Universidade Federal do Paraná

Edson Tavares de Camargo

Tolerância a Falhas em Sistemas MPI com Grupos Dinâmicos de Processos Recomendados e Registro de Mensagens Distribuído Baseado em Paxos

Curitiba PR
2017

Edson Tavares de Camargo

Tolerância a Falhas em Sistemas MPI com Grupos Dinâmicos de Processos Recomendados e Registro de Mensagens Distribuído Baseado em Paxos

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Elias P. Duarte Jr.

Curitiba PR
2017

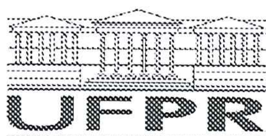
C172t Camargo, Edson Tavares de
Tolerância a falhas em sistemas MPI com grupos dinâmicos de processos recomendados e registro de mensagens distribuído baseado em Paxos / Edson Tavares de Camargo. – Curitiba, 2017.
110 f. : il. color. ; 30 cm.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2017.

Orientador: Elias P. Duarte Jr.

1. Ciência da computação. 2. Registro de mensagens. 2. DGRP. 3. Paxos paralelo. I. Universidade Federal do Paraná. II. Duarte Jr, Elias P. III. Título.

CDD: 004.6



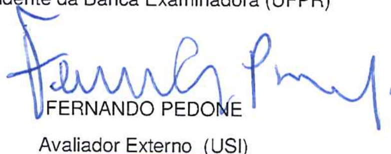
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós-Graduação INFORMÁTICA

TERMO DE APROVAÇÃO

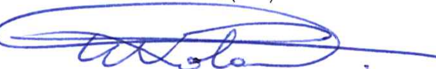
Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **EDSON TAVARES DE CAMARGO** intitulada: **Tolerância a Falhas em Sistemas MPI com Grupos Dinâmicos de Processos Recomendados e Registro de Mensagens Distribuído Baseado em Paxos**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVAÇÃO.

Curitiba, 11 de Maio de 2017.

ELIAS PROCÓPIO DUARTE JUNIOR
Presidente da Banca Examinadora (UFPR)



FERNANDO PEDONE
Avaliador Externo (USI)



WAGNER MACHADO NUNAN ZOLA
Avaliador Externo (UFPR)

ALDELIR FERNANDO LUIZ
Avaliador Externo (IFC)

DORGIVAL OLAVO GUEDES NETO
Avaliador Externo (UFMG)

Aos meus pais, Carlito (in memoriam) e Maria, à minha esposa Heloísa e ao meu filho Pedro.

"A fé e a razão constituem como que as duas asas pelas quais o espírito humano se eleva para a contemplação da verdade."

– João Paulo II – Carta Encíclica Fides et Ratio

Agradecimentos

Foram muitos os colaboradores diretos e indiretos envolvidos neste trabalho. Nesses quatro anos diversas pessoas me ajudaram pelo simples fato de me incentivarem, entenderem minha ausência ou torcerem por mim em silêncio. Os nomes descritos aqui neste espaço refletem os colaborares mais diretos mas, sem dúvida, não todos.

Meu mais sincero agradecimento, em primeiro lugar, ao meu orientador Prof. Elias. Só posso dizer que foi excepcional trabalhar contigo! Desde o nosso primeiro contato em 2012 no SBRC e até hoje, eu o admiro pela forma como ele lida e trata os seus alunos e orientandos. É um excelente professor e orientador. As inúmeras dúvidas e incertezas que surgiam ao longo do trabalho foram suavemente respondidas pelo Prof. Elias. Vou sentir saudades desse período qualificado dedicado à pesquisa e das nossas reuniões de orientação. E dos chopes também, é claro!

Este doutorado também teve a excelente e valorosa contribuição do Prof. Fernando Pedone. Eu jamais imaginava que o meu doutorado sanduíche pudesse ser tão precioso. Meu obrigado ao Prof. Fernando por me receber na USI, em Lugano, e me orientar durante o doutorado sanduíche. Além do enorme aprendizado adquirido no grupo de pesquisa do Prof. Fernando, Lugano ficará marcada pelas excelentes amizades. Agradeço os meus colegas de laboratório em Lugano pelo auxílio na minha pesquisa e pelas excelentes horas de convivência: Paulo Coelho, Odorico, Daniele Sciascia, Tu Dang, Long Le, Leandro, Eduardo, Samuel e Henrique. Os *hikings*, cafés, chopes, comemorações são inesquecíveis. Muito obrigado, meu caros!

Também agradeço aos meus colegas do PPGInf e do LaRSis. As trocas de ideias e os momentos de descontração foram essenciais. Obrigado ao Guilherme Galante, Luiz Rodrigues, Edmar, Rogério Tuchetti, Célio, Weyne, Giovanni, Daniel... Ao PPGInf o meu agradecimentos aos coordenadores do programa, aos professores e à secretaria representada pelos servidores Rafael, Roberto e Jucélia.

Agradeço aos membros da banca, professores Dorgival Guedes, Jaime Cohen, Fabiano Silva , Aldelir Luiz e Wagner Zola pelas críticas construtivas e pelas sugestões.

A minha família foi meu porto seguro durante essa jornada. Obrigado aos meus pais Maria e Carlito. O meu pai viu a conclusão do meu trabalho em um lugar especial no céu. Obrigado a ambos por toda a educação recebida, os exemplos, incentivos. Vocês são exemplo de que a pobreza e a vida simples, por mais restrições que imponham, não são limitações. Obrigado a minha esposa pelo apoio, não tenho palavras para te agradecer, minha linda!

Agradeço à UTFPR e aos meu colegas de trabalho por assumirem as minhas aulas para que eu pudesse me dedicar exclusivamente ao doutorado!

Ainda agradeço a Capes pelo apoio financeiro durante o doutorado sanduíche através do programa PDSE e do programa de Demanda Social de Bolsas.

Resumo

Os sistemas HPC (*High-Performance Computing*) são geralmente empregados para executar aplicações de longa duração, incluindo, por exemplo, simulações científicas e industriais complexas. Construir sistemas HPC tolerante a falhas permanece um desafio à medida que o tamanho desses sistemas aumenta. Esta tese de doutorado apresenta duas estratégias de tolerância a falhas para sistemas HPC baseados em MPI. A primeira contribuição apresenta uma solução para lidar com a variabilidade de desempenho que afeta negativamente ou inviabiliza a execução das aplicações HPC. Este é o caso dos *clusters* compartilhados onde um nodo computacional pode se tornar muito lento e comprometer a execução de toda a aplicação. Esta tese propõe um novo modelo de diagnóstico em nível de sistema onde os processos executam testes entre si a fim de determinar se são *recomendados* ou *não-recomendados*. Os processos classificados como recomendados formam um grupo dinâmico, chamado de DGRP (*Dynamic Group of Recommended Processes*), e são responsáveis por executar a aplicação. Os processos testados como não-recomendados são removidos do DGRP. Um processo pode reingressar ao DGRP após uma rodada de consenso executada pelos processos do DGRP. O modelo foi implementado e empregado para monitorar os processos em um *cluster* compartilhado multiusuário. No estudo de caso apresentado, os processos do DGRP executam o algoritmo de ordenação paralela *Hyperquicksort*. O *Hyperquicksort* é implementado e adaptado para se reconfigurar em tempo de execução a fim de suportar até $n-1$ processos não-recomendados (em um sistema com n processos). Os resultados obtidos demonstram a sua eficiência. A segunda contribuição desta tese se insere na técnica de *rollback-recovery* na sua variante chamada de registro de mensagens. O registro de mensagens não requer a sincronização dos processos para salvar o estado da aplicação e evita que todos os processos reiniciem a partir do último estado salvo. No entanto, a maioria dos protocolos de registro de mensagens conta com um componente centralizado e que não tolera falhas, chamado de *event logger*, para armazenar as informações de recuperação, isto é, os determinantes. Esta tese de doutorado propõe o primeiro *event logger* distribuído e tolerante a falhas para os protocolos de registro de mensagens. Duas implementações baseadas no algoritmo de consenso Paxos, chamadas de Paxos Clássico e Paxos Paralelo, foram realizadas para o *event logger*. Um protocolo pessimista de registro de mensagens é construído e implementado para interagir com o *event logger* proposto e realizar a recuperação automática das aplicações MPI. O desempenho dos *event loggers* é avaliado perante a aplicação AMG (*Algebraic MultiGrid*) e as aplicações do *NAS Parallel benchmark*. A recuperação é avaliada através do algoritmo paralelo de Gusfield e a aplicação AMG. Resultados demonstram que o *event logger* baseado em Paxos Paralelo tem desempenho comparável ou superior ao da abordagem centralizada e que o protocolo proposto realiza a recuperação da aplicação eficientemente.

Palavras-chave: Tolerância a Falhas em MPI, DGRP, Registro de Mensagens, Paxos Paralelo.

Abstract

HPC systems are employed to execute long-running applications including, for example, complex industrial and scientific simulations. Building robust, fault-tolerant HPC systems remains a challenge as the size of the system grows. This doctoral thesis presents two fault-tolerant strategies for HPC systems based on MPI. Our first contribution presents a solution to deal with the performance variation of HPC system processes that negatively affect or even prevent the execution of HPC applications. This is the case in shared clusters in which a single node can become too slow and can thus compromise the entire application execution. This thesis proposes a new system-level diagnosis model in which processes execute tests among themselves in order to determine whether they are *recommended* or *non-recommended*. Processes classified as recommended form a Dynamic Group of Recommended Processes (DGRP), which is responsible for running the application. A process can rejoin the DGRP after a round of consensus executed by the DGRP processes. The model was implemented and used to monitor processes in a shared multi-user cluster. In the case study presented, the DGRP processes execute the parallel sorting algorithm Hyperquicksort. Hyperquicksort is implemented and adapted to reconfigure itself at runtime in order to proceed even if up to $N - 1$ processes become non-recommended (N is the total number of processes). Results are presented showing that the strategy is efficient. The second contribution of this thesis is in the field of the rollback-recovery technique in its variant based on message logging. Message logging does not require all processes to coordinate in order to save their states during normal execution. Neither does it require to restart all processes from the last saved states after a single process fails. However, most existing message logging protocols rely on a centralized entity which does not tolerate failures, called event logger, which stores recovery information called determinants. This thesis proposes, to the best of our knowledge, the first distributed and fault-tolerant event logger. Two implementations are presented based on the Paxos consensus algorithm, called Classic Paxos and Parallel Paxos. A pessimistic message logging protocol is built and implemented based on the proposed event logger to perform automatic recovery of MPI applications after failures. We evaluate the performance of the event logger using both the AMG (Algebraic MultiGrid) application and NAS Parallel benchmark applications. Application recovery is evaluated in two case studies based on Gusfield's parallel cut tree algorithm and the AMG application. Results show that the event logger based on Parallel Paxos performs as well as or better than a centralized event logger and that the proposed recovery protocol is also efficient.

Keywords: Fault Tolerance in MPI, DGRP, Message Logging, Parallel Paxos.

Sumário

1 Introdução **14**

2 Tolerância a Falhas em MPI **17**
2.1 Tolerância a Falhas: Definições Básicas . . . 17
2.1.1 Detectores de Falhas . . . 18
2.1.2 Paxos . . . 19
2.1.3 *Rollback-Recovery* . . . 21
2.2 Tolerância a Falhas em MPI . . . 24
2.2.1 FT-MPI . . . 25
2.2.2 MPI/FT . . . 25
2.2.3 Programas MPI Tolerante a Falhas . . . 26
2.2.4 NR-MPI . . . 27
2.2.5 Grupo de Trabalho de Tolerância a Falhas do MPI-Fórum . . . 27
2.2.6 Detecção de Falhas em MPI . . . 30
2.2.7 Consenso Distribuído para MPI . . . 31
2.2.8 *Rollback-Recovery* em MPI . . . 32
2.2.9 Replicação Máquina de Estado Aplicada para MPI . . . 35
2.2.10 Tolerância a Falhas Codificada no Algoritmo da Aplicação . . . 36
2.2.11 Detecção de Falhas de Desempenho . . . 38
2.3 Conclusão . . . 38

3 Diagnóstico em Nível de Sistema **42**
3.1 O Modelo PMC . . . 42
3.2 O Modelo BGM . . . 44
3.3 Modelo Adaptativo e Distribuído . . . 44
3.4 Modelo Hierárquico . . . 47
3.5 Outros Modelos de Diagnóstico . . . 51
3.6 Conclusão . . . 52

4 O Grupo Dinâmico de Processos Recomendados **53**
4.1 DGRP: Grupo Dinâmico de Processos Recomendados . . . 54
4.1.1 Modelo de Sistema . . . 56
4.1.2 Implementação do DGRP . . . 57
4.2 Estudos de Caso: o Algoritmo *Hyperquicksort* . . . 58
4.2.1 Algoritmo *Hyperquicksort* . . . 59
4.2.2 *Hyperquicksort* sobre o DGRP . . . 61
4.3 Resultados . . . 62
4.3.1 Monitoramento através do DGRP . . . 63
4.3.2 Desempenho do *Hyperquicksort* sobre o DGRP e ULFM . . . 67

4.4 Conclusão 71

5 Um Protocolo Pessimista para Registro de Mensagens Baseado em um *Event Logger* Distribuído e Tolerante a Falhas **72**

5.1 Um Protocolo para Registro de Mensagens Baseado em Consenso 73

5.1.1 Modelo de Sistema 73

5.1.2 Descrição do Protocolo 74

5.1.3 O Serviço Proposto de *Event Logger* Baseado em Consenso 75

5.2 Implementação 78

5.3 Resultados Experimentais 82

5.3.1 Os *Event Loggers* 82

5.3.2 AMG 83

5.3.3 LU e MG 84

5.3.4 Gusfield 86

5.3.5 Recuperação do Algoritmo de Gusfield e da Aplicação AMG 87

5.4 Conclusão 90

6 Conclusão **91**

Referências Bibliográficas **93**

A Anexo **104**

A.1 Difusão Atômica 104

A.1.1 Implementação de Difusão Atômica sobre o DCSP 105

B Publicações **109**

B.1 Trabalhos Publicados no Âmbito da Tese 109

B.2 Trabalhos Publicados no Âmbito do Grupos de Pesquisa 110

Lista de Figuras

2.1 Paxos em duas fases 20
2.2 *Checkpoint* não coordenado e uma linha de recuperação. 22
2.3 Detecção de falhas ULFM - Processo B detecta a falha em A. 29

3.1 Um sistema S composto por cinco unidades [139]. 43
3.2 Exemplo de um Sistema S e o conjunto de testes $T(S)$ [11]. 46
3.3 Hipercubo de 3 dimensões. 48
3.4 Sistema com 8 nodos organizados em *clusters* de tamanho progressivo. 48
3.5 Organização hierárquica dos testes no *Hi-ADSD*. 49
3.6 Teste executados pelos nodo 0 em um sistema com oito nodos [57]. 50
3.7 Estratégia de teste do Hi-ADSD e do Vcube sob o ponto de vista do nodo 0 na terceira rodada de testes. 51

4.1 Variação de desempenho para aproximação do π - único nodo com 16 núcleos. 55
4.2 DGRP formado pelos processos 0, 2, 5 e 7. 55
4.3 DGRP formado pelos processos 2, 5, 6 e 7. 55
4.4 Uma rodada de computação encerra em uma barreira. 56
4.5 Arquitetura proposta para o sistema. 59
4.6 Algoritmo paralelo *Hyperquicksort* com 8 processos 61
4.7 Processo 1 testa o processo 0. 63
4.8 Processo 11 testa o processo 2. 64
4.9 Processo 6 testa o processo 2. 65
4.10 Processo 0 testa o processo 11. 66
4.11 Processo 6 testa o processo 2. 66
4.12 Zeta para o processo 2. 68
4.13 Variação na composição do DGRP. 68
4.14 Desempenho de 16 processos ordenando 1.024 x 10^6 números inteiros. 69
4.15 *Hyperquicksort* sobre o modelo proposto e a estratégia ULFM. 70

5.1 *Checkpoint* não coordenado e um linha de recuperação. 74
5.2 Três implementações de um *event logger*. 75
5.3 Recuperação do comunicador MPI e lançamento de um novo processo [75]. . . 79
5.4 Vazão e latência para três as abordagens de *event loggers*. 83
5.5 Tempo de execução e vazão da aplicação AMG. 84
5.6 Tempo de execução e vazão da aplicação LU classe C. 85
5.7 Tempo de execução e vazão da aplicação LU classe D. 86
5.8 Tempo de execução e vazão da aplicação MG classe C. 87
5.9 Tempo de execução e vazão da aplicação MG classe D. 87

5.10 Tempo de execução do algoritmo de Gusfield usando o Paxos Paralelo com diferentes configurações. 88
5.11 Recuperação do algoritmo Gusfield. 88
5.12 Recuperação da aplicação AMG. 89

A.1 Arquitetura do sistema implementado. 105
A.2 Difusão atômica com 32 nodos, 2 nodos instáveis e k = 5. 106
A.3 Difusão atômica baseada em consenso. 107
A.4 Comparação das abordagens de difusão atômica. 107

Lista de Tabelas

2.1 Tolerância a falhas em MPI categorizada em cinco grupos. 41

3.1 Vetor de testes no nodo n_2 46
3.2 $C_{i,s}$ para um sistema com 8 nodos. 49

4.1 Carga no *cluster* e resultado dos testes. 67

5.1 Diferentes implementações de um *event logger*. Valores para o cenário de melhor caso com um quorum de *acceptors*/réplicas; n: número de processos da aplicação; f: número de falhas toleradas pelos *acceptors*. 78
5.2 Comportamento das primitivas MPI durante a recuperação. 81
5.3 Porcentagem de eventos não-determinísticos nos *benchmarks* empregados. . . . 83
5.4 Sobrecarga das implementações de *event loggers* Centralizado, Paxos Clássico e Paxos Paralelo no *kernel* MG classes C e D. 86

Capítulo 1

Introdução

Os sistemas de computação de alto desempenho (*High-Performance Computing* - HPC) são empregados para executar aplicações que requerem o uso intensivo de computação, incluindo, por exemplo, simulações científicas e industriais complexas [158]. Em geral, sistemas HPC executam aplicações de longa duração e que potencialmente lidam com um volume gigantesco de dados. Nesses sistemas, um dos maiores desafios diz respeito à confiabilidade (*reliability*) [60, 153, 151], ou seja, a capacidade de oferecer serviços corretos ininterruptamente [5]. Estima-se que para os sistemas HPC de grande escala o tempo médio entre as falhas (*Mean Time Between Failures* - MTBF) é de poucas horas [59, 60, 127]. Por exemplo, o sistema *petascale* Blue Waters [133] possui um MTBF médio de 4,2 horas [51]. Os futuros sistemas *exascale*, que atingem 10^{18} operações de ponto flutuante por segundo, e poderão reunir até muitos milhões de núcleos de processamento, executando até um bilhão de *threads*, devem apresentar um MTBF ainda menor e apresentar vários tipos de falhas [9, 37]. Nesses sistemas, projetar mecanismos de tolerância a falhas efetivos e permitir que as aplicações completem adequadamente as suas execuções é uma tarefa árdua.

O MPI [129], o padrão de *facto* para desenvolvimento de aplicações paralelas e distribuídas baseado no paradigma de troca de mensagens [62], atualmente não inclui a tolerância a falhas em sua especificação. O padrão parte do princípio que as aplicações MPI executam em uma infraestrutura computacional confiável. Consequentemente, novas abordagens de tolerância a falhas e mesmo o aprimoramento de técnicas e padrões existentes no contexto de sistemas HPC vêm sendo propostos. Destacam-se as seguintes ações: os esforços para adicionar novas primitivas e uma semântica para lidar com as falhas, e recuperação, no MPI [62, 80, 91, 16, 17]; as novas abordagens e otimizações na técnica de *rollback-recovery* [158, 127, 59, 28]; a aplicação de técnicas que fazem uso das propriedades matemáticas da aplicação, como a técnica ABFT (*Algorithm-Based Fault Tolerant*) [48, 53, 90, 164]; a detecção e correção de erros antes que levem à reinicialização do sistema [66] e; o desenvolvimento de abordagens de alta disponibilidade através da técnica de replicação máquina de estado [65, 162]. Além desses esforços, destacam-se ainda as soluções que lidam com “falhas de desempenho”, termo introduzido em [78] para descrever anomalias de desempenho que prejudicam a operação das aplicações HPC tanto quanto falhas: por exemplo, um processador pode reduzir a frequência de operação de um núcleo quando a temperatura atingir um limiar de segurança, ou mesmo para manter o gasto energético do sistema dentro do orçamento disponível. A variabilidade no desempenho de nodos computacionais também é observada nos aglomerados de computadores (*clusters*) compartilhados.

Esta tese de doutorado apresenta duas estratégias de tolerância a falhas em sistemas HPC baseados em MPI. A primeira é uma estratégia para identificar e manter um grupo dinâmico

de processos recomendados, também chamado de DGRP (*Dynamic Group of Recommended Processes*). A segunda estratégia é um protocolo de *rollback-recovery* baseado em registro de mensagens pessimista que conta com um *event logger* distribuído e tolerante a falhas.

O DGRP se apoia em um novo modelo de diagnóstico em nível de sistema [126, 56] que é baseado em testes imperfeitos [33]. Processos executam testes entre si para determinar se processadores (ou núcleos) sobre os quais executam são recomendados ou não-recomendados. Em outras palavras, um procedimento de testes é definido para "medir" se o comportamento de um determinado processo é "bom o suficiente" para executar determinada aplicação. Um processo que não passa nos testes é considerado *não-recomendado* pelos demais. Por outro lado, um processo é definido como *recomendado* quando funciona corretamente, isto é, apresenta o comportamento esperado. Vale a pena notar que o comportamento de um processo pode variar durante a sua execução. Um processo pode estar recomendado por um curto período de tempo devido, por exemplo, a um pico de carga. Tal processo pode não estar apto à executar a aplicação durante um intervalo de tempo, mas posteriormente pode reverter o seu estado e reingressar no sistema. No entanto, outros processos podem se manter não-recomendados durante a maior parte do tempo. Nesse caso, esses processos deveriam ser removidos do sistema assim que possível. Um processo compõe o DGRP se não foi testado como não-recomendado por qualquer outro processo no DGRP. Os processos testados como não-recomendados são assim removidos do DGRP. Um processo que não está no DGRP porém é testado como estável por ζ testes consecutivos podem reingressar ao DGRP após uma rodada de consenso executada pelos processos do DGRP. No modelo proposto, a aplicação executa em rodadas de computação e cada nova rodada tem a sua disposição uma nova visão do DGRP.

O DGRP é implementado em MPI e aplicado para monitorar os processos de um *cluster* compartilhado multiusuário e empregado para executar o algoritmo de ordenação paralela *Hyperquicksort*. O *Hyperquicksort* organiza os processos em um hipercubo virtual. Uma nova versão do algoritmo *Hyperquicksort* foi proposta onde os processos se reconfiguraram em tempo de execução para suportar até $n - 1$ processos não-recomendados, onde n é o total de processos que executam a aplicação. Os resultados obtidos são apresentados e mostram a eficiência da ordenação e a viabilidade do DGRP.

A segunda contribuição desta tese de doutorado se insere no contexto da técnica de tolerância a falhas *rollback-recovery*. *Rollback-recovery*, também conhecido como *checkpoint-restart*, é a abordagem de tolerância a falhas tradicionalmente aplicada em sistemas HPC [59]. Grosso modo, no *rollback-recovery* os processos salvam informações de recuperação em memória não-volátil durante a sua execução sem-falhas. Após a ocorrência de uma falha e de posse das informações de recuperação, a aplicação reinicia a sua computação a partir de um estado anterior [61]. Na abordagem de *rollback-recovery* conhecida como coordenada os processos se sincronizam para realizar o *checkpoint* e, perante a falha de um único processo, todos os processos reiniciam a partir do último *checkpoint*. Um dos problemas da abordagem coordenada é o seu alto custo. Estima-se que o *rollback-recovery* coordenado não deve ser efetivo frente a um MTBF cada vez menor [37, 59]. É possível que a aplicação passe mais tempo se recuperando de falhas do que executando computação útil. Uma variante promissora de *rollback-recovery* capaz de lidar com um MTBF menor é a abordagem baseada em registro de mensagens. No registro de mensagens a sincronização dos processos não é obrigatória e somente o processo que falha é reiniciado. Nos protocolos de registro de mensagens, os eventos não-determinísticos que um processo executa são identificados e a informação necessária para reaplicar cada evento é registrada em tuplas chamadas de determinantes. Os determinantes são armazenados de forma confiável em um componente chamado de *event logger*. No entanto, o *event logger* é geralmente implementado de forma centralizada ou sem tolerar falhas [29, 147, 24]. Esta tese de doutorado

propõe o primeiro *event logger* distribuído e tolerante a falhas baseado em consenso para os protocolos de registro de mensagens.

O *event logger* distribuído e tolerante a falhas proposto tem desempenho comparável ou superior à abordagem centralizada. Um protocolo pessimista de registro de mensagens é construído em MPI para interagir com o *event logger* proposto e realizar a recuperação automática da aplicação. Em particular, o *event logger* replicado não requer recursos extras (nodos físicos) em comparação a um *event logger* centralizado. Os *event loggers* replicados podem ser hospedados nos mesmos nodos dos processos da aplicação. Além disso, o *event logger* distribuído pode tolerar um número configurável de falhas. Quando configurado para suportar uma única falha, o *event logger* distribuído requer o mesmo número de mensagens e de passos de comunicação que um *event logger* centralizado para armazenar um determinante.

Duas implementações do *event logger* proposto foram realizadas. Ambas são baseadas no algoritmo Paxos [113]. A primeira implementação se apoia em uma configuração tradicional do Paxos, chamadas de Paxos Clássico. A segunda configuração é chamada de Paxos Paralelo. As duas implementações são comparadas com uma implementação de um *event logger* centralizado. Um protocolo pessimista de registro de mensagens é implementado em MPI para interagir com o *event logger* proposto. Entre as principais características do protocolo proposto estão as seguintes: a distinção entre eventos determinísticos e não-determinísticos em MPI; o emprego da abordagem de *checkpoint* não-coordenado; e a recuperação automática da aplicação perante falhas de processos. Os *event loggers* foram avaliados usando as aplicações LU e MG do *NAS Parallel Benchmark*, a aplicação AMG (*Algebraic MultiGrid*) e o algoritmo paralelo de Gusfield. O algoritmo de Gusfield e aplicação AMG foram empregadas para avaliar a recuperação. Resultados demonstram que o *event logger* baseado na abordagem Paxos Paralelo tem desempenho comparável ou superior ao *event logger* centralizado e que o protocolo proposto realiza a recuperação da aplicação eficientemente.

O restante do texto está organizado nos seguintes capítulos. O Capítulo 2 descreve as definições básicas, a tolerância a falhas em sistemas MPI e as principais abordagens de tolerância a falhas empregadas em sistemas HPC baseados em MPI. O Capítulo 3 apresenta uma visão geral da teoria de diagnóstico em nível de sistemas e os principais trabalhos da área. O Capítulo 4 apresenta o grupo dinâmico de processos recomendados, o modelo de diagnóstico baseado em testes imperfeitos, o estudo de caso desenvolvido e os resultados obtidos. O Capítulo 5 apresenta o protocolo pessimista de registro de mensagens construído sobre um *event logger* baseado em consenso assim como a sua implementação e os resultados obtidos. As conclusões estão no Capítulo 6.

Capítulo 2

Tolerância a Falhas em MPI

O padrão MPI (*Message Passing Interface*) é padrão de *facto* para o desenvolvimento de aplicações paralelas e distribuídas baseado no paradigma de troca de mensagens [129, 62]. O padrão parte do princípio que as aplicações MPI executam em uma infraestrutura computacional confiável e, dessa forma, não contempla a tolerância a falhas em sua definição. A falha de um único processo implica na interrupção da aplicação. Mesmo as mais importantes implementações do padrão MPI, como a Open MPI [73, 134] e a MPICH [79, 130] não são capazes de manter uma aplicação em execução perante a falha de um único processo.

Os ambientes de computação de alto desempenho (*High Performance Computing* - HPC) empregam amplamente o padrão MPI [62]. Os sistemas HPC costumam executar aplicações que levam semanas ou meses para completar sua execução. Nesses ambientes, uma única falha pode interromper toda a computação realizada e levar a prejuízos econômicos, uma vez que o consumo de energia elétrica desses sistemas é enorme. A frequência de falhas na qual um sistema HPC está sujeito é cada vez maior. Estima-se que os sistemas HPC de grande escala têm que lidar com a ocorrência de falhas em um intervalo de poucas horas [59]. Consequentemente, abordagens e técnicas eficientes para tolerar falhas em aplicações HPC baseadas no padrão MPI são indispensáveis. Este capítulo apresenta o padrão MPI e um apanhado geral das principais técnicas e trabalhos que abordam a tolerância a falhas no contexto de MPI.

2.1 Tolerância a Falhas: Definições Básicas

Um sistema confiável, ou tolerante a falhas, é aquele em que se pode ter confiança no seu funcionamento (em inglês *dependability*) [5]. A tolerância a falhas é a propriedade que garante a correta e eficiente operação de um sistema apesar da ocorrência de falhas em qualquer um dos seus componentes, ou unidades [5, 106].

Um sistema é projetado para executar corretamente e entregar serviços ao usuário de acordo com a sua especificação. Uma falha no serviço (*failure*) é caracterizada pelo não cumprimento da sua especificação ou devido à mesma não descrever adequadamente as funções do sistema. Uma falha no serviço é provocada por um erro (*error*). Um erro é ocasionado pela manifestação de uma falha (*fault*), ou defeito, em um dos componentes do sistema, que pode tanto ser interno quanto externo ao sistema. Por exemplo, estão suscetíveis a falhas os processos, os processadores, a memória ou a rede [5, 94]. O tempo médio entre a ocorrência de falhas (*Mean Time Between Failures* - MTBF) é uma medida primária de confiabilidade do sistema baseada em análises estatísticas do sistema e de seus componentes [106]. É usualmente empregada para indicar o tempo previsto até uma falha no serviço. O sistemas HPC de grande escala, em especial os sistemas *petascale* e os futuros sistemas *exascale*, que podem atingir a

ordem de 10^{15} e 10^{18} operações de ponto flutuante por segundo, respectivamente, apresentam um MTBF que pode chegar a poucas horas ou mesmo minutos [51, 59, 37, 60, 127]. Esses sistemas estão sujeitos a diversos tipos de falhas, conforme apontado em [60, 151].

Um modelo de falhas define o modo pelo qual as unidades do sistema podem falhar. Além disso, oferece uma classificação que especifica as suposições que podem ser feitas sobre o comportamento do componente quando o mesmo falha [106, 94]. Jalote [94] classifica as falhas em uma das seguintes categorias: parada (*crash*), omissão, temporização e bizantinas. Laranjeira [116] adiciona a essa classificação o modelo de falha por computação incorreta. Além dessas, pode-se citar ainda os modelos de falhas *crash-recovery* e *fail-stop*. A seguir os modelos de falhas são descritos.

A falha *crash* ocasiona a parada permanente do componente ou a perda do seu estado interno. Ao falhar, o componente não é submetido a qualquer transição incorreta de estado. Porém, a unidade não executa qualquer ação e tampouco responde a estímulos externos. Na falha por omissão o componente não responde tanto ocasionalmente quanto sistematicamente aos estímulos. Ou seja, na falha por omissão um processo não envia e/ou recebe a algumas mensagens. A falha que leva o componente a responder muito cedo ou a muito tarde é chamada de falha de temporização, isto é, a falha viola uma propriedade temporal do sistema. Esse tipo de falha não é contemplada nos sistemas assíncronos, pois nesses não há definições temporais. Na falha bizantina a unidade se comporta de maneira arbitrária e, inclusive, maliciosamente. A falha bizantina é a mais abrangente e a falha por parada a mais restritiva. Ou seja, a falha bizantina inclui todas as outras. A falha de temporização contém as falhas por omissão e por parada. Por sua vez, a falha por omissão envolve a falha por parada.

Laranjeira [116] ainda cita um subconjunto da falha bizantina: a falha do tipo computação incorreta. Esse tipo de falha ocorre quando uma unidade não produz o resultado correto em resposta a uma tarefa correta recebida como entrada. Diferentemente da falha bizantina, na falha por computação incorreta não há o comportamento malicioso. Seguindo a classificação de Jalote a falha por computação incorreta inclui todas as demais exceto a bizantina.

O modelo de falha *crash* pode considerar a possibilidade de recuperação do processo. Se a recuperação é possível, o modelo de falhas é dito *crash-recovery*. Nesse modelo, um processo que constantemente para e se recupera é chamado de instável [2]. Um processo geralmente guarda suas informações em um armazenamento do tipo estável ou volátil. Durante a recuperação apenas os dados armazenados em dispositivo estável são recuperados. De acordo com Guerraoui [82] a falha de um processo pode ainda ser classificada de acordo com o modelo *fail-stop*. O modelo *fail-stop* inclui a falha do tipo *crash*, porém em um falha do tipo *fail-stop* qualquer processo pode detectar a falha por parada do processo. A maioria trabalhos de tolerância a falhas em MPI assumem o modelo *fail-stop* [28, 17].

2.1.1 Detectores de Falhas

Um detector de falhas, a grosso modo, é um serviço que responde se determinado processo é falho e é frequentemente implementado como um objeto local a cada processo [39, 64, 82]. Distinguir entre um processo falho ou sem-falha é condição essencial para solucionar diversos problemas presentes nos sistemas distribuídos, como o consenso [67]. O consenso, informalmente, permite que um conjunto de processos concordem sobre um valor único com base em valores propostos inicialmente, considerando que esses valores iniciais podem ser diferentes para cada processo. Um detector de falhas nem sempre detecta com precisão a falha de um processo [82]. Da forma como foram propostos por Chandra e Toueg [39], o detector é dito

não-confiável, isto é, é possível que um processo tenha falhado mas não seja suspeito de ter falhado, bem como um processo tenha sido suspeito sem ter realmente falhado.

No modelo de Chandra e Toueg cada processo mantém uma lista dos processos possivelmente falhos e acessa um módulo local através do qual pode consultar o estado dos demais processos. Como os detectores podem errar e eventualmente adicionar à lista um processo correto, é possível mudar o estado do processo posteriormente. Uma vez que os módulos são locais, em um mesmo instante dois módulos de detectores podem possuir uma visão diferente do sistema como um todo. Com base no comportamento dos detectores não-confiáveis e considerando as propriedades de completude (*completeness*) e exatidão (*accuracy*), Chandra e Toueg definiram oito diferentes classes de detectores. A completude assegura que processos falhos terminarão por ser suspeitos. A exatidão restringe os equívocos que podem ser cometidos pelo detector.

Entre as classes de detectores de falhas definidas, destacam-se a P (*perfect*), com as propriedades de completude forte e exatidão forte e o $\diamond$W que apresenta completude fraca e exatidão fraca. Chandra e Toueg provaram que detectores que possuem a completude fraca são equivalentes aos detectores com completude forte e provaram ser o detector de falhas $\diamond$W o mais fraco que permite o consenso [39].

A implementação de detectores de falhas em sistemas distribuídos geralmente faz uso de mensagens de monitoramento. O monitoramento de processos é realizado através de trocas de mensagens. O envio e de recebimento das mensagens deve obedecer a um limite de tempo (*timeout*). De acordo com Felber et al. [64] existem basicamente dois modelos de monitoramento, conhecidos como *push* e *pull*. No primeiro, cada processo monitorado envia periodicamente mensagens do tipo "*I am alive*", também chamadas de *heartbeat*, para o detector que o está monitorando. No modelo *pull* a direção é oposta, isto é, o detector questiona o processo monitorado ("*Are you alive?*"). Se a troca de mensagens ocorrer dentro do intervalo de tempo definido significa que os processos monitorados estão operacionais.

É possível citar ainda outros modelos de detectores, como o modelo Dual [64], o modelo Gossip [144] e o detector *heartbeat* proposto por Aguilera et al. [3]. Basicamente, o modelo Dual combina o modelo *push* e o modelo *pull*. No modelo Gossip o monitoramento é probabilístico. Os processos que recebem as informações são escolhidos aleatoriamente. Uma das vantagens é que o modelo apresenta boa escalabilidade. O modelo de Aguilera, ao invés de usar um *timeout*, somente incrementa um contador a cada mensagem de monitoramento. Outras abordagens de monitoramento, porém baseadas na teoria de diagnóstico em nível de sistemas, serão descritas no Capítulo 3.

A seguir, apresenta-se brevemente o algoritmo Paxos, largamente empregado para obter o consenso em sistemas distribuídos apesar de falhas de processos.

2.1.2 Paxos

Paxos é um algoritmo de consenso tolerante a falhas projetado para replicação de máquina de estado [114, 113]. O consenso está alicerçado em quatro propriedades, sendo uma propriedade de progressão (*liveness*) e três propriedades de segurança (*safety*): terminação, validade, integridade e acordo. A terminação assegura que cada processo correto em algum momento decide por algum valor, sendo esta a propriedade que garante a progressão. A validade significa que se um processo decide por um valor v, então v foi proposto por algum processo. A propriedade de integridade determina que nenhum processo decide duas vezes e a propriedade de acordo afirma que dois processos corretos não decidem valores diferentes [82].

O algoritmo Paxos possui características importantes: garante a segurança do consenso em um sistema assíncrono sujeito a falhas e o progresso sob suposições de assincronia fraca. O

algoritmo assume o modelo de falhas de parada com recuperação (*crash-recovery*). Os canais são não-confiáveis, assim mensagens podem ser perdidas. No Paxos, os processos assumem os seguintes papéis distintos: *proposers*, *acceptors* e *learners*. Os *proposers* propõem um valor, os *acceptors* escolhem um valor e os *learners* aprendem o valor decidido. Um único processo pode assumir qualquer uma dessas funções e múltiplas funções simultaneamente. Paxos é ótimo em termos de resiliência [115]: para tolerar f falhas ele requer $2f + 1$ *acceptors*—isto é, para assegurar o progresso, um quórum de $f + 1$ *acceptors* devem estar sem-falha.

Para garantir que diversas execuções do consenso sejam realizadas, o algoritmo executa sequências separadas de instâncias do Paxos [113]. Cada instância corresponde à execução do consenso e está associada a um valor decidido. Uma instância de Paxos executa em duas fases. A Figura 2.1 apresenta um cenário de execução em duas fases com dois *proposers* (P_0 e P_1), três *acceptors* (A_0, A_1 e A_2) e dois *learners* (L_0 e L_1). O *proposer* P_1 inicia o consenso ao receber um valor v através de uma requisição *propose*. Então, durante a primeira fase P_1 seleciona um número único de rodada e o envia em uma solicitação *prepare* aos *acceptors*. Ao receber uma solicitação *prepare* com um número de rodada maior que qualquer rodada que o *acceptor* tenha recebido previamente, o *acceptor* responde ao *proposer* prometendo que rejeitará qualquer requisição futura com números de rodada menores. Se o *acceptor* já aceitou um valor para instância atual (explicado a seguir), ele irá retornar esse valor ao *proposer* juntamente com o número de rodada recebido quando o valor foi aceito. Na Figura 2.1, a resposta dos *acceptors* é realizada em uma requisição *prepareResp* com os seguintes parâmetros: *n-rod* é o número de rodada, *v-rod* é a rodada em que o comando foi aceito e *v-accept* é o valor aceito. Caso não haja valor aceito, os dois últimos parâmetros são nulos. Quando o *proposer* recebe respostas de um quórum, ou seja, uma maioria de *acceptors*, o algoritmo segue para a segunda fase.

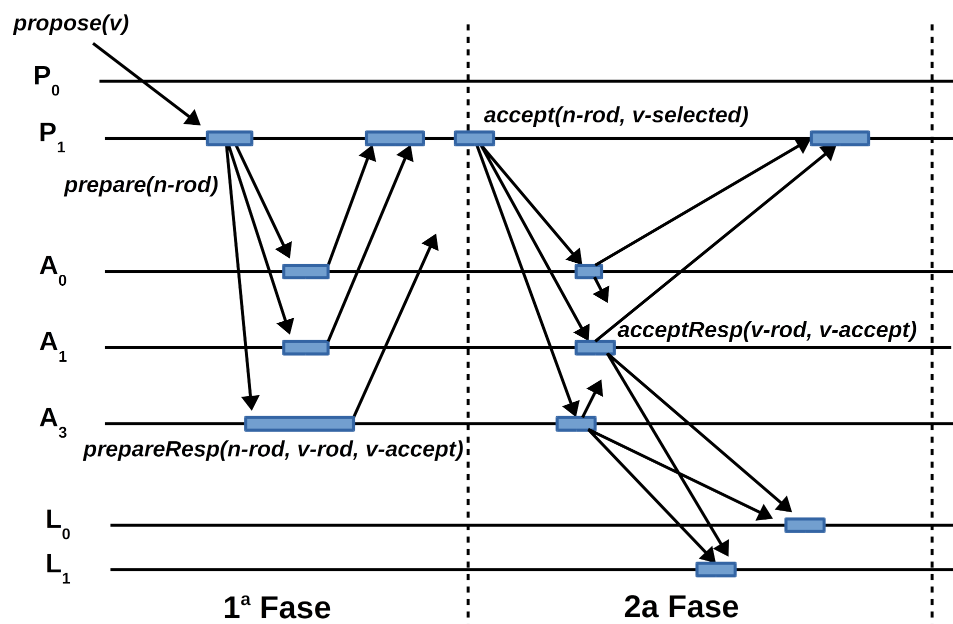


Figura 2.1: Paxos em duas fases

Na segunda fase, o *proposer* seleciona um valor de acordo com a seguinte regra: se nenhum *acceptor* no quórum de respostas aceitou um valor, o *proposer* pode selecionar um novo valor para a instância (no caso, o valor v recebido na requisição *propose*). No entanto, se qualquer um dos *acceptors* retornou um valor na primeira fase, o *proposer* escolhe o valor com o maior número de rodada. O *proposer* então envia uma solicitação *accept* com o número de rodada usado na primeira fase e o valor escolhido (*v-selected*) para os *acceptors*. Quando recebem essa

solicitação, os *acceptors* enviam mensagens de confirmação (*acceptResp*) ao *proposer* e aos *learners*, a menos que os *acceptors* já tenham confirmado outra solicitação com um número de rodada maior. Quando um quórum de *acceptors* aceita o valor, alcança-se o consenso.

Se múltiplos *proposers* simultaneamente executam o procedimento anterior para a mesma instância, então nenhum *proposer* pode conseguir executar as duas fases do protocolo e alcançar o consenso. Por exemplo, P_0 e P_1 na Figura 2.1. Para evitar esse cenário onde *proposers* competem indefinidamente, um processo *coordenador* pode ser escolhido. Neste caso, *proposers* submetem valores ao coordenador, que então executa a primeira e a segunda fase do protocolo. Se o coordenador falha, outro processo assume a sua função. Paxos garante consistência apesar de *proposers* concorrentes e terminação na presença de um único coordenador.

Um coordenador pode otimizar o desempenho executando a primeira fase do protocolo para um lote de instâncias de consenso antes de receber qualquer valor [113]. Isso é possível porque o coordenador somente submete valores na segunda fase do protocolo. Com essa otimização, um valor pode ser escolhido em três passos de comunicação: a mensagem do *proposer* ao coordenador, a solicitação *accept* do coordenador para os *acceptors*, e a resposta a esta solicitação dos *acceptors* para o coordenador e *learners*.

2.1.3 *Rollback-Recovery*

A técnica de tolerância a falhas *rollback-recovery* é frequentemente empregada para prover tolerância a falhas em aplicações de alto desempenho [154, 61, 62, 59]. Desse modo, as aplicações podem reiniciar a partir de um estado salvo previamente. A técnica assume um sistema distribuído onde os processos da aplicação se comunicam através de uma rede e têm acesso a um dispositivo de armazenamento confiável que sobrevive a falhas [61]. Periodicamente, os processos salvam informações de recuperação no dispositivo confiável durante a sua execução sem-falhas. Após a ocorrência de uma falha, a aplicação usa as informações de recuperação para reiniciar a sua computação a partir de um estado anterior. As informações de recuperação incluem os *checkpoints*, isto é, os estados dos processos participantes. Alguns protocolos também incluem informações sobre a recepção de mensagens. Basicamente, um estado global consistente é aquele em que se o estado de um processo reflete uma mensagem recebida, então o estado correspondente do emissor deve refletir o envio daquela mensagem [106]. Um conjunto de *checkpoints* que corresponde a um estado consistente é chamado de *linha de recuperação*. O principal objetivo dos protocolos de *rollback-recovery* é restaurar o sistema a partir da mais recente linha de recuperação após uma falha.

O nível de integração da implementação da técnica de *rollback-recovery* a uma aplicação pode ser classificado de três formas: usuário, aplicação e sistema. No nível de aplicação (*application-level*), o programador ou algum mecanismo de pré-processamento insere o código de *checkpoint* diretamente no código da aplicação. A vantagem dessa abordagem é a independência de plataforma. Porém, há a falta de transparência. Exige-se do programador um bom conhecimento da aplicação para decidir em que momento o estado da aplicação deve ser salvo. Na abordagem em nível de usuário (*user-level*), uma biblioteca é ligada à aplicação e usada para realizar o *checkpoint*. Na abordagem em nível de sistema (*system-level*) o sistema operacional é responsável por salvar o estado da aplicação. É uma abordagem totalmente transparente à aplicação e não há necessidade de alterações de código. A grande desvantagem é a falta de portabilidade [59].

Os protocolos de *rollback-recovery* podem ser classificados em duas categorias: baseados somente no *checkpoint* (*checkpoint-based*) ou baseados em registro de mensagens (*log-based*), descritos a seguir.

Rollback-Recovery* Baseado em *Checkpoints

Os protocolos de *rollback-recovery* baseados em *checkpoints* podem ser divididos em três abordagens: coordenada, não coordenada e induzida pela comunicação (CIC). Quando o *checkpoint* é realizado independentemente em cada processo, sem uma coordenação global, é chamado de não coordenado. A vantagem da abordagem não coordenada está em cada processo criar o seu *checkpoint* quando lhe é mais conveniente. Entretanto, com essa abordagem, um estado global consistente pode nunca ser atingido. Nesse caso, os *checkpoints* realizados tornam-se inúteis e devem ser descartados [61, 106].

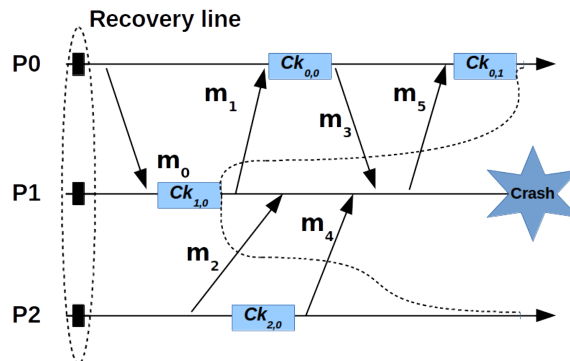


Figura 2.2: *Checkpoint* não coordenado e uma linha de recuperação.

A Figura 2.2 apresenta um cenário no qual o mais recente conjunto de *checkpoints* (isto é, $Ck_{0,1}$, $Ck_{1,0}$, e $Ck_{2,0}$) não resulta em uma linha de recuperação. Isso se deve ao fato de que a mensagem m_5 é recebida por P_0 mas não enviada por P_1 — nesse caso, m_5 é chamada de uma *mensagem órfã* e P_0 um *processo órfão*. P_0 é então obrigado a retroceder a um *checkpoint* anterior (isto é, $Ck_{0,0}$). O problema entretanto persiste, pois m_1 também foi recebida por P_0, mas não enviada por P_1. Desta forma, a única linha de recuperação corresponde ao estado inicial da aplicação. Esse fenômeno é conhecido como *efeito dominó*. O *checkpointing* não coordenado é suscetível ao efeito dominó.

O *checkpointing* coordenado evita o efeito dominó. Os processos se sincronizam para realizar os *checkpoints* e, consequentemente, criar um estado global consistente [106]. Embora a abordagem coordenada seja relativamente fácil de implementar, a sua execução impõe uma sobrecarga considerável à aplicação, uma vez que os processos precisam se coordenar e salvar os seus estados simultaneamente no dispositivo de armazenamento. Além disso, mesmo que um único processo falhe, todos os processos precisam retroceder ao último *checkpoint*.

A abordagem denominada CIC (*communication-induced checkpointing*) força cada processo a realizar os *checkpoints* com base em informações inseridas nas mensagens recebidas dos outros processos. Os *checkpoints* são realizados de forma a manter a estado consistente em todo o sistema. A técnica reúne as vantagens das abordagens coordenada e não coordenada. No entanto, a abordagem relaxa a necessidade de coordenação global, o que a torna ineficiente na prática [28]. Além disso, gera um grande número de *checkpoints*, resultando em sobrecarga no armazenamento e sobrecarga no canal de comunicação devido às informações inseridas nas mensagens da aplicação [106, 59].

O primeiro algoritmo a coordenar todos os *checkpoints* é apresentado por Chandy e Lamport [40]. O algoritmo assume canais FIFO. Qualquer processo pode decidir iniciar um *checkpoint* e quando o faz envia uma mensagem especial chamada *marker* no seu canal de comunicação. Ao receber uma mensagem *marker* pela primeira vez um processo realiza o *checkpoint*. Após o *checkpoint*, todas a mensagens recebidas são armazenadas até uma nova

mensagem *marker* ser recebida. Entre os trabalhos que fazem uso desse algoritmo há o CoCheck [154] e o LAM/MPI [31, 149], descritos na Seção 2.2.8.

Rollback-Recovery Baseado em Registro de Mensagens

Os protocolos de *rollback-recovery* baseados em registro de mensagens empregam tanto *checkpoints* quanto o registro de eventos não-determinísticos com o objetivo de evitar as desvantagens das abordagens coordenada e não coordenada. Um *evento* corresponde a um passo de comunicação ou um passo de computação de um processo. Um evento é determinístico quando a partir do estado atual existe somente um estado resultante possível para o evento. Se um evento pode resultar em estados diferentes, então é dito não-determinístico. A recepção de mensagens com uma identificação de emissor explícita é um evento determinístico e não requer o seu registro. Ao contrário, quando se aguarda uma mensagem de um emissor desconhecido então a recepção é dita não-determinística [25].

Os protocolos de registro de mensagens assumem que todos os eventos não-determinísticos executados por um processo podem ser identificados e a informação necessária para reproduzir cada evento durante a recuperação pode ser codificada em tuplas chamadas de *determinantes* [106]. A maioria dos protocolos de registro de mensagens assume que a recepção das mensagens é o único evento não-determinístico. O registro de mensagens evita o efeito dominó do *checkpointing* não coordenado salvando todas as mensagens recebidas. Por exemplo, na Figura 2.2, as mensagens m_2, m_4 e m_3 recebidas pelo processo P_1 devem ser salvas, assim como os determinantes que contém a ordem de recepção das mensagens. Durante a recuperação do processo P_1 somente P_1 retrocede. Assim, o estado de P_1 eventualmente será o mesmo ao anterior à falha, uma vez que as mensagens m_2, m_4 e m_3 são reaplicadas na mesma ordem.

Dependendo de como os determinantes são registrados, os protocolos de registro de mensagem podem ser classificados em pessimista, otimista ou causal [61]. No registro pessimista, um processo primeiro armazena o determinante antes de entregar a mensagem. Apesar de simplificar a recuperação e a coleta de lixo, a abordagem pessimista gera uma sobrecarga durante a execução da aplicação: a aplicação precisa aguardar pelo armazenamento de cada determinante para então prosseguir. No registro de mensagens otimista, os processos armazenam os determinantes assincronamente, reduzindo assim a sobrecarga. Entretanto, a abordagem otimista pode gerar processos órfãos devido as falhas e, com isso, tornar a recuperação complexa. O registro causal busca combinar as vantagens do registro pessimista e otimista [24]: ter baixa sobrecarga e evitar processos órfãos. Entretanto, os protocolos causais requerem que o determinante seja inserido em cada mensagem trocada pela aplicação até que esse seja confiavelmente armazenado.

O registro de mensagens geralmente faz uso da abordagem *sender-based* [97]. Nessa abordagem, durante a operação normal cada mensagem enviada é salva no emissor. Dessa forma, o receptor da mensagem somente armazena o determinante correspondente, descrevendo o evento de entrega.

O *event logger* desempenha um papel crucial nos protocolos de registro de mensagens [24]. O *event logger* recebe os determinantes dos processos da aplicação, armazena-os localmente, e notifica os processos da aplicação após armazená-los. Apesar do *event logger* exercer um grande impacto na eficiência dos protocolos de registro de mensagens, muitos protocolos o implementam como um componente centralizado e incapaz de tolerar falhas [147, 28, 148]. Uma vez que *event logger* precisa notificar os processos da aplicação ao salvar um determinante, um *event logger* centralizado facilmente se torna em um gargalo conforme aumenta o número de

processos. Além disso, a falha do *event logger* pode paralisar a aplicação ou levá-la a um estado inconsistente durante a recuperação.

Trabalhos que empregam o registro de mensagens como estratégia de tolerância a falhas são descritos na Seção 2.2.8. Descreve-se a seguir o padrão MPI e diversos trabalhos que abordam a tolerância a falhas em aplicações HPC baseadas em MPI.

2.2 Tolerância a Falhas em MPI

O padrão MPI (*Message Passing Interface*) oferece um dos principais modelos para o desenvolvimento de aplicações paralelas e distribuídas baseado no paradigma de troca de mensagens. O paradigma de troca de mensagens se destina a ambientes computacionais em que os nodos acessam uma memória local e estão conectados através de uma rede - que pode ser tanto um barramento de alta velocidade quanto uma rede local de computadores. Embora o MPI seja baseado no paradigma de troca de mensagens, o padrão também pode ser utilizado em sistemas que fazem uso de memória compartilhada.

O MPI consiste de um conjunto de bibliotecas de funções padronizadas pelo MPI-Fórum [71]. Há rotinas para comunicação direta entre dois processos, chamada de comunicação ponto-a-ponto, e rotinas para comunicação coletiva. Além disso, há primitivas para o gerenciamento e criação de processos, entrada e saída de dados em paralelo, gerenciamento de grupos e sincronização de processos e o estabelecimento de topologias virtuais, onde os processos são organizados de forma virtual para realizar a comunicação. O MPI-Fórum é a entidade composta por pesquisadores, desenvolvedores e organizações responsáveis por desenvolver e manter a norma MPI, atualmente na sua versão 3.1 [129]. Entre as principais implementações do MPI, destacam-se a MPICH [79] e a Open MPI [73].

Um conceito fundamental em MPI é o comunicador (*communicator*), uma importante estrutura de dados que define o contexto da comunicação e o conjunto de processos pertencentes a esse contexto. No comunicador, os processos são identificados unicamente por meio de um número inteiro positivo chamado *rank*. Há um comunicador pré-definido chamado `MPI_COMM_WORLD` que reúne todos os processos disponíveis no início da execução de uma aplicação ou programa MPI. Um programa MPI pode possuir um ou mais comunicadores.

Uma propriedade fundamental, porém ausente na especificação original MPI, é a tolerância a falhas [80]. O padrão MPI assume que a infraestrutura subjacente é totalmente confiável [129]. Dessa forma, o padrão não define o comportamento preciso que as implementações MPI devem adotar perante falhas [17, 80]. Basicamente, uma falha é tratada como um erro interno da aplicação como, por exemplo, a violação de um espaço de memória. Dessa forma, as falhas de processo ou de rede são repassadas à aplicação simplesmente como se fossem erros de chamadas de funções. Consequentemente, desloca-se a responsabilidade de detectar e de tratar as falhas para as implementações MPI. A norma define os manipuladores de erros (*error handlers*), que são associados ao comunicador MPI, para lidar com os erros do programa.

O manipulador de erros `MPI_ERRORS_ARE_FATAL` é associado por padrão ao comunicador `MPI_COMM_WORLD`. Esse manipulador define que a manifestação de um erro durante a chamada de uma função MPI leva os processos no comunicador a abortar a sua execução, encerrando assim toda a aplicação. Por outro lado, o manipulador `MPI_ERRORS_RETURN` retorna um código de erro que indica que a ocorrência de uma falha [129]. Mesmo que um código de erro seja retornado ao programa MPI, o padrão MPI não estabelece mecanismos para lidar com as falhas. Por essa razão, o suporte a tolerância a falhas pela norma MPI é considerado inadequado [18, 19]. Além disso, as duas principais implementações MPI citadas anteriormente adotam o manipulador de erro `MPI_ERRORS_ARE_FATAL` por padrão e não dão

suporte adequado ao manipulador de erros `MPI_ERRORS_RETURN`, impedindo a continuidade da aplicação no caso de falha.

Diversos trabalhos visam adicionar à implementação MPI rotinas específicas para lidar com as falhas. Entre esses estão o FT-MPI [62], FT/MPI [7], Gropp e Lusk [80] e recentemente o NR-MPI [156]. O MPI-Fórum também criou um grupo de trabalho específico para abordar a tolerância a falhas na norma MPI. Desse grupo surgiram as propostas chamadas de *Run-through Stabilization Proposal* (RTS) [91] e de *User-Level Failure Mitigation* (ULFM) [16, 17]. A seguir esses trabalhos são descritos resumidamente.

2.2.1 FT-MPI

O FT-MPI (*Fault-Tolerant MPI*), proposto por Fagg e Dongarra no ano 2000, foi desenvolvido como parte do projeto *HARNESS* (*Heterogeneous Adaptive Reconfigurable Networked SyStem*), seguindo a versão 1.2 da norma MPI. O trabalho é a primeira implementação a definir rotinas e uma semântica de tolerância a falhas em MPI e o primeiro a oferecer uma alternativa ao tradicional *checkpoint-restart* (Seção 2.1.3). Entre os objetivos do FT-MPI está construir uma implementação MPI que sobreviva a falhas e ao mesmo tempo forneça um leque de opções de recuperação ao desenvolvedor da aplicação que vão além de apenas retornar à aplicação de estado anterior (*checkpoint*) [62, 63].

Diferentemente na norma padrão, onde o comunicador MPI está no estado válido ou inválido, no FT-MPI o comunicador pode estar em um dos seguintes estados: `FT_OK`, `FT_DETECTED`, `FT_RECOVER`, `FT_RECOVERED` e `FT_FAILED`. Além do comunicador, o processo também recebe diferentes estados: `OK`, `UNAVAILABLE`, `JOINING` e `FAILED`. O estado `UNAVAILABLE` inclui ainda os estados `UNKNOWN`, `UNREACHABLE` ou "não houve votação para remover o processo". Um processo que termina prematuramente, ou a falha de comunicação, modifica o comunicador MPI relacionado para um estado inválido. O módulo chamado *failure handler* é responsável por capturar as notificações de falhas tanto das rotinas usadas na comunicação quanto do sistema operacional. Entretanto, de acordo com Rajanikanth et al. [7], a FT-MPI não define a estratégia usada por este módulo para detectar as falhas.

A implementação atua imediatamente perante falhas de acordo com o estado do comunicador e o modo de recuperação. O modo de recuperação descreve como o FT-MPI deve lidar com os estados do comunicador e dos processos. No modo *shrink* permanecem somente os processos corretos e o *rank* de cada processo é modificado para manter a sequência numérica. Por exemplo, se há cinco processos e o processo de *rank* 2 está falho, os processos de *rank* 4 e 5 serão atualizados para 3 e 4, respectivamente. No modo *blank*, os processos falhos são atualizados para `MPI_PROC_NULL`, o *rank* dos processos são mantidos e as mensagens enviadas ou recebidas pelo processo falho são ignoradas. No modo *rebuild*, os processos falhos são atualizados por novos processos. Em todos esses casos, somente o comunicador `MPI_COMM_WORLD` é reparado e a aplicação é a responsável por reparar qualquer outro comunicador. Uma das desvantagens do FT-MPI de acordo com Bouteiller et al. [28] é a falta de transparência para o desenvolvedor. De acordo com [18], o FT-MPI também não conduziu a esforços de padronização da tolerância a falhas em MPI. Posteriormente, a implementação foi descontinuada.

2.2.2 MPI/FT

O MPI/FT [7] é uma implementação da versão MPI 1.2 estendida da implementação MPI/Pro [1] e acrescida com funcionalidades para detectar e recuperar falhas de processos. Os serviços de detecção e recuperação são projetados de acordo com dois modelos específicos

de execução da aplicação: o modelo mestre-escravo (*master-slave*) e o modelo SPMD (*Single Program Multiple Data*), chamado de *regular-SPMD*. Com base nesses modelos, o MPI/FT constrói estratégias que auxiliam a diminuir a sobrecarga nos serviços de tolerância a falhas. No modelo mestre-escravo, o nodo (ou processo) mestre se comunica com os escravos e os escravos se comunicam com o mestre. Os escravos não se comunicam entre si. Não há chamadas coletivas e a comunicação entre mestre e escravo é ponto-a-ponto. A topologia de comunicação é a estrela. No modelo de execução *regular-SPMD* todos os processos se comunicam entre si. Nesse modelo, o processo de *rank* 0 assume a responsabilidade de recuperar os processos falhos. Diferentemente do FT-MPI, onde o usuário define como o programa deve se comportar após uma falha, o FT/MPI apresenta métodos de recuperação predefinidos de acordo com os modelos de execução.

Para cada um dos modelos há três submodelos que adotam diferentes estratégias de recuperação, tais como: *checkpoint-restart* ou replicação ativa ou passiva. Na replicação ativa, a computação de um nodo é duplicada por um nodo extra que substitui o principal no caso de falha. Na replicação passiva, um processo fica disponível para assumir a execução do processo falho, mas não há duplicação da computação. O *checkpoint-restart* é aplicado somente para alguns nodos. Por exemplo, no modelo mestre-escravo, somente o mestre constantemente salva o seu estado (*checkpoint*). No caso de falha do mestre, a computação é reiniciada de um *checkpoint* anterior.

A detecção de falhas conta com *threads* de autoverificação que monitoram a execução através de um mecanismo de *heartbeat*, que pode ser interno ou externo. O modelo de falhas assume que processos que não respondem a requisições são considerados falhos. No *heartbeat* interno, uma área de memória compartilhada é usada para que as *threads* informem o estado de cada processo. No *heartbeat* externo um processo definido como mestre recebe a informação sobre o estado dos processos escravos. Em ambos os casos o intervalo de tempo do *heartbeat* é definido pelo usuário como um parâmetro na execução da aplicação. O MPI/FT aumenta o intervalo do *heartbeat* interno quando identifica operações de longa duração. De acordo com Genaud et al. [77], o MPI/FT é o primeiro trabalho a usar a técnica de replicação em MPI. No entanto, segundo Genaud et al. e Gropp e Lusk [77, 80], o fato de um processo controlar o estado dos demais acarreta problemas de escalabilidade.

2.2.3 Programas MPI Tolerante a Falhas

O trabalho de Gropp e Lusk [80], proposto em 2004, discute de forma geral o significado de tolerância a falhas em MPI e qual o papel da norma MPI nesse contexto. Com base na norma original, sem qualquer modificação, os autores escrevem programas capazes de sobreviver a falhas, isto é, resilientes. Além disso, é um dos primeiros a defender a modificação e a extensão do MPI a fim de suportar falhas nas aplicações sem, no entanto, comprometer a sua compatibilidade com versões anteriores.

Para Gropp e Lusk a tolerância a falhas não é responsabilidade somente da implementação MPI nem somente de uma aplicação MPI, mas sim de ambas. A resiliência de um programa é classificada em 4 níveis. No primeiro nível, a implementação MPI automaticamente se recupera das falhas e a aplicação MPI continua a sua execução sem mudanças significas no seu comportamento. No segundo nível, o programa é notificado e adota ações corretivas. O terceiro nível de resiliência se caracteriza por certas operações MPI se tornarem inválidas. Em cada um desses três casos os processos sem-falhas retêm o suficiente do estado dos processos falhos para continuar a aplicação. O quarto nível de resiliência é onde o programa aborta e reinicia a partir

de um *checkpoint*. De acordo com os autores, uma combinação entre os níveis também poderia ser usada.

Os autores constroem um programa MPI tolerante a falhas baseado em um algoritmo mestre-escravo através dos recursos existentes na norma original. Seguindo o exemplo de uma comunicação cliente-servidor, onde a falha de um cliente não interfere no trabalho do servidor, a aplicação mestre-escravo é codificada com intercomunicadores. O intercomunicador é um recurso para associar dois grupos distintos de processos, onde cada grupo se relaciona com um comunicador. O autor afirma que a sua aplicação não executa na maioria das implementações MPI, pois embora as mesmas devessem suportar o manipulador de erros `MPI_ERRORS_RETURN`, a maioria não o faz. Por fim, o trabalho apresenta sugestões de modificações na semântica do MPI e de extensões na norma tomando algumas ideias do FT-MPI a fim de suportar aplicações resilientes.

2.2.4 NR-MPI

A NR-MPI [156] (*Non-stop and Fault Resilient MPI*), proposta em 2013, é uma implementação de tolerância a falhas em MPI que segue muitas das definições apresentadas na FT-MPI (Seção 2.2.1). A NR-MPI é construída sobre a implementação de referência MPICH [79]. A principal contribuição do trabalho é implementar a tolerância a falhas de acordo com a FT-MPI. Além disso, o trabalho apresenta respostas para as seguintes questões: a detecção de falhas na biblioteca MPI, a recuperação do estado do comunicador, a recuperação dos dados da aplicação, rotinas para a programação de tolerância a falhas e a conversão de um programa convencional em um programa resiliente por meio da NR-MPI.

A NR-MPI segue o modelo de falhas *fail-stop*. A recuperação das falhas é interna e automática. Se uma falha não pode ser recuperada o estado do comunicador torna-se inválido. Após uma falha, os processos falhos podem ser atualizados tanto por novos processos quanto por processos criados no início da execução para esse fim. A NR-MPI desenvolve um gerenciador de recursos (*Resource Management System* - RMS), semelhante ao SLURM [95]. O RMS é usado, basicamente, para gerenciar e monitorar aplicações paralelas que executam em um *cluster*. O RMS pode ser dividido em um gerente de recursos e um gerente de processos.

A detecção e notificação de falhas conta com dois módulos chamados *Failure Arbiter* (FA) e *Failure Detector* (FD). Esses módulos são integrados ao gerente de recursos e ao gerente de processos, respectivamente. Os FDs detectam a interrupção de um processo através de chamadas do sistema (*SIGCHLD*). O FA usa um *heartbeat* periódico para detectar as falhas dos FDs. A comunicação entre os processos MPI e FDs de um mesmo nodo é através de memória compartilhada. Uma vez que os módulos de detecção são integrados ao RMS, não há interferência no desempenho das tarefas em execução. No entanto, NR-MPI é dependente do RMS, o que pode ser uma desvantagem. Outra desvantagem está o fato de a NR-MPI exigir a modificação do programa.

2.2.5 Grupo de Trabalho de Tolerância a Falhas do MPI-Fórum

A falta de primitivas e de uma semântica de tolerância a falhas na norma MPI, que permitam às aplicações sobreviverem e se recuperarem de falhas de processos, aliada à frequente ocorrência de falhas nos sistemas HPC de grande escala, incentivaram o MPI-Fórum a criar um grupo de trabalho específico para o tema. O Grupo de Trabalho de Tolerância a Falhas (*Fault Tolerance Working Group* - FTWG) [70] foi estabelecido pelo MPI-Fórum, por volta do ano de 2009, com a responsabilidade de otimizar o padrão MPI para permitir o desenvolvimento

de programas HPC portáveis, escaláveis e tolerantes a falhas [91, 90]. Os esforços do grupo de trabalho resultaram em duas propostas: a RTS (*Run-through Stabilization Proposal*) [91, 72] e a ULFM (*User-Level Failure Mitigation*) [18, 17, 69].

A RTS foi a primeira proposta do grupo de tolerância a falhas. Devido à complexidade presente na implementação das primitivas, a proposta RTS não prosseguiu o seu desenvolvimento. A especificação ULFM é o mais recente esforço do MPI-Fórum para padronizar a semântica de tolerância a falhas em MPI. A implementação da ULFM está em desenvolvimento como um subprojeto do projeto Open MPI [73, 128] e na implementação MPICH [15]. Existe a expectativa de que a adotação da ULFM pelo padrão MPI se dê a partir das próximas versões da norma MPI [68].

A RTS e a ULFM possuem semelhanças. Em ambas há o mínimo de interfaces necessárias para recuperar a capacidade do MPI de continuar transportando suas mensagens após uma falha. Além disso, as propostas não definem uma estratégia de recuperação específica. Ao invés disso, disponibilizam um conjunto de funções às aplicações a fim de repararem o seu estado. As novas funções propostas tanto na RTS quanto na ULFM permitem que o desenvolvedor escolha a técnica de tolerância a falhas que melhor se adequa ao programa. A compatibilidade de código com as versões anteriores do MPI também está entre os requisitos observados.

Tanto na RTS quanto na ULFM a aplicação é notificada da falha de um processo ao tentar se comunicar diretamente (comunicação ponto-a-ponto, por exemplo através de um `MPI_Send()` e um `MPI_Receive`) ou indiretamente (operação coletiva, por exemplo através de um `MPI_Bcast()`) com o processo falho. Basicamente, as propostas se comprometem a informar quais condições específicas impedem que a entrega da mensagem ocorra com sucesso, sem que isso promova a interrupção automática da aplicação. A RTS e a ULFM adotam o modelo de falhas *fail-stop* e os manipuladores de erros propostos na norma MPI são os meios para informar a aplicação sobre as falhas de processos.

Na especificação RTS, a implementação MPI deve fornecer um detector de falhas perfeito [39]. Isso significa que em algum momento todo processo falho será conhecido por todos os outros processos. Na RTS os processos ganham um de três estados: *OK*, *FAILED* or *NULL*. Os processos *OK* são os que executam normalmente. Os processos com o estado *FAILED* foram detectados como falhos. Os processos marcados com *NULL* são processos falhos cujos *ranks* recebem a constante `MPI_PROC_NULL`.

A RTS trata as falhas de processos de acordo com o modelo de comunicação empregado [91]. A comunicação ponto-a-ponto recebe um tratamento diferente da comunicação coletiva: a comunicação realizada por um par de processos raramente é afetada pela falha de outro processo; na comunicação coletiva, que envolve um grupo de processos, a falha de um único processo afeta os demais. Dessa forma, a RTS fornece duas abordagens para o tratamento de falhas: local e global. Enquanto o tratamento das local das falhas se destina à comunicação ponto-a-ponto, o tratamento global é destinado à comunicação coletiva.

Na RTS, um processo usa uma função de validação para atualizar, acessar e modificar o estado dos processos no comunicador MPI. Com isso, a RTS mantém forte consistência entre os processos. A primitiva `MPI_Comm_validate` é usada para o tratamento local da falha e a primitiva `MPI_Comm_validate_all` para o tratamento global da falha. A última é também responsável por retornar a mesma lista de processos falhos para todos os processos. Um algoritmo de consenso distribuído é empregado pela primitiva `MPI_Comm_validate_all`. Essa primitiva sincroniza os detectores de falhas, reabilita as comunicações coletivas, identifica todos os processos falhos e fornece um valor de retorno uniforme às funções coletivas. A Seção 2.2.7 descreve os algoritmos de consenso empregados na ULFM e na RTS.

Na especificação ULFM, a falha de um processo somente é detectada se esse processo participa ativamente de uma comunicação (ponto-a-ponto ou coletiva). Isto é, somente os processos que se comunicam diretamente com o processo falho o detectam. Ao contrário da RTS, a ULFM não cita o uso de um detector de falhas. A aplicação é notificada da falha durante a execução das operações de comunicação. A Figura 2.3 apresenta um exemplo com três processos (A, B e C) que realizam uma comunicação ponto-a-ponto.

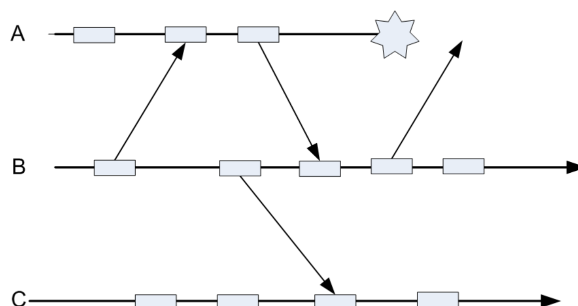


Figura 2.3: Detecção de falhas ULFM - Processo B detecta a falha em A.

Conforme apresenta a Figura 2.3, o processo B detecta a falha do processo A após enviar mensagem para A. Por sua vez, o processo C não identifica a falha em A. Essencialmente, a ocorrência de uma falha indica que a comunicação não pôde ser executada com sucesso. Por razões de desempenho, não há propagação automática sobre a ocorrência de falhas. Se durante uma operação coletiva um processo falhar, é possível que somente alguns processos identifiquem a falha. Ao todo, a ULFM disponibiliza ao usuário cinco funções para lidar com as situações de falhas. Entre essas, algumas permitem estabelecer uma visão consistente entre os processos. As primitivas da ULFM são descritas a seguir.

A operação de revogação, realizada pelo construtor `MPI_Comm_revoke`, é a mais crucial e complexa entre os construtores. Essa operação notifica todos os processos que o comunicador MPI a que pertencem está inválido. Dessa forma, evita a inconsistência entre os processos associados a um comunicador. O comunicador torna-se inválido e as comunicações futuras, ou as comunicações pendentes, são interrompidas e marcadas com um código de erro. A operação de revogação da ULFM conta com requisitos semelhantes à difusão confiável (*reliable broadcast*) [82]. Nessa implementação, a ULFM usa um grafo binomial (*binomial graph*) onde o iniciador marca o comunicador como revogado e envia uma mensagem de revogação a outros $log(n)$ processos, considerando n processos. O processo, ao receber a mensagem de revogação, verifica se o comunicador foi marcado como inválido e, em caso contrário, atua como novo iniciador.

A primitiva `MPI_Comm_agree` é empregada para determinar uma visão consistente entre os processos. Essa função executa uma operação coletiva e faz com que os processos concordem com um valor booleano, mesmo se o comunicador foi revogado. Basicamente, o valor booleano pode significar o sucesso (0) ou a falha (1) na comunicação com um processo específico. Para fazer uso dessa primitiva o processo que identifica a falha deve antes revogar o comunicador. O construtor `MPI_Comm_shrink` permite criar um novo comunicador, eliminando todos os processos falhos de um comunicador inválido. Essa operação é coletiva e executa um algoritmo de consenso para assegurar que todos os processos tenham a mesma visão no novo comunicador. Por fim, os construtores `MPI_Comm_failure_ack` e `MPI_Comm_failure_get_acked` são usados para informar quais processos dentro do comunicador se encontram falhos.

Por exemplo, na Figura 2.3, o processo B identifica a falha do processo A e então executa a função de revogação para que todos os processos corretos, no caso o processo C,

tornem o seu comunicador inválido. Após isso, todos os processos executam a operação de acordo (`MPI_Comm_agree`) para garantir uma visão consistente sobre o estado do comunicador. Então, um novo comunicador válido, somente com processos corretos, é criado por meio da função `MPI_Comm_shrink`. Se houver a necessidade de identificar qual processo falhou (no caso o A), as primitivas `MPI_Comm_failure_ack` e `MPI_Comm_failure_get_acked` são empregadas.

As falhas temporárias, tanto de rede quanto de processo, não fazem parte do escopo da ULFM, mas podem ser tratadas em nível de implementação. Uma falha temporária pode ser promovida a uma falha permanente (conforme o modelo *fail-stop*). Ou seja, se um processo sem-falha detecta que um processo deixa de responder, mesmo que temporariamente, o processo sem-falha classifica esse processo como falho e continuamente ignora e descarta qualquer comunicação com o processo falho. Nesse caso, como dito anteriormente, para evitar que os processos tenham uma visão diferente sobre o estado de algum processo, as rotinas da ULFM (`MPI_Comm_revoke`, `MPI_Comm_agree` e `MPI_Comm_shrink`) são usadas. Os algoritmos de consenso empregados nas primitivas da ULFM são apresentados na Seção 2.2.7.

Em relação a tolerância a falhas em MPI, é possível ainda citar outros trabalhos, como [14]. Esse trabalho propõe modificações na implementação Open MPI para permitir às aplicações MPI sobreviverem e continuarem sua execução apesar de falhas. A proposta é codificada na implementação Open MPI [73]. Basicamente, a proposta requer modificar o ambiente de execução do Open MPI (*Open MPI Run Time Environment* - ORTE) para incluir a detecção e a notificação de falhas. Laguna et al. [111, 112] discutem algumas desvantagens da proposta ULFM e propõem uma abordagem diferente para lidar com as falhas em MPI. De acordo com os autores, uma importante desvantagem é que a ULFM não fornece meios explícitos para reiniciar processos que falharam. Os autores propõem uma nova estratégia onde o MPI é responsável por reiniciar os processos falhos.

2.2.6 Detecção de Falhas em MPI

Conforme apresentado, a ULFM trata de falhas de processos envolvidos na comunicação da aplicação. Um processo é considerado envolvido na aplicação se: 1) o processo participa de um operação coletiva (`MPI_Bcast`) e aparece em um dos grupos associados ao comunicador; 2) o processo é a origem ou o destino em uma comunicação ponto-a-ponto (`MPI_Send`) ou (`MPI_Recv`) ou; 3) o processo está programado para receber uma mensagem de uma origem qualquer (`MPI_ANY_SOURCE`). A aplicação é notificada da falha de um processo através de uma mensagem de exceção somente quando há um processo falho envolvido na comunicação. A ULFM fornece primitivas para que os demais processos tomem conhecimento da falha, se for necessário. É possível ainda, através das primitivas fornecidas, excluir o processo falho associado ao comunicador e continuar a execução da aplicação com os processos que não falharam.

Kharbas et al. [103] fazem a avaliação prática de detectores de falhas em HPC no contexto de aplicações MPI. Ao contrário da ULFM, o trabalho se fundamenta na definição clássica de detectores de falhas [40], ou seja, componentes que servem de oráculos ao sistema informando os processos falhos. O trabalho assume o modelo de falhas *fail-stop* e um sistema parcialmente síncrono. O monitoramento dos processos é realizado via mensagens chamadas de *ping-pong* que monitoram o tempo das respostas de acordo com um *timeout*. Se um processo não responde dentro do intervalo de tempo definido é considerado falho. O trabalho implementa duas abordagens de detectores de falhas. A primeira utiliza o monitoramento periódico onde os nodos ou processos são dispostos em uma rede de sobreposição no formato de anel. Um nodo i monitora o seu vizinho $i + 1$. As falhas são propagadas no anel. A segunda abordagem

é chamada de esporádica. Um processo p monitora um processo q somente se ambos estão envolvidos em alguma comunicação. De acordo com o trabalho, o monitoramento periódico apresenta melhor desempenho que o monitoramento esporádico para aplicações que exigem comunicação intensiva.

O trabalho de Genaud et al. [77] apresenta um estudo sobre o gerenciamento de falhas em um *middleware* de *grid* computacional chamado de P2P-MPI. Um *grid* é um sistema distribuído geralmente espalhado em diversos domínios administrativos que oferece aos usuários acesso transparante aos seus recursos. O gerenciamento de falhas abrange a tolerância a falhas e a detecção de falhas. A tolerância a falhas é obtida por meio da técnica de replicação (Seção 2.2.9). No *middleware* há um serviço de detecção de falhas local a cada *host* ou nodo. Em cada nodo o serviço é responsável por notificar os processos da aplicação sobre a ocorrência de falhas. Localmente, o detector de falhas realiza o monitoramento dos seus processos via mensagens de *heartbeat*, semelhante ao MPI/FT (Seção 2.2.2). Cada detector de falhas periodicamente (30 segundos, por exemplo) questiona os processos se os mesmos estão vivos. Então, os detectores de falhas trocam informações sobre o estado dos processos. O monitoramento segue um protocolo baseado em disseminação epidêmica (*gossip*). O trabalho assume o modelo de falhas *fail-stop* e o modelo parcialmente síncrono. Os canais são confiáveis, ou seja, não há perda ou replicação das mensagens.

2.2.7 Consenso Distribuído para MPI

Um algoritmo de consenso escalável e distribuído para suporte às primitivas de tolerância a falhas propostas pelo MPI-Fórum é proposto por Buntinas [30]. O algoritmo é usado para implementar a primitiva `MPI_Comm_validate` da proposta RTS. O algoritmo assume as seguintes condições: os processos falham permanentemente de acordo com o modelo *fail-stop*, partições na rede não são consideradas e há um detector de falhas *eventually perfect* [39] com o seguinte requisito adicional: se qualquer processo suspeita de um processo ter falhado, então esse processo será suspeito permanentemente e, em algum momento, os demais processos suspeitarão desse processo. O algoritmo ainda assume que um processo suspeito de ter falhado não faz parte da execução. Um algoritmo de *broadcast* em árvore é empregado para assegurar que todos os processos recebam as mensagens apesar de processos falhos.

O algoritmo de Buntinas executa em três fases, descritas brevemente a seguir. Na primeira fase o processo chamado de raiz gera um valor (*ballot*) e o dissemina aos demais. O processo raiz é o processo com menor identificador entre os processos não suspeitos. Cada processo então responde com uma mensagem `ACCEPT` ou `REJECT` no caso de ter aceito ou não o valor. O processo raiz coleta as respostas e se o valor for rejeitado um novo valor será gerado, repetindo a primeira fase. Se o valor for aceito por todos os processos, o processo raiz inicia a segunda fase disseminando uma mensagem `AGREE` a todos os processos. Uma vez que os demais processos recebem a mensagem `AGREE`, eles sabem que todos os processos concordaram com valor emitido na primeira fase. Na terceira fase então o processo raiz dissemina uma mensagem `COMMIT`. Ao receber o `COMMIT` os processos decidem no valor da primeira fase. Otimizações do algoritmo também são descritas. Os experimentos são realizados considerando a latência para 4096 processos. Segundo o autor, o algoritmo requer $O(\log n)$ passos para finalizar no cenário livre de falhas.

O algoritmo de consenso Paxos (descrito na Seção 2.1.2) e os detectores de falhas não-confiáveis de Chandra e Toueg [39] são geralmente empregados para alcançar consenso em sistemas distribuídos. De acordo com Buntinas, esses algoritmos apresentam problemas de escalabilidade: o coordenador envia e recebe mensagens individualmente de cada processo.

No entanto, diversas versões do Paxos diminuem a sobrecarga no coordenador permitindo maior escalabilidade ao algoritmo [125, 124]. O trabalho de Buntinas, emprega um algoritmo de *broadcast* em árvore tolerante a falhas para distribuir e coletar as mensagens, tornando o algoritmo mais escalável. No entanto, ao contrário do Paxos, o trabalho de Buntinas não lida com partições de rede e não prevê a recuperação de processos. Além disso, no algoritmo de Buntinas, todos os processos não suspeitos precisam aceitar o valor da primeira fase e não apenas uma maioria de processos corretos como no Paxos. O trabalho também não implementa um detector de falhas. O trabalho de Ranganathan [142] propõe uma abordagem escalável para a detecção de falhas e um algoritmo de consenso distribuído para *clusters* heterogêneos. Em Ranganathan a detecção de falhas é baseada em *gossip*.

De acordo com [18], um dos algoritmos de acordo usado na ULFM é o proposto por Hursey et al. [92]. O algoritmo se baseia no procolo 2PC (*Two-Phase Commit*). Um coordenador é eleito no comunicador e é responsável por coletar os valores dos processos participantes. Então, o coordenador decide de acordo com os valores de entrada e difunde o valor a todos os processo sem-falha no comunicador. O algoritmo usa uma topologia em árvore tanto para reunir os valores de entrada quanto para difundir a decisão. A complexidade do algoritmo é logarítmica e devido à estrutura de árvore o número de mensagens trocadas é da ordem de $nlog(n)$.

Recentemente, o algoritmo de acordo chamado de *Early Returning Agreement* (ERA) foi proposto e avaliado para a ULFM [85]. Entre as características do algoritmo está o seu comportamento logarítmico. De acordo com os autores, a propriedade logarítmica é importante para qualquer algoritmo a ser executado nos sistemas de grande escala, como o *exascale*. Os autores ainda detalham a implementação do algoritmo no contexto de uma implementação MPI. Apesar de o algoritmo Paxos também ser citado como um algoritmo eficiente, duas razões são citadas para não utilizá-lo no contexto da ULFM: 1) no contexto do MPI todos os processos contribuem para o valor decidido, isto é, o resultado é a combinação dos valores propostos por um grupo de processos; 2) as suposições válidas em um ambiente MPI são diferentes daquelas assumidas pelo Paxos.

2.2.8 *Rollback-Recovery* em MPI

Conforme descrito na Seção 2.1.3, o *Rollback-Recovery* é a principal técnica de tolerância empregado em aplicações HPC baseadas no padrão MPI. A seguir alguns trabalhos que fazem uso dessa técnica são descritos brevemente.

O ambiente CoCheck [154] é o primeiro esforço para incluir a tolerância a falhas em MPI. O ambiente faz uso da técnica de *checkpoint-restart* e de migração de processos. A sua implementação é em nível de usuário. Tanto o reinício da aplicação quanto a migração de processo são transparentes à aplicação. O ambiente executa acima da biblioteca MPI e o programador precisa associá-lo à aplicação. A partir de então, as primitivas do CoCheck são usadas para comunicação ao invés das primitivas originais. Há um processo especial para coordenar os *checkpoints*. O mesmo envia uma notificação para todos os processos envolvidos na execução da aplicação realizarem o *checkpoint*. Para tanto, cada processo envia mensagens especiais, chamadas *ready messages*, nos canais para assegurar que não há mensagens em trânsito e assim garantir a consistência global. Cada processo independentemente mantém um *checkpoint* consistente. É possível também realizar um *checkpoint* de toda a aplicação. A principal desvantagem do CoCheck está na necessidade de sincronizar toda a aplicação para efetuar o *checkpoint*, o que pode levar a problemas de escalabilidade. Outra desvantagem do ambiente é implementar a sua própria versão do MPI, chamada tuMPI [63].

O LAM/MPI é uma implementação de referência do MPI [31]. No trabalho de Sankaran et al. [149] o LAM/MPI é estendido para suportar o *checkpoint* coordenado. A abordagem integra o LAM/MPI com a implementação de *checkpoint* em nível de sistema denominada BLCR (*Berkeley Lab's linux Checkpoint/Restart*) [58] através de interfaces definidas para *checkpoint-restart*. O BLCR é uma implementação que suporta aplicações com múltiplas *threads* no ambiente Linux. O LAM/MPI implementa o *checkpoint* em nível de usuário. Há algum tempo os esforços de desenvolvimento do LAM/MPI e dos seus mecanismos de tolerância a falhas foram portados para o desenvolvimento do Open MPI. No entanto, o Open MPI atualmente não dá suporte a qualquer estratégia de *checkpointing*.

A aplicação da técnica de *rollback-recovery* puramente baseada em *checkpoints* para sistemas HPC de larga escala vem sendo colocada em cheque devido ao MTBF cada vez menor desses sistemas [37, 59]. Embora eficiente, o custo para armazenar e recuperar os *checkpoints* pode exceder o MTBF dos futuros sistemas HPC *exascale*. Mais tempo será gasto lidando com as falhas do que realizando a computação útil [146, 37]. Por exemplo, o Blue Waters [133], um sistema HPC *petascale* da universidade de Illinois, apresenta um MTBF médio de 4,2 horas [51]. Por outro lado, como apontado em Tiwari et al. [158] uma aplicação de astrofísica possui um volume de dados de 160TB e pode levar 360 horas para finalizar sua execução. Realizar um *checkpoint* a cada 1 hora, por exemplo, causa um grande impacto no sistema. Aumentar o intervalo de *checkpoint* pode reduzir o impacto no sistema, porém aumenta a quantidade de trabalho perdido no caso de falha: o processamento realizado entre o último *checkpoint* e a falha.

Diversos trabalhos buscam melhorar o desempenho da técnica de *rollback-recovery* baseado em *checkpoints* para sistemas HPC de larga escala através de diferentes propostas, como o Multi-level Checkpointing [127], protocolos híbrido ou hierárquicos que combinam o *checkpoint* coordenado com o registro de mensagens [145, 146] e versões otimizadas [158, 27]. A seguir apresentamos algumas dessas abordagens, incluindo a abordagem de registro de mensagens.

O Diskless Checkpoint [138, 44] é uma técnica que elimina a sobrecarga imposta pelo armazenamento estável do *checkpoint* tradicional. Nessa técnica, o estado de uma aplicação distribuída de longa duração é persistido tanto em memória quando em disco local. Adicionalmente, codificações relacionadas a esses *checkpoints* são armazenadas em processos redundantes - que podem ou não estar envolvidos na computação. Quando uma falha ocorre, os processos que não falharam recuperam o seu último *checkpoint*. O estado dos processos falhos pode ser calculado a partir do *checkpoint* dos processos que não falharam e das codificações relacionadas aos *checkpoints*.

Multi-level Checkpointing [127] é uma abordagem que emprega múltiplos tipos de *checkpoints*, com diferentes níveis de resiliência e custo, em uma única execução da aplicação. O nível inferior é o mais lento e o mais resiliente. Nesse nível o *checkpoint* é escrito em um sistema de arquivos paralelo - o qual pode suportar a falha de todo o sistema. Os níveis superiores, apesar de serem mais rápidos, são os menos resilientes: os *checkpoints* são salvos em armazenamento local, tais como a memória RAM, memória *flash*, disco local ou cópias redundantes entre os nodos do sistema. Os trabalhos descritos em [8] e [50] propõem, respectivamente, otimizações à abordagem através de códigos de proteção de dados e estudos sobre o intervalo de *checkpoint* em execuções onde o número de processadores e/ou núcleos envolvidos na computação é variável.

Fenix [75, 76] é um arcabouço que permite a recuperação transparente e em tempo de execução de aplicações MPI. O arcabouço faz uso da especificação ULFM para sobreviver as falhas e emprega a técnica de Diskless Checkpoint: os dados da aplicação são salvos na memória dos nodos vizinhos [165]. Também são disponibilizadas primitivas para o desenvolvedor realizar o *checkpoint* dos dados essenciais da aplicação. Fenix adota uma abordagem de chamada

de *checkpoints* implícitos, onde os *checkpoints* são salvos de forma não-coordenada, porém, considerando a posição onde os *checkpoints* são inseridos no código há a garantia de que estados globais consistentes são sempre gerados pela aplicação. A avaliação do Fenix foi realizada em uma aplicação MPI executando milhares de processos. O arcabouço não está disponível para uso.

O projeto MPICH-V [28] apresenta três protocolos de registros de mensagens que trabalham em conjunto com o *checkpointing* não coordenado. Dois protocolos são pessimistas e um é causal. O MPICH-V1 é um protocolo pessimista projetado para ambientes heterogêneos e com alta volatilidade, tais como grades computacionais formadas por *desktops*. O MPICH-V1 faz uso de um componente remoto e confiável, chamado de *Channel Memory* (CM) responsável por armazenar o conteúdo das mensagens e a ordem de recepção das mensagens MPI. Cada processo primeiro envia a sua mensagem para o CM do receptor. Então, o receptor solicita a mensagem do seu próprio CM. Apesar de haver um CM para cada processo, eles não suportam falhas. O MPICH-V2 [26] é um protocolo pessimista destinado a grandes *clusters*. O MPICH-V2 conta com a abordagem *sender-based*. Os processos se comunicam diretamente. Ao invés de usar os CMs, o MPICH-V2 emprega *event loggers* que são usados como armazenamento remoto confiável. Quando um processo recebe uma mensagem, envia o determinante da mensagem para o *event logger*.

Em Lemarinier et al. [118] uma estratégia de *checkpointing* coordenada baseada no algoritmo de Chandy e Lamport é comparada com o protocolo MPICH-V2 usando o *checkpointing* não coordenado em diferentes frequências de falhas e volume de dados da aplicação. A principal conclusão é a de que o registro de mensagens se torna relevante para *clusters* de grande escala a partir de uma taxa de falhas de uma falha a cada hora para aplicações com grande volume de dados.

Em Bouteiller et al. [24], os autores investigam os benefícios de um *event logger* no protocolo de registro de mensagens causal. Três protocolos foram implementados e comparados com e sem um *event logger*: Manetho, LogOn and Vcausal. A conclusão dos autores é a de que o *event logger* exerce um grande impacto em diversos aspectos do desempenho, incluindo o desempenho da aplicação e da recuperação de falhas. O trabalho assume um *event logger* confiável. Os autores destacam que empregar apenas um *event logger* para consistência ocasiona um gargalo conforme aumenta o número de processos. Os autores ainda afirmam que é necessário investigar como distribuir o registro de eventos entre múltiplos *event loggers*.

No trabalho em [29], Bouteiller et al. comparam experimentalmente um protocolo pessimista e otimista de registro de mensagens considerando o refinamento proposto em um trabalho anterior envolvendo os mesmos autores [25] que distingue eventos determinísticos dos não-determinísticos em MPI. Esse refinamento é codificado em um protocolo chamado de `Vprotocol` na implementação Open MPI. Como consequência, o número de mensagens enviadas ao *event logger* diminui consideravelmente. No `Vprotocol` o *event logger* não é tolerante a falhas e é implementado como um processo especial disponível à aplicação em um grupo externo ao grupo MPI principal, ou seja, o `MPI_COMM_WORLD`. Atualmente, o `Vprotocol` não está disponível na biblioteca Open MPI.

Geralmente, os protocolos de registro de mensagens criam determinantes para todas as mensagens recebidas. No entanto, é possível reduzir o número total de determinantes armazenados distinguindo os eventos determinísticos dos não-determinísticos [25]. Por exemplo, um evento não-determinístico ocorre em MPI quando o processo receptor usa uma marcação `MPI_ANY_SOURCE` na primitiva `MPI_Recv`. Conforme definido em Cappello et al. [38], muitas aplicações MPI contêm somente eventos de comunicação determinísticos. Alguns protocolos de tolerância a falhas foram propostos para essa classe de aplicação [81, 117, 146]. Porém,

importantes aplicações MPI são não-determinísticas. Além disso, os desenvolvedores geralmente incluem o não-determinismo na codificação para melhorar o desempenho da aplicação.

O trabalho de Ropars e Morin [147] propõe o O2P, um protocolo ativo de registro de mensagens otimista. Nesse protocolo o *event logger* é implementado como um processo MPI capaz de manipular comunicações assíncronas. O *event logger* é inicializado separadamente da aplicação MPI. Os processos da aplicação se conectam ao *event logger* ao iniciar o registro dos determinantes. O O2P assume um *event logger* confiável. Os experimentos com um grande número de processos e uma alta taxa de comunicação mostram que o *event logger* é um gargalo para o desempenho do sistema.

Um *event logger* distribuído para o protocolo O2P também é proposto por Ropars e Morin [148]. O *event logger* aproveita a arquitetura multi-core dos processadores para ser executado em paralelo com os processos da aplicação. Cada nodo executa um *event logger*. Por exemplo, um nodo que possua um processador com quatro núcleos de processamento pode ter três processos da aplicação e um *event logger*. Os determinantes são salvos na memória volátil do *event logger* e são replicados entre os *event loggers*. Há um parâmetro chamado `replicationdegree` que informa ao *event logger* original quantos processos devem receber a cópia do determinante. Por exemplo, se o `replicationdegree` é dois, um processo envia seus determinantes para outros dois *event loggers*. Quando um *event logger* recebe duas respostas de confirmações o determinante é considerado estável. O autor ainda propõe um protocolo de disseminação epidêmica (*gossip*) para espalhar os determinantes estáveis para todos os *event loggers*. Apesar de esse protocolo oferecer uma forma distribuída de salvar os determinantes, a solução falha se uma única resposta de confirmação não for recebida pelo emissor. Isto é, a tolerância a falhas da solução não é garantida.

Um protocolo híbrido que combina o *checkpointing* coordenado e o registro de mensagem otimista é proposto por Riesen et al.[145]. O protocolo faz uso de nodos adicionais que agem como *event loggers* ou nodos extras no caso da falha de um nodo. O *checkpointing* coordenado auxilia o registro de mensagens limitando o tamanho dos registros e evitando retornar à aplicação ao seu estado inicial no caso de um estado inconsistente se fazer presente. Por sua vez, o registro de mensagens evita reiniciar todos os processos no caso de falhas na maioria das vezes. O trabalho assume a presença de um serviço para detectar processos falhos e reiniciar processos em nodos extras. Se o *event logger* falha, a aplicação continua sua execução. Ao atingir um *checkpoint* global, o serviço usa um nodo extra para lançar um novo *event logger* e informar cada processo sobre o novo *event logger*. No entanto, se um nodo precisa recuperar determinantes que foram perdidos devido à falha do *event logger*, então a aplicação reinicia a partir do último *checkpoint* coordenado. O *event logger* não é distribuído. Apesar de o protocolo de registro de mensagens otimista diminuir a sobrecarga, processos órfãos podem ser criados. Outro protocolo híbrido é proposto em [27]: *checkpoint* coordenado é usado dentro dos nodos, onde os processo são relacionados, e *checkpoint* não coordenado com registro de mensagens pessimista é empregado entre os nodos.

2.2.9 Replicação Máquina de Estado Aplicada para MPI

A replicação máquina de estados (*State-Machine Replication* - SMR) é uma das mais importantes técnicas de tolerância a falhas [41]. Nessa técnica, frequentemente empregada para fornecer serviços com alta disponibilidade, o estado de um processo é replicado de tal forma que, se um processo falhar, a sua réplica, ou cópia, mantém o serviço disponível. O serviço é definido por uma máquina de estados, que consiste de variáveis e de operações. Uma operação pode tanto ler o estado das variáveis quanto modificá-las. A execução das operações são determinísticas,

isto é, se duas réplicas executam a mesma sequência de operações na mesma ordem, o mesmo estado deve ser produzido em ambas [150].

O trabalho de Ferreira et al. [65] avalia a viabilidade da técnica de replicação máquina de estados como principal mecanismo de tolerância a falhas para os sistemas HPC *exascale*. A justificativa dos autores para usar a técnica de replicação é a diminuição drástica do tempo médio de interrupção devido a uma falha (*Mean Time To Interrupt* - MTTI) e o fato de estudos apresentarem que os sistemas *exascale* podem levar mais do que 50% do seu tempo lendo e escrevendo *checkpoints*. O *checkpoint-restart* seria aplicado em algumas situações como, por exemplo, quando todas as réplicas falhassem. As aplicações MPI são o objeto de estudo do autor: as cópias redundantes dos processos MPI permitem que perante uma falha do processo original a aplicação continue a sua execução de forma transparente, sem a necessidade de *rollback-recovery*. Segundo o autor, a técnica de replicação também poderia ser usada para detectar um leque maior de falhas, potencialmente incluindo as falhas maliciosas (bizantinas).

Ferreira et al., assim como no trabalho de Gropp e Luks [80] (Seção 2.2.3), também argumentam que não são todos os processos que precisariam ser duplicados. Por exemplo, considerando o modelo mestre-escravo somente o processo mestre seria replicado. Para realizar a avaliação, o trabalho combina modelagem, análise empírica e simulação para estudar os custos e benefícios da replicação em comparação com *checkpoint-restart*. Para estudar a sobrecarga imposta pela replicação dos processos, a ferramenta rMPI é projetada e implementada. A rMPI é implementada acima das implementações MPI existentes e fornece redundância de computação transparentemente às aplicações MPI determinísticas. O autor conclui que a técnica de replicação máquinas de estados tem potencial para atender as demandas de HPC.

A Ferramenta rMPI adota o modelo de falhas *fail-stop*: um processo falha por parada e então a sua réplica assume. O trabalho de Fiala et al. [66] propõe a ferramenta redMPI com o objetivo de detectar e corrigir erros do tipo computação incorreta através da replicação da computação. Basicamente, a ferramenta compara as tarefas executadas pela réplica principal com as executas pelas suas cópias.

O trabalho de Bougeret et al. [23] adota uma estratégia de replicação de grupo ao invés de replicação de processos proposto em Ferreira et al. [65]. A replicação de grupo consiste em executar múltiplas instâncias da aplicação concorrentemente. Ao contrário do trabalho de Ferreira et al., a estratégia pode ser utilizada em qualquer modelo de programação de sistemas HPC. Um estratégia de *checkpoint* também é usada pelos autores. Os resultados obtidos no trabalho demostram que a replicação de grupo pode apresentar vantagens em sistemas HPC de grande escala.

2.2.10 Tolerância a Falhas Codificada no Algoritmo da Aplicação

A técnica de ABFT (*Algorithm-Based Fault Tolerance*) faz uso das propriedades do algoritmo da aplicação para recuperá-la de falhas durante a sua execução, como se ignorasse a existência de falhas [48, 53, 90, 164]. A técnica não é transparente à aplicação. Os requisitos mínimos para utilizar a técnica são a detecção, notificação e propagação de falhas, assim como o suporte do ambiente de execução, que deve ser resiliente. Dessa forma, um dos empecilhos para a ampla adoção da técnica em aplicações MPI é falta de primitivas e de uma semântica padronizada de tolerância a falhas. A especificação MPI e suas implementações de referência não fornecem meios para detectar e sobreviver as falhas de rede e de processos (Seção 2.2). A maioria dos trabalhos que aplicam a técnica em MPI usam as implementações FT-MPI (Seção 2.2.1), a RTS ou a ULFM (Seção 2.2.5).

A técnica foi originalmente proposta por Huang e Abraham para detectar e corrigir erros em algumas operações em matrizes causados por falhas transientes ou permanentes no hardware [89]. De acordo com os autores, para algumas operações em matrizes há uma relação entre o *checksum* de entrada e o *checksum* apresentado nos resultados finais. Com base nessa relação, a técnica é desenvolvida para detectar, localizar e corrigir certos erros de cálculo do processador nas operações de matrizes.

A técnica ABFT é adaptada para sistemas HPC por Chen e Dongarra para suportar falhas de acordo com o modelo *fail-stop* durante a execução de programas que envolvam operações em matrizes [42, 43]. Assim como Huang e Abraham, Chen e Dongarra fazem uso da relação do *checksum*, mencionado acima. A técnica é aplicada sem a necessidade de qualquer mecanismos de *checkpoint-restart*. Um estado global consistente é mantido em memória por meio da relação do *checksum*. Então, perante uma falha, a computação pode ser recuperada. No entanto, os processos corretos precisam aguardar a recuperação para continuar a execução da aplicação. A implementação FT-MPI é usada pelos autores [43].

O trabalho de Wang et al. [164] propõe uma estratégia, chamada de ABFT-*hot-replacement*, para evitar que os processos corretos tenham que parar e aguardar pela recuperação dos dados do processo falho. Quando as falhas ocorrem durante a execução da aplicação, o trabalho atualiza o processo falho com um processo redundante correspondente. O trabalho também se apoia na relação do *checksum* e também é aplicado em operação de matrizes envolvendo transformações lineares. A implementação MPICH é adaptada para lidar com as falhas em nível de aplicação. Um trabalho semelhante é o de Bosilca et al. [22] onde os nodos redundantes são usados juntamente com uma abordagem de *diskless checkpoint* (Seção 2.1.3). Em Davies et al. [48] a técnica é usada para fatoração ou decomposição LU em matrizes. Os trabalhos em [45, 96] adotam abordagens semelhantes.

Hursey e Graham [90] apresentam como as primitivas da especificação RTS podem ser empregadas para tornar uma aplicação MPI que se comunica em uma topologia de anel tolerante a falhas. O objetivo é apresentar as primitivas da RTS aos desenvolvedores de aplicações MPI interessados em aplicar a técnica ABFT. A preocupação do autor é apresentar um exemplo de como é possível desenvolver uma aplicação tolerante a falhas e não especificamente aplicar a técnica ABFT. A recuperação dos processos não é abordada no trabalho. Duas versões do código da aplicação de comunicação em anel são apresentadas: uma que não tolera falhas e outra mantém a comunicação mesmo perante falhas. Hursey e Graham discutem ainda questões como duplicação de mensagens, detecção das falhas, terminação e eleição de um novo líder.

O *Checkpoint-on-Failure* (CoF) [19] propõe uma estratégia de *checkpoint-restart* em nível de aplicação para ser usada juntamente com a técnica ABFT. A estratégia se apoia na possibilidade de as aplicações MPI serem notificadas de falhas de processos através da constante `MPI_ERRORS_RETURN` e também na técnica ABFT para restaurar os dados dos processos falhos através de métodos matemáticos. No CoF não há um *checkpoint* periódico, o mesmo é acionado quando ocorre uma falha em um processo, a partir de então: 1) os processos corretos não mais executam chamadas MPI e realizam o *checkpoint* do seu estado atual; 2) processos corretos finalizam sua execução; 3) inicia-se a execução de uma nova aplicação; 4) o *checkpoint* é recuperado nessa nova aplicação; 5) através da técnica ABFT os dados do processo falho que não puderam ser salvos são restaurados e; 6) um estado global consistente é obtido e a aplicação retoma a sua execução. O CoF possui a vantagem de não exigir *checkpoints* periódicos, porém está restrito a algoritmos que suportam a técnica ABFT. No trabalho o CoF usado em uma aplicação matemática de fatoração linear.

Outras técnicas de tolerância a falhas aplicadas a sistemas baseados em MPI incluem a migração de processos [163, 154], a predição de falhas [141, 74], a exploração do determinismo

na comunicação dos sistemas HPC [38] e a redundância como técnica para detectar e corrigir erros do tipo de computação incorreta [66].

2.2.11 Detecção de Falhas de Desempenho

Um sistema para monitoramento em tempo de execução para detecção de falhas de desempenho é proposto por Gioiosa et al. [78]. Os autores introduzem o termo "falha de desempenho" (*performance fault*) para descrever comportamentos anômalos em sistemas HPC. Os autores propõem o projeto e a implementação de um sistema de monitoramento que inspeciona continuamente a evolução das aplicações em execução com o objetivo de relatar anomalias no desempenho. Um exemplo de anomalias de desempenho seria o seguinte: um processador pode reduzir a frequência de operação do seu núcleo quando a sua temperatura sobe acima de um limiar de segurança, ou para manter o consumo de energia dentro do orçamento.

No modelo de monitoramento proposto, a execução da aplicação é confrontada com um modelo de desempenho [137] da aplicação fornecido previamente como patamar. O monitoramento é executado em nível de sistema operacional (*kernel*) através de um módulo chamado *Kernel RunTime Monitor* (kRTM). O módulo kRTM monitora a evolução das aplicações e coleta informações sobre a aplicação e o estado do sistema através de sensores. O usuário executa sob demanda a desambiguização de anomalias de desempenho e ações corretivas em nível de usuário através de um módulo chamado *User RunTime Monitor* (uRTM). Esse módulo implementa o modelo de desempenho da aplicação e as ações corretivas em resposta às anomalias no desempenho. Com base no modelo de desempenho da aplicação, o módulo uRTM define o nível esperado de desempenho. Uma anomalia de desempenho em potencial ocorre quando a informação coletada dos sensores não encontra o nível esperado de desempenho. A execução é considerada correta se os níveis de desempenho ditados pelo modelo são repetidos pela aplicação. Quando uma anomalia é detectada, o módulo kRMT notifica o módulo uRTM para que este tome as devidas ações. Por exemplo, se o kRTM notifica o uRTM que um núcleo em particular não é mais confiável, o módulo uRTM poderia migrar a *thread* em execução no núcleo falho para outro núcleo e desligar o núcleo falho. Uma desvantagem da abordagem fornecida nesse trabalho é que o usuário precisa fornecer um modelo de desempenho previamente. Os autores também não realizam a migração ou descrevem como adaptar as aplicações em tempo de execução de acordo com as anomalias identificadas.

A implementação chamada de *Adaptive MPI*(AMPI) [88] é uma implementação MPI construída sobre o arcabouço Charm++ [101]. Charm++ provê um modelo de programação paralela orientado a objetos baseado na troca de mensagens assíncronas. Seu ambiente possibilita a sobreposição de cálculo e comunicação de forma automática, balanceamento de carga, tolerância a faltas e *checkpoints* para execuções particionadas. Uma das desvantagens nessa abordagem é que o código MPI precisa ser modificado para ser transformado em código Charm++.

2.3 Conclusão

O MPI é o padrão de *facto* para o desenvolvimento de aplicações paralelas e distribuídas baseado no paradigma de troca de mensagens. Neste capítulo foram apresentados diversos esforços na tentativa de tornar os sistemas baseados em MPI tolerantes a falhas. Além disso, foram descritas as principais técnicas de tolerância a falhas aplicadas em ambientes MPI. Como foi apresentado, um dos grandes desafios dos sistemas baseados em MPI é permitir que aplicações de longa duração executem corretamente enquanto lidam com falhas em ambientes HPC de larga escala.

O padrão MPI assume que a infraestrutura subjacente é totalmente confiável. Diversos trabalhos buscam tornar aplicações HPC baseadas em MPI tolerante a falhas. É possível classificar esses trabalhos em 5 grupos de soluções: 1) os trabalhos que buscam definir primitivas e uma semântica de tolerância a falhas para o padrão MPI; 2) os trabalhos baseados na técnica de *Rollback-Recovery*; 3) as soluções que fazem uso da replicação máquina de estado; 4) as abordagens que fazem uso das propriedades do algoritmo da aplicação para recuperá-la de falhas durante a sua execução, ou seja, a técnica ABFT e; 5) os trabalhos que monitoram o sistemas a fim de detectar anomalias no desempenho da aplicação. A Tabela 2.1 apresenta os trabalhos descritos nesta seção categorizados em cada um dos grupos de solução.

Entre os trabalhos do grupo 1 estão o FT-MPI, o MPI/FT, o trabalho de Gropp e Lusk, a NR-MPI e as especificações de tolerância a falhas do MPI-Fórum, isto é, a RTS e a ULFM. O FT-MPI foi a primeira implementação a oferecer uma alternativa ao tradicional *rollback-recovery*. O desenvolvedor da aplicação escolhe o modo de recuperação do comunicador MPI e, perante falhas de processos, a aplicação continua a sua execução sem a necessidade de interromper e reexecutar a computação realizada. O MPI/FT implementa os mecanismos de detecção e correção de falhas de acordo com o modelo da aplicação paralela. No MPI/FT há métodos de recuperação predefinidos para cada modelo de execução da aplicação. Há ainda a possibilidade de usar replicação ativa ou passiva e um mecanismos de detecção de falhas baseada em *threads* que monitoram o sistema através de *heartbeats*. Gropp e Lusk foram os primeiros a defenderem a modificação e extensão do MPI a fim de suportar falhas nas aplicações mantendo a compatibilidade com versões anteriores.

As especificações RTS e ULFM oferecem um conjunto de primitivas para o desenvolvedor da aplicação adequar a sua aplicação à uma técnica de tolerância a falhas de sua preferência. A ULFM é a proposta vigente e, ao contrário da RTS, não exige um detector de falhas explícito. De fato, na ULFM a falha de um processo somente é detectada se esse processo está diretamente envolvido em uma comunicação. A partir de então é possível usar as primitivas da ULFM para recuperar o estado do comunicador MPI e continuar a execução. A NR-MPI é uma implementação de tolerância a falhas em MPI posterior as propostas do MPI-Fórum mas, que ao invés de seguir ou adotar algumas definições do MPI-Fórum, baseia-se na implementação FT-MPI. No contexto desses trabalhos ainda é possível citar aqueles que lidam com a detecção de falhas em MPI e os trabalhos que propõem algoritmos de consenso para dar suporte as primitivas de tolerância a falhas pelo MPI-Fórum. Importante notar que os autores desses algoritmos geralmente fazem uma breve comparação do seu algoritmo com o algoritmo Paxos, porém nenhum desses trabalhos o emprega no contexto de aplicações HPC baseadas em MPI.

Dentre as técnicas de tolerância a falhas para sistemas MPI, o mecanismo de *rollback-recovery* é o mais tradicional e o mais empregado. A técnica pode ser baseada em *checkpoints* ou em registro de mensagens. A primeira exige a sincronização dos processos para garantir um estado global consistente. Os protocolos de *rollback-recovery* baseados em registro de mensagens empregam tanto *checkpoints* quanto o registro de eventos não-determinísticos com o objetivo de evitar as desvantagens das abordagens coordenada e não coordenada. Basicamente, a técnica consiste em forçar a reexecução dos processos falhos a partir de determinantes armazenados em um *event logger*.

O CoCheck foi a primeira implementação de *checkpoint-restart* e de migração de processos em MPI. Implementações como o Diskless *checkpoint* e o CoF buscam eliminar a necessidade de armazenamento estável através de codificações relacionadas ao *checkpoint*. O Multi-Level Checkpointing busca diminuir a sobrecarga do *checkpoint* coordenado realizando o *checkpointing* em diferentes níveis. Os nível inferior é o mais lento e o mais resiliente. Ao contrário, os níveis superiores são mais rápidos porém menos resilientes. Entre os trabalhos

citados, ainda se destaca o Fenix. O Fenix é arcabouço que emprega *checkpoints* implícitos e a especificação ULFM para recuperar a aplicação em tempo de execução e de forma transparente. Importante notar que a maioria dos trabalhos que empregam a abordagem de registros de mensagens faz uso de um *event logger* centralizado e que não tolera falhas para armazenar os determinantes. Apesar de o protocolo O2P implementar um *event logger* distribuído o mesmo não garante a tolerância a falhas. Há ainda os protocolos híbridos que empregam tanto a abordagem baseada em *checkpoints* quanto o registro de mensagens como estratégia de tolerância a falhas.

Os trabalhos do terceiro grupo adotam a técnica de replicação. A técnica surge como uma possibilidade para fornecer alta disponibilidade aos sistemas HPC. O trabalho de Ferreira et al. emprega a replicação máquina de estado como principal mecanismo de tolerância a falhas para os sistemas HPC *exascale*. Nesse trabalho o *checkpoint-restart* surge como segunda alternativa. A replicação também é técnica empregada em trabalhos como o MPI/FT e o P2P-MPI. Ferramentas como a rMPI e a redMPI usam a replicação para substituir um processo falho por uma réplica e para detectar e corrigir erros do tipo de computação incorreta, respectivamente. Uma grande desvantagem dessa técnica está na utilização de recursos extras pelas réplicas.

A técnica ABFT, classificada no quarto grupo, move a tolerância a falhas para o código da aplicação paralela. A técnica é altamente dependente da especificação MPI. Os trabalhos que defendem a inclusão de primitivas de tolerância a falhas na norma MPI usam como uma das justificativas a possibilidade de empregar a técnica ABFT. Apesar de eficiente, umas das desvantagens da técnica é a sua aplicação em um domínio específico. Grande parte dos trabalhos se apoia na verificação de *checksums* para detectar falhas.

No quinto grupo está o trabalho de Gioiosa et al. e a implementação *Adaptive* MPI (AMPI). Enquanto Gioiosa et al. apresenta um sistema para monitoramento em tempo de execução para detecção de falhas de desempenho, o AMPI usa o arcabouço Charm++ para lidar com o balanceamento de carga e a tolerância a falhas em MPI. Para Gioiosa, falhas de desempenho são anomalias de desempenho que podem prejudicar a execução das aplicações tanto quanto falhas. Apesar de fornecer um mecanismo para detectar tais falhas, os autores não apresentam uma proposta para adaptar a aplicação perante falhas de desempenho.

Tabela 2.1: Tolerância a falhas em MPI categorizada em cinco grupos.

Abordagens		Trabalhos
Definição de primitivas e uma semântica de tolerância a falhas no padrão MPI, Detecção de Falhas e Consenso		FT-MPI [62], FT/MPI [7], Gropp e Lusk [80], NR-MPI [156], RTS [91], ULFM [18], Kharbas et al. [103], Genaud et al. [77], Buntinas [30], Ranganathan [142], Hursey et al. [92], *Early Returning Agreement* [85]
Rollback-Recovery	Baseados em checkpoint	CoCheck [154], LAM/MPI [31], LAM/MPI com checkpoint coordenado [149], Multi-level Checkpointing [127], Lazy Checkpointing[158], Diskless Checkpoint [138, 44], Fenix [75, 76]
	Baseado em registros de mensagens	MPICH-V [28], MPICH-V2 [26], Compara protocolo pessimista e otimista [29], Protocolo causal com *event logger* [24] Distinção entre eventos [25], O2P [147], O2P com *event logger* distribuído [148]
	Abordagem Híbridas	Riesen et al. citeRiesen12, Bouteiller et al.[27]
Replicação Máquina de Estado		Ferreira et al. [65], Gropp e Lusk[80], Fiala et al. [66], Bougeret et al. [23]
ABFT (Algorithm-Based Fault Tolerance)		Davies et al. [48], Du et al. [53], Hursey e Graham [90], Wang [164], ABFT para falhas fail-stop [42, 43], ABFT-*hot-replacement* [164], Bosilca et al. [22], *Checkpoint-on-Failure* (CoF),[19]
Detecção de falhas de desempenho		Gioiosa et al. [78], *Adaptive MPI*(AMPI) [88]

Capítulo 3

Diagnóstico em Nível de Sistema

O diagnóstico em nível de sistema é uma abordagem para identificar quais unidades de um sistema estão funcionando corretamente, de acordo com a sua especificação, e quais se encontram falhas [94, 126, 56]. O diagnóstico se baseia nos resultados de um conjunto de testes executados entre as unidades do sistema. O modelo PMC [139], proposto há 50 anos (1967), é a primeira proposta para diagnóstico em nível de sistema. Desde então, um grande número de modelos, abordagens e resultados teóricos foram apresentados e aplicados a um amplo e abrangente contexto, incluindo o diagnóstico de circuitos integrados e o diagnóstico de computadores multiprocessados baseados nas mais diversas topologias. Esses modelos têm sido usados para monitorar redes de computadores (cabeadas e sem fio), entre muitos outros [56].

Este capítulo apresenta uma visão geral da teoria de diagnóstico em nível de sistema e alguns dos seus principais trabalhos.

3.1 O Modelo PMC

A primeira proposta de um modelo de diagnóstico em nível de sistema foi o modelo PMC [139], cujo nome provém das iniciais de seus autores: Preparata, Metze e Chien. O trabalho aborda o problema do diagnóstico automático de falhas em um sistema com múltiplas unidades. A teoria dos grafos serve como base para o modelo proposto.

O modelo PMC assume um sistema S composto por um conjunto de n unidades independentes e não necessariamente idênticas que formam o conjunto $U = \{u_0, u_1, \dots, u_n\}$. Cada unidade u_i (nesta tese também chamada nodo i ou processo i) é uma parte bem definida do sistema e não pode ser mais decomposta para o propósito do diagnóstico. Cada unidade u_i assume um entre dois possíveis estados: falha ou sem-falha. Isto é, a unidade é considera completamente funcional ou completamente falha. O modelo PMC assume ainda que o sistema é completamente conectado (*fully connected*), isto é, cada unidade do sistema se comunica diretamente (sem intermediários) com todas as demais. Uma unidade u_i é ainda capaz de testar as outras unidades do sistema e determinar se estão corretas ou falhas. Conforme afirma Jalote [94], "o teste é o coração de um algoritmo de diagnóstico".

No modelo PMC, uma unidade é testada como um todo e o estado da unidade não muda durante o diagnóstico. Um teste envolve a aplicação controlada de estímulos e a observação da resposta correspondente retornada pela unidade testada. Um teste é definido como um "procedimento de diagnóstico" adaptado para cada sistema. Uma unidade que realiza um teste também é chamada de unidade testadora.

Uma importante definição no modelo PMC assume que uma unidade sem-falha determina corretamente o estado de outra unidade testada. E, além disso, é capaz de reportar os

resultados dos testes com precisão. Mais precisamente, com base no resultado dos estímulos aplicados, o resultado do teste é classificado como *pass* (0) ou *fail* (1). Nenhuma premissa é feita sobre uma unidade falha, isto é, os testes executados por uma unidade falha podem produzir resultados incorretos. A definição de qual unidade é a testadora e qual é a testada é chamada de atribuição de testes (*connection assigment*) e é representada por um grafo direcionado. O conjunto de resultados com todos os testes realizados é chamado de síndrome do sistema. Como exemplo, considere o modelo de um sistema apresentado na Figura 3.1. O sistema consiste de cinco unidades e a atribuição de testes é representada no grafo. A unidade u_1 testa a unidade u_2, a unidade u_2 testa a unidade u_3, e assim por diante. A síndrome do sistema é um vetor de 5 posições: $(u_{1,2}, u_{2,3}, u_{3,4}, u_{4,5}, u_{5,1})$. Se apenas a unidade u_1 estiver falha então a síndrome será da seguinte forma: $(x, 0, 0, 0, 1)$. Assim, u_5 corretamente identifica u_1 como falho. Os demais, exceto u_i, também obtêm o estado correto. O valor de $u_{1,2}$ pode ser tanto 0 ou 1, pois u_1 está falho, e é representado por x.

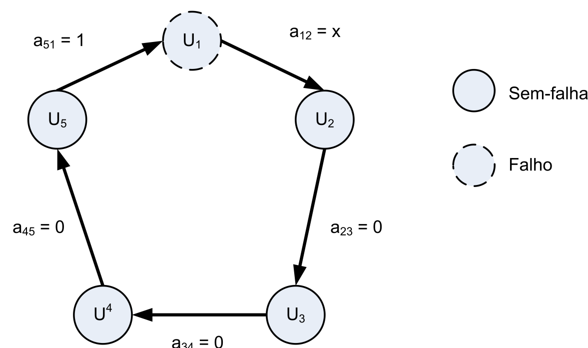


Figura 3.1: Um sistema S composto por cinco unidades [139].

No modelo PMC, cada unidade conhece somente os resultados dos seus testes. A síndrome é processada por uma entidade externa, confiável, chamada de observador central, que realiza o diagnóstico do sistema, ou seja, determina o estado de todas as unidades do sistema. Uma vez que um nodo falho produz um resultado não confiável, o observador central possui um papel determinante. Dependendo do número de unidades falhas e da atribuição de testes, é impossível diagnosticar corretamente o sistema.

Um sistema é definido como t-diagnosticável em um passo (ou *one-step t-diagnosable)* se, dada uma síndrome, todas as unidades falhas no sistema podem ser identificadas no processamento da síndrome quando o número de unidades falhas não é maior do que t. Por exemplo, o sistema apresentado na Figura 3.1 é 1-diagnosticável, mas não é 2-diagnosticável. Não é 2-diagnosticável pois se tanto o nodo u_1 e u_2 estão falhos a síndrome será $(x, x, 0, 0, 1)$. Supondo que u_2 retorne 0 então a síndrome será idêntica ao exemplo com somente o nodo u_1 falho: $(x, 0, 0, 0, 1)$.

Um sistema é definido como sequencialmente t-diagnosticável (*sequentially t-diagnosable*), também chamado de diagnóstico com reparos, se ao menos uma unidade falha puder ser identificada e reparada ou substituída para que então o teste prossiga usando a unidade reparada e, em algum momento, todas as unidades falhas do sistema sejam identificadas. O problema de encontrar o máximo valor de t é chamado de problema de diagnosticabilidade. Nesta tese, quando se fala em diagnosticabilidade se está se referindo à t-diagnosticável em um passo. O problema de determinar as unidades falhas de uma síndrome, sendo que há no máximo t unidades falhas, é chamado de problema de diagnóstico. Enquanto o problema de diagnosticabilidade se preocupa somente com o que é teoricamente possível, o problema do diagnóstico é voltado a

encontrar um algoritmo para o diagnóstico, caso o sistema seja diagnosticável, a partir de uma síndrome [94].

Hakimi e Amin [83] caracterizaram o modelo PMC e provaram que as seguintes condições são, além das condições necessárias, as condições suficientes para um sistema ser t-diagnosticável: duas unidades não se testam entre si; cada unidade é testada por no mínimo t outras e; $N \geq 2t + 1$.

3.2 O Modelo BGM

Outro importante modelo inicial na teoria de diagnóstico é o modelo BGM [6], das iniciais dos seus autores: Barsi, Grandoni e Maestrini. Este modelo é semelhante ao modelo PMC, assume o mesmo grafo de testes, porém diferentes resultados para os testes. As suas suposições básicas são as seguintes: cada teste é executado por uma única unidade; nenhuma unidade testa a si mesma e, para qualquer par de unidades u_i, u_j, a unidade u_i executa pelo menos um teste na unidade u_j. O modelo de diagnóstico é definido da seguinte forma:

- Se o testador u_i está sem-falha, o resultado do teste é 0 se u_j está sem-falha; o resultado do teste é 1 se u_j está falho;
- Se o testador u_i está falho e u_j está sem-falha, ambos resultados são possíveis;
- Se u_i e u_j estão falhos, o resultado dos testes é necessariamente 1.

No modelo BGM se o resultado do teste $a_{i,j} = 0$, isto é, a unidade i testa a unidade j como sem-falha, então é possível concluir que u_j não é falho. Porém, se $a_{i,j} = 1$ então não é possível que tanto u_i quanto u_j estejam sem-falha, mas nenhuma outra possibilidade pode ser excluída. O modelo BGM também fornece as condições suficientes e necessárias para a t-diagnosticabilidade tanto para o diagnóstico de um passo quanto para o diagnóstico sequencial. Se cada unidade é testada por pelo menos t outras unidades, e $N \geq t + 2$, então a diagnosticabilidade é no máximo $N - 2$. Para o diagnóstico sequencial, se uma unidade falha é encontrada, essa é reparada e o processo é sequencialmente repetido até que todas as unidades falhas sejam diagnosticadas e reparadas. Posteriormente, a diagnosticabilidade de grafos simétricos sobre o modelo BGM foi determinada em [4].

3.3 Modelo Adaptativo e Distribuído

A introdução do diagnóstico adaptativo por Hakimi e Nakajima [132, 84] foi um importante resultado no diagnóstico em nível de sistema. Ao invés de determinar o conjunto fixo de testes sempre executado por cada nodo, como nos modelos anteriores, no diagnóstico adaptativo os testes são executados em rodadas. Os nodos determinam os testes que devem executar em uma rodada dependendo dos testes realizados na rodada anterior. Ou seja, os testes são determinados dinamicamente.

O diagnóstico adaptativo assume um sistema S de n nodos com não mais do que t nodos falhos. No modelo os testes são definidos de forma adaptativa, repetindo o processo até que um nodo sem-falha seja identificado. Então, esse nodo sem-falha é utilizado como um nodo testador a partir do qual todos os nodos falhos são então identificados. Nakajima provou que $(n-1)+t(t+1)$ testes são suficientes para identificar todos os nodos falhos do sistema [132]. Em [84] Hakimi e Nakajima apresentam outro algoritmo adaptativo onde o diagnóstico é realizado com no máximo $(n + 2t - 2)$ testes.

Tanto no modelo adaptativo quanto nos modelos anteriores, os resultados dos testes são coletados e processados por uma entidade externa responsável por determinar o estado de todos os nodos do sistema. No diagnóstico em nível de sistema distribuído, proposto por Kuhl e Reddy [107, 108, 109], os próprios nodos sem-falha do sistema diagnosticam o estado de todos os nodos. Os nodos sem-falha executam os testes e recebem resultados de testes através dos seus vizinhos. Dessa abordagem, o algoritmo distribuído *SELF* foi proposto e embora totalmente distribuído não é adaptativo, ou seja, cada nodo tem um conjunto fixo de testes a executar. Entre as asserções do algoritmo *SELF* estão que as falhas são permanentes. Para evitar disseminar informações de diagnóstico no sistema, assume-se que o nodo n_i somente aceita a informação de diagnóstico do nodo n_j após testar n_j como sem-falha. O algoritmo *NEW-SELF* posteriormente foi introduzido por Hosseini, Kuhl e Reddy [87] como uma extensão do algoritmo *SELF*.

A diferença entre os algoritmos *SELF* e *NEW-SELF* reside no fato de este permitir a reentrada no sistema de nodos falhos que foram reparados ou substituídos e, ainda, a adição de novos nodos no sistema. O algoritmo *NEW-SELF* assume que um nodo não pode falhar e então se recuperar sem que sua falha seja detectada. Um nodo é capaz de testar os seus nodos vizinhos. Assim como no algoritmo *SELF*, para o correto diagnóstico, as informações sobre os testes são propagadas na rede. Nenhuma definição é feita sobre o comportamento dos nodos falhos e o resultado de testes proveniente de um nodo falho não é considerado. O diagnóstico correto é realizado se cada nodo é testado por pelo menos $t + 1$ nodos (t número de unidades falhas). O algoritmo requer pelo menos $n(t + 1)$ testes. O número de mensagens trocadas entre os nodos é grande: $n^2(t + 1)^2$. A partir do algoritmo *NEW-SELF*, o algoritmo *EVENT-SELF* foi proposto [100]. Esse algoritmo usa técnicas baseadas em eventos para melhorar tanto a latência do diagnóstico quanto o impacto do algoritmo sobre o desempenho da rede.

Um algoritmo de diagnóstico em nível de sistema que é tanto adaptativo quanto distribuído, chamado *Adaptive-DSD*, foi proposto por Bianchini e Bunskens [11, 10]. Assim como no modelo PMC, o modelo de sistema S é representado por um grafo completo. Os nodos assumem um de dois estados: falho ou sem-falha. Um evento é definido como a mudança de estado de um nodo. Um evento somente ocorre após os eventos anteriores terem sido diagnosticados. O sistema é representado por um conjunto de vértices $V(S)$, que representa as unidades ou os nodos do sistema; um conjunto de arestas $E(S)$, que representa os enlaces de comunicação e; um conjunto de testes $T(S)$, que representa os testes realizados. O modelo de falhas é o *crash*. Os canais de comunicação são assumidos como confiáveis e, assim como no modelo PMC, um nodo testador detecta e reporta com precisão o estado de um nodo testado.

O *Adaptive-DSD* é executado em cada nodo do sistema em *intervalos de testes* pré-definidos, por exemplo, a cada 30 segundos. Cada nodo executa o algoritmo periodicamente em intervalos de testes. Uma *rodada de testes* é definida como o período de tempo no qual todos os nodos do sistema executam os seus testes. O nodo finaliza os seus testes quando encontra um outro nodo sem-falha. Dessa forma, pelo menos um teste é executado por cada nodo durante uma rodada de teste. Todos os nodos sem-falha atingem um diagnóstico consistente em no máximo n rodadas de testes. Até $n - 1$ nodos podem falhar e ainda assim os nodos sem-falha conseguem diagnosticar o sistema.

O funcionamento do *Adaptive-DSD* ocorre da seguinte forma: periodicamente, um nodo sem-falha executa sequencialmente seus testes nos demais nodos até encontrar outro nodo sem-falha ou testar todos os nodos do sistema como falhos. Quando o nodo testador encontra um nodo sem-falha, esse nodo obtém informações de diagnóstico do nodo testado. Nesse caso, se há pelos menos dois nodos sem-falha o grafo de testes é um anel que conecta todos os nodos sem-falha. A Figura 3.2 apresenta um exemplo de execução do *Adaptive-DSD*. O nodo 0 testa o nodo 1 como falho e, dessa forma, executa um teste no próximo nodo, o nodo 2. Como o

nodo 2 está sem-falha, o nodo 0 finaliza os seus testes nesse intervalo de testes e atualiza as suas informações de diagnóstico de acordo com o nodo 2. Por sua vez, o nodo 2 realiza um teste no nodo 3. Como o nodo 3 está sem-falha, o nodo 2 para os seus testes. O nodo 3 testa os nodos 4 e 5 como falhos e o nodo 6 como sem-falha. O nodo 6 testa o nodo 7 como sem-falha. O nodo 7 testa o nodo 0 como sem-falha.

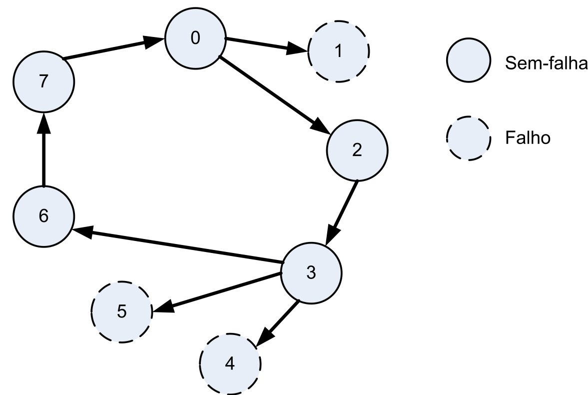


Figura 3.2: Exemplo de um Sistema S e o conjunto de testes $T(S)$ [11].

Sempre que um nodo é testado como sem-falha, o nodo testador recebe as informações de diagnóstico do nodo testado e atualiza as suas informações de diagnóstico de acordo com as informações de diagnóstico recebidas. Ou seja, através da propriedade de transitividade um nodo sem-falha pode obter de maneira confiável informações de outro nodo sem-falha e assim por diante. Vale notar que as informações de diagnóstico são transmitidas pelos nodos na direção reversa da execução dos testes.

Tabela 3.1: Vetor de testes no nodo n_2

$TESTED_UP_2[0] = 2$
$TESTED_UP_2[1] = *$
$TESTED_UP_2[2] = 3$
$TESTED_UP_2[3] = 6$
$TESTED_UP_2[4] = *$
$TESTED_UP_2[5] = *$
$TESTED_UP_2[6] = 7$
$TESTED_UP_2[7] = 0$

No algoritmo *Adaptive-DSD* cada nodo mantém um estrutura de dados para armazenar o resultado dos testes: o vetor $TESTED_UP_x$. Esse vetor é mantido em cada nodo n_x. O vetor contém n elementos, onde n é número de nodos no sistema, indexados por um identificador de nodo i ($TESTED_UP_x[i]$), onde $0 \leq i \leq n-1$. Cada posição no vetor $TESTED_UP_x$ identifica um nodo. Quando um nodo i testa outro nodo j como sem-falha essa informação é salva em $TESTED_UP_x[i] = j$. Por exemplo, $TESTED_UP_x[i] = j$ indica que o nodo n_x testou o nodo j como sem-falha se $x = i$ ou que o nodo x recebeu informações de diagnóstico de um nodo sem-falha indicando que o nodo i testou o nodo j como sem-falha, no caso de $x \neq i$. A Tabela 3.1 apresenta o vetor do nodo n_2 ($TESTED_UP_2$) para o exemplo da Figura 3.2. O valor $*$ é um

valor arbitrário e significa que o nodo n_i se encontra falho. De acordo com a Tabela 3.1, o nodo n_2 possui as seguintes informações: o nodo n_0 testou o nodo n_2 como sem-falha. O nodo n_2 testou o nodo n_3 como sem-falha. O nodo n_3 testou o nodo n_6 como sem-falha, que por sua vez testou o nodo n_7 também como sem-falha. Há um algoritmo específico em cada nodo, chamado *Diagnose*, responsável por fornecer o diagnóstico do sistema.

De acordo com Jalote [94], no algoritmo *Adaptive-DSD* o teste pode ser implementado da seguinte forma. Quando um nodo i testa um nodo j, uma *thread* é criada no nodo j. A criação dessa *thread* atesta que o escalonador de tarefas está funcional. A *thread* verifica tanto os recursos de *hardware* quando de *software*: o dispositivo de armazenamento e executa algumas operações aritméticas. Se os resultados do teste não forem fornecidos dentro de um período de tempo (*timeout*) então o nodo testado é assumido como falho. Ainda de acordo com Jalote, na implementação do algoritmo *Adaptive-DSD* descrita em [11], caso os resultados dos testes não sejam fornecidos dentro de um limite de tempo os nodos testados são assumidos como falhos. Porém, se o algoritmo do *Adaptive-DSD* classificar um nodo erroneamente como falho em uma determinada rodada poderá mudar a classificação desse nodo em uma rodada seguinte. Essa interpretação de Jalote sobre a classificação errônea de um nodo não é encontrada em outros autores.

A latência do diagnóstico é o número de rodadas de testes necessárias para detectar um novo evento, ou seja, para que todos os nodos sem-falha completem o diagnóstico do sistema. No pior caso, a latência do algoritmo *Adaptive-DSD* é n rodadas de testes. O número de testes executados em 1 rodada de teste é n. O número de mensagens de diagnóstico requeridas pelo algoritmo é n^2 para todos os nodos alcançarem o diagnóstico. O *Adaptive-DSD* foi implementado para monitorar uma rede local e os resultado práticos mostraram a sua eficiência. Para diminuir a latência do diagnóstico, Bianchini e Bunskens discutem informalmente algumas alterações no algoritmo como, por exemplo, algumas estratégias dirigidas a eventos e disseminação da informação via *multicast*.

O algoritmo *Adaptive-DSD* assume que nenhum evento ocorre antes do evento anterior ter sido diagnosticado. Quando ocorre um evento, ou seja, um nodo sem-falha se torna falho ou vice-versa, então haverá um "período de convergência". Durante esse período o correto diagnóstico por todos os nodos não é garantido.

3.4 Modelo Hierárquico

O diagnóstico hierárquico foi proposto com o objetivo de reduzir a latência do diagnóstico adaptativo e distribuído [57]. O trabalho propõe o algoritmo chamado *Hi-ADSD* (*Hierarchical Adaptive Distributed System-level Diagnosis*). O algoritmo *Hi-ADSD* é o primeiro algoritmo hierárquico de diagnóstico totalmente distribuído e adaptativo. Assim como no algoritmo *Adaptive-DSD* cada nodo sem-falha executa testes até que outra unidade sem-falha seja encontrada. Quando um nodo sem-falha é testado, o testador obtém informações de um conjunto de nodos a partir do nodo testado. Outra herança do algoritmo *Adaptive-DSD* é a definição de rodadas de testes: uma rodada de testes é definida como o período de tempo na qual cada nodo sem-falha realiza os seus testes. Um nodo somente para de efetuar os seus testes quando encontra um outro nodo sem-falha ou testa todos os nodos como falhos. O *Hi-ADSD* tem uma latência de diagnóstico de no máximo $log_2{}^2N$ rodadas de testes para um sistema com N nodos.

O algoritmo considera um sistema S consistindo de n nodos e um grafo direcionado $T(S)$ que representa os nodos de S que foram testados como sem-falha. Um nodo pode estar falho ou sem-falha. O algoritmo assume canais confiáveis e um sistema completamente conectado. O algoritmo *Hi-ADSD* assume o mesmo modelo de falhas do modelo PMC e também

parte do princípio que um nodo é capaz de testar outro nodo com precisão. Assim como no algoritmo *Adaptive-DSD*, o *Hi-ADSD* não impõe limites no número de nodos falhos para atingir o diagnóstico. Os nodos executam os testes de forma assíncrona. Quando todos os nodos estão sem-falha, o grafo de testes $T(S)$ forma um hipercubo.

O hipercubo é uma importante estrutura largamente utilizada, seja como topologia de interligação e comunicação de nodos ou para a execução de algoritmos paralelos [136]. O hipercubo apresenta importante características, como: simetria, diâmetro logarítmico e propriedades importantes para o desenvolvimento de tolerância a falhas [157, 105, 159]. Um hipercubo de s-dimensões consiste de uma rede com 2^s nodos numerados de 0 a $2^s - 1$. Cada nodo j é identificado pelo código binário $(n_0, ..., n_{2^s-1})$ do seu identificador. Uma aresta entre dois nodos existe se os seus códigos diferem em um *bit*, ilustrado na Figura 3.3.

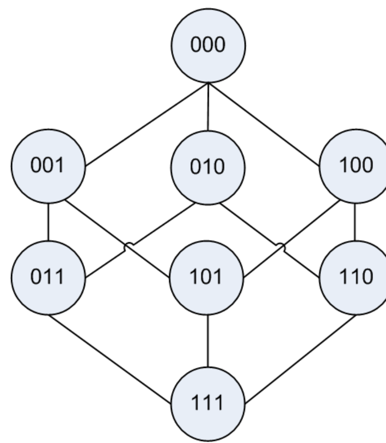


Figura 3.3: Hipercubo de 3 dimensões.

O algoritmo *Hi-ADSD* emprega uma estratégia de divisão e conquista para permitir os nodos atingirem independentemente o diagnóstico. Para tanto, os nodos são agrupados em *clusters* para o propósito dos testes. Os *clusters* são conjuntos de nodos. O número de nodos no *clusters* é sempre uma potência de 2. Inicialmente há n nodos no sistema, onde n também é uma potência de 2. Um *clusters* de n nodos $u_j, ..., u_{j+n-1}$, onde $j \mod n = 0$, é recursivamente definido tanto com um nodo, no caso de $n = 1$; ou com a união de dois *clusters*, um contendo os nodos $u_j, ..., u_{j+n/2-1}$ e o outro com os nodos $u_{j+n/2}, ..., u_{j+n-1}$. A Figura 3.4 apresenta um sistema com 8 nodos organizados em *clusters*. Inicialmente há um nodo em cada *cluster*. Posteriormente, há 2 nodos em cada *cluster*, formando 4 *clusters* ao todo. Logo após, há 4 nodos em cada um dos 2 *clusters*. Por fim, 8 nodos em um único *cluster*.

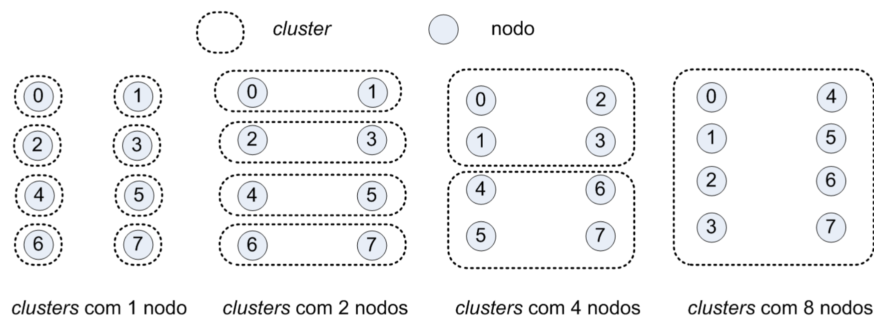


Figura 3.4: Sistema com 8 nodos organizados em *clusters* de tamanho progressivo.

Os nodos são organizados em *clusters* virtuais de tamanho progressivo. Os testes são realizados de forma hierárquica em s rodadas de testes, numeradas de 1 a $\log_2 n$, onde $s = \log_2 n$.

A cada rodada s, um nodo sem-falha testa um nodo em um *cluster* e obtém informações sobre os demais nodos naquele *cluster*. Conforme apresenta a Figura 3.5, no primeiro intervalo de testes ($s = 1$), cada nodo testa um *cluster* com um nodo. No segundo intervalo de testes ($s = 2$), o *cluster* com 2 nodos. No terceiro intervalo de testes ($s = 3$), com quatro nodos, e assim sucessivamente até que o *cluster* tenha $n/2$ nodos. Em seguida, após mais um intervalo de testes, o processo se reinicia. Se todos os nodos estão sem-falha, o número de testes executado por cada nodo é $\log_2 n$. A latência do diagnóstico é no máximo $\log_2^2 n$ rodadas de testes.

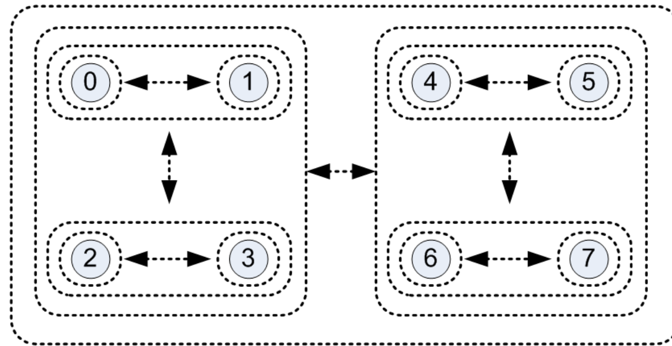


Figura 3.5: Organização hierárquica dos testes no *Hi-ADSD*.

Em cada rodada de teste um nodo sem-falha testa o próximo nodo no *cluster* s até encontrar um nodo sem-falha ou testar todos os nodos naquele *cluster* como falhos. Cada nodo executa os seus testes em um *cluster* s de acordo com a função chamada de $C_{i,s}$. A função $C_{i,s}$ informa quais são os nodos integrantes de um *cluster* s e a ordem em que os nodos desse *cluster* são testados por um nodo i.

$$c_{i,s} = (i \oplus 2^{s-1}, c_{i \oplus 2^{s-1},1}, \ldots, c_{i \oplus 2^{s-1},s-1})$$

A Tabela 3.2 apresenta um exemplo para um sistema com 8 nodos. Por exemplo, conforme apresentado, o nodo 0 quando $s = 3$ deve testar os nodos 4, 5, 6 e 7, nessa ordem, até encontrar o primeiro nodo sem-falha ou testar todos os nodos como falhos.

Tabela 3.2: $C_{i,s}$ para um sistema com 8 nodos.

s	$C_{0,s}$	$C_{1,s}$	$C_{2,s}$	$C_{3,s}$	$C_{4,s}$	$C_{5,s}$	$C_{6,s}$	$C_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

A Figura 3.6 apresenta a hierarquia de testes para oito nodos, a partir do ponto de vista do nodo 0. Quando o nodo 0 testa o terceiro *cluster*, o primeiro nodo testado é o 4. Se o nodo 4 está sem-falha, o nodo 0 copia do nodo 4 a informação de diagnóstico relativa aos nodos 4, 5, 6 e 7. Se o nodo 4 está falho, o nodo 0 testa o nodo 5, e assim por diante até encontrar um nodo do *cluster* sem-falha ou até testar todos os nodos do *cluster* como falhos.

O algoritmo *Hi-ADSD* foi implementado e integrado a um sistema de gerenciamento de rede baseado em SNMP (*Simple Network Management Protocal*) a fim de monitorar os nodos de uma rede local. Outros algoritmos baseados no algoritmo *Hi-ADSD* foram propostos, como o *Hi-ADSD with Detours* [98], o *Hi-ADSD with Timestamps* [54] e o algoritmo *DiVHA* (*Distributed Virtual Hypercube Algorithm*) [21]. O primeiro foi proposto para garantir que o

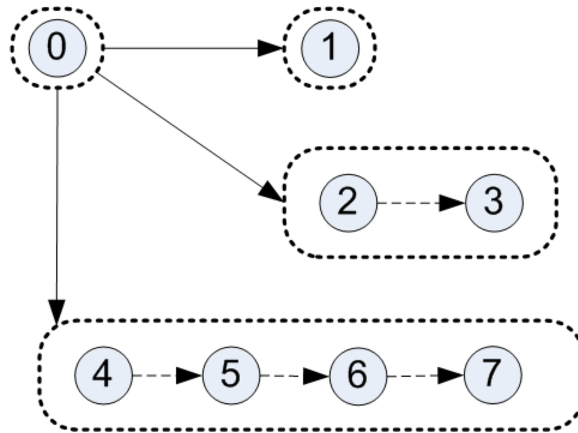


Figura 3.6: Teste executados pelos nodo 0 em um sistema com oito nodos [57].

número máximo de testes executados continue a ser uma função logarítmica. Para tanto, usa uma abordagem onde um nodo testador pode usar caminhos alternativos, chamados de desvios, para obter informações do *cluster*. O segundo algoritmo considera que há diversos caminhos a partir de um testador até um nodo testado. Ao invés de usar um único caminho, o algoritmo permite ao testador obter informações de diagnóstico de todos os outros caminhos. A informação mais recente recebida é identificada por meio de *timestamps* que são contadores modificados a cada novo evento. No algoritmo *DiVHA* os testes são assinalados aos nodos de forma que cada nodo tem apenas um testador em cada *cluster*. Para garantir esta unicidade, o algoritmo faz buscas e comparações no grafo local e determina quais os testes que cada nodo do sistema deve executar de acordo com a situação atual.

Ainda é possível citar o algoritmo VCube [55], que além de reunir as características dos algoritmos *Hi-ADSD with Detours*, *Hi-ADSD with Timestamps* e *DiVHA* também apresenta as provas de correção. O número máximo de testes executados é $nlog_2n$ em cada log_2n rodadas de testes. Ao invés de executar testes em todos os nodos de um *cluster* até encontrar o primeiro nodo sem-falha ou testar todos como falhos em uma rodada de testes, como no *Hi-ADSD*, o VCube emprega a seguinte estratégia: antes de um nodo i executar um teste no nodo $j \in C_{i,s}$, o nodo i verifica se é o primeiro nodo sem-falha em $C_{j,s}$ (observe que o índice é j). No *Hi-ADSD* diversos nodos podem executar testes no mesmo nodo de um determinado *cluster*, consequentemente o número de testes executados é quadrático. Isso não ocorre no VCube, é provado que a estratégia de testes é logarítmica.

As Figuras 3.7(a) e 3.7(b) ilustram um exemplo considerando a estratégia de testes adotada pelo *Hi-ADSD* e pelo VCube. O nodo 0 está testando o terceiro cluster (S = 3) em um sistema com 8 nodos, numerados de 0 a 7, nos quais os 4 últimos estão falhos. A Figura 3.7(a) apresenta o *Hi-ADSD* em execução e a Figura 3.7(b), por sua vez, apresenta o VCube em execução para o mesmo cenário. No *Hi-ADSD* São realizados 4 testes pelo nodo 0, pois sua função $C_{i,s}$ ($C_{0,3}$) corresponde aos nodos 4, 5, 6 e 7 e todos esses nodos estão falhos, o que o faz interromper os testes somente após testar todos os nodos do *cluster*. No VCube, a $C_{i,s}$ do nodo 0 também corresponde aos mesmos nodos 4, 5, 6 e 7. No entanto, cada um desses nodos é um elemento de j. Então, para descobrir quais nodos o nodo 0 deve testar é necessário o nodo 0 se perguntar se é o primeiro nodo sem-falha considerando cada $C_{j,s}$. O resultado será o seguinte:

- $C_{4,3} = (0,1,2,3)$
- $C_{5,3} = (1,0,3,2)$
- $C_{6,3} = (2,3,0,1)$

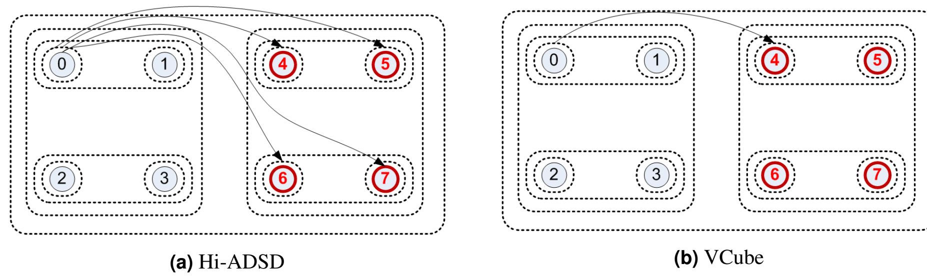


Figura 3.7: Estratégia de teste do Hi-ADSD e do Vcube sob o ponto de vista do nodo 0 na terceira rodada de testes.

- $C_{7,3} = (3, 2, 1, 0)$

Dessa forma, o nodo 0 faz o teste apenas no nodo 4, pois é o primeiro nodo sem-falha de $C_{4,3}$. Considere ainda como exemplo o nodo 1, na terceira rodada de testes e os nodos 4, 5, 6 e 7 como falhos, como no exemplo anterior. De acordo com a estratégia do *Hi-ADSD*, o nodo 1 testa os nodos 5, 4, 7 e 6. Porém, assumindo a estratégia do VCube, o nodo 1 testa apenas o nodo 5. No VCube o testador captura todas as informações de diagnóstico do nodo testado, diferentemente do *Hi-ADSD* que atualiza somente as informações referentes ao *cluster* testado. Para evitar que informações de diagnóstico inválidas sejam repassadas, o VCube usa *timestamps*, os quais são incrementados perante um novo evento. Apenas *timestamps* mais atuais que o seu são atualizados pelo testador.

3.5 Outros Modelos de Diagnóstico

Muitas outros modelos de diagnóstico têm sido propostos. O modelo PMC assume falhas do tipo permanente. Mallela e Masson [123] assumem falhas intermitentes. As falhas intermitentes afetam o comportamento do sistema somente parte do tempo [102]. Nesse tipo de falhas os nodos alternam os seus estados entre falha e sem-falha conforme o sistema evolui ao longo do tempo. O diagnóstico torna-se complexo uma vez que a avaliação de um teste executado em um nodo falho por um nodo sem-falha pode ser incorreto devido a testes insuficientes. Basicamente, para o diagnóstico de falhas intermitentes, a rotina de teste deve ser executada repetidamente antes que os nodos falhos sejam identificados. Os resultados dos testes são acumulados para permitir o diagnóstico correto do sistema. O trabalho de Serafine et al. considera falhas do tipo transientes, intermitentes e permanentes [152]. As falhas transientes diferem das falhas intermitentes por ocorrem somente por um curto período de tempo.

O modelo PMC assume que todos os nodos do sistema possuem igual probabilidade de se tornarem falhos. Os modelos de diagnóstico probabilístico são motivados pelo fato de que alguns nodos podem possuir maior probabilidade de falharem do que outros. A diagnosticabilidade é calculada com base nessa probabilidade. O trabalho em [56] afirma que existem duas abordagens probabilísticas básicas para resolver o problema do diagnóstico. A primeira é restringir o diagnóstico a um conjunto de nodos falhos, com uma probabilidade suficientemente alta. A segunda abordagem é realizar o diagnóstico para todo o sistema e então provar que o diagnóstico é correto com uma alta probabilidade. Nessa segunda abordagem é possível citar o modelo BSM [20]. O BSM aborda também as falhas intermitentes.

O diagnóstico baseado em comparações [122, 46] é uma abordagem que compara os resultados das tarefas produzidas por pares de nodos para determinar quais são os nodos sem-falha e quais são os nodos falhos. Basicamente, o modelo proposto originalmente por

Malek [122] funciona da seguinte forma. O nodo testador envia um tarefa para um par de nodos. Os nodos testados executam a tarefa e devolvem o resultado ao testador. O testador compara os resultados: se os resultados são idênticos ambos os nodos estão sem-falha; caso contrário, um ou ambos os nodos estão falhos. De acordo com [56] o modelo de falha por computação incorreta (descrito na Seção 2.1 desta tese) é o que melhor descreve o tipo de falhas assumido pelo diagnóstico baseado em comparações.

Apesar de diversos modelos e algoritmos para o diagnóstico em nível de sistema assumirem uma rede totalmente conectada, algumas propostas consideram uma rede de topologia arbitrária [143, 155, 99, 56]. O algoritmo RDZ [143] é distribuído e voltado para redes de topologia arbitrária. O algoritmo é o primeiro que permite a cada nodo, além de determinar o estado de outros nodos, identificar quais partes da rede estão inalcançáveis. Subbiah e Blough [155] definem um arcabouço teórico chamado de *bounded correctness*. Esse modelo define que um algoritmo de diagnóstico para estar correto em uma situação dinâmica de eventos deve satisfazer três propriedades: 1) deve detectar a ocorrência de eventos com latência limitada; 2) deve permitir aos nodos que se recuperam conhecer o estado dos demais nodos do sistema com um atraso limitado e; 3) não deve detectar eventos espúrios. Os autores introduzem o algoritmo chamado *ForwardHeartbeat*, que permite o diagnóstico de redes de topologia geral em situações de falhas dinâmicas. O *bounded correctness* assume um modelo síncrono de sistema.

Em Duarte et al. [99] é proposto o algoritmo DNR (*Distributed Network Reachabillity*). O DNR é um algoritmo de diagnóstico distribuído onde cada nodo de uma rede de topologia arbitrária que permite partições determina quais partes da rede são alcançáveis ou inalcançáveis. O algoritmo DNR assume falhas segundo o modelo *crash* e o modelo de falhas de temporização. No DNR, a falha de um nodo é indistinguível de uma partição na rede. O algoritmo é especificado e analiticamente provado, incluindo as provas formais no arcabouço *bounded correctness*. O algoritmo também é avaliado experimentalmente.

3.6 Conclusão

Este capítulo apresentou uma visão geral da teoria de diagnóstico em nível de sistema. O diagnóstico é uma abordagem para identificar em um sistema quais unidades estão funcionando corretamente, de acordo com a sua especificação, e quais se encontram falhas. O diagnóstico é baseado em testes executados pelas unidades do sistema.

O modelo PMC é a primeira proposta para o diagnóstico em nível de sistema. Muitas das premissas apresentadas no modelo PMC são adotadas em modelos posteriores. Outro importante modelo, semelhante ao modelo PMC, é o modelo BGM. O modelo BGM assume diferentes resultados para os testes.

O modelo adaptativo, ao contrário do modelo PMC, determina dinamicamente o conjunto de testes a ser realizado por uma unidade. No diagnóstico em nível de sistema distribuído não há uma unidade central, como no modelo PMC. Os próprios nodos sem-falha do sistema diagnosticam o estado de todos os nodos. Um algoritmo tanto adaptativo quando distribuído é o *Adaptive-DSD*.

O modelo de diagnóstico hierárquico foi proposto com o objetivo de reduzir a latência do diagnóstico distribuído e adaptativo. O algoritmo *Hi-ADSD* é o primeiro algoritmo hierárquico totalmente distribuído e adaptativo. O algoritmo VCube reduz o número de testes do *Hi-ADSD*.

Outros modelos de diagnóstico existem, como o diagnóstico probabilístico, o diagnóstico baseado em comparações e o diagnóstico para redes de topologia arbitrária.

Capítulo 4

O Grupo Dinâmico de Processos Recomendados

O Capítulo 2 desta tese apresentou padrões, técnicas e diversos trabalhos que objetivam prover os sistemas HPC baseados em MPI com a capacidade de sobreviverem a falhas e detectarem anomalias de desempenho. Conforme foi visto, apesar do vasto trabalho teórico e prático, permanece o desafio de desenvolver sistemas MPI baseados em HPC resilientes e adaptáveis. Muitas das soluções apresentadas assumem o modelo de falhas *fail-stop* onde um processo classificado como falho é excluído permanentemente do sistema. O trabalho em [78] introduz o termo “falha de desempenho” (*performance fault*) para descrever anomalias no desempenho dos sistemas HPC que podem ser indistinguíveis de falhas. Apesar de projetar um arcabouço para monitorar a ocorrência de anomalias, o trabalho não fornece um mecanismo para as aplicações se adaptarem ao detectar uma anomalia. É preciso considerar ainda que tais anomalias podem abranger inclusive a falha de um processo. Cappello et al. [37] é um dos autores que afirmam que a natureza instável dos sistemas HPC de grande escala exige não somente adaptar e otimizar as técnicas já conhecidas, mas também tornar todo o seu software tolerante a falhas e garantir que o gerenciamento e a detecção de falhas seja consistente em todo software envolvido.

Este capítulo apresenta um novo modelo para identificar e manter um Grupo Dinâmico de Processos Recomendados (*Dynamic Group of Recommended Processes* - DGRP) em sistemas HPC baseados em MPI. O modelo proposto é baseado na teoria de diagnóstico em nível de sistemas, apresentado no Capítulo 3. O DGRP inspira-se em sistemas de grupos, tais como o Isis [12]. Assim como o Isis, o DGRP é um grupo de processos que auto gerencia os seus membros. Um DGRP também pode ser visto como um *wormhole* [160] consistindo de um conjunto dinâmico de processos classificados como suficientemente adequados para executar as tarefas de um aplicação paralela MPI.

No modelo proposto para o DGRP, os processos executam testes entre si para determinar quais processos estão *recomendados* ou *não-recomendados*. O critério de recomendação é definido através do procedimento de testes empregado. O modelo permite identificar e manter um grupo dinâmico de processos recomendados em sistemas instáveis. Um processo recomendado é classificado como correto pelos demais processos. O DGRP é dinâmico no sentido de permitir o reingresso de processos ao grupo enquanto o sistema é executado. Nesse caso, o DGRP executa um algoritmo de consenso envolvendo os processos recomendados.

O modelo proposto foi implementado em um sistema MPI e aplicado para monitorar um *cluster* compartilhado. Em um *cluster* compartilhado, nodos são usados simultaneamente por diversos usuários e a carga de processamento em cada processador e/ou núcleo de processamento é desconhecida de antemão. Os resultados apresentados demonstram a eficácia da proposta. O

modelo implementado também fornece os processos recomendados para a execução do algoritmo de ordenação paralela chamado *Hyperquicksort*. O *Hyperquicksort* ordena os números em processos organizados na forma de um hipercubo virtual. O *Hyperquicksort* foi adaptado para se reconfigurar em tempo de execução, a fim de manter a ordenação sem perdas com até $n-1$ processos não-recomendados, onde n é o número total de processos. O DGRP foi implementado sobre a mais recente proposta de tolerância a falhas em MPI, a ULFM. Resultados preliminares obtidos da implementação do modelo proposto são descritos no Apêndice A.

Este capítulo descreve o modelo proposto bem como a sua implementação. Descreve-se ainda a implementação do algoritmo *Hyperquicksort*. Os resultados experimentais obtidos a partir do monitoramento do *cluster* compartilhado LCPAD/UFPR e os resultados da execução do *Hyperquicksort* sobre o DGRP também são apresentados. Os resultados obtidos foram publicados e encontram-se nas referências [32, 35, 36, 33, 34].

4.1 DGRP: Grupo Dinâmico de Processos Recomendados

Conforme apresentada no Capítulo 3 desta teste, o diagnóstico em nível de sistema é uma abordagem empregada para monitorar um sistema através da execução de testes entre os nodos do sistema. O DGRP é construído com base no modelo de diagnóstico adaptativo e distribuído. Os processos executam testes entre si para determinar se são *recomendados* ou *não-recomendados*. Em outras palavras, um procedimento de teste é definido para avaliar se o comportamento de um determinado processo é adequado, ou "bom o suficiente" para executar a aplicação disponível.

Em *clusters* compartilhados, são esperadas oscilações no desempenho devido às variações na quantidade de tarefas executadas pelos processadores, que podem ser indistinguíveis de falhas. No entanto, problemas de desempenho ocorrem não somente devido à variabilidade na carga dos processadores. Outra situação é o fato de processadores estarem aptos a reduzirem a sua frequência de operação quando a temperatura atinge um determinado limiar de segurança, ou para manter o custo energético do sistema dentro do orçamento [78]. A Figura 4.1 apresenta um exemplo onde o comportamento inicial dos processos pode ser impeditivo para executar uma aplicação.

A Figura 4.1 apresenta o desempenho de um único processador com 16 núcleos do *cluster* compartilhado LCPAD/UFPR que executa uma aplicação MPI para aproximação do número Π. Cada processo MPI executa em um dos 16 núcleos. A Figura 4.1 corresponde a 100 execuções consecutivas e, como é possível notar, há variações significantes no desempenho ao longo do tempo. Entre os intervalos 0 e 55 há uma ou outra execução mais rápida que as demais. No entanto, uma mudança brusca de desempenho ocorre a partir da sexagésima execução. O tempo de execução passa a ser muito superior ao registrado anteriormente. É possível que no início da execução a máquina estivesse ocupada com outras aplicações. Nesse caso, uma máquina do *cluster* menos ocupada poderia ter sido escolhida.

De acordo com o modelo, se o processo for considerado adequado então é classificado como recomendado. Os processos recomentados formam o DGRP e são responsáveis por executar a aplicação. Um processo que pertence ao DGRP não foi testado como não-recomendado por nenhum outro processo que compõe o DGRP. Processos testados como não-recomendados são removidos do DGRP. A Figura 4.2 apresenta um DGRP formado pelos processos 0, 2, 5 e 7.

Um processo fora do DGRP, mas testado pelos outros processos do DGRP como recomendado por ζ testes consecutivos pode reintegrar o grupo após uma rodada de consenso executada pelos processos do DGRP. A Figura 4.3 apresenta o DGRP formado pelos processos 2, 5, 6 e 7. Ao comparar o DGRP da Figura 4.2 com o DGRP da Figura 4.3, o processo 0 foi testado

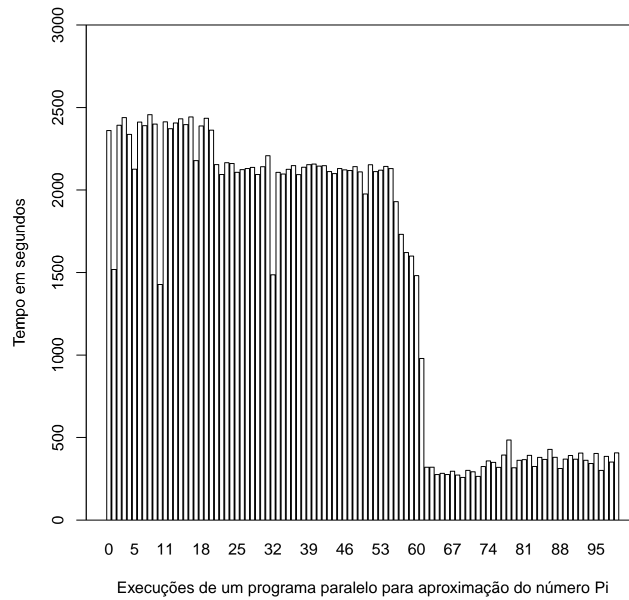


Figura 4.1: Variação de desempenho para aproximação do π - único nodo com 16 núcleos.

como não-recomendado e removido do DGRP. Por outro lado, o processo 6 foi reintegrado ao DGRP.

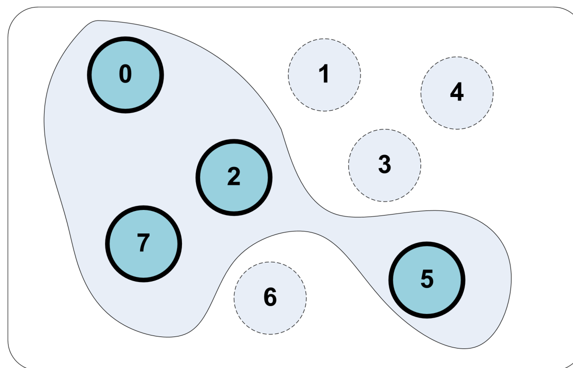


Figura 4.2: DGRP formado pelos processos 0, 2, 5 e 7.

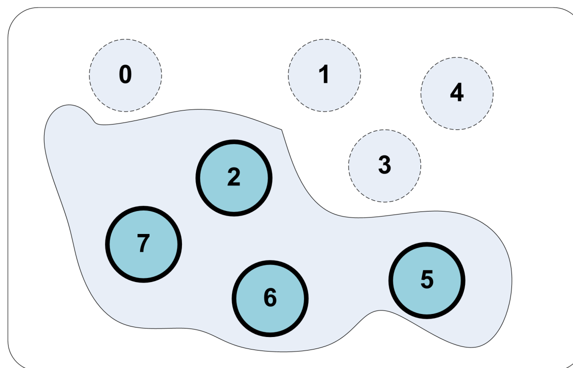


Figura 4.3: DGRP formado pelos processos 2, 5, 6 e 7.

Os processos no modelo proposto executam em *rodadas de computação*. A Figura 4.4 apresenta a execução de uma aplicação paralela (barra horizontal escura) e de forma subjacente o diagnóstico. Em uma rodada de computação, a aplicação paralela é executada até atingir uma *barreira*. A barreira é uma abstração responsável por sincronizar os processos, isto é, um processo somente prossegue com a sua execução após todos os demais atingirem a barreira. O diagnóstico executa concorrentemente, como um sistema separado para monitorar a estabilidade. Assim que os processos atingem a barreira os mesmos verificam a composição do DGRP para reatribuir as tarefas aos processos recomendados. A quantidade de tempo de uma rodada de computação pode ser configurada através da barreira. Nesse caso, é possível levar em consideração o MTBF esperado. Em se tratando de aplicações iterativas, aquelas em que executam uma computação consecutivas vezes em um laço de repetição, a barreira pode ser incluída no final de cada laço.

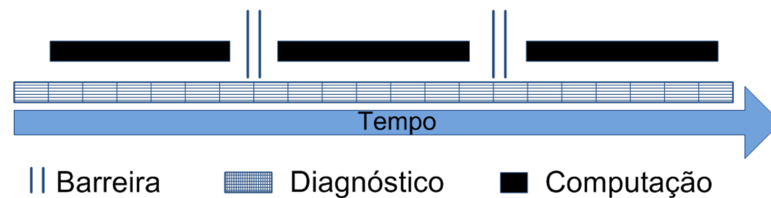


Figura 4.4: Uma rodada de computação encerra em uma barreira.

Vale a pena notar que neste modelo a comunicação não precisa ser uniforme, um processo não-recomendado que é posteriormente reintegrado ao DGRP recebe toda a informação necessária para continuar a execução.

O DGRP foi inspirado em sistemas de grupo, tais como Isis [13, 12]. A principal diferença está no fato do Isis e de outros sistemas de grupos estarem baseados na conceito de sincronia virtual. Dessa forma, oferecerem características como primitivas de difusão atômica/causal que suportam sistemas distribuídos que requerem diferentes níveis de consistência. O DGRP é simplesmente um grupo de processos com autogestão de seus membros e compartilhamento de tarefas. As tarefas da próxima rodada de computação são atribuídas com base nos resultados da rodada anterior. Um DGRP pode também ser visto como um *wormhole* [160] consistindo de um conjunto dinâmico de processos que são suficientemente bons para executar tarefas de aplicações paralelas MPI. A seguir são descritos o modelo do sistema e a implementação do DGRP.

4.1.1 Modelo de Sistema

O modelo utilizado representa o sistema como um grafo completo $G = (V, E)$; o conjunto de vértices V corresponde ao conjunto de processos e uma aresta $i, j \in E | i, j \in V$ representa a capacidade do processos i e j de se comunicarem diretamente, sem intermediários. O sistema não é síncrono, isto é, não há limites temporais tanto na transmissão das mensagens quanto na velocidade de execução dos processos. Os canais de comunicação não são confiáveis. Um processo i assume um de dois possíveis estados: *recomendado* ou *não-recomendado*. Um processo executa um procedimento de teste em outro processo para determinar o seu estado. Um processo testado que passa nos testes é classificado como recomendado, caso contrário é considerado não-recomendado. Um evento corresponde a troca de estado: recomendado para não-recomendado e vice-versa.

Os testes são executados em intervalos de testes periódicos, por exemplo 10 milissegundos ou 3 segundos. A cada intervalo de teste, um processo recomendado no sistema executa testes nos outros processos de acordo com uma atribuição de testes ou grafo de testes. O grafo

de testes $T = (V, A)$ é um grafo direcionado no qual o conjunto de vértices V corresponde ao conjunto de processos. Um arco $(i, j) \in A$ representa um teste que o processo i executou no processo j. Cada processo determina o seu intervalo de testes de acordo com o seu relógio local. Assume-se que o procedimento de teste é suficientemente completo para o testador avaliar o estado do processo testado (do seu ponto de vista). Assim, a especificação de testes frequentemente depende da tecnologia do sistema. Uma *estratégia de testes* define quais testes são atribuídos a quais testadores. Uma *rodada de testes* é definida como o período de tempo no qual cada processo recomendado no sistema executa a sua atribuição de testes. A *latência do diagnóstico* é definida como o número de rodadas de testes necessárias para que todos os processos estáveis do sistema realizem o diagnóstico de um evento.

Cada processo armazena as informações sobre o estado dos demais processos localmente. O estado é armazenado como um contador de eventos [143]. Inicialmente, assume-se cada processo como recomendado e o correspondente contador dos processos é definido em 0. Assim que um evento é detectado o contador é incrementado em 1; e assim por diante. Um contador par corresponde a um processo recomendado e um contador ímpar corresponde a um processo não-recomendado. Um processo recomendado, ao receber informações de diagnóstico de outro processo recomendado, verifica se há algum processo com o contador de estados maior do que o contador de estados mantido localmente no processo. Nesse caso, o contador de estados do processo é atualizado com a nova informação. Os contadores ajudam a identificar o quão frequentemente um processo alterna o seu estado.

O Grupo Dinâmico de Processos Recomendados (*Dynamic Group of Recommended Processes* - DGRP) é definido como segue. Se o processo $i \in$ DGRP então $\forall j \in$ DGRP $| j$ testou i e i passou no teste, isto é, i é testado como recomendado por todos os processos recomendados. Após um processo recomendado ser testado como não-recomendado por um processo $\in$ DGRP, o testador dissemina esta informação de evento para todos os demais processos no DGRP. A implementação realizada emprega a difusão confiável para disseminar os eventos que indicam que uma nova instabilidade foi detectada. Um testador aguarda até que todos os seus testes sejam executados em uma rodada para então informar aos demais, por meio de uma única mensagem de difusão confiável, os processos que testou como não-recomendados. Os processos recomendados usam a informação recebida para atualizar a composição do DGRP. Note que a informação recebida por um processo $\notin$ DGRP é ignorada.

Vale a pena notar que este modelo de diagnóstico permite que dois processos distintos obtenham resultados diferentes sobre os testes executados. Por exemplo, dois processos recomendados i e j podem testar um determinado processo k na mesma rodada de teste e obterem resultados diferentes para o estados de k. Nesse caso, como a informação de diagnóstico chega a ambos os processos, o processo k é removido do DGRP antes da próxima rodada de computação.

Um processo não-recomendado pode reingressar ao DGRP se for testado como recomendado por uma sequencia de ζ rodadas de testes. A constante ζ é configurável. Após essas rodadas, os processos do DGRP executam um algoritmo de consenso para acordar no grupo que compõe o DGRP. Nossa implementação empregou uma versão do Paxos [104, 113] como algoritmo de consenso. A seguir, descreve-se a implementação do modelo proposto.

4.1.2 Implementação do DGRP

O modelo proposto foi implementado usando Open MPI [73, 134] estendida com a implementação ULFM [69]. Cada processo i pertence a um único comunicador MPI e tem um único identificador, chamado de *rank*. O comunicador MPI é a estrutura que define o contexto de comunicação e o conjunto de processos associados a esse contexto. Os processos trocam

mensagens através de primitivas de comunicação ponto-a-ponto `MPI_Send()` e `MPI_Recv()`. Essas primitivas usam por padrão o protocolo TCP.

Cada processo i mantém em um vetor local $syndrome_i[]$ informações sobre os resultados de testes executados nos outros processos. Uma posição j do vetor, isto é, $syndrome_i[j]$ retorna o contador de eventos para o processo de *rank* j. Inicialmente todos os contadores são definidos para 0 assumindo que os processos estão recomendados. Após um teste ser executado, caso o testador detecte que um processo se tornou não-recomendado, então o contador respectivo é incrementado. Note que se $syndrome_i[j]$ é par então i considera j como recomendado, ao contrário i considera j como não-recomendado. O processo i pode obter a *síndrome* a partir de outros processos. Se $syndrome_j[k] > syndrome_i[k]$ então o processo i atualiza a sua entrada local referente ao processo k.

Um testador utiliza o seu relógio local para calcular o tempo total que leva para receber a resposta a um teste executado. Um *limiar de recomendação* é adaptativamente calculado baseado na quantidade de tempo que o processo testado leva para responder. Um processo é considerado não-recomendado se a resposta ao teste ultrapassa o limiar. Um *timeout* também é empregado e calculado adaptativamente para cada processo. Os *timeouts* são importantes para um processo não aguardar indefinitivamente por uma resposta. Note que a própria especificação MPI não considera falhas de rede. Por exemplo: se a ligação entre dois processos é rompida e um processo aguarda por uma mensagem de outro processo, esse processo pode aguardar indefinidamente. Na implementação, tanto o limiar de recomendação quanto o *timeout* são calculados usando uma versão do algoritmo do TCP proposta por Van Jacobson [93], com a adição de um fator a fim de aumentar a tolerância no atraso.

A implementação seguiu uma abordagem onde todos os processos se testam. O procedimento de teste consiste de um programa simples no qual o testador solicita ao processo testado a quantidade de números primos menores que um determinado valor. Para ser considerado recomendado, o processo testador deve responder corretamente e dentro do intervalo de tempo definido pelo limiar de recomendação. Ao final do intervalo de teste, se um testador detectou eventos no qual ao menos um processo é considerado não-recomendado, então esse processo comunica de forma confiável (utilizando *reliable broadcast*) aos demais processos.

Se um processo não-recomendado, ou seja, que não está no DGRP é testado como recomendado por ζ intervalos de testes consecutivos, os processos do DGRP executam uma rodada de consenso para possivelmente reincorporar o processo ao DGRP. Notar que o consenso é executado dentro do DGRP. É possível afirmar que o DGRP corresponde a um *wormhole* estável que preserva as propriedades temporais necessárias para executar o consenso de forma eficaz. O líder no DGRP é o processo com menor *rank*. Inicialmente, o líder recebe uma solicitação de um processo do DGRP (referente a um processo não-recomendado k) para então iniciar o consenso. Caso o líder receba mais de uma solicitação de consenso relativa ao mesmo processo somente uma é considerada. Após iniciar o consenso, o líder aguarda pelas respostas e o processo k é reintegrado se a maioria dos processos atualmente no DGRP aceitarem a reintegração do nodo. Se a reintegração do processo k ocorrer, então todos os processos incrementam o valor referente ao processo k ($syndrome_i[k]$).

4.2 Estudos de Caso: o Algoritmo *Hyperquicksort*

Apresenta-se a seguir o principal estudo de caso desenvolvido para o modelo proposto e a sua implementação. O estudo de caso se refere a implementação do algoritmo *Hyperquicksort* usando somente os processos do DGRP. Outro estudo de caso realizado preliminarmente está descrito no Anexo A. A Figura 4.5 apresenta a arquitetura do modelo proposto. O sistema de

monitoramento é implementado de acordo com o modelo proposto. O DGRP é formado através do monitoramento dos processos e a aplicação. No caso, o algoritmo *Hyperquicksort* executa usando o grupo de processos recomendados. Apresenta-se a seguir o algoritmo *Hyperquicksort* e a sua versão executando sobre o DGRP.

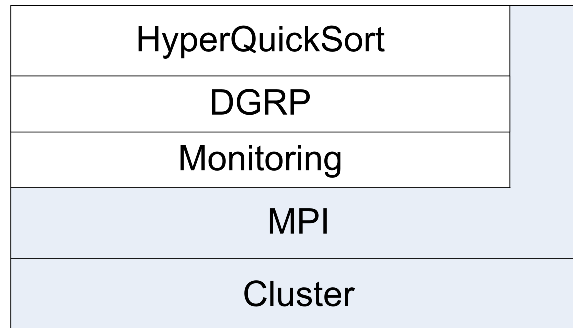


Figura 4.5: Arquitetura proposta para o sistema.

4.2.1 Algoritmo *Hyperquicksort*

Os algoritmos paralelos de ordenação são utilizados, por exemplo, em processamento de imagens, geometria computacional e teoria dos grafos [140]. O algoritmo *Hyperquicksort* [161] combina a topologia e as características do hipercubo com a estratégia de ordenação empregada no *Quicksort* [86]. O algoritmo *Quicksort* é um dos algoritmos de ordenação sequenciais mais rápidos. É intuitivamente recursivo e baseia-se na comparação de chaves. Seu tempo de execução para ordenar n elementos está estimado em $O(n \log n)$ para o melhor e para o médio caso e n^2 para o pior caso [47].

O problema da ordenação no hipercubo consiste em empregar um conjunto P de processos, $P = \{p_0, p_1, ..., p_{2^{dim}-1}\}$, para ordenar uma lista K de números, $K = \{a_0, a_1, ..., a_{k-1}\}$. Os processos são organizados na forma de um hipercubo virtual de dimensão dim. Inicialmente os $|K|$ números são divididos igualmente entre os $|P|$ processos. Cada processo é responsável por ordenar uma lista de $\frac{|K|}{|P|}$ números. A ordenação é executada em rodadas, na quais os processos p_i e p_j trocam entre si parte da sua lista, determinada com base em um *número pivô* distribuído por um dos processos. Ao final de dim rodadas de ordenação, as listas estão ordenadas de forma que em cada processo p_i o maior número é menor ou igual ao menor número no processo $p_i + 1$, onde $0 \leq i \leq |P| - 2$. O algoritmo 1 apresenta o pseudocódigo do *Hyperquicksort*.

Inicialmente, cada processo ordena localmente a sua lista (linha 5). Os processos são organizados em *clusters* virtuais de tamanho regressivo (potência de 2) a cada rodada de ordenação (linha 7). A Figura 4.6 representa o tamanho dos *clusters* para um hipercubo de 3 dimensões. Inicialmente, na primeira rodada de ordenação há oito processos agrupados em um único *cluster*. Para a segunda rodada, há dois *clusters* que agrupam quatro processos cada. Por último, há quatro *clusters* para cada dois processos. A partir de então, durante dim rodadas de ordenação o algoritmo executa os passos descritos a seguir.

Os *clusters* são formados na respectiva rodada de ordenação (linha 7). O processo com menor *rank* em cada *cluster* é definido como o *processo raiz* (linha 8). Esse processo é o responsável por distribuir um número pivô aos demais processos do seu *cluster* (linhas 9-12). O número pivô é o número médio obtido a partir da sua lista de números (linha 10). Esse número pivô tem a mesma função da chave empregada no algoritmo *Quicksort*. Após receber o número pivô, cada processo divide a sua lista em duas outras listas: uma lista com números maiores que

Algoritmo 1 ***Hyperquicksort*** (para cada processo p executando em paralelo)

```
1: dim ← log2 |P|                                          {Dimensão do hipercubo}
2: rank ← process_id                {Cada processo tem um valor único entre 0..2^dim − 1}
3: list ← K                                  {lista de números inicial em cada processo}
4: n ← |K|                             {tamanho da lista de números em cada processo}
5: quicksort(list, n)
6: while dim > 0 do
7:    cluster_i ← processes(rank, dim)
8:    processo_raiz ← root(rank, dim)
9:    if rank == processo_raiz then
10:      pivo ← median(list)
11:   end if
12:   broadcast(processo_raiz, pivo, cluster_i)
13:   create_lists(higher_list, lower_list, list, pivo)
14:   partner ← rank ⊕ 2^(dim−1)                                       {ou exclusivo}
15:   if rank > partner then
16:      send(lower_list, partner)
17:      receive(new_higher_list, partner
18:      list ← union(higher_list, new_higher_list)
19:   else if rank < parter then
20:      send(higher_list, partner)
21:      receive(new_lower_list, partner)
22:      list ← union(lower_list, new_lower_list)
23:   end if
24:   dim ← dim − 1
25:   quicksort(lista, n)
26: end while
```

o número pivô e outra lista com os números menores que o número pivô (linha 13). Então, cada processo encontra um parceiro no seu *cluster* usando uma operação de ou exclusivo aplicada no seu próprio *rank* (linha 14). O processo de maior *rank* envia a sua lista de números menores que o número pivô ao seu parceiro (que possui *rank* maior) e recebe deste a lista com números maiores que o número pivô (linhas 15-23). Após a troca, cada processo une a lista recebida com a lista que não foi trocada (linha 18 e 22) e realiza a ordenação nessa nova lista (linha 25).

A Figura 4.6 apresenta um exemplo de execução do algoritmo *Hyperquicksort* para 8 processos. A ordenação é realizada em 3 rodadas. Na primeira rodada, há um *cluster* com 8 processos e o processo raiz é o 0. O processo 0 distribui o seu número pivô aos demais processos do seu *cluster*. Os seguintes pares de processos são estabelecidos e trocam as suas listas de acordo com o número pivô recebido do processo 0: (0, 4), (1, 5), (2, 6) e (3, 7). Todos os processos reorganizam as suas listas e as ordenam localmente. Na segunda rodada de ordenação há dois *clusters* cada um com 4 processos. Os processos 0 e 4 são os processos raízes de seus respectivos *clusters*. Cada processo raiz envia o seu número pivô aos outros processos do seu *cluster*. Então as listas são trocadas entre os processos pares na rodada 2. Finalmente, na terceira rodada, todo o processo é repetido considerado os *clusters* e os processos em cada *cluster* de acordo com a terceira rodada de ordenação. A seguir, descreve-se a implementação do algoritmo *Hyperquicksort* usando somente processos do DGRP.

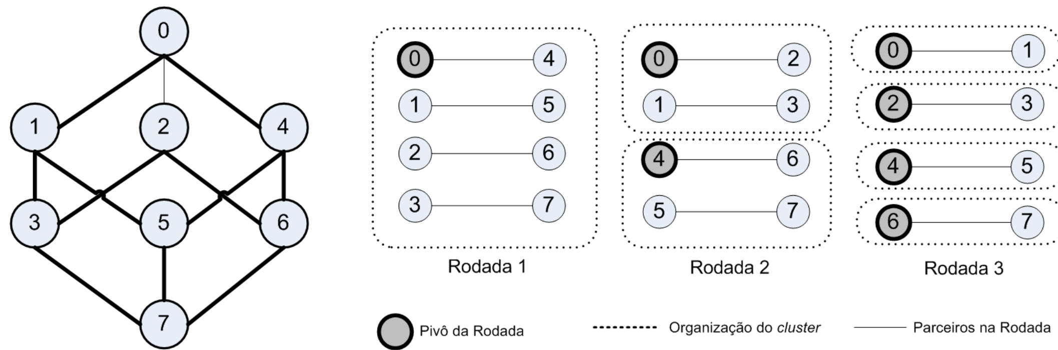


Figura 4.6: Algoritmo paralelo *Hyperquicksort* com 8 processos

4.2.2 *Hyperquicksort* sobre o DGRP

No início de cada rodada de ordenação, cada processo possui uma lista com os processos não-recomendados. A partir de então, uma função de mapeamento é invocada em cada processo. Tal função auxilia a formar os pares de processos p_i e p_j em uma rodada de ordenação e a definir qual processo recomendado assume as tarefas do processo não-recomendado. Um processo p_i pode assumir até $n-1$ processos não-recomendados (no caso de somente um processo permanecer recomendado). O processo p_i, além de executar normalmente as suas funções no algoritmo, também executa as funções do processo não-recomendado. Cada processo p_j também precisa conhecer o processo p_i que assume as funções do processo não-recomendado. Para essas duas importantes tarefas foi empregada a função $c_{i,s}$, definida no algoritmo *Hi-ADSD* [57] e apresentada na Tabela 3.2 da Seção 3.4.

A função $c_{i,s}$ considera que os processos estão organizados logicamente em um hipercubo: i representa o processo p_i e s está relacionado a uma determinada rodada de ordenação. Inicialmente $s = dim$. O símbolo $\oplus$ representa a operação binária de OU exclusivo (XOR). A Tabela 3.2 apresenta um exemplo da função $c_{i,s}$ aplicada a um hipercubo de dimensão 3, ou seja, com 8 processos. O processo p_0 quando $s = 3$ tem o seguinte resultado: 4, 5, 6 e 7.

Os pares de processos são formados de acordo com o Algoritmo 2. Conforme apresenta o algoritmo, o parceiro de um processo p_i na rodada de ordenação s é o primeiro processo recomendado na $c_{i,s}$. Portanto, para a primeira rodada de ordenação o processo p_0 deve trocar a sua lista com o primeiro processo recomendado resultante de $c_{0,3}$. Se p_4 está recomendado então p_0 e p_4 formam um par e trocam suas lista de acordo com as linhas 17-24 do Algoritmo 1 (considerando que p_0 também está recomendado). Se p_4 se encontra não-recomendado então p_0 deve trocar a sua lista com p_5 e assim por diante. Se tanto p_5 e p_6 estiverem não-recomendados então p_0 não troca a sua lista com nenhum processo nessa rodada de ordenação. Cada processo recomendado faz um *checkpoint* local da sua lista de números ao finalizar a sua tarefa na rodada de ordenação em um diretório compartilhado.

O processo que substitui um processo não-recomendado p_i em uma rodada de ordenação é escolhido de acordo com o Algoritmo 3. O substituto de um processo será o primeiro processo recomendado de $c_{i,s}$, onde inicialmente $s = 1$ e i é o identificador do processo não-recomendado. Se não há processo recomendado naquele *cluster*, s é incrementado até que um processo recomendado seja encontrado. Considerando a primeira rodada de ordenação e os processos 0 e 4, se o processo p_4 está não-recomendado então p_5 assume as funções de p_4 na respectiva rodada de ordenação ($c_{4,1} = 5$ ver Tabela 3.2). Se p_5 também está não-recomendado então p_6 substitui p_4 pois é o primeiro processo recomendado em $c_{4,2}$.

Um outro exemplo é fornecido a seguir. Suponha que o processo 2 está não-recomendado (Figura 4.6) na primeira rodada de ordenação. Essa rodada corresponde ao

Algoritmo 2 Função para encontrar um parceiro no cluster dim

```
 1: Function parceiro(rank, rodada)
 2: nodes ← c_{rank,rodada}
 3: j ← 0
 4: while j ≤ size(nodes) do
 5:    if nodes[j] ∉ non-recommended then
 6:       return nodes[j]
 7:    end if
 8:    j ← j + 1
 9: end while
10: return ⊥
```

Algoritmo 3 Função para encontrar um parceiro no cluster dim

```
 1: function substitui(rankNon-recommended, dim)
 2: s ← 1
 3: while s ≤ dim do
 4:    j ← 0
 5:    nodes ← c_{rankNon-recommended,s}
 6:    while j ≤ size(nodes) do
 7:       if nodes[j] ∉ non-recommended then
 8:          return nodes[j]
 9:       end if
10:       j ← j + 1
11:    end while
12:    s ← s + 1
13: end while
14: return ⊥
```

maior *cluster* ($s = 3$). Nessa rodada, os processos 2 e 6 forma um par quando ambos estão recomendados. Entretanto, o primeiro processo recomendado em $c_{6,3}$ é o processo 3, isto é, $c_{6,3} = 3$. O processo 3 é responsável pelo processo 2 pois é o primeiro nodo recomendado de $c_{2,1}$. O processo 3 então se torna responsável pelas tarefas do processo 2. O processo 3 então realiza a leitura da lista de números do processo 2 e interage com o processo 6 substituindo o processo 2. Vale lembrar que o processo 3 também realiza normalmente a sua tarefa com o processo 7, pois ambos estão recomendados na rodada. Ao fim de cada rodada de ordenação cada processo salva a sua lista ordenada (linha 26 do Algoritmo 1) em disco compartilhado.

4.3 Resultados

Nesta seção são apresentados dois conjuntos de resultados experimentais. No primeiro conjunto, o DGRP é empregado para monitorar um *Cluster* compartilhado. O segundo conjunto consiste dos resultados obtidos na execução da implementação MPI do *Hyperquicksort* sobre o DGRP e sobre a ULFM. Vale lembrar que a ULFM exclui definitivamente os processos falhos. Uma vez que o DGRP executa sobre a implementação ULFM, o objetivo é apresentar quanto de sobrecarga o DGRP acrescenta à implementação quando os processos não-recomendados não são reintegrados ao grupo.

Os experimentos foram executados no *Cluster* do LCPAD (Laboratório Central de Processamento de Alto Desempenho) da UFPR. O *Cluster* é multiusuário e compartilhado. Os

experimentos foram executados em 18 máquinas, cada uma com 32 núcleos de processamento (Intel(R) Xeon(R) CPU E5-2670) executando a 2,60 GHz. Cada máquina possui 128 Gbytes de RAM e 20480 Kbytes de cache. A rede é uma *Gigabit Ethernet*. O código MPI foi escrito em liguagem C. A biblioteca MPI utilizada foi a Open MPI versão 1.7 estendida com a ULFM [128].

4.3.1 Monitoramento através do DGRP

Os resultados apresentados nesta subseção são referentes à implementação do DGRP e a sua aplicação no monitoramento do *Cluster* compartilhado enquanto programas de diferentes usuários são executados. O objetivo é identificar, em tempo de execução, as máquinas e/ou processos recomendados e não-recomendados. Cada máquina executa um único processo. As máquinas e processos são associados aos identificadores 0 a 17. O intervalo de testes foi definido em 1 segundo. O procedimento de teste consiste do cálculo dos números primos entre 1 e 1000. Como mencionado anteriormente, o limiar de recomendação é calculado através do algoritmo do *timeout* do TCP multiplicado por uma constante; inicialmente, esse valor é definido em 8 segundos. Conforme os testes são executados, o limiar de recomendação se aproxima ao tempo médio em que um processo leva para responder a um teste.

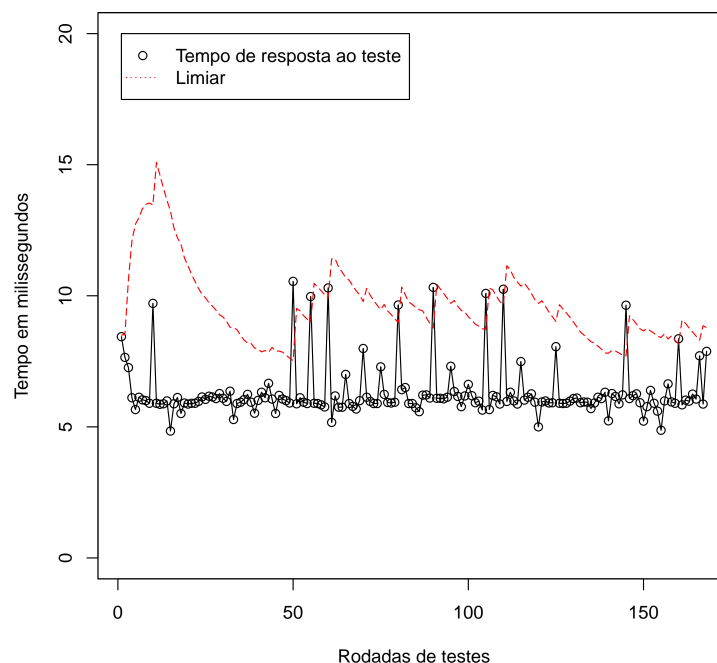


Figura 4.7: Processo 1 testa o processo 0.

Durante o monitoramento, um processo é classificado como recomendado se responde corretamente a um teste, dentro do intervalo do limiar de recomendação. Em caso contrário, esse processo é classificado como não-recomendado. Para todos os experimentos, ζ (o número de vezes que um processo deve ser testado como recomendado antes de poder reingressar ao DGRP) foi definido em 5 ($\zeta = 5$).

A Figura 4.7 apresenta o resultado para o processo 1 testando o processo 0 por 150 intervalos de testes consecutivos. Neste experimento particular, o limiar de recomendação foi definido para muito próximo do tempo de execução de um teste. O gráfico apresenta o tempo

para completar o teste. A linha tracejada apresenta o limiar de recomendação correspondente. É possível concluir através dos vários picos na linha do tempo do teste que o processo 0 muito frequentemente leva mais tempo para responder a um teste do que o esperado. Conforme é possível perceber, o limiar de recomendação é atualizado de acordo com a variação do tempo de teste, isto é, o limiar é adaptativo. Em alguns pontos, o tempo do teste excede o limiar. Nesses pontos, o processo 1 classifica o processo 0 como não-recomendado.

Os resultados a seguir são referentes ao monitoramento de 18 nodos do *cluster* do LCPAD. Foram realizadas 30 mil rodadas de testes. Para este e os próximos experimentos o limiar de recomendação foi definido em 2,5 vezes o valor do *timeout* calculado pelo algoritmo do TCP. Neste experimento o processo 11 permaneceu recomendado em todas as rodadas de testes. O processo 2 terminou o experimento como recomendado, mas esse foi o processo que mais frequentemente se tornou não-recomendado, alternando o seu estado 11 vezes. A Figura 4.8 apresenta o ponto de vista do processo testador 11 sobre o processo 2. A Figura 4.9 apresenta o ponto de vista do processo testador 6 sobre o processo testado 2. É possível perceber que o processo 2 apresenta comportamento similar nas duas Figuras. O processo 6 identificou que o processo 2 se tornou não-recomendado por 7 vezes. Por outro lado, o processo 11 não identificou qualquer mudança no estado do processo 2.

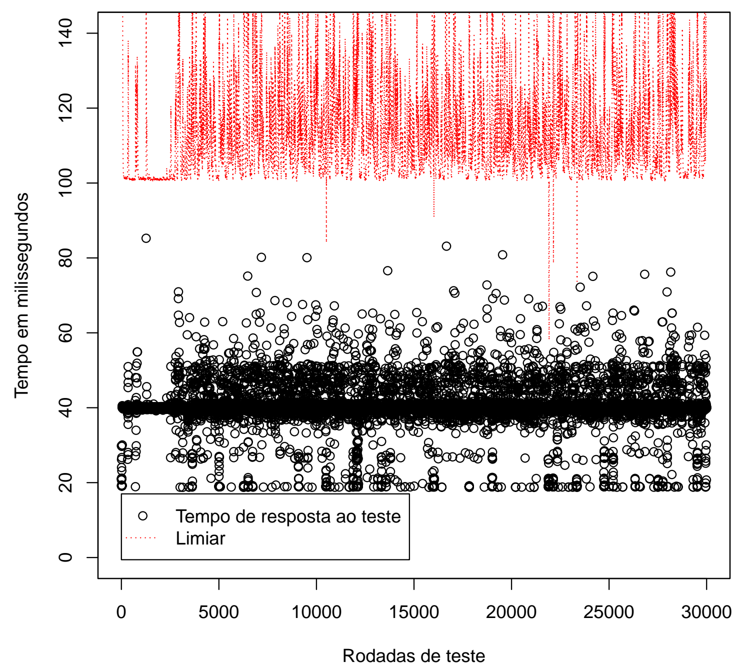


Figura 4.8: Processo 11 testa o processo 2.

A Figura 4.10 apresenta o ponto de vista do processo 0 (testador) sobre o processo 11 (testado). Ao comparar os resultados obtidos nas Figuras 4.9 e 4.8 com os da Figura 4.10 é possível notar que nas primeiras duas Figuras o processo 2 mostra muito mais variações nas respostas aos testes. A Figura 4.10, por sua vez, apresenta o processo 11 com menos variações, um comportamento mais estável.

A partir dos experimentos, foi possível perceber que no *cluster* do LCPAD as máquinas com os menores identificadores estavam quase sempre com a totalidade dos seus núcleos ocupada executando tarefas. A Tabela 4.1 correlaciona a carga no *cluster* com o contador de estado de

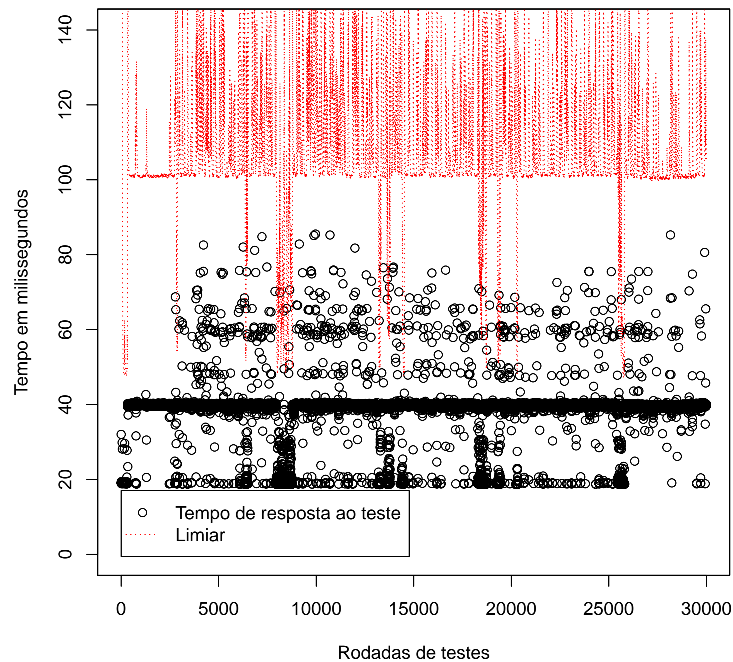


Figura 4.9: Processo 6 testa o processo 2.

cada processo ao fim do experimento. Cada máquina possui 32 núcleos. É possível constatar que a máquina que executa o processo 2 ($máquina_2$) tem todos os seus núcleos executando tarefas. Isso pode explicar o porquê do processo 2 apresentar o comportamento apresentado nas Figuras 4.8 e 4.9. É possível concluir que os processos que se tornaram não-recomendados menos vezes executaram nas máquinas com menos tarefas.

A Figura 4.11 é uma ampliação da Figura 4.9 e apresenta o processo 6 testando o processo 2 a partir do intervalo de teste 13.200 até o 13.700. É possível ver que o processo 6 detecta duas vezes que o processo 2 se tornou não recomendado (círculos azuis na Figura 4.11). Na rodada de teste 13.226 a resposta a um teste foi de 62,39 milissegundos e o limiar de recomendação foi igual a 48,42 milissegundos. Na rodada de teste 13.641 a resposta a um teste foi de 68,16 milissegundos e o limiar de recomendação foi igual a 62,1 milissegundos.

Cada vez que o processo não-recomendado 2 é testado como recomendado o processo 6 incrementa o respectivo contador. A Figura 4.12 apresenta a variação desse contador. Quando o contador alcança a constante $\zeta = 5$ o consenso é acionado para possivelmente permitir o processo 2 reintegrar o DGRP. O processo 6 é um dos responsáveis por invocar o algoritmo Paxos, enviando uma solicitação ao líder, por 9 vezes para mudar o estado do processo 2 (torná-lo estável). A partir do intervalo de testes 5.000 até o 8.500 o estado do processo 2 é alterado por 5 vezes.

A Figura 4.13 apresenta a composição do DGRP durante os 30.000 intervalos de testes. Uma ampliação apresentando a composição do DGRP entre os intervalos de testes 1.200 e 1.300 também é apresentada. No início todos os 18 processos estão no DGRP. Por diversas vezes 1 único processo foi removido do DGRP. Um momento maior de variação é encontrado na rodada de testes 1.271, quando o DGRP consiste de apenas 4 processos. Logo após, o grupo de processos recomendados se recupera. Na rodada de testes 1.298 o DGRP é formado por

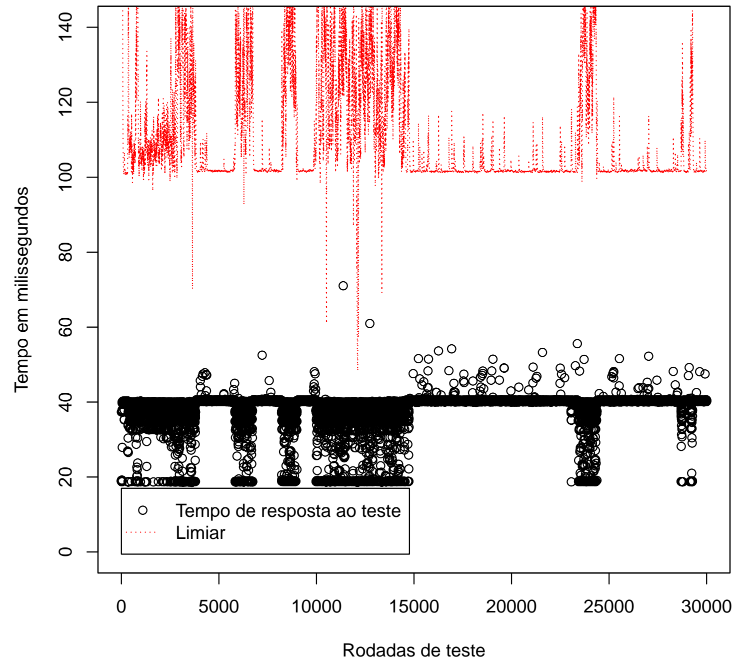


Figura 4.10: Processo 0 testa o processo 11.

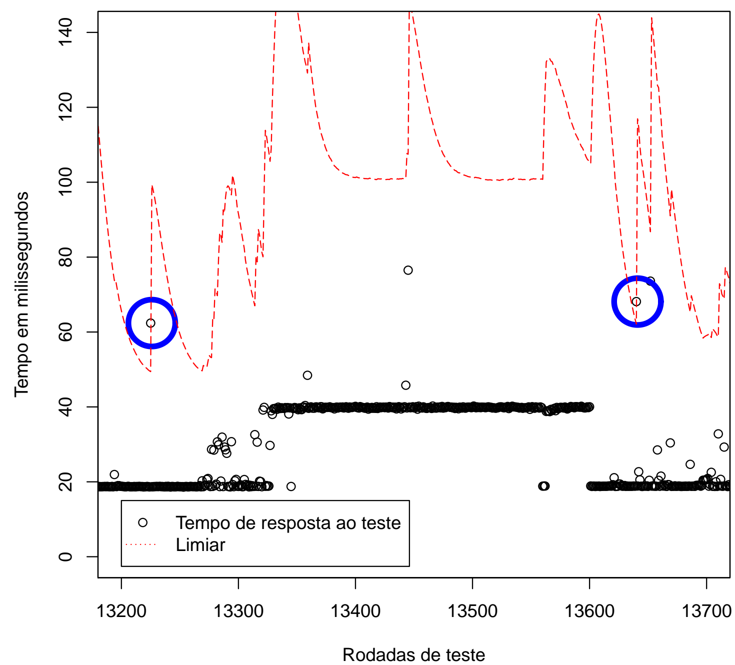


Figura 4.11: Processo 6 testa o processo 2.

8 processos. Uma situação similar acontece novamente na rodada de testes 13.149, quando o DGRP consiste somente de 3 processos.

Tabela 4.1: Carga no *cluster* e resultado dos testes.

$máquina_i$	# tarefas em execução	contador de estado
$máquina_0$	9	6
$máquina_1$	33	10
$máquina_2$	33	22
$máquina_3$	32	14
$máquina_4$	32	12
$máquina_5$	32	12
$máquina_6$	3	2
$máquina_7$	2	4
$máquina_8$	3	6
$máquina_9$	4	6
$máquina_{10}$	8	6
$máquina_{11}$	5	0
$máquina_{12}$	32	6
$máquina_{13}$	2	2
$máquina_{14}$	1	2
$máquina_{15}$	31	12
$máquina_{16}$	31	8
$máquina_{17}$	31	14

4.3.2 Desempenho do *Hyperquicksort* sobre o DGRP e ULFM

O *Hyperquicksort* foi executado através do DGRP, que executa acima da ULFM, e executado somente através da ULFM. Na Figura 4.14, o *Hyperquicksort* sobre o DGRP é executado para permitir que os processos deixem de participar temporariamente de uma ou mais rodadas de ordenação, conforme a variação do DGRP. Ao final da ordenação haverá a mesma quantidade de listas de números originalmente distribuídas e ordenadas conforme preconiza o algoritmo *Hyperquicksort*. Na execução *Hyperquicksort* sobre a ULFM, processos classificados como falhos são excluídos permanentemente da computação. O DGRP também foi executado para não considerar a recuperação de processos. Nesse caso, não há o algoritmo de consenso envolvido, pois os processos não reingressam ao núcleo estável. Os resultados dessa versão do DGRP são apresentados com o nome de DGRP sem recuperação.

A seguir, descreve-se uma visão geral da implementação realizada através da ULFM. Vale lembrar que, por padrão, a detecção de falhas na ULFM é local, isto é, somente os processos que efetivamente se comunicarem com o processo que falhou irão detectar a falha. No entanto, a ULFM permite, através das suas primitivas, implementar uma abordagem global de detecção de falhas. Dessas forma, uma abordagem de detecção global foi implementada da forma descrita a seguir.

Um função similar a função `MPI_Barrier` foi implementada para detectar todos os processos que falharam em determinada rodada de computação. Se ao menos uma falha de processo ocorrer, essa função retorna um código de erro do tipo `MPI_ERR_PROC_FAILED`. A partir de então o comunicador MPI é revogado (`MPI_Comm_revoke()`), o que torna o comunicador inválido. Dessa forma, todos os processos chamam uma função de acordo (`MPI_Agree()`). A função `MPI_Comm_agree` executa uma operação coletiva entre os processos corretos no comunicador. Essa função notifica os processos que o comunicador está in-

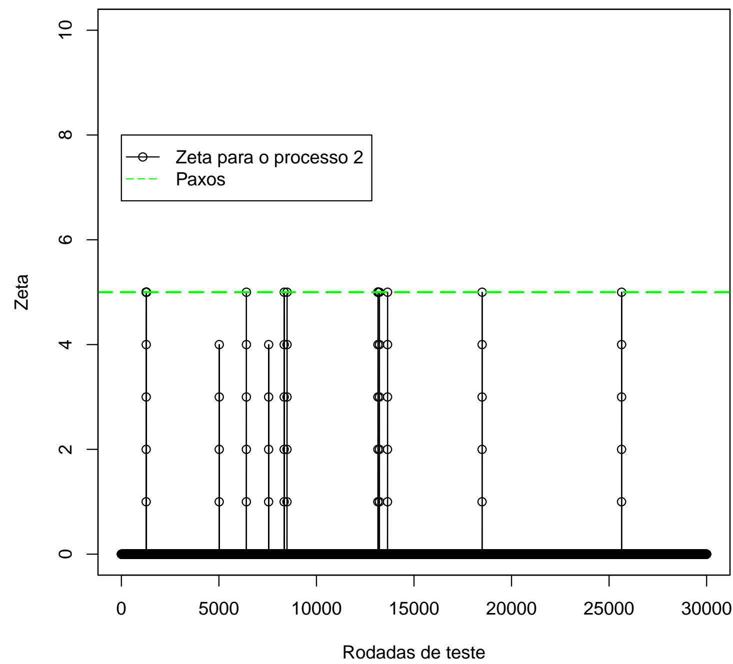


Figura 4.12: Zeta para o processo 2.

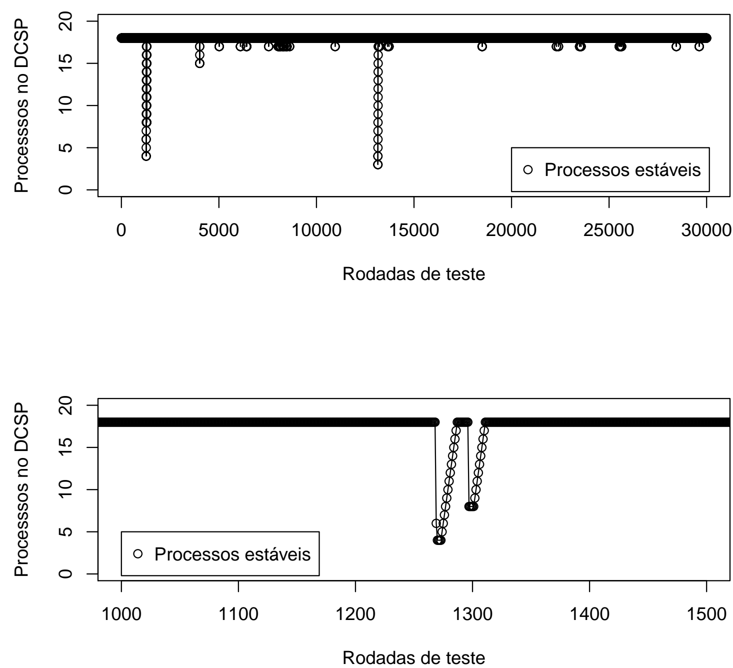


Figura 4.13: Variação na composição do DGRP.

válido. As rotinas `MPI_Comm_failure_ack()` e `MPI_Comm_failure_get_acked()` são empregadas para identificar quais processos dentro do comunicador estão falhos. Após isso, a rotina `MPI_Comm_shrink()` permite a aplicação criar um novo comunicador, eliminando

todos os processos que falharam em um comunicador inválido. Essa primitiva é coletiva e executa um algoritmo de consenso para garantir que todos os processos tenham a mesma visão no novo comunicador. De acordo com Bland et al. [18] o algoritmo de consenso implementado na ULFM é o proposto por Hursey et at. [92].

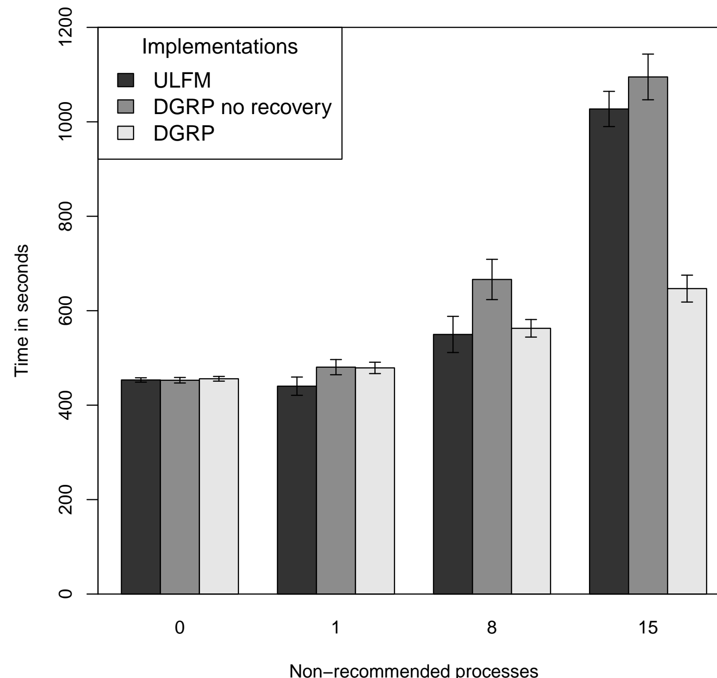


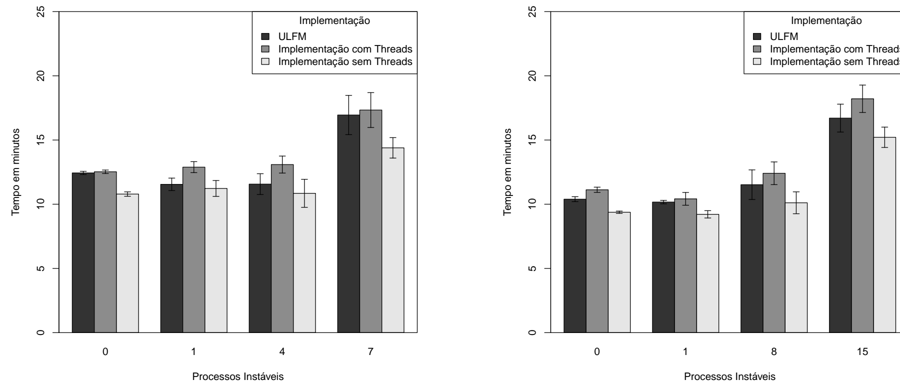
Figura 4.14: Desempenho de 16 processos ordenando 1.024 x 10^6 números inteiros.

A Figura 4.14 apresenta o desempenho do *Hyperquicksort* executando sobre 16 processos para ordenar 1 bilhão de números inteiros. O objetivo não é apresentar o *speedup*, mas a capacidade do algoritmo de se manter em execução mesmo com alterações no grupo de processos recomendados e com exclusão dos processos falhos (no caso da ULFM). Os resultados apresentam um intervalo de confiança de 95%. Os seguintes cenários foram executados: 1) somente processos recomendados; 2) somente 1 processo se torna não-recomendado; 3) metade dos processos se tornam não-recomendados ($n/2$); 4) somente 1 processo se mantém recomendado. A instabilidade foi injetada aleatoriamente durante a execução, isto é, um nodo por vir a se tornar não-recomendado (ou falho no caso da ULFM) em qualquer rodada de ordenação. Como dito anteriormente, o desempenho do *Hyperquicksort* é avaliado sobre a ULFM, sobre o DGRP e sobre o DGRP sem recuperação. Conforme apresenta a Figura 4.14, a barra mais escura se refere ao desempenho da ULFM. A barra do meio apresenta resultados para a implementação sobre o DGRP sem recuperação e a barra mais clara apresenta o desempenho do DGRP.

A sobrecarga imposta pelo DGRP é em média 9% quando compara-se o *Hyperquicksort* usando somente a ULFM com o *Hyperquicksort* usando o DGRP sem recuperação. Como era de se esperar, o tempo de execução quando os processos podem reingressar ao DGRP é menor do que o tempo de execução do DGRP sem recuperação. Isso porque mesmo que um processo não participe de uma rodada de ordenação, posteriormente o mesmo pode vir a participar da computação.

Os resultados a seguir se referem a uma versão preliminar do modelo proposto e apresentado em Camargo et al. [35] e Camargo e Duarte [32]. Nessa versão, o algoritmo

Hyperquicksort não contava com o reingresso de nodos classificados como não-recomendados. Há também uma versão do DGRP implementada com o uso de *threads* e outra implementada sem o uso de *threads* que foram executados em uma implementação Open MPI sem a ULFM. No entanto, nesse caso o DGRP não suporta falha de processos. Os experimentos foram executados no sistema operacional *Linux Kernel* 3.2.0 com 8 e 16 processadores AMD *Opteron* com 2.400 MHz. Para todos os resultados apresentados cada experimento foi repetido 30 vezes; são apresentadas a média e intervalo de confiança de 95%.



(a) 8 nodos ordenando 1.024 x 10^6 números.

(b) 16 nodos ordenando 1.024 x 10^6 números.

Figura 4.15: *Hyperquicksort* sobre o modelo proposto e a estratégia ULFM.

O desempenho do *Hyperquicksort* é avaliado de acordo com as três abordagens implementadas: executando sobre modelo de diagnóstico com e sem o uso de *threads* e executando sobre a abordagem de detecção global de falhas via ULFM. Cada abordagem é avaliada em 4 cenários: a) não há processo não-recomendados; b) 1 processo não-recomendado; e) 50% de processos não-recomendados e; d) $N - 1$ processos não-recomendados. A instabilidade e as falhas (no caso da ULFM) ocorrem aleatoriamente antes do início de cada rodada de ordenação.

Os gráficos da Figura 4.15(a) e 4.15(b) apresentam os resultados para o cenário com 8 e 16 processos, respectivamente, ordenando 1.024 x 10^6 números. Observa-se que o tempo de ordenação com 16 nodos é ligeiramente inferior ao tempo com 8 nodos. O cenário com 16 nodos realiza a ordenação com uma rodada a mais de ordenação, o que exige mais troca de dados entre os processos. Por outro lado, porções menores de dados diminuem o tempo de ordenação de dados em cada processo.

Nessa versão do *Hyperquicksort* os dados dos processos não-recomendados são incorporados pelos processos recomendados, o que diminui o custo da troca de dados entre os processos, ou seja, o custo dos `MPI_Sends` e `MPI_Receives`. A exceção é no cenário onde há $N - 1$ falhas, onde o custo de um processo recomendado assumir e incorporar os dados dos processos não-recomendados se mostrou superior.

Dentre as três abordagens comparadas, o melhor desempenho foi observado quando o *Hyperquicksort* fez uso do modelo de diagnóstico na sua versão sem o uso de *threads*. O segundo melhor desempenho foi apresentado pela abordagem implementada na ULFM. Porém, essa abordagem ficou próxima da abordagem que fez uso do modelo de diagnóstico na sua versão com as *threads*.

O custo de sincronização entre os processos é observado na abordagem ULFM, a qual exige a sincronização dos processos. Em contrapartida, a abordagem de diagnóstico com *threads* tem o custo da troca de contexto entre as execuções das *threads*, sem falar que o último nível de *threads* disponível é o `MPI_THREAD_SERIALIZED` que impede chamadas MPI simultâneas.

A abordagem sem o uso de *threads* apresentou melhor desempenho pois não exige a mesma sincronização de processos da estratégia implementada na ULFM e não apresenta o custo das *threads*.

4.4 Conclusão

Neste capítulo, foi apresentado um novo modelo baseado na teoria de diagnóstico em nível de sistemas para recomendar um grupos de processos para executar uma aplicação HPC baseada em MPI. O modelo, chamado de DGRP, mantém um núcleo de processos recomendados. A recomendação é baseada em testes executados sobre os processos que elimina tanto os processos falhos quanto os processos que apresentam responsividade lenta. Um DGRP é definido como um grupo dinâmico autogerido de processos recomendados que emprega o monitoramento e reconfiguração em tempo de execução de forma a permitir à aplicação manter a sua execução mesmo enquanto processos se tornam não-recomendados e são removidos do DGRP. Um processo não-recomendado pode posteriormente reingressar ao DGRP se passar em uma sequência de testes e após uma rodada de consenso executada pelo processos do DGRP. A execução da aplicação é organizada em rodadas de computação que usa barreiras para sincronizar os processos e assim obter uma nova visão do DGRP com o objetivo de atribuir as tarefas da aplicação aos processos do DGRP. Como estudo de caso foi implementado o algoritmo *Hyperquicksort*. Uma versão tolerante a falhas do algoritmo *Hyperquicksort* também foi implementada e apresentada. Resultados experimentais apresentam a execução do *Hyperquicksort* em um *cluster* compartilhado executando somente através da ULFM, a mais recente proposta de implementação de tolerância a falhas em MPI, e executando sobre o DGRP. O DGRP foi implementado em cima da especificação ULFM . Os resultados demonstram que o modelo proposto é eficiente e efetivo.

Capítulo 5

Um Protocolo Pessimista para Registro de Mensagens Baseado em um *Event Logger* Distribuído e Tolerante a Falhas

Conforme apresenta o Capítulo 2, *Rollback-recovery* é uma técnica de tolerância a falhas tradicionalmente empregada em sistemas de alto desempenho e de longa duração [59]. O objetivo da técnica é restaurar o sistema a um estado consistente após uma falha [61]. Para tanto, o estado de um processo é salvo periodicamente durante a sua execução e, perante uma falha, reiniciado a partir de um estado anterior, reduzindo assim a quantidade de trabalho perdido. O *checkpoint* é a ação de armazenar periodicamente o estado de um processo sem-falha em execução. Protocolos de registro de mensagens (Seção 2.1.3) são uma categoria de *rollback-recovery* que, ao contrário da abordagem coordenada, não exigem a sincronização dos *checkpoints*. Além disso, na sua abordagem pessimista apenas o processo que falha é reiniciado. A abordagem garante uma recuperação consistente a partir de *checkpoints* tomados independentemente em cada processo [119, 120, 25].

Protocolos de registro de mensagens assumem que todos os eventos não-determinísticos que um processo executa podem ser identificados e a informação necessária para reaplicar cada evento durante a recuperação pode ser registrada em tuplas chamadas de *determinantes* [61]. Considerando que os determinantes são salvos em uma entidade confiável, um processo em recuperação pode recriar deterministicamente o estado anterior à falha ao reaplicar os determinantes na sua ordem original. Consequentemente, uma tarefa crucial nos protocolos de registro de mensagens é salvar e recuperar de forma confiável os determinantes sem penalizar o desempenho da aplicação.

O componente responsável por armazenar de forma confiável os determinantes é chamado de *event logger*. O *event logger* recebe os determinantes dos processos da aplicação, armazena-os localmente, e notifica os processos da aplicação. Protocolos de registro de mensagens normalmente assumem que o *event logger* é uma entidade centralizada que não tolera falhas [29, 147, 24]. De fato, a falha de um *event logger* centralizado pode paralisar os processos da aplicação, uma vez que esses não mais conseguem salvar os determinantes.

Este capítulo descreve um *event logger* distribuído e tolerante a falhas que tem desempenho comparável ou superior à abordagem centralizada, bem como um protocolo pessimista de registro de mensagens para interagir com o *event logger* proposto. Em particular, o *event logger* replicado não requer recursos extras (nodos físicos) em comparação a um *event logger* centralizado. Os *event loggers* replicados podem ser hospedados nos mesmos nodos dos processos da aplicação. Além disso, o *event logger* distribuído pode tolerar um número configurável de

falhas. Quando configurado para suportar uma única falha, o *event logger* distribuído requer o mesmo número de mensagens e de passos de comunicação que um *event logger* centralizado para armazenar um determinante.

Duas implementações do *event logger* proposto foram realizadas. Ambas são baseadas no algoritmo Paxos (Seção 2.1.2). A primeira implementação se apoia em uma configuração tradicional do Paxos, chamadas de Paxos Clássico. A segunda configuração é chamada de Paxos Paralelo. As duas implementações são comparadas com uma implementação de um *event logger* centralizado. Um protocolo pessimista de registro de mensagens é implementado em MPI para interagir com o *event logger* proposto. Entre as principais características do protocolo proposto estão as seguintes: a distinção entre eventos determinísticos e não-determinísticos em MPI; o emprego da abordagem de *checkpoint* não-coordenado; e a recuperação automática da aplicação perante falhas de processos. Os *event loggers* foram avaliados usando as aplicações LU e MG do *NAS Parallel Benchmark*, a aplicação AMG (*Algebraic MultiGrid*) e o algoritmo paralelo de Gusfield. O algoritmo de Gusfield também foi empregado para avaliar a recuperação. Resultados demonstram que o *event logger* baseado na abordagem Paxos Paralelo tem desempenho superior ao *event logger* centralizado e que o protocolo proposto realiza a recuperação da aplicação eficientemente.

5.1 Um Protocolo para Registro de Mensagens Baseado em Consenso

Nesta seção apresentamos o modelo do sistema, a descrição do protocolo proposto e do serviço de *event logger* propriamente dito, bem como a estratégia emprega na recuperação da aplicação.

5.1.1 Modelo de Sistema

Considere um sistema representado por um grafo completo $G = (V, E)$, o conjunto de vértices V corresponde ao conjunto de processos e uma aresta $(i, j) \in E \mid i, j \in V$ representa a habilidade dos processos i e j de se comunicar diretamente, sem intermediários. O sistema é assíncrono com detectores de falhas. Um processo pode estar em um de dois estados: falho ou sem-falha. O modelo de falhas é o de parada com recuperação (*crash-recovery*). Os canais de comunicação são confiáveis e FIFO. Destaca-se que mensagens recebidas concorrentemente de diferentes emissores são entregues em qualquer ordem. Uma execução é uma sequência alternada de eventos e estados de processos. O efeito de um evento no estado de um processo conduz o processo a um novo estado. A ordem parcial entre os eventos é obtida através da relação *happened-before* [106]. Os eventos são classificados em determinísticos e não-determinísticos.

Os eventos não-determinísticos de um processo são identificados e armazenados em determinantes. Um determinante de um processo i é formado pelo seu identificador, o identificador do emissor da mensagem e o tipo de evento (por exemplo, se uma recepção ou verificação de mensagem recebida), ou seja: $det_i \leftarrow (det_{id}, id_j, event_type)$. Inicialmente, det_{id} é definido em 0 e é incrementado a cada novo determinante armazenado. Os processos têm acesso a um serviço de *event logger* para salvar, remover e recuperar os determinantes. Ao salvar um determinante, o *event logger* retorna ao processo o identificador do determinante. Um processo i incrementa contadores $sent_i[j]$ e $recv_i[j]$ ao enviar ou receber uma mensagem do processo j, respectivamente. Os contadores são representados por vetores de tamanho $|V|$, onde cada posição no vetor é indexada pelo identificador do processo correspondente e inicializada em

0. O protocolo assume a abordagem *sender-based* [97]: toda vez que o processo i envia uma mensagem msg ao processo j, uma cópia da mensagem é mantida na memória volátil de i. O identificador da mensagem é formado pelo id do emissor e a sequência de envio, ou seja $msgs_i \leftarrow (id_j, sent_i[j], msg)$.

5.1.2 Descrição do Protocolo

O algoritmo 4 descreve o protocolo de registro de mensagens. Cada operação de envio de mensagem realizada pela aplicação incrementa o contador de mensagens enviadas, ou seja, cada vez que o processo i envia uma mensagem a j, $v_sent_i[j]$ é incrementado (linhas 70 a 73). Além disso, i mantém em sua memória uma cópia da mensagem enviada a j (linha 72). Por sua vez, sempre que o processo i recebe uma mensagem de j, i verifica se o evento de recebimento é não-determinístico. Caso seja, a mensagem é submetida ao *event logger*. Logo após, o contador de mensagem recebidas é incrementado (linhas 74 a 79).

Periodicamente, usando o seu relógio local, cada processo realiza um *checkpoint*, o qual é armazenado em um dispositivo confiável (linhas 14 a 19). O *checkpoint* inclui o estado do processo, as suas mensagens enviadas mantidas em memória volátil, o identificador do último determinante submetido ao *event logger* e os contadores de mensagens enviadas e recebidas. O estado do processo é definido pelas variáveis do processo (linhas 11 a 13). Ou seja, o *checkpoint* em um processo i tem o seguinte formato: $ck_i \leftarrow (p_i, msgs_i, det_{id}, sent_i[], recv_i[])$. Apenas o último *checkpoint* do processo é mantido. A cada *checkpoint* em i, todos os determinantes de i mantidos até então no *event logger* podem ser removidos (linha 18). Além disso, uma cópia do contador de mensagens recebidas por i é realizada (linha 17) com o objetivo de posteriormente informar ao processo j que as mensagens de i mantidas na memória de j podem ser apagadas.

Periodicamente, o processo i comunica ao processo j quais mensagens de i podem ser apagadas na memória de j (linhas 20 a 24). Cada processo i pode apagar as mensagens de j mantidas em sua memória tal que $sent_i[j] \leq recv_j[i]$ ao receber de j o seu contador de mensagens recebidas (linhas 25 a 30).

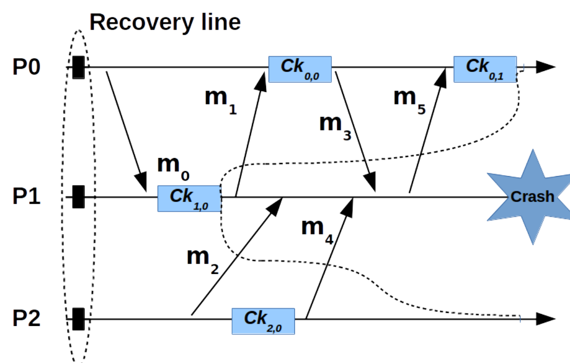


Figura 5.1: *Checkpoint* não coordenado e um linha de recuperação.

A recuperação de um processo ocorre da seguinte forma. Ao detectar a falha de j, o processo i solicita a criação de um novo processo para substituir j e envia ao processo j os contadores de mensagens recebidas e enviadas referentes a j (linhas 31 a 38). Os contadores são enviados para que j orquestre a sua recuperação. A partir da criação de um novo processo, inicia-se a recuperação (linhas 42 a 52). O processo i (supondo que i está em recuperação) lê o seu *checkpoint*, obtém os seus determinantes do *event logger* a partir do último determinante armazenado (linhas 59 a 69) e solicita a cada processo j que reenvie as mensagens de i a partir

da última mensagem recebida por i, ou seja, $\forall j \in V$, i envia $recv_i[j]$ a j (linhas 50 a 52). Então, j reenvia msg tal que $msg.id > recv_i[j]$ (linhas 53 a 58). As mensagens $sent_i[j] \leq recv_j[i]$ não serão reenviadas, pois j as recebeu antes da falha de i.

O processo i então reinicia sua computação a partir do seu *checkpoint*. Cada evento determinístico encontrado será reproduzido. Os eventos não-determinísticos serão substituídos pelas informações contidas nos determinantes lidos do *event logger*. Por exemplo, considere a falha de P_1 (Figura 5.1), apresentada previamente na Seção 2. Antes de falhar, os determinantes det_1, det_2 e det_3 correspondentes às mensagens m_2, m_4 e m_3 foram salvos no *event logger*. Em sua execução normal, P_1 aguarda mensagens de quaisquer emissores. Mas, em recuperação, e de posse dos determinantes det_1, det_2 e det_3 recuperados do *event logger*, P_1 aguarda as mensagens vindas de P_2, P_2 e P_0, nessa ordem. Uma vez que cada mensagem é reaplicada, a mensagem m_5 é então gerada e armazenada em memória volátil. A partir de então, a execução de P_0, P_1 e P_2 segue normalmente.

5.1.3 O Serviço Proposto de *Event Logger* Baseado em Consenso

O serviço de *event logger* proposto é baseado em consenso. O consenso é uma abstração fundamental na computação distribuída tolerante a falhas. O consenso pode ser usado para construir um serviço de *event logger* usando a abordagem de máquina de estado [150]. A replicação máquina de estado regula como os comandos devem ser propagados e executados pelas réplicas para que o serviço seja consistente. No serviço proposto, os comandos são solicitações para salvar um determinante, propagados e executados pelas réplicas do *event logger*. A propagação dos comandos possui dois requisitos: (i) cada réplica sem-falha deve receber cada comando e (ii) duas réplicas quaisquer não podem discordar na ordem dos comandos recebidos e executados. Se a execução é determinística, então as réplicas alcançarão o mesmo estado e produzirão a mesma saída ao executar a mesma sequência de comandos.

A replicação máquina de estado pode ser implementada como uma série de instâncias de consenso, onde a i-ésima instância de consenso decide sobre o i-ésimo comando (ou lote de comandos) a ser executado pelas réplicas [113]. A replicação máquina de estado e o consenso fornecem a abordagem principal para assegurar que as réplicas são consistentes apesar de falhas [67].

Dois procolos baseados em Paxos Clássico e Paxos Paralelo são propostos para construir o *event logger* replicado. A Figura 5.2 apresenta as três abordagens.

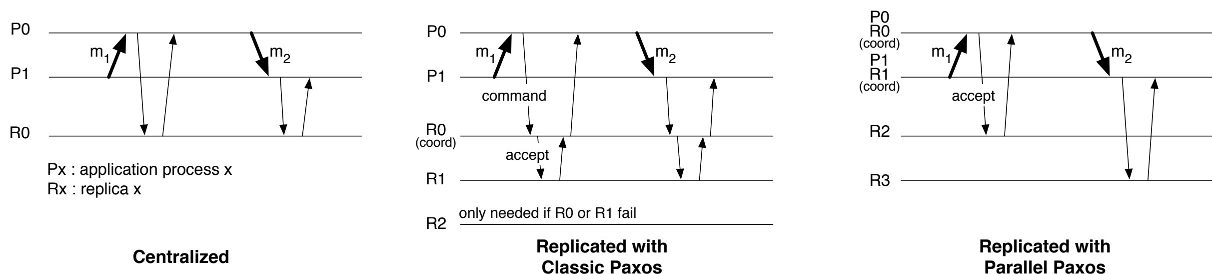


Figura 5.2: Três implementações de um *event logger*.

Os processos da aplicação são representados por $P0$ e $P1$. As mensagens m_1 e m_2 são mensagens trocadas entre os processos da aplicação. Cada mensagem recebida pelo processo da aplicação gera uma requisição ao *event logger*, representado por Rx em cada abordagem. A abordagem centralizada conta apenas com um único *event logger* (R_0) e assim não tolera falhas. As abordagens baseadas em Paxos contam com 3 *event loggers* replicados e assim podem tolerar

Algoritmo 4 Registro de Mensagens (pseudocódigo para cada processo i em paralelo)

```
1:  Initialization
2:     v_sent[] ← 0                        {contador de mensagens enviadas para cada processo j}
3:     v_recv[] ← 0                        {contador de mensagens recebidas de cada processo j}
4:     v_recv_stable[] ← 0                 {cópia de v_sent[] ao realizar um checkpointing}
5:     last_id ← 0                         {identificador do último log armazendo no event logger }
6:     msg_sent[] ← NULL                   {cópia de cada mensagem enviada ao processo j}
7:     failures[] ← 0                      {processos detectados como falhos}
8:     state ← NULL                        {estado do processo i}
9:     last_sent[] ← 0                     {última mensagem enviada a j antes da falha }
10:    determinants[] ← 0                  {última mensagem enviada a j antes da falha }
11: procedure process_state(var)
12:    state ← state ∪ var
13: end
14: procedure checkpoint()
15:    ck ← state ∪ msg_sent[] ∪ last_id ∪ v_sent[] ∪ v_recv[]
16:    save(ck)
17:    copy(v_recv[], v_recv_stable[])
18:    el_remove(last_id)
19: end
20: procedure garbage_collection()
21:    for each process j do
22:       send(v_recv_stable_i[j], j)
23:    end for
24: end
25: upon receive(v_recv_stable_j[i], j)
26:    for each msg ∈ msg_sent[] do
27:       if (msg.dest = j) and (msg.dest.id ≤ v_recv_stable_j[i]) then
28:          delete(msg, msg_sent[])
29:       end if
30:    end for
31: upon detect_failure()
32:    for each process j detected as failed do
33:       failures[j] ← FAILED
34:    end for
35:    replace_failed_by_new(failures[])   {exclui processos falhos e os substitui por novos}
36:    for each process j ∈ failures[] do
37:       send(v_recv_i[j], v_sent_i[j], j)   {ultima mensagem recebida e enviada de j antes da falha de j}
38:    end for
39: upon receive(v_recv_j[i], v_sent_j[i], j)
40:    last_received[j] ← v_recv_j[i]      {última mensagem que j recebeu de i}
41:    last_sent[j] ← v_sent_j[i]          {última mensagem que j enviou para i}
```

uma falha (isto é, $f = 1$). Considera-se que os coordenadores do Paxos executaram previamente a primeira fase do protocolo e podem seguir com a segunda fase ao receber um comando.

O Paxos clássico baseia-se na replicação máquina de estado clássica. Os processos da aplicação são *proposers* e as réplicas de *event logger* são os *acceptors* e os *learners*. Cada

Algoritmo 4 Registro de Mensagens (pseudocódigo para cada processo i em paralelo)

```
42: upon recovery()
43:    read(ck)
44:    state ← ck.state
45:    v_sent ← ck.v_sent
46:    v_recv ← ck.v_recv
47:    last_id ← ck.last_id
48:    msg_sent ← ck.msg_sent
49:    el_recovery(last_id)
50:    for each process j  do
51:       send(v_recv_i[j], j)
52:    end for
53: upon receive(v_recv_j[i], j)
54:    for each msg ∈ msg_sent[] do
55:       if (msg.dest = j) and (msg.dest.id > v_recv_j[i]) then
56:          send(msg, j)
57:       end if
58:    end for
59: procedure recovery_determinants()
60:    deteminant ← NULL
61:    dets ← el_request_since(last_id)
62:    for each d ∈ dets do
63:       determinant.sender ← sender(d)
64:       determinant.iter ← iter(d)
65:       determinant.type ← type(d)
66:       determinants.seq ← determinants.seq + 1
67:       determinants ← determinants ∪ determinant
68:    end for
69: end
70: upon application_send(msg, j)
71:    id ← (j, v_sent[j])
72:    msg_sent[] ← (id, msg)
73:    v_sent[j] ← v_sent[j] + 1
74: upon application_receive(msg, j)
75:    if msg = NON_DETERMINISTIC event e then
76:       el_save(id, j, e.type)
77:       id ← id + 1
78:    end if
79:    v_recv[j] ← v_recv[j] + 1
```

processo da aplicação submete comandos ao coordenador, um processo entre os *acceptors* para registrar os determinantes. O coordenador recebe o comando, executa o Paxos para registrar o comando em um quórum de réplicas e responde ao processo da aplicação (ver Figura 5.2). Por exemplo, um determinante é recebido pelo coordenador (R_0), que também é um *acceptor*. O coordenador executa diretamente a segunda fase do Paxos. Ao receber a resposta de um quórum de *acceptors*, no caso R_0 e R_1, o consenso é finalizado e R_0 responde ao processo $P0$. Em "boas execuções" (isto é, na ausência de falhas de processos) um determinante é registrado em quatro passos de comunicação e $2f + 2$ troca de mensagens ponto-a-ponto. Em contraste, um *event*

logger centralizado pode registrar eventos após dois passos de comunicação e duas trocas de mensagens.

No segundo protocolo, Paxos Paralelo, há um sequência separada de execuções do Paxos associada a cada processo da aplicação. Isso significa que cada processo possui seu próprio conjunto de réplicas o que permite certas otimizações. Primeiro, uma vez que cada processo tem sua própria sequência de execuções do Paxos, o processo não compete com outros processos da aplicação nas execuções de Paxos e assim não há necessidade de um único coordenador que recebe requisições de diferentes processos. Em boas execuções, o processo é o único *proposer* em sua sequência de Paxos. Uma segunda otimização é a seguinte: ao usar diferentes conjuntos de réplicas, o desempenho não é mais limitado pelo o que o coordenador e os *acceptors* podem suportar. Terceiro, é possível co-alocar os processos da aplicação e o *acceptor*-coordenador no mesmo processador. Este esquema pode registrar um determinante após dois passos de comunicação e $2f$ mensagens.

		Centralizado	Paxos Clássico	Paxos Paralelo
Atrasos		2	4	2
Mensagens	geral	2	$2f + 2$	$2f$
	$f = 1$	—	4	2
Recursos (nodes)	geral	$n + 1$	$n + f + 1$	$n + f$
	$f = 1$	—	$n + 2$	$n + 1$
Desempenho		limitado	limitado	escalável
Tolerância a Falhas		bloqueante	f falhas	f falhas

Tabela 5.1: Diferentes implementações de um *event logger*. Valores para o cenário de melhor caso com um quorum de *acceptors*/réplicas; n: número de processos da aplicação; f: número de falhas toleradas pelos *acceptors*.

A Tabela 5.1 apresenta uma comparação das três estratégias descritas para implementar um *event logger*. Um *event logger* implementado com Paxos Paralelo apresenta o mesmo número de passos de comunicação de um *event logger* centralizado, com a vantagem de tolerar um número configurável de falhas e apresentar desempenho escalável. Quando configurado para tolerar uma falha ($f = 1$), o número de trocas de mensagens é o mesmo de um *event logger* centralizado. Além disso, para economizar recursos, *acceptors* não co-alocados com os processos da aplicação podem ser hospedados em um mesmo nodo físico. Por exemplo, um único nodo pode hospedar todos esses *acceptors*. Nesse caso, o Paxos Paralelo usa o mesmo número de nodos requerido pelo abordagem centralizada.

Ao se recuperar de uma falha, um processo de aplicação deve recuperar todos os seus determinantes. Na abordagem centralizada, o processo da aplicação contacta o *event logger*. Com a abordagem replicada, isso é feito contactando um quórum de *acceptors*.

5.2 Implementação

Esta seção descreve as implementações do protocolo de registro de mensagens pessimista em MPI e dos *event loggers*. A comunicação entre os processos em MPI é FIFO e implementada através do protocolo TCP. A implementação do protocolo de registro de mensagens proposto se apoia na especificação ULFM (Seção 2.2.5), pois, tradicionalmente, as aplicações MPI abortam toda a aplicação se um único processo falhar. A Figura 5.3 apresenta o esquema que o MPI em conjunto com a ULFM emprega para lançar novos processos ao detectar

um processo falho. Inicialmente, o comunicador MPI contém um conjunto de processos, por exemplo, quatro processos (Figura 5.3 (a)). A recuperação do comunicador inicia quando um dos processos detecta a falha de outro processo. Por exemplo, o processo 2 detecta a falha do processo 1 ao se comunicar com o mesmo (Figura 5.3 (b)). A partir de então o processo 2 revoga o comunicador (`MPI_Comm_revoke()`)(Figura 5.3 (c)). Conforme descrito na Seção 2.2.5, a operação de revogação invalida o comunicador MPI. Então o comunicador é reparado, o processo falho é excluído e os processos que permanecem no comunicador têm o seu *rank* modificado através da operação `MPI_Comm_shrink()` (Figura 5.3 (d)). O lançamento de novos processos ocorre através da operação `MPI_Comm_spawn()` (Figura 5.3 (e)). Essa operação cria dois grupos de comunicadores MPI (Figura 5.3 (f)). Os processos do novo comunicador recebem o seu novo *rank* do processo 0 do grupo original (Figura 5.3 (g)). Então o comunicador original reatribui os *ranks* dos seus processos (Figura 5.3 (h)). Por fim, os grupos são unidos através da operação `MPI_Intercomm_merge` e o comunicador MPI restabelecido com o mesmo número de processos anterior à falha (Figura 5.3 (i)). No entanto, é importante destacar que o novo processo inicia a sua computação a partir da primeira linha de código, ou seja, o novo processo não conhece o estado dos demais processos e tampouco o estado do processo que está substituindo. Vale notar que o novo processo pode ser iniciado em uma máquina diferente do processo que falhou. O MPI distribui os processos nas máquinas da rede de acordo com um arquivo de configuração. Dessa forma, o novo processo pode ser lançado na próxima máquina disponível no arquivo de configurações.

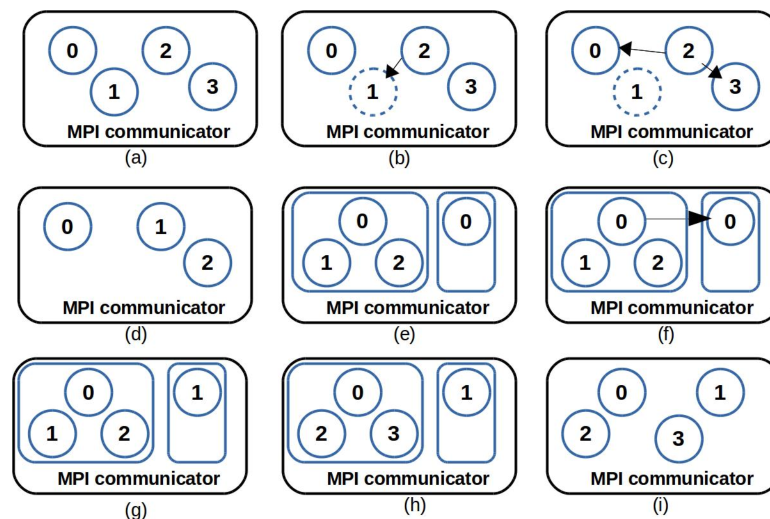


Figura 5.3: Recuperação do comunicador MPI e lançamento de um novo processo [75].

A implementação do protocolo de registro de mensagens foi encapsulada em uma biblioteca para facilitar o seu uso por uma aplicação MPI. A seguir destacamos brevemente as principais primitivas. A primitiva `framework_init()` inicia o interceptador de eventos (descrito mais adiante) e as estruturas de dados necessárias, como os contadores de mensagens enviadas e recebidas. Essa primitiva também define parâmetros como, por exemplo, a periodicidade do *checkpointing* e o seu local de armazenamento. No entanto, a sua principal função é classificar o processo MPI como um processo original ou um novo processo em recuperação. A primitiva `aloc_data()` define quais variáveis da aplicação serão incluídas no *checkpointing*. Se o processo MPI for um novo processo em recuperação, a primitiva `recovery()` é responsável por restaurar o *checkpoint* do processo, solicitar os seus determinantes do *event logger* e orquestrar a recuperação conforme definido na Seção 4.1.1. A primitiva `chkp()` é responsável por realizar o *checkpointing* do processo. A primitiva `garbage_colletor()`

realiza a limpeza periódica das mensagens antigas mantidas em memória. Cada mensagem enviada é copiada para a memória e armazenada em uma tabela *hash*. A biblioteca khash[1] foi usada como implementação de uma tabela *hash*. A implementação do protocolo foi realizada em linguagem C.

A interface PMPI [129] é utilizada para modificar o comportamento das primitivas MPI durante a recuperação e interceptar as chamadas MPI de forma transparente para o desenvolvedor. Dessa forma, é possível inserir código antes e depois de cada chamada MPI. Os eventos não-determinísticos são detectados de acordo com o refinamento proposto em [25]. Toda primitiva de recepção, como `MPI_Recv`, que aguarda uma mensagem de uma fonte qualquer (`MPI_ANY_SOURCE`) gera um evento não-determinístico. Assim também acontece com primitivas que verificam se há uma mensagem pronta para ser recebida, como a `MPI_Probe`. Além das primitivas de recepção, as primitivas não bloqueantes que inspecionam se o recebimento da mensagem foi concluído como, por exemplo, `MPI_Waitsome`, `MPI_WaitAny`, `MPI_Test` também geram eventos não-determinísticos.

A Tabela 5.2 apresenta o comportamento das primitivas MPI durante a recuperação e apresenta o formato do determinante gerado como cada evento não-determinístico. Cada primitiva, apresentada na primeira coluna, acompanha uma breve descrição da sua semântica e do seu comportamento padrão (segunda coluna). O comportamento durante a recuperação é descrito na terceira coluna. Na quarta coluna é apresentado o tipo de comunicação e se a primitiva é bloqueante ou não bloqueante. Uma primitiva bloqueante aguarda a conclusão da sua tarefa para então prosseguir. Uma primitiva não bloqueante realiza a sua tarefa de forma assíncrona. Por exemplo, a primitiva `MPI_Irecv` é não bloqueante. Após a execução de uma chamada `MPI_Irecv`, a aplicação prossegue executando outras tarefas. É possível verificar se a mensagem correspondente à chamada `MPI_Irecv` foi recebida, por exemplo, através de uma primitiva do tipo `MPI_Test`. Na quinta coluna o formato do determinante é apresentado: o primeiro parâmetro é o tipo de determinante, o segundo parâmetro descreve a origem da mensagem recebida no caso de uma recepção de mensagem e o terceiro parâmetro apresenta o número de invocações da primitiva até a sua conclusão. Por exemplo, a primitiva `MPI_Iprobe` possui uma variável do tipo verdadeiro ou falso que é definida para verdadeiro somente quando a mensagem é realmente recebida. Da mesma forma, uma primitiva `MPI_Test` somente terá sua variável definida para verdadeira quando o recebimento ou envio de uma mensagem for efetivado. A implementação PMPI foi realizada em C.

A recuperação inicia automaticamente quando uma das primitivas de comunicação MPI como, por exemplo, `MPI_Recv`, retorna um valor diferente de `MPI_SUCCESS`. Tal fato indica que uma das primitivas MPI retornou uma mensagem `MPI_ERR_REVOKED`, que significa que o comunicador se tornou inválido, ou `MPIX_ERR_PROC_FAILED`, que significa que a falha de um processo ocorreu. Uma vez que todas as primitivas MPI são interceptadas usando a interface PMPI, é possível verificar de forma transparente ao usuário o retorno de cada primitiva e, a partir de então, iniciar a recuperação do comunicador e o lançar novos processos. Uma das dificuldades nesse caso é em relação ao momento em que a falha é detectada. Uma primitiva `MPI_Send`, por exemplo, continua o seu processamento assim que a mensagem a ser enviada foi copiada para um *buffer* de saída. Nesse caso, o processo receptor dessa mensagem pode estar falho, porém a detecção se dará em outro momento, quando outra comunicação com o processo falho for realizada. Nesse caso, a falha de um processo pode ser detectada bem posteriormente.

Um interceptador de eventos, implementado através de uma *thread*, é responsável por submeter e recuperar os determinantes do *event logger*. Ao iniciar, cada processo MPI instancia um interceptador que se conecta ao *event logger*. O interceptador pode ser configurado para

[1]http://attractivechaos.awardspace.com/khash.h.html

Tabela 5.2: Comportamento das primitivas MPI durante a recuperação.

Primitiva MPI	Descrição	Comportamento durante a recuperação	Tipo	Formato do determinante
MPI_Recv	Aguarda o recebimento de uma mensagem tanto de um emissor conhecido quanto de um emissor qualquer (MPPI_ANY_SOURCE). Este último caso gera um determinante.	Se o processo aguarda uma mensagem de um de um emissor qualquer, então é necessário extrair do determinante o emissor da mensagem	Comunicação ponto a ponto, bloqueante	(ANY_SOURCE, origem, repetições)
MPI_Irecv	Semelhante à primitiva MPI_Recv, porém não é bloqueante. Isso significa que só há a garantida de que a mensagem foi recebida após a verificação por uma das primitivas de conclusão, por exemplo: MPI_Test ou MPI_Wait.	Mesmo comportamento da primitiva MPI_Recv, com a diferença de que há um objeto MPI_Request associado à primitiva e que será verificado posteriormente em uma primitiva de conclusão.	Comunicação ponto a ponto, não bloqueante	(ANY_SOURCE, origem, repetições)
MPI_Probe	Aguarda até que uma mensagem de um emissor conhecido ou desconhecido (MPI_ANY_SOURCE) esteja pronta para ser entregue.	Se o processo aguarda a recepção de uma mensagem de uma fonte qualquer, então o emissor da mensagem é extraído do determinante.	Verificação de recebimento, bloqueante	(MPI_PROBE, origem, repetições)
MPI_Iprobe	Semelhante à primitiva MPI_Probe, porém verifica continuamente se há uma mensagem a ser entregue. Geralmente, essa primitiva é encontrada em um laço de repetição.	Se o emissor é conhecido então o o número de vezes em que a primitiva foi invocada até receber a mensagem é inserido no determinante. Se o emissor é desconhecido, então a identificação do emissor também é incluída no determinante. Durante a recuperação, o emissor e o número de vezes em que a primitiva foram invocada são extraídos do determinante. O emissor desconhecido é substituído pela identificação do emissor presente no determinante e o número de vezes em que a primitiva foi invocada será repetido. Ao atingir o mesmo número de repetições a primitiva bloqueia até que uma mensagem seja recebida.	Verificação de recebimento, não bloqueante	(MPI_IPROBE_ ANY_SOURCE, origem, repetições) ou (MPI_IPROBE, origem, repetições)
MPI_Test	Verifica a conclusão de um recebimento ou de um envio não bloqueante (MPI_Irecv() ouMPI_Isend()). Sempre associado a um objeto MPI_Request	Durante a recuperação, o número de vezes em que a primitiva foi invocada é extraído do determinante e a primitiva repete o mesmo número de invocações. Ao alcançar o número exato de invocações, há o bloqueio da primitiva até que haja a conclusão do recebimento ou da emissão.	Verificação de conclusão, não bloqueante	(MPI_TEST, origem, repetições)
MPI_Testall	Verifica a conclusão de recebimento ou de envio de um conjunto de mensagens. Associado a mais de um objeto MPI_Request	Mesmo do MPI_Test	Verificação de conclusão, não bloqueante	(MPI_TESTALL, origem, repetições)
MPI_Send	Envia uma mensagem a um destinatário específico. Caso o recebimento de alguma outra mensagem esteja pendente, então um determinante deve ser gerado.	Durante a recuperação, as mensagens não são enviadas. No caso de um envio antes da conclusão de um recebimento ou envio em andamento, extrai-se do determinante o momento do envio.	Comunicação ponto a ponto, bloqueante	(MPI_TEST_SEND, origem, repetições) ou (MPI_TESTALL_ SEND, origem, repetições)
MPI_Isend	Semelhante à primitiva MPI_Send, porém é não bloqueante.	Mesmo do MPI_Send	Comunicação ponto a ponto não bloqueante	(MPI_TEST_ISEND, origem, repetições) ou (MPI_TESTALL_ ISEND, origem, repetições)
MPI_Bcast	Um processo, chamado de raiz (root), envia uma mensagem a todos os processos do grupo.	Se o processo em recuperação é o processo raiz, então a mensagem não é enviada. Somente o processo raiz guarda a mensagem enviada em sua memória. Caso o processo seja somente um receptor da difusão, então ele solicita a mensagem diretamente do processo raiz através de uma primitiva MPI_Recv.	Comunicação coletiva, bloqueante	-
MPI_Allgather	Reúne elementos de cada processo e os entrega a todos os processos	Na recuperação, a primitiva MPI_Allgather será substituída pela primitiva MPI_Gather. O processo em recuperação assume a função de processo raiz da função MPI_Gather, ou seja, recebe os elementos presentes nos vários processos.	Comunicação coletiva, bloqueante	-
MPI_Allgatherv	Reúne uma quantidade definida de elementos em cada processo os entrega a todos os processos	Na recuperação, a primitiva MPI_Allgatherv será substituída pela primitiva MPI_Gatherv. O processo em recuperação assume como processo raiz da função MPI_Gatherv reunindo em si os elementos.	Comunicação coletiva, bloqueante	-
MPI_Allreduce	Reúne um vetor de números em cada processo de acordo com uma operação (some, mínimo, por exemplo) e distribui a todos os processo.	Na recuperação, essa primitiva é substituída pela MPI_Reduce. Apenas o processo em recuperação recebe os elementos.	Comunicação coletiva, bloqueante	-

submeter determinantes de forma síncrona ou assíncrona. No modo síncrono, após submeter um determinante, o processo da aplicação aguarda a confirmação do *event logger* antes de prosseguir. No modo assíncrono o processo da aplicação pode prosseguir antes de receber a confirmação. Por padrão, todos os resultados foram obtidos usando o modo síncrono. O protocolo pessimista permite atrasar o recebimento das confirmações até o envio de uma mensagem a outro processo. Essa otimização foi aplicada na implementação do protocolo proposto. Foram implementadas três versões de *event loggers*: centralizado, baseado no Paxos Clássico e baseado no Paxos

Paralelo. O interceptador e os *event loggers* foram implementados em C usando a biblioteca *libevent* versão 2.022[2]. A biblioteca *libpaxos*[3] versão 3 foi utilizada para desenvolver os *event loggers* baseado em Paxos. A seguir são apresentados os resultados experimentais.

5.3 Resultados Experimentais

Esta seção apresenta os resultados obtidos da execução dos *event loggers* propostos, incluindo a recuperação de aplicações MPI. Os *event loggers* baseados em consenso são comparados à implementação do *event logger* centralizado. A latência e a vazão, em termos do número de determinantes armazenados por segundo, para cada um dos *event loggers* são apresentados em quatro execuções MPI: a aplicação AMG (Algebraic MultiGrid solver) [110], os *kernels* LU (Lower-Upper Gauss-Seidel solver) e MG (Multi-Grid solver) do NAS *Parallel Benchmark* versão 3.2 [52], e o algoritmo de árvores de cortes de Gusfield [135]. O algoritmo de Gusfield também foi empregado para avaliar a recuperação. As aplicações foram executadas usando a implementação OpenMPI/ULFM 1.1 [128]. Os experimentos foram conduzidos em um *cluster* dedicado que consiste de 40 nodos, cada um com dois processadores Intel(R) Quad-Core Xeon L5420 2.5 GHz e 8 Gbytes de RAM. Os nodos estão conectados através de uma rede *Gigabit Ethernet*.

5.3.1 Os *Event Loggers*

Primeiramente, o desempenho dos *event loggers* foram avaliados através de uma simples aplicação MPI onde cada processo submete um novo determinante ao *event logger* ao receber a confirmação de que o determinante submetido anteriormente está armazenado. Nessa aplicação, os processos MPI não trocam mensagens entre si e os determinantes têm o tamanho de 50 bytes. A Figura 5.4 apresenta a vazão e a latência (ambas em milissegundos) conforme o número de processos aumenta até 128. Nesse experimento há um nodo dedicado que hospeda o *event logger* centralizado. O *event logger* baseado em Paxos Clássico conta um *proposer* em um nodo dedicado e três *acceptors* (isto é, $f = 1$) e três *learners*; cada *acceptor* e *learner* executa em 1 nodo dedicado. A configuração do Paxos Paralelo é a seguinte: cada sequência de Paxos usa um *proposer*, que também é *learner*, e três *acceptors* (isto é, $f = 1$). O *proposer/learner* está hospedado junto ao processo MPI. As sequências de *acceptors* estão em três nodos dedicados, com cada *acceptor* da sequência em um nodo. Os processos MPI estão executando nos 40 nodos do *cluster*.

A Figura 5.4 apresenta a vazão e latência para os três *event loggers* implementados conforme o número de processos MPI aumenta até 128. O pequeno gráfico interno é um zoom dos 20 primeiros processos. O *event logger* centralizado atinge a vazão máxima de 83 determinantes por milissegundo (det/ms) com 16 processos. O *event logger* baseado em Paxos Clássico alcança a vazão máxima de 28 det/ms com uma latência de 4,6 ms com 128 processos MPI. Já o *event logger* baseado em Paxos Paralelo nunca chega a saturar nesse experimento: a vazão aumenta proporcionalmente ao número de processos e a latência permanece aproximadamente constante, abaixo de 4 ms. Com 128 processos, a abordagem baseada em Paxos Paralelo tem 5 vezes a vazão da abordagem centralizada e uma latência muito menor.

Apresenta-se a seguir resultados de tempo de execução e vazão das aplicações AMG, LU classes C e D e MG classes C e D (Figuras 5.5 a 5.9) usando os *event loggers* implementados.

[2]http://libevent.org/
[3]https://bitbucket.org/sciascid/libpaxos

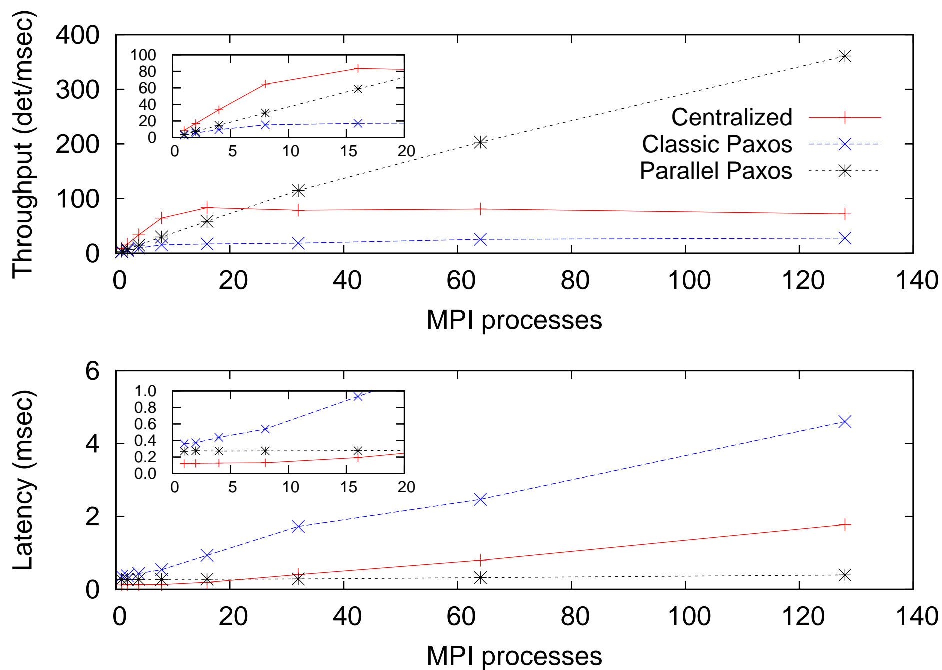


Figura 5.4: Vazão e latência para três as abordagens de *event loggers*.

Benchmark	Número de processos da aplicação			
	16	**32**	**64**	**128**
AMG	0.05%	0.02%	0.05%	0.04%
LU class C	0.63%	0,63%	0.63%	0.63%
LU class D	0.25%	0.25%	0.25%	0.25%
MG class C	99.81%	99.81%	99.81%	99.80%
MG class D	99.93%	99.93%	99.93%	99.93%

Tabela 5.3: Porcentagem de eventos não-determinísticos nos *benchmarks* empregados.

Nas Figuras onde se apresenta o tempo de execução (esquerda), a primeira barra corresponde ao desempenho da aplicação sem o uso de qualquer *event logger* e sem registrar qualquer tipo de evento. A segunda barra indica os resultados para a abordagem centralizada e as duas últimas para os *event loggers* baseados em Paxos Clássico e Paxos Paralelo, respectivamente. Nas Figuras onde se apresenta a vazão, o Paxos Paralelo, o que apresentou a maior vazão segundo a Figura 5.4, é executado com as configurações síncrona e assíncrona. A configuração assíncrona não aguarda a confirmação do registro de um determinante e, dessa forma, apresenta a vazão desejada para obter o menor tempo de execução. Para os resultados da aplicações AMG, LU e MG os *event-loggers* são configurados conforme descrito no início desta seção. Foram usados 16, 32, 64 e 128 processos MPI.

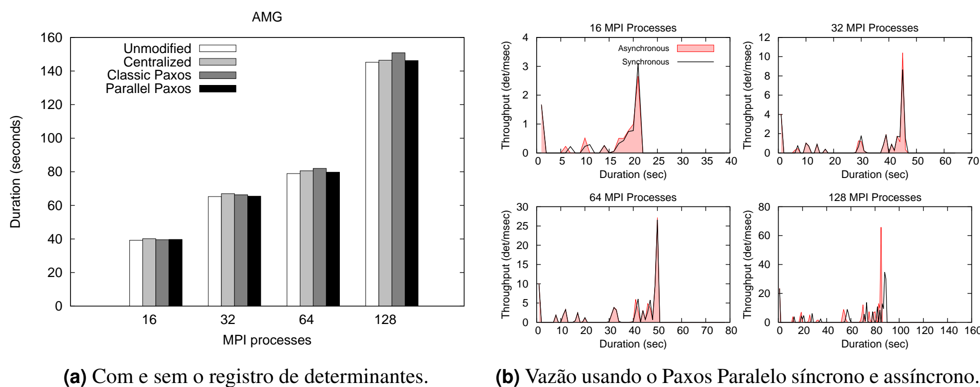
5.3.2 AMG

A implementação MPI do método *multigrid* algébrico (AMG) para resolução de equações algébricas lineares é uma aplicação classificada como não-determinística devido à presença de recepções do tipo `any_source` e entregas não-determinísticas. Todas as chamadas `MPI_Iprobe` usam a marcação `MPI_ANY_SOURCE` e somente uma chamada `MPI_Recv`, entre muitas, usa tal marcação. A aplicação também possui primitivas de verificação de re-

cebimento `MPI_Test` e `MPI_Testall`. A Tabela 5.3 apresenta o percentual de eventos não-determinísticos nas execuções da aplicação AMG para 16, 32, 64 e 128 processos MPI. Considerando todos os eventos gerados, os eventos não-determinísticos da aplicação AMG correspondem a menos de 0,1% do total. Nos experimentos realizados, os parâmetros *rx, ry* e *rz* foram definidos em 40. Embora as primitivas de verificação de recebimento gerem muitos eventos não-determinísticos, um determinante somente é gerado quando a mensagem está pronta para ser entregue. Isso acontece com as primitivas `MPI_Iprobe`, `MPI_Test` e `MPI_Testall`. O número de vezes em que a mensagem não estava pronta para ser recebida é incluído no determinante.

A Figura 5.5(a) apresenta os resultados da execução da aplicação AMG incluindo os *event loggers* distribuídos e replicados. Apesar de todas as estratégias introduzirem uma sobrecarga, a maior sobrecarga foi observada para Paxos Clássico com 64 e 128 processos: aproximadamente 3,8%. A estratégia centralizada apresenta uma sobrecarga de aproximadamente 2% para 16, 32 e 64 processos. O Paxos Paralelo apresenta a menor sobrecarga, abaixo de 1,3% para todas as execuções.

A Figura 5.5(b) apresenta o número de determinantes registrados por milissegundos (det/ms) considerando tanto para as configurações síncronas e assíncronas do Paxos Paralelo para 16, 32, 64 e 128 processos MPI. No modo assíncrono, os processos da aplicação nunca aguardam pelo registro do determinante; assim, esse modo fornece um limite superior para a vazão do *event logger*. Como pode ser visto, as solicitações de registro de determinantes não são distribuídas uniformemente durante a execução. Para o caso com 128 processos, entre os instantes de tempo 80 e 90 segundos, um pico que atinge aproximadamente 34 det/ms é observado no modo síncrono e 66 det/ms é observado no modo assíncrono. Esses resultados ajudam a compreender como a sobrecarga do Paxos Paralelo na abordagem síncrona é distribuída ao longo da execução.



(a) Com e sem o registro de determinantes. **(b)** Vazão usando o Paxos Paralelo síncrono e assíncrono.

Figura 5.5: Tempo de execução e vazão da aplicação AMG.

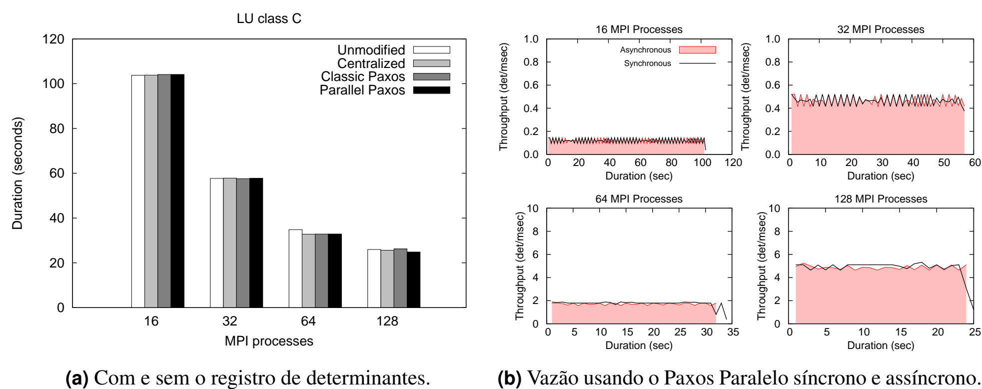
5.3.3 LU e MG

Os *kernels* LU e MG são *kernels* do NAS *Parallel Benchmark* versão 3.2. Ambos são os únicos dessa versão do NAS a apresentam eventos não-determinísticos. Em ambos, os únicos eventos não-determinísticos são recepções com a marcação `MPI_ANY_SOURCE`. Resultados das classes C e D são apresentados em execuções com 16, 32, 64 e 128 processos MPI. No NAS, o tamanho das entradas para os experimentos é categorizado em classes. A diferença entre as classes C e D está no tamanho do problema a ser resolvido. O problema da classe D é maior que o da classe C. O *kernel* LU contém recepções através das primitivas `MPI_Recv` e `MPI_Irecv`.

Essa última com a marcação MPI_ANY_SOURCE. O *kernel* MG recebe todas as suas mensagens através da primitiva MPI_Irecv com a marcação MPI_ANY_SOURCE.

De acordo com a Tabela 5.3, o número total de eventos não-determinísticos registrado no LU é menor do que 1% do total de todas as recepções tanto na classe C quanto D. O *kernel* MG, por sua vez, contém quase 100% de eventos não-determinísticos entre todas as suas recepções. Apesar de a aplicação AMG e o *kernel* MG resolverem problemas similares, o MG possui muito mais eventos não-determinísticos devido à sua implementação. Diferentemente do MG, a aplicação AMG não recebe todas as suas mensagens através da primitiva MPI_Irecv com a marcação MPI_ANY_SOURCE. Isso ilustra o fato de o não-determinismo no código da aplicação ser frequentemente uma escolha do programador (por exemplo, para aumentar o desempenho), ao invés de um requisito proveniente do problema a ser resolvido.

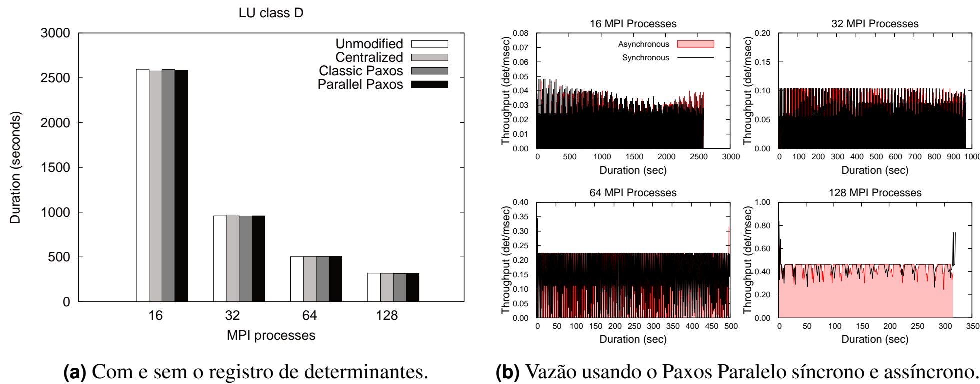
A partir dos experimentos realizados para o LU classe C e D, conclui-se que o registro de determinantes usando qualquer um dos 3 *event loggers* implementados virtualmente não impactam no desempenho do LU. De certa forma, isso é surpreendente uma vez que comparado com a aplicação AMG, tanto a classe C quanto a classe D do LU contém uma porcentagem maior de eventos não-determinísticos (ver Tabela 5.3). As Figuras 5.6(a) e 5.7(a) apresentam o tempo de execução das classes C e D, respectivamente. Ao inspecionar a quantidade de determinantes armazenados durante a execução nas Figuras 5.6(b) e 5.7(b), percebe-se que o registro de determinantes é uniformemente distribuído no *kernel* LU, o que não é verdade para a aplicação AMG. Além disso, o registro de determinantes no LU ocorre em uma taxa abaixo do limite suportado pelos *event loggers* (ver Seção 5.3.2). Consequentemente, o *event logger* nunca se torna um gargalo na execução do LU. A diferença entre as classes C e D é que a última tem maior duração e menor vazão.



(a) Com e sem o registro de determinantes. **(b)** Vazão usando o Paxos Paralelo síncrono e assíncrono.

Figura 5.6: Tempo de execução e vazão da aplicação LU classe C.

Ao contrário do *kernel* LU, o *kernel* MG classes C e D introduz uma sobrecarga considerável ao registrar os seus eventos não-determinísticos (Figuras 5.8 e 5.9, respectivamente). Na classe C, enquanto o Paxos Clássico apresenta uma sobrecarga maior do que 125% e 200% para 64 e 128 processos, a abordagem centralizada apresenta sobrecarga abaixo de 31% e 55% para 64 e 128 processos MPI, respectivamente. O Paxos Paralelo apresenta sobrecarga ainda inferior: 17,71% e 24,26% para 64 e 128 processos, respectivamente. Os resultados para a classe D apresentam uma tendência similar, a abordagem Paxos Paralelo apresenta desempenho superior comparado as outras duas abordagens. A Tabela 5.4 apresenta as sobrecargas para todas as abordagens dos *event loggers* e configurações do MG classes C e D. O MG é altamente limitado pela comunicação e como todas as suas recepções usam a marcação MPI_ANY_SOURCE. Conforme aumenta o número de processos, os *event loggers* nas abordagens centralizado e



(a) Com e sem o registro de determinantes. **(b)** Vazão usando o Paxos Paralelo síncrono e assíncrono.

Figura 5.7: Tempo de execução e vazão da aplicação LU classe D.

Paxos Clássico atingem o seu limite. A abordagem Paxos Paralelo é capaz de dimensionar o seu desempenho distribuindo a carga entre as várias sequências de Paxos.

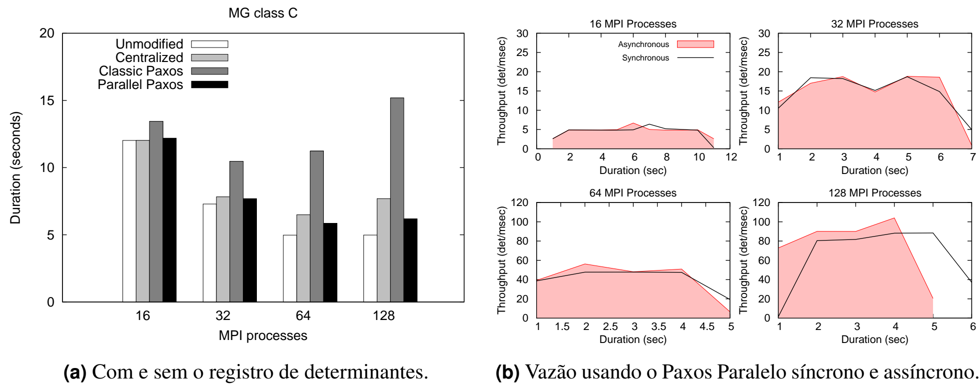
	Proc	Centralizado	Paxos Clássico	Paxos Paralelo
MG classe C	16	0.03%	11.81%	1.43%
	32	7.39%	43.61%	5.53%
	64	30.52%	125.78%	17.71%
	128	54.38%	204.86%	24.26%
MG classe D	16	0.60%	0.29%	0.42%
	32	0.69%	0.45%	0.72%
	64	4.60%	25.43%	2.24%
	128	9.48%	41.91%	2.93%

Tabela 5.4: Sobrecarga das implementações de *event loggers* Centralizado, Paxos Clássico e Paxos Paralelo no *kernel* MG classes C e D.

As Figuras 5.8(b) e 5.9(b) apresentam a taxa de determinantes registrados por milissegundos (det/ms) para o *event logger* baseado em Paxos Paralelo síncrono e assíncrono. A vazão do modo síncrono é próxima do modo assíncrono para MG classe C com 16, 32 e 64 processos. Para 128 processos, o modo assíncrono apresenta uma vazão maior que o modo síncrono e termina a sua execução 1 segundo mais cedo. O MG classe D tem menor vazão que a classe C. A vazão para os modos síncrono e assíncrono do Paxos Paralelo é muito similar. Em todas as configurações, tanto o modo síncrono quanto o assíncrono apresentam uma taxa uniforme ao longo da execução.

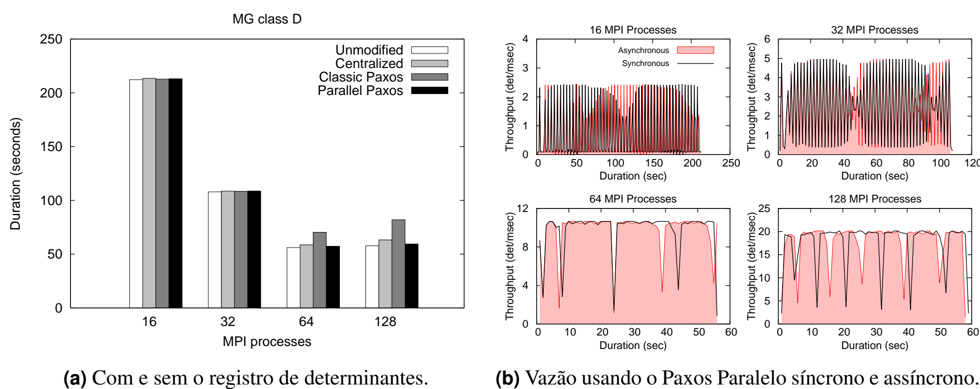
5.3.4 Gusfield

O algoritmo de Gusfield, Figura 5.10, foi executado com 4, 8, 12 e 32 processos MPI para uma entrada de 2000 vértices e 21.990 arestas. Esta é uma aplicação do tipo mestre-escravo. Somente o processo mestre gera eventos não-determinísticos uma vez que todas as suas recepções usam a marcação `MPI_ANY_SOURCE`. Nesse experimento foram avaliados a aplicação sem modificações (barra clara) e com a abordagem Paxos Paralelo. Porém, há duas configurações distintas do Paxos Paralelo. Na primeira, cada sequência de *acceptor* usa 3 nodos dedicados, como nos experimentos anteriores (barra cinza). Na segunda configuração (barra preta), os *acceptors* são co-alocados com os processos da aplicação e, assim, não há nodo extra



(a) Com e sem o registro de determinantes. **(b)** Vazão usando o Paxos Paralelo síncrono e assíncrono.

Figura 5.8: Tempo de execução e vazão da aplicação MG classe C.



(a) Com e sem o registro de determinantes. **(b)** Vazão usando o Paxos Paralelo síncrono e assíncrono.

Figura 5.9: Tempo de execução e vazão da aplicação MG classe D.

usado. Como é possível perceber, não houve diferença significativa de sobrecarga entre as duas configurações de Paxos Paralelo nesse experimento.

5.3.5 Recuperação do Algoritmo de Gusfield e da Aplicação AMG

O protocolo proposto, incluindo a recuperação, foi avaliado através do algoritmo de Gusfield e da aplicação AMG. Todas as primitivas do protocolo proposto descritas na seção 5.2 foram utilizadas. Ambas as aplicações foram modificadas para incluir cenários de falha e recuperação. Basicamente, o algoritmo de Gusfield executa cálculos de fluxos máximos em um laço de repetição. O mestre solicita que seus escravos calculem uma parte do problema e ao receber o resultado de um escravo envia uma nova parte do problema a esse escravo. A primitiva `chkp()`, por exemplo, foi inserida no fim do laço de repetição. Já a primitiva `recovery()` é inserida antes do laço. Foram incluídos no *checkpointing*, entre outros, os vetores que guardam os valores calculados a cada iteração. Por sua vez, a aplicação AMG é do tipo sequencial. A função principal (*main()*) possui mais de mil linhas de código e não conta com laços de repetição. No entanto, há a invocação de dezenas de funções, cada uma com suas centenas de linhas de código fonte. Assim como no algoritmo de Gusfield, na aplicação AMG a primitiva `framework_init()` foi iniciada logo após a primitiva `MPI_Init()`. A primitiva `recovery()` também foi posicionada logo nas linhas seguintes. Ambas as aplicações fazem uso do comunicador `MPI_COMM_WORLD` explicitamente declarado no código. O mesmo foi trocado em todas as linhas de código para o comunicador utilizado no `framework_init`, que é basicamente uma cópia do comunicador `MPI_COMM_WORLD` a fim de permitir a recuperação.

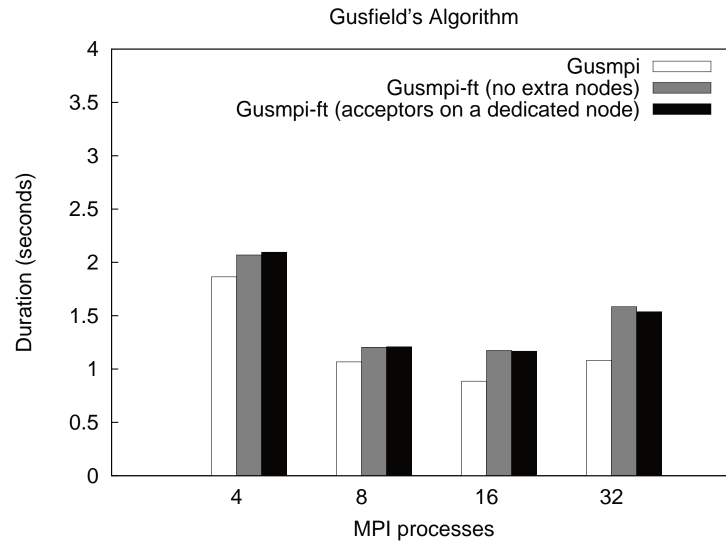


Figura 5.10: Tempo de execução do algoritmo de Gusfield usando o Paxos Paralelo com diferentes configurações.

A Figura 5.11 apresenta a execução do algoritmo para 4, 8, 16, 32 e 64 processos em uma entrada com 1000 vértices e 99.900 arestas. O *event logger* baseado na abordagem Paxos Paralelo foi empregado na execução. A primeira barra corresponde à execução do algoritmo original, isto é, sem modificações. A segunda barra apresenta a sobrecarga causada pelo registro dos determinantes no *event logger*. A terceira barra diz respeito a um cenário de falha e recuperação do mestre. Nessa execução houve a inserção de falhas aleatórias durante a execução. A quarta barra se refere à falha e recuperação dos escravos. Durante a execução um dos escravos falhava aleatoriamente. A quinta barra representa a adição ao cenário de falha do mestre de um *checkpointing* durante a metade da execução. A sexta barra adiciona a esse último cenário a coleta de mensagens antigas, configurado para executar pelo menos uma vez durante a execução.

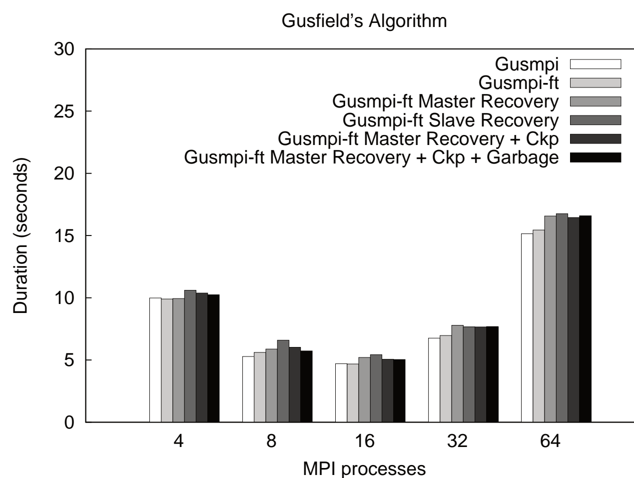


Figura 5.11: Recuperação do algoritmo Gusfield.

De acordo com o gráfico da Figura 5.11 é possível perceber que o tempo que a aplicação leva para se recuperar de forma consistente usando o protocolo proposto não é significante. Lembrando que o processo de recuperação inclui lançar um novo processo, recuperar os seus determinantes do *event logger*, ler o seu *checkpoint* e refazer a computação perdida. Tendo como base a aplicação sem qualquer modificação (primeira barra da Figura 5.11), quando o mestre

se recupera de uma falha (terceira barra), a maior sobrecarga observada foi de 15,16% para 32 processos. Porém, quando o mestre realiza um *checkpointing* (quinta barra), a sobrecarga para esse mesmo caso cai para 13,2%. O mesmo ocorre no cenário de 64 processos, a sobrecarga com o mestre se recuperando cai de 9,42% para 8,62%. Ou seja, o tempo que o mestre leva realizar um *checkpoint* é compensado no processo de recuperação. O tempo de recuperação dos escravos (quarta barra) foi superior ao tempo de recuperação do mestre nesse experimento porque os escravos repetem a sua computação desde a última falha. Além disso, o mestre precisa se certificar que não deixou de receber mensagens enviadas por algum outro escravo devido à detecção da falha. É possível otimizar a implementação nesse cenário porque os escravos não precisam guardar um estado prévio. Nesse caso, ao detectar a falha de um escravo o mestre faria um *checkpoint* do seu estado e simplesmente lançaria um novo escravo que não precisaria refazer a sua computação. É possível observar também que ao adicionar a limpeza de mensagens antigas (última barra) não houve impacto significativo no tempo de execução (se comparado a penúltima barra).

A Figura 5.12 apresenta a recuperação da aplicação AMG para 8, 16 e 32 processos. O Paxos Paralelo foi aplicado como *event logger*. Nas execuções a seguir, cada processo executa em uma máquina. Além disso, a AMG foi configurada para realizar o processamento sem utilizar as diretivas do OpenMP (*Open Multi-Processing*). O OpenMP é uma biblioteca que suporta o processamento através de memória compartilhada. Dessa forma, a AMG não conta com os núcleos de um processo para auxiliar na execução. Importante notar também que conforme aumenta o número de processos envolvidos aumenta o tamanho do problema a ser computado. A recuperação da aplicação AMG envolveu todas as primitivas MPI descritas na Tabela 5.2, exceto a `MPI_Probe`. Na Figura 5.12, a primeira barra representa execuções da aplicação original, sem o registro de determinantes ou capacidade de recuperação. A segunda barra corresponde a um cenário de falha e recuperação da aplicação AMG. Foram inseridas falhas em 6 diferentes pontos no código da aplicação. O resultado apresentado corresponde a uma média do primeiro ao terceiro ponto de falha onde o processo falho enviou uma média de 35 mensagens a cada processo e recebeu uma média de 35 mensagens por processo.

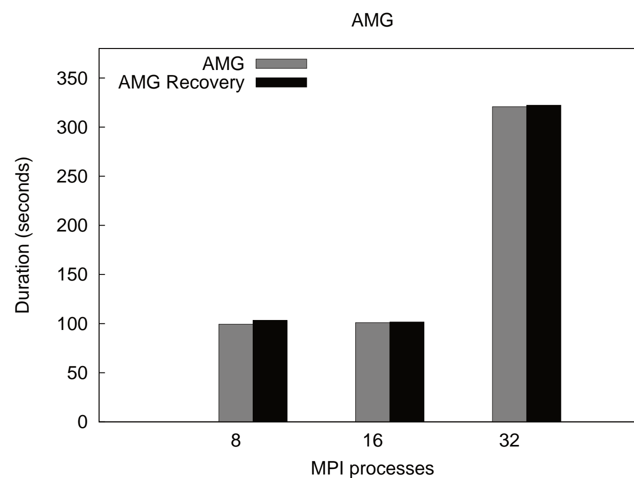


Figura 5.12: Recuperação da aplicação AMG.

5.4 Conclusão

Neste trabalho apresentamos um protocolo pessimista para registro de mensagens construído com um *event logger* distribuído e tolerante a falhas baseado em consenso. O *event logger* é o componente responsável por armazenar de forma confiável um determinante e o seu desempenho representa um impacto significante na eficiência dos protocolos de registro de mensagens. Dois *event loggers* baseados no algoritmo Paxos foram implementados e avaliados. Ao empregar o algoritmo de consenso Paxos, os *event logger* propostos garantem a propriedade de segurança mesmo se o sistema for assíncrono e o progresso apesar de falhas de processos. O primeiro *event logger* é baseado na replicação máquina de estado clássica. O segundo *event logger*, chamado de Paxos Paralelo, atribui uma sequência separada de execuções do Paxos a cada processo da aplicação. Os *event loggers* baseados em consenso e um *event logger* centralizado foram avaliados experimentalmente. Nos experimentos realizados, usando as aplicações AMG, LU e MG, o *event logger* baseado em Paxos Paralelo apresentou desempenho superior ao da abordagem centralizada. O algoritmo de Gusfield foi também avaliado com diferentes configurações dos *acceptors*.

A implementação da recuperação também foi realizada usando a especificação ULFM. Para auxiliar na implementação da recuperação, a mesma foi encapsulada em uma biblioteca para facilitar o seu uso por uma aplicação MPI. Entre as funcionalidades presentes nas primitivas de recuperação estão o *checkpointing* não-coordenado, onde o desenvolvedor define quais variáveis da aplicação serão armazenadas no *checkpoint*, e a coleta de lixo responsável por realizar a remoção de mensagens antigas mantidas na memória do emissor. O algoritmo de Gusfield e aplicação AMG foram modificados para incluir as primitivas implementadas. Os determinantes são adequadamente recuperados do *event logger* e reaplicados na recuperação. As primitivas MPI foram adaptadas para suportar a recuperação da aplicação. O algoritmo de Gusfield oferece um modelo de execução do tipo mestre-escravo onde os processos basicamente recebem e enviam mensagem ao mestre. Por outro lado, a aplicação AMG emprega diversas primitivas MPI e foi originalmente projetada para executar em sistemas HPC com centenas de processadores. Resultados demonstram que, perante falhas, os processos da aplicação se recuperam eficientemente. Trabalhos futuros incluem avaliar a recuperação da aplicação AMG executando sobre centenas de processadores.

Capítulo 6

Conclusão

Projetar mecanismos de tolerância a falhas efetivos e permitir que as aplicações HPC completem adequadamente as suas execuções apesar de falhas é uma tarefa árdua. No contexto dos sistemas HPC, a diminuição de tempo médio entre as falhas (MTBF) exige o constante aprimoramento das técnicas e das estratégias de tolerância a falhas. Esta tese de doutorado propôs duas estratégias de tolerante a falhas para sistemas HPC baseados em MPI. A primeira contribuição é uma estratégia para identificar e manter um grupo dinâmico de processos recomendados, também chamado de DGRP (*Dynamic Group of Recommended Processes*). O DGRP é baseado em um novo modelo de diagnóstico em nível de sistema. A segunda estratégia é um protocolo de *rollback-recovery* baseado em registro de mensagens pessimista que conta com um *event logger* distribuído e tolerante a falhas.

Um DGRP é definido como um grupo dinâmico autogerenciado composto por processos recomendados que permite às aplicações continuar a sua execução mesmo enquanto processos se tornam não-recomendados e são removidos do DGRP. No modelo de diagnóstico que serve como base para o DGRP, dois testadores podem obter resultados diferentes ao executar um teste em um terceiro processo. Os processos executam testes entre si a fim de determinar se estão recomendados ou não-recomendados. Um processo não-recomendado é removido do DGRP. Porém, esse mesmo processo é reintegrado ao DGRP se posteriormente passa em um sequência de teste. A reintegração é realizada pelos processos do DGRP através da execução do consenso. A execução da aplicação é organizada em rodadas de computação. De acordo com a composição do DGRP as tarefas da aplicação são adequadamente atribuídas aos processos recomendados. A abordagem do DGRP pode ser empregada em diversos tipos de aplicações. O modelo foi implementado em MPI e foi empregado para classificar os processos de um *cluster* compartilhado multiusuário. Como estudo de caso foi implementado o algoritmo *Hyperquicksort*. Uma versão tolerante a falhas do algoritmo *Hyperquicksort* também foi desenvolvida para executar sobre o DGRP. Nessa versão, os processos do *Hyperquicksort* se adaptam em tempo de execução conforme a formação vigente do DGRP. Resultados experimentais apresentam a execução do *Hyperquicksort* em um *cluster* compartilhado executando somente através da ULFM, a mais recente proposta de implementação de tolerância a falhas em MPI, e executando sobre o DGRP. Para suportar a falha de processos, o DGRP foi implementado em cima da especificação ULFM. Os resultados demonstram que o modelo proposto é eficiente e efetivo. Ao ser usado para monitorar os processadores de *cluster* compartilhado, o DGRP foi capaz de identificar os nodos mais ocupados e identificar a variabilidade de desempenho entre os processadores do *cluster*.

Na segunda contribuição desta tese, um protocolo de registro de mensagens pessimista baseado em um *event logger* distribuído e tolerante a falhas foi proposto. O *event logger* se baseia no algoritmo de consenso Paxos e, consequentemente, herda as propriedades do mesmo:

garante a propriedade de segurança mesmo se o sistema for assíncrono e o progresso apesar de falhas de processos. Além disso, os *event loggers* replicados podem ser posicionados em conjunto com os processos da aplicação. Nesse caso, evita-se o uso de servidores dedicados para armazenar as réplicas. Duas implementações do *event logger* foram realizadas e comparadas com um *event logger* centralizado. A primeira implementação usa uma configuração tradicional do Paxos, chamada de Paxos Clássico. A segunda implementação emprega sequências de consenso para os processos da aplicação e é chamada de Paxos Paralelo. No Paxos Paralelo, cada processo MPI tem acesso a uma sequência particular de consenso. Os resultados obtidos demonstram que o *event logger* baseado em Paxos Paralelo possui desempenho comparável ou superior ao *event logger* centralizado. No entanto, outro importante diferencial do Paxos Paralelo é tolerar um número configurável de falhas. Os *event loggers* propostos foram avaliados usando as aplicações AMG, MG, LU e o algoritmo de Gusfield. O protocolo pessimista desenvolvido se baseia no *event logger* proposto para realizar a recuperação da aplicação. O protocolo proposto faz a distinção dos eventos determinísticos e não-determinísticos em aplicações MPI. Uma cópia de cada mensagem enviada é mantida pelo emissor na sua memória volátil. Periodicamente os processos realizam um *checkpointing* não-coordenado e executam a limpeza de mensagens antigas. As primitivas MPI formam reprogramadas para lidar com a recuperação da aplicação. Nesse caso, os determinantes são recuperados do *event logger* e a computação é refeita a partir do último *checkpoint*. O protocolo foi desenvolvido em um conjunto de primitivas que são inseridas na aplicação MPI. A recuperação da aplicação foi avaliada através do algoritmo de Gusfield e da aplicação AMG. Os resultados demonstram que as aplicações MPI são corretamente recuperadas apesar da falha de processos.

Trabalhos futuros no âmbito desta tese incluem definir primitivas que permitam a aplicação se comunicar com o DGRP; aplicar o novo modelo de diagnóstico para monitorar diversos tipo de redes e sistemas e definir outras estratégias para manutenção do DGRP. Por exemplo, processos instáveis, ou seja, que constantemente se tornam não-recomendados podem vir a ser excluídos permanentemente ou por períodos maiores de tempo; implementar estratégias de testes baseadas no modelo de diagnóstico hierárquico para diminuir número de mensagens trocadas na rede. Destaca-se ainda a possibilidade de aplicar outros modelos de diagnóstico em sistemas HPC de larga escala como, por exemplo, o diagnóstico baseado em comparações para lidar com falhas do tipo computação incorreta em aplicações HPC. Além disso, implementar versões tolerante a falhas de algoritmos paralelos, aproveitando principalmente as vantagens presentes da topologia do hipercubo.

No contexto da segunda contribuição desta tese, trabalhos futuros incluem avaliar a aplicações AMG em um cenário com centenas ou milhares de processos; projetar protocolos de registro de mensagens otimista e causal baseados no Paxos Paralelo e compará-los entre si; aprimorar as primitivas definidas para o protocolo, incluindo-as e avaliando-as em outras aplicações HPC. Destaca-se ainda estudar como o Paxos Paralelo pode mandar a ordem total apesar das diferentes sequências de consenso sem perder a alta vazão e mantendo a baixa latência demonstrada; construir outros estratégias de *rollback-recovery* baseado em modelos híbridos ou hierárquicos, ou seja, que mesclem a estratégia coordenada com o registro de mensagens; avaliar as diferentes variações do Paxos em sistemas HPC de larga escala onde centenas ou milhares de processos estão envolvidos no consenso e; estudar adaptações na topologia empregada para realizar o consenso.

Referências Bibliográficas

[1] Lecture Notes in Computer Science. 2003.

[2] Aguilera, Chen, and Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing Journal*, 13, 2000.

[3] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. *Lecture Notes in Computer Science*, 1320:126–??, 1997.

[4] Luiz Carlos Pessoa Albini, Stefano Chessa, and Piero Maestrini. Diagnosis of symmetric graphs under the BGM model. *The Computer Journal*, 47(1):85–92, 2004.

[5] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[6] F. Barsi, F. Grandoni, and P. Maestrini. A theory of diagnosability of digital systems. *IEEE Transactions on Computers*, C-25(6):585–593, 1976.

[7] Rajanikanth Batchu, Yoginder S. Dandass, Anthony Skjellum, and Murali Beddhu. MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 7(4):303–315, 2004.

[8] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *SC*, 2011.

[9] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead, 2008.

[10] Bianchini and Buskens. Implementation of on-line distributed system-level diagnosis theory. *IEEETC: IEEE Transactions on Computers*, 41, 1992.

[11] Jr. Bianchini, R. and R. Buskens. An adaptive distributed system-level diagnosis algorithm and its implementation. In *Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium*, pages 222–229, 1991.

[12] Ken Birman. A history of the virtual synchrony replication model. In *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 91–120, 2010.

[13] Kenneth Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5:47–76, 1987.

[14] W. Bland. Enabling application resilience with and without the mpi standard. In *CCGrid*, pages 746–751, 2012.

[15] W. Bland, H. Lu, S. Seo, and P. Balaji. Lessons learned implementing user-level failure mitigation in mpich. In *CCGrid*, pages 1123–1126, 2015.

[16] Wesley Bland, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. A proposal for user-level failure mitigation in the mpi-3 standard. Technical report, Department of Electrical Engineering and Computer Science, University of Tennessee, 2012.

[17] Wesley Bland, Aurelien Bouteiller, Thomas Hérault, George Bosilca, and Jack Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of HPC Applications*, 27(3):244–254, 2013.

[18] Wesley Bland, Aurelien Bouteiller, Thomas Hérault, Joshua Hursey, George Bosilca, and Jack J. Dongarra. An evaluation of user-level failure mitigation support in MPI. In *EuroMPI*, pages 193–203, 2012.

[19] Wesley Bland, Peng Du, Aurelien Bouteiller, Thomas Hérault, George Bosilca, and Jack Dongarra. A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI. In *Euro-Par*, 2012.

[20] D. M. Blough, G. F. Sullivan, and G. M. Masson. Almost certain diagnosis for intermittently faulty systems. In *International Symposium on Fault-Tolerant Computing*, pages 260–265, 1988.

[21] Luis C. E. Bona, Keiko Veronica Ono Fonseca, Elias Procopio Duarte, Jr., and Samuel L. V. de Mello. Hyperbone: A scalable overlay network based on a virtual hypercube. In *CCGrid*, pages 58–64, 2008.

[22] George Bosilca, Remi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[23] Marin Bougeret, Henri Casanova, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Using group replication for resilience on exascale systems. *International Journal of High Performance Computing Applications*, 28(2):210–224, 2014.

[24] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier, and F. Cappello. Impact of event logger on causal message logging protocols for fault tolerant mpi. In *IPDPS*, pages 97–97, 2005.

[25] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, 2010.

[26] Aurelien Bouteiller, Franck Cappello, Thomas Herault, Geraud Krawezik, Pierre Lemarinier, and Frederic Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *SC*, 2003.

[27] Aurelien Bouteiller, Thomas Hérault, George Bosilca, and Jack J. Dongarra. Correlated set coordination in fault tolerant message logging protocols for many-core clusters. *Concurrency and Computation: Practice and Experience*, 25(4):572–585, 2013.

[28] Aurelien Bouteiller, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Franck Cappello. MPICH-V project: A multiprotocol automatic fault-tolerant MPI. *The International Journal of High Performance Computing Applications*, 20(3):319–333, 2006.

[29] Aurelien Bouteiller, Thomas Ropars, George Bosilca, Christine Morin, and Jack Dongarra. Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure, recovery. In *Cluster*, 2009.

[30] Darius Buntinas. Scalable distributed consensus to support MPI fault tolerance. In *IPDPS*, pages 1240–1249, 2012.

[31] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI.

[32] E. T. Camargo and E. P. Duarte. Running fault-tolerant mpi-based applications in unstable systems. *Workshop on Exascale MPI*, 2014.

[33] E. T. Camargo and E. P. Duarte. Network monitoring with imperfect tests. In *LANCOMM*, pages 49–51, 2016.

[34] E. T. Camargo and E. P. Duarte. Running resilient mpi applications on a dynamic group of recommended processes. In *LADC*, pages 15–24, 2016.

[35] E. T. Camargo, E. P. Duarte., and W. C. Pietniczka. Diagnóstico distribuído com testes imperfeitos aplicado à detecção de estabilidade em sistemas baseados em mpi. In *WSCAD*, page 12, 2014.

[36] E. T. Camargo, E. P. Duarte, and W. C. Pietniczka. Implementação de difusão atômica baseada em diagnóstico com testes imperfeitos. In *CTIC*, pages 515–524, 2014.

[37] Franck Cappello, Al Geist, Bill Gropp, Laxmikant V. Kalé, Bill Kramer, and Marc Snir. Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[38] Franck Cappello, Amina Guermouche, and Marc Snir. On communication determinism in parallel HPC applications. In *ICCCN*, pages 1–8, 2010.

[39] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[40] M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *Transactions on Computer Systems*, 3(1):63–75, 1985.

[41] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

[42] Zizhong Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *IPDPS*, pages 10 pp.–, 2006.

[43] Zizhong Chen and Jack Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions Parallel Distributed Systems*, 19(12):1628–1641, 2008.

[44] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Building fault survivable mpi programs with ft-mpi using diskless checkpointing. In *PPoPP*, pages 213–223, 2005.

[45] Zizhong Chen and Panruo Wu. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1323–1335, 2015.

[46] Kyung-Yong Chwa and S. Louis Hakimi. Schemes for fault-tolerant computing: A comparison of modularly redundant and t-diagnosable systems. *Information and Control Journal*, 49(3):212–238, 1981.

[47] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed. edition, 2001.

[48] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *ICS*, pages 162–171, 2011.

[49] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[50] S. Di, L. Bautista-Gome, and F. Cappello. Optimization of a multilevel checkpoint model with uncertain execution scales. In *SC*, 2014.

[51] C. Di Martino, Z. Kalbarczyk, R.K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *DSN*, pages 610–621, 2014.

[52] NASA Advanced Supercomputing Division. Nas parallel benchmarks. `https://www.nas.nasa.gov/publications/npb.html`, year = 2017, note = Acessado em 01/02/2017.

[53] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *PPoPP*, pages 225–234, 2012.

[54] E. P. Duarte, A. Brawerman, and L. C. P. Albini. An algorithm for distributed hierarchical diagnosis of dynamic fault and repair events. In *ICPADS*, pages 299–306, 2000.

[55] Elias P. Duarte, Jr., Luis C. E. Bona, and Vinicius K. Ruoso. Vcube: A provably scalable distributed diagnosis algorithm. In *ScalA Workshop*, pages 17–22, 2014.

[56] Elias Procópio Duarte, Roverli Pereira Ziwich, and Luiz Carlos Pessoa Albini. A survey of comparison-based system-level diagnosis. *ACM Computing Surveys*, 43(3):22, 2011.

[57] E. P. Duarte, Jr. and T. Nanya. A hierarchical adaptive distributed system-level diagnosis algorithm. *IEEE Transactions on Computers*, 47(1):34–45, January 1998.

[58] Jason Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Technical report, Lawrence Berkeley National Laboratory, 2003.

[59] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[60] Nosayba El-Sayed and Bianca Schroeder. Reading between the lines of failure logs: Understanding how HPC systems fail. In *DSN*, pages 1–12, 2013.

[61] Elnozahy, Alvisi, Wang, and Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34, 2002.

[62] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent advances in PVM and MPI*, LNCS. Springer, 2000.

[63] Graham E. Fagg and Jack Dongarra. Building and using a fault-tolerant MPI implementation. *The International Journal of High Performance Computing Applications*, 18(3):353–361, 2004.

[64] Pascal Felber, Xavier Défago, Rachid Guerraoui, and Philipp Oser. Failure detectors as first class objects. In *DOA*, pages 132–141, 1999.

[65] Kurt B. Ferreira, Jon Stearley, James H. Laros, III, Ron Oldfield, Kevin T. Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *SC*, page 44, 2011.

[66] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC*, 2012.

[67] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[68] MPI Forum. Mpi 4.0. `http://mpi-forum.org/mpi-40/`, year = 2017, note = Acessado em 01/02/2017,.

[69] MPI Forum. User-level failure mitigation. `http://fault-tolerance.org/ulfm/ulfm-specification/`, year = 2017, note = Acessado em 01/02/2017,.

[70] MPI Forum. Mpi forum fault tolerance working group. `https://github.com/mpiwg-ft`, 2017. Acessado em 01/02/2017.

[71] MPI Forum. Mpi forum website. `http://mpi-forum.org/`, 2017. Acessado em 26/01/2017.

[72] MPI Forum. Run-through stabilization process fault tolerance proposal. `https://github.com/mpi-forum/mpi-forum-historic/issues/276`, 2017. Acessado em 01/02/2017.

[73] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[74] Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Failure prediction for HPC systems and applications: Current situation and open issues. *The International Journal of High Performance Computing Applications*, 27(3):273–282, 2013.

[75] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In *SC*, 2014.

[76] Marc Gamell, Keita Teranishi, Michael A. Heroux, Jackson Mayo, Hemanth Kolla, Jacqueline Chen, and Manish Parashar. Local recovery and failure masking for stencil-based applications at extreme scales. In *SC*, 2015.

[77] Stéphane Genaud, Emmanuel Jeannot, and Choopan Rattanapoka. Fault-management in P2P-MPI. *International Journal of Parallel Programming*, 37(5):433–461, 2009.

[78] R. Gioiosa, G. Kestor, and D.J. Kerbyson. Online monitoring system for performance fault detection. In *IPDPS Workshops*, 2014.

[79] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[80] William Gropp and Ewing L. Lusk. Fault tolerance in message passing interface programs. *The International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[81] Amina Guermouche, Thomas Ropars, Marc Snir, and Franck Cappello. HydEE: Failure containment without event logging for large scale send-deterministic MPI applicat. In *IPDPS*, 2012.

[82] Rachid Guerraoui, Christian Cachin, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[83] S. L. Hakimi and A. T. Amin. Characterization of connection assignment of diagnosable systems. *IEEE Transactions on Computers*, C-23(1):86–88, January 1974.

[84] S. L. Hakimi and K. Nakajima. On adaptive system diagnosis. *IEEE Transactions on Computers*, 33(3):234–240, 1984.

[85] Thomas Herault, Aurelien Bouteiller, George Bosilca, Marc Gamell, Keita Teranishi, Manish Parashar, and Jack Dongarra. Practical scalable consensus for pseudo-synchronous distributed systems. In *SC*, 2015.

[86] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(4):10–15, 1962.

[87] Seyed H. Hosseini, Jon G. Kuhl, and Sudhakar M. Reddy. A diagnosis algorithm for distributed computing systems with dynamic failure and repair. *IEEE Transactions on Computers*, 33(3):223–233, 1984.

[88] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance Evaluation of Adaptive MPI. In *PPoPP*, 2006.

[89] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers (TOC)*, C-33(7):518–528, June 1984.

[90] Joshua Hursey and Richard L. Graham. Building a fault tolerant MPI application: A ring communication example. In *IPDPS Workshops*, pages 1549–1556, 2011.

[91] Joshua Hursey, Richard L. Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G. Solt. Run-through stabilization: An MPI proposal for process fault tolerance. In *EuroMPI*, volume 6960, pages 329–332, 2011.

[92] Joshua Hursey, Thomas Naughton, Geoffroy Vallée, and Richard L. Graham. A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI. In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors, *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 255–263. Springer, 2011.

[93] Van Jacobson and Michael J. Karels. Congestion avoidance and control. *ACM Computer Communications Review*, 18(4):314–329, August 1988.

[94] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice Hall, 1994.

[95] M Jette and M Grondona. SLURM: Simplex linux utility for resource management, November 30 2007.

[96] Yulu Jia, George Bosilca, Piotr Luszczek, and Jack J. Dongarra. Parallel reduction to hessenberg form with algorithm-based fault tolerance. In *SC*, pages 1–11, 2013.

[97] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *FTCS*, 1987.

[98] Elias Procópio Duarte Jr., Luiz Carlos Pessoa Albini, Alessandro Brawerman, and André Luiz Pires Guedes. A hierarquical distributed fault diagnosis algorithm based on clusters with detours. In *LANOMS*, 2009.

[99] Elias Procópio Duarte Jr., Andréa Weber, and Keiko Verônica Ono Fonseca. Distributed diagnosis of dynamic events in partitionable arbitrary topology networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1415–1426, 2012.

[100] Ronald Bianchini Jr., Ken Goodwin, and Daniel S. Nydick. Practical application and implementation of distributed system-level diagnosis theory. In *FTCS*, pages 332–339. IEEE Computer Society, 1990.

[101] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA*, 1993.

[102] S. Kamal and Carl V. Page. Intermittent faults: A model and a detection procedure. *IEEE Transactions on Computers*, C-23(7):713–719, 1974.

[103] Kishor Kharbas, Donghoon Kim, Torsten Hoefler, and Frank Mueller. Assessing HPC failure detectors for MPI jobs. In *PDP*, pages 81–88, 2012.

[104] Jonathan Kirsch and Yair Amir. Paxos for system builders: An overview. In *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 3:1–3:6, New York, NY, USA, 2008. ACM.

[105] J. W. Krull, A. M. Molina, and J. Wu. Evaluation of a fault tolerant distributed broadcast algorithm in hypercube multicomputers. In Jagan P. Agrawal, Vijay Kumar, and Virgil Wallentine, editors, *Proceedings of the Conference on Computer Science*, pages 459–466, New York, NY, USA, March 1992. ACM Press.

[106] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, Cambridge, UK, March 2011.

[107] J. G Kuhl. Fault diagnosis in computing networks. Technical report, Dep. Elec. Comput. Eng. Univ. of Iowa, Aug 1980.

[108] J. G. Kuhl and S. M. Reddy. Distributed fault-tolerance for large multiprocessor systems. In *Proceedings of the 7th annual symposium on Computer Architecture*, ISCA '80, pages 23–30, New York, NY, USA, 1980. ACM.

[109] J. G. Kuhl and S. M. Reddy. Fault diagnosis in fully distributed systems. In *Proc. of the 11th IEEE Fault-Tolerant Computing Symp.*, pages 100–105, June 1981.

[110] Lawrence Livermore National Lab. Algebraic multigrid solver for linear systems. `https://codesign.llnl.gov/amg2013.php`, year = 2017, note = Acessado em 01/02/2017.

[111] Ignacio Laguna, Todd Gamblin, Martin Mohror, Kathryn Schulz, Howard Pritchard, and Nickolas Davis. A global exception fault tolerance model for mpi. *Workshop on Exascale MPI*, 2014.

[112] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, and Bronis R. de Supinski. Evaluating user-level fault tolerance for mpi applications. In *EuroMPI/ASIA*, pages 57–62, 2014.

[113] Lamport. Paxos made simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 32, 2001.

[114] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[115] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

[116] L.A. Laranjeira, M. Malek, and R. Jenevein. On tolerating faults in naturally redundant algorithms. In *Reliable Distributed Systems, 1991. Proceedings., Tenth Symposium on*, pages 118–127, Sep 1991.

[117] Arnaud Lefray, Thomas Ropars, and André Schiper. Replication for send-deterministic MPI HPC applications. In *FTXS Workshop at HPDC*, 2013.

[118] Pierre Lemarinier, Aurelien Bouteiller, Geraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. *International Journal of High Performance Computing and Networking*, 2:146–155, 2006.

[119] J. Lifflander, E. Meneses, H. Menon, P. Miller, S. Krishnamoorthy, and L. V. Kale. Scalable replay with partial-order dependencies for message-logging fault tolerance. In *CLUSTER*, 2014.

[120] Xunyun Liu, Xinhai Xu, Xiaoguang Ren, Yuhua Tang, and Ziqing Dai. A message logging protocol based on user level failure mitigation. In *ICA3PP*, 2013.

[121] M. H. MacDougall. *Simulating Computer Systems. Techniques and Tools*. Computer Systems Series. MIT, 1987. Discrete Event Simulation mittels SMPL.

[122] Miroslaw Malek. A comparison connection assignment for diagnosis of multiprocessor systems. In *ISCA*, pages 31–36, 1980.

[123] S. Mallela and G.M. Masson. Diagnosable systems for intermittent faults. *IEEE Transactions on Computers*, 27(6):560–566, 1978.

[124] P. J. Marandi, M. Primi, and F. Pedone. Multi-ring paxos. In *DSN*, 2012.

[125] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *DSN*, 2010.

[126] Gerald M. Masson, Douglas M. Blough, and Gregory F. Sullivan. Fault-tolerant computer system design. chapter System Diagnosis, pages 478–536. Prentice-Hall, Inc., 1996.

[127] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC*, 2010.

[128] MPI-Forum. User-level failure mitigation. `https://bitbucket.org/icldistcomp/ulfm/`, year = 2017, note = Acessado em 01/02/2017,.

[129] MPI Forum. Document for a standard message-passing interface 3.1. Technical report, University of Tennessee, http://www.mpi-forum.org/docs/mpi-3.1, 2015.

[130] mpich.org. High-performance portable mpi. `http://www.mpich.org/`, 2017. Acessado em 26/01/2017.

[131] Sape Mullender. *Distributed Systems:*. Addison Wesley, 2nd ed. edition, 1993.

[132] K Nakajima. A new approach to system diagnosis. *Proc. of the 19th Allerton Conf. on Communication, Control and Computing*, pages 697–706, 1981.

[133] NCSC University of Illinois. Blue waters. `https://bluewaters.ncsa.illinois.edu/`, year = 2017, note = Acessado em 01/02/2017,.

[134] open mpi.org. Open mpi: Open source high performance computing. `https://www.open-mpi.org/`, 2017. Acessado em 26/01/2017.

[135] Parallel Cut Tree Algorithms Page. Gusfield's algorithm implementation. `http://www.deinfo.uepg.br/~jcohen/parallel-cuttree.html`, year = 2017, note = Acessado em 01/02/2017.

[136] Behrooz Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[137] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *SC*, 2003.

[138] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, PDS-9(10):972–986, 1998.

[139] Preparata, Metze, and Chen. On the connection assignment problem of diagnosable systems. In *IEEE Transactions on Electronic Computers*, volume 16. 1967.

[140] Michael J. (Michael Jay) Quinn. *Parallel programming in C with MPI and OpenMP*. McGraw-Hill, pub-MCGRAW-HILL:adr, 2003.

[141] Raghunath Rajachandrasekar, Xavier Besseron, and Dhabaleswar K. Panda. Monitoring and predicting hardware failures in HPC clusters with FTB-IPMI. In *IPDPS Workshops*, pages 1136–1143, 2012.

[142] Sridharan Ranganathan, Alan D. George, Robert W. Todd, and Matthew C. Chidester. Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. *Cluster Computing*, 4:197–209, 2001.

[143] Sampath Rangarajan, Anton T. Dahbura, and Eric A. Ziegler. A distributed system-level diagnosis algorithm for arbitrary network topologies. *IEEE Transactions on Computers*, 44(2):312–334, 1995.

[144] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. Technical report, Cornell University, May 1998.

[145] Rolf Riesen, Kurt Ferreira, Dilma Da Silva, Pierre Lemarinier, Dorian Arnold, and Patrick G. Bridges. Alleviating scalability issues of checkpointing protocols. In *SC*, 2012.

[146] Thomas Ropars, Tatiana V. Martsinkevich, Amina Guermouche, André Schiper, and Franck Cappello. SPBC: leveraging the characteristics of MPI HPC applications for scalable checkpointing. In *SC*, page 8, 2013.

[147] Thomas Ropars and Christine Morin. Active optimistic message logging for reliable execution of MPI applications. In *Euro-Par*, 2009.

[148] Thomas Ropars and Christine Morin. Improving message logging protocols scalability through distributed event logging. In *Euro-Par*, 2010.

[149] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Vishal Sahay, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *The International Journal of High Performance Computing Applications*, (4):479–493, 2005.

[150] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(3):299, December 1990.

[151] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–351, 2010.

[152] Marco Serafini, Andrea Bondavalli, and Neeraj Suri. Online diagnosis and recovery: On the choice and impact of tuning parameters. *IEEE Transactions on Dependable and Secure Computing*, 4(4), 2007.

[153] Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V. Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A. Chien, Paul Coteus, Nathan DeBardeleben, Pedro C. Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications*, 28(2):129–173, 2014.

[154] Georg Stellner. Cocheck: Checkpointing and process migration for MPI. In *IPPS*, pages 526–531, 1996.

[155] Arun Subbiah and Douglas M. Blough. Distributed diagnosis in dynamic fault environments. *IEEE Transactions on Parallel and Distributed Systems*, PDS-15(5):453–467, May 2004.

[156] Guang Suo, Yutong Lu, Xiangke Liao, Min Xie, and Hongjia Cao. Nr-mpi: A non-stop and fault resilient mpi. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 190–199, Dec 2013.

[157] Sing-Ban Tien and C. S. Raghavendra. Algorithms and bounds for shortest paths and diameter in faulty hypercubes. *IEEE Transactions on Parallel and Distributed Computing*, PDS-4(6):713–718, June 1993.

[158] D. Tiwari, S. Gupta, and S.S. Vazhkudai. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *DSN*, pages 25–36, 2014.

[159] Tzeng and Chen. Structural and tree embedding aspects of incomplete hypercubes. *IEEE Transactions on Computers*, 43, 1994.

[160] Paulo Veríssimo. Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.

[161] Bruce Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. *Hypercube Multiprocessors*, 1987:292–299, 1987.

[162] John Paul Walters and Vipin Chaudhary. Replication-based fault tolerance for MPI applications. *IEEE Transactions on Parallel and Distributed Systems*, 20(7):997–1010, 2009.

[163] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration and back migration in HPC environments. *Journal of Parallel and Distributed Computing*, 72(2):254–267, 2012.

[164] Rui Wang, Erlin Yao, Mingyu Chen, Guangming Tan, Pavan Balaji, and Darius Buntinas. Building algorithmically nonstop fault tolerant MPI programs. In *HiPC*, pages 1–9, 2011.

[165] G. Zheng, Xiang Ni, and L. V. Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *DSN Workshop 2012*, 2012.

Apêndice A

Anexo

Este anexo apresenta resultados preliminares obtidos logo após a definição do modelo de diagnóstico baseado em testes imperfeitos. Na ocasião, o DGRP (Seção 4) era chamado de Núcleo Dinâmico de Processos Estáveis (*Dynamic Core of Stable Processes* - DCSP). Um processo era classificado como estável ou instável. No resultado apresentado a seguir, um algoritmo de difusão atômica foi executado entre os processos do DCSP, de forma semelhante a que o algoritmo *Hyperquicksort* (Seção 4.2.1) foi executado usando o DGRP.

A.1 Difusão Atômica

Este estudo de caso faz uso do modelo para monitorar os processos do sistema. O objetivo do trabalho é descrever duas variantes de implementação da difusão atômica executada sobre o núcleo estável.

A difusão atômica, (*atomic broadcast* ou *total order broadcast*), tem por objetivo garantir que todos os processos corretos entreguem o mesmo conjunto de mensagens na mesma ordem total [131]. A difusão atômica é composta pelas três propriedades da difusão confiável (*reliable broadcast*) mais a propriedade de ordem total. As propriedades da difusão confiável asseguram que uma mensagem enviada por um processo correto será entregue, sem duplicação, a todos os demais processos corretos. Se um processo falhar após ter difundido uma mensagem, a mensagem será entregue a todos os demais se pelo menos um processo tiver entregue a mensagem. A propriedade de ordem total define que se dois processos corretos *p* e *q* entregam duas mensagens *m* e *m'*, então *p* entrega *m* antes de *m'* se e somente se *q* entrega *m* antes de *m'* [49].

A difusão atômica é um serviço fundamental para manter a consistência em base de dados replicadas. O trabalho de [49] apresenta diversas abordagens para a implementação da difusão atômica. No trabalho de Camargo et al. [36], duas dessas abordagens foram implementadas. Em ambas, somente os processos que compõem o núcleo de processos estáveis enviam suas mensagens. Dessa forma, as mensagens são entregues na mesma ordem a todos os processos do núcleo estável. A primeira abordagem é baseada em consenso. Nesse caso, a ordenação das mensagens é realizada por consecutivas execuções do consenso pelos processos com base nas mensagens recebidas.

A segunda abordagem considera que um processo é responsável por definir a sequência das mensagens. Nessa abordagem foram implementadas duas variantes: a UB (*unicast-broadcast*) e a BB (*broadcast-broadcast*). Na variante UB, um processo envia sua mensagem ao sequenciador, que insere na mensagem uma etiqueta com seu número de sequência e a envia aos demais processos. Na variante BB, o processo envia sua mensagem a todos os processos. O sequenciador,

ao receber a mensagem, envia aos demais apenas o número de sequência daquela mensagem e não a mensagem em si. A segunda variante apesar de gerar mais mensagens diminui a carga sobre o sequenciador.

A abordagem baseada no consenso foi implementada através do algoritmo de consenso Paxos, descrito na Seção 2.1.2.

Implementação das Abordagens de Difusão Atômica

A difusão atômica é implementa sobre um núcleo de processos estáveis, conforme apresenta a Figura A.1. Uma das abordagens implementada faz uso do algoritmo de consenso Paxos. Todas as primitivas implementadas consideram apenas os processos do núcleo estável, isto é, as mensagens são enviadas e recebidas apenas pelos processos do núcleo estável. A difusão atômica foi implementada conforme a descrição a seguir.

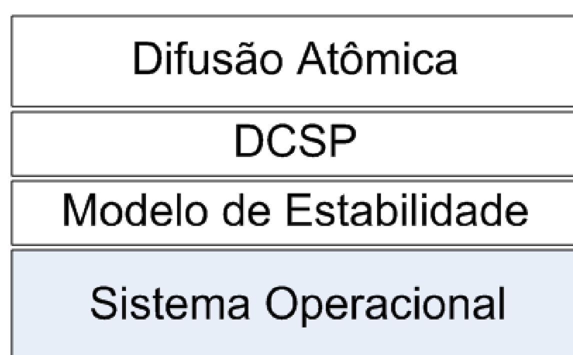


Figura A.1: Arquitetura do sistema implementado.

Cada processo p_i do núcleo estável envia sua mensagem aos demais processos do núcleo estável. As mensagens recebidas pelos processos do núcleo estável são armazenadas em um vetor de mensagens recebidas. Há uma constante chamada k que determina a quantidade de mensagens recebidas que o processo deve aguardar para então invocar uma instância de consenso. O líder, ou coordenador, do consenso será o primeiro nodo do núcleo estável.

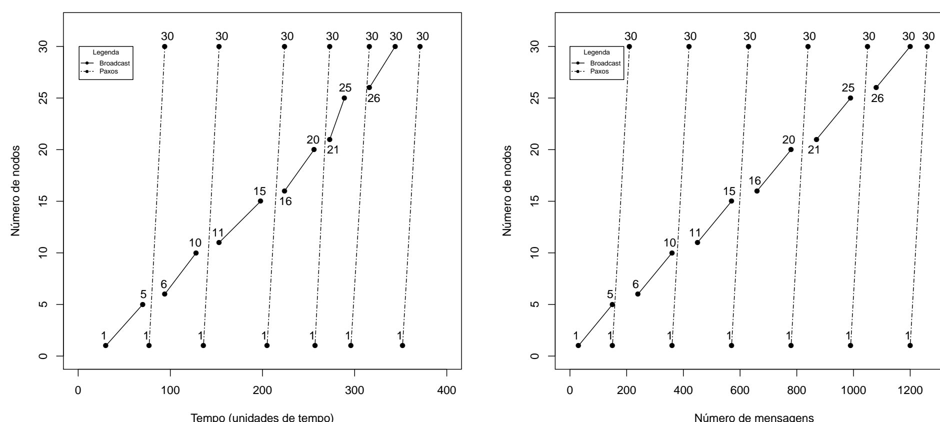
Uma instância de consenso inicia quando o coordenador envia uma mensagem aos *acceptors* na primeira fase, de acordo com a Figura 2.1. Se o líder receber resposta da maioria dos processos do núcleo estável poderá então propor o seu valor na segunda fase do Paxos. Uma vez que a maioria dos processos do núcleo estável aceita o valor do coordenador, então o coordenador comunica a decisão a todos os processos do núcleo estável, que também são *learners*.

Na segunda abordagem da difusão atômica implementada, o primeiro processo do núcleo estável assume a função de sequenciador das mensagens. O trabalho foi simulado e os resultados de simulação da estratégia de monitoramento e das abordagens de difusão atômica são apresentados na próxima seção.

A.1.1 Implementação de Difusão Atômica sobre o DCSP

A seguir são apresentados o conjunto de resultados da difusão atômica executada sobre um núcleo de processos estáveis. A simulação é efetuada através da linguagem SMPL (*Simple Portable Simulation Language*) [121]. Os resultados da simulação apresentam a latência e o número total de mensagens das duas abordagens implementadas para a difusão atômica. No trabalho Camargo et al. [36] resultados da simulação e obtenção do núcleo estável também são

apresentados. Nos resultados a seguir os nodos iniciam a transmissão de suas mensagens na unidade de tempo 30 após a formação do núcleo estável. Todos os nodos estáveis enviam uma mensagem entre si. Assume-se também que as mensagens enviadas sofrem atraso, porém não são perdidas. Os processos podem apresentar velocidades de execução diferentes e permanecem estáveis durante a execução da difusão atômica. Em todos os cenários há dois processos instáveis.



(a) Latência para k=5. **(b)** Número de mensagens para k=5.

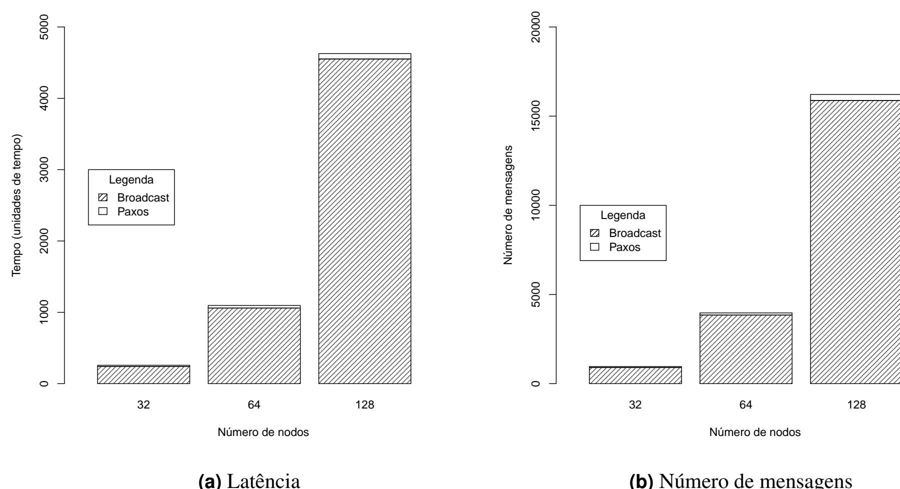
Figura A.2: Difusão atômica com 32 nodos, 2 nodos instáveis e k = 5.

O primeiro gráfico, Figura A.2, é de um cenário com 32 nodos e $k = 5$. Conforme descrito anteriormente, na Seção A.1, a constante k define a quantidade de mensagens que um processo deve receber para então iniciar uma rodada de consenso. As Figuras A.2(a) e A.2(b) apresentam a latência e o número de mensagens respectivamente considerando as sucessivas instâncias de consenso. O gráfico destaca a latência e o número de mensagens da difusão (*broadcast*) e a latência e o número de mensagens do consenso (Paxos).

O gráfico da Figura A.2(a), informa que o consenso, em cada execução, leva aproximadamente 16 unidades de tempo e a transmissão das mensagens em media 33 unidades de tempo. A latência total foi de aproximadamente 380 unidades de tempo. A Figura A.2(b), que apresenta o número de mensagens, informa que a cada 5 mensagens o consenso é chamado. Como são 30 processos do núcleo estável, 150 mensagens são transmitidas ao todo até a execução do consenso. O consenso em si demanda 60 mensagens a cada execução.

A Figura 6 apresenta a latência e o número de mensagens da difusão atômica em cenários com 32, 64 e 128 nodos totais com a constante k igual ao número de processos estáveis. Também há a distinção entre a latência e o número de mensagens da difusão (*broadcast*) e do consenso (Paxos). De acordo com a Figura A.3(a), percebe-se que 258 unidades de tempo são necessárias para a execução completa da difusão atômica pelo núcleo estável para o cenário de 32 nodos, enquanto que 1097 unidades são requisitadas no cenário de 64 nodos e 4627 para o cenário de 128 nodos. Como pode ser visto, o custo do consenso para a latência total da difusão atômica é mínimo.

Em relação ao número de mensagens, a Figura A.3(b), conclui-se que 960 mensagens totais são trocadas pelo núcleo na execução da difusão atômica no cenário de 32 nodos, 3968 mensagens são necessárias no cenário de 64 nodos e 16218 são trocadas no cenário de 128 nodos. Considerando somente o Paxos, o número de mensagens enviadas chega a 60, 124 e 342 para os cenários de 32, 64 e 128 nodos, respectivamente.

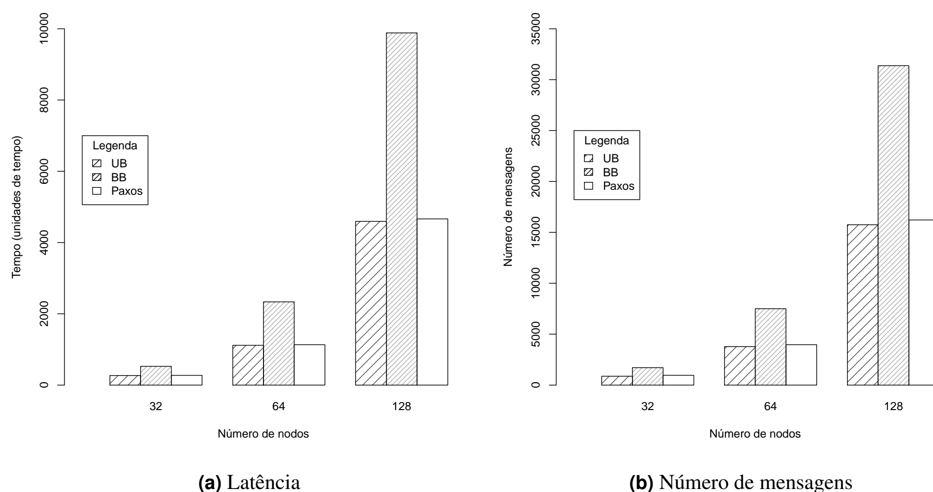


(a) Latência **(b)** Número de mensagens

Figura A.3: Difusão atômica baseada em consenso.

Conforme pode ser observados, a latência e o número de mensagens nos cenários apresentados cresce consideravelmente à medida que aumenta o número de nodos do sistema. Vale observar que os cenários simulados consideram o pior caso, onde todos os nodos enviam mensagens a todos os demais.

A Figura A.4(a) e a Figura A.4(b) fazem um comparativo da latência e do número de mensagens totais da difusão atômica implementada através das três abordagens: UB (*unicast-broadcast*), BB (*broadcast-broadcast*) e consenso (*k* igual número de processos estáveis). Vale lembrar que na variante UB um processo envia sua mensagem ao sequenciador, que insere na mensagem uma etiqueta com seu número de sequência e a envia aos demais processos. Na variante BB, o processo envia sua mensagem a todos os processos. O sequenciador, ao receber a mensagem, envia aos demais apenas o número de sequência daquela mensagem e não a mensagem em si.



(a) Latência **(b)** Número de mensagens

Figura A.4: Comparação das abordagens de difusão atômica.

Os cenários comparados são os de 32, de 64 e de 128 nodos totais, considerando 2 nodos instáveis. Em relação a latência (Figura A.4(a)), percebe-se que a variante UB do sequenciador fixo e o consenso caminham muito próximas entre si. No entanto a latência da variante BB do

sequenciador fixo aumenta consideravelmente a medida que cresce o número de nodos totais. Em números, no contexto de 128 nodos, a latência da abordagem BB chega ao patamar de 9884 unidades de tempo, o dobro da latência das abordagens UB (4594 unidades de tempo) e consenso (4663 unidades de tempo).

Em relação ao número de mensagens nas três abordagens (Figura A.4(b)), verifica-se que o mesmo mais do que quadruplica a medida que o número de nodos aumenta. Em termos numéricos, a abordagem UB começa com 870 mensagens no contexto de 32 nodos, sobe para 3782 ao considerar 64 nodos e encerra com 15750 mensagens no contexto de 128 nodos. Já a abordagem BB inicia com 1711 mensagens com 32 nodos, aumenta para 7503 com 64 nodos e finaliza com 31375 mensagens trocadas ao considerar 128 nodos. O consenso tem número de mensagens próxima a variante UB.

Apêndice B

Publicações

Neste anexo estão listados as publicações obtidas durante o desenvolvimento da tese, bem como as contribuições no âmbito do Grupo de Pesquisa LARSIS - Laboratório de Redes e Sistemas Distribuídos.

B.1 Trabalhos Publicados no Âmbito da Tese

Trabalhos completos publicados em anais de congressos

1. **Edson T. Camargo**, Fernando Pedone, Elias Duarte Jr. "A Consensus-based Fault-Tolerant Event Logger for High Performance Applications". In: 23rd International European Conference on Parallel and Distributed Computing (Euro-Par' 2017), pp. 1-12, Santiago de Compostela, Spain. (*submitted*)

2. **Edson T. Camargo**, Elias Duarte Jr., Fernando Pedone. "Um Protocolo Pessimista para Registro de Mensagens Baseado em um *Event Logger* Distribuído e Tolerante a Falhas". In: XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2017), pp. 1-14, Belém, PA, Brasil, 2017, (*aceito para publicação*).

3. **Edson T. Camargo**, Elias P. Duarte Jr., "Running Resilient MPI Applications on a Dynamic Group of Recommended Processes". In: The 7th IEEE Latin American Symposium on Dependable Computing (LADC'2016), pp. 1-10, Cali, Colombia, 2016. (*Best paper award*)

4. **Edson T. Camargo**, Elias P. Duarte Jr. "Network Monitoring with Imperfect Tests". In: ACM SIGCOMM Workshop on Fostering Latin-American Research in Data Communication Networks (LANCOMM 2016), pp. 1-3, Florianópolis, Brazil, 2016.

5. **Edson T. Camargo**, Elias P. Duarte Jr. "Uma Implementação MPI Tolerante a Falhas do Algoritmo *Hyperquicksort*". In: XVII Workshop de Testes e Tolerância a Falhas (WTF'2016), XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2016), pp. 1-14, Salvador, Brasil, 2016.

6. Weyne Cassou-Pietniczka, **Edson T. de Camargo**, e Elias Duarte Jr. "Implementação de difusão atômica baseada em diagnóstico com testes imperfeitos". In: Anais do 34o Congresso da Sociedade Brasileira de Computação, 33o Concurso de Trabalhos de Iniciação Científica (CTIC'2014), pp. 1-10, Brasília, DF, 2014. (*premiado entre os 10 melhores trabalhos*)

7. **Edson T. Camargo**, Elias P. Duarte Jr., Weyne Cassou-Pietniczka. "Diagnóstico distribuído com testes imperfeitos aplicado à detecção de estabilidade em sistemas baseados em mpi". In: Anais da 15o edição do Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD'2014), pp. 1-12, São José dos Campos, SP, 2014.

8. **Edson T. Camargo**, Elias P. Duarte Jr. "Running Fault-Tolerant MPI-based Applications in Unstable Systems". (extended abstract) Workshop on Exascale MPI (ExaMPI), at The Supercomputing Conference 2014 (SC'2014), pp. 1-4, New Orleans, USA, 2014.

Trabalhos em Periódicos

1. **Edson T. Camargo**, Elias P. Duarte Jr., "A System-level Diagnosis Model with Imperfect Tests Applied to Recommend MPI Processes". The Journal of the Brazilian Computer Society (JBCS), 2017 (*accepted for publication*)

2. Weyne Cassou-Pietniczka, **Edson T. Camargo**, Elias P. Duarte Jr. "Implementação de Difusão Atômica Baseada em Diagnóstico com Testes Imperfeitos". Revista Eletrônica de Iniciação Científica, v. 14, p. 1, 2014.

B.2 Trabalhos Publicados no Âmbito do Grupos de Pesquisa

1. Giovanni V. Souza, Rogério Turchetti, **Edson T. Camargo**, Elias Duarte Jr. "Uma Função Virtualizada de Rede para a Sincronização Consistente do Plano de Controle em Redes SDN". In: XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2017), pp. 1-14, Belém, PA, Brasil, 2017, (*aceito para publicação*).