UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

CACIANO DOS SANTOS MACHADO

MPI sobre MOM para suportar log de mensagens pessimista remoto

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Philippe O. A. Navaux
Orientador

Porto Alegre, agosto de 2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Caciano dos Santos Machado,

MPI sobre MOM para suportar log de mensagens pessimista remoto /

Caciano dos Santos Machado. – Porto Alegre: PPGC da UFRGS, 2010.

88 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. Orientador: Philippe O. A. Navaux.

I. Navaux, Philippe O. A.. II. Título.

AGRADECIMENTOS

- Aos colegas do CPD pelo companheirismo e pelas contribuições no meu crescimento profissional e pessoal.
- Aos colegas do GPPD pelas conversas e discussões nas minhas visitas esporádicas aos laboratórios.
- À UFRGS e ao Instituto de Informática pelo ensino de qualidade e infraestrutura exemplar que permitiram a conclusão deste trabalho.
- A todos os familiares e amigos que souberam compreender minha ausência durante fins de semana e feriados que foram dedicados ao Mestrado.

“I know how to make 4 horses pull a cart.
I don’t know how to make 1024 chickens do it.”
— Enrico Clementi

“What happens if the mean-time to failure for nodes
on the Tflops machine is shorter than the boot time?”
— Courtenay Vaughan

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AMQP	Advanced Message Queueing Protocol
API	Application Programming Interface
AMPI	Adaptive MPI
BTL	Byte Transfer Layer
BLCR	Berkeley Lab Checkpoint Restart
DARPA	Defense Advanced Research Projects Agency
GPL	General Public Licence
IPC	Inter-Process Communication
MCA	Modular Component Architecture
MOM	Message-Oriented Middleware
MPI	Message Passing Interface
MPICH	MPI Chameleon
MTBF	Mean Time Between Failures
MTL	Matching Transport Layer
NASA	National Aeronautics and Space Administration
NAS	National Aerodynamic Simulation
NPB	NAS Parallel Benchmarks
OPAL	Open Portability Abstraction Layer
ORTE	Open RunTime Environment
PWD	Piecewise Deterministic Assumption

RESUMO

O aumento crescente no número de processadores das arquiteturas paralelas que estão no topo dos rankings de desempenho, apesar de permitir uma maior capacidade de processamento, também traz consigo um aumento na taxa de falhas diretamente proporcional ao número de processadores. Atualmente, as técnicas de tolerância a falhas com recuperação retroativa são as mais empregadas em aplicações MPI, principalmente a técnica de *checkpoint* coordenado. No entanto, previsões afirmam que essa última técnica será inadequada para as arquiteturas emergentes. Em contrapartida, as técnicas de log de mensagens possuem características que as tornam mais apropriadas no novo cenário que se estabelece. O presente trabalho consiste em uma proposta de log de mensagens pessimista remoto com *checkpoint* não-coordenado e a avaliação de desempenho da comunicação MPI sobre *Publish/Subscriber* no qual se baseia o log de mensagens. O trabalho compreende: um estudo das técnicas de tolerância a falhas mais empregadas em ambientes de alto desempenho e a motivação para a escolha dessa variante de log de mensagens; a proposta de log de mensagens; uma implementação de comunicação Open MPI sobre OpenAMQ e sua respectiva avaliação de desempenho com comunicação tradicional TCP/IP e com o log de mensagens pessimista local da distribuição do Open MPI. Os *benchmarks* utilizados foram o NetPIPE, o NAS Parallel Benchmarks e a aplicação Virginia Hydrodynamics (VH-1).

Palavras-chave: Processamento de alto desempenho, computação baseada em clusters, tolerância a falhas, log de mensagens pessimista, message-oriented middleware.

MPI over MOM to support remote pessimistic message logging

ABSTRACT

The growing number of processors in parallel architectures at the top of performance rankings allows a higher processing capacity. However, it also brings an increase in the fault rate which is directly proportional to the number of processors. Nowadays, coordinated checkpoint is the most widely used rollback technique for system recovery in the occurrence of faults in MPI applications. Nevertheless, projections point that this technique will be inappropriate for the emerging architectures. On the other hand, message logging seems to be more appropriate to this new scenario. This work consists in a proposal of pessimistic message logging (remote based) with non-coordinated checkpoint and the performance evaluation of an MPI communication mechanism that works over Publish/Subscriber channels in which the proposed message logging is based. The work is organized as following: an study of fault tolerant techniques used in HPC and the motivation for choosing this variant of message logging; a message logging proposal; an implementation of Open MPI communication over OpenAMQ; performance evaluation and comparision with the tradicional TCP/IP communication and a pessimistic message logging (sender based) from Open MPI distribution. The benchmark set is composed of NetPIPE, NAS Parallel Benchmarks and Virginia Hydrodynamics (VH-1).

Palavras-chave: High performance computing, cluster based computing, fault tolerance, pessimistic message logging, message-oriented middleware.

LISTA DE FIGURAS

1 INTRODUÇÃO

A evolução das arquiteturas paralelas de computadores tem recentemente criado novas tendências e desafios tanto para desenvolvedores de aplicações e ambientes quanto para os usuários finais. Computadores de alto desempenho monolíticos têm sido substituídos continuamente por clusters de computadores devido à sua relação custo/desempenho mais atrativa. Clusters compostos centenas de milhares de processadores estão disponíveis nos dias de hoje e sistemas com milhões são esperados para os próximos anos (DONGARRA et al., 1997) (KOGGE et al., 2008) (GEIST; LUCAS, 2009). Esses recursos dos clusters e outros sistemas muitas vezes são integrados para formar grades computacionais (FOSTER; KESSELMAN, 2003). Embora um grande aumento do número de processadores desses sistemas permita alcançar patamares de desempenho muito superiores, esse aumento também acompanha um problema relacionado à taxa de falhas do sistema. Além disso, o próprio aumento do tempo de execução que as aplicações paralelas vêm apresentando nesses ambientes também acaba aumentando a probabilidade de ocorrência de falhas durante suas execuções.

O MPI (MPI FORUM, 1994) (GROPP; THAKUR, 1999) é o *middleware* mais amplamente utilizado para aplicações paralelas científicas de alto desempenho em clusters. A sua especificação abrange essencialmente as primitivas de comunicação, além de funcionalidades adicionais. Apesar disso, não existem extensões para suporte a tolerância a falhas. A inexistência de padronização somada com a necessidade cada vez mais presente de tolerância a falhas levou ao aparecimento de várias implementações distintas com esse tipo de suporte.

Entre as técnicas de tolerância a falhas do MPI, as que se sobressaem em número de implementações são as de recuperação retroativa (ELNOZAHY et al., 2002), mais especificamente as de *checkpoint* coordenado. Nas técnicas de recuperação retroativa a computação da aplicação retrocede para um ponto anterior à falha sempre que ela ocorre. No *checkpoint* coordenado o estado de todos os processos é guardado periodicamente em mídia confiável para posterior restauração simultânea em caso de falha. O *checkpoint* coordenado funciona de maneira que o conjunto de estados armazenados forma um estado global consistente que pode ser restaurado sem risco de inconsistências na troca de mensagens entre os processos da aplicação.

A restauração do estado global consistente exige a restauração de todos os processos da aplicação, o que implica em um desperdício de processamento diretamente proporcional ao número de processos restaurados. Sendo assim, em arquiteturas de exaescala [1], previstas para 2015 a 2020, e que espera-se que tenham um número de processadores na

[1] São denominados sistemas de exaescala os que apresentam desempenho superior a 1 exaflops, ou seja 10^{18} operações em ponto flutuante por segundo

ordem de milhões, esse desperdício se tornaria bastante acentuado. Algumas projeções afirmam que para a exaescala o tempo necessário para a realização dos *checkpoints* ultrapassará o próprio MTBF (*Mean Time Between Failure*) dos sistemas paralelos, o que, consequentemente, impossibilitaria a utilização dessa técnica. Nesse cenário, a aplicação não teria como progredir com sua computação pois as operações de *checkpoint* seriam quase sempre interrompidas.

As técnicas de *checkpoint* coordenado com log de mensagens (ALVISI; MARZULLO, 1998) utilizadas por algumas implementações de MPI não sofrem esses efeitos de forma tão drástica quando a taxa de falhas aumenta (LEMARINIER et al., 2004). Normalmente, nessas técnicas apenas os processos que falham precisam ser restaurados, juntamente com suas mensagens recebidas que são guardadas durante a execução normal. Essa característica é muito vantajosa no momento da restauração, conquanto apresente um elevado sobrecusto na comunicação, durante a execução normal da aplicação. Essa técnica possui algumas variações na forma como o log é implementado.

As variações mais interessantes para arquiteturas de exaescala são as de log causal e a de log pessimista. Nelas, é garantido que no caso de falha de apenas um processo, somente o processo que falhou necessita ser restaurado, juntamente com o seu log de mensagens. O log de mensagens pessimista remoto é uma variação que armazena os logs das mensagens em meio de armazenamento confiável remoto e exige que apenas os processos que falham sejam restaurados, mesmo no caso de múltiplas falhas. Considerando ambientes nos quais a falha de processos é regra e não a exceção, como é o esperado para as arquiteturas de exaescala, essa propriedade se torna bastante interessante, pois evita a necessidade de restaurar processos além dos que falharam. Apesar disso, o log de mensagens pessimista remoto possui um sobrecusto maior na comunicação normal da aplicação.

Através de comunicação *Publish/Subscriber* (EUGSTER et al., 2003) com MOM (*Message-Oriented Middleware*) (BANAVAR et al., 1999) sobre MPI é possível realizar o log de mensagens remoto e ainda se beneficiar de alguns recursos do MOM. Um desses recursos é a distribuição da carga em vários servidores de *brokers* que ficariam responsáveis tanto pelo log quanto pelo próprio encaminhamento das mensagens entre si. Além disso, a distribuição desses servidores de *broker* pode ser feita de forma que atendam domínios administrativos diferentes e otimizem a comunicação entre os sites. Existe a possibilidade de otimizar determinadas primitivas de comunicação coletiva através da utilização de *Publish/Subscriber* conforme será apresentado.

1.1 Objetivos

As tendências na taxa de falhas e número de processadores das arquiteturas paralelas de alto desempenho emergentes são os fatores motivadores desse trabalho. Nesse contexto, as técnicas de log de mensagens apresentam melhor escalabilidade quando comparadas com as técnicas de *checkpoint* coordenado tradicionalmente utilizadas. O objetivo desse trabalho é avaliar o sobrecusto envolvido na comunicação de aplicações com a técnica de log de mensagens pessimista remoto através de uma implementação de MPI com comunicação sobre canais *Publish/Subscriber*.

Para isso partiu-se de um estudo dos diferentes *middlewares* de comunicação (SANTOS MACHADO, 2007), em especial os MOM. Foi realizado um levantamento dos problemas recentes e emergentes relacionados a tolerância a falhas em arquiteturas paralelas, e as soluções existentes para MPI. Desses estudos originou-se a proposta de log de mensagens e a implementação da comunicação Open MPI (GABRIEL et al., 2004) so-

bre OpenAMQ (iMatix Corporation, 2010). Esse último é um *middleware* de código aberto que implementa a especificação de AMQP (*Advanced Message Queueing Protocol*) (KRAMER, 2009) enquanto aquele primeiro é uma implementação bastante conhecida do MPI. Para as medidas de desempenho da implementação foram utilizados os *benchmarks* NPB (BAILEY et al., 1991) e NetPIPE (SNELL; GUSTAFSON, 1996), e a aplicação real VH-1 (VH-1, 2010). Os resultados são execuções da implementação de Open MPI sobre Open AMQ apresentada neste trabalho e duas outras implementações do Open MPI que realizam comunicação sobre TCP/IP. Uma realiza comunicação TCP/IP tradicional e outra, além disso, destina-se ao suporte de log de mensagens pessimista local, que é a implementação de tolerância a falhas de MPI que mais se assemelha com a da proposta do presente trabalho e que ainda possui suporte dos desenvolvedores.

1.2 Organização do Trabalho

Este trabalho está organizado em 6 capítulos, incluindo esta introdução. O Capítulo 2 apresenta o estado da arte em tolerância a falhas de sistemas paralelos, em especial de implementações do MPI. O Capítulo 3 apresenta a própria proposta de log de mensagens aqui comentada. O Capítulo 4 descreve a implementação de comunicação do MPI através de MOM. No Capítulo 5, estão os resultados de desempenho obtidos através de *benchmarks* e respectivas avaliações. Fechando, segue o Capítulo 6, com considerações finais sobre o trabalho realizado e a sua continuidade.

2 ESTADO DA ARTE

2.1 Tolerância a Falhas em Arquiteturas Paralelas

Conforme o número de componentes das arquiteturas paralelas de alto desempenho cresce, também aumenta a taxa de falhas da arquitetura como um todo. Atualmente, os clusters que estão no topo dos rankings de desempenho (DONGARRA et al., 1997) possuem um número de processadores na ordem de centenas de milhares. Esse número expressivo de processadores, apesar de permitir uma maior capacidade de processamento, também implica em um aumento da taxa de falhas do cluster. Adicionalmente, as aplicações vêm aumentando seu tempo de execução, o que aumenta a probabilidade de que uma falha ocorra na aplicação.

Tendo em vista esses fatos, as técnicas de tolerância a falhas são necessárias para um aproveitamento apropriado das arquiteturas paralelas de alto desempenho. A seguir serão mostrados os modelos de falhas que descrevem o comportamento dos sistemas na presença de falhas. Depois serão apresentadas as principais técnicas de tolerância a falhas das duas classes fundamentais que podem ocorrer em sistemas de clusters: falhas de componentes centralizados e falhas da aplicação. Posteriormente, serão apresentados alguns desafios de tolerância a falhas em aplicações de exaescala seguindo com um resumo das implementações de MPI que visam tratar esses problemas. Por fim, conclui-se o capítulo com algumas considerações finais.

2.1.1 Modelos de Falhas em Sistemas Distribuídos

Existem várias maneiras de uma aplicação distribuída falhar. Essas falhas podem ser categorizadas em modelos abstratos que descrevem como o sistema se comportará na presença de falhas. O modelo de Cristian (CRISTIAN, 1985), ilustrado na Figura 2.1, classifica as falhas em falhas por colapso, omissão, temporização, resposta e arbitrárias. O modelo de Schneider (SCHNEIDER, 1986) estende esse modelo adicionando a classe *fail-stop*, que é um subconjunto da classe de falhas por colapso, e especializando as falhas de omissão em: falhas de omissão de envio e falhas de omissão de recepção.

A classe de falhas *fail-stop* é a mais específica desses modelos e comporta as falhas que são detectadas antes que o agente falho possa causar alterações no restante do sistema. Por exemplo, nesse modelo, um processador que apresente uma falha na execução de alguma operação deixará de funcionar em vez de fornecer resultados incorretos. A classe de falhas arbitrárias, também chamadas de Bizantinas, são as mais abrangentes. Nessa classe o agente falho pode apresentar um comportamento fora da sua especificação, totalmente imprevisível e podendo comprometer o restante do sistema sem que esse perceba a falha. Naturalmente, um sistema que assume o tratamento somente da classe *fail-stop* é bem mais simples de ser implementado do que um que trate também as falhas arbitrárias.

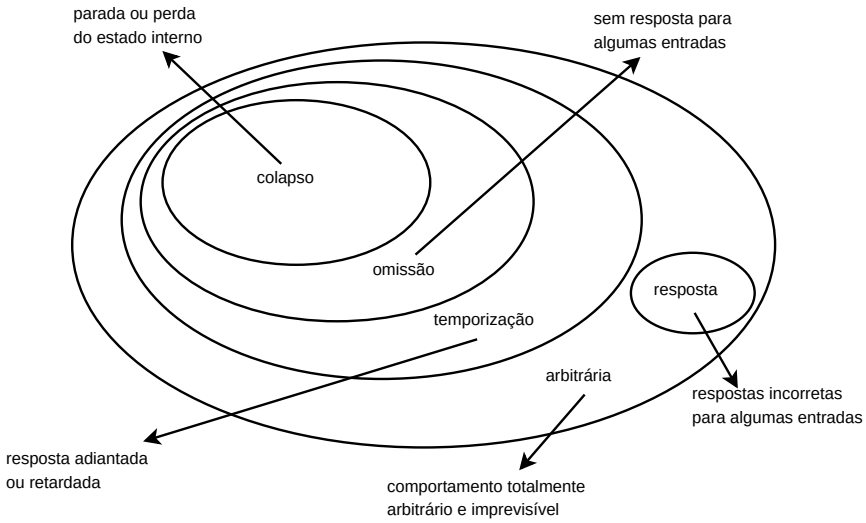

Figura 2.1: Modelos de Falhas em Sistemas Distribuídos

As classes mais específicas (colapso, omissão, temporização e resposta) apresentam comportamentos mais restritos e previsíveis que a classe de falhas arbitrárias. A seguir são apresentadas diversas técnicas de tolerância a falhas empregadas em sistemas de alto desempenho. Basicamente, esses sistemas assumem que haverá apenas falhas da classe *fail-stop*.

2.1.2 Tolerância a Falhas de Componentes Centralizados

Os sistemas de clusters dependem de elementos centralizados para operar. Nodos de gerência são utilizados para escalonamento de tarefas e monitoramento. Nodos de armazenamento oferecem acesso a discos de grande capacidade e altas taxas de transferência. Nodos de *front-end* permitem que os usuários interajam com o sistema sem interferir no tempo de processamento dos nodos de computação. Devido à importância crítica desses componentes, e por eles representarem um pequena fração do sistema que eles compõem, é viável e recomendado dedicar uma atenção extra para esses recursos.

A forma mais comum de suporte a tolerância a falhas para componentes centralizados é a replicação da funcionalidade que pode ser feita de duas formas: replicação ativa e replicação passiva. Na replicação ativa, um nodo secundário recebe uma cópia de todas as entradas do nodo primário e mantêm uma cópia idêntica do estado do sistema em execução. Além disso, o nodo secundário monitora o nodo primário para detectar comportamentos incorretos. Se o nodo secundário detectar uma falha ele assume o comando da funcionalidade crítica do sistema (GOLDBERG et al., 2001) (KIM, 1998). Esse modelo suporta apenas falhas do modelo *fail-stop*.

Uma versão aprimorada da replicação ativa utiliza múltiplas réplicas do nodo primário em vez de uma (REITER, 1995) (LI, 2006). Quando o sistema gera um resultado, todas as cópias comparam seus valores para votar qual é o resultado correto. A votação é feita através de um algoritmo Bizantino para suportar as falhas desse modelo. Quaisquer nodos que tenham gerado resultados incorretos um número de vezes maior que um limite estabelecido são marcados como falhos e ignorados nas votações subsequentes.

Na replicação passiva, uma máquina secundária aguarda a falha do nodo primário sem manter o estado do sistema. Tipicamente, essa máquina permanece inativa, porém com

todo software necessário para substituir o nodo primário quando necessário. Podem acontecer interrupções do serviço durante a substituição do nodo primário pelo secundário dependendo do estado da réplica. A não ser que sejam empregadas técnicas de recuperação do estado da aplicação, essa técnica não é adequada para componentes que mantém estruturas de dados complexas difíceis de ser recuperadas. No entanto, se o estado da aplicação que precisa ser recuperado for simples ou inexistente então essa técnica é bastante adequada. Um exemplo de sistema que pode utilizar replicação passiva é o monitoramento de subsistemas do cluster (LEANGSUKSUN et al., 2005).

Também é comum replicar componentes internos das máquinas em vez das máquinas inteiras. Por exemplo, uma máquina cuja operação no sistema é crítica pode ter processadores, bancos de memória, placas-mãe, discos e outros componentes que sejam propensos a falha replicados. Monitorando o componente primário é possível detectar sua falha e prontamente substituir pelo componente reserva sem interrupções. Assim que a falha é corrigida no componente falho, o administrador pode colocar o sistema em operação normal (COMPUTINA et al., 2000).

O monitoramento das réplicas normalmente é implementado utilizando um dos três seguintes mecanismos: *heartbeat* (LEANGSUKSUN et al., 2005) (BAGCHI et al., 1999), consenso Bizantino (REITER, 1995) (LAMPORT; PEASE, 1982) (GOLDBERG et al., 2001) ou pelo autodiagnóstico periódico de consistência. No monitoramento por *heartbeat* um processo recebe mensagens periódicas dos componentes monitorados. Essas mensagens indicam que o componente funciona de forma correta o suficiente para conseguir enviá-las. Se o processo monitor deixa de recebê-las de um determinado componente além de um limite de tempo estabelecido o nodo é considerado falho. Esse tipo de monitoramento permite a detecção de falhas no modelo *fail-stop* e não abrange falhas Bizantinas. Uma variação do *heartbeat* utiliza protocolos epidêmicos ou hierárquicos (COMPUTINA et al., 2000) para a propagação das mensagens no sistema. Essas variantes permitem que o *heartbeat* escale com um número maior de nodos.

No consenso Bizantino as réplicas votam por um resultado ou ação a ser tomada com base nas entradas observadas. Quando existe um desacordo entre alguns nodos então os nodos que tem minoria na votação são considerados falhos. Como o nome sugere esse tipo de monitoramento consegue detectar falhas Bizantinas nos componentes. No entanto, o custo de comunicação desse tipo de monitoramento é muito maior do que o monitoramento por *heartbeat* já que no pior caso do algoritmo é necessário que todos os nodos se comuniquem entre si. Apesar disso, para um pequeno número de réplicas, esse algoritmo ainda é aceitável.

Outra forma de monitoramento pode ser realizada através de um autodiagnóstico periódico de consistência em cada nodo. Nessa técnica, cada nodo realiza uma checagem para verificar se seus componentes internos estão operando corretamente. Além disso, os outros nodos podem compartilhar informações sobre os estados dos componentes centralizados. Os próprios nodos de computação podem detectar mensagens mal formadas ou perdidas do componente em questão e disparar um diagnóstico de consistência.

2.1.3 Tolerância a Falhas de Aplicações Paralelas com Troca de Mensagens

O valor real dos clusters de computadores não está nos componentes centralizados de gerência e monitoramento, mas sim na execução das aplicações de grande escala. Sendo assim, mesmo que os nodos de gerenciamento estejam cobertos com técnicas de tolerância a falhas, os nodos de computação e as aplicações que neles executam também precisam ser protegidos. De outra forma, os esforços em proteção dos nodos de gerenciamento serão

pouco significativos. As mesmas técnicas empregadas para os componentes centralizados são muito caras para serem implementadas em nodos de execução das aplicações, visto que esses últimos são muito mais numerosos.

A forma predominantemente empregada para recuperação de falhas em aplicações de alto desempenho é a recuperação retroativa (*rollback recovery*) (ELNOZAHY et al., 2002) com *checkpoint*. Essa técnica abrange aplicações trabalhando sobre plataformas de memória distribuída com troca de mensagens e consiste na criação arbitrária de cópias do estado dos processos da aplicação em mídia confiável para posterior restauração em caso de falha. O estado do processo é formado pelos valores de suas variáveis e registradores em um determinado momento da sua execução. O modelo de falhas que essas técnicas de recuperação retroativa assumem é o *fail-stop*.

Para explicar as técnicas de recuperação retroativa cada processo da aplicação será modelado como uma sequência de intervalos de estados, cada qual iniciada com um evento não-determinístico. Esse evento pode ser tanto uma entrada de dados do usuário como um recebimento de mensagem de outro processo. De agora em diante, serão considerados como eventos não-determinísticos apenas as mensagens, sem perda de generalidade dos conceitos das técnicas apresentadas.

Para recuperar um processo falho é necessário determinar qual foi o seu último estado global consistente. Um conjunto de estado de processos faz parte de um estado global consistente quando não existem processos órfãos. Ou seja, não existe nenhum processo cujo estado atual possua uma mensagem entregue sem que haja outro processo cujo estado atual possua um envio correspondente a essa mensagem. Em outras palavras, um processo órfão é um processo que possui uma entrega de mensagem que não foi enviada por nenhum outro processo da aplicação.

Um estado global consistente de uma aplicação com troca de mensagens é ilustrado na Figura 2.2. A figura mostra três estados globais da execução de uma aplicação com três processos (A, B e C). As setas representam mensagens entre processos. Os vértices são os estados dos processos individuais e cada estado global é representado pelas linhas tracejadas (S, S' e S"). Infelizmente, nem todo estado global é um estado global consistente. No exemplo, os estados S e S' levariam a um resultado correto se uma restauração fosse realizada a partir deles. No entanto, o estado S" é inconsistente pois o processo B não possui registro de envio da mensagem $M_{b_{j+7}a_{i+8}}$. Se a aplicação fosse reiniciada a partir desse estado global e o processo A enviasse uma mensagem para qualquer outro processo antes da mensagem $M_{b_{j+7}a_{i+8}}$ ser enviada por B, teria-se uma execução inconsistente.

Quando um processo restaurado está em um estado órfão pode acontecer o denominado efeito dominó. Nesse caso, é necessário restaurar estados dos processos que deveriam ter enviado alguma mensagem que não tem registro de envio no estado global. Essa restauração pode criar novos processos órfãos, indefinidamente, retrocedendo a computação e desperdiçando o processamento já concluído. Nos casos que há necessidade de retroceder a computação para tratar o efeito dominó cada processo precisa manter múltiplos estados dos processos. A seguir serão apresentadas as técnicas utilizadas para evitar os processos órfãos baseadas apenas em *checkpoint* e as que, adicionalmente, empregam log de mensagens.

2.1.3.1 Protocolos de recuperação baseados em checkpoint

Essencialmente, a recuperação de falhas baseada em *checkpoints* salva periodicamente o estado de cada processo envolvido na computação de forma que um estado anterior da aplicação possa ser restaurado quando ocorrer uma falha. Existem três variantes desse tipo

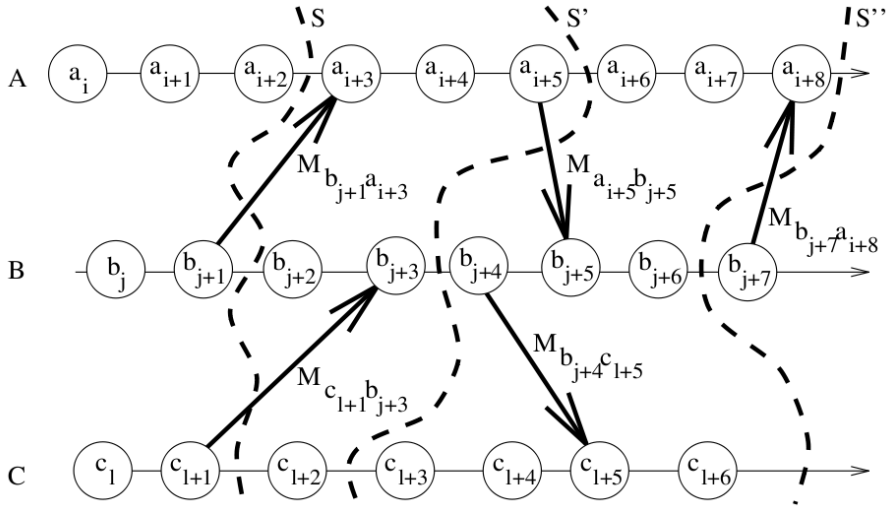

Figura 2.2: Estados Globais de uma Aplicação com Troca de Mensagens

de recuperação: o *checkpoint* não-coordenado, o *checkpoint* coordenado e o *checkpoint* induzido pela comunicação. Daqui em diante, o termo *checkpoint* será utilizado para denominar o estado de um processo armazenado em algum meio.

No *checkpoint* não-coordenado cada processo decide independentemente quando deve criar seus *checkpoints*. Isso elimina a necessidade de sincronização entre os processos diminuindo o sobrecusto do mecanismo de *checkpoint*. No entanto, essa abordagem é suscetível ao efeito dominó, o que pode acrescentar um sobrecusto substancial no procedimento de recuperação e no armazenamento de múltiplos *checkpoints*.

O *checkpoint* coordenado se baseia no algoritmo de Chandy-Lamport (DIJKSTRA, 1986) para criar *checkpoints* dos processos nos estados globais consistentes da aplicação. O algoritmo previne o efeito dominó nas recuperações e requer o armazenamento de apenas um *checkpoint* por processo. Considerando os métodos de armazenamento atuais, dependendo do número de processos sendo executados, a criação de *checkpoints* em estados globais consistentes se torna inviável conforme será explicado na Seção 2.2.

O *checkpoint* induzido pela comunicação mistura o comportamento dos últimos dois protocolos. Cada processo gera localmente seus *checkpoints* independentemente, como no protocolo não-coordenado. Adicionalmente, são realizados *checkpoints* forçados para manter um estado global consistente e impedir que o sistema fique vulnerável ao efeito dominó. Cada mensagem trocada entre os processos possui informação extra que indica se o processo deve ou não realizar um *checkpoint* forçado. Como desvantagem, o algoritmo exige o armazenamento de múltiplos *checkpoints* por processo.

2.1.3.2 Protocolos de recuperação baseados em log de mensagens

Os protocolos de log de mensagens (ALVISI; MARZULLO, 1998) são utilizados tipicamente para complementar os protocolos baseados em *checkpoint* com registros das mensagens entregues pelos processos. Se um processo falha, os logs podem ser utilizados em conjunto com o *checkpoint* para restaurar o processo até o estado mais recente e evitar processos órfãos. Além disso, a abordagem utilizando logs não sofre o efeito dominó já que o processo restaurado pode avançar a computação através dos logs em vez de precisar retroceder a computação dos demais processos em busca de um estado consistente.

A rigor, esses protocolos não exigem necessariamente a utilização conjunta com *checkpoints* dos processos. Na prática, a utilização dos *checkpoints* auxilia economizando memória que seria necessária para o log de todas as mensagens desde o início da computação. Assim, os *checkpoints* se tornam pontos de corte, a partir dos quais os logs de mensagens anteriores aos *checkpoints* mantidos para os processos são descartados. Além disso, a existência de *checkpoints* como pontos de corte implica que a computação não precisará retroceder ao início da aplicação para a restauração dos logs armazenados.

Esses protocolos consideram que todos processos são formados por sequências determinísticas de estados separadas por eventos não-determinísticos. Os eventos não-determinísticos, no caso, são os recebimentos de mensagens. Para que esses protocolos funcionem, assume-se que seja possível identificar os respectivos envios para cada um dos recebimentos de mensagem[1]. Naturalmente, os envios de mensagem devem ser eventos determinísticos, assim como a execução das demais operações dos processos.

A forma como o protocolo evita ou trata os processos órfãos é a característica que define cada tipo de técnica, e se reflete no sobrecusto da execução da aplicação sem falhas e no desempenho do procedimento de recuperação. As três técnicas conhecidas atualmente são a de log pessimista, log causal e log otimista.

Em todas as técnicas, a informação realmente importante para o log é o determinante do evento de entrega de cada mensagem. O determinante de cada mensagem é uma tupla que contém as informações de origem, destino, número de sequência de mensagem enviada e número de sequência de mensagem recebida. Observa-se que os dados da mensagem não estão incluídos no determinante. De fato, os log dos dados não é essencial para restauração das aplicações nessas técnicas, no entanto, para efeitos práticos, o log dos dados acarreta em um ganho de desempenho tão grande que o faz ser utilizado em quase todas as implementações de log de mensagens.

Na técnica de log pessimista a recuperação de um processo que falhou é simplificada às custas de um maior sobrecusto durante uma execução sem falhas. Isso acontece porque, para cada mensagem entregue no processo, o determinante dessa mensagem é guardado antes que qualquer outra mensagem seja enviada. Com isso se garante que nunca haverá processos órfãos na aplicação.

No log otimista a execução sem falhas é otimizada. Nesse tipo de log não existe a garantia de que o processo guardará em log o determinante da mensagem imediatamente após a sua entrega. A garantia é de que, em algum momento, o determinante da mensagem será entregue. Nesse meio tempo, entre a entrega e o log do determinante, se ocorrer alguma falha, a aplicação poderá criar processos órfãos e será necessário um procedimento de restauração mais oneroso que poderá envolver a recuperação de *checkpoints* de outros processos.

Por exemplo, se um processo A recebe uma mensagem m, envia uma mensagem m' para um processo B e em seguida falha, antes de realizar o log do determinante de m, então o processo B se torna um processo órfão e precisará restaurar um *checkpoint* anterior ao recebimento da mensagem m'. Em alguns casos, dependendo de quantos determinantes deixaram de ser guardados em log, pode ser necessário retroceder os processos muito atrás na computação até encontrar um estado global consistente. A combinação desse tipo de log de mensagens com *checkpoints* não-coordenados exige o armazenamento de vários *checkpoints* por processo a fim de tratar os casos com processos órfãos. O log pessimista só precisa armazenar um *checkpoint* por processo pois os determinantes armazenados em log garantem que não haverá inconsistências na recuperação da aplicação.

[1]Essa exigência é conhecida como *Piecewise Deterministic Assumption* (PWD)

A última técnica é a de log causal que tenta combinar o baixo sobrecusto durante a execução sem falhas do log otimista com a recuperação rápida do log pessimista. Diferente do log pessimista, que guarda imediatamente o log do determinante após a entrega da respectiva mensagem, o log causal envia os determinantes junto com as mensagens de aplicação através de técnicas de *piggybacking*. O log causal é menos estrito do que o log pessimista no sentido de que garante o armazenamento das informações de causalidade das mensagens sem a necessidade de exigir que o processo receptor guarde o determinante das mensagens antes da sua entrega.

Os determinantes recebidos pelos processos são utilizados para montar um grafo de dependência causal das mensagens, denominado grafo de precedência. Uma mensagem m' tem dependência causal de uma mensagem m se ela foi enviada por um processo P_1 após P_1 ter feito a entrega de m. Essa propriedade é transitiva, de forma que se um processo P_2 recebe m' e depois envia m'' para outro processo, m'' também tem dependência causal de m. Dessa forma, os grafos de precedência podem se tornar bastante grandes e enviá-las cumulativamente por *piggybacking* é bastante oneroso. Para contornar esse problema as implementações da técnica utilizam métodos para podar o grafo e minimizar o sobrecusto de sua propagação.

Cada técnica de log de mensagens aqui introduzida apresenta um compromisso entre desempenho do procedimento de recuperação e o sobrecusto da aplicação sem erros. A adoção de uma determinada técnica deve levar em consideração a plataforma de execução bem como o comportamento das aplicações. Os fatores determinantes na escolha da técnica empregada são: o número de processos da aplicação; o tamanho dos *checkpoints* gerados; a quantidade e o tamanho das mensagens comunicadas entre os processos; o desempenho e a capacidade do meio de armazenamento confiável; a taxa de falhas da plataforma; a conformidade da aplicação com PWD.

A Tabela 2.1 (TREASTER, 2005) apresenta um comparativo das principais características das técnicas de recuperação retroativa, nessa ordem: o número de *checkpoints* criados para cada processo; se o mecanismo é suscetível ao efeito dominó; se podem ocorrer processos órfãos após uma falha; se a recuperação retroativa pode se estender por vários *checkpoints* até encontrar um estado global consistente; se o mecanismo assume que a aplicação respeita PWD; e se o protocolo de recuperação é distribuído ou local a cada processo.

Tendo em vista as tendências no aumento no número de nodos das arquiteturas de clusters e o consequente aumento na taxa de falhas nas arquiteturas, algumas técnicas se apresentam mais apropriadas que outras. Um elemento que não é desejável em uma técnica de recuperação retroativa nesse cenário é a necessidade de armazenamento de múltiplos *checkpoints* por processo pois isso pode sobrecarregar o meio de armazenamento confiável conforme será mostrado na próxima seção. Com isso, as técnicas mais adequadas entre as apresentadas são a de *checkpoint* coordenado, e as de log pessimista e causal. Além disso, considerando as perspectivas de tempo necessário para realização de *checkpoint* coordenado que serão apresentadas na Seção 2.2, essas últimas duas técnicas de log de mensagens se tornam mais atrativas, já que ambas utilizam checkpoint não coordenado.

Entre as técnicas de log pessimista e causal, a causal apresenta um sobrecusto menor quando se considera a necessidade de bloqueio do processo que o log pessimista impõe para realizar log do determinante das mensagens entregues. No entanto, a técnica de log causal apresenta o problema do armazenamento e propagação do grafo de precedência das mensagens, que possui um tratamento bastante oneroso (RAO; ALVISI; VIN, 2000).

Checkpoint	**Não-Coordenado**	**Coordenado**	**Induzido pela Comunicação**
Número de *Checkpoints*	≥ 1	1	≥ 1
Efeito Dominó	Sim	Não	Não
Processos Órfãos	Sim	Não	Sim
Extensão da Recuperação	Sem limite	Último CKPT	Vários CKPTs
Assume PWD?	Não	Não	Não
Protocolo de Recuperação	Distribuído	Distribuído	Distribuído
Log de Mensagens	**Pessimista**	**Otimista**	**Causal**
Número de *Checkpoints*	1	≥ 1	1
Efeito Dominó	Não	Não	Não
Processos Órfãos	Não	Sim	Não
Extensão da Recuperação	Último CKPT	Vários CKPTs	Último CKPT
Assume PWD?	Sim	Sim	Sim
Protocolo de Recuperação	Local	Distribuído	Distribuído

Tabela 2.1: Comparativo das Diferentes Técnicas de Recuperação Retroativa

2.2 Tolerância a Falhas em Petaescala/Exaescala

As arquiteturas de petaescala atualmente encabeçam as listas dos supercomputadores de melhor desempenho (DONGARRA et al., 1997). Essas arquiteturas alcançam a ordem dos petaflops e possuem centenas de milhares de processadores. Alguns esforços atuais pretendem determinar quais são as principais barreiras tecnológicas para atingir o desempenho de exaescala, ou seja arquiteturas que atinjam os exaflops. Com base nas arquiteturas atuais de petaescala, estima-se que as arquiteturas de exaescala possuirão milhões de processadores.

O relatório da DARPA (KOGGE et al., 2008) sobre exaescala afirma que os principais desafios remetem a problemas de economia de energia, refrigeração, tolerância a falhas, escalabilidade, modelos de computação e paradigmas de programação mais adequados. No caso da tolerância a falhas utilizando *checkpoints* pode-se fazer uma projeção do desempenho a partir de uma arquitetura atual.

O Blue Gene/L, por exemplo, foi projetado para realizar *checkpoints* coordenados em 12 minutos e implementa diversas técnicas para redução do sobrecusto envolvido na criação do *checkpoint* (*checkpoint* incremental, hash de memória, etc). Com esse tempo de *checkpoint* e um intervalo de *checkpoints* de 2,5 horas obtém-se um sobrecusto total de 8% do tempo de execução de uma aplicação sem falhas ($12min/150min$). Sabendo o tempo necessário para realizar um *checkpoint* (t), o período entre um *checkpoint* e outro (p), e o tempo médio entre falhas (MTBF), pode-se calcular o sobrecusto das interrupções por *checkpoint* e por falhas:

$$Sobrecusto = \frac{t}{p} + \frac{p}{2 \times MTBF} \tag{2.1}$$

com sobrecusto mínimo quando

$$0 = \frac{t}{p} + \frac{p}{2 \times MTBF} \tag{2.2}$$

$$\frac{t \times (2 \times MTBF) + p^2}{p \times (2 \times MTBF)} = 0 \tag{2.3}$$

$$p = \sqrt{2 \times t \times MTBF)} \tag{2.4}$$

Extrapolando esse comportamento para sistemas de exaescala, se a taxa de falhas por chip de processador não reduzir, então o MTBF permanecerá na escala de poucas horas. Se assumirmos que a memória dos processos de exaescala será algumas ordens de magnitude maior que as de arquiteturas de Teraescala, com o Blue Gene então o tempo de duração do *checkpoint* passará de 12 minutos para um valor que tornará o sistema inutilizável. A projeção é de que seja preciso 5 horas para realização do *checkpoint* se não houverem progressos nas tecnologias de armazenamento, algoritmos de *checkpoint* e taxa de falhas por chip de processador. Obviamente, 5 horas inviabilizaria o progresso das aplicações caso o intervalo entre *checkpoints* fosse de 2,5 horas.

No relatório da DARPA são sugeridas algumas abordagens para solucionar esses problemas com o *checkpoint*: utilização de memória de estado sólido (flash) para armazenamento dos *checkpoints*; *checkpoints* inteligentes, nos níveis de aplicação ou de sistema, que salvam apenas as estruturas de dados necessárias; suporte dos compiladores ou do ambiente de execução, que seleciona automaticamente (ou baseado em diretivas do usuário) o que salvar; mudar para um modelo de programação no qual a recuperação é inerente, como transações de sistemas comerciais e de bancos de dados.

As técnicas de *checkpoint* não-coordenado com log de mensagens têm sido menos utilizadas até então devido à sua implementação mais complicada e seu sobrecusto mais elevado na execução normal (LEMARINIER et al., 2004). Estima-se que essas técnicas sejam mais adequadas para as arquiteturas de exa/petaescala por algumas razões. Uma delas é que necessitam restaurar os *checkpoints* de todos os processos durante a restauração da aplicação como nos *checkpoints* coordenados. A restauração simultânea de todos os processos implica em um desperdício substancial de computação e também sobrecarrega os meios de armazenamento confiável (CAPPELLO, 2009) (OUYANG; GOPALAKRISHNAN; PANDA, 2009). Esse desperdício de computação e também a sobrecarga nos meios de armazenamento se tornam ainda mais impactantes quando o número de processos sobe para a ordem de grandeza esperada para as arquiteturas de exaescala. Outra razão, é que não existe a necessidade de criação de *checkpoints* globais coordenados, o que exigiria uma quantidade de tempo maior que o próprio MTBF nas arquiteturas de exaescala.

2.3 MPI

O MPI (*Message Passing Interface*) (MPI FORUM, 1994) é uma especificação de interface para bibliotecas de troca de mensagens. Essencialmente, essa interface prevê rotinas para comunicação e sincronização de aplicações paralelas. O MPI foi derivado de outras interfaces de troca de mensagens independentes que foram desenvolvidas por fabricantes de hardware e grupos desenvolvedores de aplicações no início da década de 90. Nesse período, houve um crescente interesse pelo desenvolvimento em arquiteturas de memória distribuída. Assim, o MPI serviu como padronização para um modelo de programação em crescente popularização. O sucesso do MPI pode ser atribuído ao próprio modelo de programação que estava se estabelecendo, ao seu alto desempenho e escalabilidade, e às várias implementações existentes.

A primeira versão da especificação prevê rotinas para envio e recebimento de mensagens ponto-a-ponto de forma síncrona e assíncrona. Os recebimentos assíncronos do MPI podem evitar cópias de *buffer* desnecessárias deixando o programador especificar um *buffer* alocado antes da mensagem ser recebida. Além disso, essas rotinas permitem

uma concorrência maior na aplicação pois não são bloqueantes. Em contrapartida, a utilização de rotinas síncronas em aplicações MPI é bem mais simples. Além das mensagens ponto-a-ponto, também são previstas rotinas de comunicação coletiva que permitem que um subconjunto de processos execute operações como `MPI_Bcast`, `MPI_Allreduce` ou `MPI_Alltoall`. O MPI-2 (MPI-2: EXTENSIONS TO THE MESSAGE-PASSING INTERFACE, 1997) é a segunda versão do padrão MPI e estende a primeira especificação com interfaces para rotinas de escrita em memória remota, criação dinâmica de processos, comunicação coletiva entre intercomunicadores e operações de E/S.

O MPI é o ambiente predominantemente utilizado em sistemas paralelos de alto desempenho de todas as escalas. Apesar disso, existem controvérsias sobre a continuidade de sua utilização devido à dificuldade de programar sistemas mais complexos e à pouca expressividade do modelo de troca de mensagens. Outro fator é o advento das arquiteturas *multicore* e a crescente exposição do paralelismo no chip. Uma forma de explorar mais eficientemente essas arquiteturas é a utilização de modelos híbridos como troca de mensagens e memória compartilhada com MPI e OpenMP (CHAPMAN; JOST; PAS, 2007). No entanto, isso acrescenta mais complexidade ainda à programação.

Embora a especificação abranja o suporte a muitos recursos para aplicações paralelas, ainda deixa em aberto os problemas relacionados a tolerância a falhas. Com o advento das arquiteturas de peta/exaescala, e sua crescente suscetibilidade a falhas, a necessidade de suporte para esses problemas se faz cada vez mais presente. Mesmo ainda não havendo especificação definida no padrão já existem várias implementações que tentam tratar esses problemas. Na seção seguinte, são apresentadas as técnicas de tolerância a falhas implementadas para o MPI.

2.4 Tolerância a Falhas e MPI

A Figura 2.3 é uma extensão do panorama extraído do artigo do MPICH-V (BOSILCA et al., 2002) e apresenta as principais implementações de MPI que oferecem mecanismos de tolerância a falhas. Como pode-se perceber, as técnicas de recuperação retroativa são as mais estudadas para tolerância a falhas com MPI, apesar de existirem algumas abordagens diferentes.

Conforme ilustra a figura, de acordo com o nível da implementação essas implementações podem ser classificadas da seguinte forma: as que são implementadas nas bibliotecas de comunicação; as que disponibilizam uma API com funcionalidades de tolerância a falhas para o programador; as que consistem em um *framework* que utiliza a API MPI de forma totalmente transparente. O outro elemento considerado no panorama é o mecanismo de tolerância a falhas utilizado, desde as implementações de *checkpoint* até as várias formas de log de mensagens e outros mecanismos que serão comentados em seguida. A seguir serão apresentadas as implementações mais proeminentes: o MPICH-V, AMPI e Open MPI. Depois serão brevemente introduzidas outras implementações de tolerância a falhas que também são consideradas relevantes.

2.4.1 MPICH-V

O projeto MPICH-V (BOSILCA et al., 2002) (*Volatile Resources*) visa oferecer suporte a tolerância a falhas para arquiteturas dinâmicas. Existem diversas implementações, cada qual utilizando uma técnica diferente. O MPICH-V é um conjunto de módulos do MPICH (GROPP et al., 1996) que, por sua vez, é uma das implementações pioneiras de MPI, entre as mais utilizadas, tanto em ambientes de pesquisa quanto de produção.

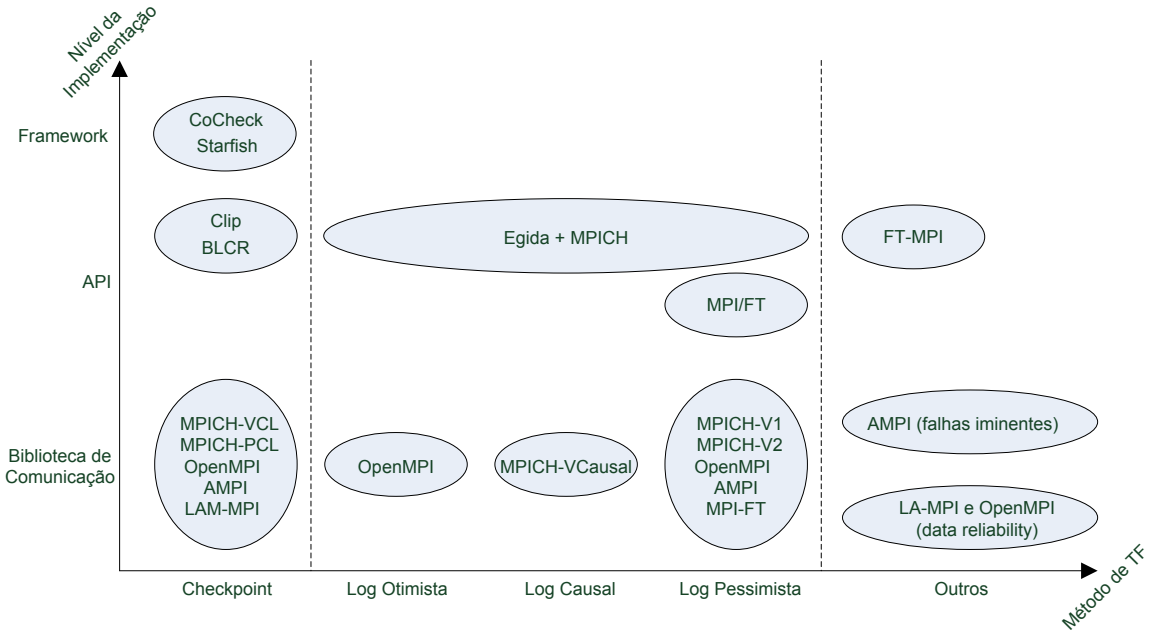

Figura 2.3: Panorama das implementações de MPI com suporte a tolerância a falhas

As implementações MPICH-V1 e MPICH-V2 (BOUTEILLER et al., 2003) utilizam protocolos de log de mensagens para o suporte a tolerância a falhas. Ambas utilizam *checkpoint* não-coordenado com log pessimista, diferindo no local de armazenamento dos logs. Enquanto o MPICH-V1 armazena os logs em servidores remotos dedicados, o MPICH-V2 guarda as mensagens no próprio nodo remetente. O armazenamento local permite obter um melhor desempenho da aplicação, no entanto, não cobre a recuperação de falhas em múltiplos nodos em determinados casos. A ocorrência de falhas em múltiplos nodos pode ocorrer, por exemplo, devido a uma falha no abastecimento de energia de um *switch*.

Supondo o caso de um nodo com um processo X falhar logo após ter recebido uma mensagem de um processo Y em outro nodo. Na implementação MPICH-V2, se o nodo do processo Y também sofrer uma falha então o processo X não terá como restaurar a mensagem que havia recebido, pois essa foi perdida na falha do processo Y. Por outro lado, o MPICH-V1 armazena os logs em servidores dedicados e, portanto, suporta múltiplas falhas de nodos.

A Figura 2.4 ilustra os elementos da arquitetura do MPICH-V1. Os processos nos nodos de computação realizam a troca de mensagens através de canais de memória localizados em servidores remotos para permitir a realização do log pessimista remoto. Com esses canais, tanto as mensagens quanto os seus determinantes são armazenados. Cada conjunto de processos é atendido por um canal de memória. Como pode-se imaginar, esses canais de memória podem se tornar um gargalo para a comunicação dependendo do número de processos clientes que atendem. Pensando nisso, o MPICH-V1 permite configurar o número de canais de memória por servidor e o número de processos por canal de memória do ambiente. Além disso, servidores dedicados armazenam os *checkpoints* dos processos. Esses *checkpoints* são gerados através da biblioteca Condor (LITZKOW et al., 1997).

A arquitetura do MPICH-V2 e do MPICH-VCausal (BOUTEILLER et al., 2005) está apresentada na Figura 2.5. No MPICH-V2, o log de mensagens deixa de ser feito em

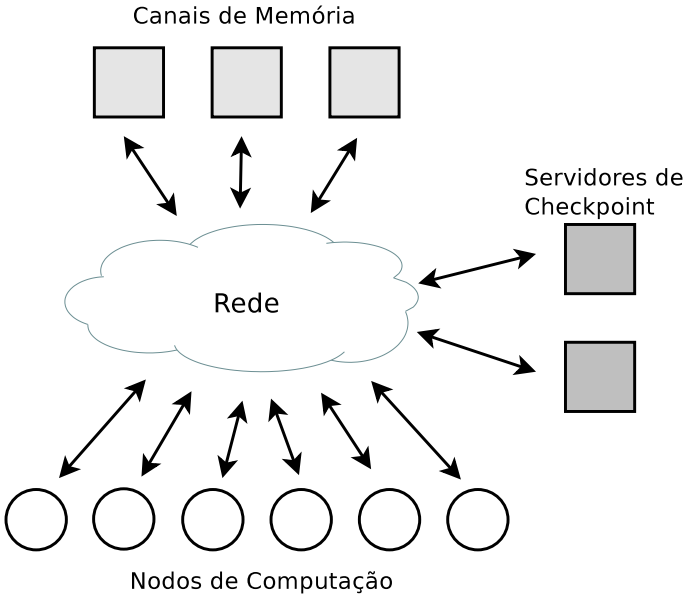

Figura 2.4: Arquitetura do MPICH-V1

canais de memória e passa a ser realizado nos próprios nodos remetentes das mensagens. As informações dos determinantes das mensagens são armazenadas em um log de eventos remoto para identificar a ordem das mensagens nas fases de recuperação. Podem haver múltiplos servidores de log de eventos para melhor escalabilidade do sistema. A partir dessa versão do MPICH-V os *checkpoints* passaram a ser realizados através da biblioteca BLCR (HARGROVE; DUELL, 2006). O MPICH-VCausal é a implementação de log de mensagens causal do MPICH-V que armazena periodicamente os determinantes em servidores de eventos remotos para podar os grafos de precedência propagados entre os processos.

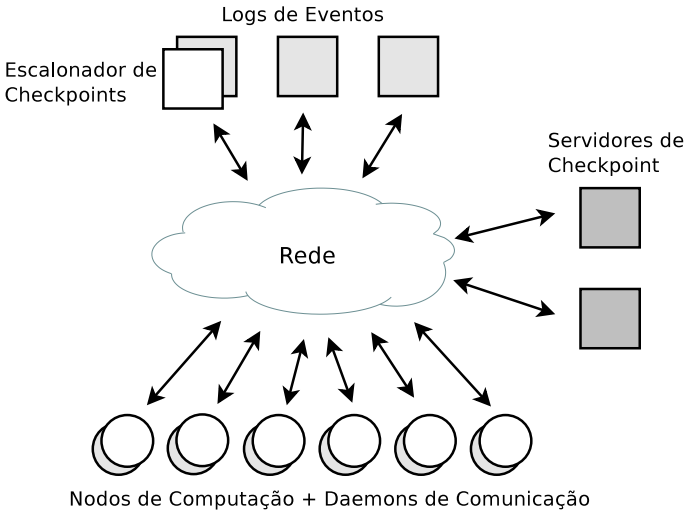

Figura 2.5: Arquitetura do MPICH-V2 e MPICH-VCausal

Nas outras duas implementações do MPICH-V, o MPICH-VCL (LEMARINIER et al., 2004) e o MPICH-PCL (COTI et al., 2006) são realizados *checkpoints* globais coordena-

dos para tolerância a falhas. A Figura 2.6 apresenta os elementos da arquitetura, semelhante à do MPICH-V2, diferindo na inexistência de um servidor de log de eventos. Os *daemons* de comunicação, nesse caso, não fazem log de mensagens.

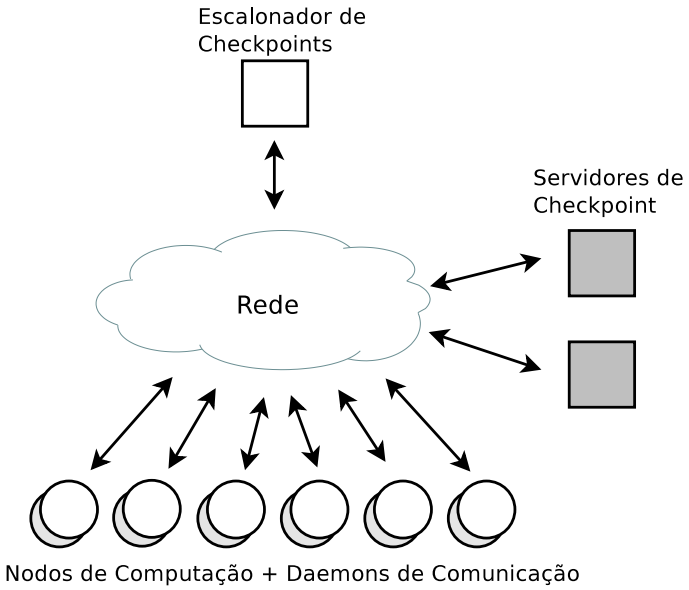

Figura 2.6: Arquitetura do MPICH-VCL e MPICH-PCL

A diferença entre o MPICH-VCL e o MPICH-PCL é a forma como é implementado o algoritmo de *checkpoint* coordenado. O MPICH-VCL usa uma implementação pura do algoritmo de Chandy-Lamport enquanto no MPICH-PCL, o processo que inicia o *checkpoint* coordenado força o envio dos marcadores imediatamente para todos os processos, para fazer a limpeza dos canais de comunicação. Essa modificação evita o sobrecusto existente no MPICH-VCL durante a execução normal causado pelos marcadores enviados juntamente com as mensagens. Entretanto, o método do MPICH-PCL causa o bloqueio de todos os processos da aplicação durante os *checkpoints*.

De todas as implementações do MPICH-V, apenas o MPICH-VCL e o MPICH-PCL continuam em desenvolvimento. As demais implementações tiveram o desenvolvimento descontinuado.

2.4.2 Adaptive MPI

O AMPI (CHAO HUANG ORION LAWLOR, 2003) (*Adaptive MPI*) utiliza o *framework* orientado a objetos Charm++ (KALE; KRISHNAN, 1993). Esse *framework* emprega o conceito de processadores virtuais. As implementações convencionais de MPI dividem a computação em P processos. O AMPI, por sua vez, divide a computação entre V processadores virtuais que são mapeados em processadores físicos.

A idéia essencial do AMPI é deixar apenas a tarefa de definir “o que” paralelizar para o usuário e tratar das questões sobre “onde” e “quando” paralelizar. Em outras palavras, o AMPI trata do balanceamento de carga dinâmico das aplicações de forma transparente. Baseado em critérios de carga pré-definidos, o AMPI migra processos entre processadores virtuais automaticamente para evitar processadores sobrecarregados.

O AMPI possui quatro tipos de suporte a tolerância a falhas: *checkpoint* coordenado; *checkpoint* coordenado duplo em memória remota ou disco local; um esquema de eva-

cuação proativa de processos em nodos com falha iminente; e um protocolo de log de mensagens pessimista.

A implementação de *checkpoint* coordenado (HUANG, 2004) do AMPI apresenta os problemas de desempenho descritos anteriormente na Seção 2.2. Posteriormente, para minimizar o problema do gargalo de acesso a mídia confiável durante as etapas de recuperação, a implementação foi alterada de forma que o *checkpoint* também é guardado na memória local de cada nodo (ZHENG; KALÉ, 2004). Isso diminui drasticamente o tempo de recuperação, uma vez que evita o acesso concorrente ao meio de armazenamento, e acessa a memória local que é muito mais veloz. No entanto, essa modificação só é viável se o tamanho dos processos permitir que seja guardada uma cópia de *checkpoint* na memória local. De qualquer forma, representa um desperdício substancial de memória.

O esquema de evacuação proativa do AMPI (CHAKRAVORTY; KALÉ, 2006) migra processos de nodos com falhas iminentes. O hardware moderno pode prever falhas com um bom grau de precisão e o AMPI se vale desses recursos para evitar que os processos sejam comprometidos migrando os processos antes que as falhas ocorram.

Na implementação de log de mensagens pessimista (CHAKRAVORTY; KALE, 2007) o AMPI cria uma cópia de cada mensagem na memória dos processos remetentes. Essa implementação explora os recursos de virtualização de processos de forma que, quando os processos de um nodo que falhou são recuperados, esses são distribuídos entre os nodos para equilibrar a carga da aplicação durante a recuperação.

2.4.3 Open MPI

O Open MPI (GABRIEL et al., 2004) é uma implementação relativamente nova mantida por um consórcio de órgãos acadêmicos, de pesquisa e da indústria. O grupo de desenvolvedores é formado por membros de várias implementações de MPI e representa a consolidação de suas experiências e conhecimentos. O foco dessa implementação é no melhoramento da confiabilidade e alguns recursos específicos estão baseados na tolerância a falhas de processos e da comunicação de rede.

Atualmente, o Open MPI oferece suporte para *checkpoint* coordenado e possui implementações experimentais para log de mensagens pessimista/otimista, tolerância a falhas na comunicação de rede, e notificação de erros de comunicadores e processos.

O suporte a *checkpoint* coordenado (HURSEY et al., 2007) é baseado na implementação do LAM-MPI (SQUYRES, 2003) (SANKARAN et al., 2003). Os *checkpoints* individuais dos processos são realizados através da BLCR, inclusive no LAM-MPI.

O Open MPI também possui um mecanismo baseado no LA-MPI (AULWES, 2004) que assume que o meio de transmissão de dados não é confiável. O DR (*Data Reliability*) (SHIPMAN; BOSILCA, 2007) descarta e reinicia conexões comprometidas, e trata da retransmissão de dados corrompidos ou perdidos para assegurar a confiabilidade da comunicação.

Um outro mecanismo de tolerância a falhas foi herdado do FT-MPI (FAGG; DONGARRA, 2000) e serve para aplicações dinâmicas nas quais os processos que falham não devem comprometer todo o restante da aplicação. A semântica original do MPI indica que quando um processo falha todos os seus comunicadores associados se tornam inválidos, inclusive o MPI_COMM_WORLD [2]. Dessa maneira toda a aplicação é comprometida. Esse mecanismo estende os estados possíveis dos comunicadores e processos de forma que o usuário possa verificá-los e programar o tratamento apropriado para a sua

[2]Comunicador global da aplicação MPI

aplicação.

O módulo Vprotocol é a base para a implementações de log de mensagens pessimista (BOUTEILLER et al., 2009) e otimista do Open MPI. O log pessimista, é similar ao do MPICH-V2 e guarda as mensagens na memória do nodo remetente. Cada fragmento de mensagem enviado é mantido em uma região de memória alocada com *mmap* (QUARTERMAN, 1992) para que possa ser recuperado em caso de falha. Para diminuir a utilização de memória, essas regiões alocadas são armazenadas em disco periodicamente. Para armazenar os eventos de mensagens existe um processo dedicado que não executa processamento da aplicação, apenas recebe os eventos não determinísticos. A diferença entre a implementação pessimista e a otimista está no momento no qual os eventos são logados, conforme explicado na Seção 2.1.3.2.

2.4.4 Outras implementações

O CoCheck (STELLNER, 1996) é um *framework* que trabalha sobre a API do MPI e torna as aplicações paralelas tolerantes a falhas através da utilização de *checkpoint* coordenado, baseado no algoritmo de Chandy-Lamport. O disparo dos procedimentos de *checkpoint* e de restauração são feitos por um coordenador centralizado.

O Starfish (AGBARIA; FRIEDMAN, 1999) é a implementação de tolerância a falhas mais próxima do CoCheck. No Starfish é possível que o usuário controle as operações de *checkpoint* e recuperação através de uma API e existem duas modalidades de *checkpoint*: o coordenado, similar à implementação do CoCheck, e o não-coordenado, apropriado para aplicações trivialmente paralelizáveis. Nessa implementação de *checkpoint* não-coordenado, quando ocorre a falha de um processo da aplicação, todos os demais que permanecem ativos são notificados. A aplicação deve decidir a ação a ser tomada para prosseguir com a execução.

O Clip (CHEN; PLANK; LI, 1997) é uma biblioteca de *checkpoint* coordenado de nível de aplicação dedicada aos sistemas IntelParagon. Essa biblioteca trabalha de forma não transparente e pode ser utilizada em aplicações MPI através de chamadas explícitas às funções de *checkpoint*.

O MPI/FT (BATCHU et al., 2004) faz *checkpoint* pessimista, e utiliza vários sensores para detectar falhas nos níveis de aplicação, MPI, rede e sistema operacional. Um coordenador centralizado monitora o progresso da aplicação, realiza log de mensagens, reinicia os processos falhos a partir dos últimos *checkpoints* e gerencia mensagens de controle para autodiagnóstico. Uma desvantagem MPI/FT apontada pelos autores é o coordenador centralizado que faz com que as aplicações escalem somente até 10 processos.

O MPI-FT (LOUCA et al., 2000) é uma implementação bastante similar ao MPICH-V1, entretanto é baseada no LAM-MPI. Essa implementação possui um processo denominado *observer* que oferece detecção e recuperação de falhas de processos MPI. A detecção de falhas funciona com ajuda de um script Unix que verifica periodicamente a existência de todos os processos. O MPI-FT oferece duas modalidades de log, a otimista e a pessimista, ambas sem suporte a *checkpoint*. A inexistência de *checkpoints* implica em grandes requerimentos de capacidade de armazenamento para os logs de mensagens, visto que todas as mensagens, desde o início do processamento da aplicação, precisam ser guardadas. Outra desvantagem da falta de *checkpoint* verifica-se no momento da recuperação de um processo falho. Nesse caso, é necessário realizar novamente toda a computação desse processo desde o início.

O Egida (RAO; ALVISI; VIN, 1999) é uma biblioteca que oferece suporte a vários métodos de recuperação baseada em log de mensagens para o MPICH. Para a imple-

mentação de cada método de tolerância a falhas, o Egida possui uma linguagem que permite expressar eventos tais como a detecção de falhas e os *checkpoints*, e as respectivas respostas para esses eventos. Apesar de ser necessário especificar o mecanismo de tolerância a falhas através dessa linguagem, a aplicação MPI não precisa ser modificada.

A *Berkeley Lab Checkpoint Restart* (BLCR) (HARGROVE; DUELL, 2006) é uma biblioteca de nível de sistema que opera de forma preemptiva para realização de *checkpoint/restart* em clusters Linux. Os autores creditam à preemptividade as seguintes vantagens: o *checkpoint* transparente, sem intervenção do usuário; o escalonamento das operações de *checkpoint* permite uma melhor utilização do sistema. A BLCR é utilizado por várias implementações de MPI (LAM-MPI, Open MPI, MPICH-V) como suporte para a realização de várias técnicas de *checkpoint restart* de aplicações paralelas.

2.5 Considerações Finais

Os sistemas de computação paralela de alto desempenho contam com o suporte de diversas técnicas de tolerância a falhas para suas aplicações e ambientes de execução. Para resolver o problema de servidores críticos e centralizados existem técnicas de replicação e monitoramento. A aplicação paralela, por sua vez, tipicamente é tratada através de técnicas de recuperação retroativa baseadas em *checkpoint* com ou sem log de mensagens. Cada técnica empregada apresenta um compromisso entre o desempenho da aplicação sem falhas e o tempo necessário para sua recuperação.

Atualmente, os sistemas que estão no topo dos rankings de desempenho dispõem de centenas de milhares de computadores. Alguns desses sistemas já ultrapassaram a barreira dos petaflops e, por isso, são denominados sistemas de petaescala. Apesar do potencial de desempenho, essa grande quantidade de processadores acompanha desafios relacionados a tolerância a falhas.

Conforme o número de processadores aumenta, também aumenta a probabilidade de falhas no sistema. As técnicas tradicionalmente empregadas começam a se tornar inapropriadas conforme a escala de número de processadores aumenta. Considerando os sistemas de exaescala previstos para 2015, com milhões de processadores e capacidade de processamento superior a 1 exaflops, as técnicas atuais serão ineficientes e em alguns casos até inúteis, pois estima-se que com as técnicas de *checkpoint* tradicionalmente utilizadas o tempo necessário previsto para sua criação é superior ao próprio MTBF da arquitetura. Dessa forma, a aplicação não teria como progredir no processamento pois as operações de *checkpoint* seriam quase sempre interrompidas. Existem idéias sugeridas pela DARPA de como aprimorar o *checkpoint* para evitar esse problema. Mesmo em arquiteturas nas quais o *checkpoint* ainda é viável o processo de restauração de *checkpoints* coordenados, tipicamente utilizado, desperdiça uma quantidade substancial de processamento, já que todos os processos necessitam ser restaurados para o último estado global consistente.

As técnicas de *checkpoint* coordenado com log de mensagem têm sido menos utilizadas até então devido ao seu sobrecusto na comunicação sem falhas e implementação mais complexa, se comparados com as técnicas de *checkpoint* coordenado. Com o novo cenário de arquiteturas na escala de milhões de processadores elas se tornam mais atraentes. A técnica apresenta a vantagem no procedimento de *checkpoint*, que não é coordenado, e, por isso, pode ser viável de ser empregado em arquiteturas de exaescala. Também é vantajosa na própria restauração da aplicação pois não precisa restaurar todos os processos da aplicação, evitando o desperdício de computação que já foi concluída.

As implementações do padrão MPI são os ambientes predominantemente utilizados para computação paralela de alto desempenho. Apesar do padrão não especificar mecanismos de tolerância a falhas, existem muitas implementações desses mecanismos. A maioria delas usa técnicas de recuperação retroativa com *checkpoint* e percebe-se atualmente uma grande preocupação com a implementações de técnicas de log de mensagens pessimista. Todas as grandes implementações de MPI, como o MPICH, AMPI e Open MPI já possuem implementações desse tipo.

Tendo em vista que as falhas nas arquiteturas de exaescala não serão a exceção mas sim a regra, é condição *sine qua non* o emprego de técnicas de tolerância a falhas para viabilizar a utilização desses sistemas. Considerando que as técnicas de log de mensagens são as mais promissoras desenvolveu-se um estudo acerca dessas. Segue no próximo capítulo a proposta de solução utilizando log de mensagem pessimista remoto sobre MOM *(Message-Oriented Middleware*) com comunicação *Publish/Subscriber*.

3 LOG DE MENSAGENS PESSIMISTA REMOTO SOBRE MOM

Como visto no capítulo anterior, existe uma preocupação muito grande da comunidade de alto desempenho sobre o problema de tolerância a falhas em arquiteturas de exaescala (KOGGE et al., 2008). Existem questões não respondidas sobre como as técnicas atualmente utilizadas irão escalar com as arquiteturas emergentes. As projeções feitas, com base no desempenho das arquiteturas atuais, sugerem que essas técnicas terão de ser aprimoradas ou substituídas por outras mais eficientes (CAPPELLO, 2009).

Uma solução que foi implementada recentemente em várias versões do MPI é a de log pessimista local (BOUTEILLER et al., 2003) (BOUTEILLER et al., 2009). Essa solução apresenta algumas características que a tornam apropriada para as arquiteturas com uma grande escala de processos e taxas de falha muito altas.

A proposta de solução para o problema de tolerância a falhas visa estender a solução de log de mensagens através da utilização de MOM (Message-Oriented Middleware) com comunicação hierárquica através de múltiplos servidores aproveitando as propriedades da comunicação *Publish/Subscriber*. A solução é destinada a comportar aplicações de forma transparente. Devido à grande complexidade das aplicações com um número de processos na ordem dos milhões, é desejável evitar que o usuário precise alterar a aplicação ou tenha que definir o tratamento de tolerância a falhas explicitamente.

A seguir são apresentados os conceitos de MOM e as suas propriedades de desacoplamento de comunicação. Depois será mostrada a arquitetura proposta sobre MOM e a operação das primitivas MPI sobre esta arquitetura. Por último serão feitas algumas considerações finais sobre esta proposta.

3.1 MOM e Noções de Desacoplamento de Comunicação

A utilização de MOM (*Message-Oriented Middleware*) (HOHPE; WOOLF, 2003) (BANAVAR et al., 1999) possui algumas propriedades bastante úteis na implementação de comunicação de aplicações paralelas e distribuídas. A seguir são apresentadas algumas dessas propriedades e são destacadas as que são particularmente úteis para a proposta de solução do presente trabalho.

A principal aplicação e motivação para o surgimento dos MOM é a integração de aplicações corporativas. A integração se refere tanto à necessidade de interoperabilidade de aplicações de diferentes instituições quando ocorrem fusões, cisões ou incorporações, como também na necessidade de comunicação entre plataformas diferentes (PDAs, celulares, sistemas operacionais e softwares distintos, etc). Nesses casos o MOM é utilizado como uma interface de comunicação uniforme entre os diferentes sistemas e incorpora

alguns recursos que auxiliam na interoperabilidade dos sistemas.

Os dois principais modelos de comunicação utilizados em sistemas de MOM são o *Publish/Subscriber* e o PTP (*Point-to-Point*). A comunicação PTP segue o modelo de filas FIFO produtor-consumidor no qual um processo produtor coloca em uma fila mensagens que são lidas e apagadas por um processo consumidor na mesma ordem que foram armazenadas na fila. O modelo *Publish/Subscriber* é visto por alguns como uma evolução natural do modelo de comunicação em grupo (BANAVAR et al., 1999). Esse modelo se baseia na utilização de tópicos hierárquicos nos quais as aplicações interessadas se inscrevem. As aplicações interessadas podem publicar mensagens nos tópicos de forma que todos os inscritos a recebem.

A Figura 3.1 exemplifica o funcionamento da comunicação *Publish/Subscriber*. Nessa figura, o MOM possui dois tópicos para publicação de mensagens, `/cotacao/euro` e `/cotacao/dolar`. O processo C se inscreve (registra interesse em receber mensagens do tópico) no tópico `/cotacao/euro` e o processo D no tópico `/cotacao/dolar`. O processo E se inscreve em ambos os tópicos através da utilização de um *wildcard* `/cotacao/*`. Nota-se que é possível que um processo se inscreva em mais de um tópico e que um mesmo tópico tenha inscrições de mais de um processo.

Os processos A e B anunciam mensagens no MOM. Os anúncios podem ser feitos para tópicos específicos como `/cotacao/dolar` ou mesmo para um conjunto de tópicos, tipicamente endereçado através de *wildcards*. No exemplo, para enviar mensagens para ambos os tópicos utiliza-se o *wildcard* `/cotacao/*`.

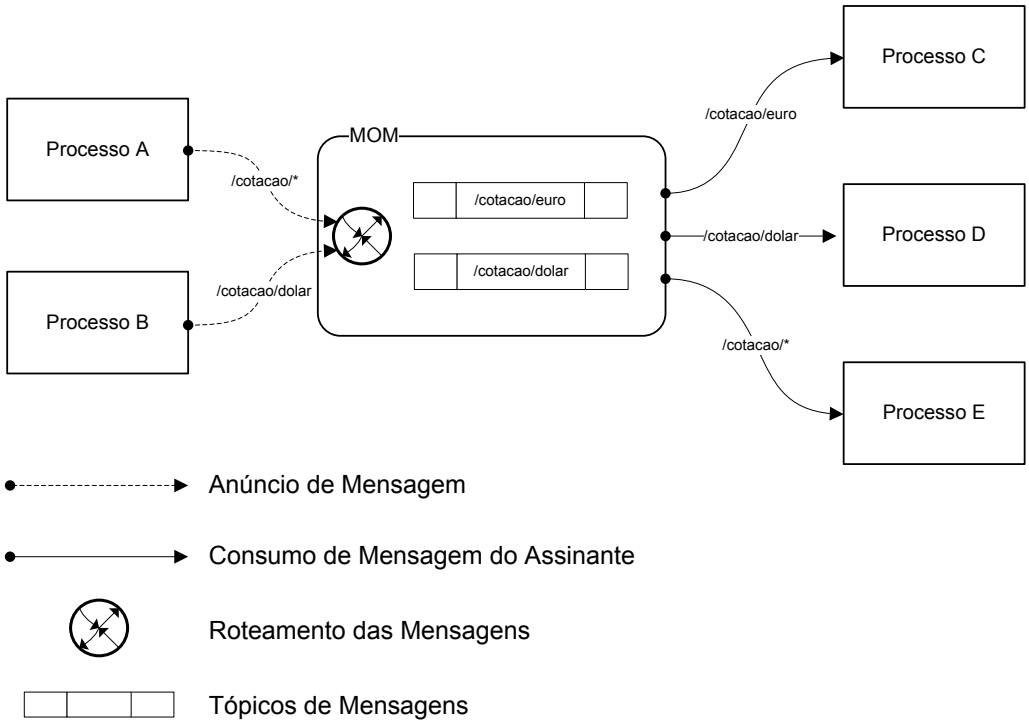

Figura 3.1: Anúncio e Consumo de Mensagens em Tópicos *Publish/Subscriber*

Termos como síncrono, assíncrono, bloqueante, não-bloqueante, direcionado e não-direcionado são geralmente utilizados para se referir ao grau de acoplamento na comunicação de determinados sistemas. Porém, esses termos são utilizados com diferentes conotações por diferentes autores e fabricantes de software (ALDRED et al., 2005).

Por exemplo, quando se diz que uma aplicação possui envio de mensagens assíncrono, são encontradas no mínimo duas interpretações. Uma interpretação se refere ao conceito de *rendez vous* (COULOURIS, 2001) dos *middlewares* de mensagem. Nesse contexto, trocas de mensagens assíncronas são feitas com *buffers* auxiliares para permitir o envio da mensagem concorrentemente com a computação do processo, sem a necessidade de *rendez vous*, ou seja, sem a necessidade de interação direta entre os processos comunicantes. A outra interpretação é relacionada com uma propriedade do MOM que permite que os envios de mensagens e os seus correspondentes recebimentos possam ser feitos mesmo que o par da comunicação não esteja disponível (GIOTTA et al., 2000). Essas duas interpretações correspondem, respectivamente, aos conceitos de desacoplamento de sincronização e desacoplamento temporal que serão explicados a seguir.

Utilizaremos a terminologia proposta por Eugster (EUGSTER et al., 2003) a fim de distinguir conceitos que possam ser interpretados ambiguamente. A terminologia é baseada em três propriedades essenciais oferecidas nos softwares de MOM cujo modelo de comunicação se caracteriza por permitir que as aplicações comunicantes não precisem fazer suposições a respeito das suas disponibilidades e localizações:

- **Desacoplamento Temporal** quando o emissor ou o receptor de uma mensagem não necessita que seu correspondente esteja disponível ao mesmo tempo durante a operação. Está relacionado com as propriedades de persistência de mensagens dos *middlewares*.

- **Desacoplamento Espacial** quando o emissor ou o receptor referenciam um endereço simbólico e não um endereço real nas operações de envio e recebimento. Está relacionado com o conceito de canais de comunicação.

- **Desacoplamento de Sincronização** indica se a operação é não bloqueante, ou seja: no caso de envio, devolve o controle da execução para a aplicação enquanto executa a operação de envio em segundo plano; no caso de recebimento a operação é realizada em segundo plano e, ou a primitiva de recebimento é chamada explicitamente para entregar a mensagem à aplicação, ou é disparado um evento que faz a entrega assim que a mensagem for recebida.

Essas três propriedades são ortogonais entre si. Isto é, combinando todas as possibilidades obtém-se um total de 16 formas de comunicação possíveis (4 de acoplamento de sincronização, 2 de acoplamento temporal e 2 de acoplamento espacial). As combinações abrangem todas as possibilidades de interação entre aplicações e estão ilustradas nas Figuras 3.2 e 3.3. Cada interação possui 2 processos, A e B, representadas por uma seta para baixo e um círculo cinza que representa a mensagem sendo transferida.

Na Figura 3.2, as interações possuem acoplamento espacial. Isto é, as terminações da comunicação conhecem os endereços reais umas da outras. Já a Figura 3.3, representa os casos de desacoplamento espacial que utilizam um canal de comunicação. Os tópicos do MOM são bons exemplos de canais de comunicação com desacoplamento espacial. Quando se publicam mensagens no tópico, não é necessário saber os endereços reais dos destinatários.

As linhas tracejadas indicam trechos nos quais ocorre bloqueio da execução do processo para espera da execução das primitivas de envio ou recebimento. As interações que não possuem essas linhas tracejadas representam os casos de desacoplamento de sincronização. Já o desacoplamento temporal é representado por uma descontinuidade na

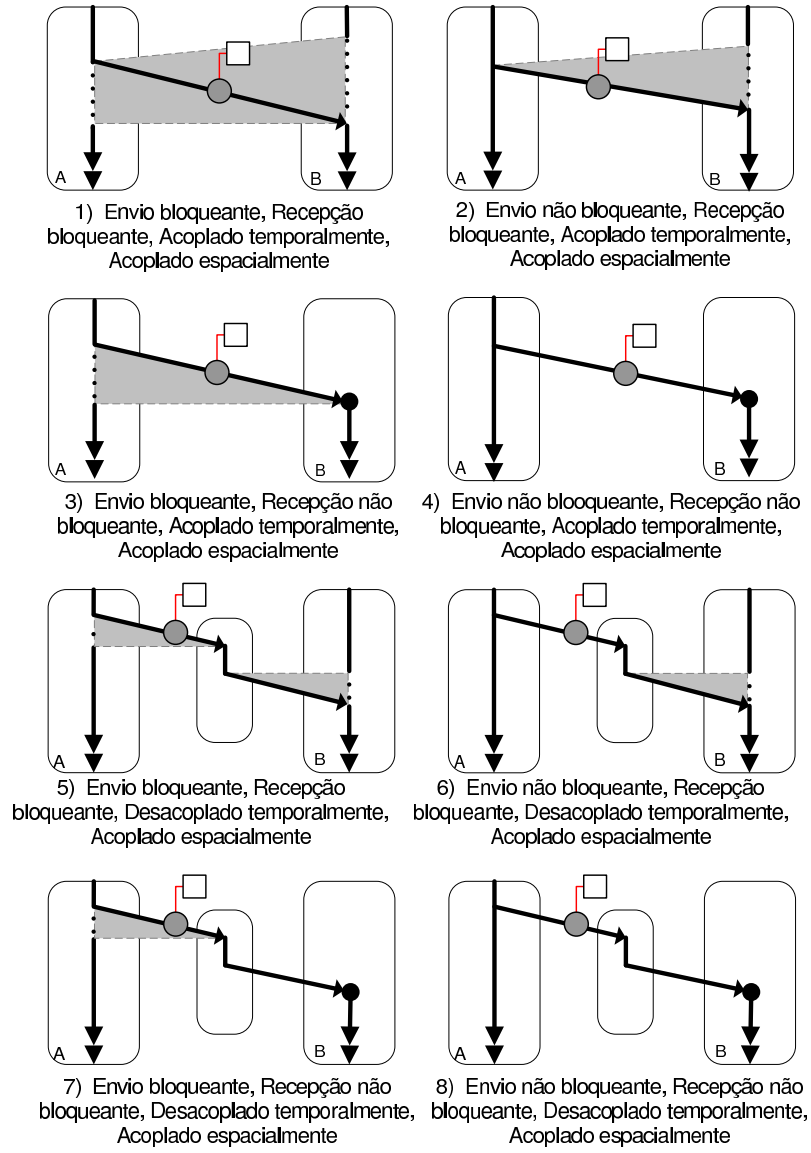

Figura 3.2: Possibilidades de acoplamento de comunicação (Acoplado espacialmente)

reta da mensagem entre os processos A e B, normalmente implementado através da utilização de filas de mensagens.

3.2 Proposta de Log de Mensagens Pessimista Remoto sobre MOM

A arquitetura da presente proposta, representada na Figura 3.4, se baseia na utilização de comunicação *Publish/Subscriber* para realização de tolerância a falhas com *checkpoint* não-coordenado e log de mensagens [1] pessimista remoto. A camada do *Backend* é responsável pelo mapeamento da comunicação MPI para a comunicação com tópicos *Publish/Subscriber* e também pela geração dos *checkpoints* periódicos em mídia confiável remota.

A criação de *checkpoints* não-coordenados dos processos das aplicações MPI é uti-

[1] Lembrando que a utilização de log de mensagens para tolerância a falhas implica no suporte apenas de aplicações que respeitem PWD (*Piecewise Deterministic*) como visto na Seção 2.1.3.2.

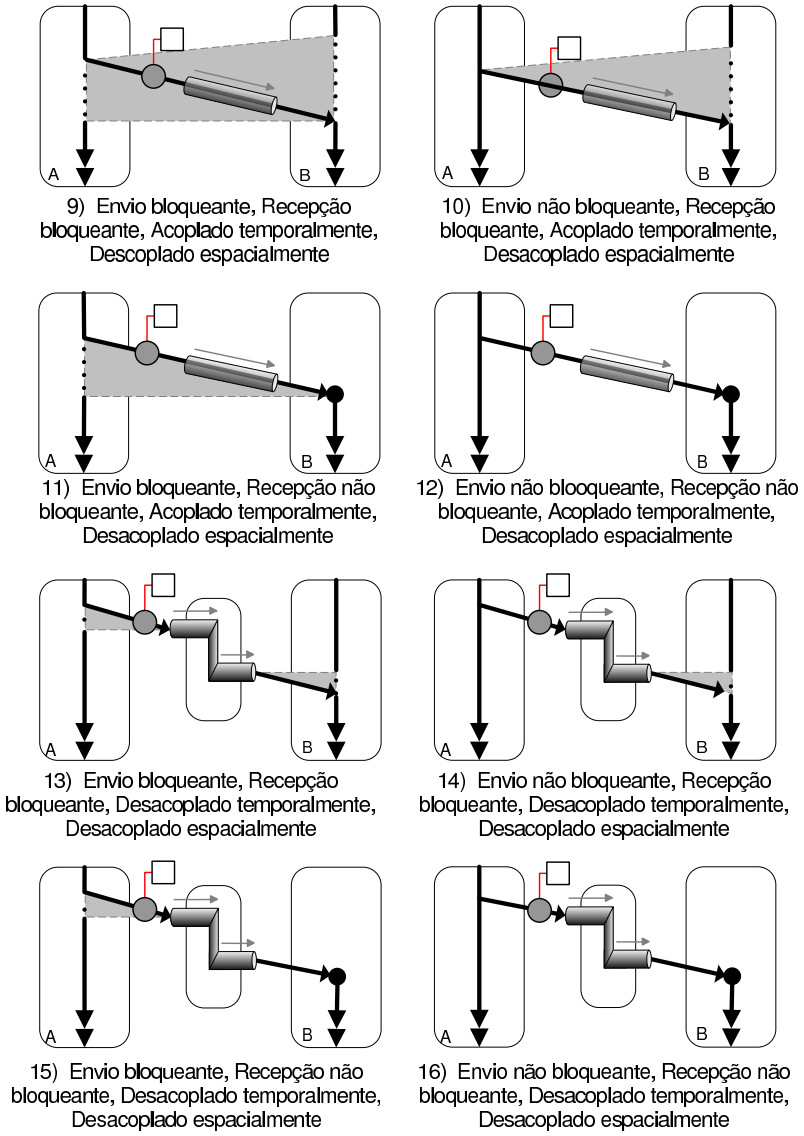

Figura 3.3: Possibilidades de acoplamento de comunicação (Desacoplado espacialmente)

lizada em combinação com log de mensagens para restaurar as aplicações para o seu estado mais recente. Os *checkpoints* não-coordenados não exigem nenhum tipo de sincronização entre os processos que os realizam. Logo, os *checkpoints* podem ser realizados independentemente. Essa característica é muito vantajosa para aplicações com extrema escala de processos pois desonera o *middleware* de sincronizações que poderiam se tornar um gargalo para o desempenho da aplicação.

A periodicidade da criação dos *checkpoints* deve levar em consideração o tempo necessário para concluí-los e a taxa de falhas da arquitetura utilizada. Uma frequência de *checkpoints* muito alta pode causar um sobrecusto muito grande e uma frequência mais baixa que a taxa de falhas pode inutilizar o mecanismo de tolerância a falhas como um todo.

Quando uma falha [2] é detectada o procedimento de recuperação deve buscar o *checkpoint* do processo armazenado em mídia confiável e retomar sua execução. Se houver

[2] A proposta prevê o tratamento de falhas no modelo *fail-stop*

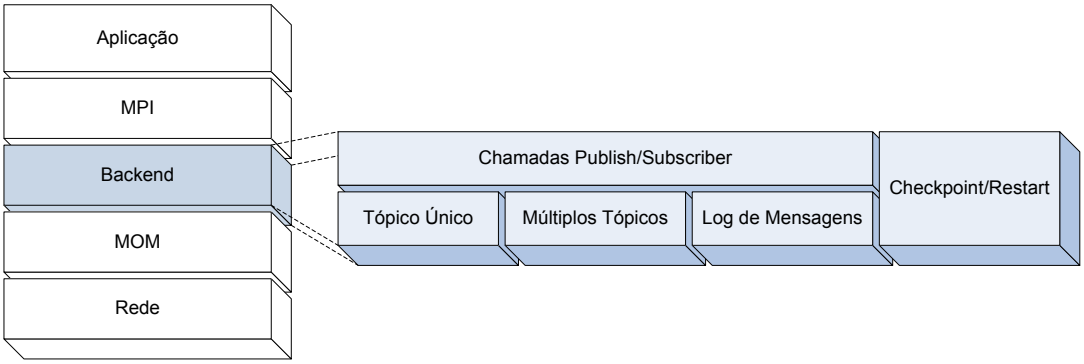

Figura 3.4: Camadas da arquitetura proposta

mensagens no log para serem recuperadas o processo irá primeiramente receber essas. A forma como é realizada a comunicação normal e o recebimentos de mensagens logadas é detalhada a seguir.

A Figura 3.5 ilustra uma aplicação MPI hipotética para exemplificar o mapeamento da comunicação MPI para *Publish/Subscriber*. Nela cada quadrado representa um processo. Existem dois grupos criados pela aplicação do usuário além do grupo do comunicador global `MPI_COMM_WORLD` [3]. Para cada grupo que um determinado processo pertence existe um identificador distinto para esse processo (*rank*). Por exemplo, o processo com *rank* 4 no grupo do comunicador `MPI_COMM_WORLD` também possui *rank* 0 no grupo do comunicador `MPI_COMM_GRP2`. No mapeamento para *Publish/Subscriber*, cada *rank* de cada um desses grupos possuirá um tópico correspondente para recebimento de mensagens, como será detalhado a seguir.

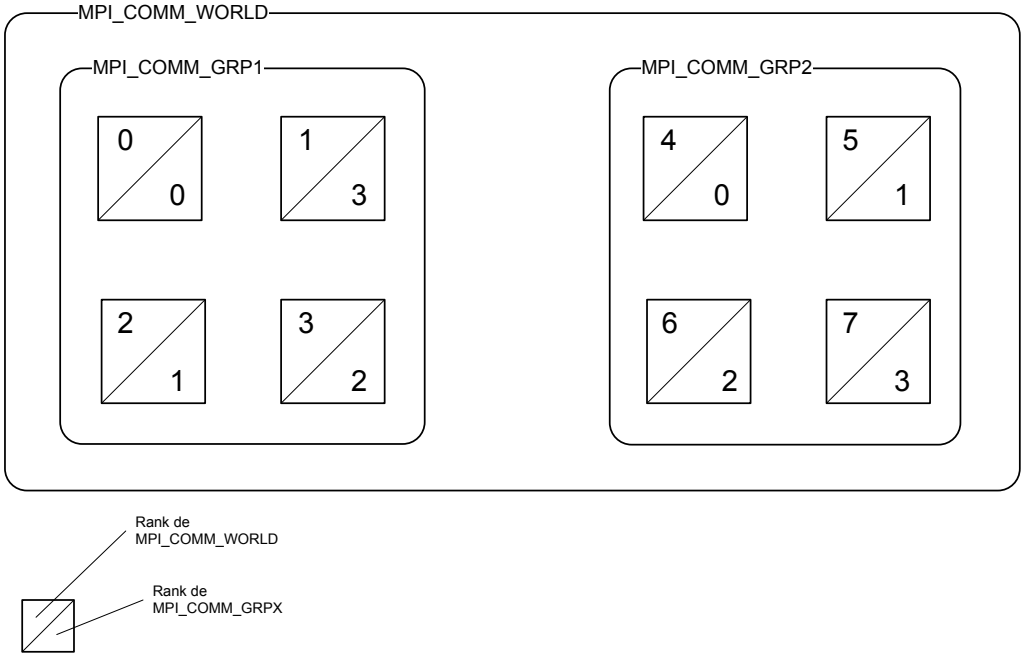

Figura 3.5: Comunicadores e *Ranks* dos Processos de uma Aplicação MPI

[3]No exemplo apresentado os grupos do usuário formam conjuntos disjuntos, no entanto a proposta não impõe essa restrição às aplicações e permite intersecções entre grupos de processos.

A comunicação *Publish/Subscriber* é utilizada para a comunicação normal da aplicação e para a comunicação durante as etapas de restauração de logs de mensagens. Durante a execução normal da aplicação podem haver dois tipos de publicação de mensagens. As mensagens podem ser publicadas em tópicos de *ranks* ou em múltiplos tópicos. A publicação para tópicos de *rank* serve para mensagens ponto a ponto enquanto as publicações em múltiplos tópicos servem para difusão (*broadcast*) de mensagens em grupos de processos, o que pode ser utilizado em algumas primitivas de comunicação coletiva.

Para exemplificar, nos basearemos na Figura 3.6 que ilustra a inscrição em tópicos da aplicação MPI da Figura 3.5 e o repositório de *checkpoints* remoto. Assim como na Figura 3.5, os quadrados representam processos da aplicação com seus respectivos *ranks*. As linhas contínuas representam as inscrições para consumo de mensagens durante a execução normal da aplicação enquanto as linhas tracejadas servem para o log de mensagens que será explicado mais adiante. Os tópicos nomeados `/WORLD/0`, `/WORLD/3`, `/GRP1/0`, ... pertencem aos *ranks* de processos das aplicações. Para publicação de mensagens para os *ranks* dos processos basta utilizar o nome dos tópicos. Também é possível publicar mensagens em todos os tópicos de *rank* de processos pertencentes a um determinado grupo utilizando *wildcards* como `/WORLD/*` e `/GRP1/*`[4]. Nesse tipo de publicação, em múltiplos tópicos, o MOM se encarrega de enviar uma cópia da mensagem para cada tópico de *rank* que case com o *wildcard*. Na Seção 3.3 serão detalhados os tipos de publicação utilizados pelas principais primitivas do MPI.

O desacoplamento temporal do sistema de MOM permite que as mensagens sejam enviadas e recebidas pelos processos sem que o receptor/emissor correspondente da operação esteja disponível. O desacoplamento temporal é feito tipicamente através da utilização de servidores com filas de mensagens que funcionam como um repositório intermediário no qual as mensagens são armazenadas antes de serem recebidas pelos destinatários. A presente proposta utiliza essa propriedade para realizar log de mensagens.

No caso, o log das mensagens fica armazenado nas próprias filas dos tópicos *Publish/Subscriber* do MOM. O comportamento típico dos sistemas de MOM é apagar dos tópicos as mensagens que já foram consumidas por todos os seus processos inscritos. Logo, para manter a mensagem no tópico e, dessa forma, realizar o log, uma inscrição extra é utilizada para cada tópico de *rank*. Essas inscrições extra nos tópicos, representadas pelas linhas tracejadas na Figura 3.6, são feitas pelos processos da aplicação e servem apenas para garantir que o MOM não vai apagar as mensagens até que o próximo *checkpoint* periódico seja realizado. Após a criação bem sucedida de um *checkpoint*, as inscrições de *rank* do processo destinadas a log consomem todas as mensagens para apagá-las do tópico. Elas são apagadas pois já não são mais necessárias para restaurar o estado mais recente do processo.

As falhas de processos devem ser detectadas por *daemons* locais que também são responsáveis pela recuperação dos *checkpoints* armazenados remotamente. Devido ao desacoplamento temporal do MOM, as mensagens desse processo continuarão sendo recebidas na sua respectiva fila mesmo após a falha e mesmo durante a etapa de recuperação do *checkpoint*. Depois da recuperação do *checkpoint* de um processo que falhou, a inscrição destinada ao log de mensagens é estabelecida e são recuperadas as mensagens que já haviam sido entregues ao processo até o momento da falha. Somente depois de recuperadas todas as mensagens e a execução do processo atingir o último estado possível antes da falha, que as inscrições normais voltam a consumir as mensagens e as inscrições de log voltam a servir apenas para os log de mensagens. É importante observar que essa

[4]Para isso é necessário que o MOM utilizado suporte o envio de mensagens para múltiplos tópicos

solução não suporta falhas do processo no momento da sua restauração, uma vez que as mensagens da inscrição de log são apagadas depois de serem consumidas na restauração do processo.

Muitos sistemas de MOM podem ser configurados para guardar as mensagens nos tópicos por um tempo máximo estipulado (*time-to-live*). Para o modelo proposto essa funcionalidade deve ser desabilitada pois isso pode fazer os sistemas de MOM apagarem mensagens das aplicações indevidamente. Além disso, as inscrições nos tópicos devem ser sempre do tipo durável. Isto é, inscrições que permanecem ativas mesmo quando o processo que está inscrito estiver indisponível. Isso é necessário para que o o recebimento e log de mensagens continue sendo realizado mesmo em caso de falha do processo que fez a inscrição no tópico.

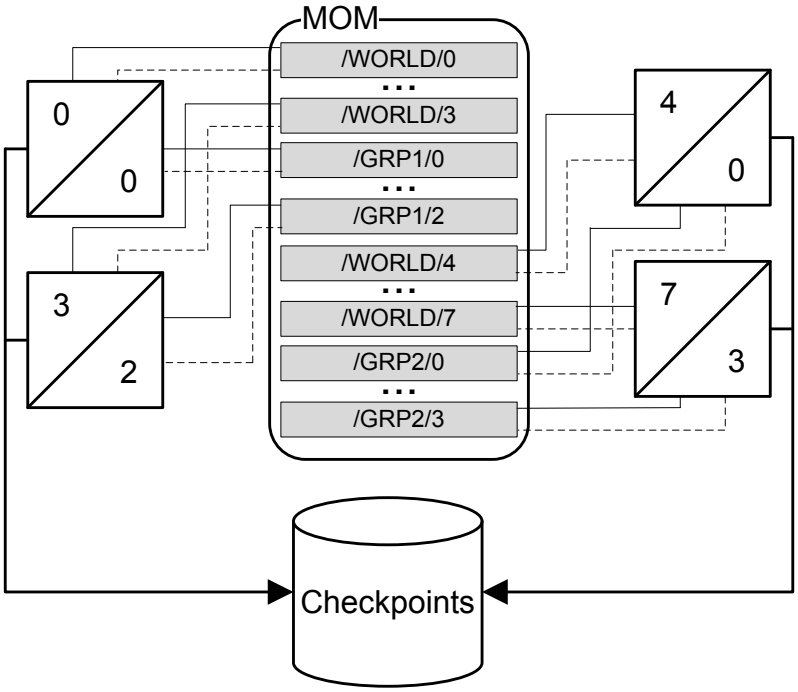

Figura 3.6: Tópicos para Comunicação/Log de Mensagens e Armazenamento de *Checkpoints*

Através da utilização de tópicos distribuídos em diferentes servidores, é possível distribuir a carga do sistema. A Figura 3.7 ilustra a distribuição de servidores para tratamento de tópicos baseado nos grupos apresentados na Figura 3.5. Cada servidor de entrada do MOM, denominado *broker*, fica responsável por determinados tópicos. O MOM fica responsável pelo roteamento das mensagens entre os servidores.

No exemplo da figura, os *brokers* permitem comunicação hierárquica da seguinte forma. Supondo que qualquer um dos processos da figura realizasse um `MPI_Bcast` para o comunicador `MPI_COMM_WORLD`, cujos tópicos dos processos que fazem parte têm os nomes iniciados por `/WORLD/`. A publicação em múltiplos tópicos, utilizando *wildcards* se daria endereçando o envio para `/WORLD/*`. Nesse caso, o *broker* deve encaminhar 3 mensagens, uma para cada um dos outros *brokers* que possuem tópicos sob a hierarquia de `/WORLD/`. Por fim, cada um desses *brokers* se responsabiliza por armazenar as mensagens nas filas correspondentes aos tópicos que estão sob a hierarquia de `/WORLD/`. Nesse exemplo, em vez de 7 trocas de mensagem necessárias para a pu-

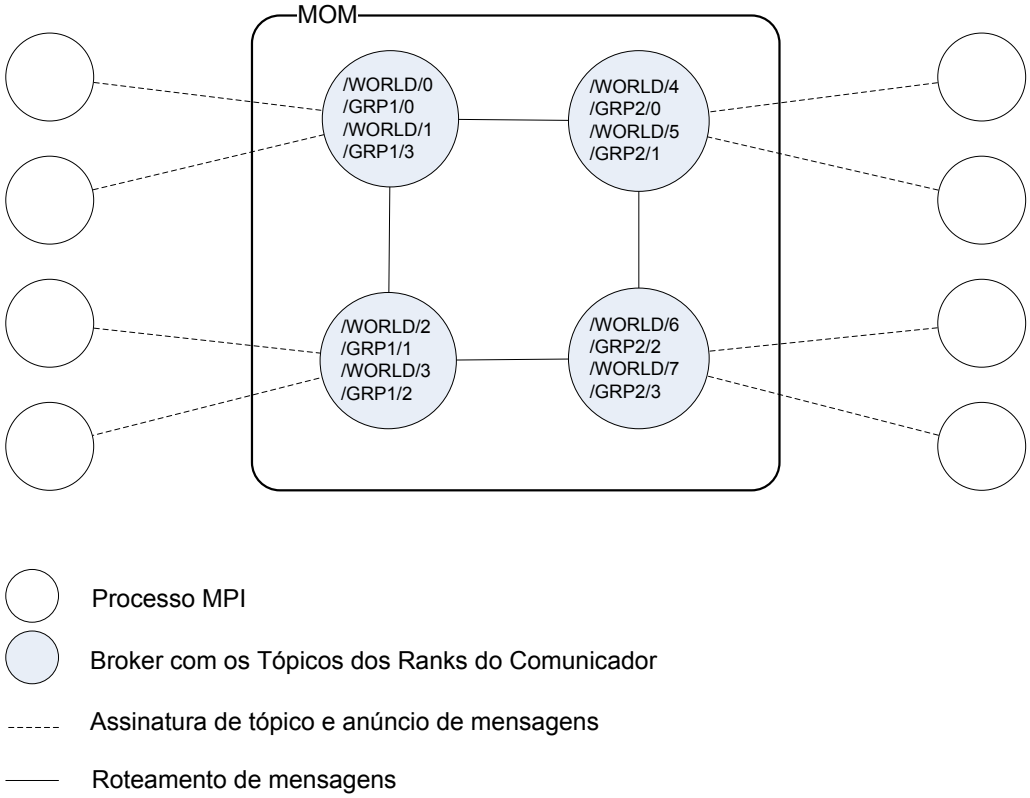

Figura 3.7: Tópicos Distribuídos em Múltiplos *Brokers*

blicação em cada um dos tópicos, são necessárias 4 trocas de mensagem, sendo que após a primeira o processo remetente já retorna o controle para a aplicação, e as outras três acontecem de forma independente dos processos da aplicação, entre os *brokers*.

3.3 Primitivas de Comunicação do MPI

Como visto na seção anterior as primitivas do MPI são mapeadas para o MOM através de publicações de mensagens em tópicos de *rank* e em múltiplos tópicos. No caso de múltiplos tópicos, se utilizam *wildcards* para fazer a difusão da mensagem em todos os tópicos de *rank* de determinado grupo.

Podemos dividir as primitivas MPI em dois tipos, de acordo com o tipo de comunicação de MOM que elas utilizam. As primitivas que utilizam somente publicações em tópicos individuais de *rank* fazem parte do primeiro tipo de mensagens. Já as primitivas que utilizam também publicações em múltiplos tópicos, para difusão de mensagens, fazem parte do segundo tipo.

A seguir apresentamos as primitivas ponto a ponto e as primitivas de comunicação coletiva do MPI. Essas últimas são subdivididas nas primitivas de comunicação coletiva que utilizam difusão de mensagens através de publicação em múltiplos tópicos e nas que utilizam apenas publicação em tópicos individuais de *rank*.

3.3.1 Primitivas de comunicação ponto a ponto

As primitivas de envio ponto a ponto no MPI recebem como parâmetro o comunicador do grupo, o identificador do processo destino (*rank*) e uma *tag* para o usuário poder classificar tipos de mensagem em um mesmo processo ou grupo de processos. Basicamente, o que interessa para o MOM é saber para quais tópicos as mensagens devem ser enviadas. Sendo assim, somente o comunicador e o *rank* são importantes na camada do MOM. A *tag* pode ser encapsulada na própria mensagem e recuperada apenas posteriormente no processo destino pela camada de integração do MPI com o MOM.

- **MPI_Send/MPI_Isend** As mensagens direcionadas a processos são enviadas para os tópicos dos processos correspondentes. No caso das primitivas não bloqueantes, a comunicação com o *broker* não é feita de forma síncrona com a chamada da primitiva. Ou seja, possui além do desacoplamento espacial e temporal do MOM, o desacoplamento de sincronização.

- **MPI_Recv/MPI_Irecv** As mensagens são recebidas diretamente nos tópicos correspondentes dos seus processos. O `MPI_Irecv` pode ser feito de duas formas com base nos modelos de entrega de mensagens existentes no sistemas de MOM. O modelo *Pull*, mais comum, exige que o receptor requisite a próxima mensagem. Se esse modelo for utilizado o sistema deve contar com um mecanismo de recebimento auxiliar para permitir as chamadas de `MPI_Irecv` com desacoplamento de sincronização. Esse mecanismo deve trazer as mensagens do tópico para o processo cliente em segundo plano e consumir as mensagens do tópico assim que essas forem recebidas pelo processo após uma chamada bem sucedida de `MPI_Irecv`. No modelo *Push*, no entanto, o próprio MOM se encarrega dessa tarefa.

3.3.2 Primitivas de comunicação coletiva

As primitivas de comunicação coletiva podem ser divididas em dois grupos no mapeamento para o MOM. As que utilizam somente comunicações ponto a ponto e as que fazem difusão de mensagens em grupo. As que fazem apenas comunicações ponto a ponto seguem a mesma linha de funcionamento anterior, enviando mensagens diretamente para os tópicos dos *ranks* específicos. As primitivas que fazem difusão em grupo, como `MPI_Alltoall` e `MPI_Allreduce`, recebem como parâmetro o grupo no qual a mensagem deve ser difundida e utilizam *wildcards* tais como `/WORLD/*` e `/GRP1/*` para enviar mensagens para todos os tópicos pertencentes ao grupo correspondente.

- **MPI_BCast** Para as operações de difusão *broadcast* pode-se utilizar operações de publicação em múltiplos tópicos suportadas pelos sistemas de MOM. Por exemplo, para publicar em todos os tópicos dos *ranks* do comunicador MPI_COMM_GRP1 da Figura 3.5 bastaria publicar uma mensagem para um *wildcard* como */GRP1/**.

- **MPI_Alltoall** Uma chamada dessas equivale a várias mensagens ponto a ponto nos tópicos dos *ranks*. Todos os processos do grupo trocam determinados blocos do *buffer* nas mensagens.

- **MPI_Scatter** Equivale a várias mensagens ponto a ponto nos tópicos dos *ranks* do grupo. Cada mensagem transfere uma porção do buffer para o processo correspondente.

- **MPI_Gather** É operação reversa do scatter. Cada processo com uma porção do buffer envia uma mensagem para o tópico do *rank* que agrega o buffer.

- **MPI_Reduce** As primitivas de redução do padrão MPI assumem que as operações são associativas e as implementações podem utilizar essas propriedades para otimizar a comunicação. Com ou sem a otimização, as comunicações são feitas ponto a ponto. Cada parcela da operação é enviada para o tópico do *rank* que deve fazer as operações parciais.

- **MPI_Allreduce** Funciona da mesma maneira que o `MPI_Reduce`. Adicionalmente, o resultado é difundido a todos os tópicos do grupo como um `MPI_Bcast`.

3.4 Considerações Finais

Entre as técnicas de tolerância a falhas conhecidas em aplicações paralelas, o log pessimista remoto é uma das técnicas que possui características mais adequadas para execuções com o número de processos previsto para a exaescala. O MPICH-V1 e a proposta apresentada neste capítulo implementam esse tipo de log de mensagens. A principal diferença entre o MPICH-V1 e a proposta aqui apresentada está no modo como o log de mensagens é feito. Na primeira, o log utiliza múltiplos servidores dedicados com canais FIFO de comunicação que atuam como repositórios intermediários das mensagens. Como visto, a utilização de tópicos *Publish/Subscriber* passa a tarefa dos canais FIFO para o MOM. Além disso, o MOM permite a utilização de tópicos distribuídos e hierárquicos, que auxiliam na distribuição da carga da comunicação e na otimização de comunicações com difusão, como o `MPI_Bcast`.

O MOM realiza o log de mensagens armazenando-as integralmente nas filas dos tópicos *Publish/Subscriber* durante as execuções normais das aplicações. Dessa forma, tanto o determinante quanto os dados da mensagem ficam armazenados em mídia remota confiável. A comunicação baseia-se na inscrição dos processos em tópicos para recebimento de mensagens. Assim, cada processo possui um tópico no qual os processos remetentes publicam mensagens que desejam enviar. O log das mensagens se dá através de uma inscrição extra que serve exclusivamente para manter as mensagens na fila do tópico e restaurá-las em caso de falha do processo. As mensagens só são apagadas efetivamente da fila de um tópico durante as restaurações dos processos falhos ou após a finalização bem sucedida do *checkpoint* do processo correspondente ao tópico.

A utilização de *checkpoints* não-coordenados associada ao log de mensagens permite diminuir o tamanho do log, descartando mensagens anteriores ao último procedimento de *checkpoint*. Um log menor é vantajoso para economia de memória e disco, e, principalmente, para otimizar as etapas de recuperação. Com os *checkpoints* a restauração das mensagens em log de um processo que falhou parte do seu último estado armazenado em vez de retroceder ao início da computação.

No próximo capítulo será apresentada a implementação de comunicação MPI sobre *Publish/Subscriber*. Essa comunicação é a base para a implementação do log de mensagens pessimista remoto da aqui apresentado. A partir disso, é possível integrar o *checkpoint* não-coordenado e os procedimentos de restauração, que complementariam a proposta deste capítulo.

4 IMPLEMENTAÇÃO DO COMPONENTE MTL/MOM

No Capítulo 3, foi apresentada a arquitetura de suporte a *checkpoint* não-coordenado e log de mensagens pessimista remoto. Como visto, essa arquitetura se baseia na comunicação MPI sobre MOM. A implementação desenvolvida, e apresentada no presente capítulo, serve como prova de conceito para a viabilidade de utilização de MOM como meio de comunicação para o MPI. A criação de *checkpoints* de processos individuais também foi implementada, no entanto, não foi integrada com um sistema de log de mensagens para restauração dos processos. Para finalização da arquitetura proposta ainda é necessária a integração do log de mensagens e *checkpoints* não-coordenados.

A seguir, serão apresentados os elementos da implementação. Essencialmente, dos blocos da arquitetura previamente apresentados na Figura 3.4 do Capítulo 3, foi implementado o bloco referente à comunicação *Publish/Subscriber* para tópicos únicos. Para isso, foi desenvolvido um módulo do Open MPI que mapeia sua API de comunicação ponto a ponto para o MOM OpenAMQ. Nas próximas seções serão detalhados os softwares Open MPI e OpenAMQ, o funcionamento do módulo Open MPI desenvolvido, bem como as decisões de projeto relacionadas com a sua implementação.

4.1 Open MPI

Decidiu-se utilizar o Open MPI por se tratar de uma das implementações do MPI mais utilizadas atualmente e por possuir um bom suporte da comunidade de alto desempenho. A versão utilizada é a 1.3.2.

A base do Open MPI é o MCA (*Modular Component Architeture*) que é subdividida em três entidades representadas pelos blocos cinza da figura 4.1: a base da arquitetura do MCA, os *frameworks* e os componentes. A base da arquitetura do MCA serve essencialmente para encontrar, carregar e descarregar bibliotecas de vínculo dinâmico, denominadas componentes, que implementam funções definidas por APIs internas do Open MPI, denominadas *frameworks*. Dessa forma, os *frameworks* definem espaços de nomes e interfaces para seus respectivos componentes que, por sua vez, implementam as funcionalidades previstas pelos *frameworks*. Podem existir diversos componentes para um mesmo *framework*, ou seja, diversas implementações de uma mesma funcionalidade com a mesma API definida no *framework*.

Os *frameworks* do Open MPI são divididos em três classes, de acordo com o tipo de funcionalidades que implementam: OPAL (*Open Portability Abstraction Layer*), ORTE (*Open RunTime Environment*) e OMPI. O OPAL oferece primitivas para auxiliar na interoperabilidade de diferentes sistemas, com *frameworks* para suporte a *threads*, temporizadores, afinidade de processador/memória e outros recursos que dependem do sistema operacional. O ORTE é responsável pela descoberta, mapeamento e alocação dos re-

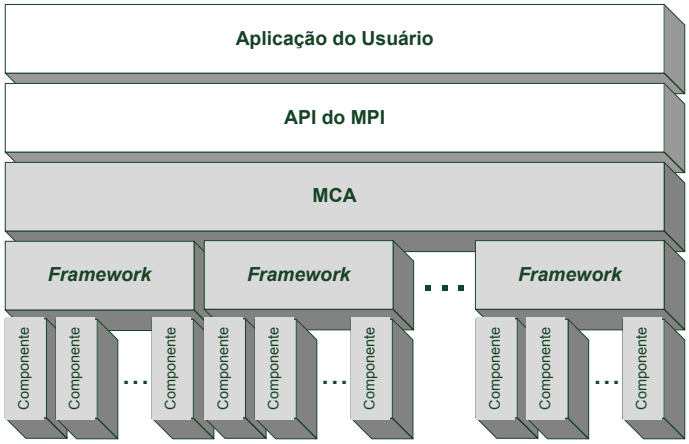

Figura 4.1: MCA (*Modular Component Architecture*)

cursos de processamento disponíveis no ambiente de execução. Além disso, também é responsável pelo disparo das tarefas, monitoramento do seu estado e tratamento de erros. Nos *frameworks* da classe OMPI são implementadas as primitivas da biblioteca do MPI e é onde se encontra o *framework* PML (*Point-to-Point Message Layer*), responsável pelas primitivas de comunicação ponto a ponto.

A Figura 4.2 apresenta os componentes do *framework* PML que trabalham sobre outros *frameworks* que, por sua vez, possuem componentes que realizam a comunicação através das diferentes tecnologias de interconexão. Os blocos em branco representam os *frameworks* e os blocos em cinza os respectivos componentes. Por exemplo, conforme a figura, o componente OB1 do *framework* PML faz interface com o *framework* BTL (*Byte Transfer Layer*), que possui uma série de componentes, cada qual implementando a comunicação em uma diferente tecnologia de interconexão. Os quatro principais componentes PML são o OB1, o DR, o V e o CM [1].

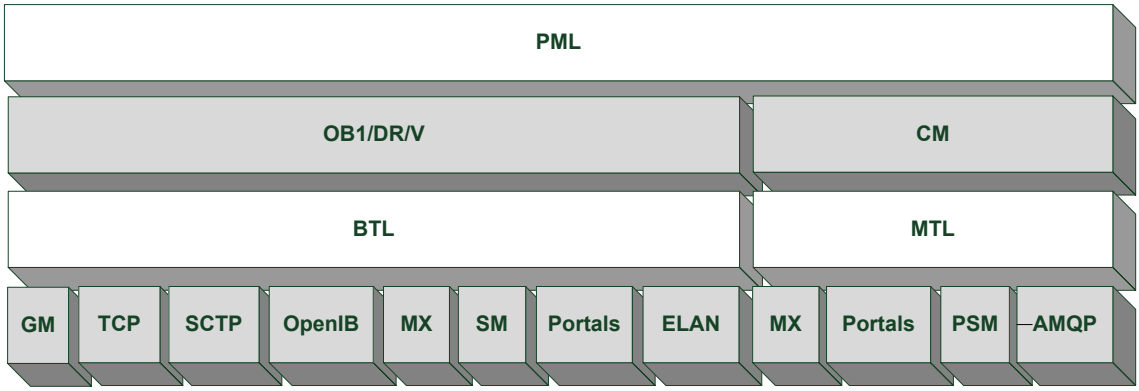

Figura 4.2: PML (*Point-to-Point Message Layer*)

[1]Os nomes dos componentes OB1 e CM têm significados alusivos a personagens dos filmes Star Wars e Highlander. Respectivamente, os personagens Obi-Wan Kenobi e Connor MacLeod. O primeiro não tem significado especial e o segundo é uma metáfora para a restrição de ser possível ativar somente um componente no CM, ou seja, só pode haver um.

O OB1, o DR e o V implementam reconhecimento de mensagens e transferência de dados, e trabalham sobre o *framework* BTL como interface para as tecnologias de interconexão. O componente OB1 oferece comunicação de alto desempenho e é capaz de utilizar os recursos de acesso direto à memória (RDMA) disponíveis em algumas redes. O componente DR e o componente V, implementam mecanismos de tolerância a falhas, conforme explicado na Seção 2.4.3.

O componente CM gerencia as requisições de comunicação ponto-a-ponto e trabalha sobre o *framework* MTL (GRAHAM et al., 2007) (*Matching Transport Layer*), que é responsável pelo reconhecimento de mensagens e transferência dos dados. O MTL foi projetado especificamente para redes como a Portals (RIESEN et al., 2008) e a Myrinet MX (GEOFFRAY, 2004) que são capazes de implementar reconhecimento de mensagens na sua própria biblioteca de comunicação. Só é possível utilizar um componente MTL por aplicação enquanto que é possível utilizar mais de um componente BTL.

A implementação consiste, em sua maior parte, de um componente do *framework* MTL. O OB1, da forma como foi projetado, assume que os blocos das camadas inferiores oferecem somente as funcionalidades de transferência de bytes, sendo necessário implementar o reconhecimento de mensagens no próprio BTL. Já o CM, assume que o MTL consegue tratar mensagens, e delega algumas tarefas como reconhecimento de cabeçalhos e confirmação de entrega de mensagens para os componentes desse *framework*. No caso da implementação realizada, o próprio componente MTL sobre MOM realiza essas tarefas.

4.2 AMQP e OpenAMQ

Após estudos preliminares sobre *middlewares* de comunicação (SANTOS MACHADO, 2007), foi definido que seria utilizado um MOM da especificação AMQP (*Advanced Message Queuing Protocol*) (KRAMER, 2009) para realizar a comunicação do Open MPI. Essa especificação foi criada a fim de permitir a interoperabilidade entre as diversas implementações de MOM que até então, em sua maioria, seguiam a especificação do JMS (GIOTTA et al., 2000). No JMS, somente a API das primitivas é prevista, ou seja, somente a interface de programação para o usuário. A principal lacuna do JMS, utilizada como motivação para o surgimento do AMQP, é a inexistência de uma definição de protocolo de comunicação entre os MOMs. Dessa forma, as diferentes implementações do JMS, de diferentes fabricantes, não são necessariamente compatíveis entre si.

Neste trabalho foi utilizada a implementação OpenAMQ (iMatix Corporation, 2010) da especificação AMQP. No entanto, qualquer outro MOM que pudesse trabalhar com o modelo *Publish/Subscriber* na linguagem C poderia ser utilizado sem perda de generalidade da proposta. Esse *middleware* mostrou-se bastante apropriado devido à sua relativa estabilidade e por possuir código aberto. A versão utilizada é a 1.4.2b.

A semântica de operação dos AMQP se baseia em três tipos de elementos responsáveis pelo tratamento das mensagens:

- **Exchange** Recebem mensagens de aplicações anunciantes e encaminham essas mensagens para filas com base em critérios arbitrários.
- **Filas** São filas do tipo FIFO que armazenam as mensagens até que elas possam ser seguramente processadas pelas aplicações consumidoras.
- **Bindings** Definem as relações entre as filas de mensagens e as *exchanges*, ou seja, os critérios de encaminhamento.

A Figura 4.3 ilustra como esses elementos se relacionam. No lado esquerdo estão os processos clientes que publicam mensagens nos *brokers* através de conexões com as *exchanges*. No lado direito estão os processos clientes que consomem mensagens através de conexões com as suas filas. O servidor de MOM, ou *broker*, agrega várias instâncias dos elementos que tratam as mensagens. A configuração desses elementos é que define o comportamento de como as mensagens são encaminhadas, armazenadas e consumidas. Cada operação de publicação de mensagem recebe como parâmetro uma *routing-key*, que nada mais é do que um nome utilizado pelos *bindings* como critério para encaminhamento das mensagens.

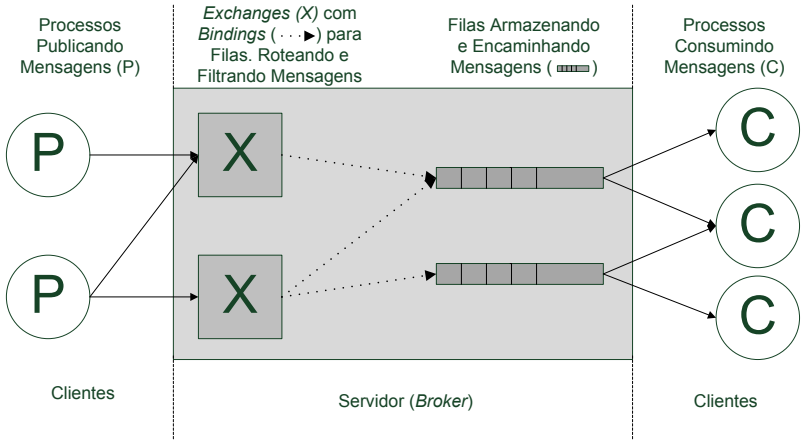

Figura 4.3: Elementos responsáveis pelo encaminhamento das mensagens no AMQP

Por mensagens, entende-se os dados da aplicação que trafegam entre clientes e *brokers*. Para o transporte entre o clientes e o *brokers* as mensagens são divididas em *frames*[2]. Cada *frame* transportado possui um cabeçalho que identifica o tamanho da área de dados. A especificação não prevê limites de tamanho para o *frame* e dá liberdade para as implementações utilizarem ou não a divisão das mensagens em mais de um *frame*.

As filas são os principais elementos do AMQP e são utilizadas para armazenamento e distribuição das mensagens. Uma fila pode ter várias propriedades, entre elas as mais importantes são:

- **Exclusiva** Indica se uma fila pode ou não ser usada por outros processos além do criador.
- **Durável** Indica se uma fila possui persistência de mensagens em disco.
- **Auto-Remoção** Indica se a fila deve ser destruída quando não houver mais clientes.

As *exchanges* são os canais de publicação das mensagens, cada processo publica suas mensagens em uma *exchange* que, por sua vez, fica responsável por realizar a classificação e encaminhamento das mensagens através das configurações dos *bindings*. Cada *exchange* pode ser de um dos tipos a seguir:

- **Direct** Encaminha as mensagens para uma única fila de mensagens (modelo de comunicação ponto a ponto).

[2]Esses *frames* fazem parte do MOM, portanto fazem parte da camada de aplicação OSI. Não devem ser confundidos com os *frames* da camada de enlace OSI.

- **Topic** Encaminha as mensagens para todas as filas com clientes inscritos no tópico da *exchange* (modelo de comunicação *Publish/Subscriber*).

- **Header** Critérios de consulta mais flexíveis baseadas na *routing-key* utilizada na publicação da mensagem.

Todos esses elementos podem ser configurados em tempo de execução no OpenAMQ, permitindo que se estabeleçam os canais de comunicação do componente que mapeia o MPI para o MOM. Seguem dois trechos de código para ilustrar como funciona a programação da comunicação de processos com o OpenAMQ. O trecho de código apresentado na Figura 4.4 mostra como se faz o empacotamento e a publicação de uma mensagem em uma fila.

```
// Cria nova mensagem para ser publicada
content = amq_content_basic_new();
amq_content_basic_set_body (
    content,            // estrutura da mensagem
    msg_body,           // conteúdo a ser enviado
    strlen (msg_body),  // tamanho do buffer
    free);              // liberar buffer após a publicação

// Publica mensagem
amq_client_session_basic_publish (
    session,            // estrutura da sessão com o broker
    content,            // mensagem a ser publicada
    0,                  // parâmetro não utilizado
    "amq.topic",        // exchange
    "/stock",           // routing-key
    FALSE,              // publicação mandatória
    FALSE);             // publicação imediata
}
```

Figura 4.4: Empacotamento e Publicação de Mensagem em Filas no OpenAMQ

A função `amq_content_basic_set_body` recebe como parâmetro um *buffer* de dados e empacota em uma estrutura `content` que é passada como parâmetro para a função de publicação, `amq_client_session_basic_publish`. Essa função também recebe como parâmetro a estrutura da sessão que deve ser previamente iniciada com o *broker*, a *exchange* na qual o processo vai publicar a mensagem e a *routing-key* que será utilizada pela *exchange* para encaminhamento da mensagem. No exemplo, foi utilizada a *exchange* padrão do OpenAMQ para trabalhar com tópicos de mensagens no modelo *Publish/Subscriber* (`amq.topic`). Além disso, os dois últimos parâmetros definem o comportamento da primitiva em dois casos de exceção. O penúltimo parâmetro define se a publicação é mandatória, que, no caso, significa que a *exchange* retornará a mensagem para o cliente se não conseguir encaminhá-la imediatamente. O último parâmetro indica se a publicação é imediata, que quer dizer que a mensagem retornará para o cliente caso não possa ser consumida imediatamente da fila na qual for publicada.

O trecho de código da Figura 4.5 ilustra um processo consumidor de mensagens. A função `amq_client_session_queue_declare` é utilizada para a criação das filas. Nessa função que são definidas as características da fila (durável, exclusiva e com auto-remoção). O quarto parâmetro indica se a função fará uma declaração de fila de forma passiva. No OpenAMQ, isso quer dizer que a fila não será criada de fato, mas apenas que essa estrutura de sessão tratará as mensagens dessa fila que já deverá estar previamente criada no *broker*. A função `amq_client_session_queue_bind` faz o vínculo de uma *routing-key* com a fila na *exchange*. Após a realização dessa operação as mensagens recebidas pela *exchange*, que possuam a *routing-key* especificada, serão armazenadas na respectiva fila. Para definir que o processo atual irá consumir mensagens dessa fila utiliza-se a função `amq_client_session_basic_consume`. O parâmetro `no-local` indica que as mensagens publicadas nessa fila, que sejam originadas do próprio processo, não serão consumidas por esse processos. Isso é útil pois muitas vezes não há utilidade em um processo enviar uma mensagem para si mesmo. O penúltimo parâmetro (consumo exclusivo) deve ser o mesmo que foi utilizado na declaração da fila.

Para consumo das mensagens utiliza-se as funções `amq_client_session_wait`, `amq_client_session_basic_arrived` e `amq_client_session_get`. A primeira função consome todas as mensagens recebidas até o momento na fila e as armazena em uma lista. O segundo parâmetro dela indica o número de milissegundos que a função aguardará pela mensagem na fila. Para indicar uma espera infinita utiliza-se o valor 0, como apresentado no trecho de código. Para realizar apenas *polling* na fila utiliza-se o valor -1 nesse parâmetro. Essa função é utilizada em conjunto com `amq_client_session_basic_arrived` que tem a tarefa de recuperar uma por uma das mensagens recebidas. Dessa maneira, se forem recebidas várias mensagens, seria necessário realizar uma chamada dessa função para cada uma delas. Outra forma de receber mensagens mais simples é com a função `amq_client_session_get`. No entanto, esse método não é tão eficiente, pois consome apenas a primeira mensagem da fila a cada chamada da função.

Apesar dos dois trechos de código apresentarem, cada qual, um papel específico (publicador ou consumidor de mensagem), não existe restrição de um processo poder realizar somente um desses papéis. No caso da implementação do componente MTL/MOM, apresentada a seguir, cada processo da aplicação é tanto publicador como consumidor de mensagens para que seja possível realizar os respectivos envios e recebimentos das primitivas MPI.

4.3 Componente MTL/MOM

O componente MTL/MOM é o elemento central desenvolvido para implementar a presente proposta. Esse componente implementa a API do *framework* MTL utilizando comunicação *Publish/Subscriber* através do MOM OpenAMQ. Essa implementação permite que seja realizada troca de mensagens com as características de desacoplamento temporal e espacial descritas na Seção 3.1. Para a explicação do funcionamento da operação do componente MTL primeiramente será mostrado como é o ciclo de vida de um componente do Open MPI. A Figura 4.6 apresenta todas as etapas na ordem que acontecem, de cima para baixo.

Os blocos em cinza representam as etapas que são executadas pelo MCA enquanto os blocos em branco são funções implementadas no próprio componente. Cada componente

```
// Cria uma fila
amq_client_session_queue_declare(
    session,        // estrutura da sessão com o broker
    0,              // parâmetro não utilizado
    "stockqueue",   // nome da fila
    FALSE,          // passiva
    FALSE,          // durável
    TRUE,           // exclusiva
    TRUE,           // auto-remoção
    NULL);          // parâmetro não utilizado

// Cria binding da fila com a exchange amq.topic
amq_client_session_queue_bind(
    session,        // estrutura da sessão com o broker
    0,              // parâmetro não utilizado
    "stockqueue",   // nome da fila
    "amq.topic",    // exchange
    "/stock",       // routing-key
    NULL);          // parâmetro não utilizado

// Conecta na fila para receber mensagens
amq_client_session_basic_consume(
    session,        // estrutura da sessão com o broker
    0,              // parâmetro não utilizado
    "stockqueue",   // nome da fila
    NULL,           // parâmetro não utilizado
    TRUE,           // no-local
    TRUE,           // parâmetro não utilizado
    TRUE,           // exclusiva
    NULL);          // parâmetro não utilizado

// Recupera todas as mensagens recebidas ou aguarda
// indefinidamente pela chegada de mensagens
amq_client_session_wait (session, 0);

// Recupera a primeira das mensagens recebidas
content = amq_client_session_basic_arrived(session);

// Recupera buffer da estrutura de mensagem
amq_content_basic_get_body(
    content,
    (byte*) msg_body,
    sizeof(msg_body));
```

Figura 4.5: Criação, Vínculo e Consumo de Mensagens de uma Fila no OpenAMQ

é uma biblioteca de vínculo dinâmico que é carregada pelo MCA. Após o carregamento dos componentes existe uma etapa que define quais deles serão selecionados para a exe-

Figura 4.6: Ciclo de vida de um componente MCA

cução da aplicação. Essa seleção é baseada nos parâmetros de MCA dos arquivos de configuração e variáveis de ambiente do Open MPI. No caso do componente não ser selecionado, o MCA passa para a etapa de fechamento e descarrega a respectiva biblioteca.

Após a seleção dos componentes, as funções de inicialização implementadas são chamadas automaticamente pelo MCA e servem para configurar o componente para a correta operação das suas funcionalidades. As primitivas do componente implementadas para o *framework* são empregadas efetivamente na etapa de utilização. As etapas de finalização servem para executar operações que possam ser necessárias após a utilização do componente. Finalmente, o MCA fecha a biblioteca de vínculo dinâmico. Para a maior parte dos componentes essa etapa acontece no final da execução da aplicação do Open MPI.

Serão abordadas agora as etapas de abertura e inicialização do componente MTL/MOM. Em seguida, será detalhada a implementação da comunicação através do componente, ou seja, a etapa de utilização.

4.3.1 Carregamento e Inicialização do Componente MTL/MOM

A seleção do *framework* e dos respectivos componentes de comunicação que serão carregados é realizada em tempo de execução através dos parâmetros MCA do Open MPI. Esses parâmetros podem ser configurados de várias maneiras descritas na documentação do Open MPI. Uma das formas é a utilização de um arquivo de configuração que pode ser passado como parâmetro para o comando `mpirun` responsável pelo disparo dos processos da aplicação. Um arquivo de configuração típico para configurar o componente deve conter os seguintes parâmetros:

```
pml=cm
mtl=mom
mtl_mom_server=nome_ou_ip_do_broker
```

Observa-se que no arquivo deve-se informar o componente do *framework* PML e o componente do *framework* MTL. Esses parâmetros são utilizados na etapa de seleção do componente. Além disso, deve-se informar qual é o servidor de *broker* através do parâmetro `mtl_mom_server` para a etapa de inicialização do componente. Atualmente, a implementação do componente permite apenas um servidor de *broker*. Logo,

o mesmo parâmetro deve ser utilizado em todos os arquivos de configuração dos processos da aplicação MPI.

A Figura 4.7 mostra os passos da inicialização do módulo MTL/MOM. A figura está separada em duas partes por uma linha tracejada. No lado esquerdo estão os elementos que fazem parte do cliente e no lado direito os que fazem parte do servidor. No lado do cliente existem duas *threads*, uma delas é a *thread* principal do Open MPI que realiza a maioria das tarefas, inclusive o processamento da aplicação de usuário. A outra é a *thread progress* utilizada para fazer o tratamento de requisições do MPI em segundo plano.

Na inicialização, primeiramente o processo se conecta e inicia uma sessão AMQP no *broker* OpenAMQ[3] através das chamadas a `amq_connect` e `amq_session`. Com a sessão criada, o processo cria uma fila de mensagens na qual receberá as mensagens endereçadas para si (`amq_queue_declare`).

O tipo de fila criada é durável, para que seja utilizada persistência de mensagens em mídia confiável. Assim, se o *broker* sofrer uma falha, as mensagens recebidas até então serão mantidas e disponibilizadas nas filas na próxima inicialização do *broker*. Também foi utilizada fila sem auto-remoção. Isso é particularmente importante para a presente proposta de tolerância a falhas para que a fila permaneça no *broker* mesmo que não hajam clientes consumindo mensagens dela. Um exemplo de situação em que isso acontece é durante a falha de um processo da aplicação. Nesse caso, o *broker* detectaria através de *heartbeat* que não existe nenhum cliente na fila e, se a fila estivesse configurada para auto-remoção, ela seria removida junto com todas as suas mensagens.

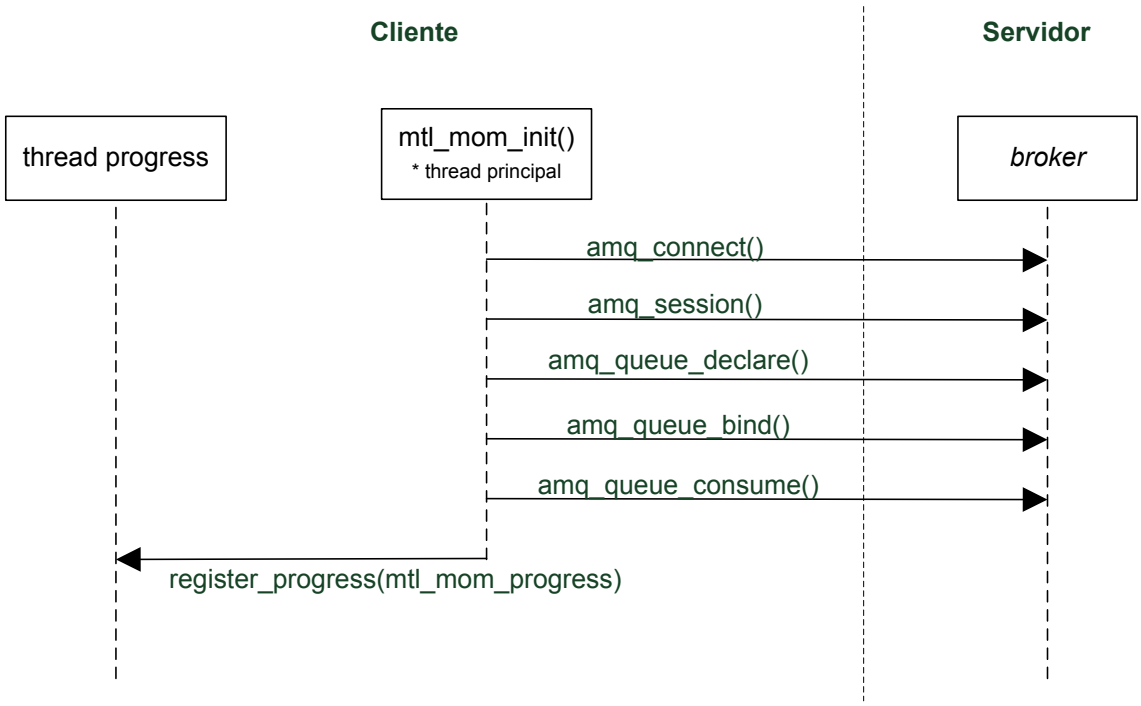

Figura 4.7: Inicialização do componente MTL/MOM

O *binding* do AMQP é feito através da chamada `amq_queue_bind`. Esse procedimento recebe como parâmetros a fila recém criada, a *exchange* padrão de tópicos

[3]No AMQP, múltiplas sessões podem compartilhar a mesma conexão para utilizar o mesmo controle de *heartbeat* do *broker* para o cliente. Dessa forma, não é necessário um *heartbeat* por sessão, basta um controle de *heartbeat* por conexão.

(`amq.topic`) e a *routing-key*. Após a execução da operação, as mensagens encaminhadas para a *exchange* que tenham a mesma *routing-key* informada passam a ser armazenadas na fila.

As *routing-keys* utilizadas para os tópicos são os próprios números de *rank* dos processos. Dessa forma, não é necessário que os processos conheçam nenhuma informação adicional, além do *rank* dos processos destinatários, para enviar mensagens. Para que um processo envie uma mensagem para um outro processo, basta publicar uma mensagem no tópico cuja *string* é igual ao número do *rank* do processo destino.

Assim, o módulo permite trabalhar com comunicação no modelo *Publish/Subscriber* e garante o desacoplamento temporal através das filas, já que não é necessário que as partes da comunicação estejam disponíveis simultaneamente. Também garante o desacoplamento espacial, com os tópicos, já que não é necessário que os remetentes conheçam o endereço dos destinatários, apenas o número do *rank*.

Para que o processo possa receber as mensagens, é necessário vincular-se à fila criada através da chamada `amq_consume`. Depois disso, é possível receber mensagens através da chamada `amq_wait` que busca da fila todas as mensagens recebidas até então.

O último passo da etapa de inicialização do componente registra uma função que é chamada pela *thread progress* do Open MPI para tratar requisições MPI pendentes. As primitivas `*_progress()` são chamadas continuamente enquanto houver requisições pendentes no seu componente. O funcionamento da função `mtl_mom_progress` é explicado na seguinte seção que descreve a API de comunicação do MTL/MOM.

4.3.2 API de Comunicação MTL

A API MTL é utilizada pela camada PML para implementar as primitivas de comunicação do Open MPI. A Figura 4.8 apresenta as principais estruturas de dados e funções do componente implementado.

As funções responsáveis pela comunicação do componente MTL estão agrupadas na figura de acordo com a *thread* do Open MPI na qual são executadas. As rotinas responsáveis pela inicialização e finalização do módulo foram omitidas. As funções da API MTL, `mtl_mom_{irecv,isend,send,iprobe}`, no agrupamento do lado esquerdo, são executadas pela *thread* principal do Open MPI, enquanto a função `mtl_mom_progress` é executada pela *thread* `progress`.

Essas funções utilizam duas listas circulares. Uma delas para gerenciar as requisições pendentes criadas por `mtl_mom_isend` e `mtl_mom_irecv`, e a outra para armazenar as mensagens recebidas que ainda não foram repassadas para o PML. Como a função `mtl_mom_progress` é executada em uma *thread* diferente das funções da API MTL, foram utilizados *mutexes* para garantir o acesso exclusivo às listas circulares.

A primeira lista, `pending_requests`, armazena as requisições `isend` e `irecv` que estão pendentes. A cada chamada `mtl_mom_isend` ou `mtl_mom_irecv` realizada, uma entrada na lista é criada e o controle do processo é devolvido imediatamente para a computação da aplicação. A função `mtl_mom_progress` verifica as requisições MTL que estão pendentes e as completa se possível. Isso permite que as funções `mtl_mom_isend` e `mtl_mom_irecv` sejam executadas concorrentemente com a computação da aplicação.

A chamada a `mtl_mom_progress` é executada continuamente, enquanto houver requisições pendentes. Se após uma chamada ainda houver requisições pendentes então o Open MPI chama novamente a função, até que não existam mais requisições. O trecho de código resumido da Figura 4.9 mostra o funcionamento básico da função

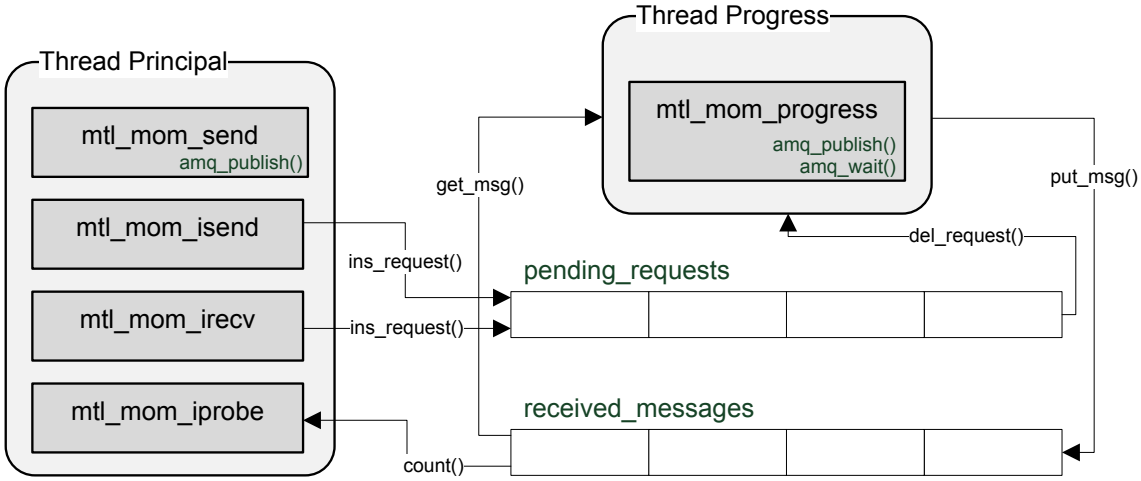

Figura 4.8: Operação do módulo MTL/MOM

`mtl_mom_progress`. O tratamento de erros, parâmetros de funções, tratamento de *tags* de mensagens MPI e outras informações de menor relevância para compreensão do princípio de funcionamento foram omitidas.

A cada chamada, `mtl_mom_progress` verifica a última requisição MTL pendente (`ompi_mtl_mom.pending_head`) na lista circular. As requisições MTL estão associadas a requisições PML, que por sua vez, estão associadas a requisições da própria aplicação MPI. Por exemplo, quando um usuário executa uma chamada a `MPI_Irecv`, essa chamada, ao chegar ao PML, através da respectiva função, `pml_cm_irecv`, criará uma requisição PML. Da mesma maneira, quando a função `pml_cm_irecv` faz a chamada para sua função correspondente do MTL, `mtl_mom_irecv`, essa função cria uma requisição MTL. Assim que as requisições da camada mais baixa são atendidas, deve-se informar as camadas superiores para que atualizem as respectivas requisições. O PML é avisado que a requisição MTL foi finalizada através de uma função da camada PML que pode ser referenciada na própria camada MTL através da chamada `req->super.completion_callba`

Se a requisição MTL é um `isend` então ela é atendida imediatamente através de uma chamada a `amq_publish` que faz a publicação da respectiva mensagem no *broker*. Se a requisição for um `irecv` então, primeiramente, é verificada a segunda lista circular, `received_messages`, que armazena as mensagens recebidas. A função `get_msg()` foi criada para varrer essa lista em busca de mensagens que casem com a requisição. Essa função recebe o *rank* de origem da mensagem e a sua *tag* MPI como critérios de procura. Se a mensagem já estiver disponível na lista, a função copia os dados da mensagem para o *buffer* da requisição através da função `amq_get_body`, remove a requisição MTL da fila com `del_request` e por fim notifica a camada PML.

Se a mensagem ainda não tiver sido recebida, a função realiza uma chamada a `amq_wait` que recupera todas as mensagens disponíveis na fila do *broker*. As mensagens recuperadas são armazenadas na lista circular, na mesma ordem em que são recebidas, para que sejam entregues à aplicação efetivamente nas próximas chamadas de `mtl_mom_progress`.

A função `mtl_mom_send` não cria estrutura de requisição e apenas faz uma chamada a `amq_publish`. A função de `mtl_mom_iprobe` simplesmente testa se existem mensagens pendentes na lista `received_messages` e retorna o resultado da requisição para o PML sem necessidade da função de *progress*.

Como pode-se notar, algumas funções não possuem mapeamento direto no *frame-*

```
mtl_mom_progress(...) {
  ...
  struct
    mca_mtl_mom_request_t *req = ompi_mtl_mom.pending_head;
  if (req != NULL) {
    if (req->type == OMPI_MTL_MOM_ISEND) {
      /* publica mensagem da requisição */
      amq_publish(...);
      /* remove requisição MTL da lista */
      del_request(...);
      /* avisa PML que a requisição MTL foi completada */
      req->super.completion_callback(&req->super);
    } else if (req->type == OMPI_MTL_MOM_IRECV) {
      if (rcvd = get_msg(...)) {
        amq_get_body(rcvd->content, req->buf);
        del_request(...)
        /* avisa PML que a requisição MTL foi completada */
        req->super.completion_callback(&req->super);
      /* recebe novas mensagens e armazena na fila */
      } else {
        amq_wait(..., 1) /* timeout de 1 ms */
        while(amq_arrived(...)) {
          put_msg(...)
    }
  }
  ...
}
```

Figura 4.9: Funcionamento básico da função `mtl_mom_progress`

work MTL. A função `MPI_Send`, possui a equivalente PML `pml_cm_send`, que é executada no MTL pela função `mtl_mom_send`. No entanto, o mesmo não acontece, por exemplo, com a função `MPI_Recv`. Essa função, apesar de possuir a equivalente PML, `pml_cm_recv`, não possui uma no MTL. Mesmo assim, o PML/CM gerencia a camada do MTL de forma que o conjunto de funções previsto na API MTL é suficiente para emular todas as demais funções. A função `pml_cm_recv`, por exemplo, é realizada através da função `mtl_mom_irecv`.

4.3.3 Comunicação Coletiva

As primitivas de comunicação coletiva do Open MPI possuem um *framework* específico para implementação de seus componentes. O *framework* COLL possui vários componentes e apenas um deles é selecionado durante a execução do `mpirun` para tratar as chamadas de comunicação coletiva. A maioria desses componentes utiliza o *framework* de comunicação ponto-a-ponto PML como base.

Praticamente todos os componentes COLL do Open MPI trabalham sobre PML. O único componente do *framework* que possui uma versão otimizada é o SM (*shared memory*). Os demais variam o tipo de otimização sobre a utilização das primitivas PML para

realizar a comunicação coletiva sobre comunicação ponto-a-ponto. No presente trabalho, não foi implementado nenhum componente do *framework* COLL que realize a comunicação otimizada pelo MOM. Logo, a comunicação coletiva se dá sobre primitivas de comunicação ponto-a-ponto. Como apresentado na Seção 3.2, acredita-se que a implementação da comunicação coletiva com difusão em múltiplos tópicos possa otimizar o envio de mensagens de primitivas como `MPI_Bcast` e `MPI_Allreduce`. No entanto, tal implementação foi deixada para trabalhos futuros.

Para que o PML possa identificar se as requisições das camadas superiores são para primitivas de comunicação coletiva ou primitivas de comunicação ponto-a-ponto, são utilizadas *tags* MPI especiais nas mensagens. As *tags* que o usuário da aplicação pode utilizar para classificar as suas mensagens ponto-a-ponto são sempre inteiros positivos. No caso das primitivas de comunicação coletiva utilizam-se inteiros negativos, impossíveis de serem atribuídos pelo usuário. Assim, cada tipo de primitiva de comunicação coletiva de componente do *framework* COLL atribuirá às suas requisições um número negativo de *tag* e o PML terá informação suficiente para identificar as primitivas correspondentes a cada mensagem recebida na aplicação.

4.4 Considerações Finais

Neste capítulo, foram apresentadas as bases do Open MPI e de uma implementação da especificação AMQP, o OpenAMQ, que foram utilizados na comunicação desacoplada para o MPI utilizando MOM. Essa implementação é parte da proposta de tolerância a falhas através de log de mensagens com *checkpoint* não-coordenado.

Foi apresentado o funcionamento básico da arquitetura modular do Open MPI, o MCA. O MCA é baseado no conceito de *frameworks*, que definem APIs internas para a implementação de funcionalidades. Cada *framework* pode ter vários componentes, que implementam as funções que estão definidas na API do seu respectivo *framework*.

Os componentes são, de fato, bibliotecas de vínculo dinâmico que são carregadas e utilizadas pela aplicação MPI. Essas bibliotecas implementam as diversas funcionalidades do Open MPI, como comunicação ponto-a-ponto nas várias tecnologias de interconexão, e até mesmo os temporizadores que serão utilizados pela aplicação.

O trabalho apresentado é um componente MCA do *framework* MTL que realiza a comunicação ponto-a-ponto através do OpenAMQ pelo modelo *Publish/Subscriber* publicando e consumindo mensagens em tópicos. O *framework* MTL é utilizado pelo componente CM do *framework* PML, que serve para as primitivas de comunicação ponto-a-ponto do usuário do MPI. O PML também é utilizado na maioria dos componentes do *framework* COLL, responsável pela interface de comunicação coletiva. Inclusive, para a comunicação coletiva da presente implementação, é necessário utilizar o *framework* PML, o que quer dizer que as primitivas de comunicação coletiva trabalham sobre primitivas de comunicação ponto-a-ponto.

A implementação desenvolvida é parte da proposta do Capítulo 3. Para completar a arquitetura proposta ainda resta finalizar a integração do *checkpoint* não-coordenado e do log de mensagens. Para isso, é necessário criar o sistema de geração e armazenamento de *checkpoints* periódicos em mídia confiável remota, e o log de mensagens pessimista remoto. Também são necessários os mecanismos de detecção de processos falhos, e a recuperação de *checkpoint* e log de mensagens nos casos de falha.

Além disso, a proposta também prevê a comunicação coletiva através da publicação de mensagens em múltiplos tópicos, que é possível no modelo *Publish/Subscriber*. Essa

implementação poderia ser realizada para otimização do desempenho da comunicação via OpenAMQ.

No próximo capítulo, serão apresentadas as medidas e a análise de desempenho realizadas com algumas aplicações de *benchmark*. As aplicações utilizadas são o NetPIPE, parte do NAS Parallel Benchmarks e o Virginia Hydrodynamics.

5 RESULTADOS E AVALIAÇÃO DE DESEMPENHO

Este capítulo apresenta os resultados obtidos para a avaliação da comunicação desacoplada sobre OpenAMQ implementada para o Open MPI (MTL/MOM) a ser utilizada como base de uma solução de *checkpoint* não-coordenado com log de mensagens remoto. Sabe-se de antemão que a solução implementada implicaria em uma latência de, no mínimo, o dobro de uma comunicação normal, já que são necessárias duas cópias de mensagem pela rede, mas esse é o preço a ser pago para se permitir o desacoplamento temporal. O objetivo deste capítulo é avaliar esse sobrecusto.

As medidas de desempenho foram realizadas também em dois componentes da distribuição do Open MPI para fins de comparação. O primeiro é a implementação tipicamente utilizada para comunicação TCP/IP (BTL/TCP). O segundo é um componente utilizado para comunicação de suporte para a realização de *checkpoint* não-coordenado com log de mensagens pessimista local nos processos remetentes (BTL/V), apresentado na Seção 2.1.3.2.

O mecanismo de comunicação das implementações de comunicação foi medido através de *micro-benchmarks*, *macro-benchmarks* e uma aplicação real. *Os micro-benchmarks* nos permitem estimar as limitações e o potencial da abordagem proposta analisando características pontuais como a latência da comunicação entre dois processos. O *micro-benchmark* utilizado foi o NetPIPE (SNELL; GUSTAFSON, 1996). Os *macro-benchmarks* são pequenas aplicações que simulam o comportamento típico de aplicações reais. Para cada um deles, existe um comportamento predominante, que é utilizado para verificar o desempenho dos sistemas na execução de cálculos específicos. No caso das medidas aqui apresentadas, procurou-se utilizar *benchmarks* com comportamentos diversos no que se refere aos seus padrões de comunicação/computação (FARAJ, 2002), com um subconjunto do NAS Parallel Benchmarks (NPB) (BAILEY et al., 1991). A aplicação real utilizada, o Virginia Hydrodynamics (VH-1, 2010), possui um padrão de comunicação/computação mais complexo que os demais testes.

Os testes foram realizados no cluster ICE do Instituto de Informática da UFRGS cuja plataforma é descrita a seguir. Depois, serão apresentados, a metodologia empregada na elaboração dos testes, os resultados obtidos nas medidas, juntamente com sua análise, e, por fim, as conclusões.

5.1 Plataforma de execução

A plataforma de execução dos testes foi o cluster ICE do Instituto de Informática da UFRGS. Esse cluster é composto por 14 nodos interconectados por dois switches Gigabit Ethernet. As interfaces de interconexão são duas placas de rede Gigabit Ethernet Broadcom NetXtreme II BCM5708. Cada uma dessas interfaces está conectada em um dos

switches do cluster.

Cada nodo possui 2 chips Intel® Xeon® E5310 (Clovertown) de 1.6GHz. Esses chips possuem 4 núcleos de processamento totalizando 112 núcleos no cluster inteiro. Uma cache L2 de 4MB é compartilhada por cada par de núcleos, num total de 8MB de cache L2 por chip. A quantidade de memória RAM por nodo é de 16GB acessada através de um barramento de 266MHz trabalhando com QDR (*quad data transfer rate*).

5.2 Metodologia Empregada

A metodologia empregada para a realização dos testes de avaliação de desempenho diferem em algumas características dependendo do *benchmark*. Entretanto, existem características que todos os testes realizados compartilham. Os nodos do cluster utilizados foram alocados através do escalonador de tarefas OAR (CAPIT et al., 2005) para execução exclusiva dos testes aqui apresentados, sem concorrência com o processamento de outras aplicações. Cada nodo utilizou o sistema operacional GNU/Linux, distribuição Debian Lenny com kernel versão 2.6.26, compilado com suporte a 64 bits e SMP, e inicializado através do Kadeploy (KADEPLOY, 2010). Como compilador foi utilizado o GCC e o GFortran versão 4.3.2. A distribuição do Open MPI que foi utilizada foi a 1.3.2 e a do OpenAMQ a 1.4b0. As versões dos softwares de *benchmark* utilizadas foram o NPB 2.4, o VH-1 2.0 e o NetPIPE 3.7.1.

Em todos os testes do MTL/MOM foi utilizado um nodo dedicado para o servidor *broker* OpenAMQ. Essa medida foi tomada a fim de evitar que o servidor de *broker* tivesse seu desempenho comprometido por conta da execução processos das aplicações MPI.

Apesar de todos os nodos possuírem 8 processadores cada, procurou-se evitar que mais de um processo executasse no mesmo nodo para minimizar interferências de rede e de acesso à memória. No entanto, em alguns dos testes dos *benchmarks* NPB e VH-1, que utilizam um maior número de processos, isso não foi possível. Como desejava-se averiguar o comportamento da aplicação e do servidor de *broker* executando com um número maior de processos, foi feita a distribuição dos processos nos nodos da forma descrita a seguir.

Essencialmente, foram utilizados apenas 8 nodos do cluster, além do nodo dedicado ao servidor de *broker*. Foi escolhido esse número pois a maioria das aplicações do NPB executam sempre com um número de processos que é potência de 2. Para execuções com até 8 processos, não foi necessário mais de um processo por nodo. Para um número maior de processos seguiu-se a relação de número de nodos, número de processos e processos por nodo descrita a seguir.

Considerando n o número de nodos utilizados para a execução, P_a o número de processos da execução da aplicação e P_n o número de processos da aplicação executando em cada nodo. Sendo P_a múltiplo de 8 e n igual a 8 então segue-se a relação $P_n = P_a/n$ para realizar a distribuição dos processos da aplicação nos nodos. Mantendo P_n igual em todos os nodos pretende-se realizar um balanceamento de carga estático no disparo da aplicação. O valor limite para P_n que foi utilizado foi 8, já que é exatamente o número de processadores disponível em cada nodo do cluster.

Alguns dos testes como o VH-1 e o NPB EP poderiam utilizar outros valores para P_a e n pois não têm a restrição de executar somente com um número de processos que seja potência de 2. No entanto, pelos resultados obtidos e análise apresentada adiante, considerou-se que testes adicionais nesse sentido não acrescentariam dados relevantes para a análise realizada.

Em todos os testes, a memória disponível nos nodos foi suficiente para acomodar a memória alocada pelos processos das aplicações. Mesmo nas execuções com um maior número de processos não houveram problemas de restrição de memória, pois nas aplicações executadas o tamanho da entrada de cada processo é inversamente proporcional ao número de processos da aplicação.

Apesar do cluster disponibilizar duas interfaces de rede por nodo, somente uma foi utilizada em cada nodo. Também não foi utilizada nenhuma técnica que forçasse que cada processo alocasse processadores específicos dos nodos. Essa tarefa foi deixada a cargo do escalonador de processos do sistema operacional. Quanto ao tamanho das mensagens, são consideradas pequenas as mensagens menores ou iguais a 1KB, e grandes as maiores que esse valor.

Nos gráficos de desempenho, são apresentados os resultados correspondentes a três implementações de comunicação do Open MPI. O BTL/TCP é a implementação de comunicação TCP/IP para o Open MPI, e é utilizada como referência nos testes de desempenho. O BTL/V é uma implementação de comunicação para suporte a log de mensagens pessimista local, apresentada na Seção 2.4.3. Esse tipo de log de mensagens é inerentemente menos oneroso do que o remoto, que é utilizado na nossa implementação. Seus testes foram realizados a fim de se obter uma estimativa de quanto cada tipo de log de mensagens impactaria no desempenho. O MTL/MOM é a própria implementação de comunicação através do OpenAMQ apresentada na Seção 4.3. Além desses, os testes de *Ping-Pong* também possuem uma versão que realiza a comunicação diretamente sobre o OpenAMQ. Esse teste foi realizado para analisar o impacto do MOM utilizado no desempenho da implementação.

Os resultados dos testes de *Ping-Pong* com o NetPIPE são as saídas de execuções normais desse *benchmark* via linha de comando. Nesse modo de execução o próprio software, baseado nas métricas de avaliação de desempenho do HINT (GUSTAFSON; SNELL, 1995), determina quantas repetições devem ser realizadas para a média de cada resultado. As medidas apresentadas para os testes do NPB e do VH-1 são uma média do tempo total de execução das aplicações (*walltime*). As médias são de no mínimo 100 execuções e possuem intervalos de confiança de 95% de probabilidade.

5.3 NetPIPE

O NetPIPE (SNELL; GUSTAFSON, 1996) é um software para avaliação de desempenho de comunicação que suporta várias tecnologias de interconexão, alguns *middlewares* de programação paralela e comunicação através de memória compartilhada. Para estimar a banda passante e a latência da comunicação foram utilizados testes de *Ping-Pong* com esse software nos componentes Open MPI e no próprio OpenAMQ, isoladamente. Entende-se por latência o tempo médio necessário para um processo MPI enviar uma mensagem para outro processo MPI da aplicação e por banda passante a média de informação transferida de um processo MPI para outro em um determinado intervalo de tempo. Os testes realizados somente com o OpenAMQ visam estimar qual a parcela do sobrecusto de comunicação da implementação é gerada pelo próprio MOM.

Os *micro-benchmarks* de *Ping-Pong* são importantes para demonstrar o potencial das implementações de comunicação sem maiores interferências. Ou seja, quando não há processamento de aplicações que possam interferir no processamento do tratamento das mensagens.

Na Figura 5.1 e 5.2 estão apresentados os resultados obtidos com as implementações do Open MPI sobre BTL/TCP, BTL/V e MTL/MOM, e a implementação sobre OpenAMQ. No gráfico de banda passante, pode-se observar uma descontinuidade na curva do BTL/TCP e do BTL/V próximo dos 64KB. Esse comportamento se deve a uma mudança na política de tratamento de mensagens no BTL a partir desse tamanho. Os gráficos do MTL/MOM e OpenAMQ não apresentam nenhuma descontinuidade semelhante pois não há distinção no tratamento de mensagens com tamanhos diferentes.

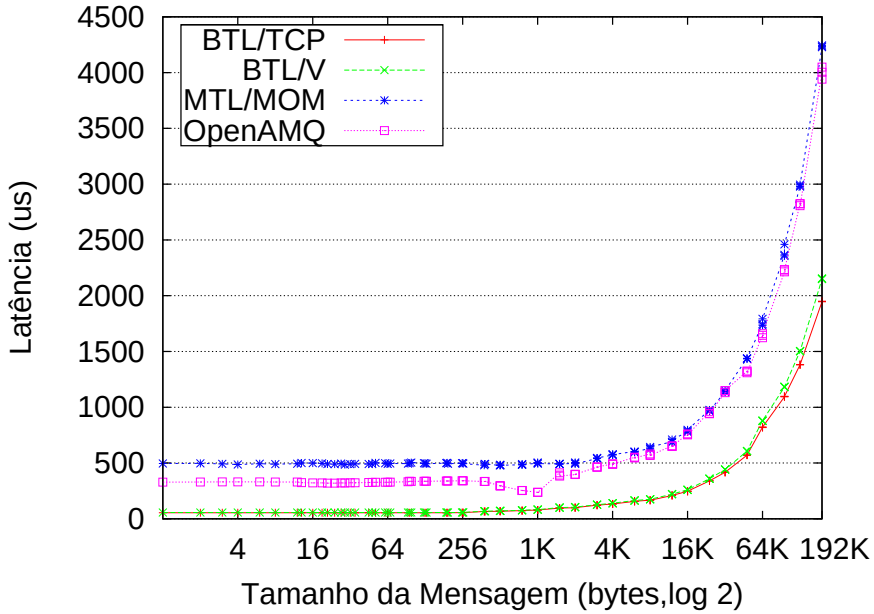

Figura 5.1: Resultados dos testes com o NetPIPE (Latência)

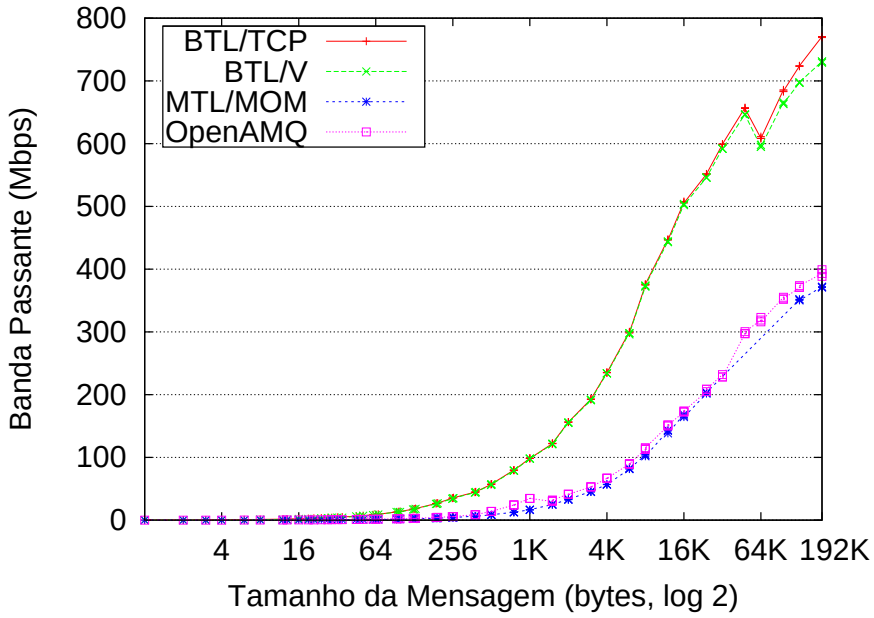

Figura 5.2: Resultados dos testes com o NetPIPE (Banda Passante)

Os resultados obtidos mostram aproximadamente o que era esperado. Em mensagens de 128KB, a banda passante do MTL/MOM (350,111Mbps) é cerca da metade da do BTL/TCP (724,13Mbps) e do BTL/V (698,055Mbps). A latência obtida com o MTL/MOM (2995,05μs) nesse tamanho de mensagem é aproximadamente o dobro da do BTL/TCP (1381μs) e do BTL/V (1502,172μs). Como é necessário que o remetente envie a mensagem para o *broker* e, posteriormente, o destinatário a receba do *broker*, são necessárias na realidade duas transmissões pela rede. Isso justifica essa relação.

Para valores de mensagem menores que 128KB, o sobrecusto do MOM começa a ser bem mais significativo. Por exemplo, para um tamanho de mensagem de 4KB os valores de banda passante são de 56,652Mbps para o MTL/MOM, 235,59Mbps BTL/TCP e 234,773Mbps para o BTL/V. As latências são de 578,40μs para o MTL/MOM, 132,65μs para o BTL/TCPμs e 139,57μs para o BTL/V.

Também verifica-se, comparando os resultados com o MTL/MOM e o OpenAMQ, que, em qualquer tamanho de mensagem, o próprio OpenAMQ é responsável pela maior parcela da latência da implementação. Grosso modo, considerando que o MTL/MOM é uma implementação sobre OpenAMQ, a latência atribuída à camada de implementação do componente MTL/MOM do Open MPI seria a diferença entre os valores de latência do MTL/MOM e os valores do OpenAMQ. O que se observa para todos os valores de mensagem é que essa diferença é relativamente pequena. Por exemplo, em 16KB o OpenAMQ apresenta uma latência de 757,11μs e o MTL/MOM de 787,73μs.

5.4 NAS Parallel Benchmarks

O NPB (*NAS Parallel Benchmarks*) (BAILEY et al., 1991) (BAILEY et al., 1995) é composto por dois tipos de *benchmarks*, os *kernels* e as aplicações simuladas. Os *kernels* (EP, FT, IS, CG e MG) consistem em aplicações que sintetizam, cada qual, um comportamento típico de uma vasta gama de aplicações. As aplicações simuladas (LU, SP e BT), por sua vez, incorporam vários cálculos de dinâmica de fluídos que compõem grande parte de códigos completos congêneres. Os *benchmarks* a seguir foram selecionados tomando como critérios a diversidade do comportamento das aplicações, selecionando os *benchmarks* que possuíam os padrões de comunicação mais distintos.

O NPB divide cada problema em classes que definem o tamanho da entrada dos problemas e, consequentemente, os seus tempos de execução. Foram escolhidas as classes que mais se adequavam para os testes, ou seja, cujos tamanhos de entrada de problema se acomodavam na memória do cluster e os tempos de execução fossem suficientes para realizar a comparação entre as implementações do Open MPI.

5.4.1 Kernel FT

Os algoritmos de transformadas rápidas de Fourier (FFT – *Fast Fourier Transform*) são importantes para solucionar sistemas de equações e são utilizados em várias aplicações científicas. O algoritmo do NPB resolve a equação diferencial parcial parabólica $u_t = \nabla^2 u$ discretizando os dados em uma matriz 3D de tamanho variável, de acordo com a classe do problema. Para a resolução, em cada iteração do algoritmo são executadas 3 fases de transformadas rápidas discretas 1D, para cada dimensão da matriz. Por exemplo, em uma matriz 512x512x512, serão realizadas 512x512 transformadas 1D no eixo x, 512x512 transformadas no eixo y e 512x512 transformadas no eixo z.

A implementação do NAS organiza a matriz 3D como apresenta a Figura 5.3. Essa figura ilustra a matriz 3D em uma aplicação FT com 4 processos (P0 a P3). Como pode-se

observar, uma das dimensões da matriz terá seus elementos espalhados entre os processos da aplicação. No caso, a dimensão do eixo k da figura. Para realizar o algoritmo da transformada 1D de forma efetiva é necessário que todos os elementos da dimensão estejam no mesmo processo. Assim, para realizar a transformada 1D, a aplicação realiza uma transposição da matriz na dimensão na qual os elementos estão espalhados. Essa transposição normalmente é realizada através de operações `MPI_Alltoall` entre os processos da aplicação. Como essa operação é realizada a cada iteração, a demanda de comunicação do algoritmo é bastante intensa e, por isso, ele é muito utilizado para medidas de desempenho de comunicação de sistemas paralelos.

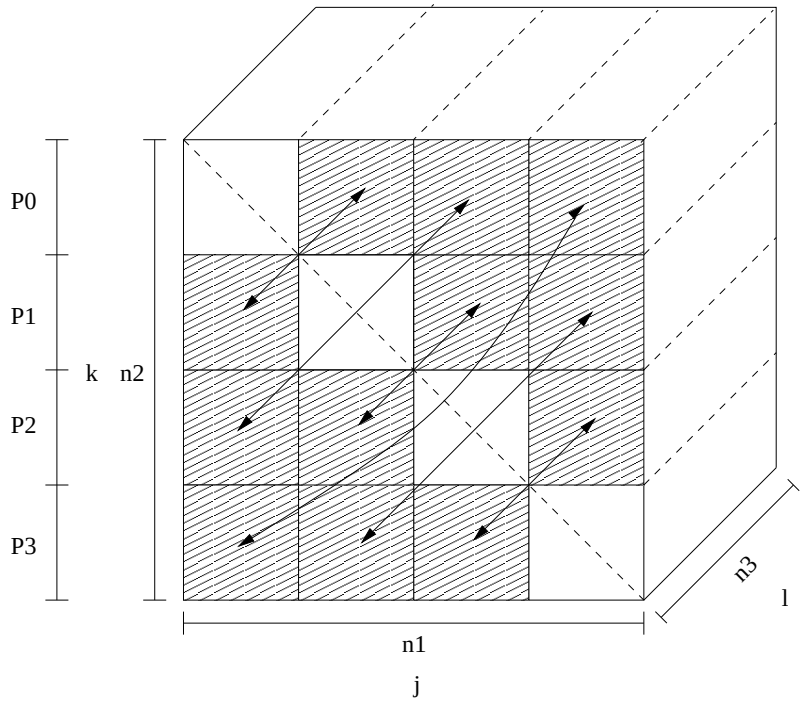

Figura 5.3: Transposição de matrizes com MPI_Alltoall no NPB FT

O gráfico da Figura 5.4 mostra um comparativo do tempo de execução do *benchmark* FT executando com as versões de Open MPI sobre BTL/TCP, BTL/V e MTL/MOM. Para a avaliação de desempenho com esse *benchmark* foi utilizada a classe B da distribuição do NAS que utiliza uma matriz de 512x512x256 números complexos.

Nota-se um desempenho inferior do MTL/MOM, se comparado com as outras implementações, pois esse *benchmark* realiza comunicação intensivamente. O FT do NPB utiliza muitas chamadas `MPI_Alltoall` para transposição de matrizes, cada chamada produzindo $2(n-1)$ trocas de mensagem ponto a ponto, sendo n o número de processos da execução. Também é possível observar que os tempos de execução apresentam uma variância maior para 16, 32 e 64 processos. Suspeitava-se que essa variância maior estava relacionada com uma maior incidência de *cache miss* nesses casos. Após uma análise instrumentando o sistema operacional com a ferramenta OProfile (LEVON, 2004), pôde-se constatar que essas execuções com maior variância acompanhavam um aumento considerável na taxa de *cache miss*, chegando a 57% dos acessos à memória com *cache miss* nas execuções com 64 processos. Estima-se que essa incidência maior seja causada pela intensa cópia de dados desse *benchmark* durante as etapas de transposição das matrizes, e também entre os *buffers* do Open MPI e OpenAMQ durante as trocas de mensagem.

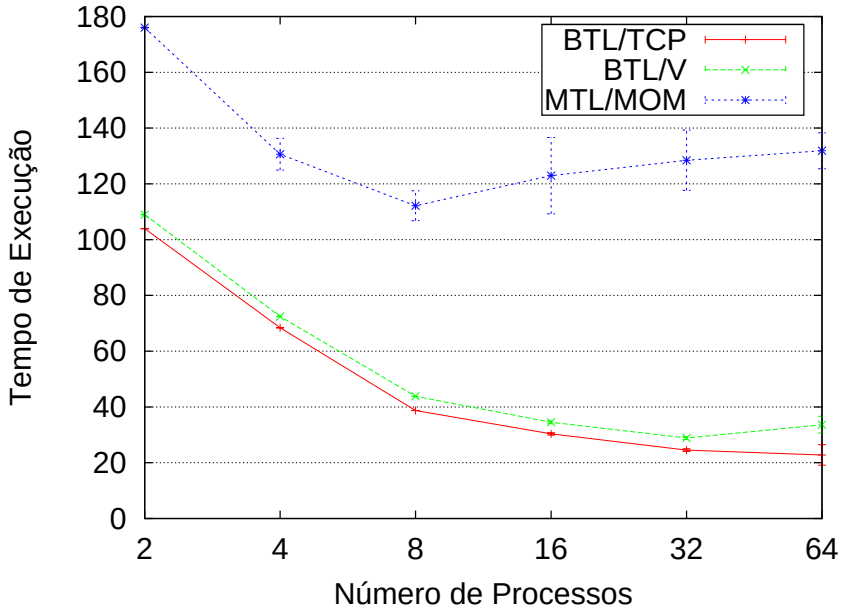

Figura 5.4: Resultados dos testes com o NPB FT (Classe B)

Uma melhoria que poderia ser incorporada é a alteração do OpenAMQ para utilizar diretamente os *buffers* de mensagens do Open MPI em vez de fazer cópias dos *buffers* de mensagens. Essa modificação poderia diminuir a taxa de *cache miss* nos casos de execuções com mais de um processo por nodo.

5.4.2 Kernel CG

O *benchmark* CG (*Conjugate Gradient*) resolve um sistema de equações lineares esparso, $Az = x$, através do método do gradiente conjugado. O CG utiliza o método da potência inversa para calcular uma estimativa do maior autovalor de uma matriz esparsa simétrica positiva definida com um padrão aleatório de valores diferentes de zero.

O comportamento predominante é basicamente uma sequência de iterações executando `MPI_Irecv`, `MPI_Send` e `MPI_Wait`, com mensagens pequenas ($\leq$1KB). O tamanho da entrada do *benchmark* depende de variáveis que definem o tamanho do sistema de equações (n) e o número de iterações do algoritmo ($niter$). Na classe B do *benchmark*, que foi utilizada, os valores são $niter = 75$ e n com 75000 elementos.

A Figura 5.5 faz a comparação do desempenho obtido com as implementações avaliadas. O ponto de menor tempo de execução de todas as implementações é a com 8 nodos, na qual o BTL/TCP leva em média 28,56s, o BTL/V leva 39,66s e o MTL/MOM leva 103.53s. Esse é o caso no qual a diferença de desempenho entre as implementações BTL e a MTL/MOM é mais alta. Entretanto, se forem comparados os desempenhos das implementações BTL com a MTL/MOM no caso de 64 processos nota-se que a diferença se torna menos significativa, visto que os tempos de execução são bem maiores e a diferença não varia tanto. Com 64 processos, os tempos obtidos foram de 267,02 para o BTL/TCP, 278,79 para o BTL/V e 396,92 para o MTL/MOM. Apesar de não haver comunicação tão intensiva como no *benchmark* FT, justifica-se o desempenho obtido pelo fato da aplicação trocar mais mensagens pequenas e o sobrecusto dessas comunicações ter um peso maior no MTL/MOM do que nas implementações de BTL.

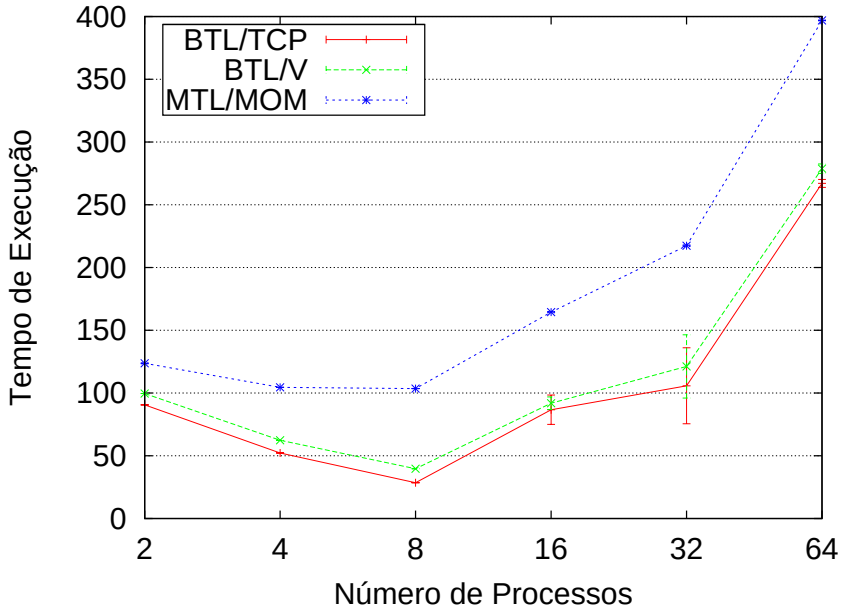

Figura 5.5: Resultados dos testes com o NPB CG (Classe B)

5.4.3 Kernel EP

O *benchmark* EP (*Embarassingly Parallel*) não apresenta comunicação considerável. Como o próprio nome sugere, se trata de um problema trivialmente paralelizável. Nesse *benchmark*, só existe comunicação no final da execução para concentrar os resultados no processo 0 e apresentá-los. O EP gera e tabula $n = 2^m$ pares de desvios gaussianos aleatórios de acordo com um algoritmo específico. O valor de m é 28 para a classe B, que foi utilizada para a avaliação.

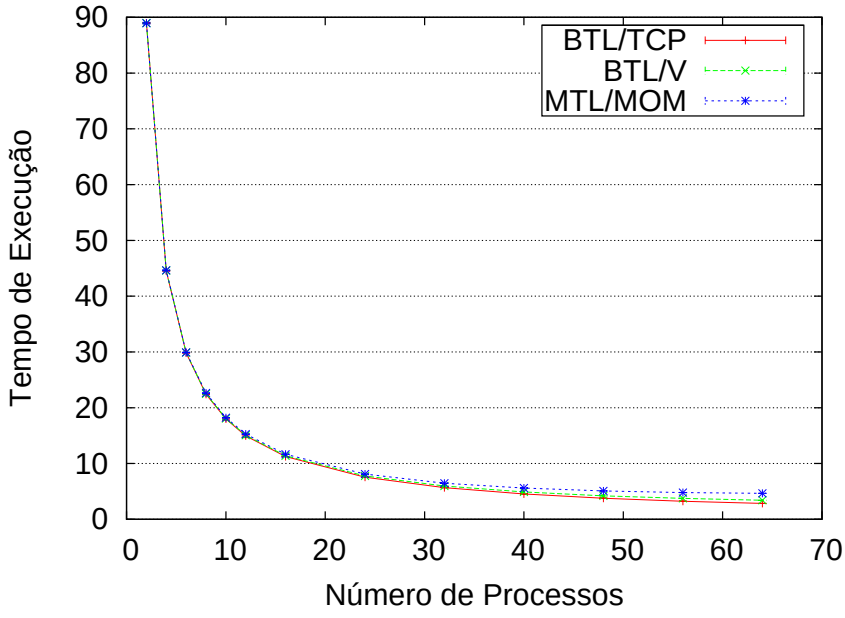

Figura 5.6: Resultados dos testes com o NPB EP (Classe B)

A inexistência de comunicação considerável se reflete nos resultados apresentados na Figura 5.6. Como pode-se observar, o desempenho das implementações é muito próximo e somente com um número de processos maior percebe-se um desempenho mais baixo do MTL/MOM se comparado com as outras implementações. Essa diferença refere-se às rotinas de inicialização que precisam conectar cada processo ao *broker* OpenAMQ. A versão atual só permite um servidor centralizado, logo há uma sobrecarga durante a inicialização que justifica a diferença no gráfico de desempenho quando há um maior número de processos. Como pode-se inferir, o servidor centralizado não escalaria com um número muito grande de processos. A solução proposta de múltiplo servidores para tratamento de tópicos distribuídos, apresentada na Seção 3.2 é a solução para minimizar essa sobrecarga.

5.4.4 Kernel MG

O *benchmark* de MG (*Multigrid*) do NPB consiste em quatro iterações de um algoritmo de multigrade cíclico, com uso do ciclo V, que encontra a solução u do problema de Poisson discretizado $\nabla^2 u = v$. Esse *benchmark* é bastante simples do ponto de vista de comunicação. Cada processo intercala fases de curtas de computação com fases de comunicação que utilizam chamadas a `MPI_Irecv`, `MPI_Send` e `MPI_Wait`, nessa ordem. A maioria das comunicações acontecem entre processos vizinhos que possuem bordas de matrizes adjacentes.

Nesse *benchmark* foi utilizada a classe C do problema, pois o tempo de execução na classe B era muito baixo nas medidas. Essa classe trabalha com uma entrada de 512^3 elementos.

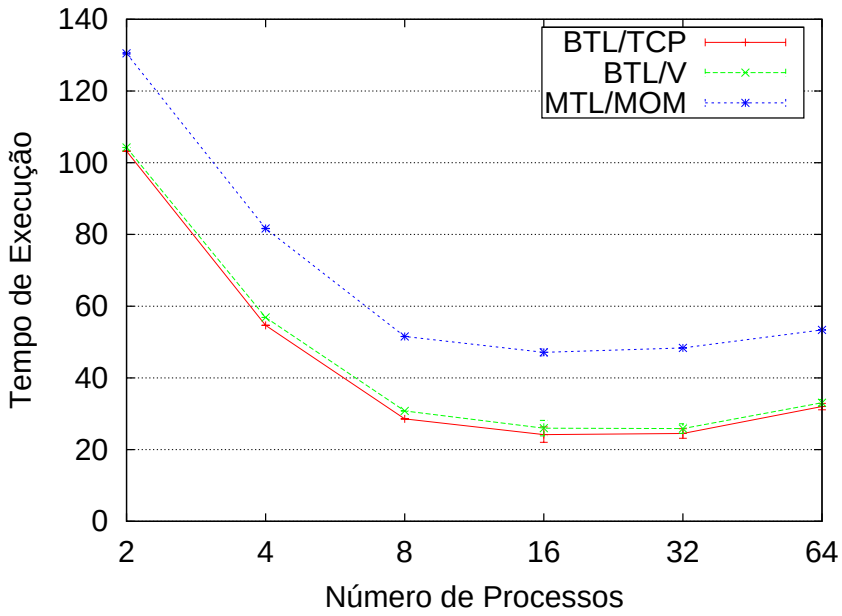

Figura 5.7: Resultados dos testes com o NPB MG (Classe C)

Na Figura 5.7 nota-se que a diferença de desempenho das implementações é muito menor do que a observada no *benchmark* FT. Isso se deve ao fato de não haver comunicação tão intensiva no *benchmark* MG. Observa-se que as diferenças nos tempos de execução entre as implementações de BTL e a MTL/MOM se mantém próximas mesmo

em execuções com números de processos diferentes. Os pontos críticos de desempenho, nos testes realizados, são as execuções a partir de 8 processos. Com 8 processos, as execuções com MTL/MOM levaram 51,54s, em média, enquanto as execuções com BTL/TCP e BTL/V levaram 28,56s e 30,8s, respectivamente. Como a diferença absoluta dos tempos de execução não se altera muito, conforme se aumenta o número de processos, estima-se que a implementação MTL/MOM não apresente um impacto muito maior que a BTL/TCP e a BTL/V em execuções com um número maior de processos.

5.4.5 Aplicação LU

O *benchmark* LU (*LU Decomposition*) resolve a discretização de equações de Navier-Stokes através de fatoração LU do sistema de equações em um bloco triangular-superior e um bloco triangular-inferior. A forma fatorada é então resolvida através do método iterativo de sobre-relaxação sucessiva. Foi utilizada a classe B do problema que utiliza uma matriz de 128x128x128 elementos.

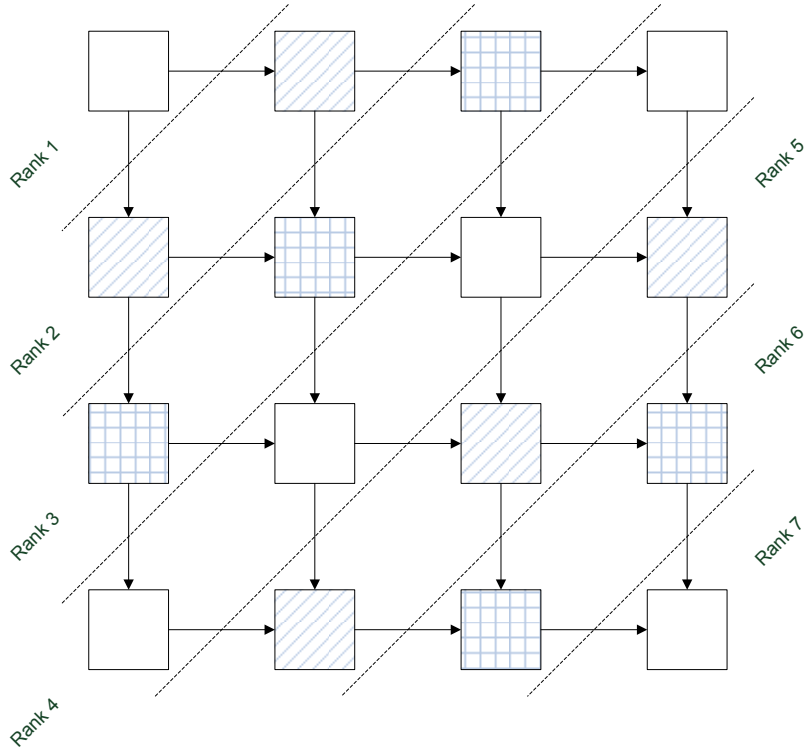

Figura 5.8: Padrão de comunicação do NPB LU

O padrão predominante de comunicação do *benchmark* LU segue a sequência de um *pipeline* conforme descrito a seguir. Considerando uma execução com 8 processos ilustrada na Figura 5.8, cada quadrado é uma subtarefa que precisa ser realizada e depende de resultados de tarefas de outros processos. Essas dependências são representadas pelas setas de uma subtarefa para outra. Partindo do processo de *rank* 1, assim que cada subtarefa sua é realizada, mensagens são enviadas para o processo de *rank* 2 para o cálculo das subtarefas correspondentes, conforme as dependências da figura. Logo após enviar as mensagens para o próximo *rank*, o processo de *rank* 1 prossegue com uma nova etapa de computação. O processo de *rank* 2, da mesma forma que o de *rank* 1, assim que termina a computação das suas subtarefas, envia os resultados para o processo seguinte. Essas

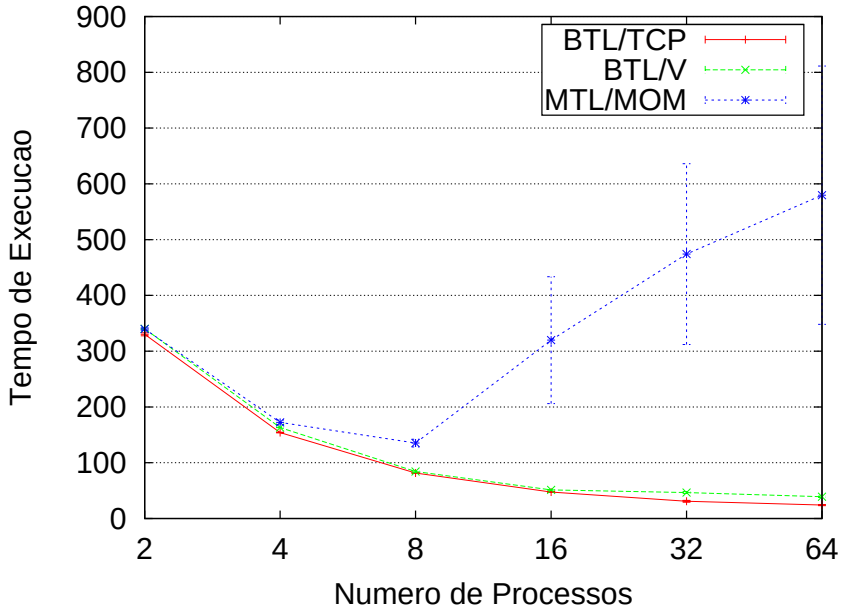

Figura 5.9: Resultados dos testes com o NPB LU (Classe B)

rotinas se repetem e quando o último processo (*rank* 7) recebe todas as mensagens com os resultados do processo de *rank* 6, o *pipeline* se enche. Os cálculos são finalizados depois que o primeiro processo não tem mais tarefas para calcular e os resultados da última computação são propagados até o processo de *rank* 7.

Pela Figura 5.9 nota-se que com até 4 processos a diferença de desempenho entre as implementações de BTL e o MTL/MOM não é tão grande. Nessas execuções o tempo médio obtido com o MTL/MOM é de 172,73s, com o BTL/TCP é de 154,16s e com o BTL/V é de 163,16. Já com 8 processos a diferença de desempenho é muito maior, os tempos são de 135,28s para o MTL/MOM, 81,45s para o BTL/TCP e de 84,16 para o BTL/V. A partir de 16 processos a aplicação começa a utilizar mais de um processo por processador do cluster e os tempos obtidos nas execuções sofrem uma intensa queda no desempenho. Através do OProfile, pode-se constatar que as execuções com mais de 16 processos acompanhavam um aumento da taxa de *cache miss* nos nodos. Acredita-se que exista uma relação de causalidade entre o número de processos executando em um mesmo nodo e o aumento do *cache miss*, e que esse aumento de *cache miss* é causado pelas cópias de buffers entre a API do OpenAMQ e do Open MPI. Além disso, a variância dos resultados a partir de 16 processos se torna bastante grande, como pode-se observar pelos intervalos de confiança do gráfico.

5.5 VH-1

Como aplicação real foi utilizado o VH-1 (*Virginia Hydrodynamics*) que foi desenvolvido na Universidade de Virginia pelo grupo de astrofísica numérica. Essa aplicação é mais complexa e elaborada que os demais *benchmarks* utilizados e os testes executados com ela visam verificar o comportamento da implementação MTL/MOM em uma aplicação real em vez de aplicações sintéticas que repetem um mesmo padrão de cálculo e comunicação continuamente. O VH-1 se baseia em um método parabólico de elementos

finitos (PPM – *Piecewise Parabolic Method*) para resolver equações de movimentos de fluídos compressíveis não viscosos. Essas equações são representadas na forma Euleriana conservativa:

$$\partial_t \rho + \nabla \cdot (\rho u) = 0,$$
$$\partial_t \rho u + \nabla \cdot (\rho u u) + \nabla p = F,$$
$$\partial_t \rho \varepsilon + \nabla \cdot (\rho \varepsilon u) + \nabla p u = G + \rho u \cdot F,$$

na qual $\varepsilon = u^2/2 + (\gamma - 1)^{-1} p/\rho$ é o total de energia específica, ρ é a densidade de massa, p é a pressão, e F e G são as fontes de energia e momento.

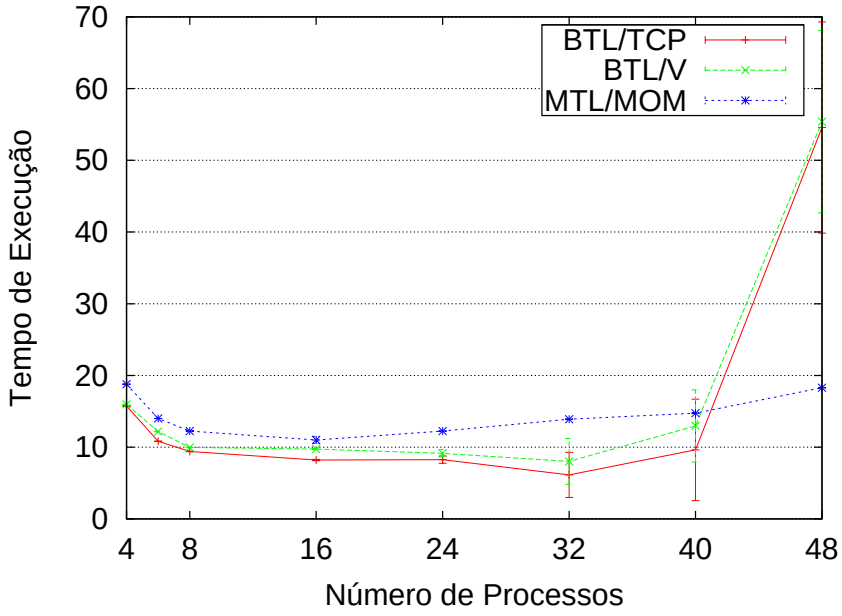

Figura 5.10: Resultados dos testes com o VH-1

As diferenças entre os tempos de execução do MTL/MOM e das implementações de BTL são muito menores que as de todos os testes realizados com os *benchmarks* do NPB. Para execuções com 16 processos, por exemplo, os valores obtidos são de 8,20s para o BTL/TCP, 9,74s para o BTL/V e 11s para o MTL/MOM. Os testes com o VH-1 mostram que existem aplicações reais que podem utilizar a implementação do MTL/MOM com pouco comprometimento do tempo total de execução.

Observa-se nas execuções com 48 processos que o desempenho das implementações sobre BTL aumentam subitamente para aproximadamente 5 vezes o valor obtido com 40 processos. Através da ferramenta OProfile, também utilizada para realizar constatações nos *benchmarks* NPB FT e LU, procurou-se descobrir se esse aumento súbito nos tempos de execução acompanhavam mudanças nas taxas de *cache miss*. Embora houvesse suspeita de alguma relação devido à necessidade de acessos concorrentes à memória pelos vários processos por nodo nas execuções com 48 processos, os resultados obtidos não mostraram um aumento da taxa de *cache miss*. Acredita-se que os tempos observados sejam causados por alguma característica da implementação do BTL que se manifeste com o padrão de comunicação do VH-1.

5.6 Considerações Finais

Os testes realizados permitiram identificar empiricamente gargalos e definir melhorias que podem ser incorporadas à implementação do Open MPI sobre MOM. Apesar das vantagens da comunicação sobre MOM expostas na Seção 3.2. Deve-se levar em consideração algumas características fundamentais da implementação de Open MPI sobre MOM durante a análise do desempenho. Cada uma dessas características compromete o desempenho da comunicação. São elas:

- Cópia dupla da mensagem necessária para o desacoplamento da comunicação.
- Sobrecusto imposto pelo próprio *middleware*.
- As primitivas de comunicação coletiva operam sobre primitivas de comunicação ponto a ponto.
- Servidor de mensagens centralizado. Pode ficar sobrecarregado e sua interconexão rede pode se tornar um gargalo quando houver muitos clientes conectados.

Foi possível observar que esses fatores impactam no desempenho da comunicação e, como consequência, no desempenho das aplicações paralelas. A cópia dupla de mensagens, necessária para o desacoplamento da comunicação, por si só, já implica em um aumento mínimo de 100% na latência e redução da largura de banda para a metade. Outro elemento é o *middleware* utilizado, que possui um sobrecusto demonstrado nos *micro-benchmarks* de latência e de banda passante, e no *benchmark* EP. Por fim, o próprio servidor centralizado, que não escalaria com aplicações utilizando um grande número de processos.

Os dois primeiros fatores são inerentes da solução proposta de comunicação MPI utilizando MOM e não é possível contorná-los. Já os dois últimos podem ser eliminados implementando as primitivas de comunicação coletiva conforme apresentado na Seção 3.3.2 e utilizando uma configuração de servidores distribuídos que possa distribuir a carga do roteamento das mensagens entre vários servidores.

Apesar de observar-se um desempenho inferior no OpenAMQ nos *micro-benchmarks*, esse foi o único *middleware* apropriado para uso quando foi realizado um levantamento inicial (SANTOS MACHADO, 2007). Os demais *middlewares*, ou possuíam licença proprietária, ou estavam em estágio de desenvolvimento pouco maduro para serem utilizados com confiabilidade razoável. Considerando que a proposta é genérica, poderia-se adaptar o sistema para outros MOMs a fim de avaliar o desempenho dos mesmos.

Como visto em alguns dos *benchmarks* (FT, LU), o desempenho da aplicação fica bastante comprometido quando há comunicação intensiva. No entanto, em *benchmarks* como o MG e CG a diferença de desempenho é menos intensa. Além disso, o desempenho observado no VH-1 comprova que a implementação pode ser utilizada em algumas aplicações reais sem perdas significativas.

O desempenho observado é esperado e justificável para uma implementação de *checkpoint* coordenado com log de mensagens pessimista remoto, pois nesse tipo de log é necessário armazenar as mensagens em um repositório remoto antes que a mensagem seja entregue no processo destinatário. Esse desempenho seria o mesmo se o log de mensagens descrito na Seção 3.2 estivesse implementado. Assim, aplicações com um padrão de comunicação não intensivo poderiam se beneficiar dos recursos de tolerância a falhas.

6 CONCLUSÃO

As previsões sobre a evolução das arquiteturas paralelas de computadores projetam grandes aumentos no número de processadores e, consequentemente, uma maior da taxa de falhas dos sistemas. Esse cenário impõe a necessidade de pesquisa em técnicas de tolerância a falhas mais adequadas. Entre as técnicas mais empregadas atualmente estão as de recuperação retroativa, mais especificamente as de *checkpoint* coordenado. Projeções afirmam que essa técnica será ineficiente em escalas muito grandes de processos e alguns estudos apontam as técnicas de log de mensagens como sendo mais apropriadas, dentre as conhecidas, para esses ambientes.

O presente trabalho propõe uma variante do log de mensagens pessimista que armazena as mensagens em mídia confiável remota para posterior restauração em caso de falha. O log de mensagens pessimista remoto é vantajoso nas etapas de recuperação, pois precisa restaurar apenas os processos que falharam juntamente com as suas mensagens. Além disso, foi implementado e avaliado o desempenho da comunicação MPI sobre canais *Publish/Subscriber* que pretende-se utilizar para suporte ao log de mensagens. A idéia é armazenar os logs de mensagem nos próprios canais para posterior restauração. A implementação se baseia em um componente do Open MPI que mapeia a comunicação para o MOM OpenAMQ.

Inicialmente, foram apresentadas as diferentes técnicas utilizadas para tolerância a falhas de aplicações paralelas de alto desempenho, com ênfase nas implementações para MPI. Também foram explicados problemas existentes na técnica de *checkpoint* coordenado, que é a mais empregada atualmente, e as vantagens que o log de mensagens tem sobre ela. Estima-se que as técnicas mais apropriadas para ambientes de Exaescala são as de log de mensagens. Entre essas técnicas, as de log causal e otimista possibilitam uma execução sem falhas com menor sobrecusto que as de log pessimista, no entanto, seus procedimentos de restauração são mais complexos e onerosos.

As implementações de log pessimista normalmente armazenam as mensagens em log remoto ou local. O log local traz a vantagem de um melhor desempenho na execução sem falhas mas pode apresentar um desempenho ruim nas execuções em ambientes nos quais múltiplas falhas são frequentes. Essa característica se deve à necessidade de retroceder a computação de mais processos nos casos em que mensagens que precisariam ser restauradas são perdidas nas múltiplas falhas. O log remoto não sofre desse problema, pois as mensagens são armazenadas em mídia confiável remota e não serão perdidas mesmo no caso de vários processos falharem na aplicação. Nesse caso, apenas os nodos que falharem precisarão retroceder a computação. Apesar dessa vantagem, existe um sobrecusto associado à dupla cópia de mensagens pela rede necessária para o log de mensagens pessimista remoto.

A implementação do componente do Open MPI utilizada para avaliar o custo do log de

mensagens pessimista remoto foi detalhada no Capítulo 4. O Open MPI foi escolhido por se tratar de uma implementação do MPI com bom suporte da comunidade de alto desempenho. O componente desenvolvido realiza a comunicação através do MOM OpenAMQ utilizando o método *Publish/Subscriber* que permite realizar comunicação desacoplada, propriedade utilizada para realizar o log de mensagens pessimista remoto. O OpenAMQ é um middleware de código aberto que implementa a especificação de MOM AMQP.

A comunicação desacoplada utilizada para suporte do log de mensagens traz consigo a necessidade de duas transferências de dados pela rede durante as trocas de mensagens. Uma transferência para o remetente enviar a mensagem para o *broker* do MOM e outra transferência para o destinatário recuperar a mensagem do *broker*. Essa dupla cópia se reflete nos desempenhos obtidos nos resultados dos testes de *benchmark*. Os testes realizados são execuções de *benchmarks* no MTL/MOM e outras duas implementações do Open MPI. As outras duas, o BTL/TCP e o BTL/V, realizam comunicação sobre TCP/IP, sendo que essa última serve como base para log de mensagens pessimista local.

Apesar dos testes apontarem um desempenho inferior na comunicação do MTL/MOM, esse era o resultado esperado devido a características inerentes à implementação de log de mensagens pessimista remoto, nos quais são necessárias duas cópias na rede para cada troca de mensagem. Além disso, os testes serviram para destacar outras características que afetam o desempenho do MTL/MOM. Apesar disso, as medidas obtidas com o VH-1 para o MTL/MOM, comprovam que existem aplicações reais que podem utilizar esse tipo de comunicação sem grande perda de desempenho. Os tempos de execução médios obtidos para 16 processos, por exemplo, foram de 8,20s para o BTL/TCP, 9,74s para o BTL/V e 11s para o MTL/MOM. Através do NPB, foi possível verificar que os *benchmarks* com padrões de comunicação menos intensivos têm seus desempenhos menos afetados, sugerindo que aplicações com menor demanda de comunicação são menos prejudicadas pelo sobrecusto de comunicação do MTL/MOM.

Por fim, para dar continuidade ao trabalho, é necessário finalizar o *checkpoint* e a restauração de processos individuais em meio de armazenamento remoto confiável. Adicionalmente, também resta a implementação da criação e restauração de logs de mensagem nos tópicos, conforme a proposta apresentada.

Uma otimização que poderia ser implementada no MTL/MOM é a utilização de múltiplos servidores de *broker* trabalhando de forma hierárquica para distribuição da carga dos processos da aplicação. Em todos os testes foi utilizado apenas um servidor de *broker* e para que o sistema possa escalar é imprescindível a distribuição dos processos clientes em múltiplos *brokers*. Inclusive, esses *brokers* poderiam ser executados em domínios administrativos diferentes de forma hierárquica para atender processos de uma mesma aplicação. Como otimização da comunicação poderia-se utilizar a comunicação *Publish/Subscriber* com anúncios em múltiplos tópicos em algumas primitivas de comunicação coletiva que realizam difusão de mensagens em grupo. Essas duas últimas configurações, apesar de não terem sido implementadas, são abordadas no Capítulo 3.

Os estudos realizados e a implementação desenvolvida, juntamente com as suas respectivas medidas de desempenho, ajudaram a esclarecer várias propriedades das técnicas de log de mensagens. Cada técnica apresenta diferentes compromissos entre o desempenho nas etapas de restauração e o sobrecusto na execução normal. Com base na experiência adquirida, algumas decisões de projeto poderiam ser avaliadas novamente.

Uma decisão diz respeito ao MOM escolhido. Quando o OpenAMQ foi escolhido para a implementação da solução não haviam outras opções de MOM de código aberto que fossem suficientemente estáveis. No entanto, poderia ser utilizado qualquer outro MOM

que pudessem trabalhar no modelo *Publish/Subscriber*, sem perda de generalidade da proposta. Poderia-se avaliar o desempenho da comunicação com outras implementações de MOM como o QPid (Apache Software Foundation, Inc, 2010).

Outra decisão é a utilização de log de mensagens pessimista remoto cujo sobrecusto é elevado durante a execução normal. Poderia ser estudada a utilização de alguma variação do log causal que apresentasse menor sobrecusto. Para isso seria necessário um tratamento mais aprimorado dos casos de múltiplas falhas simultâneas de processos, já que essa técnica não as cobre de maneira tão eficiente quanto o log pessimista. Principalmente porque, segundo as tendências de aumento da taxa de falhas nas arquiteturas paralelas, a probabilidade de ocorrência de múltiplas falhas será muito maior.

REFERÊNCIAS

AGBARIA, A.; FRIEDMAN, R. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING (HPDC), 8., Washington, DC, USA. **Anais...** IEEE Computer Society, 1999. p.31.

ALDRED, L. J. et al. On the Notion of Coupling in Communication Middleware. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED OBJECTS AND APPLICATIONS (DOA), 7., Agia Napa, Cyprus. **Anais...** Springer-Verlag, 2005. p.1015–1033.

ALVISI, L.; MARZULLO, K. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. **IEEE Transactions on Software Engineering**, Los Alamitos, CA, USA, v.24, p.149–159, 1998.

Apache Software Foundation, Inc. **Apache Qpid**. Disponível em: http://cwiki.apache.org/qpid/. Acesso em: junho de 2008.

AULWES, R. T. Architecture of LA-MPI, A Network-Fault-Tolerant MPI. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS' 04), 18. **Anais...** [S.l.: s.n.], 2004. p.15b.

BAGCHI, S. et al. Chameleon: software infrastructure for adaptive fault tolerance. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.10, p.560–579, 1999.

BAILEY, D. et al. **The NAS Parallel Benchmarks 2.0**. [S.l.: s.n.], 1995.

BAILEY, D. H. et al. **The NAS Parallel Benchmarks**. [S.l.]: The International Journal of Supercomputer Applications, 1991.

BANAVAR, G. et al. A Case for Message-Oriented Middleware. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SYSTEMS, LNCS 1693, 13., Bratislava, Slovak Republic. **Anais...** Springer Berlin/Heidelberg, 1999. p.1–18.

BATCHU, R. et al. MPI/FT: a model-based approach to low-overhead fault tolerant message-passing middleware. **Cluster Computing**, Hingham, MA, USA, v.7, n.4, p.303–315, 2004.

BOSILCA, G. et al. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In: SUPERCOMPUTING '02: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, Los Alamitos, CA, USA. **Anais...** IEEE Computer Society Press, 2002. p.1–18.

BOUTEILLER, A. et al. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In: SC '03: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, Washington, DC, USA. **Anais...** IEEE Computer Society, 2003. p.25.

BOUTEILLER, A. et al. Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI. In: IPDPS '05: 19TH IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS'05) - PAPERS, Washington, DC, USA. **Anais...** IEEE Computer Society, 2005. p.97.

BOUTEILLER, A. et al. Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure, recovery. In: CLUSTER. **Anais...** IEEE, 2009. p.1–9.

CAPIT, N. et al. A batch scheduler with high level components. In: IN CLUSTER COMPUTING AND GRID 2005 (CCGRID05). **Anais...** [S.l.: s.n.], 2005. p.776–783.

CAPPELLO, F. Fault Tolerance in Petascale/ Exascale Systems: current knowledge, challenges and research opportunities. **Int. J. High Perform. Comput. Appl.**, Thousand Oaks, CA, USA, v.23, n.3, p.212–226, 2009.

CHAKRAVORTY, S.; KALÉ, L. V. Proactive Fault Tolerance in MPI Applications Via Task Migration. In: HIPC. **Anais...** Springer, 2006. p.485–496. (Lecture Notes in Computer Science, v.4297).

CHAKRAVORTY, S.; KALE, L. V. A Fault Tolerance Protocol with Fast Fault Recovery. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 21. **Anais...** IEEE Press, 2007.

CHAO HUANG ORION LAWLOR, L. V. K. Adaptive MPI. In: INTERNATIONAL WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING (LCPC 2003), LNCS 2958, 16., Bratislava, Slovak Republic. **Anais...** [S.l.: s.n.], 2003. p.306–322.

CHAPMAN, B.; JOST, G.; PAS, R. v. d. **Using OpenMP**: portable shared memory parallel programming (scientific and engineering computation). [S.l.]: The MIT Press, 2007.

CHEN, Y.; PLANK, J. S.; LI, K. CLIP: a checkpointing tool for message-passing parallel programs. In: SUPERCOMPUTING '97: ACM/IEEE CONFERENCE ON SUPERCOMPUTING (CDROM), New York, NY, USA. **Anais...** ACM, 1997. p.1–11.

COMPUTINA, I. et al. **Algorithm-dependent Fault Tolerance for Distributed Computing**. 2000.

COTI, C. et al. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING (SC '06), New York, NY, USA. **Anais...** ACM, 2006. p.127.

COULOURIS, G. **Distributed Systems Concepts and Design**. Harlow, UK: Addison-Wesley, 2001.

CRISTIAN, F. A Rigorous Approach to Fault-Tolerant Programming. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v.11, n.1, p.23–31, 1985.

DIJKSTRA, E. W. The distributed snapshot of K.M. Chandy and L. Lamport. In: NATO ADVANCED STUDY INSTITUTE ON CONTROL FLOW AND DATA FLOW: CONCEPTS OF DISTRIBUTED PROGRAMMING, New York, NY, USA. **Proceedings...** Springer-Verlag New York: Inc., 1986. p.513–517.

DONGARRA, J. J. et al. **TOP500 supercomputer sites**. [S.l.]: Supercomputer, 1997.

ELNOZAHY, E. N. M. et al. A survey of rollback-recovery protocols in message-passing systems. **ACM Comput. Surv.**, New York, NY, USA, v.34, n.3, p.375–408, 2002.

EUGSTER, P. T. et al. The many faces of publish/subscribe. **ACM Computing Surveys**, New York, NY, USA, v.35, n.2, p.114–131, 2003.

FAGG, G. E.; DONGARRA, J. FT-MPI: fault tolerant mpi, supporting dynamic applications in a dynamic world. In: EUROPEAN PVM/MPI USERS' GROUP MEETING ON RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE, 7., London, UK. **Anais...** Springer-Verlag, 2000. p.346–353.

FARAJ, A. A. **Communication Characteristics in the NAS Parallel Benchmarks**. 2002. Dissertação (Mestrado em Ciência da Computação) — College of Art & Science, Florida State University.

FOSTER, I.; KESSELMAN, C. **The Grid 2**: blueprint for a new computing infrastructure. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

GABRIEL, E. et al. Open MPI: goals, concept, and design of a next generation MPI implementation. In: EUROPEAN PVM/MPI USERS' GROUP MEETING, 11., Budapest, Hungary. **Anais...** [S.l.: s.n.], 2004. p.97–104.

GEIST, A.; LUCAS, R. Major Computer Science Challenges At Exascale. **Int. J. High Perform. Comput. Appl.**, Thousand Oaks, CA, USA, v.23, n.4, p.427–436, 2009.

GEOFFRAY, P. Myrinet eXpress (MX): is your interconnect smart? In: HPCASIA '04: PROCEEDINGS OF THE HIGH PERFORMANCE COMPUTING AND GRID IN ASIA PACIFIC REGION, SEVENTH INTERNATIONAL CONFERENCE, Washington, DC, USA. **Anais...** IEEE Computer Society, 2004. p.452–452.

GIOTTA, P. et al. **Professional JMS Programming**. Birmingham, UK: Wrox Press, 2000.

GOLDBERG, D. et al. The Design and Implementation of a Fault-Tolerant Cluster Manager. **Tech. Rep., Xerox Systems Institute**, [S.l.], 2001.

GRAHAM, R. L. et al. An Evaluation of Open MPI's Matching Transport Layer on the Cray XT. In: EUROPEAN PVM/MPI USER'S GROUP MEETING, 14., Paris, France. **Anais...** [S.l.: s.n.], 2007. p.161–169.

GROPP, W. et al. A high-performance, portable implementation of the MPI message passing interface standard. **Parallel Computing**, [S.l.], v.22, n.6, p.789–828, Sept. 1996.

GROPP, W.; THAKUR, R. **Using MPI-2**: advanced features of the message-passing interface. Cambridge, MA, USA: MIT Press, 1999.

GUSTAFSON, J. L.; SNELL, Q. O. HINT: a new way to measure computer performance. In: ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEMS SCIENCES, IEEE COMPUTER SOCIETY PRESS, 28. **Anais...** IEEE Computer Society Press, 1995. p.392–401.

HARGROVE, P. H.; DUELL, J. C. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In: SCIENTIFIC DISCOVERY THROUGH ADVANCED COMPUTING PROGRAM, Denver, USA. **Anais...** [S.l.: s.n.], 2006.

HOHPE, G.; WOOLF, B. **Enterprise Integration Patterns** : designing, building, and deploying messaging solutions. [S.l.]: Addison-Wesley Professional, 2003.

HUANG, C. **System Support for Checkpoint and Restart of Charm++ and AMPI Applications**. 2004. Dissertação (Mestrado em Ciência da Computação) — Dept. of Computer Science, University of Illinois.

HURSEY, J. et al. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), 21. **Anais...** IEEE Computer Society, 2007.

iMatix Corporation. **Open Advanced Message Queue**. Disponível em: http://www.openamq.org/. Acesso em: junho de 2008.

KADEPLOY. **Kadeploy.** Disponível em: http://kadeploy.imag.fr/. Acesso em: junho de 2010.

KALE, L. V.; KRISHNAN, S. CHARM++: a portable concurrent object oriented system based on c++. In: OOPSLA '93: 8TH ANNUAL CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, New York, NY, USA. **Anais...** ACM, 1993. p.91–108.

KIM, K. H. ROAFTS: a middleware architecture for real-time object-oriented adaptive fault tolerance support. In: HASE '98: THE 3RD IEEE INTERNATIONAL SYMPOSIUM ON HIGH-ASSURANCE SYSTEMS ENGINEERING, Washington, DC, USA. **Anais...** IEEE Computer Society, 1998. p.50.

KOGGE, P. et al. **Exascale Computing Study**: Technology Challenges in Achieving Exascale Systems. Washington, DC, USA: DARPA, 2008. Technical Report.

KRAMER, J. Advanced Message Queuing Protocol (AMQP). **Linux J.**, Seattle, WA, USA, v.2009, n.187, p.3, 2009.

LAMPORT, L.; PEASE, M. The Byzantine Generals Problem. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.4, n.3, p.382–401, 1982.

LEANGSUKSUN, C. B. et al. Achieving high availability and performance computing with an HA-OSCAR cluster. **Future Gener. Comput. Syst.**, Amsterdam, The Netherlands, The Netherlands, v.21, n.4, p.597–606, 2005.

LEMARINIER, P. et al. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In: CLUSTER '04: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, Washington, DC, USA. **Anais...** IEEE Computer Society, 2004. p.115–124.

LEMARINIER, P. et al. Coordinated checkpoint versus message log for fault tolerant MPI. **Int. J. High Perform. Comput. Netw.**, Inderscience Publishers, Geneva, SWITZERLAND, v.2, n.2-4, p.146–155, 2004.

LEVON, J. **OProfile Manual**. [S.l.]: Victoria University of Manchester, 2004.

LI, M. **Fault-tolerant cluster management**. 2006. Tese (Doutorado em Ciência da Computação) — University of California at Los Angeles, Los Angeles, CA, USA. Adviser-Tamir, Yuval.

LITZKOW, M. et al. **Checkpoint and Migration of UNIX Processes in the COndor Distributed Processing System**. [S.l.]: University of Wisconsin - Madison Computer Sciences Department, 1997. (UW-CS-TR-1346).

LOUCA, S. et al. MPI-FT: portable fault tolerance scheme for mpi. **Parallel Processing Letters**, [S.l.], v.10, n.4, p.371–382, 2000.

MPI-2: extensions to the message-passing interface. 1997.

MPI FORUM. **The MPI Message Passing Interface Standard**. Knoxville: University of Tennessee, 1994.

OUYANG, X.; GOPALAKRISHNAN, K.; PANDA, D. K. Accelerating Checkpoint Operation by Node-Level Write Aggregation on Multicore Systems. In: ICPP '09: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, Washington, DC, USA. **Anais...** IEEE Computer Society, 2009. p.34–41.

WILHELM, S. (Ed.). **UNIX, POSIX, and Open Systems**: the open standards puzzle. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1992.

RAO, S.; ALVISI, L.; VIN, H. M. The Cost of Recovery in Message Logging Protocols. **IEEE Trans. on Knowl. and Data Eng.**, Piscataway, NJ, USA, v.12, n.2, p.160–173, 2000.

RAO, S. S.; ALVISI, L.; VIN, H. M. **Egida**: an extensible toolkit for low-overhead fault-tolerance. Austin, TX, USA: [s.n.], 1999.

REITER, M. K. The Rampart Toolkit for Building High-Integrity Services. In: SELECTED PAPERS FROM THE INTERNATIONAL WORKSHOP ON THEORY AND PRACTICE IN DISTRIBUTED SYSTEMS, London, UK. **Anais...** Springer-Verlag, 1995. p.99–110.

RIESEN, R. et al. **The Portals 4.0 Message Passing Interface**. [S.l.]: Sandia National Laboratories, 2008. Technical report. (SAND2008-2639).

SANKARAN, S. et al. **Checkpoint-Restart Support System Services Interface (SSI) Modules for LAM/MPI**. [S.l.]: Indiana University, Computer Science Department, 2003. Technical Report. (TR578).

SANTOS MACHADO, C. dos. **Middlewares de Comunicação**. [S.l.]: Universidade Federal do Rio Grande do Sul, 2007. Trabalho Individual.

SCHNEIDER, F. B. **Abstractions for Fault Tolerance in Distributed Systems**. Ithaca, NY, USA: [s.n.], 1986.

SHIPMAN, G. M.; BOSILCA, G. Network Fault Tolerance in Open MPI. In: EURO-PAR 2007, PARALLEL PROCESSING, 13TH INTERNATIONAL EURO-PAR CONFERENCE, RENNES, FRANCE, AUGUST 28-31, 2007. **Anais...** Springer, 2007. p.868–878. (Lecture Notes in Computer Science, v.4641).

SNELL, Q. O.; GUSTAFSON, J. L. NetPIPE: A Network Protocol Independent Performance Evaluator. In: IASTED INTERNATIONAL CONFERENCE ON INTELLIGENT INFORMATION MANAGEMENT AND SYSTEMS. **Anais...** [S.l.: s.n.], 1996.

SQUYRES, J. M. A component architecture for LAM/MPI. In: PPOPP '03: 9TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, New York, NY, USA. **Anais...** ACM, 2003. p.2.

STELLNER, G. CoCheck: checkpointing and process migration for mpi. In: IPPS '96: 10TH INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM, Washington, DC, USA. **Anais...** IEEE Computer Society, 1996. p.526–531.

TREASTER, M. A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems. **CoRR**, [S.l.], v.abs/cs/0501002, 2005.

VH-1. **The Virginia Numerical Bull Session ideal hydrodynamics PPMLR**. Disponível em: http://wonka.physics.ncsu.edu/pub/VH-1/. Acesso em: maio de 2010.

ZHENG, G. B.; KALÉ, L. V. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2004., San Diego, CA. **Anais...** [S.l.: s.n.], 2004. p.93–103.