *Grid SA: Uma Sociedade Autônoma.* Dissertação (Mestrado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Dezembro 2009.

SALTZER, J. H.; REED, D. P.; CLARK, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, ACM, New York, NY, USA, v. 2, n. 4, p. 277–288, 1984. ISSN 0734-2071.

SANKARAN, S.; SQUYRES, J. M.; BARRETT, B.; LUMSDAINE, A. *Checkpoint-Restart Support System Services Interface (SSI) Modules for LAM/MPI.* Bloomington, IN, USA, 2003.

SANKARAN, S.; SQUYRES, J. M.; BARRETT, B.; LUMSDAINE, A.; DUELL, J.; HARGROVE, P.; ROMAN, E. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, v. 19, n. 4, p. 479–493, Winter 2005.

SARDIÑA, I. M. *Escalonamento Estático de Tarefas Bi-objetivo e Tolerante a Falhas para Sistemas Distribuídos.* Tese (Doutorado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Junho 2010.

SARDIÑA, I. M.; BOERES, C.; DRUMMOND, L. M. A. Escalonamento tolerante a falhas na recuperação de aplicações em MPI. In: *24 Simpósio Brasileiro de Redes de Computadores (SBRC). Workshop de Grids Computacionais e Aplicações (WCGA).* Porto Alegre, RS, Brasil: Sociedade Brasileira de Computação, 2006. p. 27–38.

SARMENTA, L. F. G. Sabotage-tolerance mechanisms for volunteer computing systems. In: *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing*

and the Grid. Washington, DC, USA: IEEE Computer Society, 2001. p. 337–346. ISBN 0-7695-1010-8.

SCHNEIDER, F. B. What good are models and what models are good? ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, p. 17–26, 1993.

SENA, A. C. *Um Modelo Alternativo para Execução Eficiente de Aplicações Paralelas MPI nas Grades Computacionais*. Tese (Doutorado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Novembro 2008.

SENA, A. C.; NASCIMENTO, A. P.; BOERES, C.; REBELLO, V. E. F. Easygrid enabling of iterative tightly-coupled parallel MPI applications. *International Symposium on Parallel and Distributed Processing with Applications*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 199–206, 2008.

SENA, A. C.; NASCIMENTO, A. P.; SILVA, J. A.; VIANNA, D. Q. C.; BOERES, C.; REBELLO, V. E. F. On the advantages of an alternative MPI execution model for grids. In: *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), 14-17 May 2007, Rio de Janeiro, Brazil*. Los Alamitos, CA, USA: IEEE Computer Society, 2007. p. 575–582.

SILVA, J. A. *STATS: Uma Ferramenta para Escalonamento Estático de Tarefas em Programas MPI*. Dissertação (Mestrado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Novembro 2002.

SILVA, J. A.; REBELLO, V. E. F. Low cost self-healing in MPI applications. In: CAPPELLO, F.; HÉRAULT, T.; DONGARRA, J. (Ed.). *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007*. Berlin, Heidelberg: Springer-Verlag, 2007. (Lecture Notes in Computer Science, v. 4757), p. 144–152. ISBN 978-3-540-75415-2.

SOUTO, H. P. A.; SILVEIRA FILHO, O. T.; MOYNE, C.; DIDIERJEAN, S. Thermal dispersion in porous media: Computations by the random walk method. *Journal of Computational and Applied Mathematics*, v. 21, n. 2, p. 513–544, 2002.

STELLING, P.; FOSTER, I.; KESSELMAN, C.; LEE, C.; LASZEWSKI, G. von. A fault detection service for wide area distributed computations. *Cluster Computing*, Kluwer Academic Publishers, Hingham, MA, USA, v. 2, n. 2, p. 117–128, 1999. ISSN 1386-7857.

STELLNER, G. Cocheck: Checkpointing and process migration for mpi. In: *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*. Washington, DC, USA: IEEE Computer Society, 1996. p. 526–531. ISBN 0-8186-7255-2.

STELLNER, G. *CoCheck (Consistent Checkpoints)*. 2010. Disponível em http://www.lrr.in.tum.de/Par/tools/Projects.Old/CoCheck.html. Última visita em 07 de Março de 2010.

STERRITT, R.; BUSTARD, D. Autonomic computing-a means of achieving dependability. In: *Proceedings of 10th IEEE International Conference on the Engineering of Computer Based Systems (ECBS'03)*. Huntsville, Alabama, USA: IEEE Computer Society, 2003. p. 247–251.

STERRITT, R.; BUSTARD, D. Towards an autonomic computing environment. In: *DEXA'03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*. Washington, DC, USA: IEEE Computer Society, 2003. p. 694–698. ISBN 0-7695-1993-8.

STERRITT, R.; PARASHAR, M.; TIANFIELD, H.; UNLAND, R. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, v. 19, n. 3, p. 181–187, 2005.

STRANDTMAN, S. *Egida: Lightweight Fault Tolerance for Distributed Systems*. 2010. Disponível em http://www.cs.utexas.edu/users/dmcl/projects/egida. Última visita em 07 de Março de 2010.

TANENBAUM, A. S.; STEEN, M. V. *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN 0130888931.

TREASTER, M. *A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems*. 2005. ACM Computing Research Repository (CoRR), (cs.DC/ 0501002), January 2005.

VIANNA, D. Q. C. *Um Sistema de Gerenciamento de Aplicações MPI para Ambientes Grid*. Dissertação (Mestrado) — Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Outubro 2005.

WEBER, T. S. *Um roteiro para exploração dos conceitos básicos de tolerância a falhas*. 2002. Disponível em http://www.inf.ufrgs.br/~taisy/disciplinas/textos/Dependabilidade.pdf. Última visita em 15 de Fevereiro de 2010.

WOO, N.; YEOM, H. Y.; PARK, T. MPICH-GF: Transparent Checkpointing and Rollback-Recovery for GRID-enabled MPI Processes. *IEICE TRANSACTIONS on Information and Systems*, v. 87, p. 1820–1828, 2004.

YU, J.; BUYYA, R. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, v. 3, n. 3-4, p. 171–200, 2005.

ZHAO, S.; LO, V.; GAUTHIERDICKEY, C. Result verification and trust-based scheduling in peer-to-peer grids. In: *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*. Washington, DC, USA: IEEE Computer Society, 2005. p. 31–38. ISBN 0-7695-2376-5.

ZHENG, G.; HUANG, C.; KALÉ, L. V. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *SIGOPS Operating Systems Review*, ACM, New York, NY, USA, v. 40, n. 2, p. 90–99, 2006. ISSN 0163-5980.

ZIV, A.; BRUCK, J. Checkpointing in parallel and distributed systems. In: *Parallel and Distributed Computing Handbook*. New York, NY, USA: McGraw-Hill, 1996. chapter 10, p. 274–302.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo