Capítulo 5

Técnicas para a Construção de Sistemas MPI Tolerantes a Falhas

Edson T. Camargo e Elias P. Duarte Jr.

Resumo

O MPI é um dos principais padrões para o desenvolvimento de aplicações paralelas e distribuídas baseadas no paradigma de troca de mensagens. Diversos sistemas de computação de alto desempenho são baseados em MPI. Um dos maiores desafios dos sistemas de alto desempenho diz respeito à confiabilidade, ou seja, à capacidade de oferecer serviços corretos ininterruptamente. Este minicurso tem como objetivo apresentar as principais técnicas empregadas para a construção de sistemas MPI tolerantes a falhas. São apresentadas técnicas tradicionais aplicadas a sistemas baseados em MPI, como a técnica rollback-recovery, incluindo suas variantes baseadas em checkpoints e em registro de mensagens. Além disso, também é apresentada a especificação ULFM (User Level Failure Mitigation), a mais recente proposta de tolerância a falhas para o padrão MPI.

5.1. Introdução

O MPI (*Message Passing Interface*) pode ser considerado o padrão *de facto* para o desenvolvimento de aplicações paralelas e distribuídas [MPI Forum 2015, Fagg and Dongarra 2000]. O padrão é baseado no paradigma de troca de mensagens, ou seja, é utilizado em ambientes computacionais onde os nodos acessam uma memória local e se comunicam através mensagens transmitidas em uma rede que conecta os nodos. O MPI consiste de um conjunto de bibliotecas de funções padronizadas pelo MPI-Fórum [Forum 2017b]. O padrão especifica rotinas para a comunicação entre processos, gerenciamento e criação de processos, entrada e saída de dados, gerenciamento de grupos, sincronização e o estabelecimento de topologias virtuais.

Embora o MPI seja amplamente utilizado, uma questão ainda em aberto no padrão é a tolerância a falhas. O padrão MPI parte do princípio que suas aplicações executam em uma infraestrutura computacional confiável e, desta forma, a tolerância a falhas não é contemplada em sua especificação. A falha de um único processo implica na interrupção de toda a aplicação. Mesmo as mais importantes implementações do padrão MPI, como

a Open MPI [Gabriel et al. 2004, open mpi.org 2017] e a MPICH [Gropp et al. 1996, mpich.org 2017] não são capazes de manter a aplicação em execução perante a falha de um único processo.

Os sistemas de computação de alto desempenho (*High Performance Computing* - HPC) empregam amplamente o padrão MPI [Fagg and Dongarra 2000]. Esses sistemas geralmente executam simulações científicas e industriais complexas que, potencialmente, são de longa duração, lidam com um volume de dados gigantesco e requerem uso intensivo de computação. A falha destes sistemas se traduz em prejuízos econômicos, até mesmo em termos do desperdício de energia, uma vez que o consumo desses sistemas é enorme. Além disso, falhas frequentes podem ser um impeditivo na execução de aplicações de longa duração. Estima-se que certos sistemas HPC de grande escala têm que lidar com a ocorrência de falhas em um intervalo de poucas horas [Egwutuoha et al. 2013]. Nesse contexto, projetar e construir mecanismos de tolerância a falhas efetivos e permitir que as aplicações completem adequadamente as suas execuções, apesar da ocorrência de falhas, é uma tarefa árdua. Abordagens e técnicas eficientes para tolerar falhas em aplicações HPC baseadas no padrão MPI são indispensáveis.

Este minicurso tem como objetivo apresentar algumas das principais técnicas para a construção de sistemas MPI tolerantes a falhas. O minicurso assume que o leitor tem familiaridade com o MPI; para uma introdução ao MPI recomenda-se [Pacheco 1996].

Inicialmente, é apresentada uma visão geral dos conceitos básicos de tolerância a falhas. A seguir, as principais técnicas de tolerância a falhas empregadas em sistemas MPI são apresentadas. Destaca-se a principal técnica aplicada à sistemas MPI, a *rollback-recovery* [Elnozahy et al. 2002, Egwutuoha et al. 2013]. No contexto da técnica de *rollback-recovery*, as abordagens baseadas em *checkpoints* e em registro de mensagens são descritas. Também são apresentados as técnicas de replicação máquina de estado [Charron-Bost et al. 2010] e a técnica chamada de *Algorithm-Based Fault Tolerance* (ABFT) [Du et al. 2012]. Trabalhos relacionados a cada um dessas técnicas são descritos. Além disso, a especificação ULFM (*User Level Failure Mitigation*) [Bland et al. 2013] é apresentada.

A ULFM é a mais recente proposta do MPI-Fórum para padronizar a semântica de tolerância a falhas em MPI. Através da ULFM o desenvolvedor da aplicação pode escolher a técnica de tolerância a falhas que melhor se adequa ao seu programa. Para tanto, a ULFM apresenta um conjunto de rotinas para detectar falhas, notificá-las e recuperar a capacidade dos processos de se comunicarem. Esse conjunto de rotinas é descrito juntamente com alguns exemplos de código que demonstram o seu funcionamento na prática.

O restante do minicurso está organizado da seguinte maneira. A seção 5.2 apresentada os modelos e os conceitos principais de tolerância a falhas. Ainda na seção 5.2 a técnica *rollback-recovery* é definida, bem como as suas variantes baseadas em *checkpoints* e em registro de mensagens. A seção 5.3 descreve a tolerância a falhas no padrão MPI. Essa seção inicia com a apresentação do padrão MPI e logo após as propostas de tolerância a falhas do MPI-Fórum são apresentadas. Na sequência, trabalhos que se apoiam nas abordagens de *rollback-recovery* são descritos. A seguir, são apresentadas as técnicas de replicação máquina de estado e ABFT. Por fim, a seção 5.4 apresenta a conclusão do minicurso.

5.2. Tolerância a Falhas: Definições Básicas

Um sistema confiável, ou tolerante a falhas, é aquele em que se pode ter confiança no seu funcionamento (em inglês *dependable*) [Avizienis et al. 2004]. A tolerância a falhas é a propriedade que garante a correta e eficiente operação de um sistema apesar da ocorrência de falhas em qualquer um dos seus componentes [Kshemkalyani and Singhal 2011].

Um sistema é projetado para executar corretamente e entregar serviços ao usuário de acordo com a sua especificação. Uma falha no serviço (*failure*) é caracterizada pelo não cumprimento da sua especificação ou devido à mesma não descrever adequadamente as funções do sistema. Uma falha no serviço é provocada por um erro (*error*). Um erro é ocasionado pela manifestação de uma falha (*fault*), ou defeito, em um dos componentes do sistema, que pode tanto ser interno quanto externo ao sistema. Por exemplo, estão suscetíveis a falhas os processos, os processadores, a memória ou a rede [Avizienis et al. 2004, Jalote 1994]. O tempo médio entre a ocorrência de falhas (*Mean Time Between Failures* - MTBF) é uma medida primária de confiabilidade do sistema baseada em análises estatísticas do sistema e de seus componentes [Kshemkalyani and Singhal 2011]. A medida é usualmente empregada para indicar o tempo previsto até uma falha ocorrer. Os sistemas HPC de grande escala, em especial os sistemas *petascale* e os futuros sistemas *exascale*, que podem atingir a ordem de 10^{15} e 10^{18} operações de ponto flutuante por segundo, respectivamente, apresentam um MTBF que pode chegar a poucas horas ou mesmo minutos [Di Martino et al. 2014, Egwutuoha et al. 2013, Cappello et al. 2009, El-Sayed and Schroeder 2013, Moody et al. 2010]. Esses sistemas estão sujeitos a diversos tipos de falhas [El-Sayed and Schroeder 2013, Schroeder and Gibson 2010].

Um modelo de falhas define o modo pelo qual os componentes do sistema podem falhar. Além disso, oferece uma classificação que especifica as suposições que podem ser feitas sobre o comportamento do componente quando o mesmo falha [Jalote 1994]. As falhas podem ser classificadas de acordo com as seguintes categorias: parada (*crash*), omissão, temporização e bizantinas. Laranjeira [Laranjeira et al. 1991] adiciona a essa classificação o modelo de falha por computação incorreta. Pode-se também citar dois modelos relacionados: *crash-recovery* e *fail-stop*. A seguir os modelos de falhas são descritos.

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

Conforme mencionado acima, há ainda a falha do tipo computação incorreta, que pode ser considerada como um subconjunto da falha bizantina. Esse tipo de falha ocorre

quando o componente não produz o resultado correto em resposta a uma tarefa correta recebida como entrada. Seguindo a classificação tradicional, a falha por computação incorreta inclui todas as demais exceto a bizantina.

O modelo de falha *crash* exclui a possibilidade de recuperação do componente. Se a recuperação é possível, o modelo de falhas é dito *crash-recovery*. Nesse modelo, um componente que constantemente para e se recupera é chamado de instável [Aguilera et al. 2000]. Quando os modelos de falhas são aplicados a sistemas distribuídos, os componentes correspondem a processos. Um processo geralmente guarda suas informações em um armazenamento do tipo estável ou volátil. Durante a recuperação apenas os dados armazenados em dispositivo estável são recuperados. As falhas em um sistema podem ainda ser classificadas de acordo com o modelo *fail-stop* [Guerraoui et al. 2011]. O modelo *fail-stop* inclui a falha do tipo *crash*, porém todo processo correto pode detectar a falha por parada do processo. A maioria trabalhos de tolerância a falhas em MPI assumem o modelo *fail-stop* [Bouteiller et al. 2006, Bland et al. 2013].

5.2.1. Detectores de Falhas

Um detector de falhas, a grosso modo, é um serviço que indica quais processos de um sistema distribuído estão falhos. Um detector de falhas é frequentemente implementado como um objeto local a cada processo [Chandra and Toueg 1996, Felber et al. 1999, Guerraoui et al. 2011]. Determinar entre um processo falho ou sem-falha é condição essencial para solucionar diversos problemas presentes nos sistemas distribuídos, como o consenso [Fischer et al. 1985]. O consenso, informalmente, permite que um conjunto de processos concordem sobre um valor único com base em valores propostos inicialmente, considerando que esses valores iniciais podem ser diferentes para cada processo. Um detector de falhas nem sempre detecta com precisão a falha de um processo [Guerraoui et al. 2011]. Da forma como foram propostos por Chandra e Toueg [Chandra and Toueg 1996], o detector é dito não-confiável, isto é, é possível que um processo tenha falhado mas não seja suspeito de ter falhado, bem como um processo tenha sido suspeito sem ter realmente falhado.

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

Entre as classes de detectores de falhas definidas, destacam-se a P (*perfect*), com as propriedades de completude forte e exatidão forte e o $\diamond$W que apresenta completude fraca e exatidão fraca. Chandra e Toueg provaram que detectores que possuem a completude fraca são equivalentes aos detectores com completude forte e provaram ser o detector de falhas $\diamond$W o mais fraco que permite o consenso [Chandra and Toueg 1996].

A implementação de detectores de falhas em sistemas distribuídos geralmente faz uso de mensagens de monitoramento. O monitoramento de processos é realizado através de trocas de mensagens. O envio e de recebimento das mensagens deve obedecer a um limite de tempo (*timeout*). De acordo com Felber et al. [Felber et al. 1999] existem basicamente dois modelos de monitoramento, conhecidos como *push* e *pull*. No primeiro, cada processo monitorado envia periodicamente mensagens do tipo "*I am alive*", também chamadas de *heartbeat*, para o detector que o está monitorando. No modelo *pull* a direção é oposta, isto é, o detector questiona o processo monitorado ("*Are you alive?*"). Se a troca de mensagens ocorrer dentro do intervalo de tempo definido significa que os processos monitorados estão operacionais.

É possível citar ainda outros modelos de detectores, como o modelo Dual [Felber et al. 1999], o modelo Gossip [Renesse et al. 1998] e o detector *heartbeat* proposto por Aguilera et al. [Aguilera et al. 1997]. Basicamente, o modelo Dual combina o modelo *push* e o modelo *pull*. No modelo Gossip o monitoramento é probabilístico. Os processos que recebem as informações são escolhidos aleatoriamente. Uma das vantagens é que o modelo apresenta boa escalabilidade. O modelo de Aguilera, ao invés de usar um *timeout*, somente incrementa um contador a cada mensagem de monitoramento.

5.2.2. *Rollback-Recovery*

A técnica de tolerância a falhas *rollback-recovery* é frequentemente empregada para prover tolerância a falhas em aplicações de alto desempenho [Stellner 1996, Elnozahy et al. 2002, Fagg and Dongarra 2000, Egwutuoha et al. 2013]. Desse modo, as aplicações podem reiniciar a partir de um estado salvo previamente. A técnica assume um sistema distribuído onde os processos da aplicação se comunicam através de uma rede e têm acesso a um dispositivo de armazenamento confiável que sobrevive a falhas. Periodicamente, os processos salvam informações de recuperação no dispositivo confiável durante a sua execução sem-falhas. Após a ocorrência de uma falha, a aplicação usa as informações de recuperação para reiniciar a sua computação a partir de um estado anterior. As informações de recuperação incluem os *checkpoints*, isto é, os estados dos processos participantes. Alguns protocolos também incluem informações sobre a recepção de mensagens. Basicamente, um estado global consistente é aquele em que se o estado de um processo reflete uma mensagem recebida, então o estado correspondente do emissor deve refletir o envio daquela mensagem [Kshemkalyani and Singhal 2011]. Um conjunto de *checkpoints* que corresponde a um estado consistente é chamado de *linha de recuperação*. O principal objetivo dos protocolos de *rollback-recovery* é restaurar o sistema a partir da mais recente linha de recuperação após uma falha.

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

aplicação. É uma abordagem totalmente transparente à aplicação e não há necessidade de alterações de código. A grande desvantagem é a falta de portabilidade [Egwutuoha et al. 2013].

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

5.2.2.1. *Rollback-Recovery* Baseado em *Checkpoints*

Os protocolos de *rollback-recovery* baseados em *checkpoints* podem ser divididos em três abordagens: coordenada, não coordenada e induzida pela comunicação (CIC). Quando o *checkpoint* é realizado independentemente em cada processo, sem uma coordenação global, é chamado de não coordenado. A vantagem da abordagem não coordenada está em cada processo criar o seu *checkpoint* quando lhe é mais conveniente. Entretanto, com essa abordagem, um estado global consistente pode nunca ser atingido. Nesse caso, os *checkpoints* realizados tornam-se inúteis e devem ser descartados [Elnozahy et al. 2002, Kshemkalyani and Singhal 2011].

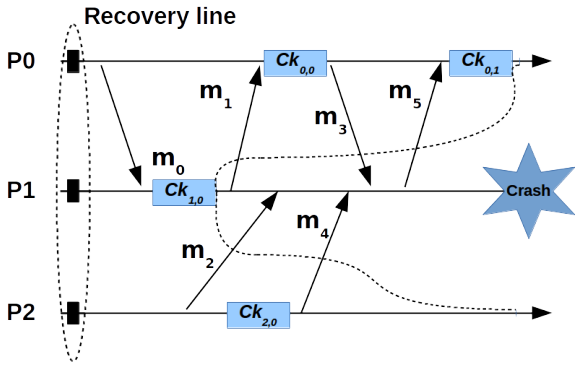

Figura 5.1. *Checkpoint* não coordenado e uma linha de recuperação.

A Figura 5.1 apresenta um cenário no qual o mais recente conjunto de *checkpoints* (isto é, $Ck_{0,1}$, $Ck_{1,0}$, e $Ck_{2,0}$) não resulta em uma linha de recuperação. Isso se deve ao fato de que a mensagem m_5 é recebida por P_0 mas não enviada por P_1 — nesse caso, m_5 é chamada de uma *mensagem órfã* e P_0 um *processo órfão*. P_0 é então obrigado a retroceder a um *checkpoint* anterior (isto é, $Ck_{0,0}$). O problema entretanto persiste, pois m_1 também foi recebida por P_0, mas não enviada por P_1. Desta forma, a única linha de recuperação corresponde ao estado inicial da aplicação. Esse fenômeno é conhecido como *efeito dominó*. O *checkpointing* não coordenado é suscetível ao efeito dominó.

O *checkpointing* coordenado evita o efeito dominó. Os processos se sincronizam para realizar os *checkpoints* e, consequentemente, criar um estado global consistente [Kshemkalyani and Singhal 2011]. Embora a abordagem coordenada seja relativamente fácil de implementar, a sua execução impõe uma sobrecarga considerável à aplicação, uma vez que os processos precisam se coordenar e salvar os seus estados simultaneamente no dispositivo de armazenamento. Além disso, mesmo que um único processo falhe, todos os processos precisam retroceder ao último *checkpoint*.

A abordagem denominada CIC (*communication-induced checkpointing*) força cada processo a realizar os *checkpoints* com base em informações inseridas nas mensagens recebidas dos outros processos. Os *checkpoints* são realizados de forma a manter a estado consistente em todo o sistema. A técnica reúne as vantagens das abordagens coordenada e não coordenada. No entanto, a abordagem relaxa a necessidade de coordenação global, o que a torna ineficiente na prática [Bouteiller et al. 2006]. Além disso, gera um grande número de *checkpoints*, resultando em sobrecarga no armazenamento e sobrecarga no canal de comunicação devido às informações inseridas nas mensagens da aplicação [Kshemkalyani and Singhal 2011, Egwutuoha et al. 2013].

O primeiro algoritmo a coordenar todos os *checkpoints* é apresentado por Chandy e Lamport [Chandy and Lamport 1985]. O algoritmo assume canais FIFO, isto é, todos os processos recebem as mensagens enviadas por um determinado processo na mesma ordem. Qualquer processo pode decidir iniciar um *checkpoint* e quando o faz envia uma mensagem especial chamada *marker* no seu canal de comunicação. Ao receber uma mensagem *marker* pela primeira vez um processo realiza o *checkpoint*. Após o *checkpoint*, todas a mensagens recebidas são armazenadas até uma nova mensagem *marker* ser recebida. Entre os trabalhos que fazem uso desse algoritmo há o CoCheck [Stellner 1996] e o LAM/MPI [Burns et al. , Sankaran et al. 2005], descritos na Seção 5.3.2.

5.2.2.2. *Rollback-Recovery* Baseado em Registro de Mensagens

Os protocolos de *rollback-recovery* baseados em registro de mensagens empregam tanto *checkpoints* quanto o registro de eventos não-determinísticos com o objetivo de evitar as desvantagens das abordagens coordenada e não coordenada. Um *evento* corresponde a um passo de comunicação ou um passo de computação de um processo. Um evento é determinístico quando a partir do estado atual existe somente um estado resultante possível para o evento. Se um evento pode resultar em estados diferentes, então é dito não-determinístico. A recepção de mensagens com uma identificação de emissor explícita é um evento determinístico e não requer o seu registro. Ao contrário, quando se aguarda uma mensagem de um emissor desconhecido então a recepção é dita não-determinística [Bouteiller et al. 2010].

Os protocolos de registro de mensagens assumem que todos os eventos não-determinísticos executados por um processo podem ser identificados e a informação necessária para reproduzir cada evento durante a recuperação pode ser codificada em tuplas chamadas de *determinantes* [Kshemkalyani and Singhal 2011]. A maioria dos protocolos de registro de mensagens assume que a recepção das mensagens é o único evento não-determinístico. O registro de mensagens evita o efeito dominó do *checkpointing* não coordenado salvando todas as mensagens recebidas. Por exemplo, na Figura 5.1, as mensagens m_2, m_4 e m_3 recebidas pelo processo P_1 devem ser salvas, assim como os determinantes que contém a ordem de recepção das mensagens. Durante a recuperação do processo P_1 somente P_1 retrocede. Assim, o estado de P_1 eventualmente será o mesmo ao anterior à falha, uma vez que as mensagens m_2, m_4 e m_3 são reaplicadas na mesma ordem.

Dependendo de como os determinantes são registrados, os protocolos de registro de mensagem podem ser classificados em pessimista, otimista ou causal [Elnozahy et al.

2002]. No registro pessimista, um processo primeiro armazena o determinante antes de entregar a mensagem. Apesar de simplificar a recuperação e a coleta de lixo, a abordagem pessimista gera uma sobrecarga durante a execução da aplicação: a aplicação precisa aguardar pelo armazenamento de cada determinante para então prosseguir. No registro de mensagens otimista, os processos armazenam os determinantes assincronamente, reduzindo assim a sobrecarga. Entretanto, a abordagem otimista pode gerar processos órfãos devido as falhas e, com isso, tornar a recuperação complexa. O registro causal busca combinar as vantagens do registro pessimista e otimista [Bouteiller et al. 2005]: ter baixa sobrecarga e evitar processos órfãos. Entretanto, os protocolos causais requerem que o determinante seja inserido em cada mensagem trocada pela aplicação até que esse seja confiavelmente armazenado.

O registro de mensagens geralmente faz uso da abordagem *sender-based* [Johnson and Zwaenepoel 1987]. Nessa abordagem, durante a operação normal cada mensagem enviada é salva no emissor. Dessa forma, o receptor da mensagem somente armazena o determinante correspondente, descrevendo o evento de entrega.

O *event logger* desempenha um papel crucial nos protocolos de registro de mensagens [Bouteiller et al. 2005]. O *event logger* recebe os determinantes dos processos da aplicação, armazena-os localmente, e notifica os processos da aplicação após armazená-los. Apesar do *event logger* exercer um grande impacto na eficiência dos protocolos de registro de mensagens, muitos protocolos o implementam como um componente centralizado e incapaz de tolerar falhas [Ropars and Morin 2009, Bouteiller et al. 2006, Ropars and Morin 2010]. Uma vez que *event logger* precisa notificar os processos da aplicação ao salvar um determinante, um *event logger* centralizado facilmente se torna em um gargalo conforme aumenta o número de processos. Além disso, a falha do *event logger* pode paralisar a aplicação ou levá-la a um estado inconsistente durante a recuperação. Em [Camargo et al. 2017] é apresentado um *event logger* distribuído e tolerante a falhas baseado em consenso para aplicações HPC.

Trabalhos que empregam o registro de mensagens como estratégia de tolerância a falhas são descritos na Seção 5.3.2. Descreve-se a seguir o padrão MPI e diversos trabalhos que abordam a tolerância a falhas em aplicações HPC baseadas em MPI.

5.3. Tolerância a Falhas em MPI

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

O MPI consiste de um conjunto de bibliotecas de funções padronizadas pelo MPI-Fórum [Forum 2017b]. Há rotinas para comunicação direta entre dois processos, isto é, para comunicação ponto-a-ponto, e rotinas para comunicação coletiva. Além disso, há primitivas para o gerenciamento e criação de processos, entrada e saída de dados em paralelo, gerenciamento de grupos e sincronização de processos e o estabelecimento de

topologias virtuais, onde os processos são organizados de forma virtual para realizar a comunicação. O MPI-Fórum é a entidade composta por pesquisadores, desenvolvedores e organizações responsáveis por desenvolver e manter a norma MPI, atualmente na sua versão 3.1 [MPI Forum 2015]. Entre as principais implementações do MPI, destacam-se a MPICH [Gropp et al. 1996] e a Open MPI [Gabriel et al. 2004].

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

O Algoritmo 5.1 apresenta um código MPI básico em linguagem C que lança um determinado número de processos em uma máquina local ou em múltiplas máquinas interligadas por uma rede (dependendo da listagem de máquinas presente no arquivo de configuração *hostfile*). A linha 9 inicia o ambiente MPI. Repare que a biblioteca mpi.h é incluída na linha 1. Após isso, na linha 10, a função `MPI_Comm_rank()` determina o *rank* dos processos dentro do comunicador. Nesse exemplo, o comunicador usado é o `MPI_COMM_WORLD`. Na linha 11, a função `MPI_Comm_size()` informa a quantidade de processos presentes no comunicador. A função `MPI_Get_processor_name()` obtém o nome da máquina onde o processo está em execução. A linha 17 finaliza o ambiente MPI.

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6      int rank, size, name_size;
7      char processor_name[MPI_MAX_PROCESSOR_NAME];
8
9      MPI_Init(&argc,&argv);
10     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11     MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13     MPI_Get_processor_name( processor_name, &name_size );
14
15     printf("Rank %d of %d running on host %s\n", rank, size, processor_name);
16
17     MPI_Finalize();
18     return 0;
19 }
```

Algoritmo 5.1. Exemplo de código MPI em linguagem C

Após compilar o código do Algoritmo 5.1 e gerar o executável `ex1`, a execução pode ser realizada da seguinte forma: $ mpiexec -np 4 ./ex1

De acordo com essa linha de código 4 processos serão lançados na máquina local.

Uma propriedade fundamental, porém ausente na especificação original MPI, é a tolerância a falhas [Gropp and Lusk 2004]. O padrão MPI assume que a infraestrutura subjacente é totalmente confiável [MPI Forum 2015]. Dessa forma, o padrão não define o comportamento preciso que as implementações MPI devem adotar perante falhas [Bland

et al. 2013, Gropp and Lusk 2004]. Basicamente, uma falha é tratada como um erro interno da aplicação como, por exemplo, a violação de um espaço de memória. Dessa forma, as falhas de processo ou de rede são repassadas à aplicação simplesmente como se fossem erros de chamadas de funções. Consequentemente, desloca-se a responsabilidade de detectar e de tratar as falhas para as implementações MPI. A norma define os manipuladores de erros (*error handlers*), que são associados ao comunicador MPI, para lidar com os erros do programa.

O manipulador de erros `MPI_ERRORS_ARE_FATAL` é associado por padrão ao comunicador `MPI_COMM_WORLD`. Esse manipulador define que a manifestação de um erro durante a chamada de uma função MPI leva os processos no comunicador a abortar a sua execução, encerrando assim toda a aplicação. Por outro lado, o manipulador `MPI_ERRORS_RETURN` retorna um código de erro que indica que a ocorrência de uma falha [MPI Forum 2015]. Mesmo que um código de erro seja retornado ao programa MPI, o padrão MPI não estabelece mecanismos para lidar com as falhas. Por essa razão, o suporte a tolerância a falhas pela norma MPI é considerado inadequado [Bland et al. 2012b, Bland et al. 2012c]. Além disso, as duas principais implementações MPI citadas anteriormente adotam o manipulador de erro `MPI_ERRORS_ARE_FATAL` por padrão e não dão suporte adequado ao manipulador de erros `MPI_ERRORS_RETURN`, impedindo a continuidade da aplicação no caso de falha.

Diversos trabalhos visam adicionar à implementação MPI rotinas específicas para lidar com as falhas. Entre esses estão o FT-MPI [Fagg and Dongarra 2000], FT/MPI [Batchu et al. 2004], Gropp e Lusk [Gropp and Lusk 2004] e recentemente o NR-MPI [Suo et al. 2013]. O MPI-Fórum também criou um grupo de trabalho específico para abordar a tolerância a falhas na norma MPI. Desse grupo surgiram as propostas chamadas de *Run-through Stabilization Proposal* (RTS) [Hursey et al. 2011] e de *User-Level Failure Mitigation* (ULFM) [Bland et al. 2012a, Bland et al. 2013], descritas a seguir.

5.3.1. RTS e ULFM

A falta de primitivas e de uma semântica de tolerância a falhas na norma MPI, que permitam às aplicações sobreviverem e se recuperarem de falhas de processos, aliada à frequente ocorrência de falhas nos sistemas HPC de grande escala, incentivaram o MPI-Fórum a criar um grupo de trabalho específico para o tema. O Grupo de Trabalho de Tolerância a Falhas (*Fault Tolerance Working Group* - FTWG) [Forum 2017a] foi estabelecido pelo MPI-Fórum, por volta do ano de 2009, com a responsabilidade de otimizar o padrão MPI para permitir o desenvolvimento de programas HPC portáveis, escaláveis e tolerantes a falhas [Hursey et al. 2011, Hursey and Graham 2011]. Os esforços do grupo de trabalho resultaram em duas propostas: a RTS (*Run-through Stabilization Proposal*) [Hursey et al. 2011, Forum 2017c] e a ULFM (*User-Level Failure Mitigation*) [Bland et al. 2012b, Bland et al. 2013, Forum b].

A RTS foi a primeira proposta do grupo de tolerância a falhas. Devido à complexidade presente na implementação das primitivas, a proposta RTS não prosseguiu o seu desenvolvimento. A especificação ULFM é o mais recente esforço do MPI-Fórum para padronizar a semântica de tolerância a falhas em MPI. A implementação da ULFM está em desenvolvimento como um subprojeto do projeto Open MPI [Gabriel et al. 2004, MPI-

Forum] e na implementação MPICH [Bland et al. 2015]. Existe a expectativa de que a adoção da ULFM pelo padrão MPI se dê a partir das próximas versões da norma MPI [Forum a].

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

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

Na especificação RTS, a implementação MPI deve fornecer um detector de falhas perfeito [Chandra and Toueg 1996]. Isso significa que em algum momento todo processo falho será conhecido por todos os outros processos. Na RTS os processos podem estar em um de três estados: *OK*, *FAILED* or *NULL*. Os processos *OK* são os que executam normalmente. Os processos com o estado *FAILED* foram detectados como falhos. Os processos marcados com *NULL* são processos falhos cujos *ranks* recebem a constante `MPI_PROC_NULL`.

A RTS trata as falhas de processos de acordo com o modelo de comunicação empregado [Hursey et al. 2011]. A comunicação ponto-a-ponto recebe um tratamento diferente da comunicação coletiva: a comunicação realizada por um par de processos raramente é afetada pela falha de outro processo; na comunicação coletiva, que envolve um grupo de processos, a falha de um único processo afeta os demais. Dessa forma, a RTS fornece duas abordagens para o tratamento de falhas: local e global. Enquanto o tratamento das local das falhas se destina à comunicação ponto-a-ponto, o tratamento global é destinado à comunicação coletiva.

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

Na especificação ULFM, a falha de um processo somente é detectada se esse processo participa ativamente de uma comunicação (ponto-a-ponto ou coletiva). Isto é, somente os processos que se comunicam diretamente com o processo falho o detectam. Ao contrário da RTS, a ULFM não cita o uso de um detector de falhas. A aplicação é notificada da falha durante a execução das operações de comunicação. A Figura 5.2 apresenta um exemplo com três processos (A, B e C) que realizam uma comunicação ponto-a-ponto.

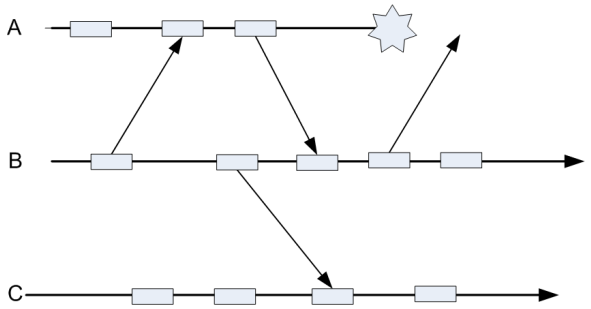

Figura 5.2. Detecção de falhas ULFM - Processo B detecta a falha em A.

Conforme apresenta a Figura 5.2, o processo B detecta a falha do processo A após enviar mensagem para A. Por sua vez, o processo C não identifica a falha em A. Essencialmente, a ocorrência de uma falha indica que a comunicação não pôde ser executada com sucesso. Por razões de desempenho, não há propagação automática sobre a ocorrência de falhas. Se durante uma operação coletiva um processo falhar, é possível que somente alguns processos identifiquem a falha. Ao todo, a ULFM disponibiliza ao usuário cinco funções para lidar com as situações de falhas. Entre essas, algumas permitem estabelecer uma visão consistente entre os processos. As primitivas da ULFM são descritas a seguir.

A operação de revogação, realizada pelo construtor `MPI_Comm_revoke`, é a mais crucial e complexa entre os construtores. Essa operação notifica todos os processos que o comunicador MPI a que pertencem está inválido. Dessa forma, evita a inconsistência entre os processos associados a um comunicador. O comunicador torna-se inválido e as comunicações futuras, ou as comunicações pendentes, são interrompidas e marcadas com um código de erro. A operação de revogação da ULFM conta com requisitos semelhantes à difusão confiável (*reliable broadcast*) [Guerraoui et al. 2011]. Nessa implementação, a ULFM usa um grafo binomial (*binomial graph*) onde o iniciador marca o comunicador como revogado e envia uma mensagem de revogação a outros $log(n)$ processos, considerando n processos. O processo, ao receber a mensagem de revogação, verifica se o comunicador foi marcado como inválido e, em caso contrário, atua como novo iniciador.

A primitiva `MPI_Comm_agree` é empregada para determinar uma visão consistente entre os processos. Essa função executa uma operação coletiva e faz com que os processos concordem com um valor booleano, mesmo se o comunicador foi revogado. Basicamente, o valor booleano pode significar o sucesso (0) ou a falha (1) na comunicação com um processo específico. Para fazer uso dessa primitiva o processo que identifica a falha deve antes revogar o comunicador. O construtor `MPI_Comm_shrink` permite criar um novo comunicador, eliminando todos os processos falhos de um comunicador inválido. Essa operação é coletiva e executa um algoritmo de consenso para assegurar

que todos os processos tenham a mesma visão no novo comunicador. Por fim, os construtores `MPI_Comm_failure_ack` e `MPI_Comm_failure_get_acked` são usados para informar quais processos dentro do comunicador se encontram falhos.

Por exemplo, na Figura 5.2, o processo B identifica a falha do processo A e então executa a função de revogação para que todos os processos corretos, no caso o processo C, tornem o seu comunicador inválido. Após isso, todos os processos executam a operação de acordo (`MPI_Comm_agree`) para garantir uma visão consistente sobre o estado do comunicador. Então, um novo comunicador válido, somente com processos corretos, é criado por meio da função `MPI_Comm_shrink`. Se houver a necessidade de identificar qual processo falhou (no caso o A), as primitivas `MPI_Comm_failure_ack` e `MPI_Comm_failure_get_acked` são empregadas.

```
1  #include <mpi.h>
2  #include <mpi-ext.h>
3  #include <stdio.h>
4  #include <signal.h>
5
6  int main(int argc, char *argv[])
7  {
8      int rank, size, name_size, result;
9      char processor_name[MPI_MAX_PROCESSOR_NAME];
10     char string_error[MPI_MAX_ERROR_STRING];
11
12     MPI_Init(&argc,&argv);
13     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14     MPI_Comm_size(MPI_COMM_WORLD, &size);
15
16     MPI_Comm_set_errhandler(MPI_COMM_WORLD,
17                             MPI_ERRORS_RETURN);
18
19     MPI_Get_processor_name( processor_name, &name_size );
20
21     if( rank == (size-1) )
22         raise(SIGKILL);
23
24     result = MPI_Barrier(MPI_COMM_WORLD);
25     MPI_Error_string(result, string_error, &name_size);
26
27     printf("Rank %d of %d running on host %s (error %s)\n", rank, size, processor_name,
       string_error);
28
29     MPI_Finalize();
30     return 0;
31 }
```

Algoritmo 5.2. Notificação de erro usando a ULFM

O Algoritmo 5.2 apresenta um exemplo de código que detecta e notifica a falha de um processo aos demais processos que não falharam. A linha 2 inclui a biblioteca ULFM. Neste exemplo, o processo com mais alto *rank* é morto, simulando uma falha. (linhas 21 e 22). Note que a linha 16 inclui a função `MPI_Comm_set_errhandler()`. Essa função associa o manipulador de erros `MPI_ERRORS_RETURN` ao comunicador `MPI_COMM_WORLD`. Através dessa função os erros detectados retornam um código. Na linha 24, se algum erro for detectado na execução da função coletiva `MPI_Barrier()`, esse erro será retornado para a variável `result`. A função `MPI_Error_string`, na linha 24, associa o código numérico a uma `string`. Ao executar o código acima com a linha de comando abaixo, a falha do processo origina um erro e a mensagem `MPI_ERR_PROC_FAILED` será exibida pelos processos que não falharam.

$ mpiexec -np 4 -am ft-enable-mpi ./ex2

```
1  #include <mpi.h>
2  #include <mpi-ext.h>
3  #include <stdio.h>
4  #include <signal.h>
5
6  static void verbose_errhandler(MPI_Comm* comm, int* err, ...) {
7      int rank, size, name_size;
8      char errstr[MPI_MAX_ERROR_STRING];
9
10     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11     MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13     MPI_Error_string( *err, errstr, &name_size );
14     printf("Rank %d / %d: recebeu uma notificação de erro %s\n",
15            rank, size, errstr);
16 }
17
18 int main(int argc, char *argv[])
19 {
20     int rank, size, name_size;
21     char processor_name[MPI_MAX_PROCESSOR_NAME];
22     MPI_Errhandler errh;
23
24     MPI_Init(&argc,&argv);
25     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
26     MPI_Comm_size(MPI_COMM_WORLD, &size);
27
28     MPI_Comm_create_errhandler(verbose_errhandler, &errh);
29     MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
30
31     MPI_Get_processor_name( processor_name, &name_size );
32
33     if( rank == (size-1) ) raise(SIGKILL);
34
35     MPI_Barrier(MPI_COMM_WORLD);
36
37     printf("Rank %d of %d running on host %s \n", rank, size, processor_name);
38
39     MPI_Finalize();
40     return 0;
41 }
```

Algoritmo 5.3. Definição de um manipulador de erros

O Algoritmo 5.3 modifica o Algoritmo 5.2 criando um manipulador de erros através da função `MPI_Comm_create_errhandler()`. Esse manipulador é posteriormente é associado ao comunicador (linha 29). A partir de então, toda a detecção de erro invoca a função `verbose_errhandler`.

A função `MPI_Comm_create_errhandler()` é modificada no Algoritmo 5.4 para identificar os processos que falharam. São empregadas as funções `failure_ack()` e `failure_get_acked()` para essa tarefa (linhas 18 e 19) juntamente com funções de manipulação de grupos em MPI. A função `failure_get_acked()` obtém o grupo de processos que tiveram suas falhas identificadas. A partir de então, através de uma comparação com o grupo original é possível identificar os processos que falharam (linha 30).

```
1
2  static void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...) {
3      MPI_Comm comm = *pcomm;
4      int err = *perr;
```

```
5       char errstr[MPI_MAX_ERROR_STRING];
6       int i, rank, size, nf, len, eclass;
7       MPI_Group group_c, group_f;
8       int *ranks_gc, *ranks_gf;
9
10      MPI_Error_class(err, &eclass);
11      if( MPIX_ERR_PROC_FAILED != eclass ) {
12          MPI_Abort(comm, err);
13      }
14
15      MPI_Comm_rank(comm, &rank);
16      MPI_Comm_size(comm, &size);
17
18      MPIX_Comm_failure_ack(comm);
19      MPIX_Comm_failure_get_acked(comm, &group_f);
20      MPI_Group_size(group_f, &nf);
21      MPI_Error_string(err, errstr, &len);
22      printf("Rank %d / %d: recebeu uma notificação de erro %s. "
23          "%d processo(s) falhou(aram): { ", rank, size, errstr, nf);
24
25      ranks_gf = (int*)malloc(nf * sizeof(int));
26      ranks_gc = (int*)malloc(nf * sizeof(int));
27      MPI_Comm_group(comm, &group_c);
28      for(i = 0; i < nf; i++)
29          ranks_gf[i] = i;
30      MPI_Group_translate_ranks(group_f, nf, ranks_gf,
31                                group_c, ranks_gc);
32      for(i = 0; i < nf; i++)
33          printf("%d ", ranks_gc[i]);
34      printf("}\n");
35  }
```

Algoritmo 5.4. Identificação dos processos falhos

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

5.3.2. *Rollback-Recovery* em MPI

Conforme descrito na Seção 5.2.2, o *Rollback-Recovery* é a principal técnica de tolerância empregado em aplicações HPC baseadas no padrão MPI. A seguir alguns trabalhos que fazem uso dessa técnica são descritos brevemente.

O ambiente CoCheck [Stellner 1996] é o primeiro esforço para incluir a tolerância a falhas em MPI. O ambiente faz uso da técnica de *checkpoint-restart* e de migração de processos. A sua implementação é em nível de usuário. Tanto o reinício da aplicação quanto a migração de processo são transparentes à aplicação. O ambiente executa acima da biblioteca MPI e o programador precisa associá-lo à aplicação. A partir de então, as primitivas do CoCheck são usadas para comunicação ao invés das primitivas originais. Há um processo especial para coordenar os *checkpoints*. O mesmo envia uma notificação para todos os processos envolvidos na execução da aplicação realizarem o *checkpoint*.

Para tanto, cada processo envia mensagens especiais, chamadas *ready messages*, nos canais para assegurar que não há mensagens em trânsito e assim garantir a consistência global. Cada processo independentemente mantém um *checkpoint* consistente. É possível também realizar um *checkpoint* de toda a aplicação. A principal desvantagem do CoCheck está na necessidade de sincronizar toda a aplicação para efetuar o *checkpoint*, o que pode levar a problemas de escalabilidade. Outra desvantagem do ambiente é implementar a sua própria versão do MPI, chamada tuMPI [Fagg and Dongarra 2004].

O LAM/MPI é uma implementação de referência do MPI [Burns et al.]. No trabalho de Sankaran et al. [Sankaran et al. 2005] o LAM/MPI é estendido para suportar o *checkpoint* coordenado. A abordagem integra o LAM/MPI com a implementação de *checkpoint* em nível de sistema denominada BLCR (*Berkeley Lab's linux Checkpoint/-Restart*) [Duell 2003] através de interfaces definidas para *checkpoint-restart*. O BLCR é uma implementação que suporta aplicações com múltiplas *threads* no ambiente Linux. O LAM/MPI implementa o *checkpoint* em nível de usuário. Há algum tempo os esforços de desenvolvimento do LAM/MPI e dos seus mecanismos de tolerância a falhas foram portados para o desenvolvimento do Open MPI. No entanto, o Open MPI atualmente não dá suporte a qualquer estratégia de *checkpointing*.

A aplicação da técnica de *rollback-recovery* puramente baseada em *checkpoints* para sistemas HPC de larga escala vem sendo colocada em cheque devido ao MTBF cada vez menor desses sistemas [Cappello et al. 2009, Egwutuoha et al. 2013]. Embora eficiente, o custo para armazenar e recuperar os *checkpoints* pode exceder o MTBF dos futuros sistemas HPC *exascale*. Mais tempo será gasto lidando com as falhas do que realizando a computação útil [Ropars et al. 2013, Cappello et al. 2009]. Por exemplo, o Blue Waters [of Illinois], um sistema HPC *petascale* da universidade de Illinois, apresenta um MTBF médio de 4,2 horas [Di Martino et al. 2014]. Por outro lado, como apontado em Tiwari et al. [Tiwari et al. 2014] uma aplicação de astrofísica possui um volume de dados de 160TB e pode levar 360 horas para finalizar sua execução. Realizar um *checkpoint* a cada 1 hora, por exemplo, causa um grande impacto no sistema. Aumentar o intervalo de *checkpoint* pode reduzir o impacto no sistema, porém aumenta a quantidade de trabalho perdido no caso de falha: o processamento realizado entre o último *checkpoint* e a falha.

Diversos trabalhos buscam melhorar o desempenho da técnica de *rollback-recovery* baseado em *checkpoints* para sistemas HPC de larga escala através de diferentes propostas, como o Multi-level Checkpointing [Moody et al. 2010], protocolos híbrido ou hierárquicos que combinam o *checkpoint* coordenado com o registro de mensagens [Riesen et al. 2012, Ropars et al. 2013] e versões otimizadas [Tiwari et al. 2014, Bouteiller et al. 2013]. A seguir apresentamos algumas dessas abordagens, incluindo a abordagem de registro de mensagens.

O Diskless Checkpoint [Plank et al. 1998, Chen et al. 2005] é uma técnica que elimina a sobrecarga imposta pelo armazenamento estável do *checkpoint* tradicional. Nessa técnica, o estado de uma aplicação distribuída de longa duração é persistido tanto em memória quando em disco local. Adicionalmente, codificações relacionadas a esses *checkpoints* são armazenadas em processos redundantes - que podem ou não estar envolvidos na computação. Quando uma falha ocorre, os processos que não falharam recuperam o seu último *checkpoint*. O estado dos processos falhos pode ser calculado a partir do *check-*

point dos processos que não falharam e das codificações relacionadas aos *checkpoints*.

Multi-level Checkpointing [Moody et al. 2010] é uma abordagem que emprega múltiplos tipos de *checkpoints*, com diferentes níveis de resiliência e custo, em uma única execução da aplicação. O nível inferior é o mais lento e o mais resiliente. Nesse nível o *checkpoint* é escrito em um sistema de arquivos paralelo - o qual pode suportar a falha de todo o sistema. Os níveis superiores, apesar de serem mais rápidos, são os menos resilientes: os *checkpoints* são salvos em armazenamento local, tais como a memória RAM, memória *flash*, disco local ou cópias redundantes entre os nodos do sistema. Os trabalhos descritos em [Bautista-Gomez et al. 2011] e [Di et al. 2014] propõem, respectivamente, otimizações à abordagem através de códigos de proteção de dados e estudos sobre o intervalo de *checkpoint* em execuções onde o número de processadores e/ou núcleos envolvidos na computação é variável.

Fenix [Gamell et al. 2014, Gamell et al. 2015] é um arcabouço que permite a recuperação transparente e em tempo de execução de aplicações MPI. O arcabouço faz uso da especificação ULFM para sobreviver as falhas e emprega a técnica de Diskless Checkpoint: os dados da aplicação são salvos na memória dos nodos vizinhos [Zheng et al. 2012]. Também são disponibilizadas primitivas para o desenvolvedor realizar o *checkpoint* dos dados essenciais da aplicação. Fenix adota uma abordagem de chamada de *checkpoints* implícitos, onde os *checkpoints* são salvos de forma não-coordenada, porém, considerando a posição onde os *checkpoints* são inseridos no código há a garantia de que estados globais consistentes são sempre gerados pela aplicação. A avaliação do Fenix foi realizada em uma aplicação MPI executando milhares de processos. O arcabouço não está disponível para uso.

O projeto MPICH-V [Bouteiller et al. 2006] apresenta três protocolos de registros de mensagens que trabalham em conjunto com o *checkpointing* não coordenado. Dois protocolos são pessimistas e um é causal. O MPICH-V1 é um protocolo pessimista projetado para ambientes heterogêneos e com alta volatilidade, tais como grades computacionais formadas por *desktops*. O MPICH-V1 faz uso de um componente remoto e confiável, chamado de *Channel Memory* (CM) responsável por armazenar o conteúdo das mensagens e a ordem de recepção das mensagens MPI. Cada processo primeiro envia a sua mensagem para o CM do receptor. Então, o receptor solicita a mensagem do seu próprio CM. Apesar de haver um CM para cada processo, eles não suportam falhas. O MPICH-V2 [Bouteiller et al. 2003] é um protocolo pessimista destinado a grandes *clusters*. O MPICH-V2 conta com a abordagem *sender-based*. Os processos se comunicam diretamente. Ao invés de usar os CMs, o MPICH-V2 emprega *event loggers* que são usados como armazenamento remoto confiável. Quando um processo recebe uma mensagem, envia o determinante da mensagem para o *event logger*.

Em Lemarinier et al. [Lemarinier et al. 2006] uma estratégia de *checkpointing* coordenada baseada no algoritmo de Chandy e Lamport é comparada com o protocolo MPICH-V2 usando o *checkpointing* não coordenado em diferentes frequências de falhas e volume de dados da aplicação. A principal conclusão é a de que o registro de mensagens se torna relevante para *clusters* de grande escala a partir de uma taxa de falhas de uma falha a cada hora para aplicações com grande volume de dados.

Em Bouteiller et al. [Bouteiller et al. 2005], os autores investigam os benefícios

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

No trabalho em [Bouteiller et al. 2009], Bouteiller et al. comparam experimentalmente um protocolo pessimista e otimista de registro de mensagens considerando o refinamento proposto em um trabalho anterior envolvendo os mesmos autores [Bouteiller et al. 2010] que distingue eventos determinísticos dos não-determinísticos em MPI. Esse refinamento é codificado em um protocolo chamado de `Vprotocol` na implementação Open MPI. Como consequência, o número de mensagens enviadas ao *event logger* diminui consideravelmente. No `Vprotocol` o *event logger* não é tolerante a falhas e é implementado como um processo especial disponível à aplicação em um grupo externo ao grupo MPI principal, ou seja, o `MPI_COMM_WORLD`. Atualmente, o `Vprotocol` não está disponível na biblioteca Open MPI.

Geralmente, os protocolos de registro de mensagens criam determinantes para todas as mensagens recebidas. No entanto, é possível reduzir o número total de determinantes armazenados distinguindo os eventos determinísticos dos não-determinísticos [Bouteiller et al. 2010]. Por exemplo, um evento não-determinístico ocorre em MPI quando o processo receptor usa uma marcação `MPI_ANY_SOURCE` na primitiva `MPI_Recv`. Conforme definido em Cappello et al. [Cappello et al. 2010], muitas aplicações MPI contêm somente eventos de comunicação determinísticos. Alguns protocolos de tolerância a falhas foram propostos para essa classe de aplicação [Guermouche et al. 2012, Lefray et al. 2013, Ropars et al. 2013]. Porém, importantes aplicações MPI são não-determinísticas. Além disso, os desenvolvedores geralmente incluem o não-determinismo na codificação para melhorar o desempenho da aplicação.

O trabalho de Ropars e Morin [Ropars and Morin 2009] propõe o O2P, um protocolo ativo de registro de mensagens otimista. Nesse protocolo o *event logger* é implementado como um processo MPI capaz de manipular comunicações assíncronas. O *event logger* é inicializado separadamente da aplicação MPI. Os processos da aplicação se conectam ao *event logger* ao iniciar o registro dos determinantes. O O2P assume um *event logger* confiável. Os experimentos com um grande número de processos e uma alta taxa de comunicação mostram que o *event logger* é um gargalo para o desempenho do sistema.

Um *event logger* distribuído para o protocolo O2P também é proposto por Ropars e Morin [Ropars and Morin 2010]. O *event logger* aproveita a arquitetura multi-core dos processadores para ser executado em paralelo com os processos da aplicação. Cada nodo executa um *event logger*. Por exemplo, um nodo que possua um processador com quatro núcleos de processamento pode ter três processos da aplicação e um *event logger*. Os determinantes são salvos na memória volátil do *event logger* e são replicados entre os *event loggers*. Há um parâmetro chamado `replicationdegree` que informa ao *event logger* original quantos processos devem receber a cópia do determinante. Por exemplo,

se o `replicationdegree` é dois, um processo envia seus determinantes para outros dois *event loggers*. Quando um *event logger* recebe duas respostas de confirmações o determinante é considerado estável. O autor ainda propõe um protocolo de disseminação epidêmica (*gossip*) para espalhar os determinantes estáveis para todos os *event loggers*. Apesar de esse protocolo oferecer uma forma distribuída de salvar os determinantes, a solução falha se uma única resposta de confirmação não for recebida pelo emissor. Isto é, a tolerância a falhas da solução não é garantida.

O trabalho proposto por Camargo et al. [Camargo et al. 2017] propõe e implementa um *event logger* distribuído e tolerante a falhas baseado em consenso para os protocolos de registro de mensagens. O protocolo se apoia no algoritmo de consenso Paxos [Lamport 2001] para replicar o *event logger*. Nesse trabalho, duas abordagens são propostas. A primeira abordagem é baseada no algoritmo Paxos tradicional e é chamada de Paxos Clássico. A segunda abordagem propõe um configuração onde cada processo da aplicação tem a sua própria instância de consenso e é chamada de Paxos Paralelo. Os resultados apresentados demonstram que a abordagem Paxos Paralelo tem desempenho superior a um *event logger* centralizado e é capaz de suportar um número configurável de falhas.

Um protocolo híbrido que combina o *checkpointing* coordenado e o registro de mensagem otimista é proposto por Riesen et al. [Riesen et al. 2012]. O protocolo faz uso de nodos adicionais que agem como *event loggers* ou nodos extras no caso da falha de um nodo. O *checkpointing* coordenado auxilia o registro de mensagens limitando o tamanho dos registros e evitando retornar à aplicação ao seu estado inicial no caso de um estado inconsistente se fazer presente. Por sua vez, o registro de mensagens evita reiniciar todos os processos no caso de falhas na maioria das vezes. O trabalho assume a presença de um serviço para detectar processos falhos e reiniciar processos em nodos extras. Se o *event logger* falha, a aplicação continua sua execução. Ao atingir um *checkpoint* global, o serviço usa um nodo extra para lançar um novo *event logger* e informar cada processo sobre o novo *event logger*. No entanto, se um nodo precisa recuperar determinantes que foram perdidos devido à falha do *event logger*, então a aplicação reinicia a partir do último *checkpoint* coordenado. O *event logger* não é distribuído. Apesar de o protocolo de registro de mensagens otimista diminuir a sobrecarga, processos órfãos podem ser criados. Outro protocolo híbrido é proposto em [Bouteiller et al. 2013]: *checkpoint* coordenado é usado dentro dos nodos, onde os processo são relacionados, e *checkpoint* não coordenado com registro de mensagens pessimista é empregado entre os nodos.

5.3.3. Replicação Máquina de Estado Aplicada para MPI

A replicação máquina de estados (*State-Machine Replication* - SMR) é uma das mais importantes técnicas de tolerância a falhas [Charron-Bost et al. 2010]. Nessa técnica, frequentemente empregada para fornecer serviços com alta disponibilidade, o estado de um processo é replicado de tal forma que, se um processo falhar, a sua réplica, ou cópia, mantém o serviço disponível. O serviço é definido por uma máquina de estados, que consiste de variáveis e de operações. Uma operação pode tanto ler o estado das variáveis quanto modificá-las. A execução das operações são determinísticas, isto é, se duas réplicas executam a mesma sequência de operações na mesma ordem, o mesmo estado deve ser produzido em ambas [Schneider 1990].

O trabalho de Ferreira et al. [Ferreira et al. 2011] avalia a viabilidade da técnica de replicação máquina de estados como principal mecanismo de tolerância a falhas para os sistemas HPC *exascale*. A justificativa dos autores para usar a técnica de replicação é a diminuição drástica do tempo médio de interrupção devido a uma falha (*Mean Time To Interrupt* - MTTI) e o fato de estudos apresentarem que os sistemas *exascale* podem levar mais do que 50% do seu tempo lendo e escrevendo *checkpoints*. O *checkpoint-restart* seria aplicado em algumas situações como, por exemplo, quando todas as réplicas falhassem. As aplicações MPI são o objeto de estudo do autor: as cópias redundantes dos processos MPI permitem que perante uma falha do processo original a aplicação continue a sua execução de forma transparente, sem a necessidade de *rollback-recovery*. Segundo o autor, a técnica de replicação também poderia ser usada para detectar um leque maior de falhas, potencialmente incluindo as falhas maliciosas (bizantinas).

Ferreira et al. argumenta que não são todos os processos que precisariam ser duplicados. Por exemplo, considerando o modelo mestre-escravo somente o processo mestre seria replicado. Para realizar a avaliação, o trabalho combina modelagem, análise empírica e simulação para estudar os custos e benefícios da replicação em comparação com *checkpoint-restart*. Para estudar a sobrecarga imposta pela replicação dos processos, a ferramenta rMPI é projetada e implementada. A rMPI é implementada acima das implementações MPI existentes e fornece redundância de computação transparentemente às aplicações MPI determinísticas. O autor conclui que a técnica de replicação máquinas de estados tem potencial para atender as demandas de HPC.

A Ferramenta rMPI adota o modelo de falhas *fail-stop*: um processo falha por parada e então a sua réplica assume. O trabalho de Fiala et al. [Fiala et al. 2012] propõe a ferramenta redMPI com o objetivo de detectar e corrigir erros do tipo computação incorreta através da replicação da computação. Basicamente, a ferramenta compara as tarefas executadas pela réplica principal com as executas pelas suas cópias.

O trabalho de Bougeret et al. [Bougeret et al. 2014] adota uma estratégia de replicação de grupo ao invés de replicação de processos proposto em Ferreira et al. [Ferreira et al. 2011]. A replicação de grupo consiste em executar múltiplas instâncias da aplicação concorrentemente. Ao contrário do trabalho de Ferreira et al., a estratégia pode ser utilizada em qualquer modelo de programação de sistemas HPC. Um estratégia de *checkpoint* também é usada pelos autores. Os resultados obtidos no trabalho demonstram que a replicação de grupo pode apresentar vantagens em sistemas HPC de grande escala.

5.3.4. Tolerância a Falhas Codificada no Algoritmo da Aplicação

A técnica de ABFT (*Algorithm-Based Fault Tolerance*) faz uso das propriedades do algoritmo da aplicação para recuperá-la de falhas durante a sua execução, como se ignorasse a existência de falhas [Davies et al. 2011, Du et al. 2012, Hursey and Graham 2011, Wang et al. 2011]. A técnica não é transparente à aplicação. Os requisitos mínimos para utilizar a técnica são a detecção, notificação e propagação de falhas, assim como o suporte do ambiente de execução, que deve ser resiliente. Dessa forma, um dos empecilhos para a ampla adoção da técnica em aplicações MPI é falta de primitivas e de uma semântica padronizada de tolerância a falhas. Conforme apresentado na Seção 5.3, a especificação MPI e suas implementações de referência não fornecem meios para detectar e sobreviver

as falhas de rede e de processos. A maioria dos trabalhos que aplicam a técnica ABFT em MPI usam as implementações FT-MPI, a RTS ou a ULFM (Seção 5.3.1).

A técnica foi originalmente proposta por Huang e Abraham para detectar e corrigir erros em algumas operações em matrizes causados por falhas transientes ou permanentes no hardware [Huang and Abraham 1984]. De acordo com os autores, para algumas operações em matrizes há uma relação entre o *checksum* de entrada e o *checksum* apresentado nos resultados finais. Com base nessa relação, a técnica é desenvolvida para detectar, localizar e corrigir certos erros de cálculo do processador nas operações de matrizes.

A técnica ABFT é adaptada para sistemas HPC por Chen e Dongarra para suportar falhas de acordo com o modelo *fail-stop* durante a execução de programas que envolvam operações em matrizes [Chen and Dongarra 2006, Chen and Dongarra 2008]. Assim como Huang e Abraham, Chen e Dongarra fazem uso da relação do *checksum*, mencionado acima. A técnica é aplicada sem a necessidade de qualquer mecanismos de *checkpoint-restart*. Um estado global consistente é mantido em memória por meio da relação do *checksum*. Então, perante uma falha, a computação pode ser recuperada. No entanto, os processos corretos precisam aguardar a recuperação para continuar a execução da aplicação. A implementação FT-MPI é usada pelos autores [Chen and Dongarra 2008].

O trabalho de Wang et al. [Wang et al. 2011] propõe uma estratégia, chamada de ABFT-*hot-replacement*, para evitar que os processos corretos tenham que parar e aguardar pela recuperação dos dados do processo falho. Quando as falhas ocorrem durante a execução da aplicação, o trabalho atualiza o processo falho com um processo redundante correspondente. O trabalho também se apoia na relação do *checksum* e também é aplicado em operação de matrizes envolvendo transformações lineares. A implementação MPICH é adaptada para lidar com as falhas em nível de aplicação. Um trabalho semelhante é o de Bosilca et al. [Bosilca et al. 2009] onde os nodos redundantes são usados juntamente com uma abordagem de *diskless checkpoint* (Seção 5.2.2.1). Em Davies et al. [Davies et al. 2011] a técnica é usada para fatoração ou decomposição LU em matrizes. Os trabalhos em [Chen and Wu 2015, Jia et al. 2013] adotam abordagens semelhantes.

Hursey e Graham [Hursey and Graham 2011] apresentam como as primitivas da especificação RTS podem ser empregadas para tornar uma aplicação MPI que se comunica em uma topologia de anel tolerante a falhas. O objetivo é apresentar as primitivas da RTS aos desenvolvedores de aplicações MPI interessados em aplicar a técnica ABFT. A preocupação do autor é apresentar um exemplo de como é possível desenvolver uma aplicação tolerante a falhas e não especificamente aplicar a técnica ABFT. A recuperação dos processos não é abordada no trabalho. Duas versões do código da aplicação de comunicação em anel são apresentadas: uma que não tolera falhas e outra mantém a comunicação mesmo perante falhas. Hursey e Graham discutem ainda questões como duplicação de mensagens, detecção das falhas, terminação e eleição de um novo líder.

O *Checkpoint-on-Failure* (CoF) [Bland et al. 2012c] propõe uma estratégia de *checkpoint-restart* em nível de aplicação para ser usada juntamente com a técnica ABFT. A estratégia se apoia na possibilidade de as aplicações MPI serem notificadas de falhas de processos através da constante `MPI_ERRORS_RETURN` e também na técnica ABFT para restaurar os dados dos processos falhos através de métodos matemáticos. No CoF não há um *checkpoint* periódico, o mesmo é acionado quando ocorre uma falha em um

processo, a partir de então: 1) os processos corretos não mais executam chamadas MPI e realizam o *checkpoint* do seu estado atual; 2) processos corretos finalizam sua execução; 3) inicia-se a execução de uma nova aplicação; 4) o *checkpoint* é recuperado nessa nova aplicação; 5) através da técnica ABFT os dados do processo falho que não puderam ser salvos são restaurados e; 6) um estado global consistente é obtido e a aplicação retoma a sua execução. O CoF possui a vantagem de não exigir *checkpoints* periódicos, porém está restrito a algoritmos que suportam a técnica ABFT. No trabalho o CoF usado em uma aplicação matemática de fatoração linear.

Outras técnicas de tolerância a falhas aplicadas a sistemas baseados em MPI incluem a migração de processos [Wang et al. 2012, Stellner 1996], a predição de falhas [Rajachandrasekar et al. 2012, Gainaru et al. 2013], a exploração do determinismo na comunicação dos sistemas HPC [Cappello et al. 2010], a detecção de falhas [Kharbas et al. 2012, Genaud et al. 2009, Bosilca et al. 2016] e a redundância como técnica para detectar e corrigir erros do tipo de computação incorreta [Fiala et al. 2012].

5.4. Conclusão

O MPI é o padrão de *facto* para o desenvolvimento de aplicações paralelas e distribuídas baseado no paradigma de troca de mensagens. Este minicurso apresentou algumas das técnicas para a construção de sistemas MPI tolerantes a falhas. Foram apresentadas uma visão geral do conceito de tolerância a falhas, as técnicas *rollback-recovery*, replicação máquina de estados e ABFT. Além disso, foram apresentadas as recentes propostas de padronização da semântica de tolerância a falhas no padrão MPI por meio das especificações RTS e ULFM.

As especificações RTS e ULFM oferecem um conjunto de primitivas para o desenvolvedor da aplicação adequar a sua aplicação à uma técnica de tolerância a falhas de sua preferência. A ULFM é a proposta vigente e, ao contrário da RTS, não exige um detector de falhas explícito. De fato, na ULFM a falha de um processo somente é detectada se esse processo está diretamente envolvido em uma comunicação. A partir de então é possível usar as primitivas da ULFM para recuperar o estado do comunicador MPI e continuar a execução.

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

Entre os trabalhos de *rollback-recovery*, estão o CoCheck, que foi a primeira implementação de *checkpoint-restart* e de migração de processos em MPI. Implementações como o Diskless *checkpoint* e o CoF buscam eliminar a necessidade de armazenamento estável através de codificações relacionadas ao *checkpoint*. O Multi-Level Checkpointing busca diminuir a sobrecarga do *checkpoint* coordenado realizando o *checkpointing* em diferentes níveis. Entre os trabalhos citados, ainda se destaca o Fenix. O Fenix emprega

checkpoints implícitos e a especificação ULFM para recuperar a aplicação em tempo de execução e de forma transparente. Importante notar que a maioria dos trabalhos que empregam a abordagem de registros de mensagens faz uso de um *event logger* centralizado e que não tolera falhas para armazenar os determinantes. Foi apresentado o primeiro *event logger* distribuído e tolerante a falhas que é baseado no algoritmo de consenso Paxos.

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

A técnica ABFT move a tolerância a falhas para o código da aplicação paralela. A técnica é altamente dependente da especificação MPI. Os trabalhos que defendem a inclusão de primitivas de tolerância a falhas na norma MPI usam como uma das justificativas a possibilidade de empregar a técnica ABFT. Apesar de eficiente, umas das desvantagens da técnica é a sua aplicação em um domínio específico. Grande parte dos trabalhos se apoia na verificação de *checksums* para detectar falhas.

Referências

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

[Aguilera et al. 1997] Aguilera, M. K., Chen, W., and Toueg, S. (1997). Heartbeat: A timeout-free failure detector for quiescent reliable communication. *Lecture Notes in Computer Science*, 1320:126–140.

[Avizienis et al. 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

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

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

[Bland et al. 2012a] Bland, W., Bosilca, G., Bouteiller, A., Herault, T., and Dongarra, J. (2012a). A proposal for user-level failure mitigation in the mpi-3 standard. Technical report, Department of Electrical Engineering and Computer Science, University of Tennessee.

[Bland et al. 2013] Bland, W., Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. (2013). Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of HPC Applications*, 27(3):244–254.

[Bland et al. 2012b] Bland, W., Bouteiller, A., Hérault, T., Hursey, J., Bosilca, G., and Dongarra, J. J. (2012b). An evaluation of user-level failure mitigation support in MPI. In *EuroMPI*, pages 193–203.

[Bland et al. 2012c] Bland, W., Du, P., Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. (2012c). A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI. In *Euro-Par*.

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

[Bosilca et al. 2016] Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Robert, Y., Sens, P., and Dongarra, J. (2016). Failure detection and propagation in hpc systems. In *SC*, pages 312–322.

[Bosilca et al. 2009] Bosilca, G., Delmas, R., Dongarra, J., and Langou, J. (2009). Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416.

[Bougeret et al. 2014] Bougeret, M., Casanova, H., Robert, Y., Vivien, F., and Zaidouni, D. (2014). Using group replication for resilience on exascale systems. *International Journal of High Performance Computing Applications*, 28(2):210–224.

[Bouteiller et al. 2010] Bouteiller, A., Bosilca, G., and Dongarra, J. (2010). Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211.

[Bouteiller et al. 2003] Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P., and Magniette, F. (2003). MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *SC*.

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

[Bouteiller et al. 2013] Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. J. (2013). Correlated set coordination in fault tolerant message logging protocols for many-core clusters. *Concurrency and Computation: Practice and Experience*, 25(4):572–585.

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

[Bouteiller et al. 2009] Bouteiller, A., Ropars, T., Bosilca, G., Morin, C., and Dongarra, J. (2009). Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure, recovery. In *Cluster*.

[Burns et al.] Burns, G., Daoud, R., and Vaigl, J. LAM: An open cluster environment for MPI.

[Camargo et al. 2017] Camargo, E. T., Duarte, E. P., and Pedone, F. (2017). *A Consensus-Based Fault-Tolerant Event Logger for High Performance Applications*, pages 415–427.

[Cappello et al. 2009] Cappello, F., Geist, A., Gropp, B., Kalé, L. V., Kramer, B., and Snir, M. (2009). Toward exascale resilience. *The International Journal of High Performance Computing Applications*, 23(4):374–388.

[Cappello et al. 2010] Cappello, F., Guermouche, A., and Snir, M. (2010). On communication determinism in parallel HPC applications. In *ICCCN*, pages 1–8.

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

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

[Charron-Bost et al. 2010] Charron-Bost, B., Pedone, F., and Schiper, A., editors (2010). *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer.

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

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

[Chen et al. 2005] Chen, Z., Fagg, G. E., Gabriel, E., Langou, J., Angskun, T., Bosilca, G., and Dongarra, J. (2005). Building fault survivable mpi programs with ft-mpi using diskless checkpointing. In *PPoPP*, pages 213–223.

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

[Davies et al. 2011] Davies, T., Karlsson, C., Liu, H., Ding, C., and Chen, Z. (2011). High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *ICS*, pages 162–171.

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

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

[Du et al. 2012] Du, P., Bouteiller, A., Bosilca, G., Herault, T., and Dongarra, J. (2012). Algorithm-based fault tolerance for dense matrix factorizations. In *PPoPP*.

[Duell 2003] Duell, J. (2003). The design and implementation of berkeley labâs linux checkpoint/restart. Technical report, Lawrence Berkeley National Laboratory.

[Egwutuoha et al. 2013] Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326.

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

[Elnozahy et al. 2002] Elnozahy, Alvisi, Wang, and Johnson (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surveys*, 34.

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

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

[Felber et al. 1999] Felber, P., Défago, X., Guerraoui, R., and Oser, P. (1999). Failure detectors as first class objects. In *DOA*, pages 132–141.

[Ferreira et al. 2011] Ferreira, K. B., Stearley, J., Laros, III, J. H., Oldfield, R., Pedretti, K. T., Brightwell, R., Riesen, R., Bridges, P. G., and Arnold, D. (2011). Evaluating the viability of process replication reliability for exascale systems. In *SC*, page 44.

[Fiala et al. 2012] Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., and Brightwell, R. (2012). Detection and correction of silent data corruption for large-scale high-performance computing. In *SC*.

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

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

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

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

[Forum 2017b] Forum, M. (2017b). Mpi forum website. `http://mpi-forum.org/`. Acessado em 26/01/2017.

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

[Gabriel et al. 2004] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

[Gainaru et al. 2013] Gainaru, A., Cappello, F., Snir, M., and Kramer, W. (2013). Failure prediction for HPC systems and applications: Current situation and open issues. *The International Journal of High Performance Computing Applications*, 27(3):273–282.

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

[Gamell et al. 2015] Gamell, M., Teranishi, K., Heroux, M. A., Mayo, J., Kolla, H., Chen, J., and Parashar, M. (2015). Local recovery and failure masking for stencil-based applications at extreme scales. In *SC*.

[Genaud et al. 2009] Genaud, S., Jeannot, E., and Rattanapoka, C. (2009). Fault-management in P2P-MPI. *International Journal of Parallel Programming*, 37(5):433–461.

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

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

[Guermouche et al. 2012] Guermouche, A., Ropars, T., Snir, M., and Cappello, F. (2012). HydEE: Failure containment without event logging for large scale send-deterministic MPI applicat. In *IPDPS*.

[Guerraoui et al. 2011] Guerraoui, R., Cachin, C., and Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer.

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

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

[Hursey et al. 2011] Hursey, J., Graham, R. L., Bronevetsky, G., Buntinas, D., Pritchard, H., and Solt, D. G. (2011). Run-through stabilization: An MPI proposal for process fault tolerance. In *EuroMPI*, volume 6960, pages 329–332.

[Jalote 1994] Jalote, P. (1994). *Fault tolerance in distributed systems*. Prentice Hall.

[Jia et al. 2013] Jia, Y., Bosilca, G., Luszczek, P., and Dongarra, J. J. (2013). Parallel reduction to hessenberg form with algorithm-based fault tolerance. In *SC*, pages 1–11.

[Johnson and Zwaenepoel 1987] Johnson, D. B. and Zwaenepoel, W. (1987). Sender-based message logging. In *FTCS*.

[Kharbas et al. 2012] Kharbas, K., Kim, D., Hoefler, T., and Mueller, F. (2012). Assessing HPC failure detectors for MPI jobs. In *PDP*, pages 81–88.

[Kshemkalyani and Singhal 2011] Kshemkalyani, A. D. and Singhal, M. (2011). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, Cambridge, UK.

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

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

[Lefray et al. 2013] Lefray, A., Ropars, T., and Schiper, A. (2013). Replication for send-deterministic MPI HPC applications. In *FTXS Workshop at HPDC*.

[Lemarinier et al. 2006] Lemarinier, P., Bouteiller, A., Krawezik, G., and Cappello, F. (2006). Coordinated checkpoint versus message log for fault tolerant MPI. *International Journal of High Performance Computing and Networking*, 2:146–155.

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

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

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

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

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

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

[Pacheco 1996] Pacheco, P. S. (1996). *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Plank et al. 1998] Plank, J. S., Li, K., and Puening, M. A. (1998). Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, PDS-9(10):972–986.

[Rajachandrasekar et al. 2012] Rajachandrasekar, R., Besseron, X., and Panda, D. K. (2012). Monitoring and predicting hardware failures in HPC clusters with FTB-IPMI. In *IPDPS Workshops*, pages 1136–1143.

[Renesse et al. 1998] Renesse, R. V., Minsky, Y., and Hayden, M. (1998). A gossip-style failure detection service. Technical report, Cornell University.

[Riesen et al. 2012] Riesen, R., Ferreira, K., Silva, D. D., Lemarinier, P., Arnold, D., and Bridges, P. G. (2012). Alleviating scalability issues of checkpointing protocols. In *SC*.

[Ropars et al. 2013] Ropars, T., Martsinkevich, T. V., Guermouche, A., Schiper, A., and Cappello, F. (2013). SPBC: leveraging the characteristics of MPI HPC applications for scalable checkpointing. In *SC*, page 8.

[Ropars and Morin 2009] Ropars, T. and Morin, C. (2009). Active optimistic message logging for reliable execution of MPI applications. In *Euro-Par*.

[Ropars and Morin 2010] Ropars, T. and Morin, C. (2010). Improving message logging protocols scalability through distributed event logging. In *Euro-Par*.

[Sankaran et al. 2005] Sankaran, S., Squyres, J. M., Barrett, B., Sahay, V., Lumsdaine, A., Duell, J., Hargrove, P., and Roman, E. (2005). The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *The International Journal of High Performance Computing Applications*, (4):479–493.

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

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

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

[Suo et al. 2013] Suo, G., Lu, Y., Liao, X., Xie, M., and Cao, H. (2013). Nr-mpi: A non-stop and fault resilient mpi. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 190–199.

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

[Wang et al. 2012] Wang, C., Mueller, F., Engelmann, C., and Scott, S. L. (2012). Proactive process-level live migration and back migration in HPC environments. *Journal of Parallel and Distributed Computing*, 72(2):254–267.

[Wang et al. 2011] Wang, R., Yao, E., Chen, M., Tan, G., Balaji, P., and Buntinas, D. (2011). Building algorithmically nonstop fault tolerant MPI programs. In *HiPC*, pages 1–9.

[Zheng et al. 2012] Zheng, G., Ni, X., and KalÃ©, L. V. (2012). A scalable double in-memory checkpoint and restart scheme towards exascale. In *DSN Workshop 2012*.