Universidade Federal Fluminense

MATHEUS BERSOT SIQUEIRA BARROS

Tolerância a Falhas em uma Aplicação de Processamento Sísmico para Clusters Multicore

NITERÓI

2011

MATHEUS BERSOT SIQUEIRA BARROS

Tolerância a Falhas em uma Aplicação de Processamento Sísmico para Clusters Multicore

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre em Computação.

Orientadores:
Lúcia Maria Assumpção Drummond
Maria Cristina Silva Boeres

Universidade Federal Fluminense

NITERÓI

2011

Tolerância a Falhas em uma Aplicação de Processamento Sísmico para *Clusters Multicore*

Matheus Bersot Siqueira Barros

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre em Computação.

Aprovada por:

Profª. D.Sc. Lúcia M. A. Drummond / IC-UFF(Presidente)

Profª. Ph.D. Maria Cristina Silva Boeres / IC-UFF

D.Sc. André Bulcão / CENPES-PETROBRAS

Prof. Ph.D. Eugene Francis Vinod Rebello / IC-UFF

Profª Ph.D. Luciana Bezerra Arantes / Paris 6

Niterói, 5 de Setembro de 2011.

Agradecimentos

Agradeço primeiramente aos meus pais, pelo amor imensurável, por todo tempo dedicado a mim, pelo exemplo de vida, motivação e apoio nas horas mais difíceis. A vocês meu eterno amor e gratidão.

Às professoras Lúcia Maria de Assumpção Drummond e Maria Cristina Silva Boeres pelos ensinamentos passados, pela confiança e por todo apoio dado durante a realização desse trabalho.

Ao Alexandre Domingues Gonçalves pelo imenso apoio dado na realização desse trabalho e pelas palavras de otimismo.

Ao professor Eugene Francis Vinod Rebello pelos ensinamentos passados, pela atenção e por permitir a reserva das máquinas do cluster Oscar, sem o qual não seria possível realizar os experimentos apresentados nesse trabalho.

À todos integrantes do Projeto Petrobrás pelos conselhos, ideias e apoio dado durante a confecção desse trabalho.

Ao Carlos Eduardo Cabral da Cunha pelo imenso suporte oferecido, até nos finais de semana, no uso do cluster Oscar e pela confiança.

Ao professor Otton Teixeira da Silveira Filho, Jacques Alves da Silva, Diego Nunes Brandão e Alexandre Sena pelos ensinamentos passados e pela atenção.

À todos os meus familiares por todo amor, carinho e atenção.

Aos amigos que fiz por aqui, pela atenção e pela amizade verdadeira.

Aos amigos, que mesmo distantes, se fizeram presentes com palavras de apoio, mostrando verdadeira amizade.

À CAPES pelo apoio financeiro.

"A persistência é o menor caminho para o êxito."

Charles Chaplin

Resumo

Aplicações de processamento sísmico são usadas para identificar estruturas geológicas onde reservatórios de petróleo e gás podem ser encontrados. Com empresas petrolíferas procurando melhor precisão sobre grandes regiões geográficas, tais aplicações requerem *clusters* maiores para manter o tempo de execução razoável. No entanto, a combinação de tempos de execução razoáveis e clusters com maior número de recursos aumenta a probabilidade de ocorrência de falhas durante a execução da aplicação. Com objetivo de tratar este problema em termos práticos para uma aplicação real, este trabalho descreve um mecanismo de tolerância a falhas no nível de aplicação composto por processos de detecção e recuperação, que utilizam o detector da classe $\diamond Q$ e o protocolo de gravação checkpoint não coordenado, respectivamente. Falhas são consideradas permanentes e ocorrem uma por vez em nós ou *links* de comunicação. Os experimentos mostram a viabilidade e a eficiência do mecanismo proposto no contexto da aplicação de processamento sísmico.

Palavras-chave: Tolerância a Falhas, Clusters multicore, Aplicação RTM.

Abstract

Seismic processing applications are used to identify geological structures where reservoirs of oil and gas may be found. With oil companies seeking better precision over larger geographical regions, such applications require larger clusters to keep execution times reasonable. However, the combination of longer run times and clusters with greater numbers of resources increases the probability of faults during the execution. To address this issue, this work describes an application-level fault tolerance mechanism composed by detection and recovery processes, which use the detector of the class $\diamond Q$ and the non-coordinated recovery protocol, respectively. Faults are considered permanent and occur one at time in nodes or communication links. Experiments show the feasibility and efficiency of the proposed mechanism in the context of this seismic processing application.

Keywords: Fault Tolerance, Multicore clusters, RTM application.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Ao longo dos últimos anos, a tecnologia empregada nos componentes computacionais atuais avançou e os preços dos mesmos sofreram redução considerável. Consequentemente, o uso de *clusters* de computadores para executar aplicações de alto desempenho aumentou, pois eles se tornaram financeiramente mais acessíveis, mais confiáveis e escaláveis [16]. Entretanto, mesmo em ambientes computacionais do estado da arte, muitas aplicações requerem dias ou mesmo meses para completar suas execuções. Dado que esses sistemas são compostos por milhares de componentes, onde qualquer um pode falhar, uma das questões chave é a execução continuada destas aplicações mesmo na ocorrência de falhas destes componentes.

Os usuários dos sistemas de alto desempenho, em geral, fazem uso de um ambiente de programação paralela/distribuída que provê abstração desses sistemas através de uma linguagem de programação ou de uma biblioteca. As aplicações desenvolvidas nestes ambientes de programação geralmente usam o paradigma de troca de mensagens como o meio de comunicação entre tarefas/processos. Chameleon [35], CharmC++ [41], P4 [15], PVM [32] e MPI [49] são exemplos de ambientes de programação paralela/distribuída que podem ser encontrados na literatura. Dentre eles, a interface de troca de mensagens MPI (Message Passing Interface) se tornou um padrão atual na construção de aplicações paralelas/distribuídas [40, 5, 62].

Embora o padrão MPI não forneça mecanismos que permitam uma aplicação continuar em presença de falhas, verifica-se que algumas de suas implementações, como MPI-FT [47], MPI/FT [5] e MPICH-V [11] dispõem desse serviço. Ainda assim, não é possível projetar uma aplicação portável para todas essas implementações ao mesmo tempo, uma vez que a aplicação deve se adequar aos mecanismos propostos por cada uma delas. No caso de implementações comerciais do padrão MPI, tal como a biblioteca Intel MPI [39] que foca

na maximização do desempenho, os programadores devem recorrer a implementação de um mecanismo de tolerância a falhas no nível de aplicação, onde uma estratégia específica pode ser adotada com a intenção de minimizar o impacto negativo no desempenho de uma execução livre de falhas.

Neste trabalho, foi utilizada uma aplicação industrial de processamento sísmico que determina as propriedades e posições de várias camadas de estratos subterrâneos, com o objetivo de identificar estruturas geológicas onde reservatórios de petróleo e gás podem potencialmente ser encontrados. Essa aplicação, em MPI, pode necessitar de meses de tempo de computação mesmo em um *cluster* com centenas de núcleos de processamento [14], sendo assim prudente tomar algumas medidas para proteger a aplicação das falhas de componentes. Dessa forma, a principal contribuição desta dissertação é propor um mecanismo de falhas agregado a aplicação alvo e avaliá-lo quanto ao impacto provocado no desempenho desta aplicação usando a biblioteca Intel MPI.

O restante do texto está organizado da seguinte forma. No Capítulo 2, as características inerentes à aplicação alvo são descritas. No Capítulo 3, são apresentadas as técnicas existentes para detecção e recuperação de falhas, assim como, o mecanismo de tolerância a falhas proposto nesse trabalho. Além disso, alguns trabalhos relacionados são introduzidos, destacando-se as suas diferenças em relação à solução proposta neste trabalho. Na Seção 4, são expostos os resultados experimentais para o mecanismo de tolerância a falhas descrito anteriormente. Por fim, na Seção 5, apresenta-se um resumo do trabalho proposto, destacando os resultados alcançados e os possíveis trabalhos futuros.

Capítulo 2

Aplicação de Processamento Sísmico

A crescente demanda por petróleo no mundo tornou imperativo o aumento na procura por novas reservas deste insumo. Através do conhecimento das estruturas da subsuperfície da Terra é possível detectar de maneira mais precisa a existência de petróleo em determinadas regiões, além de permitir a definição das melhores técnicas de extração. Este trabalho lida com a aplicação de processamento sísmico desenvolvida em (BULCÃO, 2004) que utiliza a técnica de Migração Reversa no Tempo (RTM) [6] para gerar um modelo tridimensional das estruturas geológicas presentes em uma determinada área de interesse. A seguir, são abordadas as técnicas utilizadas nesta aplicação e na Seção 2.2, é apresentado o pseudo-código da mesma, assim como, o paradigma usado para realizar sua paralelização.

2.1 Descrição da Aplicação

O primeiro passo para verificar a ocorrência de petróleo em uma certa região é fazer um levantamento sísmico da mesma para obter informações a respeito das estruturas geológicas que se encontram em sua subsuperfície. Em sua execução, dispositivos que produzem energia sísmica de forma controlada, denominados fontes sísmicas, são utilizados para gerar ondas que são propagadas em direção à subsuperfície e receptores são empregados na captação destas ondas refletidas à medida que percorreram as diversas interfaces e camadas de um determinado meio geológico [14]. Para exemplificar o procedimento de levantamento sísmico, apresenta-se na Figura 2.1 um esquema de levantamento em ambiente marinho [23], visto que no Brasil a maior parte da prospecção de petróleo é realizada em águas profundas e ultraprofundas. Neste tipo de levantamento, uma embarcação reboca cabos equipados com hidrofones que são responsáveis por registrar as amplitudes

das ondas refletidas e transformá-las em sinais elétricos. Também ligado a embarcação tem-se um canhão de ar comprimido (*air gun*) utilizado como fonte das ondas sísmicas. Para adquirir dados, o canhão é disparado e as ondas mecânicas resultantes se propagam através da água em direção ao fundo do mar e além da subsuperfície terrestre. Quando uma onda se depara com um novo meio, ocorrem dois processos físicos, uma parte dela é refletida de volta para a superfície enquanto a outra é refratada, se propagando ainda mais por esse novo meio. Além disso, vale ressaltar que as ondas refletidas em diferentes camadas da subsuperfície possuem diferentes amplitudes, devido a influência do meio na velocidade de propagação dessas ondas.

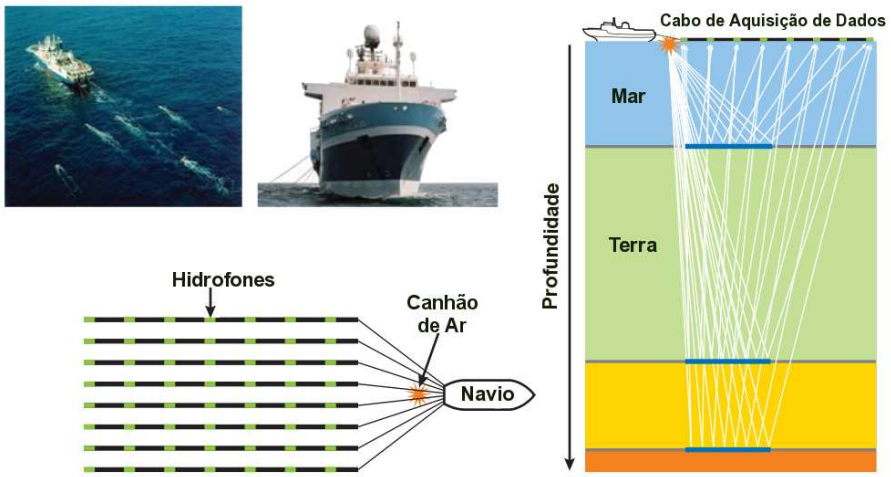


Figura 2.1: Esquema ilustrativo de levantamento sísmico em ambiente marinho [23].

A aquisição de dados sísmicos reais é um processo custoso. Antes de realizá-la, emprega-se a técnica de Modelagem Sísmica [17] para verificar a resposta sísmica, avaliar as possibilidades e limitações dos métodos sísmicos, e otimizar os parâmetros de aquisição de dados [8]. A Modelagem Sísmica consiste na simulação do fenômeno físico da propagação das ondas sísmicas em uma região geológica de interesse adotando um modelo matemático. No trabalho (BULCÃO, 2004), a Modelagem Sísmica foi utilizada, principalmente, com o objetivo de fornecer dados sísmicos de entrada para avaliação do método de Migração Reversa no Tempo que será explicado adiante.

Aplicações voltadas à indústria petrolífera empregam tradicionalmente a Equação Acústica da Onda ou a Equação Elástica da Onda [42] a fim de simular a propagação de ondas sísmicas em domínios não finitos [14]. Na prática, a simulação da propagação de ondas sísmicas é feita em um domínio semi-infinito devido ao tamanho limitado das memórias dos computadores. Dessa forma, uma borda artificial é introduzida no domínio do modelo de propagação de ondas e esta causa reflexões artificiais na simulação. Com o

objetivo de evitar o aparecimento dessas reflexões artificiais, aumenta-se o domínio onde é realizada a simulação. No entanto, esta solução implica em um alto custo computacional, pois em simulações de geofísica, por exemplo, as dimensões envolvidas são tão grandes que podem tornar o processo inviável [13]. Sendo assim, (BULCÃO, 2004) empregou a técnica de Condição de Contorno Não Reflexiva [57] em conjunto com a técnica denominada Zona de Amortecimento [18] para absorver as ondas que alcançam as bordas do domínio. Além disso, devido à complexidade do meio de propagação, não se consegue obter soluções analíticas das equações diferenciais presentes nos modelos regidos por equações da onda. Portanto, (BULCÃO, 2004) utilizou o Método de Diferenças Finitas [48] para obter soluções numéricas aproximadas referentes à Equação Acústica da Onda, assim como a derivada de segunda ordem da função Gaussiana [22] para simular a fonte sísmica e um modelo de velocidades para representar uma área geológica de interesse.

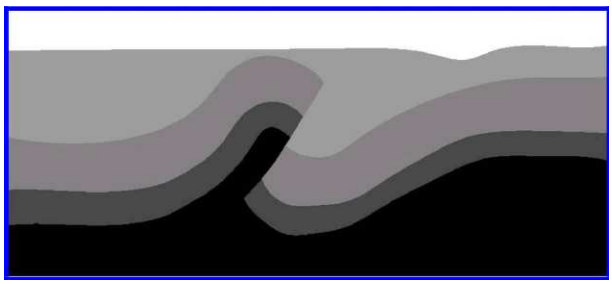


Figura 2.2: Exemplo de um Modelo de Velocidades [14].

O modelo de velocidades consiste em um mapa que indica a velocidade de propagação das ondas nos diferentes meios que podem estar presentes na subsuperfície terrestre. Na Figura 2.2 apresenta-se um exemplo de um modelo de velocidades onde os valores das velocidades de propagação estão representadas em escala da cor cinza, variando de 1500 m/s a 5000 m/s, referentes à cor branca até a cor preta, respectivamente. Além disso, sua dimensão é 4.000 x 8.990 metros, respectivamente, para as coordenadas vertical e horizontal. Dado um modelo de velocidades e sua discretização espacial, a Modelagem Sísmica calcula em cada intervalo de tempo e para cada ponto o campo de ondas ou a amplitude da onda. Na Figura 2.3, ilustra-se o campo de ondas considerando diferentes instantes de tempo, proveniente da propagação das ondas sísmicas a partir da fonte sísmica, sobrepostos a uma imagem do modelo de velocidades apresentada na Figura 2.2.

Após a coleta dos dados, emprega-se a Migração Sísmica [7] que consiste em conjunto de procedimentos pelos quais os campos de ondas registrados são transformados em imagens corretamente posicionadas dos meios refletores em subsuperfície [14]. A determinação de tais imagens pode ser empregada com o intuito de verificar as hipóteses sobre a cons-

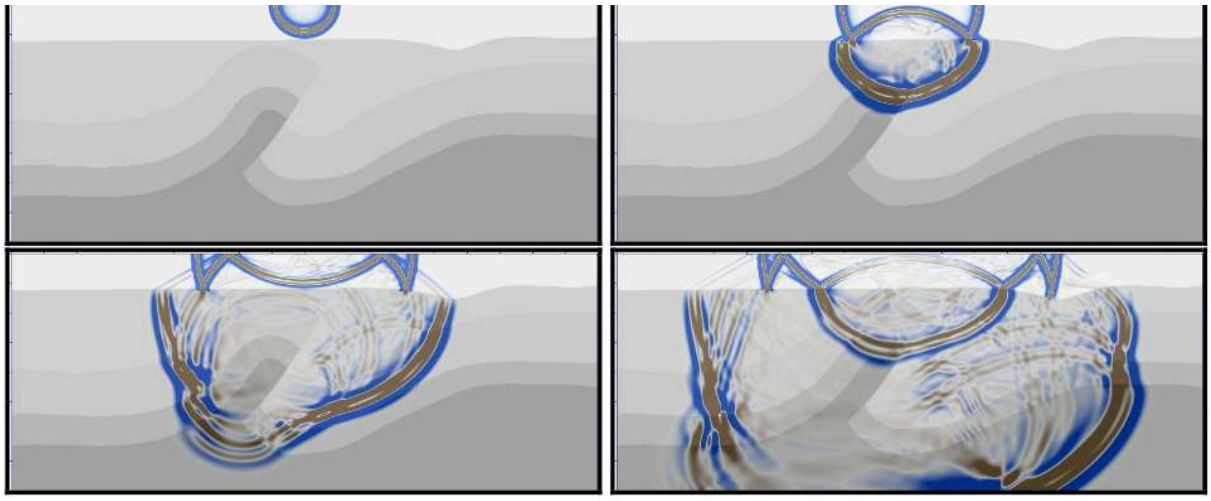


Figura 2.3: Exemplo de campo de ondas considerando diferentes instantes de tempo [14].

trução de determinado modelo geológico, assim como para facilitar a interpretabilidade dos dados sísmicos [8].

Existem diversos métodos de Migração Sísmica, dentre eles, o mais notório é a técnica de Migração Reversa no Tempo (RTM), uma vez que a mesma produz imagens do meio geológico com melhor qualidade e fidelidade. RTM utiliza as técnicas de Modelagem Sísmica para obter dados sísmicos através da simulação da propapagação de ondas à partir da fonte sísmica (*forward propagation*). Então, RTM repete a tarefa de forma inversa, isto é, a partir dos dados registrados pelos receptores (hidrofones), a onda é propagada no sentido inverso do tempo (*backward propagation*). Em seguida, aplica-se uma condição de imagem para indicar a existência de um refletor (sedimentos, rochas, etc.) em uma determinada posição em profundidade onde há a coincidência entre os tempos de trânsito do campo de ondas propagado a partir da fonte sísmica e do campo de ondas propagado à partir dos receptores. No final do processo, a partir do resultado obtido com a aplicação da condição de imagem, tem-se a imagem de saída das estruturas presentes em subsuperfície [3]. Na Figura 2.4 podemos observar um exemplo de uma imagem bidimensional resultante do método RTM.

Um ponto importante a ser destacado é que a Migração Reversa no Tempo apresenta um aparente paradoxo. No início do processo de migração, tem-se o conhecimento do modelo de velocidades. O paradoxo se encontra no fato de que a princípio deve-se saber a resposta antes do início da aplicação das metodologias de migração. Desta forma, na prática, utiliza-se um modelo de velocidades estimado e através de um esquema iterativo, pode-se obter um perfil de velocidades com um melhor grau de confiabilidade [14]. Por fim, ressalta-se que todas as técnicas citadas anteriormente podem ser aplicadas tanto

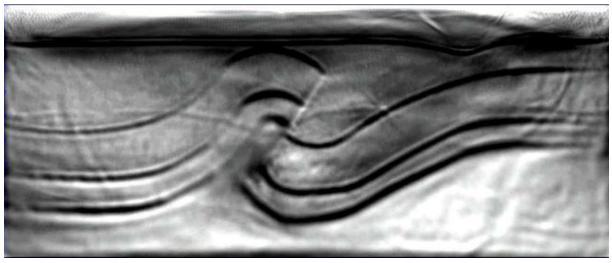


Figura 2.4: Exemplo da imagem resultante do método RTM dado o modelo de velocidades apresentado na Figura 2.2 [14].

para domínios bidimensionais quanto tridimensionais.

2.2 Algoritmo

A recente disponibilidade de processadores *multi-core* permitiu o emprego maior da técnica RTM para imageamento (*imaging*) das estruturas geológicas da subsuperfície, pois a mesma requer grande poder computacional e capacidade de armazenamento de dados em memória para apresentar soluções em tempo hábil [54]. Este trabalho utilizou um algoritmo que reflete a execução de parte da técnica de RTM, isto é, a modelagem da propagação da onda sísmica no sentido progressivo do tempo. Sem perda de expressão, o termo aplicação ou algoritmo RTM será utilizado ao longo deste trabalho.

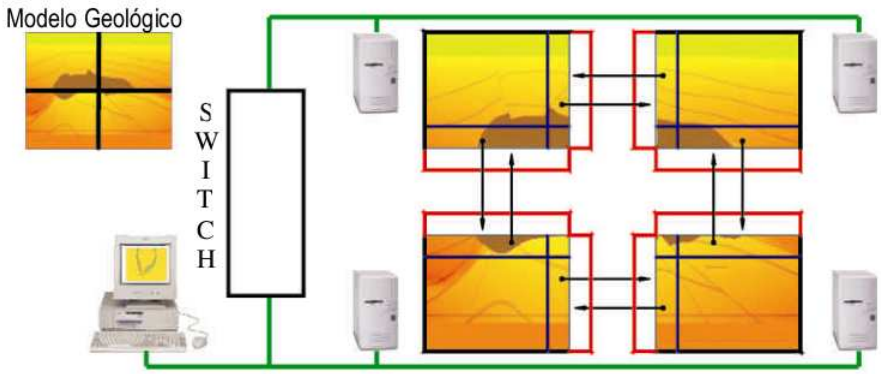


Figura 2.5: Representação esquemática do paradigma Decomposição de Domínio [14].

O algoritmo RTM desenvolvido em (BULCÃO, 2004) analisa um domínio espacial tridimensional. Em sua execução, utilizam-se os recursos do processamento paralelo/distribuído através do uso do paradigma Decomposição de Domínio que é apresentado na Figura 2.5. Este paradigma consiste na resolução de equações diferenciais parciais baseado na decomposição do domínio espacial do problema (modelo geológico ou modelo de velocidades) em subdomínios de mesmo tamanho, que são executados em diferentes núcleos de processa-

mento (*cores*) presentes em diferentes processadores [19]. Além disso, a cada instante de tempo é necessário se compatibilizar a influência da resposta entre as diversas partições e isso é realizado através da troca de mensagens entre elas contendo os pontos pertencentes a borda dos subdomínios [52].

O pseudo-código de parte da aplicação RTM desenvolvida em (BULCÃO, 2004) pode ser visto no Algoritmo 1. Essa aplicação é executada por vários processos distribuídos ao longo dos *cores* dos vários processadores utilizados. Inicialmente, as variáveis e constantes são inicializadas. Em seguida, lê-se o modelo de velocidades e se inicia o processo de decomposição de domínio. A divisão do domínio é feita levando-se em conta a dimensão do modelo de velocidades e o número de processos instanciados. Com o tamanho definido dos subdomínios resultantes, são alocadas as matrizes que irão armazenar dados do problema.

Em cada iteração do laço principal calcula-se o pulso sísmico gerado pela fonte sísmica. A partir dessa informação calcula-se o campo de ondas no sentido progressivo do tempo, para cada ponto do domínio, por meio de uma discretização em diferenças finitas de 10ª ordem no espaço e 2ª ordem no tempo. Dado que a solução para um ponto no espaço 3D não depende somente dos pontos pertencentes a um subdomínio, mas também do contorno das partições vizinhas, no fim de cada passo do laço, a borda das partições vizinhas são trocadas via mensagens. Essa troca de mensagens estabelece a conclusão de um passo da iteração e define um ponto de sincronismo entre os processos participantes.

Algoritmo 1: Algoritmo da Aplicação RTM

```
1  início
2      para todo processo de aplicação p_i faça
           /* inicialização                                          */
3          Inicializa variáveis e constantes;
4          Lê modelo de velocidades;
5          Inicia a decomposição do domínio sobre o Modelo de Velocidades;
6          De acordo com o subdomínio correspondente a p_i e seu tamanho, criam-se
           matrizes que armazenarão os resultados da simulação;
           /* laço principal                                         */
7          enquanto tempo < tempo da simulação faça
8              calcula o pulso sísmico gerado pela fonte sísmica;
9              calcula o campo de ondas;
10             troca borda do subdomínio com as partições vizinhas;
11         fim
12     fim
13 fim
```

2.3 Resumo

O principal objetivo deste capítulo foi descrever as técnicas utilizadas pela aplicação alvo, cujo fim é facilitar o conhecimento das estruturas da subsuperfície da Terra de forma a tornar possível a detecção mais precisa de petróleo em regiões de interesse. Além disso, foi apresentado o algoritmo dessa aplicação que simula a primeira propagação de ondas e é parte fundamental na realização da técnica de Migração Reversa no Tempo. Esse algoritmo é executado por um conjunto de processos, cada processo executando em um recurso e responsável pela simulação de uma parte do domínio global da aplicação. Por utilizar o paradigma Decomposição de Domínio, ao final de cada iteração, esses processos se comunicam visando compatibilizar a influência da resposta entre os diversos subdomínios.

No próximo capítulo, os conceitos principais referentes a tolerância a falhas são apresentados e aqueles utilizados neste trabalho são justificados e destacados. Adicionalmente, é feita uma revisão bibliográfica na literatura existente, onde são descritos alguns trabalhos relacionados, destacando-se as suas diferenças em relação à solução proposta neste trabalho. Encerrando o capítulo, é feita a apresentação dos algoritmos que compõem o mecanismo de tolerância a falhas e que foram incorporados à aplicação alvo.

Capítulo 3

Tolerância a Falhas

Cada vez mais *clusters* de computadores necessitarão do uso e controle simultâneo de centenas de milhares de elementos de processamento, armazenamento e redes e, por consequência, a falha nesse ambiente será cada vez mais frequente. Dessa forma, aplicações que executam nesse tipo de ambiente devem ser capazes de manter sua execução mesmo na presença de falhas. Na Seção 3.1 são apresentados os conceitos principais referentes a tolerância a falhas . Na Seção 3.2 são descritos alguns trabalhos relacionados, destacando-se as suas diferenças em relação à solução proposta neste trabalho. Por fim, na Seção 3.3 são apresentados os algoritmos que compõem o mecanismo de tolerância a falhas proposto neste trabalho.

3.1 Conceitos

Os sistemas computacionais são desenvolvidos para atender a um conjunto de requisitos que satisfaçam necessidades do usuário do sistema. Em algumas aplicações, a dependabilidade [4] é um requisito principal, que se refere a capacidade de entregar um serviço confiável e esse conceito é composto pelos seguintes atributos:

- Disponibilidade: o sistema está pronto para ser usado imediatamente;
- Confiabilidade: o sistema executa o serviço de forma correta;
- Confidencialidade: ausência de acesso a informações não autorizadas;
- Integridade: ausência de alterações impróprias no estado do sistema;
- Segurança: se refere à probabilidade do sistema ou estar operacional e executar sua

função corretamente ou descontinuar suas funções de forma a não provocar dano [66];

- Mantenabilidade: se refere à facilidade de realizar manutenção no sistema;

Os atributos acima podem ser destacados de acordo com o grau de dependabilidade exigido pela aplicação. Esses atributos devem ser interpretados de maneira probabilística e relativa, pois, na presença de falhas, sistemas nunca são totalmente disponíveis, confiáveis e seguros [4]. Para alcançar dependabilidade, os sistemas utilizam a combinação de quatro técnicas:

- Prevenção de Falhas: O objetivo desta técnica é prevenir a ocorrência ou a introdução de falhas. Este objetivo é atingido pelas técnicas de controle de qualidade empregadas durante o projeto e a fabricação do hardware e software.
- Tolerância a Falhas: Esta técnica visa fornecer um serviço de forma correta mesmo na presença de falhas. Segundo (ZIV; BRUCK, 1996), tolerância a falhas pode ser dividido em quatro estágios: Detecção da falha, Localização da falha, Contenção da falha, Recuperação da falha. A Detecção de falhas é o processo de reconhecimento da ocorrência de uma falha no sistema. Já a Localização da falha consiste na identificação da parte do sistema que causou a falha. Em seguida, a Contenção da falha visa prevenir uma possível propagação de erro para o resto do sistema, isolando o componente faltoso. Por fim, a Recuperação de falhas é o processo de restaurar o estado operacional do sistema para um estado consistente. Os processos de Detecção e Recuperação são detalhados nas SubSeções 3.1.1 e 3.1.2.
- Remoção de Falhas: Durante a fase de desenvolvimento e a fase de vida operacional do sistema, esta técnica é empregada na tentativa de reduzir o número ou severidade das falhas. Durante a fase de desenvolvimento do sistema, a remoção consiste em três passos: 1) verificação se o sistema atende a certas propriedades, nomeadas as condições de verificação; 2) em caso de falha, diagnóstico da falha que causou o não atendimento das condições de verificação; e 3) realizar as correções necessárias. Durante a vida operacional de um sistema, a remoção é realizada por manutenção preventiva ou corretiva. A manutenção preventiva visa descobrir e remover possíveis falhas antes que elas causem erros enquanto que a manutenção corretiva busca remover falhas que produziram um ou mais erros detectados [61].

- Previsão de Falhas: Uma avaliação do comportamento do sistema em relação à ocorrência ou a ativação das falhas é conduzida a fim de estimar as futuras incidências e as possíveis conseqüências das falhas. Essa avaliação possui dois aspectos: 1) avaliação qualitativa ou ordinal, que visa identificar, classificar os tipos de falhas ou a combinação de eventos que as causam no sistema; 2) avaliação quantitativa ou probabilística, que busca avaliar, em termos de probabilidade, o nível em que alguns dos atributos de dependabilidade são satisfeitos. A avaliação da cobertura fornecida por mecanismos de tratamento de falhas pode ser realizada, por exemplo, através da injeção de falhas para verificar o comportamento do sistema [61].

Neste trabalho a aplicação RTM é considerada correta e os componentes físicos empregados em sua execução podem falhar. As técnicas de prevenção e remoção de falhas não são utilizadas, uma vez que questões relativas a modelagem da aplicação, suas devidas correções e a manutenção/fabricação do *hardware* estão fora do escopo deste trabalho. Na previsão de falhas, alguns processos desligam a interface de comunicação de seus nós, produzindo assim falhas na aplicação, de modo que seu comportamento perante esse cenário possa ser verificado. Além disso, este trabalho emprega a tolerância a falhas como principal meio para alcançar dependabilidade.

Um dos princípios fundamentais na construção de um mecanismo de tolerância a falhas é a especificação de um modelo de falhas, uma vez que ele define quais falhas devem ser tratadas por esse mecanismo [43]. Sem um modelo de falhas, não existe modo de avaliar se um dado sistema é ou não tolerante a falhas. Em (KOOPMAN, 2003) foram definidas as características que devem estar presentes nos modelos de falhas: manifestação da falha, duração, fonte da falha, perfil de ocorrência e granularidade da falha.

Considerando a manifestação da falha, elas podem ser classificadas por modelos abstratos que descrevem como o sistema se comportará na presença delas. No trabalho (SCHNEIDER, 1993) foi proposto o seguinte modelo de falhas:

- *Fail-stop* : Um processador falha por travamento (*halting*), permanece no mesmo estado e esta falha pode ser detectada por outros processadores;
- Colapso (*Crash*): Um processador falha por travamento (*halting*), permanece no mesmo estado e este processador faltoso pode não ser detectado pelos outros processadores;
- Colapso + *Link* de Comunicação : Um processador falha por travamento e permanece no mesmo estado. Um *link* de comunicação falha, perde mensagens, entretanto,

nenhuma mensagem é atrasada, duplicada ou corrompida;

- Omissão de Recebimento: Um processador falha ou por travamento e permanece travado ou por receber somente um subconjunto de mensagens que foram enviadas para ele;
- Omissão de Envio: Um processador falha ou por travamento e permanece travado ou por transmitir somente um subconjunto de mensagens que ele deveria enviar;
- Omissão Geral: Omissão de Recebimento + Omissão de Envio;
- Falhas Bizantinas: Um processador falha por exibir um comportamento arbitrário.

As falhas *fail-stop* são menos perturbadoras, porque os processadores nunca realizam ações erradas e as falhas são detectáveis. Entretanto, falhas por colapso em sistemas assíncronos são mais difíceis de tratar, pois é impossível distinguir entre um processador que está executando muito lento e um outro que está travado devido a uma falha por colapso [58]. Os modelos de falhas por Colapso + *Link* de Comunicação, Omissão de Recebimento, Omissão de Envio e Omissão Geral tratam falhas envolvendo perda de mensagens. As falhas bizantinas são mais perturbadoras e um sistema que tolera este tipo de falha pode tolerar falhas de qualquer tipo [61].

Além disso, as falhas também podem ser classificadas, de acordo com a sua duração, em transitória, intermitente ou permanente [53]. A falha transitória existe por um tempo limitado e não é recorrente. Normalmente esta falha é provocada por uma interferência externa. A falha intermitente provoca uma oscilação entre operações com e livre de erros. Esta falha, na maioria dos casos, é resultado de operações em dispositivos instáveis ou sobrecarregados. Finalmente, a falha permamente ocorre devido a uma condição do dispositivo que não é corrigida com o tempo. Esta falha é provocada por defeitos de componentes, danos físicos ou erros de projeto [61].

Uma característica importante é a fonte das falhas considerada no modelo, pois essa suposição está diretamente ligada a estratégia que será usada para tratar essas falhas. Falhas podem ocorrer devido a erros de requisitos, erros de implementação, erros operacionais, entre outros. Alguns sistemas são modelados para somente tratar falhas de hardware e não falhas de software [43, 61].

Além da fonte das falhas, é necessário definir o perfil das ocorrências de falhas. As falhas consideradas podem ser somente falhas esperadas, tais como falhas observadas historicamente ou exceções definidas; falhas com base na análise da modelagem ou falhas

inesperadas. Adicionalmente, as falhas podem ser aleatórias e independentes, podem ser correlacionadas em tempo ou espaço, ou podem ser intencionais devido a ações maliciosas [43, 61].

Uma questão chave na modelagem de sistemas tolerantes a falhas é a definição da granularidade de uma falha, ou seja, o tamanho do componente comprometido por esta [43]. Uma falha pode atingir um módulo de software, uma tarefa, um conjunto de processadores ou um parque computacional inteiro. Diferentes mecanismos de tolerância a falhas podem ser utilizados dependendo da granularidade da falha que é considerada.

Neste trabalho são tratadas falhas de colapso, cuja duração é permanente, o perfil das ocorrências é aleatório e independente e os componentes comprometidos podem ser nós do *cluster* ou *links* de comunicação.

3.1.1 Detecção de Falhas

Informações sobre a situação operacional dos processos são frequentemente necessárias para implementar aplicações confiáveis e distribuídas. Nesta tarefa, empregam-se detectores de falhas, visto que eles são considerados oráculos que produzem tais informações [31]. De acordo com (CHANDRA; TOUEG, 1996), um detector de falhas é composto por vários módulos detectores, um para cada processo. Cada módulo local indica se um processo está funcionando ou não, e, em caso de não funcionamento, ele começa a suspeitar que esse processo falhou.

Na construção de detectores de falhas, podem ser empregadas diferentes suposições em relação ao comportamento dos sistemas distribuídos onde eles serão executados. Esses sistemas podem ser caracterizados por limites impostos a dois parâmetros críticos: o tempo levado para entregar uma mensagem em um canal de comunicação e o tempo levado por um processador para executar um pedaço de código [31]. Dependendo da existência ou não de limites para esses atributos e do conhecimento desses limites, os modelos de sistema podem ser classificados em síncronos, assíncronos ou parcialmente síncronos [27].

Basicamente, um sistema é considerado síncrono se há um limite superior conhecido relacionado aos atributos citados anteriormente. Por outro lado, um sistema é dito assíncrono se não há esse limite. Por fim, em um sistema parcialmente síncrono, o sistema é assíncrono inicialmente e somente depois de um tempo desconhecido t, o sistema se torna síncrono. Essa suposição captura o fato de que o sistema não se comporta sempre como

síncrono. Em geral, sistemas distribuídos são síncronos a maior parte do tempo, e então eles passam por períodos limitados de assincronia [44]. É esperado da sincronia parcial um período de sincronia longo o suficiente para terminar um algoritmo distribuído. O trabalho (CHANDRA; TOUEG, 1996) propôs um tipo de sincronia parcial, onde um limite superior desconhecido em relação ao tempo de entrega de mensagem e processamento de um código passa a existir após um tempo t também desconhecido. Neste trabalho, utiliza-se esse modelo de sistema parcialmente síncrono na elaboração de algoritmos de detectores de falhas.

Existem diferentes classes de detectores de falhas. Em (CHANDRA; TOUEG, 1996), elas foram caracterizadas em termos de duas propriedades abstratas: completude e exatidão. Completude é a capacidade do detector de suspeitar de todo processo incorreto, enquanto exatidão refere-se a capacidade do detector não suspeitar de processos corretos. Abaixo são definidos os conceitos dos diferentes tipos de completude e exatidão [34] que dão origem as oito classes de detectores apresentadas na Tabela 3.1:

- Completude forte: todos os processos corretos terminarão por suspeitar de todo processo faltoso;
- Completude fraca: algum processo correto terminará por suspeitar de todo processo faltoso;
- Exatidão forte: nenhum processo é considerado suspeito antes de falhar, ou seja, não há suspeita equivocada;
- Exatidão fraca: pelo menos um processo correto nunca é considerado suspeito;
- Exatidão forte após um tempo: existe um momento a partir do qual todos os processos corretos não são considerados suspeitos por qualquer processo correto;
- Exatidão fraca após um tempo: existe um momento a partir do qual algum processo correto não é considerado suspeito por qualquer processo correto;

Os detectores que não têm a propriedade de exatidão forte (W, $\diamond W$, S, $\diamond S$, $\diamond P$, $\diamond Q$) são considerados não confiáveis enquanto que aqueles que satisfazem a propriedade de exatidão forte (P, Q) são considerados confiáveis [34].

O detector de falhas não confiável pode cometer erros. Suponha que um módulo detector adicione um processo correto na sua lista de suspeitos. Caso o módulo detector descubra que foi um erro suspeitar de um determinado processo, então ele remove o

	Exatidão Forte	Exatidão Fraca	Exatidão Forte após um tempo	Exatidão Fraca após um tempo
Completude Forte	*Perfeito* P	*Forte (Strong)* S	*Perfeito após um tempo* $\diamond P$	*Forte após um tempo* $\diamond S$
Completude Fraca	*Quase Perfeito* Q	*Fraco (Weak)* W	*Quase Perfeito após um tempo* $\diamond Q$	*Fraco após um tempo* $\diamond W$

Tabela 3.1: *Definição das propriedades de completude e exatidão.*

processo em questão da sua lista de suspeitos. Assim, cada módulo pode repetidamente adicionar e remover processos da sua lista de suspeitos. Em qualquer momento, usando detectores de falhas confiáveis ou não confiáveis, dois módulos detectores de dois diferentes processos podem ter diferentes listas de suspeitos. Na prática, utilizar detectores de falhas não confiáveis é muito mais realista que usar detectores de falhas confiáveis, pois suas propriedades são mais facilmente garantidas. Isto ocorre porque as características inerentes aos sistemas parcialmente síncronos permitem a implementação de tais detectores [31]. Vale ressaltar que nenhuma das classes de detectores é implementável no modelo assíncrono sem que algumas hipóteses de sincronia sejam acrescentadas, uma vez que é impossível distinguir se um processador está executando muito lento ou se está travado devido a uma falha por colapso [58].

Para a implementação de detectores de falhas, é necessário que alguns processos monitorem outros processos. O monitoramento permite que um processo detecte se um outro processo falhou e tome as medidas necessárias nesse caso. Existem diversas estratégias que podem ser usadas nesse monitoramento. Tipicamente, são utilizadas as estratégias *Push* e *Pull* [59, 55].

Na abordagem denominada *Push*, cada módulo detector de falhas periodicamente envia uma mensagem "*Eu estou vivo*", isto é, mensagem de *heartbeat*, para os outros módulos, com o objetivo de informá-los de que continua vivo. Um processo **a** suspeita de falha de um processo **b**, quando o seu módulo detector de falhas local deixa de receber as mensagens "Eu estou vivo" vindas do módulo de **b**, por um período de tempo maior do que um tempo de espera (*timeout*) determinado. Na abordagem *Pull*, cada módulo detector de falhas periodicamente envia uma mensagem "*Você está vivo?*" para os outros módulos. Após seu recebimento, cada módulo responde com uma mensagem "Eu estou

vivo". Da mesma forma como ocorre na técnica *Push*, um processo **a** suspeita de falha de um processo **b**, quando o seu módulo detector de falhas local deixa de receber as mensagens "Eu estou vivo" vindas do módulo de **b**. Note que se o tempo de espera for pequeno, as falhas por colapso serão detectadas rapidamente, mas a probabilidade de detectar suspeitos errados é alta. Em contrapartida, se o tempo de espera for longo, o número de suspeitos errados é baixo, porém o tempo para detectar as falhas aumenta.

Diversos trabalhos propuseram outros detectores de falhas com diferentes motivações. Por exemplo, em (AGUILERA et al., 2000) é apresentado um detector de falhas que é útil para algoritmos que param de enviar mensagens após um tempo. Já os trabalhos [9, 37] propõem um detector de falhas adaptável, onde a frequência de envio de mensagens de hearbeat e o tempo limite de espera por uma mensagem (*timeout*) podem ser alterados durante a execução de acordo com as condições da rede, visando a otimização do tempo de detecção [55, 37].

Em (LARREA et al., 1999) são apresentados os algoritmos e provas de corretude das classes de detectores que possuem exatidão forte ou fraca após um tempo presentes na Tabela 3.1. Esses algoritmos pressupõem que os módulos detectores estão arranjados em um topologia virtual de anel, onde cada detector monitora seu vizinho. Além disso, esses algoritmos utilizam a abordagem *Pull* no monitoramento dos processos. Apesar de enviar duas vezes mais mensagens comparado a técnica *Push* , a técnica *Pull* permite um controle mais fino do monitoramento [46].

No trabalho aqui desenvolvido, considera-se que falhas ocorrem uma de cada vez ao longo do tempo. Na ocorrência de uma falha, esta será detectada por apenas um módulo detector e a tarefa de propagar essa informação é feita por um módulo a parte que será detalhado na Seção 3.3. Também, há a necessidade de permitir que qualquer processo que porventura tenha sido considerado suspeito por motivos de atraso da entrega de mensagens, possa ter o valor do seu *timeout* incrementado para que sejam evitadas novas suspeitas errôneas. Dessa forma, este trabalho emprega o algoritmo da classe $\diamond Q$ proposto em (LARREA et al., 1999) para detectar falhas e maiores detalhes serão apresentados na Seção 3.3.

3.1.2 Recuperação de Falhas

Após a ocorrência de falhas, o processo de recuperação tem por objetivo restaurar um sistema para um estado operacional correto. Nesse sentido, emprega-se a recuperação por retorno (*rollback recovery*) que trata um sistema distribuído como uma coleção de

processos de uma aplicação que se comunicam através de uma rede [28]. Os processos gravam periodicamente informações importantes para sua recuperação durante a execução livre de falhas em um dispositivo de armazenamento estável que tolera todo tipo de falhas. Após a ocorrência de uma falha, as informações armazenadas são utilizadas para reiniciar a computação a partir de um estado intermediário, reduzindo assim a quantidade de computação perdida. As informações para recuperação dos processos incluem, no mínimo, os estados desses processos, denominados *checkpoints*. Outros protocolos de recuperação podem requerer informações adicionais, tais como o registro de interações com dispositivos de entrada e saída, eventos que ocorrem com cada processo e mensagens trocadas entre os processos.

A gravação de informações para posterior recuperação é realizada a fim de se obter um estado global consistente do sistema, ou seja, um estado confiável para onde o sistema possa retornar em caso de falhas. Um estado global de um sistema distribuído é uma coleção de estados individuais de todos os processos participantes na execução de uma aplicação e dos estados dos canais de comunicação de cada processo. Um estado global consistente é aquele onde não há mensagens (ou uma cadeia casual de mensagens) enviadas por um processo depois da gravação de seu *checkpoint* local que será recebido por outro processo antes dele gravar seu *checkpoint* local [21].

Os protocolos baseados em *Checkpoint* são um tipo de protocolo de Recuperação por Retorno. Nestes protocolos, somente a gravação de *checkpoint* é utilizada para tolerar falhas. Segundo (ELNOZAHY et al., 2002), esses protocolos podem ser classificados em três categorias:

- *Checkpoint* não-coordenado ou independente;
- *Checkpoint* coordenado;
- *Checkpoint* com comunicação induzida;

No protocolo de *Checkpoint* Não-Coordenado cada processo da aplicação grava de forma independente seu estado. A principal vantagem dessa autonomia é que cada processo grava um *checkpoint* quando lhe for mais conveniente, além de ser de fácil implementação. Todavia, em uma aplicação que se baseia na troca de mensagens, a recuperação por retorno é dificultada, pois as mensagens induzem dependências entre os processos durante a execução. Quando um processo ou mais falham, estas dependências podem provocar o retorno de alguns processos que não falharam. Em alguns casos, este retorno poderá se

estender até o estado inicial da execução, onde todo o trabalho realizado antes da falha é perdido, sendo chamado de efeito dominó. Além disso, *checkpoints* inúteis podem ser feitos, aumentando ainda mais a sobrecarga sobre a aplicação.

No protocolo de *Checkpoint* Coordenado, os processos coordenam o momento da gravação do *checkpoint* a fim de armazenar um estado global consistente [21]. Em sua execução, podem ser utilizadas tanto uma abordagem bloqueante quanto não bloqueante. Na bloqueante, depois que um processo grava seu *checkpoint* local, ele permanece bloqueado até que toda atividade de gravação global seja completada. Por outro lado, na abordagem não bloqueante os processos não precisam parar suas execuções enquanto o procedimento de gravação global de *checkpoint* não terminou. Considerando canais de comunicação confiáveis FIFO (*First In First Out*), a consistência do estado global é alcançada, precedendo a primeira mensagem pós-*checkpoint* da aplicação em cada canal por uma requisição de gravação de *checkpoint*, forçando o processo destino a gravar seu *checkpoint* local antes de receber tal mensagem de aplicação. O emprego deste protocolo torna o processo de recuperação mais simples e não suscetível ao efeito dominó, visto que cada processo sempre reinicia do *checkpoint* mais recente. Entretanto, sua principal desvantagem é a sobrecarga envolvida na coordenação dos processos toda vez que se realiza a gravação de checkpoints.

Por fim, no protocolo de *Checkpoint* com Comunicação Induzida, o efeito dominó é evitado sem necessitar da coordenação da gravação de todos os *checkpoints*. Neste protocolo, os processos fazem gravação local e forçada de *checkpoint*. A gravação local pode ser feita de forma independente enquanto que a forçada deve ser feita para garantir o progresso da linha de recuperação, isto é, o mais recente conjunto de checkpoints locais consistente. Além disso, nenhuma mensagem de coordenação é trocada para determinar quando a gravação forçada de *checkpoint* deve ser feita. Ao invés disso, informações do protoloco são adicionadas nas mensagens da aplicação e através delas, o receptor decide se deve ou não iniciar a gravação de um *checkpoint*. Apesar de realizar gravações forçadas de *checkpoints* a mais do que talvez seja o necessário, o protocolo de *Checkpoint* com Comunicação Induzida, em teoria, possui boa escalabilidade em sistemas com grande número de processos participantes, já que não necessita que esses processos participem de uma gravação de checkpoint coordenada [28].

Um outro tipo de protocolo de Recuperação por Retorno são os protocolos baseados em *Log* de Mensagens ou Registro de Mensagens [28]. Esses protocolos assumem que a execução de um processo é representada por uma sequência de intervalos, onde cada um é iniciado por um evento não determinístico (recebimento de mensagens). A execução du-

rante cada intervalo é considerada determinística, de tal forma que se um processo inicia do mesmo estado e está sujeito aos mesmos eventos não determinísticos nos mesmos lugares dentro da execução, ele sempre produzirá a mesma saída. Além disso, eles consideram que todos os eventos não determinísticos podem ser identificados e seus determinantes (mensagens) correspondentes podem ser armazenados em um repositório estável.

Durante um intervalo de tempo livre de falhas, cada processo armazena no repositório estável os determinantes de todos os eventos não-determinísticos que ele observa. Adicionalmente, cada processo grava seu *checkpoint* para reduzir a extensão da perda de computação durante a recuperação. Uma vez que a execução dentro de um intervalo determinístico depende somente da sequência de eventos não determinísticos que precederam o ínicio do intervalo, depois que uma falha ocorre, a execução pré-falha dos processos pode ser reconstruída à partir do conjunto de *checkpoints* consistentes mais recente até o primeiro evento não-determinístico, cujo determinante não foi registrado. Portanto, esse tipo de protocolo é interessante para aplicações que frequentemente interagem com o mundo externo, por exemplo, dispositivos de entrada e saída que não podem ter suas ações revertidas.

Segundo (ELNOZAHY et al., 2002), os protocolos baseados em *Log* de Mensagens podem ser classificados em:

- *Log* Pessimista (*Pessimistic logging*): todas as informações dos eventos não determinísticos são gravadas no meio de armazenamento estável antes do evento afetar a computação. Este protocolo supõe que uma falha pode ocorrer depois de qualquer evento não determinístico, o que é uma visão pessimista, já que falhas são relativamente raras. Além disso, esse protocolo garante que nenhum processo órfão é criado devido a ocorrência de falhas, simplificando assim o processo de recuperação. O processo órfão é caracterizado pela dependência da execução de um evento não determinístico cujo determinante foi perdido e por isso não pode ser recuperado de um repositório estável ou de uma memória volátil dos processos sobreviventes.

- *Log* Otimista (*Optimistic logging*): os processos mantêm as informações em memória volátil e periodicamente elas são gravadas no meio de armazenamento estável. Este protocolo tem a visão otimista que a gravação no meio estável será realizada antes de uma falha ocorrer. Apesar de reduzir a sobrecarga em uma execução livre de falhas, esse protocolo permite que processos órfãos sejam criados temporariamente devido a falhas, o que implica em um processo de recuperação mais complicado se comparado ao processo realizado no protocolo de *Log* Pessimista.

- *Log* Causal (*Causal logging*): as informações de cada evento não determinístico que precede de forma causal o estado de um processo é gravado no meio estável ou está disponível localmente para aquele processo. A precedência causal é baseada na relação "aconteceu antes", introduzida em (LAMPORT, 1978), onde para quaisquer dois eventos e_i e e_j, se $e_i \rightarrow e_j$, então e_j é diretamente ou transitivamente dependente de e_i. Na Figura 3.1, considere o recebimento de mensagens como eventos não determinísticos, por exemplo, e_0 é o evento de recebimento da mensagem msg_0. Podemos dizer que $e_0 \rightarrow e_1$, pois e_1 depende diretamente do evento e_0 ou e_0 "aconteceu antes" de e_1. Da mesma forma, $e_0 \rightarrow e_4$, pois e_4 depende transitivamente de e_0. Note que, de e_0 a e_4 é possível se estabelecer um caminho causal, tal como $e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4$. Suponha que uma falha ocorreu nos processos P_1 e P_2 nos instantes indicados na Figura 3.1. Como e_4 foi o último evento não determinístico que afetou o estado de P_0 até o ponto **A**, seu determinante e todos os determinantes dos outros eventos que precedem casualmente e_4, no caso, as mensagens msg_0, msg_1, msg_2, msg_3 e msg_4, foram armazenados, antes da ocorrência das falhas, em um repositório estável ou na memória volátil do nó onde P_0 se encontra. Observe que as mensagens msg_5 e msg_6 não afetam casualmente o estado do processo P_0 e por isso elas foram perdidas após a ocorrência de falhas nos processos P_1 e P_2. Com as mensagens armazenadas, P_0 é capaz de auxiliar na recuperação dos processos P_1 e P_2, já que ele sabe a ordem na qual as mensagens msg_1 e msg_3 devem ser recebidas por P_1 para ser consistente com o estado dele. Da mesma maneira, P_0 sabe quando msg_2 deve ser recebida de tal forma que o estado de P_2 seja consistente com os estados de P_0 e P_1. Os *logs* mantidos por cada processo atuam como um seguro de proteção para falhas que ocorrem em outros processos. Assim, como o Log Pessimista, esse protocolo não cria processos órfãos.

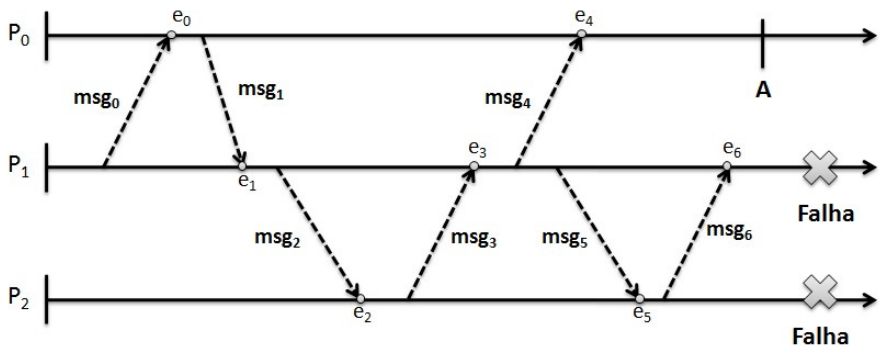


Figura 3.1: Exemplo de funcionamento do procotolo *Log* Causal [28]

Outra técnica de tolerância a falhas que pode ser utilizada é o mascaramento de falhas,

que consiste no uso de redundância suficiente para permitir a recuperação sem detecção de falha explícito [66]. As duas técnicas de replicação mais conhecidas são Replicação Ativa e Passiva [30, 5].

A Replicação Ativa consiste em um grupo de réplicas de um processo que executam os mesmos procedimentos, logo, não é necessário que seja feito algum sincronismo entre as réplicas. A falha de uma dessas réplicas é transparente para aplicação, pois haverão outras para substituir. Apesar do alto custo para execução das réplicas ativas, a recuperação é mais rápida que na replicação passiva. Por sua vez, a replicação passiva também cria uma ou mais réplicas de um processo primário com o objetivo de substituí-lo na presença de falha. Ao contrário da Replicação Ativa, as réplicas ficam em estado ocioso. Em caso de falha, uma réplica é acionada para tomar lugar do processo falho, o estado gravado no último *checkpoint* é retornado, e na existência de registros de log de mensagens, os determinante dos eventos registrados são reexecutados, alcançando o mesmo estado do processo primário no momento da falha [30, 67].

Neste trabalho, utiliza-se o protocolo de *Checkpoint* Não-Coordenado, visto que, como dito na Seção 2.2, ao fim de cada passo da execução da aplicação aqui utilizada, há uma sincronização entre os processos desta aplicação, devido ao uso do paradigma de Decomposição de Domínio. Dessa forma, garante-se que a gravação de *checkpoint* após esta etapa de sincronização produz um estado global consistente, uma vez que os processos estarão no mesmo passo e nenhuma mensagem estará trafegando nos canais de comunicação.

3.2 Trabalhos Relacionados

A Interface de Troca de Mensagens (MPI) é uma especificação do que uma biblioteca de troca de mensagens deve possuir e esta foi construída baseada no consenso do Fórum MPI [50], que é composto por mais de 40 organizações, incluindo fornecedores, pesquisadores, desenvolvedores de biblioteca de software, e usuários. O principal objetivo desta especificação é estabelecer um padrão portável, eficiente e flexível para ser amplamente utilizado para escrever programas que utilizem troca de mensagens em ambientes de alto desempenho [49]. Atualmente, essa especificação se encontra na versão 2.2 e está caminhando para a versão 3.0. Além disso, podem ser encontradas diversas implementações do MPI, de versões pagas a código aberto.

Uma aplicação MPI é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre eles. Os proces-

sos são identificados por um número inteiro único chamado *rank* que é contíguo e começa com valor zero. Esses identificadores, por exemplo, podem ser empregados para especificar a origem e o destino de mensagens. Para se comunicarem, os processos devem estar associados a um mesmo comunicador. Um comunicador é uma estrutura do MPI que especifica o domínio de comunicação de um conjunto de processos. Através dele, podem ser efetuadas operações ponto a ponto, por exemplo MPI_Send e MPI_Receive, onde mensagens são enviadas por um processo e recebidas por outros, respectivamente. Adicionalmente, um grupo de processos pode invocar operações coletivas de comunicação para executar operações de sincronização (MPI_Barrier), propagação de Informação (MPI_Bcast), entre outras [51]. É importante ressaltar que o padrão MPI fornece ao usuário uma transmissão confiável de mensagens e para tal emprega o protocolo TCP [12].

Aplicações MPI não são resilientes a falhas, uma vez que o padrão MPI fornece ao desenvolvedor somente duas abordagens: a execução da aplicação é abortada ou o controle é retornado para a aplicação do usuário sem garantia que novas comunicações possam ocorrer. Entretanto, há algumas implementações deste padrão que fornecem serviços de tolerância a falhas, algumas são detalhadas abaixo.

Starfish [1] fornece um ambiente de execução para programas MPI que se adapta as mudanças em um *cluster* de processadores causadas por falhas nos nós. Em cada nó é executado um *daemon* do Starfish e os processos de aplicação se registram com o intuito de serem avisados sobre falhas de processos. Ele provê duas formas de tolerância a falhas. A primeira consiste na escolha do usuário pelo uso de protocolo coordenado ou não coordenado na gravação de *checkpoints* que, em caso de falhas, são utilizados para reiniciar a aplicação. A segunda é mais dependente da aplicação, visto que na ocorrência de falhas os processos sobreviventes são avisados, e, no momento que ficam cientes da falha, eles repartem o conjunto de dados e continuam sua execução. Em ambas abordagens, o processo de detecção de falhas fica a cargo do sistema de comunicação de grupo *Ensemble* [10].

No MPI-FT [47], as falhas são detectadas por um processo central chamado observador (*observer*) através de um *script* que periodicamente checa a existência de todos os processos e duas abordagens para a recuperação são propostas. Na primeira, cada processo é responsável por manter uma cópia de todas as suas mensagens enviadas, enquanto que na segunda, todas as comunicações são gravadas pelo processo observador. Na presença de falhas, o processo observador é o responsável por reenviar as mensagens do

processo morto para o processo substituto. Os processos substitutos podem ser criados dinamicamente ou na inicialização do programa ficando ociosos esperando uma ativação do processo observador. É necessário que o desenvolvedor inclua pontos de verificação de estado no código da aplicação. Esse pontos de verificação permitem que o processo observador avise aos processos da aplicação sobre as falhas ocorridas e que estas executem as rotinas de recuperação. Uma suposição implícita é que o recurso onde o processo observador é instanciado nunca pode falhar.

O MPI/FT [5] fornece serviços para a detecção e a recuperação de processos faltosos. Entretanto, a aplicação é responsável por realizar a gravação de *checkpoint* e restaurar a execução à partir do último *checkpoint* válido. O MPI/FT trata falhas para aplicações que seguem os modelos Mestre-Trabalhador e SPMD (*Single Program Multiple Data*) e usa mensagens de heartbeat para detectar falhas de processos. Um único processo coordena as funções de detecção e recuperação para a aplicação inteira nos modelos Mestre-Escravo e SPMD. O MPI/FT faz uso da replicação passiva para fornecer tolerância a falhas. Assim, os processos que falharam são substituídos por processos extras que, por sua vez, reiniciam a execução à partir do último *checkpoint* válido. Uma vez que esses processos extras tenham sido utilizados, o MPI/FT não consegue mais recuperar uma falha de processo e, neste caso, a aplicação toda falha. A consistência de checkpoints é estabelecida pela participação de todos os processos em uma operação coletiva de gravação de *checkpoint* que se comporta essencialmente como uma operação de barreira.

O FT-MPI [29] permite a aplicação continuar usando um comunicador com processos falhos empregando uma das seguintes abordagens: exclusão explícita da comunicação dos processos faltosos, encolhimento do comunicador através da eliminação dos processos falhos de seu contexto ou criação dinâmica de processos que substituirão àqueles falhos. FT-MPI não fornece detalhes sobre a estratégia usada para detecção de falhas. Além disso, o desenvolvedor da aplicação é responsável por prover mecanismos de gravação de *checkpoints* e recuperação da aplicação.

O projeto MPICH-V [11] tem sua implementação baseada no MPICH e oferece múltiplos protocolos de recuperação de falhas para aplicações MPI. Atualmente ele oferece 5 protocolos: 2 protocolos de log pessimista com checkpoint não coordenado, 1 protocolo de log causal com checkpoint não coordenado e 2 protocolos de checkpoint coordenado baseado no algoritmo de Chandy-Lamport [21]. Ele é composto por um conjunto de componentes, dentre eles, o despachante que através de mensagens de *heartbeat* enviadas pelos processos da aplicação MPI detecta a ocorrência de falhas potenciais devido à demora de

uma dessas mensagens. Sendo uma falha detectada, o despachante inicia dinamicamente outro processo MPI. Esse processo novo reinicia a execução, alcança o ponto de falha e continua sua execução deste ponto. Os outros processos não ficam cientes da falha ocorrida.

O projeto RADIC-MPI [26] utiliza processos protetores e observadores distribuídos ao longo de cada nó de um *cluster* que cooperam entre si para fornecer o serviço de tolerância a falhas. O processo observador é responsável por gravar o *checkpoint* dos processos presentes em seu nó e transmití-lo a um processo protetor, presente em outro nó, para que este armazene-o em um repositório local. Além disso, os processos observadores são responsáveis por entregar todas as mensagens trocadas entre os processos de aplicação. Dessa forma, eles também salvam essas mensagens e as enviam para os processos protetores. São utilizadas mensagens de heartbeat e, gravação de *checkpoint* através da biblioteca Berkeley Laboratory Checkpoint/Restart (BLCR) [36] e Log Pessimista nos processos de detecção e recuperação de falhas, respectivamente. Os processos de aplicação remanescentes não são avisados sobre a falha. Somente o processo protetor ou observador que detectou a falha inicia o procedimento de recuperação. Novos processos são criados dinamicamente em outro nó, os *checkpoints* são lidos e mensagens recebidas pelos processos falhos são reenviadas para os processos substitutos. Vale ressaltar que RADIC suporta a ocorrência de múltiplas falhas no mesmo instante, desde que essas falhas não sejam correlacionadas, ou seja, um nó de um observador e o nó onde o seu processo protetor se encontra não podem falhar ao mesmo tempo.

LAM/MPI [62] é uma das implementações do MPI mais usada. Ele utiliza a biblioteca de gravação de *checkpoints* a nível de sistema BLCR [36]. Essa biblioteca faz *checkpoint* coordenado usando o algoritmo de Chandy-Lamport [21]. Além disso, ela pode reiniciar a execução de todos os processos da aplicação. A fim de detectar falhas, os "*daemons*" inicializados pelo LAM-MPI em cada nó trocam mensagens de *heartbeat*. Quando um nó é considerado faltoso, todos os outros processos são notificados através de um sinal de interrupção. Assim, o desenvolvedor da aplicação, através das funções *lamshrink* e *lamgrow*, pode remover os processos faltosos do contexto do comunicador e adicionar um novo nó onde processos poderão ser instanciados, respectivamente. Ambas funções são fornecidas pelo LAM-MPI e não fazem parte do padrão MPI.

O Sistema Gerenciador de Aplicações EasyGrid [60] é um sistema gerenciador de aplicações MPI, que utiliza a biblioteca LAM/MPI. Ele controla a execução dos processos da aplicação MPI através de uma hierarquia distribuída de processos gerenciadores. O

mecanismo de tolerância a falhas utiliza as mensagens de gerenciamento como forma de averiguar a existência dos processos. São utilizadas as técnicas *log* de mensagens e *checkpoint* no nível da aplicação para os processos da aplicação e os gerenciadores, respectivamente.

Em produção, a aplicação alvo utiliza a biblioteca Intel MPI [39], pois o seu foco é a maximização do desempenho. Neste trabalho, o objetivo principal é modificar a aplicação alvo para incorporar procedimentos de tolerância a falhas de forma que seja minimizada a sobrecarga imposta a esta aplicação. Nesse sentido, processos monitores distribuídos ao longo dos nós se comunicam entre si e utilizam a técnica *Pull* para detectar falhas de nós ou *links* de comunicação, enquanto que os processos de aplicação periodicamente salvam seus estados e são notificados sobre a ocorrência de falhas por seus monitores locais através de um sinal de interrupção que inicia o procedimento de recuperação, como é descrito na Seção 3.3.

3.3 Mecanismo de Tolerância a Falhas

Esse trabalho é baseado no modelo de sistema distribuído parcialmente síncrono proposto em (CHANDRA;TOUEG, 1996) e o ambiente alvo consiste em um *cluster multicore*, onde cada nó $n_i,\ i = 0, \ldots, n-1$, é composto por um conjunto de núcleos de processamento (*cores*). O modelo de execução considera um único processo de aplicação com identificação única $p_i,\ i = 0, \ldots, m-1$, por *core*.

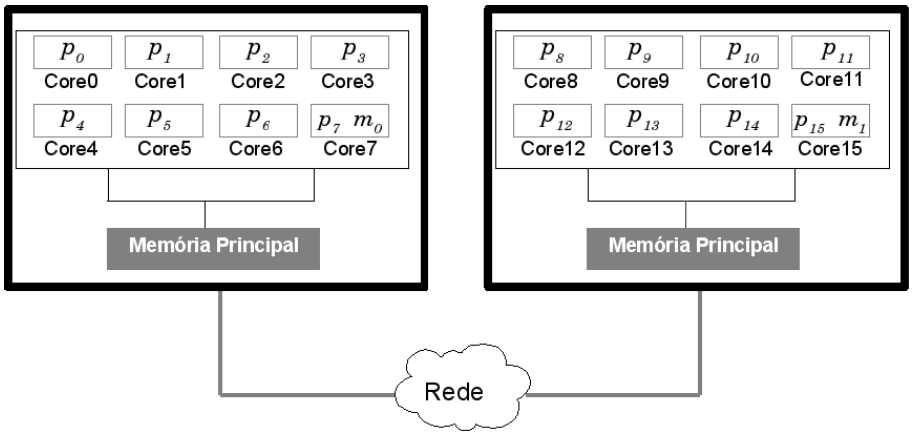


Figura 3.2: Exemplo de um *cluster* de *multicores* composto por dois nós, cada um com 8 núcleos de processamento.

Com o objetivo de detectar falhas, processos de aplicação e um processo monitor, $m_i, i = 0, \ldots, n-1$, coexistem em cada nó. Para exemplificar a organização dos processos

nos nós do *cluster*, na Figura 3.2 ilustra-se um *cluster* com dois nós, cada um com oito *cores*. Observe que os pares de processos (p_7, m_0) e (p_{15},m_1) são executados no *core* 7 e 15, respectivamente.

Assume-se que neste ambiente, múltiplas falhas irrecuperáveis nos nós ou links de comunicação podem ocorrer, mas somente uma falha de cada vez, e que o sistema está totalmente conectado de tal forma que uma falha em um link de comunicação não particione a rede. Além disso, não são consideradas falhas na inicialização e finalização da aplicação, e durante o processo de recuperação e gravação de checkpoints. Após uma falha, a aplicação usará os recursos remanescentes para continuar sua execução.

Algoritmo 2: Algoritmo do Monitor m_i

```
1 início
2 |  iniciarThread(Detecção); iniciarThread(Heartbeat);
3 |  iniciarThread(Inspeção); iniciarThread(Propagação);
  |  /* Processo de terminação                                   */
4 |  ∀p_j, se n(p_j) = n(m_i) então receber(TERMINAR, p_j);
5 |  barreira();
6 |  ∀p_j, se n(p_j) = n(m_i) então enviar(PODE_TERMINAR, p_j);
7 |  terminarThread(Detecção); terminarThread(Heartbeat);
8 |  terminarThread(Inspeção); terminarThread(Propagação);
9 fim
```

O nó onde p_i e m_i estão alocados é denotado por $n(p_i)$ e $n(m_i)$, respectivamente. De acordo com o Algoritmo 2, cada monitor m_i cria quatro *threads*: *Inspeção()*, *Propagação()*, *Detecção()* e *Heartbeat()* através da função *iniciarThread()* e coordena a terminação da aplicação. Essa coordenação é necessária, pois a rotina de terminação *MPI_Finalize*() do MPI não consegue realizar sua tarefa, em caso de falhas, uma vez que ela executa uma operação coletiva com todos os processos instanciados.

O processo de terminação empregado está situado entre as linhas 4 e 8 do Algoritmo 2. Em um primeiro momento, cada monitor m_i aguarda a mensagem *TERMINAR* de todos processos da aplicação p_j, tais que $n(p_j) = n(m_i)$. Essa mensagem indica que os processos de aplicação já acabaram sua execução. Em seguida, ele inicia a operação coletiva *barreira*() na qual são considerados somente os monitores alocados em processadores não falhos. Após a execução dessa operação, o monitor tem certeza que pode terminar, e, então, envia uma mensagem *PODE_TERMINAR* para cada processo de aplicação p_j presente no mesmo nó onde ele se encontra. Por fim, todas as *threads* são terminadas através do procedimento *terminarThread*(). Maiores detalhes sobre a operação *barreira*() serão dados na Subseção 3.3.2.

As seguintes listas são compartilhadas entre as *threads*: LS_i é a lista de monitores que supostamente falharam; processos de aplicação faltosos são listados em LPF_i; e LMF_i é a lista de todos os processos monitores (e consequentemente, seus nós) que falharam durante a execução. Semáforos são usados para coordenar o acesso entre os recursos compartilhados entre as *threads* e para garantir que elas possam fazer chamadas as funções MPI, mas somente uma por vez. Dessa forma, considere que as operações *enviar* e *receber* utilizadas em todos algoritmos apresentados estejam implicitamente protegidas por semáforos.

Baseado na classe $\diamond Q$ de detectores de falhas, as *threads* nos Algoritmos 3 and 4 [46] foram implementadas para identificar nós faltosos. Para atingir esse objetivo, os processos monitores são arranjados de tal forma que $succ(m_i)$ e $pred(m_i)$ representam, respectivamente, o sucessor e predecessor do monitor m_i no anel dos monitores ativos.

Algoritmo 3: *Detecção()*

```
1  início
2      alvo_i ← succ(m_i);
3      LS_i ← ∅; ∀m_j : △_{i,j} ← timeout;
4      loop
5          wait(mutex_i);
6          enviar("Você está vivo?", alvo_i);
7          recebeu ← falso;
8          signal(mutex_i);
9          sleep(△_{i,alvo_i});
10         wait(mutex_i);
11         se não recebeu então
12             △_{i,alvo_i} ← △_{i,alvo_i} + 1;
13             LS_i ← LS_i ⋃ {alvo_i};
14             alvo_i ← succ(alvo_i);
15         fim
16         signal(mutex_i);
17     fim
18 fim
```

Detecção() no Algoritmo 3 envia, a cada $\triangle_{i,alvo_i}$ ou *timeout* unidades de tempo, uma mensagem "Você está vivo?" para seu monitor vizinho, $alvo_i$, para descobrir se o nó onde ele se encontra falhou. Se depois de mais de $\triangle_{i,alvo_i}$ unidades de tempo, m_i não receber uma resposta de $alvo_i$ (via *Heartbeat()*), $alvo_i$ será colocado sob suspeita de ter falhado, ou seja, ele será incluído em LS_i. Dado que a falta de uma resposta pode ser devido a latência no tempo de ida e volta das mensagens maior que o valor da variável $\triangle_{i,alvo_i}$, essa mesma variável é incrementada a fim de reduzir a possibilidade de futuras suspeitas

incorretas.

Algoritmo 4: *Heartbeat()*

```
1  início
2      loop
3          receber(msg, m_j);
4          wait(mutex_i);
5          se msg = "Você está vivo?" então
6              enviar("Eu estou vivo.", m_j);
7              se m_j ∈ LS_i então
8                  LS_i ← LS_i − {m_j, ..., pred(alvo_i)};
9                  alvo_i ← m_j;
10                 recebeu ← verdadeiro;
11             fim
12         senão se msg = "Eu estou vivo." então
13             se m_j = alvo_i então
14                 recebeu ← verdadeiro;
15             senão se m_j ∈ LS_i então
16                 LS_i ← LS_i − {m_j, ..., pred(alvo_i)};
17                 alvo_i ← m_j;
18                 recebeu ← verdadeiro;
19             fim
20         fim
21         signal(mutex_i);
22     fim
23 fim
```

Após o recebimento da mensagem *msg* vinda do monitor m_j, *Heartbeat()* no Algoritmo 4 ou responde ao remetente com uma mensagem de hearbeat "Eu estou vivo" ou confirma o recebimento de tal mensagem. Se o remetente m_j está em LS_i, todos os monitores entre m_j e $pred(alvo_i)$ são removidos, já que m_j está na verdade vivo e os outros monitores serão testados por m_j e/ou seus sucessores. Este mecanismo pode, portanto, corrigir falso positivos gerados por detectores de falhas não confiáveis. Por outro lado, se um monitor que está neste caminho realmente falhou, este será detectado novamente mais tarde.

No Algoritmo 5, *Inspecção()*, a cada *timeout_ inspeção* unidades de tempo, m_i verifica se há um novo monitor faltoso (isto é, um nó) em LS_i e, se houver, atualiza LMF_i e propaga esta informação para sincronizar e atualizar os monitores remanescentes. Se *Propagação()* (Algoritmo 6) receber uma nova lista de monitores falhos de um outro monitor, m_i atualiza suas próprias listas LMF_i e LPF_i. Um sinal de interrupção e LPF_i são enviados para todos os processos de aplicação presentes no nó $n(m_i)$. Por sua vez, uma mensagem de confirmação, enviada por cada p_j em $n(m_i)$ deve ser recebida por m_i

Algoritmo 5: *Inspeção()*

```
1  início
2      loop
3          sleep(timeout_inspeção);
4          wait(mutex2_i); wait(mutex_i);
5          se LS_i ≠ ∅ então
6              se LMF_i = ∅ então
7                  LMF_i ← LS_i; LMF_aux_i ← LS_i;
8              senão
9                  LMF_aux_i ← (LS_i − LMF_i);
10                 LMF_i ← LMF_i ∪ LMF_aux_i;
11             fim
12         fim
13         signal(mutex_i);
14         se LMF_aux_i ≠ ∅ então
15             ∀m_j, se m_j ∉ LMF_i então enviar(LMF_i, m_j);
16         fim
17         signal(mutex2_i);
18     fim
19 fim
```

Algoritmo 6: *Propagação()*

```
1  início
2      loop
3          receber(LMF_k, m_k);
4          wait(mutex2_i);
5          LMF_i ← LMF_k;
6          ∀p_j, se n(p_j) = n(m_i) então enviarSinalInterrupção(p_j);
7          LPF_i ← identificarProcessosAplicaçãoFalhos(LMF_i);
8          ∀p_j, se n(p_j) = n(m_i) então enviar(LPF_i, p_j);
9          ∀p_j, se n(p_j) = n(m_i) então receber(FIM_PROPAGAÇÃO,p_j);
10         signal(mutex2_i);
11     fim
12 fim
```

para confirmar o recebimento da informação da ocorrência da falha.

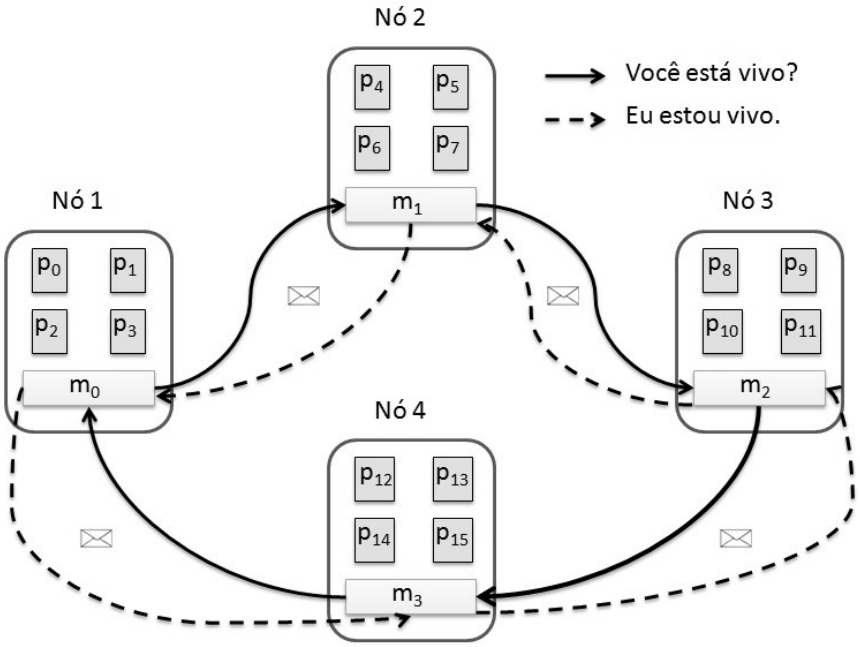
Para ilustrar todo o processo de detecção de falhas, considere o exemplo apresentado na Figura 3.3. Suponha que a aplicação foi iniciada em um *cluster* composto por quatro nós arranjados em uma topologia virtual de anel. Inicialmente, em *Detecção()* (Algoritmo 3), cada monitor define seu sucessor no anel como processo alvo, para em seguida enviá-lo uma mensagem "Você está vivo?". Além disso, ele aguarda *timeout* unidades de tempo para verificar o recebimento da mensagem "Eu estou vivo" pela tarefa *Heartbeat()* (Algoritmo 4). Em *Heartbeat()*, uma vez que cada processo recebeu uma mensagem "Você está vivo?" de um outro monitor, é retornado para o mesmo uma mensagem "Eu estou vivo", conforme demonstrado na Figura 3.3(a).

Suponha que o nó 2 falhou. Nesse caso, o monitor m_0, através de *Detecção()*, verifica que m_1 não enviou a mensagem "Eu estou vivo" dentro de *timeout* unidades de tempo, o adiciona em sua lista de suspeitos LS_0 e passa a monitorar o próximo sucessor no anel, o monitor m_2. A fim de permitir a correção de qualquer suspeita errônea, somente após $timeout\_inspeção$ unidades de tempo, m_0, em *Inspeção()* (Algoritmo 5), verifica se há algum processo em sua lista de suspeitos e em caso positivo, envia uma mensagem contendo essa lista para todos os outros monitores, conforme apresentado na Figura 3.3(b). Por fim, em *Propagação()* (Algoritmo 6), cada monitor envia um sinal de interrupção para cada processo de aplicação local informando sobre os processos que falharam para que se inicie o procedimento de recuperação da aplicação, detalhado na SubSeção 3.3.1.

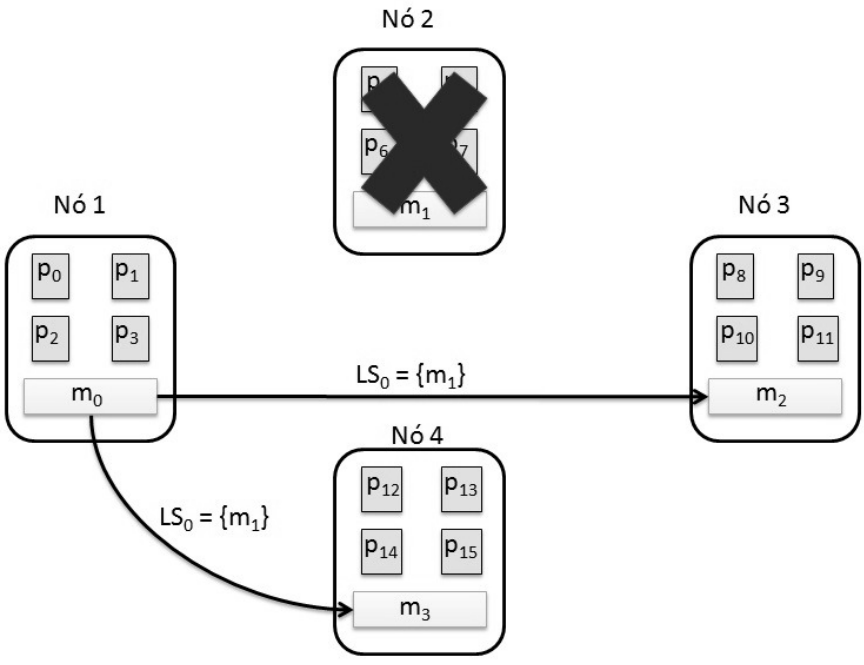
3.3.1 Modificações na Aplicação RTM

Como este trabalho se propõe a utilização de um mecanismo de tolerância a falhas a nível de aplicação, foi necessário realizar algumas modificações na aplicação alvo a fim de incorporar esse mecanismo. No Algoritmo 7 apresenta-se o algoritmo da aplicação RTM com essas modificações. Na inicialização da aplicação, passam a existir processos de aplicação e monitores, logo, é necessário diferenciá-los de tal forma que cada um execute o trecho de código correspondente ao seu papel. Os processos monitores são aqueles cuja identificação é a maior presente em seus nós. Uma vez identificados, eles executam o procedimento *monitor*() representado pelo Algoritmo 2 já apresentado. Outra mudança que pode ser notada é a introdução da função *checkpoint*() responsável por gravar as matrizes utilizadas pelo algoritmo assim como os valores de algumas variáveis em um certo intervalo de passos de execução definido pelo usuário.

Para detalhar o funcionamento do Algoritmo 7 com os mecanismos de tolerância a



(a)



(b)

Figura 3.3: Mecanismo de Detecção de Falhas

Algoritmo 7: Algoritmo da Aplicação RTM com mecanismo de tolerância a falhas

```
1  início
2      se é processo de aplicação então
           /* inicialização                                          */
3          terminou ← falso;
4          ocorreuFalha ← falso; //variável global
5          LPF_i ← ∅;
6          Lê modelo de velocidades;
7          enquanto não terminou faça
8              Inicia a decomposição do domínio sobre o Modelo de Velocidades de
               acordo com o número de processos não faltosos;
9              De acordo com o subdomínio correspondente a p_i e seu tamanho,
               criam-se matrizes que armazenarão os resultados da simulação;
10             se LPF_i ≠ ∅ então
11                 restaurar();
12             fim
               /* laço principal                                     */
13             enquanto tempo < tempo da simulação faça
14                 calcula o pulso sísmico gerado pela fonte sísmica;
15                 calcula o campo de ondas;
16                 troca borda do subdomínio com as partições vizinhas;
17                 checkpoint();
18             fim
19             se não ocorreuFalha então
20                 terminou ← verdadeiro;
21             senão
22                 ocorreuFalha ← falso;
23                 receber(LPF_j,m_j);
24                 enviar(FIM_PROPAGAÇÃO,m_j);
25                 LPF_i ← LPF_j;
26                 limparCanaisComunicação();
27             fim
28         fim
29     senão
30         monitor();
31     fim
       /* finalização                                                */
32     se é processo de aplicação então
33         enviar(TERMINAR,m_j), onde n(p_i) = n(m_j);
34         receber(PODE_TERMINAR,m_j);
35     fim
36     barreira();
37     terminarAplicação();
38 fim
```

falhas inclusos, suponha que uma falha ocorreu. Nesse caso, os processos de aplicação são notificados através de um sinal de interrupção, que consequentemente, implica no redirecionamento do fluxo da execução da aplicação para uma rotina de tratamento de interrupção definida no Algoritmo 8. Então, o valor da variável *ocorreuFalha* é atualizado, a rotina é encerrada e o fluxo de execução retorna para o ponto onde a aplicação foi interrompida. Os processos de aplicação ficam cientes da falha ocorrida através dos pontos de checagem presentes no trecho de código responsável por trocar a borda do subdomínio entre processos que possuem partições vizinhas. Esses pontos de checagem conferem se a variável *ocorreuFalha* foi atualizada para um valor que indique a ocorrência de falhas. Além disso, funções MPI não bloqueantes de envio e recebimento são utilizadas nesta etapa com o intuito de evitar o bloqueio dos processos uma vez que nesta etapa eles efetuam uma sincronização. Com a confirmação de que a falha ocorreu, o laço principal é encerrado para que se possa iniciar o procedimento de recuperação da aplicação.

Algoritmo 8: tratarInterrupção()

```
1 início
2 |  ocorreuFalha ← verdadeiro
3 fim
```

O primeiro passo a ser executado durante o processo de recuperação da aplicação é aguardar o recebimento de um lista de processos faltosos LPF_j do monitor m_j presente em seu nó. Em seguida, cada processo atualiza sua lista local LPF_i e envia uma mensagem FIM_PROPAGAÇÃO para confirmar ao monitor m_j o recebimento de LPF_j. A ciência da falha ocorre na etapa de comunicação entre os processos e isto implica no encerramento imediato do laço principal, que por sua vez provoca o não recebimento de mensagens que já estavam em trânsito nos canais de comunicação. Por esta razão, cada processo realiza a limpeza de seus canais de comunicação por meio do envio de uma mensagem LIMPEZA para seus vizinhos não falhos e posterior recebimento de quaisquer mensagens desses vizinhos. Uma vez que um processo tenha recebido mensagem de LIMPEZA de todos os seus vizinhos não falhos, ele garante que seu canal de comunicação com esses processos está limpo.

Posteriormente, uma nova decomposição do domínio é realizada de acordo com o novo número de processos não falhos. Com as novas dimensões do domínio local e o novo posicionamento no domínio global, os processos de aplicação restauram os dados de suas estruturas de dados através da função *restaurar*(). Para exemplificar esse procedimento de restauração, considere o exemplo ilustrado na Figura 3.4. Na Figura 3.4(a), apresenta-

se um domínio bidimensional com a distribuição inicial de 16 processos. Suponha falho o processo p_{10}. No processo de recuperação da aplicação é realizado o redimensionamento dos novos domínios locais, agora entre os quinze processos ativos. Com os domínios locais redefinidos como mostrado na Figura 3.4(b), os processos passam a carregar do repositório os checkpoints correspondentes à sua nova posição. O processo p_{11}, por exemplo, carregará parcialmente os dados dos *checkpoints* dos processos p_8 e p_9 para, assim, compor sua matriz. Já o processo p_9 terá que carregar, também parcialmente, os dados dos arquivos dos processos p_6, p_7, p_{14} e p_{15} a fim de montar a sua matriz.

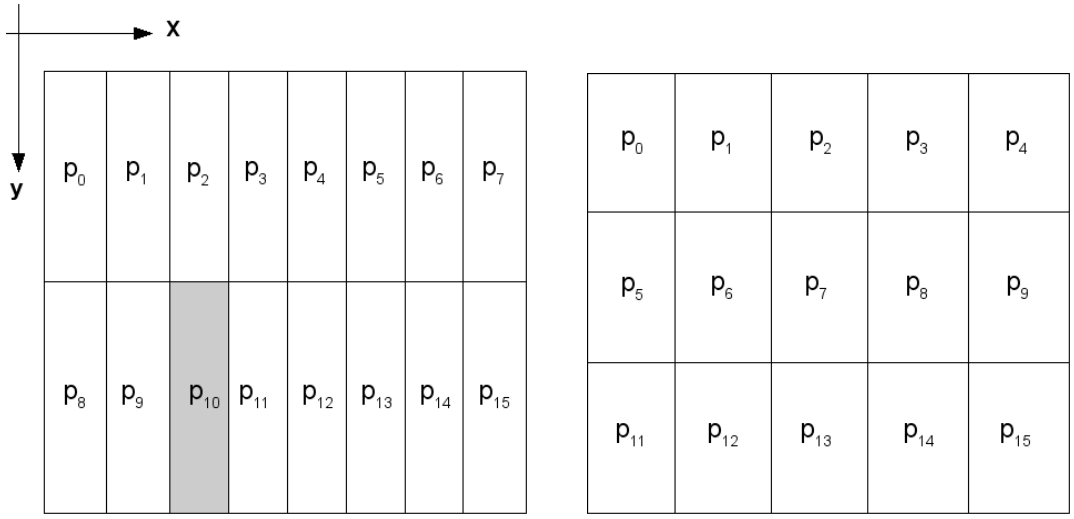


Figura 3.4: Redistribuição do domínio na recuperação de uma falha

Completado o procedimento de restauração, os processos de aplicação voltam a executar o laço principal a partir do passo que estava armazenado nos checkpoints. Supondo que nenhuma outra falha ocorra nessa execução pós-falha, os processos terminarão a execução do laço principal e do laço externo a ele. Para garantir que processos de aplicação e monitores terminem juntos, os processos de aplicação avisam aos monitores que eles já acabaram sua execução e estes retornam com uma mensagem de aprovação. Em seguida, os processos de aplicação executam uma barreira e aguardam pelos monitores. Concomitantemente, os processos monitores terminam suas *threads*, seu fluxo de execução retorna para aplicação e executam o procedimento *barreira*(). Ao término dessa operação, todos os processos finalizam a aplicação.

3.3.2 Questões de Implementação

A aplicação utilizada neste trabalho faz uso de funções coletivas do MPI[49], tais como MPI_Barrier, MPI_Bcast e MPI_Reduce. MPI_Barrier realiza a operação de barreira, MPI_Bcast propaga uma informação para todos os nós e MPI_Reduce reduz os valores de todos processos em um único valor através de operações aritméticas, booleanas ou de máximo e mínimo. Essas funções, em sua execução, consideram a participação de todos os processos inicialmente instanciados. Logo, em um cenário de falhas, a aplicação nunca terminaria, uma vez que seus processos de aplicação ficariam bloqueados. Por essa razão, essas funções foram reimplementadas de tal forma que a participação de processos faltosos fosse desconsiderada. As seguintes técnicas foram utilizadas: o algoritmo de Disseminação [38] para MPI_Barrier; a Árvore Binomial [65] para MPI_Bcast e MPI_Reduce. Além disso, como dito no ínicio da Seção 3.3, os processos monitores assumem o papel do MPI_Finalize.

Outra questão importante se refere ao procedimento de limpeza dos canais de comunicação descrito na Subseção 3.3.1. Esse procedimento não garante a limpeza dos canais entre os processos remanescentes e falhos, visto que os processos sobreviventes nunca receberão mensagem de LIMPEZA dos processos falhos. Na prática, verificou-se que mesmo executando tal procedimento, as novas operações de envio e recebimento não eram concluídas. As funções do MPI possuem um parâmetro chamado etiqueta (*tag*) que pode ser utilizado para diferenciar os tipos das mensagens transmitidas na aplicação. Para resolver o problema descrito acima, a solução experimental adotada, além de limpar os canais de comunicação entre os processos vivos, foi aplicar um esquema dinâmico na determinação dos valores das etiquetas utilizados na aplicação. Dessa forma, antes de cada operação MPI, o valor de cada etiqueta, previamente definido na aplicação, é incrementado em 1000 a cada ocorrência de falha.

Por fim, foi observado que quando uma falha ocorria, durante a avaliação experimental, o protocolo TCP [12] tentava continuamente retransmitir uma mensagem que não foi entregue, previamente enviada de um processo MPI para outro que falhou, por um número máximo de tentativas, dado pela variável *tcp_retries2*. Em seguida, o erro era propagado do TCP para o MPI, causando a terminação de toda aplicação sem ter a chance dela se recuperar. A fim de adotar o mecanismo de tolerância a falhas proposto, é necessário mudar o valor da variável *tcp_retries2* no sistema operacional Linux (*/proc/sys/net/ipv4/tcp_retries2*), onde o valor padrão é 15, para um valor maior, de tal forma que seja concedido tempo suficiente para que a estratégia proposta possa detectar

e recuperar a aplicação.

3.4 Resumo

O objetivo principal deste trabalho foi modificar a aplicação alvo para incorporar procedimentos de tolerância a falhas de forma que seja minimizada a sobrecarga imposta a esta aplicação. Dessa forma, neste capítulo, os principais conceitos relacionados a detecção e recuperação de falhas que compõem o processo de tolerância a falhas são descritos e aqueles utilizados neste trabalho são justificados e destacados. Além disso, é feita uma revisão bibliográfica, onde são apresentados alguns trabalhos que propuseram diferentes maneiras de tolerar falhas em aplicações e diferenças em relação à solução proposta são apresentadas. Finalizando o capítulo, os algoritmos que compõem o mecanismo de tolerância a falhas são descritos e questões de implementação desse mecanismo são pontuadas.

No próximo capítulo, experimentos foram propostos a fim de analisar o impacto do mecanismo de tolerância a falhas em um cenário com e sem falhas. Além disso, as técnicas hierarquização da gravação de *checkpoint* e Replicação Passiva são avaliadas em um cenário com falhas, visando a possibilidade de redução da sobrecarga do mecanismo de tolerância a falhas proposto. Por fim, testes são realizados variando os parâmetros do mecanismo de tolerância a falhas, com o objetivo de discutir a escolha dos valores desses parâmetros e medir seu impacto na aplicação.

Capítulo 4

Resultados Computacionais

Neste capítulo são apresentados os resultados experimentais do mecanismo de tolerância a falhas proposto. Os principais aspectos a serem analisados são: o impacto desse mecanismo na aplicação em um cenário com e sem falhas, a possibilidade de redução desse impacto empregando as técnicas de hierarquização da gravação de *checkpoint* e Replicação Passiva e o impacto que a escolha dos valores dos parâmetros que regulam esse mecanismo gera na aplicação. Na Seção 4.1, os experimentos são detalhados e seus resultados e conclusões são expostos. Na Seção 4.2 discute-se sobre a escolha dos parâmetros do mecanismo de tolerância a falhas e seu consequente impacto em termos de desempenho.

4.1 Experimentos

A aplicação original e os procedimentos de tolerância a falhas foram implementados usando as linguagens C++ e Fortran em conjunto com a biblioteca Intel MPI. Os experimentos foram executados em um *cluster* composto por 40 nós dos quais somente 16 foram disponibilizados. Cada nó desse *cluster* contém dois processadores Intel Xeon E5430 2.66GHz Quad core com 12MB de cache L2 cada e 16 GB de memória RAM por nó, executando o sistema operacional Red Hat Enterprise Linux (RHEL) 5.3 , e interconectados por uma rede Gigabit Ethernet.

Testes foram executados utilizando uma instância real do problema sobre um modelo de velocidades homogêneo, sendo cada processo responsável inicialmente por um subdomínio de tamanho $210 \times 210 \times 832$ pontos. Os *checkpoints* gravados nos experimentos são compostos por 16 matrizes: 2 tridimensionais com mesmo tamanho do subdomínio local e 14 bidimensionais com tamanho 210×210 pontos. Essas matrizes são preenchidas com números do tipo *float* que na Linguagem C++ possui tamanho de 4 *bytes*. Logo,

o tamanho total desse checkpoint é de aproximadamente 282 Mbytes. A fim de verificar a viabilidade da gravação de *checkpoint* com tamanho de 282 Mbytes, testes iniciais foram realizados com uma instância de 24 processos de aplicação alocados em três nós e com gravação de *checkpoint* a cada 250 iterações. Além disso, os resultados obtidos são provenientes da média de 3 execuções e o desvio padrão obtido ficou abaixo de 2%. Os resultados mostraram que a sobrecarga da gravação das matrizes sem compressão ficou em torno de 136% em relação ao tempo de execução da aplicação original, ou seja, sem qualquer mecanismo de tolerância a falhas. Logo, fica evidenciado a inviabilidade de gravar checkpoints sem que seja feito algum tipo de compressão dessas matrizes.

Nesse sentido, este trabalho emprega a biblioteca de código aberto zlib [25] para fazer compressão sem perda dos dados dessa matrizes. Além de ser de código aberto, de acordo com os experimentos realizados em (RATANAWORABHAN et al., 2006), o algoritmo de compressão Deflate [24] usado nesta biblioteca se mostrou mais rápido que os algoritmos dos compactadores 7-zip, rar, zzip e bzip2, e obteve a terceira melhor taxa de compressão dentro desse grupo de compactadores. Nos experimentos aqui propostos, cada checkpoint local é armazenado em diferentes arquivos em um repositório comum gerenciado pelo protocolo NFS (*Network File System*). Além disso, os seguintes parâmetros foram usados nos testes: $timeout = 60$ segundos; $timeout\_inspeção = 180$ segundos; e dois intervalos de gravação de *checkpoints*, a cada 250 e 500 iterações. Em todos os testes, a aplicação RTM executou 3077 iterações, resultando em 12 *checkpoints* para o intervalo de 250 iterações e 6, para 500.

4.1.1 Análise da sobrecarga do mecanismo de tolerância a falhas em um cenário sem falhas

O primeiro conjunto de testes analisa a sobrecarga do monitoramento e gravação de *checkpoints* em um cenário sem falhas. A Tabela 4.1 mostra o número de processos N onde n_x e n_y são o número de *cores* por nó e de nós, a dimensão total do problema, a média de cinco execuções TE do código original sem qualquer mecanismo de tolerância a falhas, a média de cinco execuções TE_{250} e TE_{500} com o mecanismo de tolerância a falhas e a sobrecarga desse mecanismo S_{250} e S_{500} utilizando 250 e 500 iterações como intervalo de gravação de checkpoint, respectivamente.

Como esperado, o tempo de execução da aplicação e as sobrecargas dos procedimentos de tolerância a falhas crescem com o número de processos e a sobrecarga foi menor nos testes que empregam o intervalo de 500 iterações comparado àqueles que utilizaram 250

Tabela 4.1: *Sobrecarga do mecanismo de tolerância a falhas com intervalos de gravação de checkpoint de 250 e 500 iterações (TE_{250} e TE_{500}).*

N ($n_x * n_y$)	Dimensão do Domínio Global	TE (s)	TE_{250} (s)	S_{250} (%)	TE_{500} (s)	S_{500} (%)
24 (8×3)	$1600 \times 600 \times 800$	3609.35	3648.37	1.08	3627.35	0.50
32 (8×4)	$1600 \times 800 \times 800$	3560.65	3627.33	1.87	3595.32	0.97
64 (8×8)	$1600 \times 1600 \times 800$	3812.89	3987.31	4.57	3915.06	2.68
128 (8×16)	$1600 \times 3200 \times 800$	3956.72	4165.27	5.27	4063.90	2.71

iterações. A partir desses resultados, pode-se observar que a degradação de desempenho causado pelo monitoramento e gravação de *checkpoints* não foi substancial, variando de cerca de 1% e 0.5% com 24 processos até 5.27% e 2.71% com 128 processos, para 250 e 500 iterações, respectivamente. A sobrecarga aumentou levemente quando mais de 64 processos foram empregados, indicando que a abordagem proposta pode ser escalável.

4.1.2 Análise da sobrecarga do mecanismo de tolerância a falhas em um cenário com falhas

O próximo experimento considera a ocorrência e recuperação de uma falha durante a execução da aplicação RTM nos seguintes cenários:

(i) falha no início, 50 iterações após a primeira gravação de *checkpoint*;

(ii) falha no meio da execução, depois de 50 iterações após a sexta e terceira gravação de *checkpoint*, para 250 e 500 iterações, respectivamente.

(iii) falha no fim, 50 iterações após o última gravação de *checkpoint*.

Sejam TE_m e TE_{m-1} o tempo de execução da aplicação original livre de falhas com m e $m-1$ nós, respectivamente, e seja n_t o número total de checkpoints e n_c o número de checkpoints gravados antes da falha, respectivamente, pela versão tolerante a falhas (FT-RTM). Então, o tempo mínimo de execução (TME) de FT-RTM com uma única falha é dado pela Equação 4.1 :

$$TME = \frac{n_c}{n_t + 1} * TE_m + \frac{n_t + 1 - n_c}{n_t + 1} * TE_{m-1} \tag{4.1}$$

Os resultados na Tabela 4.2 são de novo o tempo médio, em segundos, de cinco execuções com intervalo de gravação de *checkpoint* de 250 iterações, que mostraram um

desvio padrão menor que 1%. Em todos os casos, a sobrecarga de uma única falha foi menor que 14%, mostrando que a abordagem proposta é viável e mais atraente do que reiniciar a aplicação do início. Na Tabela 4.3, são apresentados os resultados quando utilizado o valor de 500 iterações para o intervalo de gravação de *checkpoint*. Os resultados mostram comportamento semelhante àquele apresentado na Tabela 4.2, mas com valores de sobrecarga menores, devido à menor quantidade de gravações de *checkpoints* realizadas durante a execução.

Tabela 4.2: *Tempos de execução médio (em segundos) com e sem falhas para os cenários (i), (ii) e (iii) com intervalo de gravação de checkpoint de 250 iterações e suas sobrecargas.*

	N	**24**	**32**	**64**	**128**
Cenário (*i*)	***TME***	4935.68	4521.16	4245.92	4497.13
	1 falha	5390.90	4894.97	4639.15	4953.95
	Sobrecarga (%)	9.22	8.27	9.26	10.16
Cenário (*ii*)	***TME***	4383.04	4120.95	4065.49	4271.96
	1 falha	4731.40	4441.49	4509.76	4743.41
	Sobrecarga (%)	7.95	7.78	10.93	11.04
Cenário (*iii*)	***TME***	3719.88	3640.69	3848.97	4001.76
	1 falha	4001.30	3957.74	4324.22	4559.41
	Sobrecarga (%)	7.57	8.71	12.35	13.94

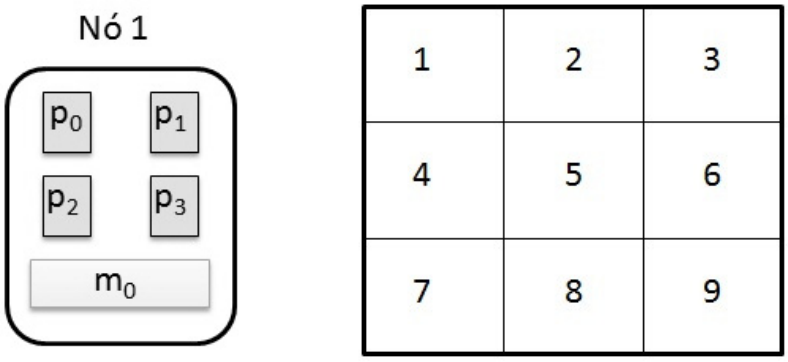
Tabela 4.3: *Tempos de execução médio (em segundos) com e sem falhas para os cenários (i), (ii) e (iii) com intervalo de gravação de checkpoint de 500 iterações e suas sobrecargas.*

	N	**24**	**32**	**64**	**128**
Cenário (*i*)	***TME***	4840.94	4452.56	4214.99	4458.52
	1 falha	5252.34	4811.36	4585.22	4867.28
	Sobrecarga (%)	8.50	8.06	8.78	9.17
Cenário (*ii*)	***TME***	4430.41	4155.25	4080.95	4291.26
	1 falha	4671.71	4393.22	4445.39	4592.24
	Sobrecarga (%)	5.45	5.73	8.93	7.01
Cenário (*iii*)	***TME***	3814.62	3709.30	3879.90	4040.36
	1 falha	3925.49	3917.55	4106.58	4316.48
	Sobrecarga (%)	2.91	5.61	5.84	6.83

Por fim, ressalta-se que os resultados obtidos até este experimento estão expostos no trabalho aceito na conferência *The European MPI Users Group* (*EuroMPI 2011*) [33].

4.1.3 Análise da hierarquização da gravação de checkpoint e da replicação passiva em um cenário com falhas

Apesar dos resultados obtidos nos experimentos anteriores, a proposta de gravação de *checkpoint* independente adotada pode se tornar um gargalo quando um grande número de processos efetuar conjuntamente a gravação de *checkpoint* e quando esta gravação for frequente. Para tratar esse problema, é possível hierarquizar o salvamento desses *checkpoints* a fim de reduzir o número de requisições simultâneas de gravação em disco. Nesta nova abordagem, em cada nó, o processo que possui o menor identificador é dito líder e responsável por gravar seu *checkpoint* e dos demais processos locais a ele. Para tal, esses processos locais dividem suas matrizes em várias partes que são enviadas já compactadas para o processo líder. O envio dos dados de cada processo em várias mensagens foi necessário, pois, através de experimentos preliminares, verificou-se que o envio de todas as matrizes de cada processo em uma única mensagem não resulta em redução de sobrecarga do mecanismo de tolerância a falhas quando empregados até 128 processos.



(a) (b)

Figura 4.1: Representação do nó de um cluster e da matriz de dados do processo p_1

Na Figura 4.1(a), observe que o processo p_0 presente no nó 1 é líder, pois ele possui o menor identificador. Sendo assim, os processos p_1, p_2 e p_3 devem enviar suas matrizes compactadas para p_0. Por exemplo, na Figura 4.1(b), p_1 divide suas matrizes em nove partes, comprime parte a parte e envia cada uma dessas em mensagens separadas para p_0. Ao receber cada uma dessas mensagens, p_0 abre o arquivo de *checkpoint* correspondente a p_1 e grava a parte da matriz recebida. Note que, apesar de reduzir o número de gravações simultâneas, o número de mensagens trocadas localmente aumenta. Dessa forma, um novo experimento foi realizado com o intuito de analisar se o emprego do método de gravação

de *checkpoint* hierarquizada surtirá efeitos na redução da sobrecarga do mecanismo de tolerância a falhas.

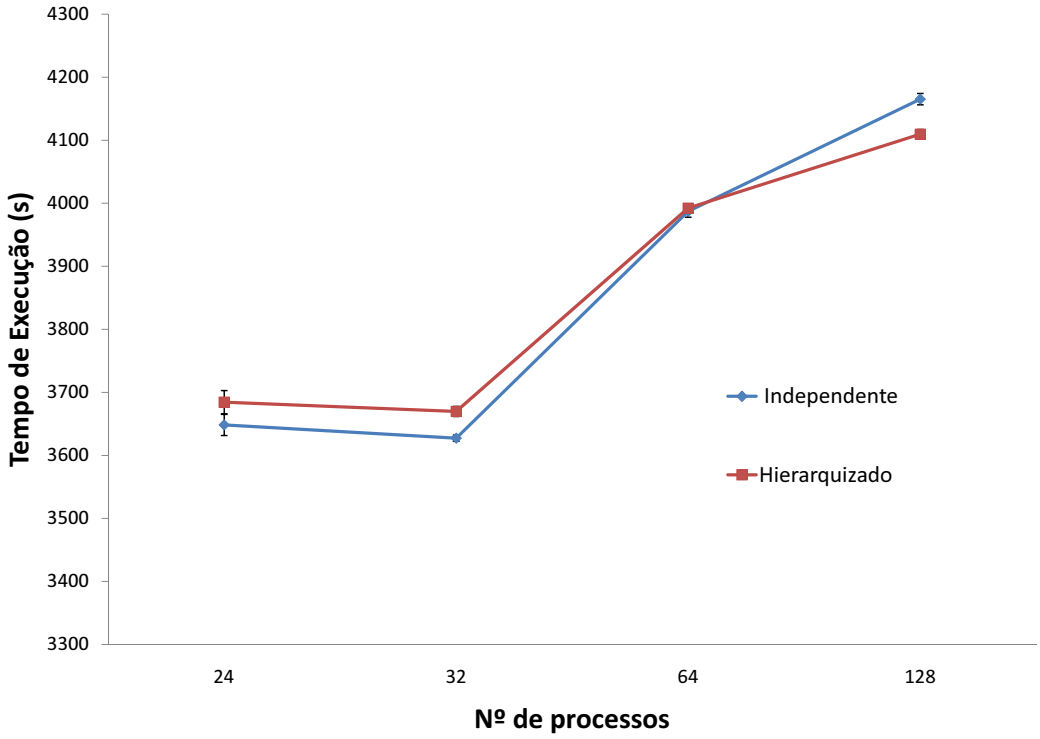


Figura 4.2: Tempo de Execução da Aplicação RTM com gravação de checkpoint independente e hierarquizado.

Na Figura 4.2 são apresentados os resultados obtidos a partir da média de cinco execuções e com intervalo de confiança de 95% quando empregou-se gravação de checkpoint hierarquizado e independente em um cenário livre de falhas. Nesse experimento foi utilizado o valor de 250 iterações para o intervalo de gravação de *checkpoints*. À partir do gráfico exibido na Figura 4.2 pode-se concluir que a aplicação dessa técnica não é vantajosa em pequena escala, pois a redução no número de requisições simultâneas de gravação em disco não compensou o aumento do número de mensagens. Observe que, em 128 processos há uma redução da sobrecarga do mecanismo de tolerância a falhas. Portanto, em grande escala, o uso dessa técnica talvez possa obter um ganho mais relevante comparado a técnica de gravação independente de *checkpoints*.

Outra técnica que pode ser empregada no mecanismo de tolerância a falhas com o intuito de reduzir sua sobrecarga sobre a aplicação é a Replicação Passiva. Na Figura 4.3, um exemplo do funcionamento dessa técnica é apresentado, onde ilustra-se um *cluster* composto por quatro nós. Considere que somente o nó 4 possui processos ociosos que serão utilizados caso ocorra falha em um outro nó e suponha que o nó 2 falhou. Nesse caso,

todos os outros monitores são avisados sobre a falha pelo monitor m_0 e, em seguida, cada um deles avisa aos seus processos locais sobre a ocorrência da falha para que os mesmos se recuperem. No caso do nó 4, os processos ociosos vão assumir o papel dos processos faltosos, restaurando o estado destes processos a partir da leitura de seus *checkpoints*. Por exemplo, p_{12} restaura o estado de p_4 à partir da leitura do checkpoint c_4 e assim sucessivamente. Se houver mais de um nó reserva, o nó que possuir o processo monitor de menor identificação é aquele terá que substituir processos faltosos pelos seus processos ociosos.

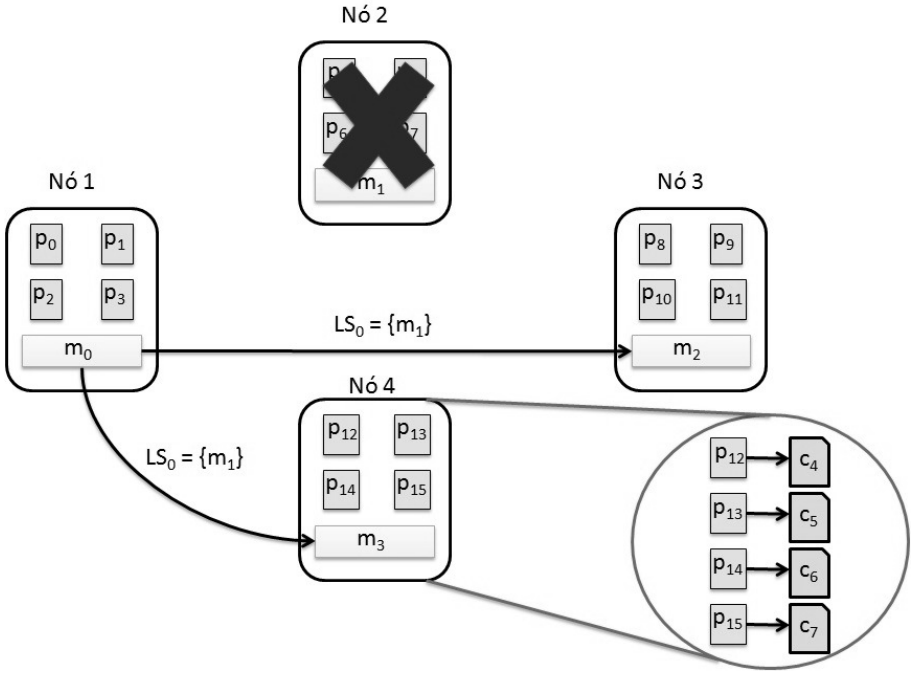


Figura 4.3: Exemplo do funcionamento da Replicação Passiva.

A principal vantagem no emprego desta técnica é evitar uma redivisão de carga entre os processos restantes, visto que um aumento no domínio local dos processos implica em maior tempo de execução. Na Tabela 4.4 são apresentadas as sobrecargas do mecanismo de tolerância a falhas quando utilizada a técnica de Replicação Passiva em um cenário com falhas no início (i), meio (ii) e fim (iii), conforme apresentado na Subseção 4.1.2. Essas sobrecargas foram calculadas em relação ao tempo médio de execução da aplicação RTM original, ou seja, sem qualquer mecanismo de tolerância a falhas. Novamente, o tempo de execução é a média de cinco execuções, todos os valores apresentados possuem desvio padrão menor que 1% e intervalo de confiança de 95%, onde não há interseção entre os intervalos estimados. Além disso, quando utilizada a Replicação Passiva, considere que todas as instâncias possuem um nó extra, ou seja, sempre 8 processos de aplicação ociosos a mais. Por exemplo, para instância de 24 processos, considere que foram inicializados 32

processos de aplicação dos quais 24 estavam executando e 8 estavam ociosos.

Tabela 4.4: *Tempos de execução médio (em segundos) da aplicação RTM original e da mesma usando a replicação passiva em um ambiente com falhas nos distintos cenários (i), (ii) e (iii) com intervalo de gravação de checkpoint de 250 iterações.*

	N	**24**	**32**	**64**	**128**
Cenário (*i*)	**Aplicação RTM**	3609.35	3560.65	3812.89	3956.72
	Replicação passiva	3912.06	4008.39	4256.60	4421.70
	Sobrecarga (%)	8.39	12.57	11.64	11.75
Cenário (*ii*)	**Aplicação RTM**	3609.35	3560.65	3812.89	3956.72
	Replicação passiva	3906.53	3924.28	4272.40	4532.55
	Sobrecarga (%)	8.23	10.21	12.05	14.55
Cenário (*iii*)	**Aplicação RTM**	3609.35	3560.65	3812.89	3956.72
	Replicação passiva	3879.89	3883.75	4275.81	4491.18
	Sobrecarga (%)	7.50	9.07	12.14	13.51

Os resultados apresentam comportamento semelhante àqueles obtidos nos experimentos da SubSeção 4.1.2, onde, em geral, a sobrecarga cresce conforme se aumenta o número de processos de aplicação envolvidos. Por outro lado, conforme esperado, observe que os tempos de execução, quando usada a replicação passiva, foram menores do que quando utilizada a redivisão de carga (Tabela 4.2). A fim de averiguar a magnitude dessa diferença entre as duas abordagens, considere a Tabela 4.5 onde é feita um comparação entre as abordagens de redivisão de carga e replicação passiva em um cenário com falhas no início (*i*), meio (*ii*) e fim (*iii*), e *checkpoints* são gravados a cada 250 iterações.

Tabela 4.5: *Tempos de execução médio (em segundos) com falhas, usando a técnica de redivisão de carga e replicação passiva para os cenários (i), (ii) e (iii) com intervalo de gravação de checkpoint de 250 iterações.*

	N	**24**	**32**	**64**	**128**
Cenário (*i*)	**Redivisão de Carga**	5390.90	4894.97	4639.15	4953.95
	Replicação passiva	3912.06	4008.39	4256.60	4421.70
	Diferença (%)	-27.43	-18.11	-8.25	-10.74
Cenário (*ii*)	**Redivisão de Carga**	4731.40	4441.49	4509.76	4743.41
	Replicação passiva	3906.53	3924.28	4272.40	4532.55
	Diferença (%)	-17.43	-11.64	-5.26	-4.45
Cenário (*iii*)	**Redivisão de Carga**	4001.30	3957.74	4324.22	4559.41
	Replicação passiva	3879.89	3883.75	4275.81	4491.18
	Diferença (%)	-3.03	-1.87	-1.12	-1.50

Note que, a diferença entre os tempos de execução das duas abordagens empregadas é maior quando a falha ocorre no início e menor quando a mesma ocorre no fim. Além disso, repare que na abordagem de redivisão de carga, a falha no início implica em maior

processamento por parte dos processos remanescentes por maior período de tempo se comparado com o cenário onde falhas ocorrem no meio e fim. Além disso, observe que a diferença é mais acentuada quando se utiliza um número menor de processos, uma vez que a perda de um nó tem um impacto maior sobre o tempo de execução da aplicação.

Vale ressaltar que apesar da vantagem apresentada pela técnica de Replicação Passiva, em sua utilização, uma vez que todos os nós reservas já foram usados na recuperação de falhas, a aplicação encerrará sem nenhuma possibilidade de recuperação se falhas adicionais ocorrerem. Logo, uma abordagem híbrida de recuperação de processos poderia ser empregada, onde os nós sobressalentes seriam usados e na falta destes, a redivisão de carga seria aplicada.

4.2 Influência dos parâmetros do mecanismo de tolerância a falhas

Segundo (STELLING et al., 1999), a decisão de quando a informação fornecida por um detector de falhas deve ser considerada verdadeira é de responsabilidade da aplicação. O detector de falhas não pode interpretar os seus resultados. Uma aplicação deve usar a informação fornecida pelo detector de falhas para tomar uma decisão com base na probabilidade de uma detecção de falhas estar errada. Esta decisão envolve o custo específico da aplicação de realizar alguma ação se a detecção for falsa e o custo de não realizar aquela ação se a detecção de fato for verdadeira.

Neste trabalho, as variáveis *timeout* e *timeout_ inspeção* utilizadas no mecanismo de tolerância a falhas proposto representam o tempo máximo de espera por uma mensagem de *heartbeat* ("Eu estou vivo") e o intervalo de tempo para considerar falho um processo suspeito, respectivamente. Além do intervalo de gravação de *checkpoint*, essas duas variáveis são parâmetros chave para redução de sobrecarga e devem ser ajustados conforme o ambiente onde este mecanismo será empregado. A fixação dos valores desses parâmetros implica em um *trade-off* entre corretude e eficiência. Para não exceder o limite de tempo tão cedo, os valores de *timeout* e *timeout_ inspeção* poderiam ser determinados de forma conservadora, ou seja, valores muito grandes. No entanto, isto implica em uma demora na detecção de falhas, consequentemente acarretando em maior sobrecarga a aplicação. A orientação é definir os valores desses parâmetros tão grandes quanto necessários, mas tão pequenos quanto possíveis [31]. Portanto, determinar bons valores para esses parâmetros ainda é um desafio. Nesse sentido, foram realizados experimentos onde diversos valores

para *timeout* e *timeout_ inspeção* foram empregados a fim de avaliar o impacto dessa escolha no tempo da aplicação. Os resultados de todos experimentos estão com 95% de intervalo de confiança e foram obtidos pela média dos resultados de três execuções.

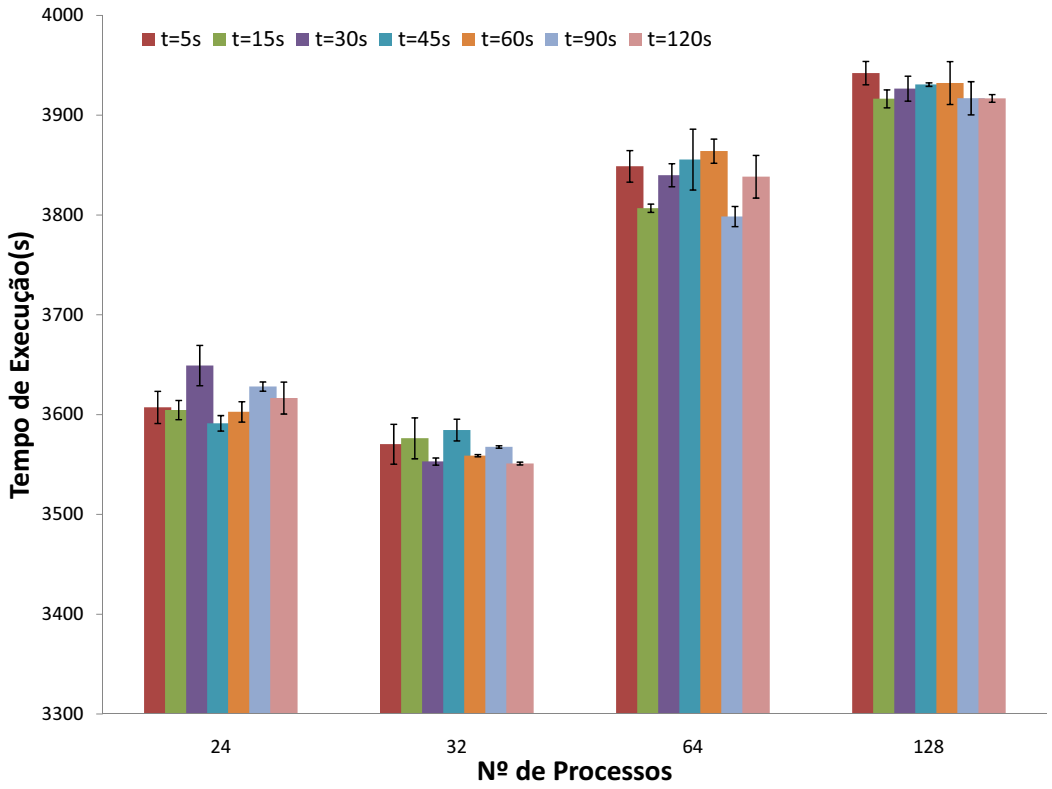


Figura 4.4: Avaliação do processo de monitoramento para diferentes valores de *timeout* em um cenário livre de falhas.

O primeiro experimento visa avaliar o impacto do monitoramento a cada *timeout* unidades de tempo realizado pelos processos monitores. Uma vez que em cada nó, os processos monitores compartilham um núcleo de processamento (*core*) com um processo de aplicação, um frequente monitoramento pode levar a um aumento no tempo de execução da aplicação. Note que esse processo de aplicação irá levar mais tempo para completar o passo de execução e atrasará todos os outros, pois no final do passo de execução uma etapa de sincronização é realizada. Na Figura 4.4 apresenta-se o tempo de execução da aplicação somente com o mecanismo de detecção para diferentes valores de *timeout* (t=5 a 120 segundos) sem ocorrência de falhas. Os valores apresentados nesse experimento estão com desvio padrão menor que 1%.

Na Tabela 4.6 são apresentadas as sobrecargas do menor e maior tempo de execução obtido entre os diferentes valores de *timeout* empregados em relação a execução da aplicação sem o mecanismo de tolerância a falhas. A partir dos resultados apresentados, constata-se que a influência de *timeout* é irrelevante, já que a diferença entre os tempos

de execução para cada valor de *timeout* é relativamente pequena, o intervalo de confiança dos valores se intercedem e a maior sobrecarga obtida é de 1.34%.

Tabela 4.6: *Sobrecarga do mecanismo monitoramento com o menor e maior tempo de execução (TE_{menor} e TE_{maior}) obtidos entre um conjunto de diferentes valores de timeout.*

N	TE (s)	TE_{menor} (s)	S_{menor} (%)	TE_{maior} (s)	S_{maior} (%)
24	3609.35	3591.35	-0.50	3649.18	1.10
32	3560.65	3551.05	-0.27	3584.58	0.67
64	3812.89	3798.59	-0.37	3864.07	1.34
128	3956.72	3916.53	-1.02	3942.31	-0.36

No segundo experimento, para os mesmos valores de *timeout* usados anteriormente, foram feitos testes em um ambiente onde falhas ocorrem no início, ou seja, 50 iterações após a gravação do primeiro *checkpoint* e foram coletados os tempos de detecção médio das três execuções. O valor de *timeout_ inspeção* foi fixado em três vezes o valor de *timeout* e se utilizou 250 iterações para o intervalo de gravação de *checkpoint*.

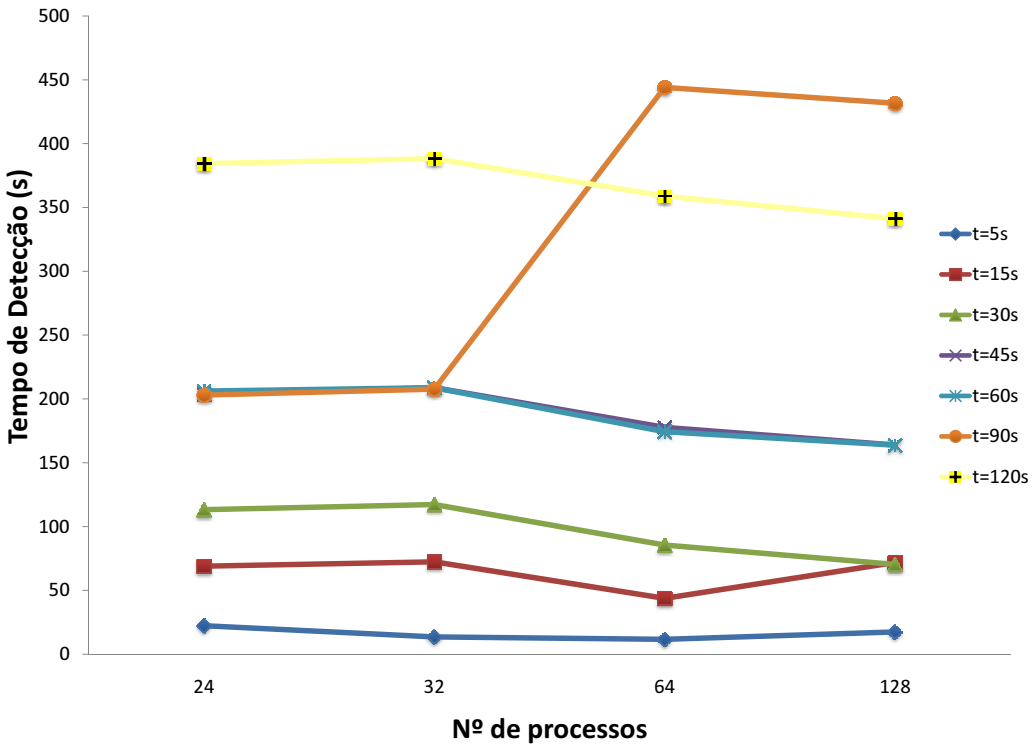


Figura 4.5: Avaliação do processo de monitoramento para diferentes valores de *timeout* em um cenário com falhas.

Os valores apresentados nesse experimento possuem desvio padrão menor que 3%. Na Figura 4.5, observa-se que, em geral, quanto maior o valor de *timeout*, maior o tempo gasto na detecção da falha. Entretanto, note que os valores de 45 e 60 segundos obtiveram

resultados muito próximos e que quando empregado o valor de 90 segundos para *timeout*, a partir de 64 processos, o tempo de detecção foi maior se comparado com os tempos obtidos quando usado 120 segundos para *timeout*.

A fim de explicar melhor esses resultados, considere o exemplo ilustrado na Figura 4.6, onde se tem a representação de um intervalo variando 0 a 180 segundos. Além disso, considere o valor de 60 segundos para *timeout*, 180 segundos para *timeout_ inspeção* e que uma falha ocorra 20 iterações após a conclusão da gravação do primeiro *checkpoint*. Dado que para diferentes instâncias são obtidos diferentes tempos de iteração, o tempo em que a falha ocorre não é o mesmo. Logo, os pontos F_1 e F_2, representam no intervalo o ponto de falha quando empregados 24 e 64 processos da aplicação, respectivamente. Vale ressaltar que o tempo de cada iteração é referente ao tempo de computação do subdomínio local acrescido do tempo para troca de mensagens.

Figura 4.6: Cálculo de tempo de detecção

Considere uma instância que utiliza 24 processos e que no instante 0 segundos os monitores trocaram mensagens de heartbeat e que as mesmas foram recebidas por todos antes do ponto de falha F_1. Quando completados os primeiros 60 segundos, os processos monitores, através do procedimento *Detecção()* (Algoritmo 3), verificam se a mensagem de *heartbeat* do monitor alvo chegou. Nesse caso, a mensagem chegou e por isso somente nos próximos 60 segundos, o processo monitor que falhou é considerado suspeito. Como os intervalos de tempo *timeout* e *timeout_ inspeção* correm em paralelo, em 180 segundos, o procedimento *Inspeção* (Algoritmo 5) de cada processo monitor verifica se existe algum processo suspeito e, em caso positivo, considera-o como falho e propaga tal informação para os outros monitores.

Em uma outra situação, com 64 processos, suponha que a falha ocorra no ponto F_2 e que dentro do intervalo onde esse ponto de falha se encontra, todos os monitores já receberam as mensagens de *heartbeat* antes do ponto F_2. Seguindo o mesmo raciocínio apresentado para a situação anterior, perceba que, somente em 180 segundos, um monitor poderá suspeitar da falha do seu monitor alvo. Note também que, em 180 segundos, o procedimento *Inspeção* (Algoritmo 5) é executado. Por questões de implementação, os intervalos de *timeout* e *timeout_ inspeção* não são sincronizados, consequentemente, a verificação da existência de processos suspeitos (*Inspeção*) pode ocorrer antes mesmo do processo que falhou ser considerado suspeito pelo procedimento *Detecção()*, implicando

assim na espera de mais 180 segundos para que esse processo suspeito seja considerado falho. Portanto, em relação a instância de 24 processos, o tempo de detecção será muito maior.

O último teste visa identificar a influência do parâmetro *timeout_ inspeção* sobre a aplicação. Para isso, empregou-se três valores distintos nesse experimento. O valor do parâmetro *timeout* e intervalo de *checkpoint* foram fixados em 60 segundos e 250 iterações, respectivamente. Além disso, os valores apresentados nesse experimento estão com desvio padrão menor que 2%.

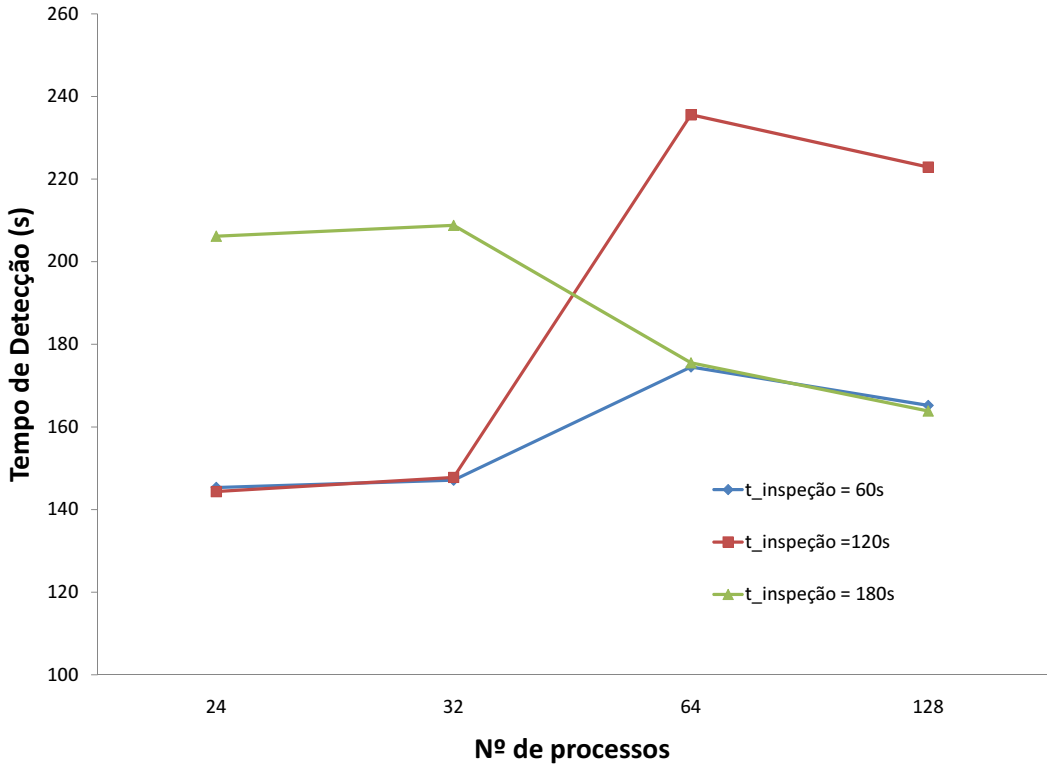


Figura 4.7: Avaliação do processo de monitoramento para diferentes valores de *timeout_ inspeção* em um cenário com falhas.

Na Figura 4.7 note que inicialmente os tempos de detecção quando empregados os valores de 60 e 120 segundos para *timeout_ inspeção* foram próximos. A partir de 64 processos, os tempos de detecção obtidos para 120 segundos de *timeout_ inspeção* foram maiores que quando utilizados 180 segundos. Esse fato pode ser explicado pelo exemplo mostrado anteriormente na Figura 4.6. Logo, conclui-se que além do parâmetro *timeout_ inspeção*, o ponto onde a falha ocorre influencia no comportamento do detector de falhas. Consequentemente, nem sempre o maior valor *timeout_ inspeção* implicará em maior sobrecarga na aplicação.

4.3 Resumo

O objetivo principal desse capítulo foi analisar o mecanismo de tolerância a falhas quanto ao seu impacto em um cenário com e sem falhas. Além disso, prevendo um gargalo em escalas maiores, as técnicas de hierarquização da gravação de *checkpoint* e Replicação Passiva foram avaliadas em um cenário com falhas, a fim de verificar a possibilidade de seu emprego para reduzir a sobrecarga do mecanismo de tolerância a falhas proposto. Por fim, testes foram realizados variando os parâmetros que regulam o mecanismo de tolerância a falhas, com o objetivo de discutir a escolha dos valores desses parâmetros e medir seu impacto na aplicação.

Os resultados obtidos mostraram que a degradação de desempenho da aplicação causado pelos procedimentos de tolerância a falhas em um cenário livre de falhas não foi substancial. Adicionalmente, concluiu-se que o emprego desse mecanismo é viável e mais atraente do que reiniciar a aplicação do início, visto que a maior sobrecarga obtida em um cenário com falhas foi menor que 14%. O emprego da técnica de hierarquização da gravação de *checkpoint* para reduzir a sobrecarga do mecanismo de tolerância a falhas se mostrou promissora, uma vez que, no experimento realizado, há uma redução dessa sobrecarga para 128 processos, que pode se acentuar em escalas maiores. Além disso, os resultados obtidos com a Replicação Passiva mostraram que a utilização de um mecanismo de recuperação híbrido, na ocorrência de falhas, implicará em menor sobrecarga para a aplicação. Por fim, como esperado, se observou que quanto menor o número de gravações de *checkpoint* realizadas, menor é a sobrecarga imposta a aplicação. Também, constatou-se irrelevante a influência do parâmetro *timeout* em um cenário sem falhas. Em um cenário com falhas, em geral, o que se observou é que quanto maior o valor de *timeout* maior o tempo levado para se detectar falhas. Entretanto, o mesmo comportamento não ocorreu com o parâmetro *timeout_inspeção*, o que levou a conclusão de que o ponto onde as falhas ocorrem também influenciam no tempo total de detecção.

Capítulo 5

Conclusões e Trabalhos Futuros

Aplicações paralelas de alto desempenho são especificadas a fim minimizar o tempo de execução para obter uma solução e, em geral, o padrão MPI de troca de mensagens tem sido utilizado para sua execução em diversas plataformas paralelas e distribuídas. A aplicação de processamento sísmico alvo deste trabalho emprega a biblioteca comercial Intel MPI em sua composição, pois seu foco se dá na maximização do desempenho. Mesmo quando executada em um *cluster* com centenas de núcleos de processamento, essa aplicação pode necessitar de dias a meses de tempo de computação, sendo assim prudente protegê-la das falhas de componentes do *cluster*. Nesse sentido, a principal contribuição deste trabalho foi desenvolver um mecanismo de tolerância a falhas viável, eficiente, agregado à aplicação alvo e que pode ser empregado em diversos problemas científicos que fazem uso da técnica de decomposição de domínio.

O mecanismo de tolerância a falhas proposto é composto por processos de detecção e recuperação, que utilizam o detector da classe $\diamond Q$ e o protocolo de gravação *checkpoint* não coordenado, respectivamente. O particionamento do mecanismo de detecção de falhas nas *threads Detecção()*, *Heartbeat()*, *Inspeção()* e *Propagação()* se mostrou eficaz, pois, por definição, *threads* são processos mais leves [63], a maior sobrecarga da camada de monitoramento obtida em um cenário sem falhas foi de 1,34% e as falhas injetadas nos experimentos foram detectadas corretamente. Ressalta-se que, devido aos valores dos parâmetros empregados nos experimentos realizados, detecções errôneas não ocorreram.

Em conjunto, as abordagens utilizadas nos processos de detecção e recuperação apresentaram sobrecargas razoavelmente baixas em um cenário livre de falhas, variando de cerca de 1% e 0.5% com 24 processos até 5.27% e 2.71% com 128 processos, quando empregados 250 e 500 iterações como intervalo de gravação de *checkpoint*, respectivamente. Como esperado, verificou-se que quanto menor o número de gravações de *checkpoint* rea-

lizadas ao longo da execução, menor é a sobrecarga imposta a aplicação. Em um cenário com falhas, o que se observou, em geral, é que quanto maior o intervalo de monitoramento de falhas, maior o tempo levado para detectá-las e que o ponto onde essas falhas ocorrem também influencia no tempo total de detecção. A maior sobrecarga apresentada nos experimentos onde falhas ocorreram em três instantes diferentes foi de 14%, mostrando assim que a abordagem proposta é aplicável em termos práticos devido ao seu baixo impacto, escalável nos experimentos realizados e melhor do que reiniciar a execução da aplicação a partir do início.

A fim de averiguar a possibilidade de reduzir ainda mais a sobrecarga do mecanismo de tolerância a falhas sobre a aplicação, foram investigados o esquema de gravação hierárquica de *checkpoint* e Replicação Passiva. A partir dos resultados obtidos, o esquema de hierarquização da gravação de *checkpoint*, onde um processo líder presente em cada nó do *cluster* recebe o *checkpoint* dos demais processos locais a ele e efetua a gravação do seu e desses *checkpoints* recebidos, se mostrou promissor, pois, verificou-se uma redução da sobrecarga sobre a aplicação a partir de 128 processos, que pode se acentuar em escalas maiores e reduzir o gargalo de usar um repositório comum para armazenamento de *checkpoint*. Adicionalmente, constatou-se que a utilização de uma abordagem de recuperação híbrida, onde, na ocorrência de falhas, a técnica de Replicação Passiva, que consiste na substituição dos processos faltosos pelos processos sobressalentes, seria usada e na falta destes, a redivisão de carga seria aplicada, pode reduzir ainda mais a sobrecarga sobre a aplicação.

Como trabalho futuro, uma análise mais aprofundada deve ser realizada em todos os experimentos com um maior número de nós. Além disso, o impacto de se utilizar um modelo de velocidades heterogêneo na aplicação pode ser investigado, uma vez que isso implica na heterogeneidade dos valores contidos nas matrizes, consequentemente refletindo na taxa de compactação de *checkpoints*. Adicionalmente, um estudo com tamanhos maiores dos subdomínios locais pode ser conduzido a fim de se investigar quão impactante será para aplicação. Também, um estudo pode ser feito empregando detectores de falhas adaptáveis [9, 37] com o objetivo de verificar uma possível redução no tempo de detecção de falhas de forma que seja minimizada ainda mais a sobrecarga sobre a aplicação. Por fim, o mecanismo de detecção de falhas proposto pode ser alterado a fim de suportar múltiplas falhas ao mesmo tempo, se aproximando assim das reais situações de ocorrência de falhas em grandes *clusters* de computadores.

Referências

[1] Agbaria, A., Friedman, R. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. *Cluster Computing 6* (2003), 227–236.

[2] Aguilera., M. K., Chen, W., Toueg, S. On Quiescent Reliable Communication. *SIAM Journal on Computing 29* (2000), 2040–2073.

[3] Araya-Polo, M., Rubio, F., de la Cruz, R., Hanzich, M., Cela, J., Scarpazza, D. 3D Seismic Imaging through Reverse Time Migration on Homogeneous and Heterogeneous Multi-Core Processors. *Scientific Programming 17*, 1-2 (2009), 185–198.

[4] Avizienis, A., Laprie, J.-C., Randell, B. Fundamental Concepts of Dependability. *Information Survivability Workshop (ISW-2000)* (2000).

[5] Batchu, R., Dandass, Y., Skjellum, A., Beddhu, M. MPI/FT: a model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing 7*, 4 (2004), 303–315.

[6] Baysal, E., Kosloff, D., Sherwood, J. Reverse Time Migration. *Geophysics 48*, 11 (1983), 1514–1524.

[7] Bednar, J. B. A Brief History of Seismic Migration. *Geophysics 70*, 3 (2005), 3MJ–20MJ.

[8] Berkhout, A. J. *Seismic Migration: Imaging of Acoustic Energy by Wave Field Extrapolation*, 1 ed. Elsevier, 1984.

[9] Bertier, M., Marin, O., Sens, P. Implementation and Performance Evaluation of an Adaptable Failure Detector. Em *Proceedings of the 2002 International Conference on Dependable Systems and Networks* (2002), DSN '02, IEEE Computer Society, p. 354–363.

[10] Birman, K. P., Constable, B., Hayden, M., Hickey, J., Kreitz, C., van Renesse, R., Rodeh, O., Vogels, W. The Horus and Ensemble Projects: Accomplishments and Limitations. Em *Proceedings of the DARPA Information Survivability Conference and Exposition* (2000), vol. 1, IEEE Computer Society, p. 149–161.

[11] Bouteiller, A., Herault, T., Krawezik, G., Lemarinier, P., Cappello, F. MPICH-V project: A multiprotocol automatic fault-tolerant MPI. *International Journal of High Performance Computing Applications 20*, 3 (2006), 319–333.

[12] Braden, R. Requirements for Internet Hosts–Communication Layers - RFC 1122. Disponível em: http://www.ietf.org/rfc/rfc1122.txt. Último acesso em Março de 2011, 1989.

[13] Brandão, D., Zamith, M., Clua, E., Montenegro, A., Bulcão, A., Madeira, D., Kischinhevsky, M., Leal-Toledo, R. C. Performance Evaluation of Optimized Implementations of Finite Difference Method for Wave Propagation Problems on GPU Architecture. *International Symposium on Computer Architecture and High Performance Computing Workshops* (2010), 7–12.

[14] Bulcão, A. *Modelagem e Migração Reversa no Tempo empregando operadores elásticos e acústicos*. Tese de Doutorado, PEC-COPPE, Universidade Federal do Rio de Janeiro, 2004.

[15] Butler, R. M., Lusk, E. L. Monitors, messages, and clusters: the P4 parallel programming system. *Parallel Comput. 20* (1994), 547–564.

[16] Buyya, R. *High Performance Cluster Computing: Architectures and Systems*, vol. 1. 1999.

[17] Carcione, J. M., Herman, G. C., ten Kroode, A. P. E. Seismic modeling. *Geophysics 67*, 4 (2002), 1304–1325.

[18] Cerjan, C., Kosloff, D., Kosloff, R., Reshef, M. A nonreflecting boundary condition for discrete acoustic and elastic wave equations. *Geophysics 50*, 4 (1985), 705.

[19] Chan, T. F., Mathew, T. P. Domain decomposition algorithms. *Acta Numerica 3* (1994), 61–143.

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

[21] Chandy, K., Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS) 3*, 1 (1985), 63–75.

[22] Cunha, P. E. M. Estratégias Eficientes para Migração Reversa no Tempo Pré-Empilhamento 3-D em Profundidade pelo Método das Diferenças Finitas. Dissertação de Mestrado, Programa de Pesquisa e Pós-Graduação em Geofísica, Universidade Federal da Bahia, 1997.

[23] Deschizeaux, B., Blanc, J. *Imaging earth's subsurface using CUDA*, vol. 3 of *GPU Gems 3*. Addison-Wesley Professional, 2007, cap. 38, p. 831–850.

[24] Deutsch, P. DEFLATE Compressed Data Format Specification version 1.3 - RFC 1951. Disponível em: http://www.ietf.org/rfc/rfc1951.txt. Último acesso em Março de 2011, 1996.

[25] Deutsch, P., Gailly, J.-L. ZLIB Compressed Data Format Specification version 3.3 - RFC 1950. Disponível em: http://www.ietf.org/rfc/rfc1950.txt. Último acesso em Março de 2011, 1996.

[26] Duarte, A., Rexachs, D., Luque, E. An Intelligent Management of Fault Tolerance in Cluster Using RADICMPI. Em *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 4192 of *LNCS*. Springer, 2006, p. 150–157.

[27] Dwork, C., Lynch, N., Stockmeyer, L. Consensus in the presence of partial synchrony. *Journal of the ACM 35*, 2 (1988), 288–323.

[28] Elnozahy, E., Alvisi, L., Wang, Y., Johnson, D. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR) 34*, 3 (2002), 375–408.

[29] Fagg, G., Dongarra, J. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2000), 346–353.

[30] Felber, P., Défago, X., Eugster, P. T., Schiper, A. Replicating corba objects: a marriage between active and passive replication. Em *Proceedings of the IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems II* (1999), Kluwer, B.V., p. 375–388.

[31] Freiling, F. C., Guerraoui, R., Kuznetsov, P. The failure detector abstraction. *ACM Comput. Surv. 43* (2011), 1–40.

[32] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT Press, 1994.

[33] Gonçalves, A., Bersot, M., Bulcão, A., Boeres, C., Drummond, L., Rebello, V. Fault Tolerance in an Industrial Seismic Processing Application for Multicore Clusters. Em *Recent Advances in the Message Passing Interface*, vol. 6960 of *Lecture Notes in Computer Science*. Springer, 2011, p. 264–271.

[34] Greve, F. G. P. Protocolos Fundamentais para o Desenvolvimento de Aplicações Robustas. *Minicursos SBRC 2005: Brazilian Symposium on Computer Networks* (2005), 330–398.

[35] Gropp, W., Smith, B. Users manual for the Chameleon parallel programming tools. *Math. and Comp. Science Div., Argonne Nat. Lab., ANL-93/23* (1993).

[36] Hargrove, P., Duell, J. Berkeley Laboratory Checkpoint/Restart (BLCR) for Linux Clusters. Em *Journal of Physics: Conference Series* (2006), vol. 46, IOP Publishing, p. 494–499.

[37] Hayashibara, N., Défago, X., Yared, R., Katayama, T. The ϕ Accrual Failure Detector. *IEEE Symposium on Reliable Distributed Systems* (2004), 66–78.

[38] Hoefler, T., Mehlan, T., Mietke, F., Rehm, W. A Survey of Barrier Algorithms for Coarse Grained Supercomputers. *Chemnitzer Informatik Berichte 04*, 03 (2004).

[39] Intel. Intel MPI Library Reference Manual. Disponível em: http://software.intel.com/en-us/articles/intel-mpi-library-documentation. Último acesso em Janeiro de 2011, 2011.

[40] Jin, H., Jespersen, D., Mehrotra, P., Biswas, R. High Performance Computing Using MPI and OpenMP on Multi-core Parallel Systems. *Parallel Computing* (2011), 1–22.

[41] Kale, L. V., Krishnan, S. CHARM++: A Portable Concurrent Object Oriented System Based on C++. Em *Proceedings of the 18th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (1993), ACM, p. 91–108.

[42] Kelly, K., Marfurt, K. *Numerical modeling of seismic wave propagation*, 13ª ed. Geophysics reprint series. Society of Exploration Geophysicists, 1990.

[43] Koopman, P. Elements of the self-healing system problem space. Em *Workshop on Software Architectures for Dependable Systems, International Conference on Software Engineering* (2003), p. 31–36.

[44] Kshemkalyani, A. D., Singhal, M. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008, cap. 15, p. 567–597.

[45] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[46] Larrea, M., Arevalo, S., Fernndez, A. Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. Em *Distributed Computing*, vol. 1693 of *Lecture Notes in Computer Science*. Springer, 1999, p. 34–49.

[47] Louca, S., Neophytou, N., Lachanas, A., Evripidou, P. MPI-FT: Portable Fault Tolerance Scheme for MPI. *Parallel Processing Letters 10*, 4 (2000), 371–382.

[48] Mitchell, A., Griffiths, D. *The finite difference method in partial differential equations*, 1ª ed. Wiley-Interscience, 1980.

[49] MPI. MPI: A Message-Passing Interface Standard, Version 2.2. Disponível em: www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf. Último acesso em Agosto de 2011.

[50] MPI Forum. Message Passing Interface (MPI) Forum. http://www.mpi-forum.org/, 2011.

[51] MPI Tutorial. Message Passing Interface (MPI) Tutorial. https://computing.llnl.gov/tutorials/mpi/, 2011.

[52] Mufti, I., Pita, J., Huntley, R. Finite-difference depth migration of exploration-scale 3-D seismic data. *Geophysics 61*, 3 (1996), 776.

[53] Nelson, V. P. Fault-Tolerant Computing: Fundamental Concepts. *Computer 23* (1990), 19–25.

[54] Ortigosa, F., Araya-Polo, M., Rubio, F., Hanzich, M., de la Cruz, R., Cela, J. M. Evaluation of 3D RTM on HPC platforms. *SEG Technical Program Expanded Abstracts 27*, 1 (2008), 2879–2883.

[55] Pasin, M., Fontaine, S., Bouchenak, S. Failure detection in large scale systems: a survey. Em *Network Operations and Management Symposium Workshops* (2008), IEEE, p. 165–168.

[56] Ratanaworabhan, P., Ke, J., Burtscher, M. Fast Lossless Compression of Scientific Floating-Point Data. Em *Proceedings of the Data Compression Conference* (2006), IEEE Computer Society, p. 133–142.

[57] Reynolds, A. Boundary conditions for the numerical solution of wave propagation problems. *Geophysics 43* (1978), 1099–1110.

[58] Schneider, F. B. *What good are models and what models are good?*, 2ª ed. ACM Press/Addison-Wesley Publishing Co., (1993), p. 17–26.

[59] Sergent, N., Défago, X., Schiper, A. Impact of a failure detection mechanism on the performance of consensus. Em *Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing* (2001), IEEE, p. 137–145.

[60] Silva, J., Rebello, V. Low Cost Self-healing in MPI Applications. Em *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 4757 of *LNCS*. Springer, (2007), p. 144–152.

[61] Silva, J. A. *Tolerância a Falhas para Aplicações Autônomas em Grades Computacionais*. Tese de Doutorado, Universidade Federal Fluminense, 2010.

[62] Squyres, J., Lumsdaine, A. A component architecture for lam/mpi. Em *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 2840 of *Lecture Notes in Computer Science*. Springer, (2003), p. 379–387.

[63] Stallings, W. *Operating Systems: Internals and Design Principles*, 6ª ed. Prentice Hall Press, 2008.

[64] Stelling, P., DeMatteis, C., Foster, I., Kesselman, C., Lee, C., von Laszewski, G. A fault detection service for wide area distributed computations. *Cluster Computing 2* (1999), 117–128.

[65] Vuillemin, J. A data structure for manipulating priority queues. *Communications of the ACM 21* (1978), 309–315.

[66] Weber, S. T. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. Disponível em: http://www.inf.ufrgs.br/∼taisy/disciplinas/textos/Dependabilidade.pdf. Último acesso em Agosto de 2011.

[67] Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., Alonso, G. Understanding replication in databases and distributed systems. Em *20th International Conference on Distributed Computing Systems* (2000), IEEE Computer Society, p. 464–474.

[68] Ziv, A., Bruck, J. *Checkpointing in parallel and distributed systems.* Parallel and Distributed Computing Handbook. McGraw-Hill, 1996, p. 274–302.