UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Mecanismos para Migração de Processos na Simulação Distribuída

Mateus Augusto Faustino Chaib Junqueira

Itajubá, Março de 2012

UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Mateus Augusto Faustino Chaib Junqueira

Mecanismos para Migração de Processos na Simulação Distribuída

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciências em Ciência e Tecnologia da Computação

Área de Concentração: Sistemas de Computação

Orientador: Edmilson Marmo Moreira

Março de 2012

Itajubá-MG

Ficha catalográfica elaborada pela Biblioteca Mauá –
Bibliotecária Margareth Ribeiro- CRB_6/1700

J95m

Junqueira, Mateus Augusto Faustino Chaib
Mecanismos de migração de processos na simulação distribuí_
da/ Mateus Augusto Faustino Chaib Junqueira. -- Itajubá, (MG) :
[s.n.], 2012.
119 p. : il.

Orientador: Prof. Dr. Edmilson Marmo Moreira.
Dissertação (Mestrado) – Universidade Federal de Itajubá.

1. Simulação distribuída. 2. Time Warp. 3. Migração de pro_
cessos. 4. Rollback solidário. 5. Baleanceamento de carga. I. Mo_
reira, Edmilson Marmo, orient. II. Universidade Federal de Itaju_
bá. III. Título.

Agradecimentos

Primeiramente a Deus pela vida que me concedeu e por iluminar os meus caminhos.

Ao professor Dr. Edmilson Marmo Moreira pela orientação, disponibilidade e solicitude na realização deste trabalho.

Ao professor Dr. Otávio Augusto Salgado Carpinteiro pela co-orientação e dicas úteis na realização deste trabalho.

À professora Dra. Thatyana de Faria Piola Seraphim pela ajuda no uso do MPI.

Ao professor Dr. Enzo Seraphim por me abrir às portas para o ingresso no mestrado em computação.

Ao colega de mestrado Márcio pela contribuição com a implementação do *Time Warp*.

Aos meus pais Guto e Tereza pelo amor, incentivo, compreensão e apoio constante.

À minha família pelo convívio, confiança e amizade por mim.

Ao grande amigo Renato Moutinho pelas prosas divertidas.

À minha grande amiga Fernanda Leal pelos conselhos recebidos.

Aos professores do IESTI por todo o conhecimento adquirido.

Ao GPESC e a UNIFEI pela oportunidade e pelos recursos oferecidos para a realização deste trabalho.

À CAPES pelo apoio financeiro.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho apresenta dois mecanismos para realizar a migração de processos em uma aplicação de simulação distribuída baseada no protocolo *Time Warp*. Ambos os mecanismos são capazes de obter informações da simulação e disponibilizá-los para um algoritmo de balanceamento. Um dos mecanismos é o de migração coletiva que consiste no término e recriação de todos os processos da simulação. O outro é o mecanismo de migração individual no qual apenas alguns processos são terminados e recriados. O projeto desses mecanismos considerou o uso da biblioteca de comunicação MPI e foi afetado pelos recursos disponíveis nessa biblioteca. Experimentalmente foi possível avaliar os mecanismos propostos e verificar que, apenas em condições particulares, o mecanismo de migração individual é o mais indicado. Considerando o uso de um algoritmo de balanceamento voltado para a simulação distribuída, foi demonstrado, teoricamente, que ambos os mecanismos de migração propostos podem melhorar o desempenho da simulação.

1 Introdução

Para Shannon, Sadowski e Pegden (1990), a simulação é o processo de construir um modelo de um sistema real e realizar experimentos com este modelo com o objetivo de compreender seu comportamento avaliando estratégias para sua operação. A simulação discreta procura repetir em um computador o comportamento de um modelo de um sistema real. É utilizada para obter respostas sobre perguntas do problema modelado. De acordo com Schriber e Brunner (2002), a simulação resulta da modelagem de um processo ou sistema, de tal forma que o modelo emite as respostas do sistema real numa sequência de eventos que ocorrem com o passar do tempo.

A simulação já vem sendo há muito tempo utilizada para prever o comportamento de sistemas complexos que envolvem áreas básicas como: Física, Matemática e Química. Também é utilizada em outras situações, tais como:

- Simulação de operações em companhias para testar e prever alterações em seus procedimentos;
- Simulação de tráfego de veículos prevendo a influência de novas ruas, sinais e regras de preferência;
- Simulação da economia de um setor de um país prevendo o efeito de mudanças políticas como subsídios;
- Simulação de caixas em supermercado ou banco para verificar a influência da adição de novos caixas no tamanho das filas.

Entre os objetivos da simulação podem-se citar:

- Prever o comportamento de sistemas usando modelos e verificar os efeitos produzidos por modificações no sistema;
- Elaborar teorias e hipóteses com a análise das observações efetuadas através da simulação dos modelos;
- Auxiliar na tomada de decisão ao se deparar com questões do tipo: "qual a influência da adição de um novo turno de produção?"; "o que acontece se aumentarmos os números de caixas de atendimento?"; "o que acontece se houver um pico de demanda?".

Segundo Banks, Carson e Nelson (1996), a simulação tem como característica a geração de um histórico artificial do comportamento de algum sistema. Com a observação deste histórico, o analista ou pesquisador pode criar hipóteses sobre as características operacionais do sistema real em estudo. Esses históricos são elaborados a partir de modelos de sistemas criados especificamente para descrever um sistema particular. O modelo de um sistema é uma representação simplificada de um conjunto de componentes interligados que interagem entre si e com o meio ambiente (PERIN, 1995).

Os modelos podem ser construídos com uso de algumas técnicas como: rede de filas (PRADO, 1999), *statecharts* (HAREL et al., 1987) e redes de petri (CARDOSO; VALLETE, 1997). A partir do modelo, pode-se obter um programa para realizar a simulação.

A simulação pode ser executada de duas formas: sequencial ou paralela. Na forma sequencial, a simulação contém apenas um processo com uma única linha de execução, ao contrário da versão paralela, que pode ter mais de um processo executando de forma concorrente ou paralela.

A versão paralela da simulação pode se beneficiar da execução em *clusters*, *grids* computacionais ou na computação em nuvens (*cloud computing*), onde seus

processos podem executar de forma paralela aumentando assim a velocidade de processamento. Neste caso, a simulação é dita ser distribuída.

1.1 Objetivos

O principal objetivo deste trabalho é propor mecanismos para migração de processos em modo usuário, na simulação distribuída com a capacidade de obter, em tempo de execução, informações sobre a execução da simulação, de modo que algoritmos de balanceamento específicos para a simulação distribuída possam ser aplicados com a finalidade de balancear a carga da aplicação.

Além disso, uma avaliação dos mecanismos propostos faz parte dos objetivos desse trabalho de forma a obter, experimentalmente, o custo da migração de processos.

Como um objetivo final, será apresentada uma previsão teórica sobre os ganhos que podem ser obtidos com a migração de processos fazendo uso de um algoritmo de balanceamento específico para a simulação distribuída.

1.2 Motivação

A motivação desse trabalho consiste no fato de que existem várias soluções para migração de processos implementadas em ambiente de usuário que, no entanto, não fazem uso de informações específicas da aplicação para a determinação de como alocar os processos entre o processadores disponíveis.

A utilização de informações da simulação distribuída pode levar a um melhor desempenho quando é empregada para determinar o escalonamento de processos.

A migração de processos pode aumentar o desempenho da simulação distribuída particularmente em sistemas heteregêneos, como por exemplo a computação em nuvens (*cloud computing*), que, atualmente, oferece a possibilidade de

realizar a simulação distribuída de forma mais acessível para a comunidade científica (FUJIMOTO; MALIK; PARK, 2010). Como os ambientes de computação em nuvem são cada vez mais comuns, a sua utilização para a Simulação Distribuída torna-se mais atraente (ANGELO, 2011).

Para o funcionamento de mecanismos de migração em sistemas heteregêneos, é interessante implementá-los em modo usuário para garantir maior compatibilidade em diferentes plataformas, escolha adotada neste trabalho.

A migração na simulação distribuída pode melhorar o desempenho quando o modelo da simulação muda dinamicamente, ou ainda, quando o mapeamento da simulação ocorrer dinamicamente para modelos grandes ou em sistemas de computação que se modificam dinamicamente (KURVE; GRIFFIN; KESIDIS, 2011).

1.3 Estrutura da Dissertação

Esta dissertação está organizada da seguinte forma: o capítulo 2 faz uma revisão de Simulação Distribuída, com ênfase no protocolo *Time Warp* que foi tomado como base para o projeto dos mecanismos de migração propostos.

O capítulo 3 apresenta conceitos básicos de migração de processos com seus procedimentos envolvidos, pois são necessários para o entendimento dos mecanismos de migração propostos.

O capítulo 4 apresenta vários algoritmos de balanceamento utilizados para a simulação distribuída, evidenciando que muitos deles podem ser utilizados em conjunto com a migração de processos.

O capítulo 5 faz uma revisão sobre a biblioteca de comunicação MPI, já que a mesma foi utilizada nas implementações realizadas afetando assim o projeto dos mecanismos de migração propostos.

O capítulo 6 apresenta os mecanismos de migração propostos, discutindo também, como realizar a implementação com MPI.

O capítulo 7 apresenta os resultados experimentais para ambos os mecanismos, avaliando seu correto funcionamento, influência no número de mensagens *stragglers* e anti-mensagens, tempo para migração e custo associado ao seu funcionamento.

O capítulo 8 descreve a conclusão sobre o trabalho apresentando suas contribuições e propostas para trabalhos futuros.

2 Simulação Distribuída

A simulação distribuída consiste na execução de um programa de simulação em um Sistema Distribuído para simular modelos complexos, procurando diminuir o tempo na obtenção dos resultados. O conceito se baseia na paralelização do modelo em partes com a intenção de aumentar a velocidade com que este é simulado (CHWIF; PAUL; BARRETTO, 2006). Também é possível executar replicações de um mesmo programa de simulação sequencial de forma a obter um paralelismo. Criada a possibilidade de paralelismo, a simulação pode então ser executada em um Sistema Distribuído ou em uma máquina paralela.

Ao realizar a paralelização do programa de simulação, surgem porém, dificuldades que em um programa sequencial não existiriam, como: a necessidade de sincronização dos processos, o balanceamento de carga do sistema e a sobrecarga na rede de comunicação. A sincronização dos processos é um assunto muito estudado na Simulação Distribuída devido à dificuldade em coordenar os relógios lógicos dos processos de forma confiável e por ser indispensável para obter resultados que poderiam ser obtidos em um sistema real. Em uma simulação sequencial não há a necessidade de sincronização explícita, pois nos algoritmos sequenciais, a ordenação dos eventos ocorre naturalmente.

Existem, basicamente, duas abordagens para se desenvolver uma aplicação de simulação distribuída. Uma delas é chamada de SRIP(*Single Replication in Parallel*) (FUJIMOTO, 1990) e a outra é chamada de MRIP (*Multiple Replication in Parallel*) (REGO; SUNDERAN, 1991; EWING; PAWLIKOWSKI; MCNICKLE, 1999).

A abordagem MRIP consiste em fazer replicações independentes de um programa de simulação sequencial de tal forma que sejam executados em paralelo. Os resultados então obtidos por cada instância do programa são analisados, as médias são calculadas e quando um critério de término é atingido, por exemplo, precisão desejada, a simulação é terminada. Pode ser utilizada em qualquer sistema. Essa abordagem só não é aplicada quando (GLYNN; HEIDELBERGER, 1992):

1. O tamanho do modelo de simulação é grande e a replicação em um único processador pode levar um tempo elevado de execução;
2. Os resultados de cada replicação são semelhantes, ou seja, a variância é pequena. Assim, a replicação se torna ineficiente.

As principais vantagens do uso da abordagem MRIP são (EWING; PAWLIKOWSKI; MCNICKLE, 1999):

1. Não haver necessidade de modificação em qualquer programa de simulação;
2. Ser de fácil aplicação;
3. Apresentar uma eficiência próxima de 100% já que os programas são independentes, ou seja, o *speedup* é próximo ao número de processadores;
4. Apresentar elevada tolerância a falhas, já que as instâncias em execução são independentes entre si, a falha de uma não afeta a outra. Uma atenção maior deve ser dada ao processo que analisa os resultados da simulação em cada replicação do programa de simulação. Se esse processo falha por algum motivo, toda a simulação pode ser perdida.

Bruschi (2003) apresentou um ambiente de simulação distribuído automático (ASDA). Resultados desse trabalho levam à definição de algumas diretrizes no uso da abordagem MRIP.

As próximas seções descrevem a abordagem SRIP uma vez que este trabalho teve por base os protocolos de sincronização otimistas, particularmente o *Time Warp*, e que se enquadram dentro dessa abordagem.

2.1 Abordagem SRIP - Single Replication in Parallel

Na abordagem SRIP, o programa de simulação é formado por vários processos lógicos, cada um representando um processo do modelo. Esse modelo pode ser representado por meio de um grafo, como ilustra a figura 1. Esta figura mostra que há quatro nós no grafo, ou seja, quatro processos no modelo representado. As arestas no grafo representam a probabilidade de comunicação entre os processos, ou ainda, a probabilidade de geração de eventos entre si ou à si mesmos.

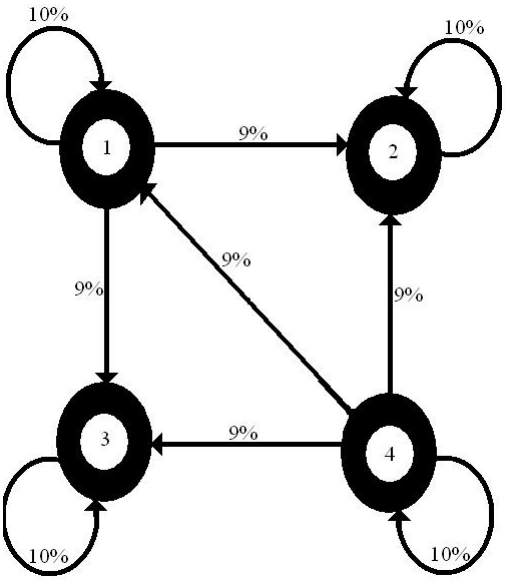

Figura 1: Modelo de Simulação Representado por Meio de Grafo

A interação entre os processos é realizada exclusivamente por mensagens (eventos com *timestamp*) enviadas entre eles. Os processos contêm uma lista de eventos futuros e um relógio lógico que representa o tempo de simulação. A execução do processo consiste basicamente de um *loop* principal no qual um evento da lista de

eventos futuros é simulado e eventos recebidos (mensagens) de outros processos são adicionados na lista de eventos futuros.

Erros de causa e efeito podem ocorrer nesta abordagem dependendo do protocolo de sincronização utilizado. Erros de causa e efeito são entendidos como situações de inconsistência ocorridas na simulação. Quando um processo recebe uma mensagem de outro processo (ou seja, recebe um evento para ser simulado), se o processo receptor possuir um relógio lógico (LVT - *Local Virtual Time*) com valor superior ao do evento recebido, (*timestamp*) tem-se um erro de causa e efeito. A figura 2 ilustra essa situação. Nesta figura podem ser observados dois processos da simulação. Esses processos podem criar eventos entre si e isso ocorre por meio de troca de mensagens. Inicialmente ambos os processos apresentam LVT igual a zero, e, à medida que simulam eventos, seus LVTs aumentam. O processo P1, quando se encontra no $LVT = 26$, cria um evento para o processo P2 enviando-o através de uma mensagem. O processo P2, ao receber o evento, possui $LVT = 38$. Esta é uma situação de inconsistência ou um erro de causalidade, pois o LVT do processo receptor (P2) é maior que o *timestamp* do evento recebido. Esta situação deve ser tratada na simulação distribuída quando verificada sua ocorrência. As mensagens que provocam erros de causalidade são denominadas de mensagens *stragglers*.

Dentro da abordagem SRIP, existem duas classes de protocolos que surgiram com o propósito de sincronizar e garantir a consistência dos resultados das simulações. A primeira classe é representada pelos protocolos conservativos que se caracterizam por impedir a ocorrência de erros de causa e efeito. Por outro lado, a segunda classe, representada pelos protocolos otimistas, permite a ocorrência dos erros de causa e efeito. Quando um erro de causa e efeito ocorre, ele é tratado logo em seguida (WANG; TROPPER, 2007).

A paralelização da simulação utilizando tais protocolos não garante que haverá uma redução no tempo de simulação, devido à computação introduzida em seus procedimentos.

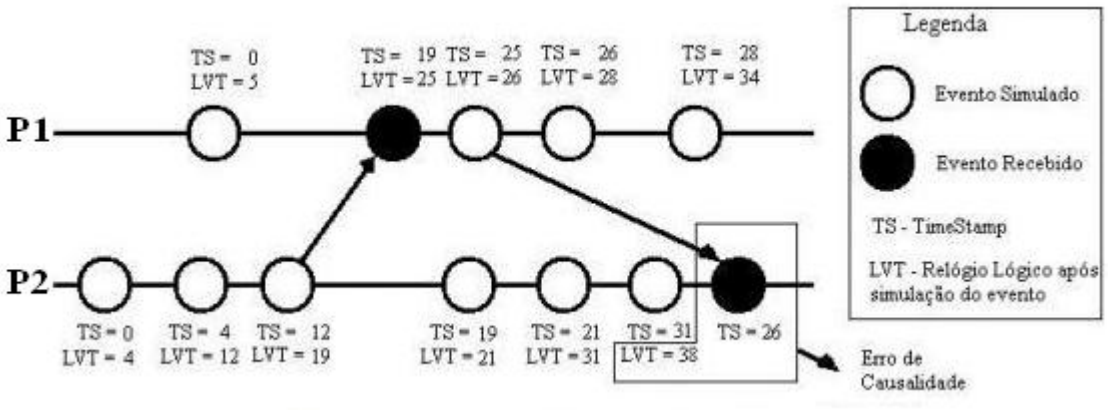

Figura 2: Erro de causa e efeito

2.1.1 Protocolos Conservativos

Os protocolos conservativos têm como principal característica o impedimento da ocorrência de erros de causa e efeito. Neste protocolo, um processo da simulação só pode tratar um evento da sua lista de eventos futuros se o sistema garantir que a simulação desse evento não levará a uma situação de risco, ou seja, uma situação em que haja a possibilidade de ocorrência de erros de causa e efeito. Um exemplo de protocolo conservativo é o desenvolvido de forma independente por Chandy e Misra (1979) e por Bryant (1977). Devido ao nome dos autores, este protocolo conservativo é conhecido como CMB (Chandy, Misra e Bryant).

Em geral, para o funcionamento dos protocolos conservativos, o sistema de computação deve possuir as seguintes características:

- A comunicação entre os processos faz uso de canais estáticos, e assim, um processo só se comunica com outro processo se existir entre eles um canal de comunicação previamento definido;
- As mensagens enviadas através dos canais de comunicação são armazenadas em fila FIFO (*first in first out*) mantendo sempre a ordem cronológica de envio e recebimento;

- Todas as mensagens contém um campo chamado de *timestamp* que indica o relógio lógico local ou LVT (*Local Virtual Time*) do processo emissor.

O funcionamento deste protocolo se baseia na determinação de quando é seguro executar um evento, garantindo que nenhum processo receberá um evento com *timestamp* menor do que seu relógio lógico. Se isso não é garantido, o processo fica parado, ocasionando uma queda de desempenho devido à ociosidade. Também podem ocorrer situações de *deadlock*.

Um conceito utilizado nos protocolos conservativos é o *lookahead*, que é definido como o intervalo de tempo (em termos de relógio lógico) em que um processo pode prever o que acontecerá no futuro com absoluta certeza (MISRA, 1986). Com este conceito, os processos podem determinar um intervalo de tempo no qual podem simular eventos de sua lista de eventos futuros sem a possibilidade da ocorrência de erros de causa e efeito. Em outras palavras, um *lookahead* l indica que o processo pode garantir que nenhuma mensagem de evento será recebida com *timestamp* menor do que $t + l$, onde t é o tempo lógico de simulação do processo (LVT).

Uma forma de um processo determinar se pode simular um evento de sua lista de eventos futuros é observar o *timestamp* das últimas mensagens recebidas dos processos com o qual ele recebe mensagens. Se seu LVT for menor que o menor *timestamp* entre essas mensagens, então o processo pode simular o próximo evento de sua lista de eventos futuros. Isso garante que esse processo não irá receber uma mensagem com *timestamp* menor que o seu (mensagem *straggler*), pois os outros processos têm um LVT maior.

Uma situação de *deadlock* pode ocorrer quando todos os processos esperam devido a possibilidade de ocorrência de erros de causalidade caso simulem o próximo evento da lista de eventos futuros. Os protocolos conservativos apresentam diferenças na forma como o *deadlock* é tratado. Alguns evitam o *deadlock* e outros permitem que ele ocorra recuperando o sistema quando da sua ocorrência.

Dentre os meios para a prevenção de *deadlocks* tem-se a técnica do envio de

mensagens nulas para cada evento processado. Estas mensagens não representam eventos trocados entre processos ou algum conteúdo que promova atividade para o sistema, mas contém apenas o relógio lógico do processo que a enviou permitindo que cada processo determine os relógios lógicos dos processos com os quais tem canais de comunicação, podendo identificar o menor relógio lógico entre eles.

Uma desvantagem desta técnica é a consequente sobrecarga nos canais de comunicação devido ao envio das mensagens nulas. Algumas modificações nessa abordagem podem ser feitas para reduzir a sobrecarga na rede. Uma delas, é fazer o envio de mensagens nulas sob demanda. O envio de mensagens nulas ocorre periodicamente, ou quando o processo que possui o menor relógio lógico também possui sua fila de eventos futuros vazia.

Quando a frequência de *deadlocks* é pequena, o uso de mensagens nulas se torna ineficiente. Outra abordagem, desenvolvida por Chandy e Misra (1981), é a detecção e recuperação de *deadlocks*. O mecanismo se caracteriza por possuir um tipo especial de mensagem que atua como um marcador percorrendo todos os canais da rede. Assim que um processo recebe o marcador, e não possuir outras mensagens para serem enviadas, este o envia através de uma determinada rota.

O marcador tem algumas informações para identificar um *deadlock*. Cada processo tem um *flag* para indicar se enviou ou recebeu alguma mensagem desde a última vez que o marcador passou por ele. Um processo é branco se não enviou e nem recebeu mensagens e preto caso contrário. Inicialmente, todos os processos são pretos. Se N é o número de canais na rede, um *deadlock* é detectado quando os últimos N marcadores que um processo recebeu forem brancos. Existem duas desvantagens nesse mecanismo. A primeira é a sobrecarga de comunicação causada pela transmissão do marcador. A segunda é que esse mecanismo apenas detecta o *deadlock* havendo assim a necessidade de um mecanismo para resolução do *deadlock* (Misra, 1986).

Existem outros protocolos conservativos que não foram abordados neste texto. Alguns desses protocolos são: SRADS (REYNOLDS, 1982), *Appointments* (NICOL;

REYNOLDS, 1984), *Turner Carrier-null Scheme* (CAI; TURNER, 1990) e SPaDES/-Java (TEO; NG, 2002).

2.1.2 Protocolos Otimistas

Os protocolos otimistas, ao contrário dos protocolos conservativos, permitem a ocorrência dos erros de causalidade, já que os processos da simulação não são impedidos de tratar um evento da sua lista de eventos futuros mesmo que isso leve à ocorrência de erros de causalidade. Neste caso, se um erro ocorrer, os protocolos otimistas possuem mecanismos para resolver os erros e assim permitir que a simulação se mantenha consistente. Uma forma de recuperar a simulação para um estado consistente é realizar o que se denomina de *rollback*. Basicamente, o *rollback* retorna a computação a um ponto especificado, consistente, para assim simular novamente os eventos afetados (WANG, 1997).

A grande vantagem dos protocolos otimistas sobre os conservativos (FUJIMOTO, 1990; FERSCHA, 1995) é permitir uma maior exploração do paralelismo nas situações em que "poderiam" ocorrer erros de causalidade, mas que não ocorrem. O tratamento da ocorrência de erros de causalidade permite aos protocolos otimistas explorar o paralelismo intrínseco dos modelos deste tipo de simulação e, desta forma, apresentar desempenhos superiores aos dos protocolos conservativos.

Os protocolos otimistas apresentam em comum as seguintes características:

- Um relógio lógico local ou *local virtual time* (LVT), que representa o tempo da simulação e permite a cada processo acompanhar a ordem cronológica da simulação dos eventos. A cada evento simulado é calculado um novo valor para o LVT, conforme definições no modelo simulado;
- Um tempo mínimo global ou *global virtual time* (GVT), que é o menor LVT de todos os processos da simulação e mensagens em trânsito. O cálculo GVT é utilizado para saber que nenhum processo do sistema irá criar um evento com o *timestamp* menor do que o valor do GVT.

- Armazenamento de estados para viabilizar o procedimento de *rollback*.

Os protocolos otimistas se diferenciam entre si na forma como realizam o tratamento dos erros de causa e efeito, ou seja, no procedimento realizado para recuperar a computação a um estado consistente. Serão abordados neste texto dois protocolos otimistas: o *TimeWarp* e o *Rollback* Solidário (MOREIRA, 2005).

2.1.2.1 Time Warp

No protocolo *Time Warp*, criado por Jefferson (1985), os processos da simulação podem tratar eventos da sua lista de eventos futuros no seu tempo de simulação local (LVT). Os mesmos são tratados sem levar em consideração a possibilidade de se criar situações inseguras, ou seja, situações que podem levar ao erro de causalidade.

O *Time Warp* apresenta dois procedimentos principais: o primeiro, um procedimento local, garante que os eventos sejam processados em ordem correta e é implementado independentemente em cada processo; e o segundo, um procedimento de controle global, que é responsável pelo controle de memória e pelo cálculo do GVT (JEFFERSON, 1985).

Cada processo lógico simula seus eventos independentemente sem considerar o estado da simulação de outros processos. Assim, erros de causa e efeito só ocorrem através da troca de eventos (por meio de mensagens) entre os processos. Quando uma mensagem provoca um erro de causalidade, os processos devem tratar essa situação através de procedimentos de *rollback* (ZIMMERMAN; KNOKE; HOMMEL, 2006).

Quando uma mensagem provoca um erro de causa e efeito (mensagem *straggler*), o processo que a recebe deve recuperar a computação para um estado consistente. No protocolo *Time Warp* os seguintes passos são realizados:

- Identificar o estado salvo com relógio lógico anterior ao *timestamp* da men-

sagem *straggler* recebida;

- No retorno ao estado consistente verificar se eventos foram criados para outros processos da simulação. Em caso afirmativo, devem ser enviadas mensagens informando que tais eventos não devem existir. Essas mensagens são denominadas anti-mensagens;
- Restaurar o estado coerente identificado e inserir o evento contido na mensagem *straggler* na lista de eventos futuros;
- Enviar anti-mensagens, para anular as mensagens da simulação previamente enviadas durante o intervalo de retorno do *rollback*.

Ao receber uma anti-mensagem, um processo da simulação deve realizar uma das duas seguintes operações:

1. Remover o evento correspondente à anti-mensagem da sua lista de eventos futuros, se o *timestamp* desse evento for maior que o relógio lógico no momento do recebimento da anti-mensagem;
2. Tratar o evento correspondente à anti-mensagem como um erro de causa e efeito, e portanto, realizar um *rollback* da mesma forma daquele realizado para uma mensagem *straggler* (ZENG; CAI; TURNER, 2004). Esses *rollbacks* ocasionados por anti-mensagens são denominados *rollbacks* secundários (FUJIMOTO, 1990).

Em decorrência desse procedimento para a recuperação da computação, três estruturas são necessárias para cada processo da simulação:

1. Uma lista de eventos futuros, cuja função é armazenar, de forma ordenada em função do *timestamp*, os eventos a serem simulados pelos processos;
2. Uma lista de mensagens enviadas, cuja finalidade é gerar as anti-mensagens correspondentes quando da ocorrência de *rollbacks*;

3. Uma lista de estados salvos, para permitir que um processo restaure sua lista de eventos futuros, lista de mensagens enviadas, relógio lógico entre outras informações sobre a simulação, quando da ocorrência de *rollbacks*.

Alguns problemas podem surgir no protocolo *Time Warp*. Um deles é devido aos *rollbacks* em cascata que provocam perda de desempenho (RAJAEI, 2007).

Outra questão que influencia na execução do protocolo *Time Warp* é o uso de memória. À medida que a simulação avança, a lista de estados salvos pelos processos aumenta, havendo um crescimento do uso de memória até o seu limite. Isso exige o uso de algum mecanismo de liberação de memória. Para isso é necessário identificar quais estados salvos não serão mais utilizados. Em geral, a liberação de memória para o sistema é implementada em um módulo denominado *Garbage Collection*, o qual deverá utilizar um algoritmo para determinar o GVT, permitindo aos processos descartar os estados salvos com LVTs menores que o valor do GVT.

Existem algoritmos para o cálculo do GVT, como o desenvolvido por Fujimoto e Hybinette (1997). Nesse algoritmo, um processo coordenador realiza o cálculo do GVT. Para toda mensagem enviada por um processo da simulação deve haver outra mensagem de confirmação pelo processo receptor da mensagem. Quando se inicia o cálculo do GVT pelo processo coordenador, este envia uma mensagem a todos os processos da simulação avisando que a partir daquele momento eles devem interromper a simulação para que seja realizado o cálculo do GVT. Ao confirmar o recebimento dessa mensagem, o processo coordenador envia uma nova mensagem para que todos os processos retornem uma mensagem com o seu LVT. Em seguida, ao receber as mensagens dos processos com os valores mínimos obtidos, o processo coordenador enviará uma mensagem com o valor do GVT do sistema. Uma desvantagem desse algoritmo é a necessidade de interromper a simulação durante o cálculo do GVT.

Outro algoritmo para cálculo do GVT é o proposto por Mattern (1993). Esse algoritmo utiliza o conceito de corte consistente que é formado por um conjunto

de eventos recebidos através de mensagens nos quais seus emissores pertencem ao corte. Com o corte consistente, pode-se obter o GVT como o menor LVT ou *timestamp* do conjunto de eventos pertencentes ao corte consistente.

A figura 3 resume o funcionamento de um processo lógico no protocolo *Time Warp*.

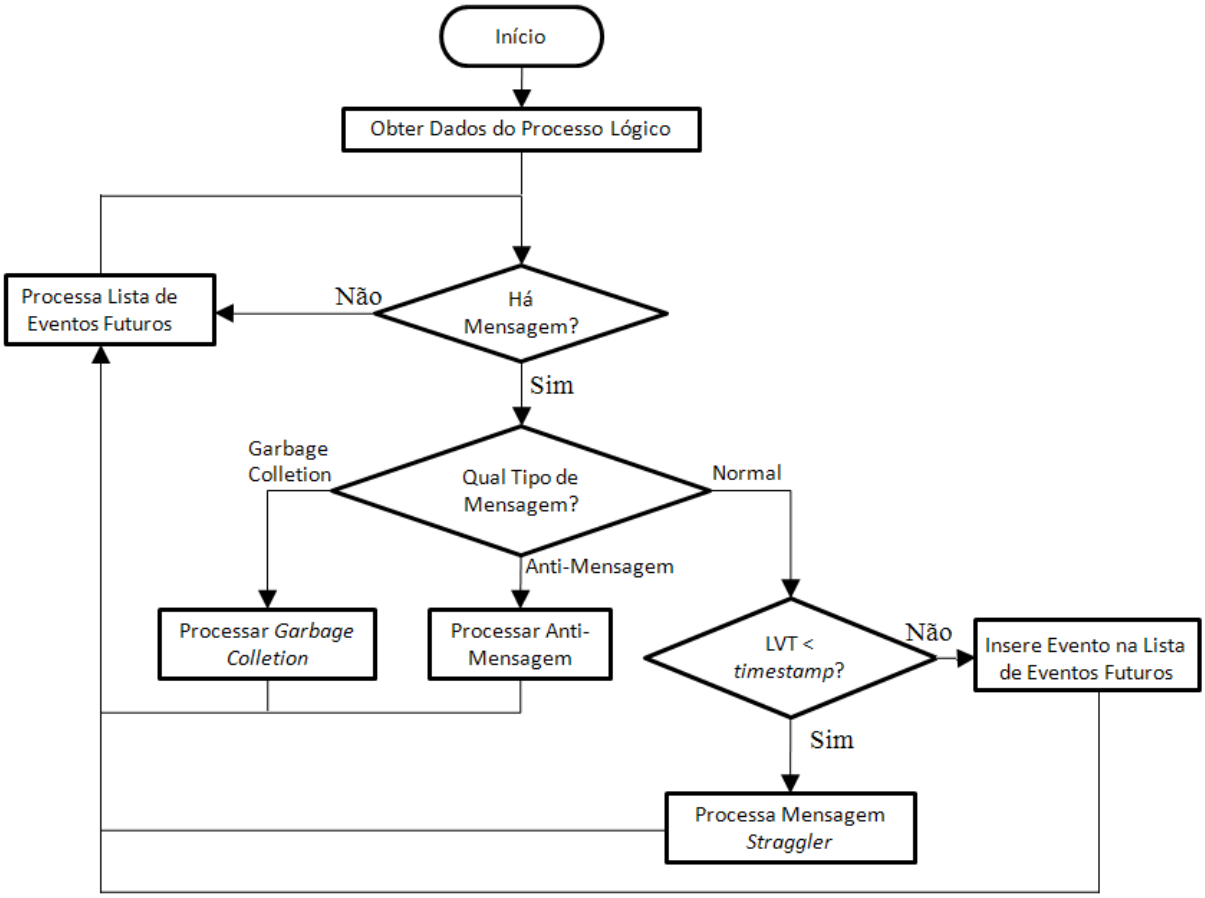

Figura 3: Funcionamento do *Time Warp*

Uma última questão a ser considerada sobre o *Time Warp* é sobre o mecanismo de salvamento de estados para permitir o retorno na ocorrência de *rollbacks*. Algumas estratégias de salvamento mais conhecidas são:

1. *Copy State Saving*: salva todos os estados da simulação distribuída. Na ocor-

rência de um erro de causalidade, o processo retorna para o estado imediatamente anterior ao tempo lógico do evento que gerou *rollback* (FUJIMOTO, 2000).

2. *Sparse State Saving*: é uma modificação do *Copy State Saving* em que os estados são salvos em um intervalo de iterações e não a cada evento. Assim, apresenta a vantagem de utilizar uma menor quantidade de memória, mas, por outro lado, quando da ocorrência de um *rollback*, este encontrará um estado salvo que, geralmente, não terá um LVT imediatamente anterior ao evento que gerou o *rollback*, havendo necessidade de reprocessar eventos que não foram diretamente afetados pelo *rollback* (SKOLD; RONNGRENN, 1996).

3. *Incremental State Saving*: salva apenas as variáveis de estado que foram modificadas. Há maior economia de memória e maior rapidez no salvamento dos estados. Entretanto, a restauração do estado do processo se torna decremental, pois restaura-se o estado imediatamente anterior ao estado atual do processo até que seja reconstituído o estado necessário à continuação da simulação. Essa restauração decremental representa uma desvantagem por degradar o desempenho do mecanismo de *rollback* (STEINMAN, 1993).

4. *Hybrid State Saving*: é uma mistura dos mecanismos *Incremental* e *Sparse State Saving*. Um processo lógico salva periodicamente seus estados, e também utiliza o método incremental a cada evento. Assim, os processos são capazes de recuperar estados fazendo uso dos dois métodos (CORTELLESSA; QUAGLIA, 1998).

Existem variações do *Time Warp* que buscam otimizar seu desempenho. Dentre elas, tem-se a proposta de Liu e Wainer (2008), em que é apresentado o protocolo *Lightweight Time Warp* (LTW), o que inclui uma regra baseada em um mecanismo de agendamento de eventos, utilizando dois tipos de filas de eventos, uma técnica de salvamento de estados globais e um algoritmo de *rollback* novo que recupera os processos sem enviar anti-mensagens. Há também aplicação de

técnicas de inteligência artificial sobre o *Time Warp* como no trabalho de Wang e Tropper (2009).

Lobato, Ulson e Santana (2004) apresentam uma taxonomia para os mecanismos de simulação baseados no *Time Warp*. Esta taxonomia é útil para decidir qual variação do protocolo *Time Warp* escolher para um modelo de simulação específico.

2.1.2.2 Rollback Solidário

Outro protocolo otimista, desenvolvido por Moreira (2005), denomina-se *Rollback* Solidário. Esse protocolo não faz uso de anti-mensagens durante o *rollback*. O tratamento do *rollback* é realizado através do uso do conceito de *checkpoints* globais consistentes (BABAOGLU; MARZULLO, 1993). Com relação ao *Time Warp*, a maior diferença do protocolo *Rollback* Solidário consiste na forma como é tratada uma mensagem *straggler*.

Como foi mencionado, no protocolo *Time Warp*, quando um processo realiza um *rollback*, ele envia anti-mensagens para cada mensagem que foi previamente enviada cujo *timestamp* é maior que o LVT do evento de *rollback*.

Ao contrário do *Time Warp*, no *Rollback* Solidário, quando uma mensagem *straggler* é recebida, um processo coordenador faz uma verificação identificando quais processos são influenciados por essa mensagem *straggler* e, portanto, quais devem realizar *rollback*.

O não uso de anti-mensagens proporciona os seguintes benefícios:

1. Economia de memória (por ser desnecessário criar uma lista de mensagens enviadas);
2. Diminuição do tráfego na rede;
3. Não ocorrência de *rollbacks* em cascata.

Um ponto em comum com o protocolo *Time Warp* é o salvamento de estados para que seja possível recuperar um estado quando da ocorrência de *rollbacks*. No *Rollback* Solidário o mecanismo é semelhante ao *Sparse State Saving*.

A forma como é realizado o tratamento da mensagem *straggler* pode ser resumida pelos seguintes passos:

1. O processo que recebeu uma mensagem *straggler* verifica em seus *checkpoints* salvos qual possui um tempo lógico mais próximo e menor ao *timestamp* do evento recebido;

2. Esse processo restaura o estado encontrado;

3. O processo informa, ao processo observador, a ocorrência do *rollback* e para qual *checkpoint* local ele retornou;

4. Com a informação do *checkpoint* local de retorno recebido, o processo observador determina o *checkpoint* global consistente ao qual o *checkpoint* local recebido pertence. Pode existir mais de um *checkpoint* global consistente, que contenha o *checkpoint* local do processo que realizou *rollback*. Nesse caso, será escolhido aquele que causar menor prejuízo à computação;

5. Determinado o *checkpoint* global consistente, o processo observador envia para todos os processos da simulação uma mensagem contendo o *checkpoint* global consistente. Desta forma, os processos podem determinar para qual *checkpoint* local salvo eles devem retornar;

6. Os processos da simulação então atualizam seu *checkpoint* local e realizam *rollback*;

7. Após esses passos é retomada a simulação pelos processos.

Como existem várias combinações de *checkpoints* locais que formam *checkpoints* globais consistentes, pode-se definir cada *checkpoint* global como uma linha

de recuperação. Na ocorrência de *rollback*, o processo observador irá possuir um conjunto de linhas de recuperação, ou seja, *checkpoints* globais consistentes, que podem ser utilizados para realizar *rollback*.

Para que o sistema retorne, quando da ocorrência de *rollback*, é necessário ter um mecanismo que obtenha os *checkpoints* globais consistentes durante a simulação. Uma abordagem semi-síncrona pode ser adotada para a obtenção dos *checkpoints* globais consistentes (MANIVANNAN; SINGHAL, 1999). Nessa abordagem, há um processo observador passivo durante a simulação, e se torna ativo apenas para coordenar a realização do *rollback*.

Os processos devem possuir um relógio lógico vetorial, que aqui será denominado de vetor de dependências. A função deste vetor é a identificação da relação de causalidade entre os *checkpoints* do sistema.

Ao enviar uma mensagem, o processo emissor deve anexar seu vetor de dependências a essa mensagem, e assim, ao recebê-la, o processo receptor atualiza seu vetor de dependências. Essa atualização é semelhante àquela que pode ser encontrada na teoria dos relógios lógicos vetoriais e segue as seguintes regras:

1. A atualização do vetor de dependências consiste em gerar um novo vetor, onde cada elemento será o maior elemento entre o vetor de dependências original e o vetor de dependências recebido;

2. Se o processo, ao receber a informação de um novo *checkpoint*, havia enviado uma ou mais mensagens após seu último *checkpoint* local, então ele realizará um *checkpoint* forçado.

Ao realizar um *checkpoint*, o processo deve incrementar sua posição no vetor de dependências, em seguida deve enviar seu vetor ao processo observador. A mensagem enviada ao processo observador deve conter também um vetor contendo os relógios lógicos dos *checkpoints* locais armazenados. Isto também é necessário quando os processos trocam mensagens entre si. Com esse vetor, os processos

receptores sabem se devem eliminar um evento de sua lista de eventos futuros quando da ocorrência de um *rollback*.

Ao receber um vetor de dependências, o processo observador pode armazená-lo de duas formas: no primeiro caso, ele atualiza uma matriz quadrada $n \times n$ (onde n é o número de processos da simulação), e no segundo, o observador adiciona este vetor em uma lista (uma lista para cada processo da simulação) de vetores de dependências. Na primeira forma de armazenamento, apesar de simples implementação, não há, em todos os casos, como o processo observador escolher de forma ótima qual *checkpoint* global consistente os processos devem retornar. Na segunda forma de armazenamento, apesar da maior complexidade de implementação e maior tempo de execução, o processo observador é capaz de determinar com precisão qual é o melhor *checkpoint* global consistente para onde a computação deve retornar.

2.2 Considerações Finais

Este capítulo apresentou uma fundamentação básica da Simulação Distribuída. Foram discutidas as duas abordagens existentes para realizá-la: o MRIP, que consiste na execução de várias replicações de um programa de simulação, e o SRIP, onde que consiste na paralelização de um programa de simulação.

Uma ênfase foi dada para a abordagem SRIP, que apresenta duas formas para o tratamento dos erros de causalidade: a conservativa, que evita a ocorrência destes erros de causalidade, e a otimista, que permite a ocorrência dos erros de causalidade, mas faz a recuperação da computação quando eles ocorrem. Os protocolos otimistas *Time Warp* e o *Rollback* Solidário foram explanados neste capítulo.

Os mecanismos de migração, que serão expostos no capítulo 6, são voltados para o protocolo *Time Warp*. Entretanto, será discutida uma possibilidade de adaptar os referidos mecanismos para o protocolo *Rollback* Solidário.

3 Migração de Processos

Este capítulo tem por objetivo introduzir os conceitos de migração de processos, bem como, os procedimentos envolvidos na mesma.

Um processo pode ser definido como um programa em execução. Para sua execução são necessários diversos recursos, tais como, tempo de processador, memória, arquivos e dispositivos de E/S. Esses recursos são disponibilizados quando o processo é criado ou quando está em execução (SILBERSCHATZ; GALVIN; GAGNE, 2009).

A migração de processos consiste na transferência de processos em execução, entre diferentes computadores de um sistema distribuído. Essa transferência deve ocorrer de forma a continuar a execução de um processo no computador destino a partir do ponto onde parou no computador de origem e de forma a não alterar o resultado de um programa (SMITH, 1988; MILOJICIC; DOUGLIS; PAINDAVEINE, 2000). Dessa forma, uma grande quantidade de informações referentes ao estado do processo deve ser transferida durante a migração do processo para que este possa recuperar seu estado.

3.1 Motivação Para a Migração de Processos

A migração de processos pode ser destinada a melhorar (SMITH, 1988):

- **O balanceamento de carga do sistema:** ao mover processos a partir

de computadores muito sobrecarregados, a carga pode ser equilibrada entre todos os computadores para melhorar o desempenho global.

- **O desempenho de comunicações:** processos que interagem intensamente podem ser migrados para o mesmo computador para reduzir o custo das comunicações. De forma semelhante, quando um processo está realizando a análise de dados em algum arquivo ou conjunto de arquivos maiores do que o tamanho do processo, pode ser vantajoso mover o processo para os dados ao invés de mover os dados até o processo.

- **A disponibilidade:** processos de longa duração podem precisar se deslocar para sobreviverem a falhas para as quais um aviso prévio é possível. Se o sistema operacional fornece notificação, e se um processo necessita continuar, este pode migrar para outro sistema ou garantir que ele pode ser reiniciado no sistema atual, em algum momento mais tarde.

- **A utilização de recursos especiais:** um processo pode mover-se para aproveitar capacidades únicas de *hardware* ou *software* em um nó particular.

De alguma forma, os objetivos de um programa paralelo podem sofrer conflitos entre si, e a política de migração de processos deve definir a importância de cada um dos objetivos da migração citados.

3.2 Questões Envolvidas na Migração de Processos

De ascordo com Stallings (2009), uma série de questões precisa ser tratada no projeto de um sistema que oferece o recurso de migração de processos. As principais são:

- Quem inicia a migração?

- Que parte do processo é migrado?

- O que acontece com mensagens e sinais pendentes?

3.2.1 Inicialização da Migração

A inicialização da migração depende do objetivo a ser atingido com ela. Se o objetivo é o balanceamento de carga, então algum módulo no sistema operacional que está monitorando a carga do sistema será responsável por decidir quando a migração deve ocorrer. Para determinar quando migrar, o módulo terá que se comunicar com os módulos de outros sistemas, trabalhando em conjunto para determinar o novo posicionamento dos processos.

Se o objetivo é levar processos a recursos específicos de *hardware*, então um processo pode migrar conforme sua necessidade em determinado momento. Neste caso, o módulo do sistema deve conhecer as necessidades dos processos e os recursos disponíveis em cada computador do sistema distribuído.

Uma proposta para tomar a decisão de iniciar a migração foi feita por Mello e Senger (2004) onde um computador analisa a ocupação de cada processo e seleciona o mais adequado para a migração. A análise e seleção são feitos através de um fator de migração. Esse fator reflete o quanto o computador ocupado será liberado e quanto o computador de destino será sobrecarregado, tendo em vista a migração de cada processo.

3.2.2 Migração do Estado do Processo

Quando um processo é migrado, é necessário destruí-lo no seu sistema de origem e recriá-lo no sistema alvo. Isto é considerado como movimento de processo e não como uma replicação de processo. Para que o movimento ocorra de forma adequada, é necessário que seus dados, bem como seu ponto de execução, sejam recuperados. Além disso, canais de comunicação entre o processo em migração e outros processos devem ser atualizados.

De acordo com Milojicic, Douglis e Paindaveine (2000), do ponto de vista de desempenho, existem algumas estratégias para a recuperação dos dados, permitindo a recuperação do estado do processo antes de migrar:

- *Eager-all*: há a transferência de todo o espaço de endereçamento em tempo de migração. Nenhum vestígio do processo é deixado para trás no antigo sistema. No entanto, se o espaço de endereçamento é muito grande e se o processo não necessita desse espaço, então sua completa transferência é um custo desnecessário.

- *Precopy*: o processo continua sua execução no sistema de origem enquanto seu espaço de endereçamento é copiado. Páginas do processo que são modificadas durante a cópia devem ser recopiadas num segundo momento. Dessa forma, há uma redução no tempo no qual o processo fica impedido de executar devido à cópia de seu espaço de endereçamento. Uma proposta de otimização desta técnica foi feita por Ibrahim et al. (2011) em que um algoritmo, que se baseia na taxa de modificação de memória, realiza o controle da migração.

- *Eager-dirty*: apenas as páginas que estão na memória e que foram modificadas são transferidas. Um bloco adicional do espaço de endereçamento virtual é transferido minimizando a quantidade de dados que são transmitidos. Esta técnica exige que o computador original continue envolvido no novo processo oferecendo entradas na tabela de página ou segmento.

- *Copy-on-reference*: esta é uma variação do *Eager-dirty* no qual páginas são apenas transferidas quando são referenciadas. Esta estratégia tem o menor custo para o início da migração, porém uma falha de página leva a um maior tempo de tratamento.

- *Flushing*: as páginas do processo são liberadas da memória principal e armazenadas na memória secundária. Assim, os dados são acessados pelo novo processo na memória secundária do computador de origem. Esta estratégia

libera o sistema de origem de armazenar os dados em memória principal, e portanto, pode alocá-la a outros processos.

3.2.3 Mensagens e Sinais na Migração

Como os processos podem se comunicar através de mensagens e sinais, durante sua migração, pode ocorrer troca de mensagens e sinais entre eles e um mecanismo de armazenamento dessas informações trocadas deve ser implementado com a finalidade de redirecioná-las para o processo após sua migração de forma que não haja perda de mensagens.

Poderá ser necessário manter informações no sistema inicial por algum tempo para garantir que mensagens e sinais pendentes sejam recebidos pelo novo processo destinatário.

3.3 Procedimentos Envolvidos na Migração

Mesmo com uma abordagem diferente envolvida na migração de processos em relação às questões mencionadas nas seções anteriores, pode-se descrever um mecanismo geral para a migração de processos. Richmond e Hitchens (1997) apresentaram os seguintes passos:

1. **Decisão de migração:** com informações do sistema, um algoritmo determina a necessidade de migração dentre os processos;
2. **Suspensão da execução dos processos:** uma vez determinada a migração, os processos devem ser suspensos;
3. **Redirecionamento da comunicação:** todas as comunicações com o processo envolvido na migração são redirecionadas para seu novo destino;
4. **Transmissão do estado do processo:** o computador de destino do processo deve receber seu estado e recriar o processo baseado neste estado;

5. **Modificação das referências:** os identificadores ou referências relacionadas ao processo no computador de origem são modificados quando o processo passar por migração e reiniciar sua execução no computador de destino;

6. **Transmissão de mensagens:** mensagens que são recebidas pelo processo durante sua suspensão devem ser transmitidas para o computador de destino de modo a permitir sua recepção pelo novo processo;

7. **Reinicialização do processo:** Com o estado do processo recebido, realiza-se a atualização de variáveis e mensagens. Por conseguinte, o processo pode ser reiniciado no computador de destino;

8. **Eliminação do processo de origem e suas informações:** uma vez que o processo já foi reiniciado, pode ser eliminado junto com suas informações na sua origem.

Nota-se que esses passos não precisam ser executados na sequência dada, salvo a decisão de migração que inicializará todo esse procedimento. Esses passos ainda podem ser executados paralela ou concorrentemente.

Gerofi, Fujita e Ishikawa (2010) realizaram uma implementação de um *middleware* para migração de processos baseada em um procedimento semelhante ao descrito. Esse trabalho tem como objetivo resolver o problema de balanceamento de carga para aplicações virtuais no nível do sistema operacional.

3.4 Implementação da Migração de Processos

A implementação da migração de processos pode ser realizada no núcleo do sistema operacional, ou seja, no *kernel* do sistema, ou em modo usuário. No caso da implementação em modo usuário, esta ainda pode ser feita com o uso de uma biblioteca fornecida pelo sistema operacional adicionada ao programa executável, ou ainda pode resultar da implementação na própria aplicação (MILOJICIC; DOUGLIS; PAINDAVEINE, 2000; JANKOWSKI et al., 2005).

3.4.1 Implementação em modo Kernel

A implementação da migração em modo *kernel* é a de maior dificuldade quando comparado com as outras formas de implementação. Isso se deve ao tratamento de instâncias como múltiplas *threads*, identificadores de arquivos abertos, soquetes e *pipes* de comunicação (GIOIOSA et al., 2005; MILOJICIC; DOUGLIS; PAINDAVEINE, 2000). Além disso, a implementação em modo *kernel* é dependente da plataforma computacional utilizada, limitando sua portabilidade. Por outro lado, esta forma de implementação apresenta como vantagens a transparência para as aplicações em modo usuário, a independência do código fonte da aplicação, bem como a independência da linguagem de programação no desenvolvimento da aplicação.

Os algoritmos para a tomada de decisão de migração e escalonamento de processos entre os computadores disponíveis apenas podem observar dados do sistema como índices de carga e dados dos processos, como falta de páginas e uso de dispositivos de entrada e saída. Isso pode significar uma limitação para a tomada de decisão, já que o próprio funcionamento da aplicação pode afetar a melhor forma de escalonar os processos. Existem soluções para essa situação onde um algoritmo de escalonamento pode ser implementado dentro da própria aplicação e essa deve se comunicar com o sistema de migração de processos através de chamadas de sistema (DOUGLIS; OUSTERHOUT, 1991). Essa solução leva à perda de transparência quanto à migração por parte da aplicação.

3.4.2 Implementação com Uso de Bibliotecas de Sistema

A implementação da migração de processos fazendo uso de bibliotecas de sistema consiste basicamente na utilização de chamadas de sistema e na ligação dessas bibliotecas no programa executável durante a sua compilação. Essa abordagem tem a vantagem de possuir menor complexidade de implementação comparada com aquela realizada em modo *kernel*, permitindo a implementação de políticas de escalonamento. Ainda apresenta certa transparência para a aplicação (MILOJICIC;

DOUGLIS; PAINDAVEINE, 2000).

Como se trata de uma solução em ambiente de usuário, não é possível obter o estado de todo o processo de dentro do próprio processo. Consequentemente, essas restrições levam a uma perda de transparência na migração.

Quanto a portabilidade, essa forma de implementação é dependente do sistema operacional utilizado devido ao uso de bibliotecas de sistema. Outra restrição geralmente imposta pela implementação com uso de bibliotecas de sistema é com relação à linguagem de programação na qual a biblioteca é oferecida limitando o desenvolvimento da aplicação a uma determinada linguagem de programação ou a uma família de linguagens de programação.

Com relação aos algoritmos de escalonamento, o uso de bibliotecas de sistema permite uma maior facilidade na tomada de decisão, pois é possível implementar o algoritmo de escalonamento em nível de usuário. Por outro lado, há uma maior dificuldade de obter informações de sistema, como frequência de *page faults* e uso do processador, que são informações úteis para decidir a adequada localização dos processos entre os computadores disponíveis no sistema.

3.4.3 Implementação em Modo Usuário

A implementação em modo usuário tem como característica uma menor complexidade de implementação, porém o código obtido é especificado para uma dada aplicação e deve implementar de maneira própria os mecanismos envolvidos na migração. Consequentemente, pode-se dizer que a implementação em modo usuário não é transparente (MILOJICIC; DOUGLIS; PAINDAVEINE, 2000).

Apesar de sua restrição a uma dada aplicação e sua não transparência, essa forma de implementação é portável entre diferentes plataformas computacionais devido à possibilidade de recompilação do código fonte entre esses diferentes sistemas.

3.5 Checkpointing

Em geral, a principal razão para o uso de *checkpoints* é aumentar a tolerância a falhas. A própria Simulação Distribuída, no caso dos protocolos otimistas, precisa de *checkpoints* para a recuperação da simulação quando se tem erros de causalidade. Em aplicações de longa duração, falhas em um dos computadores podem levar à grandes perdas de informações e processamento. Por isso, o uso do *checkpointing* nessas situações. Essa técnica ainda pode ser usada para depuração de aplicações.

No caso da migração, o termo *Checkpointing* consiste no salvamento do estado do processo em arquivos para seu futuro envio ao computador de destino (o que pode ser feito mediante um servidor de arquivos) de forma a permitir a reconstrução do processo (ROMAN, 2002; CHAUDHARY; WALTERS; JIANG, 2010). Uma imagem do processo é salva durante um *checkpointing*. O processo então se reinicia exatamente no ponto representado por um dado *checkpoint*.

A implementação do *checkpointing* pode ser realizada no núcleo do sistema operacional (*kernel*), com uso de bibliotecas de sistema e chamadas de sistema, ou ainda, pode ser implementada em ambiente de usuário.

3.6 Sistemas que Oferecem Migração de Processos

Com vários anos de pesquisa, soluções e sistemas para migração de processos foram sendo desenvolvidos. Alguns exemplos são:

- Mosix (HAKIM; JAIS; SALWA, 2008): provê gerenciamento de *clusters*, *mult-clusters* ou *clouds* de alto desempenho. Esse sistema operacional apresenta mecanismos que permitem a distribuição dinâmica da carga de trabalho.
- OpenMosix (LOTTIAUX et al., 2005): é uma extensão do *kernel* do sistema operacional Linux para a criação de *clusters* computacionais adquirindo o

conceito de Sistema de Imagem Única (*Single System Image* - SSI), transformando uma rede clássica de computadores em um super-computador para aplicações desenvolvidas em Linux.

- OpenSSI (LOTTIAUX et al., 2005): é um sistema para obtenção de *clusters* de Sistema de Imagem Única baseado em Linux. Seu projeto é decorrente das tecnologias para Unixware e outras modalidades *open source*, para fornecer um ambiente completo, de alta disponibilidade e SSI para Linux. Os objetivos do *cluster* OpenSSI incluem alta disponibilidade, escalabilidade e gerenciabilidade, construídos sobre servidores padrões.

- Kerrighed (LOTTIAUX et al., 2005): é uma base de sistema operacional para Sistema de Imagem Única destinado para *clusters* construídos a partir de computadores padrões de mercado com baixo custo. Um sistema operacional SSI dá a ilusão de uma máquina de processamento simétrico (*Symmetric Multi-Processing* - SMP) aos programas que são executados nesse sistema. Kerrighed é uma extensão do *kernel* do Linux. Um *cluster* rodando Kerrighed não é uma máquina de SMP física real.

3.7 Considerações Finais

Este capítulo apresentou uma revisão sobre os conceitos envolvidos na migração de processos bem como seus mecanismos envolvidos. Foram discutidas as questões envolvidas na migração, ou seja, quem inicia a migração, que parte do processo é migrado e o que é feito com mensagens e sinais pendentes. Um procedimento geral, que resolve as mencionadas questões, foi apresentado.

O conceito de *checkpointing* foi definido, e, para a migração de processos, envolve o salvamento do estado do processo para sua posterior recuperação quando reiniciado em outro computador.

As diferentes formas de implementação, com suas vantagens e desvantagens,

e alguns dos diferentes sistemas que empregam a migração foram descritos. Esta revisão se faz útil para entender os mecanismos de migração propostos neste trabalho.

4 Escalonamento de Processos na Simulação Distribuída

O escalonamento de processos consiste basicamente na alocação de recursos computacionais para a execução de um programa formado por vários processos. Em um ambiente distribuído é necessário o uso de um algoritmo ou política de escalonamento para definir quais recursos e por quanto tempo serão alocados para cada processo a fim de se obter o maior desempenho possível.

Esse algoritmo deve ser projetado de acordo com requisitos diferentes, e frequentemente, deve atender requisitos opostos entre si o que dificulta a escolha por uma política adequada. Além desses requisitos, deve-se levar em consideração outros elementos que influenciam o desempenho da aplicação. Alguns exemplos desses elementos são (LOW et al., 1999; FERRARI; ZHOU, 1987):

1. A plataforma de computação;
2. O *hardware* disponível;
3. O funcionamento da aplicação;
4. A carga de uso dos sistemas.

Em geral, os algoritmos de escalonamento de processos são construídos para máquinas paralelas distribuídas procurando balancear a carga gerada por programas paralelos que consistem em vários processos. A decisão de quais recursos

computacionais serão disponibilizados para cada processo das aplicações é feita utilizando-se informações sobre a plataforma e sobre as aplicações.

Um programa de simulação distribuída é afetado pela máquina paralela e pelas características do próprio modelo de simulação. Existem várias técnicas para otimizar o desempenho de um programa de simulação distribuída. As técnicas que mais se destacam são as técnicas de escalonamento e balanceamento de carga (CAROTHERS; FUJIMOTO, 2000).

A literatura apresenta muitos algoritmos de escalonamento para a simulação distribuída com diferentes conceitos e mecanismos de funcionamento. Carvalho (2008) apresenta uma classificação dessas políticas de escalonamento considerando o tempo em que ocorre o mapeamento dos processos.

Apesar de algoritmos de balanceamento para aplicações em geral não serem ideais para programas de simulação (BOUKERCHE; DAS, 1997; DEELMAN; SZYMANSKI, 1998), Voorslouys (2006) demonstrou que bons resultados ainda podem ser obtidos com a aplicação de técnicas de particionamento convencionais que não levam em conta informações inerentes da simulação distribuída.

Cada aplicação paralela tem uma necessidade em especial dependendo do seu funcionamento e granuralidade. Em geral, o desempenho das aplicações paralelas dependem, entre outros fatores, da capacidade de processamento dos núcleos de computação, da latência e da largura de banda de comunicação entre os processos. Consequentemente, os algoritmos desenvolvidos voltados para Simulação Distribuída apresentam pontos em comum e diferenças com relação aos algoritmos para outras aplicações paralelas e apresentam melhores resultados que as técnicas de mapeamento tradicionais (BOUKERCHE; DAS, 1997; DEELMAN; SZYMANSKI, 1998) que utilizam informações como carga dos processadores e número de processos na fila de espera nos computadores.

4.1 Desbalanceamento na Simulação em Protocolos Otimistas

Segundo Carothers e Fujimoto (2000), algumas causas de desbalanceamento de carga na Simulação Distribuída baseado no protocolo *Time Warp* são:

- **avanço não homogêneo dos relógios lógicos dos processos:** os processos da simulação podem estar executando em processadores com diferentes cargas, e, essa diferença pode levar à progressão mais rápida para o futuro de alguns processos da simulação. Os processos que se atrasam podem gerar *rollbacks* longos e frequentes;
- **influências de outras aplicações:** outras aplicações que concorrem com o uso dos recursos de *hardware* podem afetar o balanceamento de carga de um programa de simulação;
- **fatores internos da simulação:** a diferença de parâmetros entre os processos da simulação pode levar ao desbalanceamento de carga na simulação. Entre esses parâmetros cita-se:
  - **população de eventos:** número de eventos processados em um dado período de tempo. Assim, os processos podem impor diferentes sobrecargas ao processador caso o número de eventos tratados em um mesmo período sejam diferentes.
  - **tempo lógico de tratamento de evento:** o tempo médio lógico para tratamento de eventos pode ser diferente entre os processos, assim haverá um progresso mais rápido para o futuro em um processo, caso o tratamento dos eventos apresentem um tempo lógico médio maior neste processo.
  - **comunicação:** a comunicação entre os processos pode levar a um desbalanceamento caso processos distantes entre si (em termos de latência de comunicação), apresentem grande taxa de comunicação.

Conforme será mostrado nas próximas seções, os algoritmos de balanceamento para a simulação distribuída procuram evitar ou diminuir o desbalanceamento dentre os itens mencionados.

4.2 Escalonamento de Processos em Protocolos Otimistas

Nos protocolos otimistas, é permitida a ocorrência de erros de causa e efeito. Consequentemente, um tratamento desses erros deve ser realizado através de *rollbacks* que são realizados para restaurar a computação a um estado consistente. Assim, nota-se que a redução na ocorrência de erros de causalidade leva a um ganho de desempenho. Em geral, portanto, os algoritmos de escalonamento procuram, com informações da simulação, reduzir a ocorrência destes erros.

As próximas seções apresentam algoritmos para balanceamento de carga na simulação distribuída usando o protocolo *Time Warp*.

4.2.1 Algoritmo da Utilização Efetiva do Processador

Esse algoritmo, proposto por Reither e Jefferson (1990), baseia-se na "utilização efetiva" para alterar o mapeamento dos processos entre os processadores. A utilização efetiva representa a porcentagem de computação que não foi utilizada com a realização de *rollbacks*.

Neste algoritmo é calculado, para cada processo lógico, as respectivas utilizações. Com esses valores e com a informação de onde os processos estão sendo executados, determina-se a utilização de cada processador na máquina paralela. De acordo com as cargas de utilização assim obtidas, é determinado se algum processador está sobrecarregado ou ocioso e, desta forma, determina-se um novo mapeamento de processos para obter o balanceamento das cargas em todos os processos.

Nota-se que este algoritmo leva ao equilíbrio no avanço dos relógios lógicos.

Os processos que executam em processadores com utilização mais alta tendem a avançar para o futuro, dando a oportunidade de serem afetados por mensagens de outros processos que estão atrasados em processadores com utilização mais baixa.

4.2.2 Algoritmo da Métrica do Relógio Local

Um algoritmo proposto por Burdorf e Marti (1993), também procura balancear o avanço dos relógios lógicos dos processos. Esse algoritmo obtém, durante a execução da simulação, os valores dos relógios lógicos de cada processo. Com esses valores, são obtidas duas médias, a primeira correspondente à média dos LVTs dos processos de toda a simulação (chamado aqui de α), e a segunda correspondente à média dos LVTs de todos os processos que executam em um determinado processador (chamado aqui de β).

Se para um dado processador $\beta > \alpha$, processos que executam em outros processadores com $\beta < \alpha$ devem ser migrados para o processador com maior β acarretando dessa forma uma diminuição do avanço dos relógios lógicos de cada processador.

Consegue-se assim obter um maior equilíbrio entre o avanço dos relógios lógicos dos processadores da simulação.

4.2.3 Algoritmo com Uso de Heurística de Mapeamento

Neste algoritmo, proposto por Som e Sargent (1998), procura-se reduzir o risco de ocorrência de *rollback* atrasando os eventos cujo tratamento pode aumentar as chances de *rollbacks*.

Esse algoritmo apresenta dois passos básicos. O primeiro é identificar quais eventos que possam sofrer *rollbacks* quando forem executados. O segundo passo é atrasar a execução desse evento com o uso de uma heurística.

Segundo Som e Sargent (1998), um modo eficiente de se conseguir atrasar os

eventos de risco, evitando assim a ocorrência de *rollbacks*, é colocar os processos da simulação envolvidos no evento em um mesmo processador.

4.2.4 Strong Groups

Este algoritmo, também proposto por Som e Sargent (2000), faz uso do conceito de utilização efetiva em conjunto da introdução do conceito de *Strong Groups* (SG). Um *Strong Group* é um conjunto de processos que possui grande influência entre si. Estes conjuntos podem ser identificados com uma avaliação do grafo de interconexão, que representa as probabilidades de comunicação. Uma forma de se obter os *Strong Groups* é realizando o particionamento do grafo que representa a comunicação entre os processos.

Segundo Som e Sargent (2000), os processos pertencentes a um mesmo SG apresentam uma mesma taxa de progresso no que se refere ao relógio lógico, devido às elevadas dependências entre si independente da intervenção no mapeamento dos processos.

Com o uso das informações sobre o mapeamento, é possível equilibrar as velocidades de progresso dos *Strong Groups* da simulação. Para obter essa igualdade, apenas é necessário utilizar um fator sobre a taxa de progresso de todos os processos pertencentes ao mesmo *Strong Group* já que todos apresentam uma mesma taxa de avanço.

A taxa de avanço é proporcional à utilização do processador. Logo, para ajustar a taxa, aplica-se o fator para alterar a utilização conforme a necessidade. Em tempo de execução é feita a migração dos processos para alcançar a utilização calculada. Portanto, consegue-se aproximar a taxa de progresso de todos os *Strong Groups* da simulação diminuindo assim a ocorrência de *rollbacks*.

4.2.5 Algoritmo de Carga BGE

Este algoritmo, também conhecido como *Background Execution*, proposto por Carothers e Fujimoto (2000), apresenta um diferencial em relação aos anteriores, pois trata da adaptação dos processadores disponíveis levando em consideração a influência da carga externa. No começo da simulação, esse algoritmo faz um mapeamento dos processos do sistema unindo-os em *clusters* e tratando da questão de quais processadores usar em tempo de execução, observando o fato de que as cargas externas são muito variáveis e imprevisíveis.

A migração é feita visando um conjunto de processos pertencentes a um *cluster* ao invés de escalonar os processos individualmente. Para determinar como ocorre a migração, existem duas métricas básicas de controle denominadas de PAT (*Processor Advance Time*) e CAT (*Cluster Advance Time*).

A métrica PAT avalia um dado processador e a métrica CAT avalia um *cluster* (conjunto de processos). O PAT dos processadores permite uma avaliação do sistema. É possível, com essa métrica, determinar se existem processos com a taxa de progressão maior que outros, fazendo, se necessário, o balanceamento.

No balanceamento, a primeira etapa consiste em identificar o emissor e o receptor. Para determinar o emissor, basta verificar qual tem a métrica PAT com o maior valor entre os demais processadores do sistema. Por outro lado, o receptor será o processador com menor valor PAT cujo custo de transmissão seja vantajoso para o sistema. Determinados o emissor e o receptor, ocorre a migração para o *cluster* mais apropriado visando melhorar o balanceamento da carga do sistema.

4.2.6 Algoritmo Dinâmico Flexível

Este algoritmo, proposto por Peschlow, Honecker e Martini (2007), avalia tanto a carga de computação quanto a carga de comunicação para identificar fontes de desequilíbrio na simulação. Além disso, essa política avalia a capacidade de cada

host.

Estimativas de desempenho são calculados por uma instância de particionamento global denominada DynPart. Medições são realizadas nos *hosts* e nos processos da simulação. Os resultados assim obtidos são recolhidos periodicamente por DynPart, que, em seguida, calcula as estimativas de desempenho.

DynPart inicia um de dois procedimentos possíveis: um ciclo de equilíbrio de carga ou um ciclo de refinamento de comunicação. Dependendo do tipo de ciclo, cálculos diferentes são feitos. Depois de terminar o cálculo, DynPart envia as solicitações de movimentos de processos para realizar o balanceamento.

4.3 Considerações Finais

Primeiramente, foram descritas, neste capítulo, as causas de desbalanceamento na simulação utilizando protocolos otimistas. Em seguida, foi descrito o funcionamento de algoritmos de escalonamento para a Simulação Distribuída os quais procuram minimizar as mencionadas causas de desbalanceamento. Estes algoritmos, fazendo uso de informações da simulação, determinam como realizar o escalonamento, ou seja, determinam em quais processadores devem ser executados os processos da simulação.

Destaca-se que estes algoritmos podem ser utilizados em conjunto com os mecanismos de migração de processos apresentados neste trabalho para determinar como escaloná-los no recurso computacional disponível.

5 Biblioteca de Comunicação MPI

Este capítulo faz uma breve revisão sobre a biblioteca de comunicação MPI (*Message Passing Interface*) (MPI-FORUM, 2008), dando ênfase em determinadas características deste padrão de comunicação.

O projeto dos mecanismos de migração de processos na simulação distribuída com uso do *Time Warp* propostos neste trabalho foram implementados com o uso do MPI e, portanto, seus mecanismos são afetados pelos recursos e características desta biblioteca de comunicação.

5.1 Introdução ao MPI

O *Message Passing Interface* (MPI) é um padrão de comunicação de dados em processamento paralelo (MPI-FORUM, 2008). Na biblioteca MPI, uma aplicação paralela é constituída por um ou mais processos que se comunicam e executam de forma paralela e/ou concorrente. O MPI fornece funções para o envio e recebimento de mensagens entre os processos. Em geral, um conjunto fixo de processos é inicialmente criado. Porém, estes processos podem executar diferentes programas.

O objetivo do MPI é prover um abrangente padrão para escrever programas com passagem de mensagens de forma simples, eficiente, portátil e flexível.

5.2 Características Gerais

O padrão MPI é direcionado para desenvolvedores de programas paralelos portáveis em Fortran, C e C++ que podem explorar os recursos computacionais de uma máquina paralela como *clusters*, *grids*, multi-processadores de memória compartilhada e sistemas com memória compartilhada distribuída.

Entre os recursos disponíveis no MPI destacam-se (MPI-FORUM, 2008):

- Comunicação ponto a ponto;
- Operações coletivas;
- Grupos de processos;
- Domínios de comunicação;
- Topologias de processos;
- Ligações entre Fortran e C.

Os recursos não disponíveis no MPI são:

- Operações explícitas para memória compartilhada;
- Operações que requerem maior suporte do sistema operacional como, por exemplo, tratamento de interrupções;
- Ferramentas para implementação de programas;
- Facilidades para depurar programas;
- Suporte explícito para *threads*;
- Suporte para gerenciamento de tarefas;
- Funções para entrada e saída de dados.

Sua interface é adequada para programas MIMD (*multiple instruction multiple data*) e para programas SIMD (*single instruction multiple data*). O MPI ainda é capaz de prover mecanismos para melhorar o desempenho em sistemas computacionais escaláveis.

5.3 Comunicação Ponto a Ponto

O mecanismo de comunicação básico do MPI é a transmissão de dados entre um par de processos através do envio e do recebimento de mensagens. Esse padrão de troca de mensagens é chamado de comunicação ponto a ponto. A maior parte da construção do MPI é baseada neste conceito de comunicação (MPI-FORUM, 2008).

O MPI fornece um conjunto de funções de envio e recebimento de mensagens rotuladas que permitem a comunicação de tipos de dados definidos. A tipagem dos dados transmitidos é necessária para permitir a correta conversão entre diferentes arquiteturas computacionais. O rótulo das mensagens permite seletividade no recebimento das mesmas. No recebimento ainda é possível usar um rótulo curinga o qual permite que todas as mensagens sejam recebidas independentemente dos seus rótulos.

Existem duas modalidades de comunicação implementadas na biblioteca MPI: a comunicação bloqueante e a não-bloqueante. Ambas são descritas nas próximas seções.

5.3.1 Comunicação Bloqueante

Entende-se por comunicação bloqueante quando as operações de envio e recebimento de dados bloqueiam ou suspendem a execução do processo até que ocorra a conclusão da comunicação. Deste modo, o emissor é bloqueado até que a informação seja recebida pelo receptor, e este será bloqueado até que tenha recebido todos os dados (COULOURES; DOLLIMORE; KINDBERG, 2007). Esta modalidade de

comunicação também é denominada como comunicação síncrona.

Para o envio de dados, o MPI implementa quatro funções bloqueantes (MPI-FORUM, 2008). Estas funções são:

- `MPI_Ssend`: esta função faz com que o processo que envia a mensagem informe ao processo receptor que uma mensagem está pronta para envio e esperando por um sinal (enviado pelo receptor) para que ocorra o envio da mensagem. O custo associado a esse tipo de operação ocorre devido à cópia da mensagem do *buffer* do emissor para a rede de comunicação e desta para o *buffer* do receptor e também há um custo devido ao tempo de espera de um dos processos pelo sinal para dar início à transmissão.

- `MPI_Rsend`: nesta função, a mensagem é enviada imediatamente para a rede de comunicação. É necessário que a função de recebimento tenha sido executada anteriormente de modo a permitir que o sinal informado pelo receptor já tenha sido recebido. O custo de cópia da mensagem e da espera de sinal é reduzido, porém pode haver um maior custo no processo receptor dependendo do quanto antes a função de recebimento é executada.

- `MPI_Bsend`: esta função faz com que a mensagem seja copiada do endereço de memória da aplicação para um *buffer* alocado pelo programa, permitindo a continuidade da execução do programa. Quando um sinal é recebido pelo receptor, a mensagem é transmitida. O custo associado a esta função ocorre apenas na espera do recebimento do sinal para se dar o início da transmissão.

- `MPI_Send`: esta função tem um comportamento diferente dependendo do tamanho da mensagem. Esse comportamento pode ser idêntico ao da função `MPI_Ssend` ou da função `MPI_Rsend`. Desta forma, o custo associado a esta função será igual ao custo das referidas funções.

Quanto ao recebimento das mensagens, existe apenas a função `MPI_Recv` implementada no MPI para o recebimento bloqueante de mensagens. Basicamente,

essa função apenas retorna quando toda a mensagem é recebida.

5.3.2 Comunicação Não-Bloqueante

Entende-se por comunicação não-bloqueante quando as operações de envio e recebimento de dados não bloqueiam ou suspendem a execução do programa até que ocorra o término da comunicação. Desta forma, o emissor pode continuar sua execução sem esperar pelo término do envio da mensagem ao receptor. O receptor, por sua vez, também não será bloqueado caso a mensagem ainda não tenha chegado. Esta forma de comunicação também é denominada como comunicação assíncrona (COULOURES; DOLLIMORE; KINDBERG, 2007).

Para a comunicação não bloqueante, são implementadas quatro funções no MPI para o envio de mensagens. Essas funções se diferem daquelas de envio bloqueante apenas por não esperar o sinal do processo receptor. Essas funções recebem o prefixo `MPI_Ixxx` (MPI-FORUM, 2008).

Para o recebimento de mensagens não bloqueantes, o MPI implementa a função denominada `MPI_Irecv`. Ao ser executada, essa função retorna mesmo antes do recebimento da mensagem, alocando apenas um *buffer* para o seu recebimento.

A implementação do recebimento de mensagens de forma não bloqueante ainda pode ser feito mediante o uso da função `MPI_Iprobe`. Esta função realiza um teste não bloqueante verificando a chegada de alguma mensagem. Uma vez verificada a chegada de alguma mensagem, esta pode ser recebida pela função bloqueante `MPI_Recv`.

5.4 Agrupamento de Processos no MPI

O MPI apresenta características que permitem o desenvolvimento de bibliotecas paralelas de forma consistente e portátil (MPI-FORUM, 2008). As principais características que apóiam a criação de bibliotecas paralelas de forma robusta são

as seguintes:

- Espaço de comunicação seguro, garantindo que as bibliotecas utilizem os mecanismos de comunicação conforme suas necessidades, sem entrar em conflito com mensagens estranhas às bibliotecas;
- Escopo do grupo para operações coletivas, permitindo que as bibliotecas possam fazer uso de comunicação coletiva de forma simples;
- Nomeação das bibliotecas para permitir que descrevam a sua comunicação em termos adequados às suas próprias estruturas de dados e algoritmos;
- Capacidade de acrescentar, a um grupo de processos que se comunicam, atributos adicionais definidos pelo usuário. Este mecanismo deve fornecer um significado para o usuário ou o desenvolvedor da biblioteca de forma eficaz para estender a notação de transmissão de mensagens.

Os conceitos correspondentes no MPI que permitem a implementação dessas características são os seguintes (MPI-FORUM, 2008):

- Contexto de Comunicação (*Contexts*);
- Grupos de Processos (*Groups*);
- Topologias Virtuais (*Virtual Topologies*);
- Atributos de *caching* (*Attribute caching*);
- Comunicadores (*Communicators*).

As próximas seções descrevem, suscintamente, estes conceitos.

5.4.1 Contextos

Os contextos (*Contexts*) fornecem a capacidade de manter-se separado, de forma segura, "universos" diferentes de passagem de mensagens no MPI. Um contexto é semelhante a uma *tag* adicional que diferencia mensagens. O uso de contextos de comunicação diferentes em bibliotecas distintas isola a comunicação interna da execução da biblioteca de comunicação externa. Isso permite a invocação da biblioteca sem interferência de outras mensagens, mesmo que haja comunicações pendentes sobre outros comunicadores.

5.4.2 Topologias Virtuais

A topologia virtual (*Virtual Topologies*) do MPI permite realizar um mapeamento de processos MPI em uma "forma" geométrica, para facilitar determinados padrões de comunicação que os processos de um programa paralelo podem realizar. As duas principais topologias virtuais são Cartesianas e Gráficas. Essas topologias MPI são consideradas virtuais porque não têm nenhuma relação com a topologia física da máquina paralela.

A implementação da topologia virtual é realizada pelos grupos (*groups*) e comunicadores (*communicators*) MPI, de tal forma, que, o desenvolvedor da aplicação deve fazer uso de determinadas funções que atuam sobre os grupos e comunicadores, e assim obter a topologia adequada para a aplicação paralela.

5.4.3 Grupos

Um grupo é um conjunto de processos que são identificados através de valores inteiros compreendidos entre 0 e $n-1$ (sendo n o número de processos pertencentes ao grupo). Um grupo é usado dentro de um comunicador para definir o conjunto de processos envolvidos na comunicação e para identificar os processos participantes (dando-lhes nomes únicos dentro desse "universo" de comunicação).

No MPI existe um grupo pré-definido especial: `MPI_GROUP_EMPTY`, que é um grupo sem membros. Também existe o `MPI_GROUP_NULL` que é uma constante predefinida e é o valor usado para o grupo inválido (ponteiro nulo). Ressalta-se que `MPI_GROUP_EMPTY` é um identificador válido para um grupo vazio, mas não deve ser confundido com `MPI_GROUP_NULL`, que por sua vez é um identificador inválido. O primeiro pode ser usado como um argumento para as operações de grupo, este último, por sua vez, é retornado quando um grupo é liberado, não sendo um argumento válido.

5.4.4 Comunicador

Um comunicador (*communicator*) é um objeto com uma série de atributos, juntamente com regras simples que regem sua utilização, criação e destruição. Está associado a um grupo (*group*) que define quais são os processos pertencentes ao comunicador. O comunicador especifica um domínio que pode ser usado para definir a comunicação ponto a ponto ou o conjunto de processos pertencentes a uma comunicação coletiva.

Existem dois tipos de comunicador: *intracommunicator* e *intercommunicator*. O *intracommunicator* contém o processo local no grupo associado. O *intracommunicator* determina o conjunto de processos que podem participar em uma comunicação ou operação coletiva, restringindo o escopo de comunicação. Um *intercommunicator* é um comunicador cujo grupo não contém o processo local, mas está associado a um *intracommunicator* (que contém o processo local). O *intercommunicator* permite a comunicação ponto a ponto entre processos de diferentes grupos.

5.5 Criação Dinâmica de Processos com MPI

A criação dinâmica de processos consiste em criar novos processos a partir de um processo pai. Os novos processos são denominados de processos filhos. Os

sistemas operacionais fornecem chamadas de sistema com a finalidade de criação dinâmica de processos. Nos sistemas operacionais Unix e Linux, uma função com tal finalidade é a função *fork*.

A criação dinâmica de processos tem como característica o estabelecimento de uma hierarquia de processos conforme estes vão sendo criados. Em termos de execução existem duas possibilidades:

1. Processo pai executa concorrentemente ao processo filho;
2. Processo pai espera o término dos processos filhos.

Em termos de espaço de endereçamento há duas possibilidades:

1. O processo filho é uma duplicação do processo pai;
2. O processo filho tem um programa, diferente do pai, carregado nele.

O modelo de criação dinâmica de processos no MPI permite a criação e encerramento de processos depois que uma aplicação MPI foi iniciada. O modelo fornece um mecanismo para estabelecer a comunicação entre os processos recém-criados com processos anteriormente existentes no MPI. Também fornece um mecanismo para estabelecer a comunicação entre dois aplicativos existentes MPI, mesmo quando um não "criou" o outro.

O padrão MPI (versão 2) implementa duas funções para criação dinâmica de processos (MPI-FORUM, 2008). São elas:

1. `MPI_Comm_spawn`: esta função inicia um número de cópias idênticas de processos MPI que executam um programa especificado por um dos parâmetros da função. Um comunicador entre o processo pai e os filhos é retornado na forma de um *intercommunicator*. Os processos filhos (criados em uma mesma chamada dessa função) estão contidos em um mesmo comunicador

(MPI_COMM_WORLD), que é separado do processo pai. Os processos ainda podem ser iniciados em determinado *host* da máquina paralela especificado através de um parâmetro da função. É importante observar que processos criados por chamadas separadas dessa função, não estão em um mesmo *intracommunicator*. Logo, esses processos só podem comunicar entre si utilizando outros mecanismos proporcionados pelo padrão MPI. Esses mecanismos são abordados na seção 5.5.1.

2. MPI_Comm_spawn_multiple: é uma função semelhante à MPI_Comm_spawn exceto que são especificados múltiplos programas e locais para execução de cada um deles. Consequentemente, um dos parâmetros dessa função informa o número de especificações de programas que serão criados. Os outros parâmetros são idênticos aos da função MPI_Comm_spawn, exceto que se tratam de vetores daqueles tipos. Essa função permite que diferentes programas sejam executados em locais diferentes e ainda pertençam a um mesmo *intracommunicator* (MPI_COMM_WORLD).

Considerando a descrição anterior para a criação dinâmica de processos, pode-se dizer que o padrão MPI, em termos de execução, consiste no processo pai executando de forma concorrente ao processo filho. Quanto ao espaço de endereçamento, os processos filhos são carregados com programas diferentes (seu espaço de endereçamento não é uma duplicata do espaço de endereçamento do processo pai).

5.5.1 Comunicação Entre Processos Criados Separadamente

Processos que são criados separadamente, ou seja, em diferentes chamadas das funções MPI_Comm_spawn ou MPI_Comm_spawn_multiple, não compartilham o mesmo *intracommunicator*. Consequentemente, faz-se necessário obter um *intercommunicator* para ser possível a comunicação entre esses processos. A figura 4 ilustra essa situação. Ela mostra que na primeira execução da função MPI_Comm_spawn são criados dois processos que compartilham um mesmo comunicador (MPI_COMM_WORLD).

Na segunda execução dessa função são criados outros três processos que também compartilham um mesmo comunicador (`MPI_COMM_WORLD`). Entretanto os dois processos criados na primeira chamada não podem se comunicar com os três processos criados na segunda chamada pois não compartilham nenhum comunicador. O padrão MPI, para contornar esta situação, provê meios para conectar esses processos.

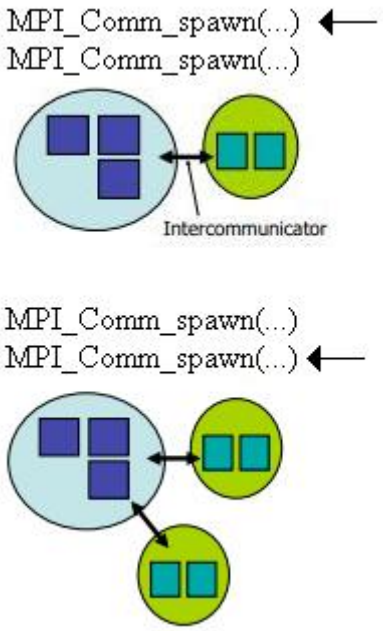

Figura 4: Criação de processos em múltiplos estágios (adaptado de Dongarra et al. (1998))

O MPI oferece funções que permitem obter *intercommunicators* entre processos que não compartilham algum *intracommunicator* (MPI-FORUM, 2008). Essas funções são descritas a seguir:

- `MPI_Open_port`: Estabelece um endereço de rede (endereço MPI) em um processo MPI para aceitar conexões de outros processos MPI. O endereço de rede estabelecido é codificado em uma *string* retornada pela função permitindo que o processo seja capaz de aceitar conexões de outros processos. Essa *string* é fornecida pelo sistema MPI, identificando a porta recém aberta

e permitindo que um processo possa entrar em contato com o processo associado a essa porta. O parâmetro para essa *string* deve ter um tamanho de `MPI_MAX_PORT_NAME`.

- `MPI_Publish_name`: Publica um nome de serviço associado a uma porta. Isso é necessário para que um processo possa recuperar a *string* que identifica uma porta. O nome de serviço utilizado só pode ser publicado uma única vez.
- `MPI_Comm_accept`: Estabelece a comunicação com um processo. Essa função retorna um *intercommunicator* que permite a comunicação com o cliente. O processo que executa essa função fica bloqueado até que solicitações de conexão sejam realizadas.
- `MPI_Lookup_name`: Encontra a *string* que identifica uma porta que está associada a um nome de serviço.
- `MPI_Comm_connect`: Estabelece a comunicação com um processo através de uma porta identificada por uma *string* que é parâmetro dessa função. A chamada `MPI_Comm_connect` só deve ser realizada depois da chamada `MPI_Comm_accept` pelo processo que mantém a porta aberta. O MPI não fornece nenhuma garantia para tentativas de conexão que ocorram concorrente ou paralelamente, ou seja, estas tentativas de conexão podem impedir que uma tentativa de conexão em particular seja satisfeita.

A ordem correta de execução dessas funções no processo que abrirá uma porta e aguardará conexões é:

1. `MPI_Open_port`
2. `MPI_Publish_name`
3. `MPI_Comm_accept`

No processo que se conectará ao processo com uma porta aberta deve-se ter a seguinte sequência:

1. `MPI_Lookup_name`

2. `MPI_Comm_connect`

A figura 5 ilustra o procedimento descrito. Nela tem-se dois processos A e B. O processo A, na etapa 1, abre uma porta e o MPI fornece uma *string* para identificá-la (`Port A`). Na etapa 2, o mesmo processo publica um nome (`ocean`) associado à porta criada para permitir localizá-la. Em seguida, o processo A espera por conexões nessa porta (`MPI_Comm_accept`). O processo B, na etapa 3, determina a *string* da porta aberta no processo A, conhecendo o nome associado a esta porta (`ocean`). Finalmente, na etapa 4, os processos A e B se conectam e um *intercommunicator* é formando em ambos os processos.

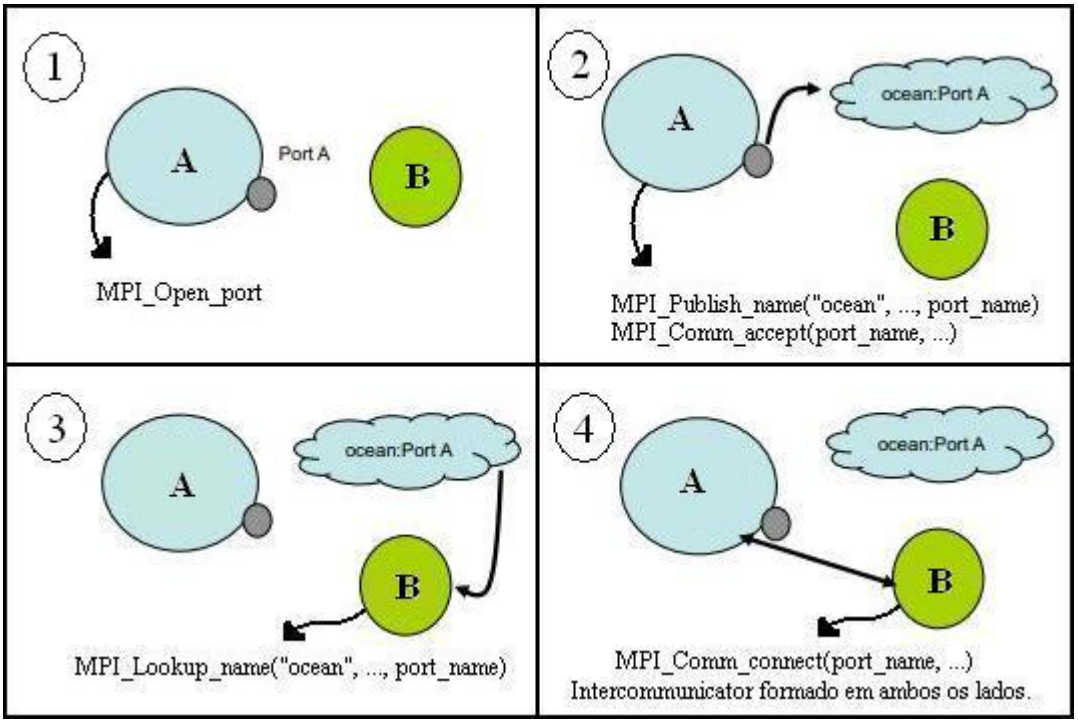

Figura 5: Passos para obter o *intercommunicator* entre dois processos (adaptado de Dongarra et al. (1998))

Caso seja necessário, durante a execução da aplicação MPI, a porta previamente aberta por um processo pode ser fechada através da função `MPI_Close_port`.

Também é possível eliminar o nome de um serviço previamente publicado através da função `MPI_Unpublish_name`. Estas funções podem ser executadas em qualquer ordem, já que, no padrão MPI, ambas as funções são independentes.

5.5.2 Desconectando Processos Associados por Comunicador

Processos que estão associados por *intracommunicator* ou *intercommunicator* não podem se encerrar isoladamente. O MPI implementa um mecanismo de sincronismo através da função `MPI_Finalize`. Esta função deve ser necessariamente executada ao término de cada processo MPI. Caso o processo se encerre sem executar essa função, o MPI interpretará tal situação como um erro e toda a aplicação, ou seja, todos os processos MPI serão finalizados.

O mecanismo de sincronismo implementado no MPI atua sobre todos os processos associados ao *intracommunicator* ou *intercommunicator*. Para liberar os processos desse mecanismo de sincronização e, dessa forma, permitir que os processos se encerrem isoladamente, pode-se fazer uso de duas funções (MPI-FORUM, 2008):

1. `MPI_Comm_disconnect`: espera que todas as comunicações pendentes sobre um comunicador, que é passado como parâmetro, termine e, em seguida, desaloca o objeto comunicador, definindo o identificador para `MPI_COMM_NULL`. Não pode ser executada sobre os comunicadores pré-definidos `MPI_COMM_WORLD` ou `MPI_COMM_SELF`.

2. `MPI_Comm_free`: esta função tem o mesmo comportamento e atuação da função `MPI_Comm_disconnect`, exceto que não aguarda pelo término das comunicações pendentes. Também não pode ser executada sobre os comunicadores pré-definidos `MPI_COMM_WORLD` ou `MPI_COMM_SELF`.

É importante notar que um processo só pode ser encerrado isoladamente se estiver desconectado de todos os comunicadores que foram utilizados para comu-

nicação em algum momento de sua execução.

5.6 Considerações Finais

Este capítulo fez uma revisão da biblioteca de comunicação MPI. Foram apresentados seus mecanismos de comunicação e criação de processos. Esses recursos oferecidos pelo MPI afetaram a forma como foram projetados os mecanismos de migração de processos neste trabalho.

Essa descrição também é importante para compreender como realizar a implementação dos mecanismos de migração que serão apresentados no próximo capítulo.

6 Mecanismos para Migração na Simulação Distribuída

O principal objetivo deste trabalho é propor dois mecanismos para permitir a migração de processos na simulação distribuída, de tal forma, que, seja possível obter medidas de parâmetros em tempo de execução para avaliação do desempenho da simulação, permitindo a aplicação de algoritmos de balanceamento de carga. Esses mecanismos se baseiam no uso do *Time Warp*, entretanto, podem ser adaptados para outros protocolos otimistas como o *Rollback* Solidário.

Além disso, os mecanismos de migração aqui apresentados são baseados em uma implementação em modo usuário, o que permite obter portabilidade sobre diferentes plataformas, favorecendo dessa forma, a obtenção de benefícios como o uso de *clouds*, bem como *grids*, ou ainda qualquer sistema heterogêneo, para realizar a simulação distribuída.

Existem trabalhos, tais como Stellner (1996), Cao, Li e Guo (2005), Liu, Ma e Ou (2009), Rodrigues, Cores e Rodrigues (2011), que apresentam mecanismos para migração de processos baseados no MPI. Ravassi (2009) apresenta uma política de escalonamento de processos com suporte à migração denominada JUMP. Entretanto, as soluções para migração de processos apresentadas nesses trabalhos não obtêm informações da própria aplicação para decidir quando migrar e quais processos migrar, o que é de grande importância para o balanceamento de carga na simulação distribuída, como descrito no capítulo 4.

O trabalho de Glazer e Tropper (1993) utiliza um mecanismo de migração para realizar a migração de processos da simulação distribuída, baseada no protocolo *Time Warp*, capaz de obter informações da simulação (LVT) e do uso do processador para calcular a taxa de avanço dos processos. No entanto, o mecanismo de migração é implementado em modo *kernel* o que diminui a sua portabilidade.

As próximas seções apresentam o funcionamento dos dois mecanismos de migração propostos neste trabalho. Também é feita uma descrição de detalhes importantes da implementação, considerando o uso da biblioteca de comunicação MPI.

6.1 Mecanismo de Migração Coletiva dos Processos

Esta seção descreve como realizar a migração de processos coletivamente. Entende-se por migração coletiva, neste trabalho, como a migração de todos os processos da simulação simultaneamente.

No mecanismo de migração coletiva, um processo mestre é responsável por criar os processos da simulação nos *hosts* adequados do sistema de computação distribuído. O processo mestre segue uma sequência de passos para gerenciar todos os processos da simulação. A sequência de passos seguida pelo processo mestre é:

1. **Aplicar o algoritmo de balanceamento:** quando o processo mestre é inicializado, a simulação ainda não começou, logo um mapeamento estático dos processos da simulação deve ser realizado por meio do algoritmo de escalonamento;

2. **Iniciar ou reiniciar os processos:** de acordo com o algoritmo de balanceamento, caso haja necessidade, o processo mestre criará os processos nos *hosts* adequados. Nota-se aqui que todos os processos da simulação são criados simultaneamente;

3. **Iniciar um temporizador:** durante este temporizador, o processo mestre permanece ocioso de forma que a simulação seja realizada sem interferência durante este período de tempo.

4. **Iniciar a obtenção das métricas de desempenho:** neste passo, o processo mestre informa a todos os processos da simulação que esses devem coletar informações conforme a necessidade do algoritmo de balanceamento empregado;

5. **Receber as informações sobre a simulação:** após informar aos processos para obterem os dados sobre a simulação, o mestre deve aguardar até receber os dados da simulação por meio de mensagens. Com as informações, o mestre pode ainda decidir se alguma condição de término foi satisfeita. Nesse caso, todos os processos da simulação recebem uma mensagem informando que devem ser encerrados;

6. **Aplicar o algoritmo de balanceamento:** com as informações recebidas, o mestre aplica o algoritmo de balanceamento. O algoritmo deve informar da necessidade de migração. Em caso afirmativo, informa também onde os processos devem ser reiniciados;

7. **Anunciar migração:** se, de acordo com o algoritmo de balanceamento, houver a necessidade de migrar os processos, todos estes devem ser informados (por meio de mensagens) que sofrerão migração;

8. **Aguardar o encerramento dos processos:** se houver necessidade de migração, os processos atuais da simulação deverão ser encerrados e reiniciados. Porém, para manter a consistência das variáveis e mensagens que ainda possam estar sendo trocadas entre os processos, o processo mestre deve aguardar o término dos processos da simulação. Imediatamente antes de encerrarem, os processos da simulação informam ao mestre tal situação;

9. **Retornar ao passo 2.**

A figura 6 apresenta o fluxograma dos passos seguidos pelo processo mestre.

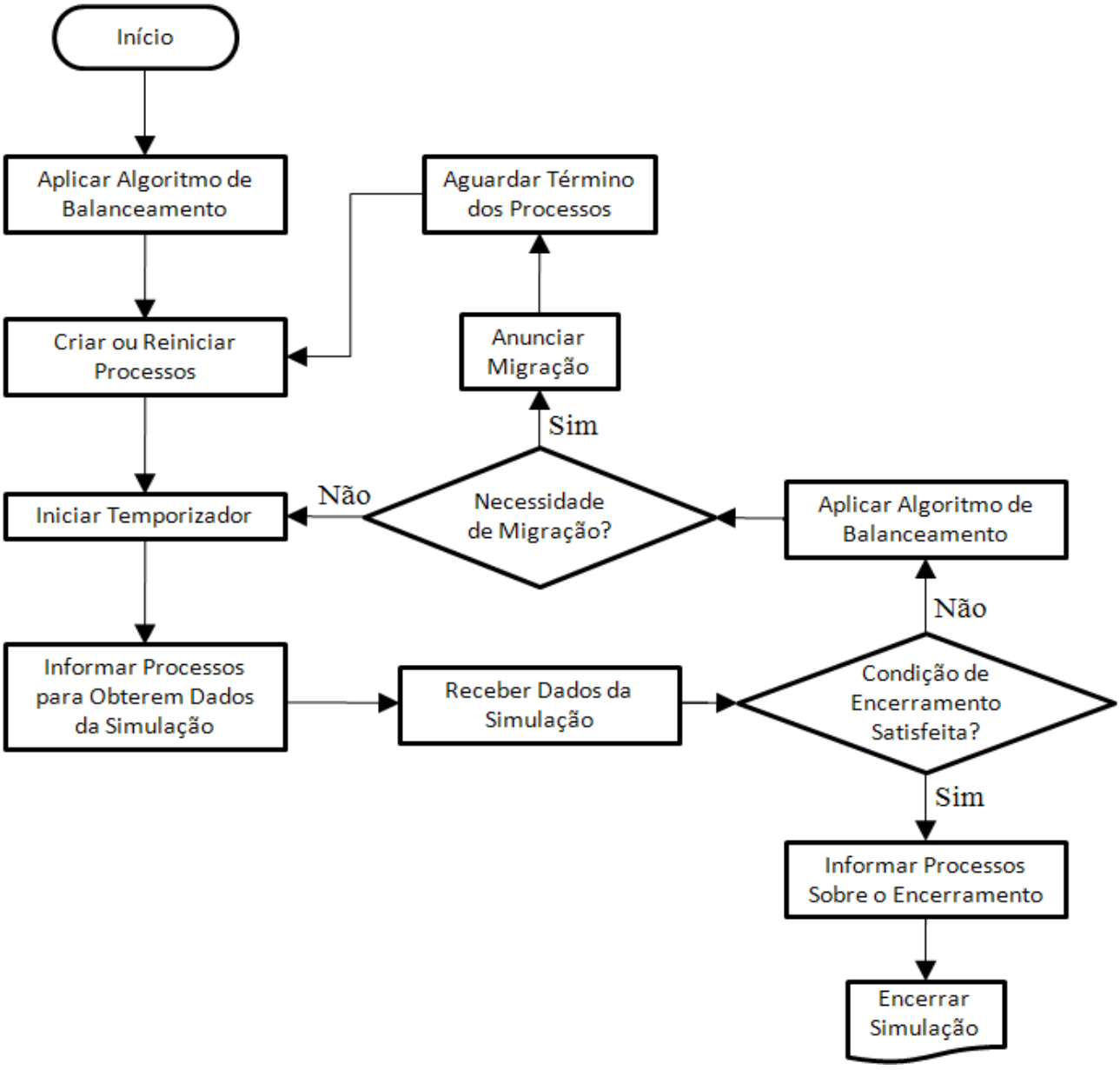

Figura 6: Fluxograma do processo mestre para o mecanismo de migração coletiva

Os processos da simulação devem passar por algumas modificações para obter um correto sincronismo com o processo mestre, bem como, passar por migrações de forma consistente. A maior modificação nos processos consiste no uso de uma

thread de sincronismo que é responsável pela troca de mensagens com o processo mestre, pela obtenção das métricas necessárias ao algoritmo de balanceamento e pela interação com a *thread* principal (que executa a simulação baseado no *Time Warp*).

Assim, com a modificação sugerida, algumas etapas extras são adicionadas à *thread* principal dos processos da simulação. Em conjunto, essas etapas são:

1. **Inicializar a *thread* de sincronismo:** antes de entrar no *loop* principal da simulação, a *thread* de sincronismo deve ser iniciada;

2. **Verificar se é a primeira execução do processo:** caso não seja a primeira execução, o processo já passou por migrações e deve ter suas variáveis recuperadas, bem como recuperadas as mensagens recebidas durante o estado de pré-migração. As variáveis são recuperadas de arquivos previamente criados para armazenar os seus valores. Da mesma forma, as mensagens recebidas durante o estado de pré-migração são recuperadas de arquivos. Esses arquivos são salvos em um servidor de arquivos, dessa forma, independente do *host* em que o processo seja reiniciado, este terá acesso ao arquivo;

3. **Verificar condição de término:** uma variável fica responsável por informar se um critério de parada da simulação foi satisfeito e deve ser verificada no início de cada *loop* principal da simulação. A *thread* de sincronismo é responsável por alterar essa variável quando o processo mestre informa a respeito do término da simulação;

4. **Verificar se há necessidade de migração:** outra variável é responsável por informar se o processo passará por migração. A *thread* de sincronismo também é responsável por alterar essa variável quando o processo mestre informa da necessidade de migração. Caso haja necessidade de migração, o processo entra no estado de pré-migração. O estado de pré-migração é discutido na seção 6.1.1. Assim que o processo sair do estado de pré-migração,

este estará apto para encerrar e, imediatamente antes de seu término, deverá notificar ao mestre que está pronto para encerrar;

5. **Realizar a iteração:** Uma vez que o processo verifica que a condição de término não foi satisfeita e que não vai passar por uma migração, este realiza normalmente a iteração da simulação;

6. **Voltar ao passo 3.**

A figura 7 apresenta o fluxograma do funcionamento da *thread* principal dos processos da simulação.

Para permitir que ocorra a comunicação entre o processo mestre e os processos da simulação, de forma que a execução da simulação não fique prejudicada, faz-se uso do que se denomina neste trabalho de *thread* de sincronismo. A função dessa *thread* é receber as solicitações do processo mestre e aplicá-las sobre o processo da simulação. Os passos que a *thread* de sincronismo executa são:

1. **Verificar se há mensagens para o processo:** se não houver mensagens, a *thread* deve dormir por um curto período de tempo;

2. **Identificar instrução recebida:** a *thread* deve identificar o tipo de mensagem recebida do processo mestre. Se é uma mensagem informando a ocorrência de migração, a *thread* deve modificar a variável que identifica essa situação (a *thread* principal deve ter acesso à variável) e em seguida será encerrada;

3. **Obter dados da simulação:** se a mensagem recebida não foi uma mensagem de migração nem informando condição de término, então a *thread* coleta dados da simulação (dados necessários para o algoritmo de balanceamento) e, em seguida, envia esses dados ao processo mestre;

4. **Voltar ao passo 1.**

A figura 8 apresenta o fluxograma da *thread* de sincronismo.

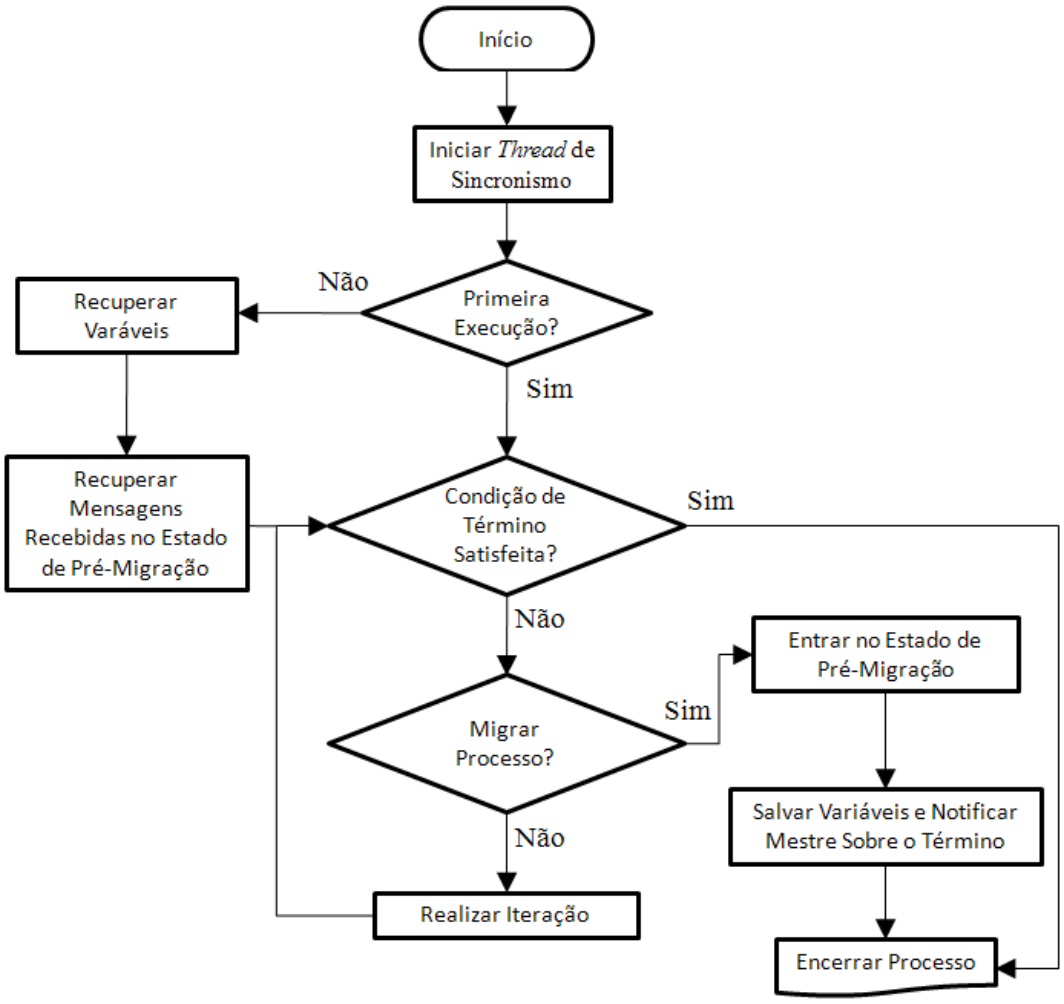

Figura 7: Fluxograma da *thread* principal para o mecanismo de migração coletiva

6.1.1 Estado de Pré-Migração

Como foi mencionado, para que os processos sejam informados que passarão por uma migração, eles recebem uma mensagem do processo mestre indicando tal situação. No entanto, os processos não podem encerrar imediatamente ao receberem a informação de migração, pois isto levaria a perda de mensagens da própria simulação (ex. eventos e anti-mensagens) em tráfego na rede ou que fossem enviadas pelos próprios processos da simulação durante o encerramento de outros.

Para resolver essa situação, introduz-se aqui, o conceito de estado de pré-

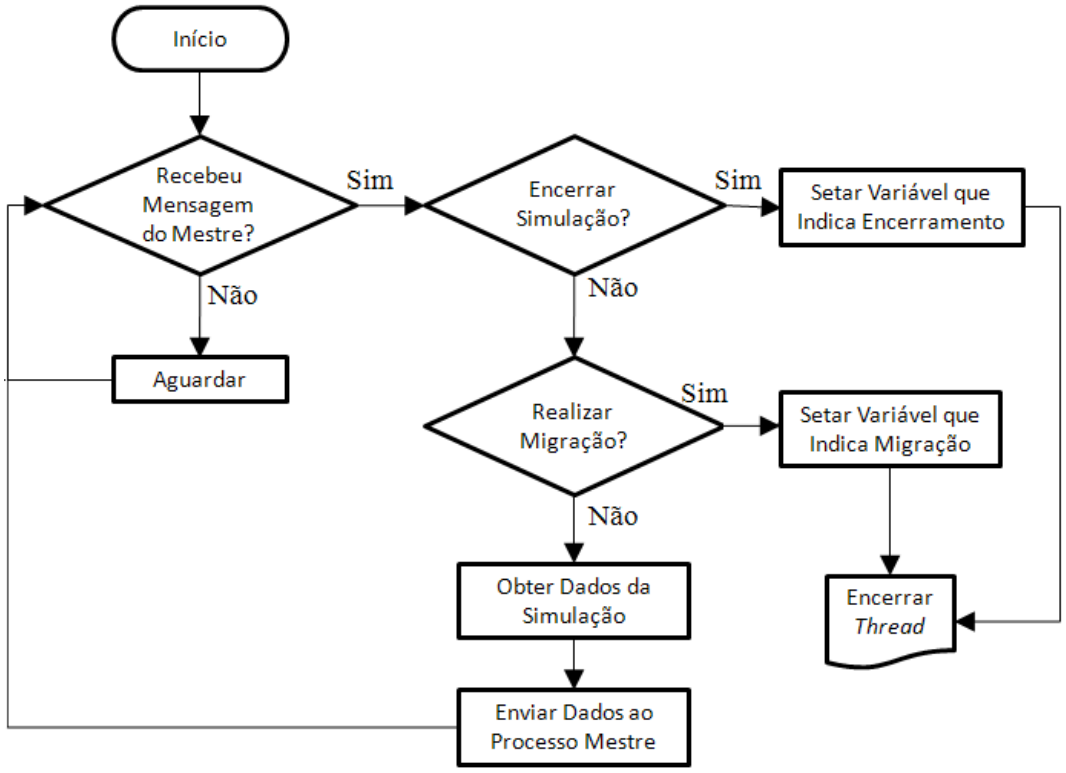

Figura 8: Fluxograma da *thread* de sincronismo para o mecanismo de migração coletiva

migração. Basicamente, um processo nesse estado, apenas recebe mensagens e não executa mais a simulação.

Supondo que o canal de comunicação garanta a ordenação das mensagens (a primeira a ser enviada é a primeira a ser recebida - FIFO), não haverá perdas de mensagens quando um processo encerrar, após sua saída do estado de pré-migração, se todos os outros processos da simulação conhecerem sua situação e, assim, não enviarem mensagens para aquele processo.

Para que um processo A possa ser informado de que outro processo B se encontra no estado de pré-migração, o processo B notifica o processo A (por meio de mensagem) assim que entrar no estado de pré-migração.

No mecanismo de migração coletiva, um processo (*thread* principal) ao receber

a notificação de migração, segue as seguintes etapas:

1. Envia uma mensagem para todos os processos informando que entrou no estado de pré-migração;

2. Recebe todas as mensagens: se as mensagens recebidas forem mensagens da própria simulação, como o processo se encontra no estado de pré-migração, este não trata essas mensagens, apenas as armazena em um arquivo (em um servidor de arquivos) para posterior recuperação. Caso a mensagem seja uma notificação de que algum processo entrou no estado de pré-migração, o processo receptor deve atualizar um vetor que indica quais processos entraram no estado de pré-migração;

3. Sai do estado de pré-migração: quando o processo verificar que todos os outros processos da simulação já estão no estado de pré-migração, este deve sair do estado de pré-migração.

Estes passos são executados exclusivamente pela *thread* principal. A figura 9 apresenta o fluxograma associado ao estado de pré-migração.

Caso o canal de comunicação não possa garantir a ordenação de mensagens (não FIFO), o procedimento descrito não garante que todas as mensagens sejam recebidas pelo destinatário antes que este se encerre. A figura 10 apresenta uma situação na qual ocorre perda de mensagem devido ao encerramento do processo no momento incorreto. Inicialmente, apenas o processo 2 se encontra em estado de pré-migração e está aguardando o processo 1 confirmar que já conhece sua situação para que possa encerrar. Assim que o processo 1 recebe a notificação de que o processo 2 se encontra no estado de pré-migração, o processo 1 envia uma mensagem ao processo 2 informando que já conhece essa situação. Porém, uma mensagem da própria simulação foi previamente enviada pelo processo 1, e, devido a não ordenação na entrega de mensagens, a mensagem da simulação chega depois da mensagem de notificação de pré-migração ocasionando assim a sua perda, já que o processo 2 se encerra devido à notificação.

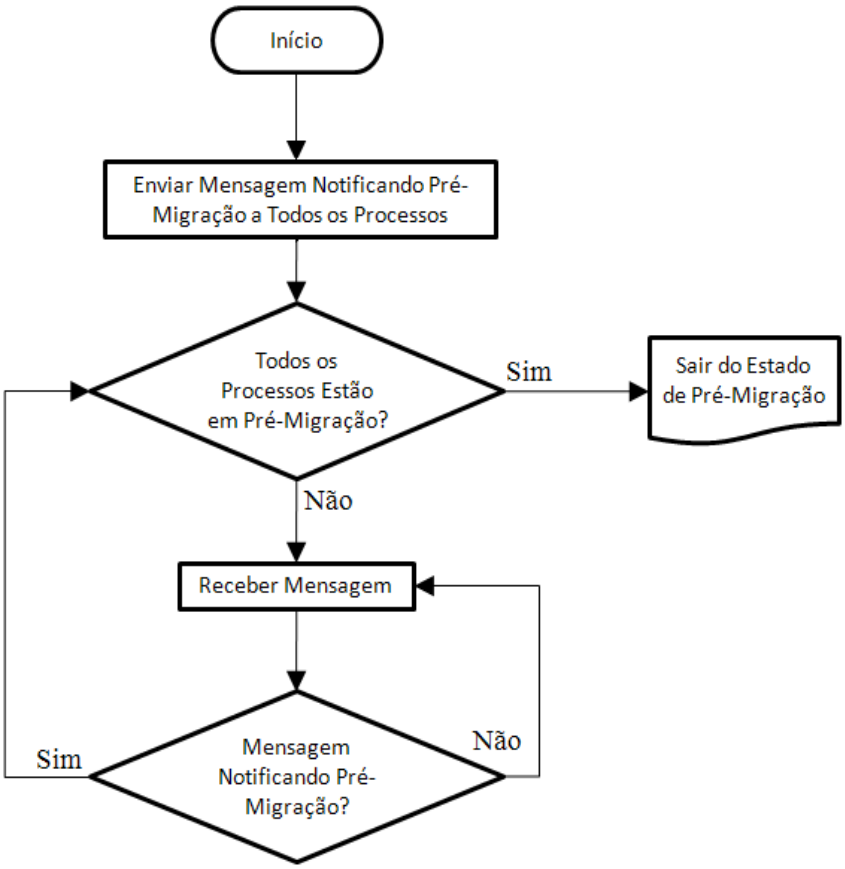

Figura 9: Fluxograma do estado de pré-migração coletiva para canal FIFO

Para resolver esse problema propõe-se aqui fazer uso de um contador de mensagens enviadas e recebidas. Toda vez que uma mensagem for enviada a um dado processo, o contador associado a este processo (ex. um elemento de vetor) é incrementado. Da mesma forma, quando um processo recebe uma mensagem, um contador associado ao processo emissor é incrementado. Quando os processos passarem por migração, os contadores podem ou não ser reiniciados, já que, o que importa é a diferença, ou seja, o valor final menos o valor inicial dos contadores em cada migração.

No caso do mecanismo de migração coletiva, quando um processo enviar a mensagem notificando sua entrada no estado de pré-migração, este anexa o valor do contador de mensagens enviadas referente ao processo que receberá a notificação.

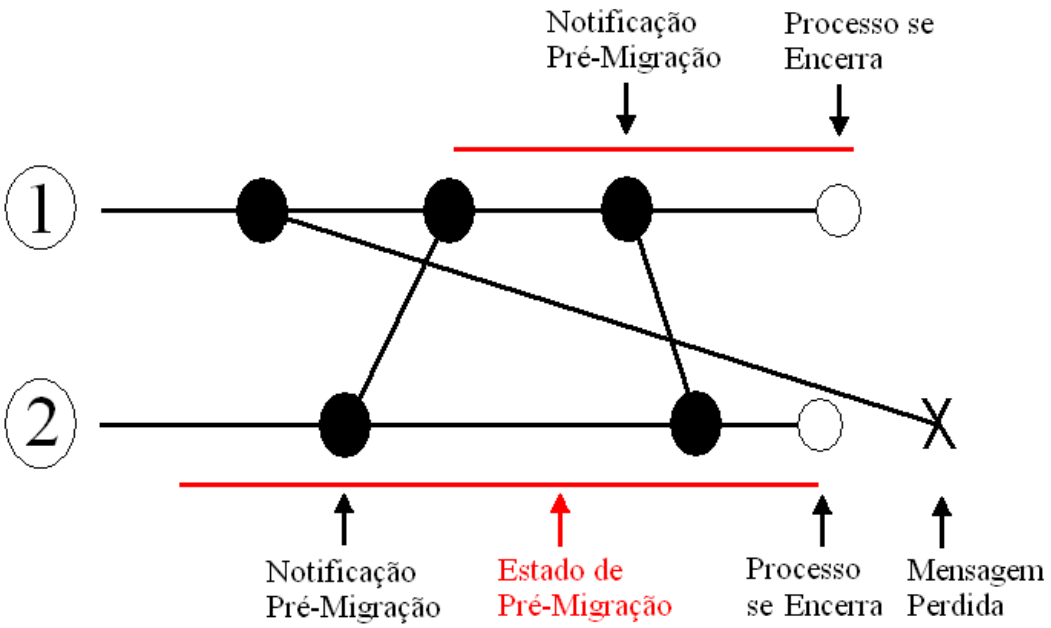

Figura 10: Mensagem transiente associado ao estado de pré-migração

O processo receptor, antes de sair do estado de pré-migração, verifica se o valor do contador de mensagens recebidas da simulação é igual ao valor recebido como anexo na notificação de entrada no estado de pré-migração para cada processo. Se for igual, para todos os processos, o processo sai do estado de pré-migração. Caso contrário, o processo deve aguardar até que todas as mensagens cheguem.

Considerando que o canal não garante a entrega ordenada de mensagens, o fluxograma de tratamento de estado de pré-migração é apresentado na figura 11.

Salienta-se que o procedimento descrito para resolver a questão da não ordenação na entrega de mensagens não se aplica ao padrão de comunicação MPI, já que este garante a ordenação das mensagens entre pares de processos.

6.1.2 Implementação com MPI

Esta seção descreve alguns pontos importantes para realizar a implementação do mecanismo de migração coletiva levando em consideração o uso da biblioteca

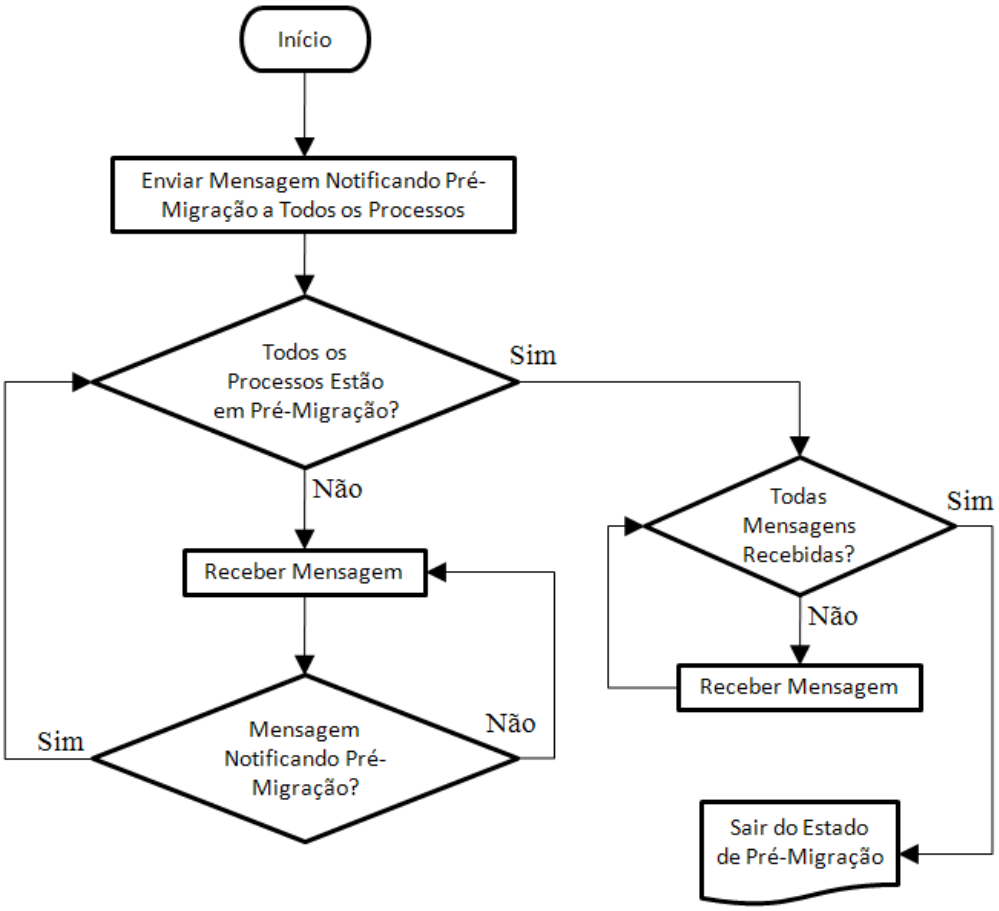

Figura 11: Fluxograma do estado de pré-migração coletiva para canal não FIFO

de comunicação MPI. São eles:

1. A criação de processos pelo processo mestre deve ser realizada mediante o uso da função `MPI_Comm_spawn_multiple`, onde um dos parâmetros identifica em quais *hosts* devem ser executados os processos;

2. Pode-se fazer uso da função `MPI_Iprobe` (em conjunto com a função `MPI_Recv`) ou `MPI_Irecv` associadas a função `sleep` (normalmente disponível em bibliotecas básicas) para aguardar mensagens vindas do processo mestre na *thread* de sincronização. A função `MPI_Recv` pode ser utilizada isoladamente, no entanto, esta é bloqueante e acarreta uso de CPU levando à perda de desempenho;

3. Como os processos são criados simultaneamente, estes compartilham o mesmo *intracommunicator* (`MPI_COMM_WORLD`). Assim, pode-se fazer uso da função `MPI_Comm_rank` para identificar os processos na simulação.

6.2 Mecanismo de Migração Individual dos Processos

Por migração individual entende-se, neste trabalho, como sendo a migração de um ou mais processos entre todos os processos da simulação. Consequentemente, existem diferenças entre o mecanismo para a migração individual dos processos em relação ao mecanismo para migração coletiva dos processos.

A mais importante modificação é a necessidade de se fazer uso de uma lista de mensagens pendentes. Esta lista é necessária, pois, já que esse mecanismo de migração tem a capacidade de manter processos que não migram simulando, estes processos podem criar eventos ou anti-mensagens para processos que estão passando por migração. Assim, como não é possível enviar mensagens desses eventos ou anti-mensagens, devido a migração dos receptores, essas mensagens são armazenadas na lista de mensagens pendentes para serem posteriormente enviadas.

Igualmente ao mecanismo de migração coletiva, um processo mestre se encarrega de controlar os processos da simulação. Algumas modificações são necessárias para seu funcionamento, cujas etapas são:

1. **Aplicar o algoritmo de balanceamento:** essa etapa consiste basicamente em um mapeamento estático dos processos da simulação, já que a simulação ainda não começou de fato;

2. **Iniciar ou reiniciar os processos:** criar os processos nos nós adequados de acordo com o algoritmo de balanceamento. Caso já tenha ocorrido migração, o processo mestre deve aguardar até que os processos da simulação que sofreram migração anunciem seus identificadores. Em seguida, o mestre informa aos processos da simulação que o identificador dos processos que sofreram

migração já está disponível;

3. **Iniciar um temporizador:** o processo mestre permanece ocioso por um determinado período de tempo no qual a simulação não é afetada;

4. **Iniciar coleta de dados:** neste passo, os processos da simulação são instruídos a obter os dados da simulação;

5. **Receber dados e aplicar o algoritmo de balanceamento:** os dados são então recebidos para que o mestre possa aplicar o algoritmo de balanceamento. O mestre ainda é capaz de determinar se algum critério de parada da simulação foi satisfeito e assim encerrar todos os processos;

6. **Anunciar migração:** caso um ou mais processos necessitem passar por migração, serão informados pelo processo mestre;

7. **Aguardar encerramento dos processos:** o mestre deve esperar que todos os processos que passarão por migração se encerrem. Imediatamente antes de encerrarem, os processos enviam ao mestre uma mensagem informando tal situação;

8. **Voltar ao passo 2.**

A figura 12 mostra o fluxograma associado ao processo mestre para o mecanismo de migração individual.

A *thread* principal neste mecanismo tem um comportamento quase idêntico ao mecanismo de migração coletiva, entretanto, existem algumas diferenças no funcionamento dessa *thread*, que são:

1. Necessidade de se verificar, a cada mensagem enviada, se é conhecido o identificador do processo destinatário. Se o identificador não for conhecido, a mensagem que seria enviada é armazenada em uma lista de mensagens pendentes. As mensagens serão enviadas posteriormente quando for obtido o identificador do processo destinatário.

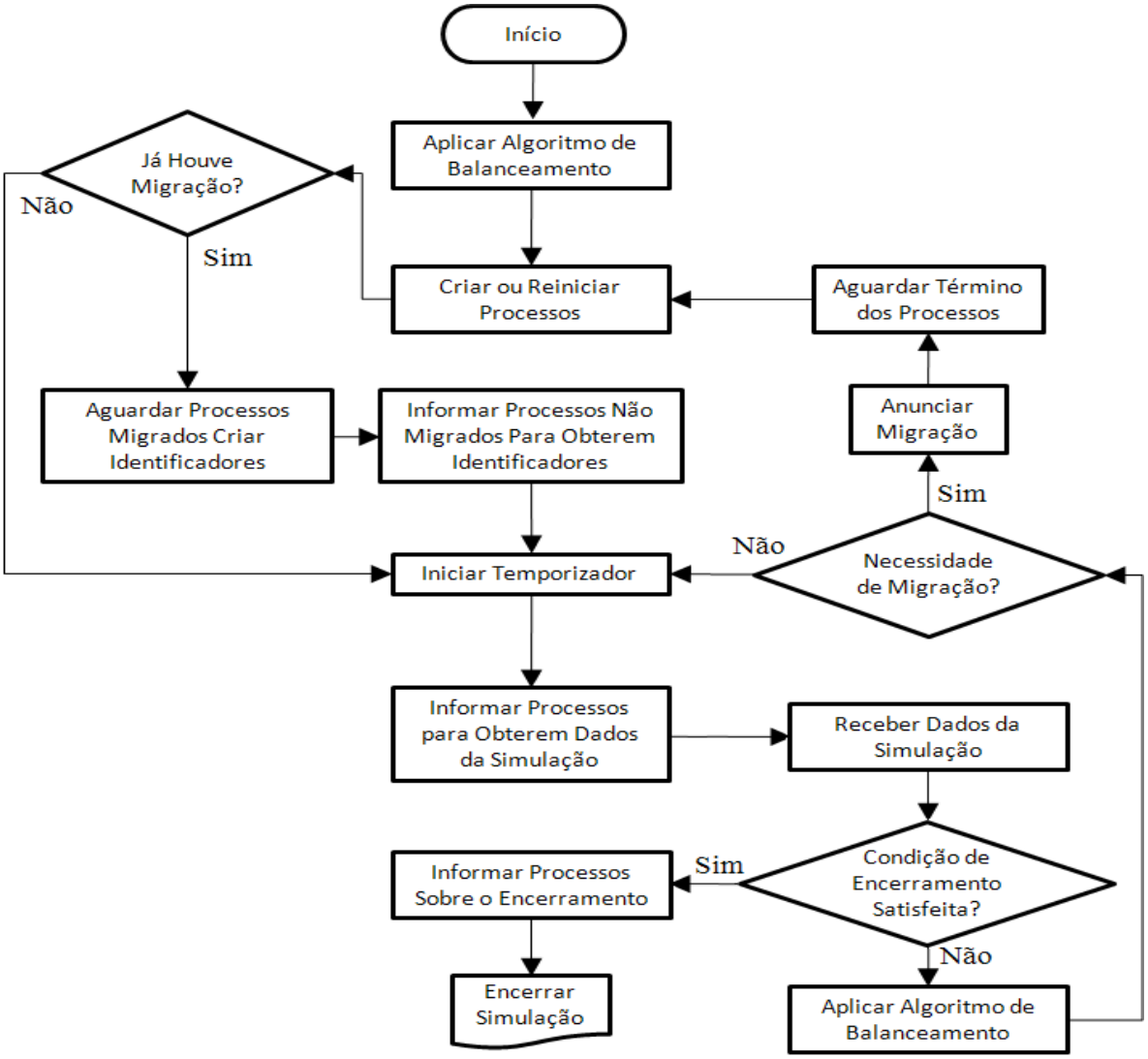

Figura 12: Fluxograma do processo mestre para o mecanismo de migração individual

2. Toda vez que o processo for reiniciado, ele deverá criar um identificador de comunicação e, em seguida, obter os identificadores dos outros processos da simulação. Esse procedimento envolve a criação de portas e o procedimento de espera por conexões nos processos. Um *deadlock* pode ocorrer se todos os processos se encontrarem simultaneamente na condição de espera por conexões. A seção 6.2.2 propõe uma solução para evitar esse *deadlock*.

3. Antes de iniciar a iteração do *Time Warp*, deve ser verificada uma condição (variável) que indica a possibilidade de obter identificadores dos processos que foram reinicializados pelo processo mestre (processos que migraram). Em caso afirmativo, devem ser obtidos os identificadores desses processos e, em seguida, enviadas as mensagens pendentes.

4. O identificador de outro processo deverá ser invalidado se este enviar uma mensagem de notificação de entrada no estado de pré-migração. Esta mensagem é recebida na iteração principal do *Time Warp*, e, assim que for recebida, uma mensagem reconhecendo a notificação de pré-migração é enviada ao processo emissor.

A figura 13 apresenta o fluxograma associado a *thread* principal no mecanismo de migração individual.

A *thread* de sincronismo apresenta uma etapa além daquela descrita no mecanismo de migração coletiva. As mensagens recebidas pelo processo mestre podem ser mensagens de notificação de identificador, além das mensagens de migração e de obtenção de dados. As mensagens de notificação de identificador indicam que os processos que passaram por migração já disponibilizaram seus identificadores. Se esta mensagem for recebida pela *thread* de sincronismo, esta *thread* indica essa situação à *thread* principal (por meio de variável compartilhada entre as *threads*) que, por sua vez, recuperará os identificadores e enviará as mensagens pendentes.

O fluxograma do funcionamento da *thread* de sincronismo é apresentado na figura 14.

6.2.1 Estado de Pré-Migração

O mecanismo de migração individual também apresenta um estado de pré-migração entre os processos e este é semelhante ao estado de pré-migração do mecanismo de migração coletiva.

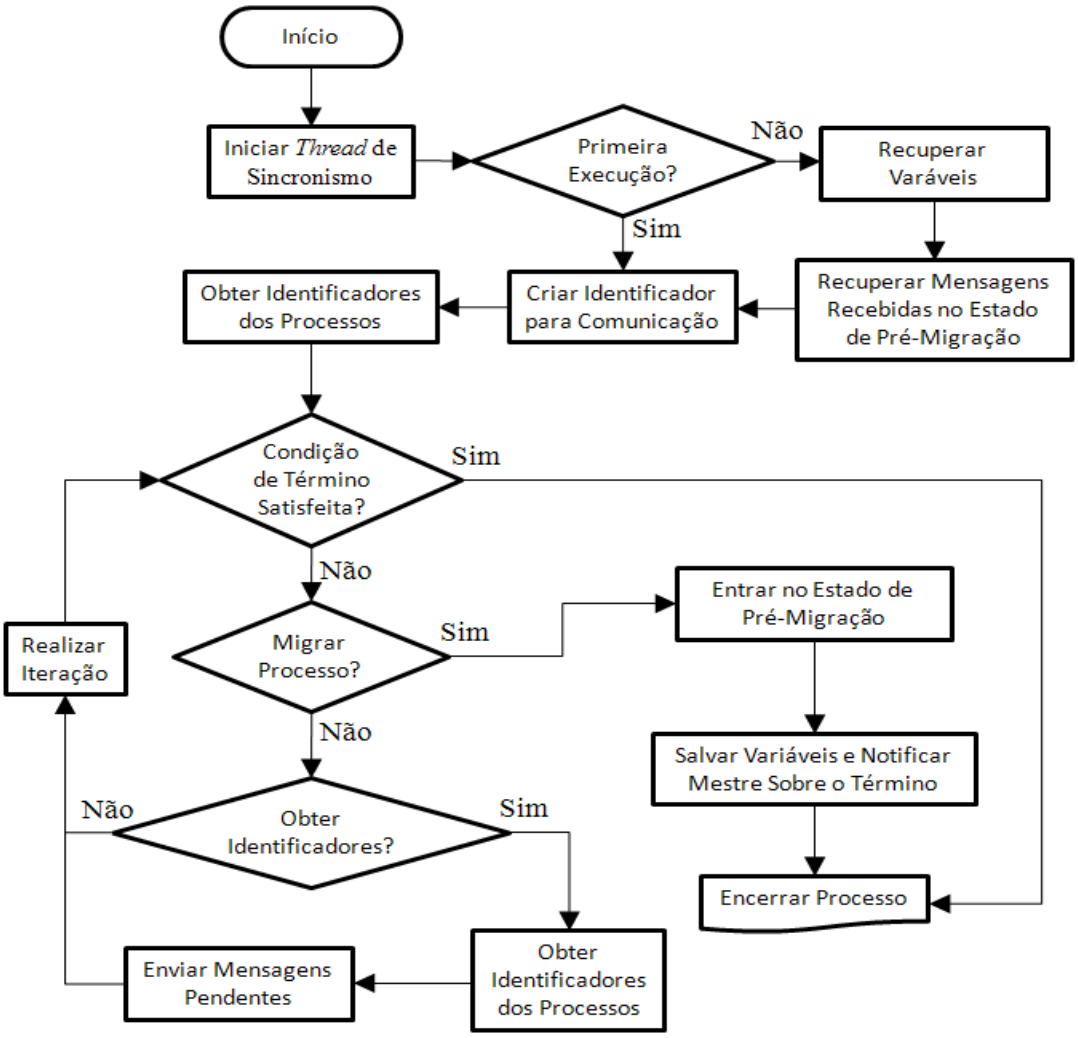

Figura 13: Fluxograma da *thread* principal para o mecanismo de migração individual

Entre as diferenças citam-se:

- Quando um processo que não passará por migração receber uma notificação de um processo informando que entrou no estado de pré-migração, o processo receptor não entra em estado de pré-migração, ao contrário do que ocorre no mecanismo de migração coletiva;
- Apenas os processos que sofrerão migração recebem a mensagem do mestre

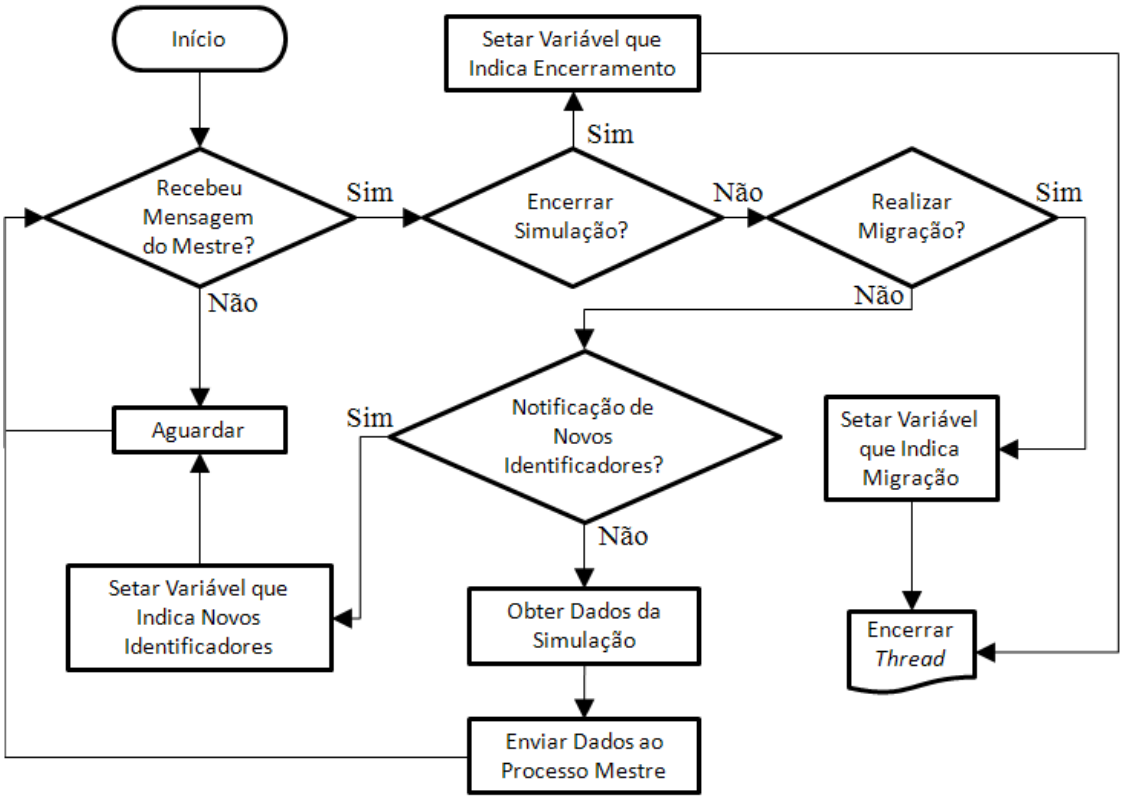

Figura 14: Fluxograma da *thread* de sincronismo para o mecanismo de migração individual

informando tal situação e entram no estado de pré-migração, ao contrário do mecanismo de migração coletiva, em que todos os processos entram no estado de pré-migração, já que todos são migrados;

- Um processo que recebe uma notificação de pré-migração de outro processo, continua sua simulação normalmente. Entretanto, mensagens destinadas a este processo não são enviadas. Estas mensagens são armazenadas na lista de mensagens pendentes.

A figura 15 descreve o comportamento do estado de pré-migração individual para o mecanismo de migração individual.

Da mesma forma como no estado de pré-migração para o mecanismo de mi-

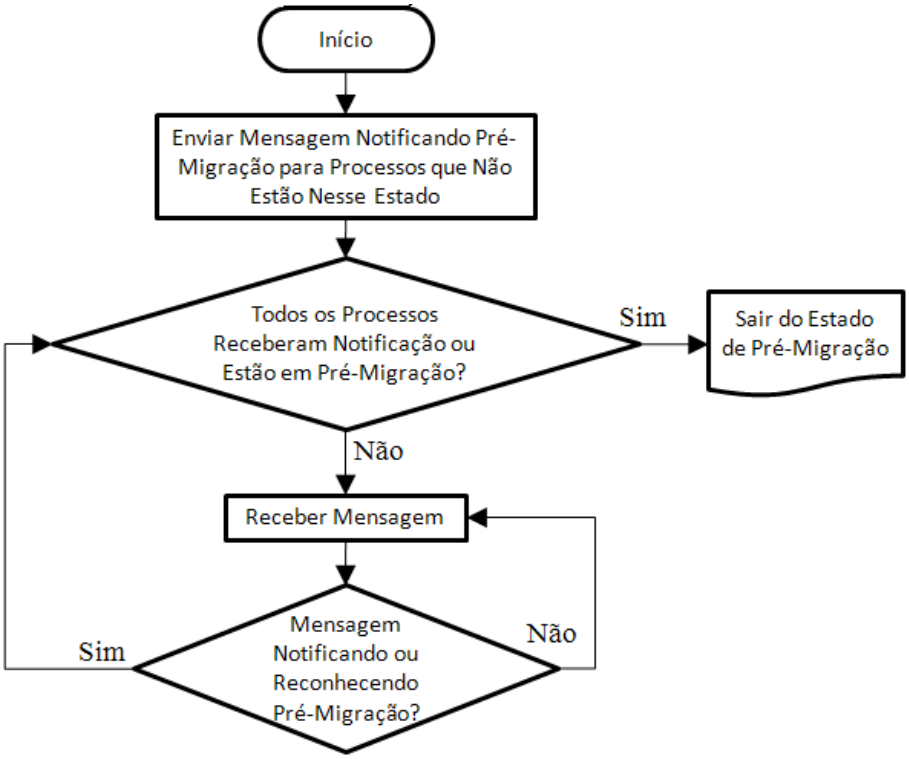

Figura 15: Fluxograma do estado de pré-migração individual para canal FIFO

gração coletiva, se o canal de comunicação não garantir a entrega ordenada das mensagens (canal FIFO), podem ocorrer perdas de mensagens devido a mensagens transientes na rede de comunicação.

A solução proposta na seção 6.1.1 também pode ser utilizada no mecanismo de migração individual para resolver o problema da mensagem transiente.

A figura 16 mostra o novo fluxograma quando considerado um canal de comunicação que não apresenta ordenação das mensagens (não FIFO).

6.2.2 Implementação com MPI

Alguns pontos importantes sobre a implementação do mecanismo de migração individual são tratados nesta seção levando em consideração o uso da biblioteca de comunicação MPI. São eles:

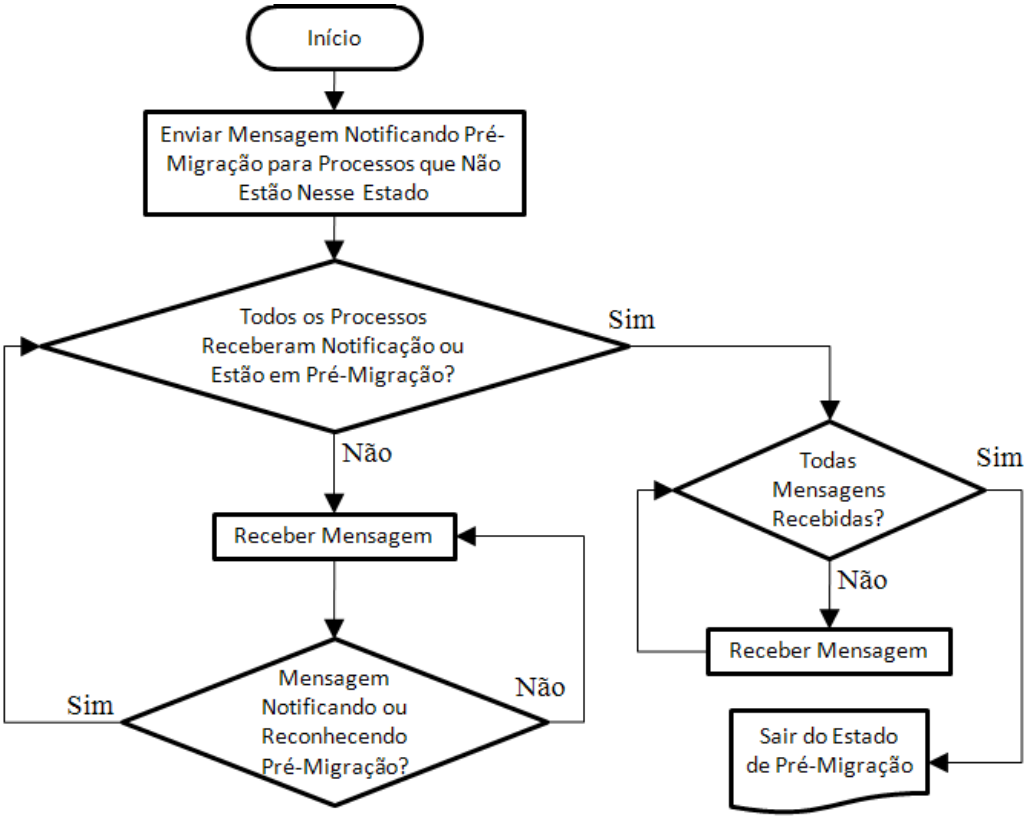

Figura 16: Fluxograma do estado de pré-migração individual para canal não FIFO

1. A criação dinâmica de processos pelo processo mestre deve ser realizada usando a função `MPI_Comm_spawn` em um número de vezes igual ao número de processos criados. Isto é necessário para criar cada processo migrado nos *hosts* adequados. Consequentemente, a comunicação com cada processo será feita por cada um dos *intercommunicators* criados em cada chamada.

2. Ao contrário do mecanismo de migração coletiva, a forma de identificar o processo receptor de uma mensagem não faz uso de um inteiro (*rank*), mas sim de um *intercommunicator*. O *rank* nesse caso é sempre zero.

3. Não é possível obter-se o identificador dos processos mediante o uso da função `MPI_Comm_rank`, pois os processos são criados individualmente de forma que não compartilham o mesmo *intracommunicator*. O identificador pode ser

obtido através do processo mestre, por meio de parâmetros de linha de comando que são passados durante a criação dos processos.

4. A criação de um identificador para a comunicação deve ser feita com o uso das funções `MPI_Open_port` e `MPI_Publish_name`. Um dos parâmetros desta última função deve identificar os processos na simulação.

5. A obtenção dos identificadores dos processos deve ser feita mediante uso das funções `MPI_Lookup_name` e `MPI_Comm_connect`.

6. Uma possibilidade para evitar um *deadlock* nas conexões entre os processos seria criar-se uma ordenação de conexões entre os processos. Por exemplo, o processo com maior identificador se conecta aos demais. Em seguida, o processo com segundo maior identificador se conecta aos $n - 1$ processos restantes, e assim sucessivamente. Neste caso, considerando o uso do MPI, o número total de conexões será, no máximo, $n(n-1)/2$, onde n é o número de processos da simulação. Nota-se que basta uma conexão entre dois processos, pois um *intercommunicator* é retornado para ambos.

7. Não é possível criar as conexões entre os processos de forma paralela (por meio de *thread*), ou seja, um processo receber conexões (`MPI_Comm_accept`) e se conectar simultaneamente (`MPI_Comm_connect`), ou ainda um processo receber conexões de outros processos simultaneamente. O padrão MPI não garante a correta conexão entre processos se estas ocorrem de forma paralela.

8. Quando um processo sofrer migração, este deve eliminar o identificador de comunicação gerado e fechar a porta MPI aberta, através das funções `MPI_Unpublish_name` e `MPI_Close_port`, respectivamente. Isto é necessário para que o processo crie novamente seu identificador após sua reinicialização.

6.3 Comparação Entre os Mecanismos de Migração

Esta seção faz uma comparação teórica entre os mecanismos projetados com relação ao desempenho, implementação e aplicações. Esta comparação é baseada na maior distinção entre os mesmos: a migração de todos os processos simultaneamente ou não.

6.3.1 Desempenho

A migração de um processo não pode ocorrer imediatamente, logo o tempo que os processos estão migrando representa uma perda. O mecanismo de migração coletiva apresenta a maior perda nesse aspecto devido a ocorrência da migração de todos os processos simultaneamente.

Quanto à complexidade extra para o funcionamento, o mecanismo de migração individual apresenta maior custo. Dois aspectos estão associados à essa maior complexidade:

1. A necessidade de verificação, para cada mensagem a ser enviada, se existe o identificador do processo destinatário;
2. O uso de diferentes *intercommunicators* para comunicação com cada processo da simulação, o que leva a necessidade de verificação da chegada de mensagens em cada um deles.

Também deve-se levar em conta a necessidade de criação de portas e conexões entre os processos, o que ocasiona um aumento no tempo de processamento no mecanismo de migração individual.

6.3.2 Implementação

A implementação do mecanismo de migração coletiva é mais simples que a implementação do mecanismo de migração individual. A principal razão para tal,

é a necessidade de se lidar com um *intercommunicator* para cada processo da simulação.

Além disso, há a necessidade de uma lista extra para armazenar mensagens que não puderam ser enviadas em um determinado momento (devido à entrada no estado de pré-migração de alguns processos).

6.3.3 Aplicações

Em função do que foi discutido na seção 6.3.1, pode-se afirmar que o mecanismo de migração coletiva é mais adequado para um modelo de simulação cujos padrões de comunicação e velocidade de avanço dos relógios lógicos são modificados em todos os processos ao longo da simulação, acarretando um desequilíbrio de carga em todo o modelo e levando a necessidade de migração de todos, ou quase todos os processos.

Já o mecanismo de migração individual é mais adequado quando apenas poucos processos apresentam modificações em seus padrões de comunicação e velocidade de avanço de seus relógios lógicos. Assim, apenas alguns processos necessitam ser migrados para realizar o balanceamento de carga.

6.4 Mecanismo de Migração para o Protocolo Rollback Solidário

Esta seção apresenta uma discussão para o desenvolvimento de mecanismos de migração para o protocolo *Rollback* Solidário. Apesar do foco desse trabalho ser o desenvolvimento de tais mecanismos para o protocolo *Time Warp*, verifica-se que, com pequenas modificações, é possível adaptar os mecanismos de migração propostos para o protocolo *Rollback* Solidário.

A grande diferença do *Rollback* Solidário para o *Time Warp* é o uso de um processo observador para gerenciamento dos estados globais consistentes e recu-

perações do sistema quando da ocorrência de mensagens *stragglers*. Esse processo observador pode ser utilizado também para realizar os procedimentos de migração do processo mestre de ambos os mecanismos de migração propostos.

Para que não ocorra nenhuma influência no funcionamento do *Rollback* Solidário, devem existir duas *threads* no processo observador. Uma delas realizará o procedimento associado ao processo mestre para migração e a outra o procedimento necessário para o funcionamento do *Rollback* Solidário.

Os processos da simulação realizam os mesmos procedimentos para permitir a migração, ou seja, apresentam uma *thread* principal e uma *thread* de sincronismo cujos funcionamentos são idênticos dàquele baseado no uso do protocolo *Time Warp*.

O mecanismo de tratamento de mensagens *stragglers* no *Rollback* Solidário pode ser aproveitado para realizar o salvamento de estado do processo (*checkpointing*) para permitir a migração. Quando o processo mestre informa quais processos irão migrar, todos os processos salvam um estado, com seu respectivo vetor de dependências. Este vetor é enviado ao mestre, que, assim que receber o vetor de cada processo, determinará para qual estado global consistente (linha de recuperação) os processos deverão retornar após a migração. Quando os processos recebem essa informação, estão aptos a migrar.

Este procedimento pode ser utilizado para ambos os mecanismos de migração (coletiva e individual) uma vez que a migração envolverá a necessidade de identificação de uma nova linha de recuperação pelo processo observador, necessitando das informações dos estados de todos os processos.

Uma melhoria que pode ser implementada no protocolo *Rollback* Solidário, em relação ao protocolo *Time Warp*, diz respeito a utilização do temporizador. Como todos os *checkpoints* locais são informados ao observador, estas mensagens podem ser também utilizadas para verificar a carga do sistema, sem a necessidade explícita do observador solicitar esta informação periodicamente.

Não há necessidade de estado de pré-migração, pois, o retorno para um estado global consistente garante que não ocorrá a perda de mensagens durante a migração. Isso ocorre, uma vez que o protocolo *Rollback* Solidário já possui em sua estrutura o tratamento para as mensagens transientes.

É possível que o custo de migração considerando o uso do protocolo *Rollback* Solidário seja menor que o custo quando considerado o uso do *Time Warp* devido a naturalidade em se utilizar os *checkpoints*, utilizados para recuperação em casos de erro de causalidade, no retorno da migração. Além disso, o tempo perdido com trocas de mensagens no estado de pré-migração é evitado, já que este estado não é necessário para o mecanismo de migração baseado no *Rollback* Solidário.

6.5 Considerações Finais

Os mecanismos de migração coletiva e individual foram abordados neste capítulo. Em ambos os casos, faz-se uso de um processo mestre que coordena os processos da simulação. Estes, por sua vez, são formados por uma *thread* principal e uma *thread* de sincronismo.

Ideias gerais de como adaptar estes mecanismos de migração para o protocolo *Rollback* Solidário foram apresentadas, destacando a semelhança com os mecanismos de migração para o protocolo *Time Warp*.

7 Resultados Experimentais

Neste capítulo, apresentam-se os resultados experimentais em relação ao custo da migração de processos para cada um dos mecanismos projetados neste trabalho. Também é avaliado para quais circunstâncias cada mecanismo pode ter o melhor desempenho.

A partir do custo de migração de processos, será dada uma previsão teórica sobre a melhora de desempenho (redução no tempo de execução da simulação para os dados desejados) considerando o ganho que pode ser obtido através do uso de algoritmos de balanceamento desenvolvidos para a simulação distribuída.

7.1 Infraestrutura Física

Para obter os resultados experimentais, foi utilizado um *cluster* computacional com as seguintes especificações técnicas:

- Cinco computadores com as seguintes características:
  - Processador Intel(R) Core(TM)2 Quad CPU 2,66GHz;
  - Cache L2 8Mb;
  - Barramento frontal 1066 Mhz;
  - Memória RAM: 2 GBytes.
- Canal de comunicação com as seguintes características:

- Rede *Ethernet* 100 Mbits/s.
- *Switch*: 3Com Baseline Switch 2948-SFP Plus 48-Port Gigabit (3CBL SG48).

As implementações realizadas foram desenvolvidas na linguagem C, com a biblioteca OpenMPI versão 1.5.4 (GABRIEL; FAGG; BOSILCA, 2004) sobre o sistema operacional Linux Fedora 13.

7.2 Verificação do Funcionamento dos Mecanismos de Migração

Duas estratégias foram empregadas para verificar o correto funcionamento dos mecanismos projetados, que na prática significa a não influência da migração nos resultados da simulação.

Como foi adotado o tipo de migração fraca, visto que apenas os dados são transmitidos, a primeira estratégia consistiu em verificar se o LVT, a lista de eventos futuros, a lista de mensagens enviadas e a lista de estados salvos dos processos eram idênticos imediatamente antes da migração e imediatamente após a migração.

Na segunda estratégia foram analisados os arquivos de *log* criados pela simulação. Cada processo gerou um arquivo com eventos, mensagens *stragglers* e anti-mensagens tratadas. Através desses arquivos foi possível obter quais eventos foram de fato simulados por meio da eliminação de mensagens *stragglers* e anti-mensagens. Dessa forma, foram constatados que os eventos simulados em cada um dos mecanismos de migração e na implementação original do *Time Warp* foram os mesmos, demonstrando a não influência deles nos resultados da simulação.

7.3 Influência da Migração no Número de Mensagens Stragglers e Anti-Mensagens

Esta seção apresenta a influência dos mecanismos de migração no aumento do número de mensagens *stragglers* e anti-mensagens. Destaca-se que não foi utilizado um algoritmo de balanceamento, logo não há balanceamento após a migração e uma redução das mensagens *stragglers* e anti-mensagens não é o esperado. Considerando que um aumento do número de mensagens *stragglers* e anti-mensagens pode ocorrer.

Para avaliar essa influência foram considerados modelos de simulação com um número variável de processos com igual probabilidade (9%) de criação de eventos entre si e o avanço no tempo lógico de tratamento de eventos entre 1 a 10 considerando uma distribuição uniforme. O algoritmo 1 apresenta o comportamento destes modelos. Eles foram simulados através da implementação do protocolo *Time Warp* (sem migração), no laboratório GPESC (Grupo de Pesquisas em Engenharia de Sistemas e de Computação) da UNIFEI. Os modelos também foram simulados em cada um dos mecanismos de migração. No caso da migração individual, avaliou-se a influência quando apenas um processo passa por migração. Assim, foi contabilizado o número de mensagens *stragglers* e de anti-mensagens ocorridas em cada caso. Este experimento foi realizado considerando um total de 10.000 iterações do *loop* principal do *Time Warp*.

Considerando o temporizador do processo mestre com um intervalo de 60 segundos e como critério de parada para a simulação o número de iterações, obteve-se os resultados experimentais apresentados na tabela 1, que mostra, em termos de aumento percentual em relação à implementação original do *Time Warp*, o número de mensagens *stragglers* e anti-mensagens para o mecanismo de migração coletiva. Por sua vez, a tabela 2 apresenta os resultados para o mecanismo de migração individual. A figura 17 apresenta um gráfico ilustrando esses resultados.

Os resultados mostram que há um aumento muito pequeno no número de

Algorithm 1 Implementação do Modelo Utilizado nos Experimentos

```
x <− número aleatório entre 0 a 99 (Distribuição Uniforme)
if x < 9 then
  Cria Evento para Processo Local e Adiciona na Lista de Eventos Futuros.
end if
if Lista_Eventos_Futuros não vazia then
  e <− primeiro evento da Lista_Eventos_Futuros
  Remover da Lista_Eventos_Futuros
  t = número aleatório entre 1 a 10 (Distribuição Uniforme)
  if LVT > timestamp do evento then
    LVT = LVT + t
  else
    LVT = t + timestamp do evento
  end if
  x <− id_processo
  while x = id_processo do
    x <− número aleatório entre 1 a 10 (Distribuição Uniforme)
    Criar evento para processo x
  end if
end if
```

Nº. de Processos	Aumento Mens. *Straggler* (%)	Aumento Anti-Mens. (%)
2	$0{,}0301 \pm 0,0028$	$0{,}210 \pm 0,019$
3	$0{,}0323 \pm 0,0031$	$0{,}213 \pm 0,018$
4	$0{,}0349 \pm 0,0030$	$0{,}215 \pm 0,017$
5	$0{,}0380 \pm 0,0034$	$0{,}220 \pm 0,021$
6	$0{,}0447 \pm 0,0035$	$0{,}224 \pm 0,023$
7	$0{,}0506 \pm 0,0037$	$0{,}228 \pm 0,023$
8	$0{,}0546 \pm 0,0032$	$0{,}239 \pm 0,025$
9	$0{,}0599 \pm 0,0043$	$0{,}250 \pm 0,024$
10	$0{,}0635 \pm 0,0042$	$0{,}264 \pm 0,025$

Tabela 1: Influência do mecanismo de migração coletiva no número de mensagens *stragglers* e anti-mensagens ($\overline{x} \pm DP,\ n = 12$)

Nº de Processos	Aumento Mens. *Straggler* (%)	Aumento Anti-Mens. (%)
2	$0{,}141 \pm 0,010$	$1{,}034 \pm 0,092$
3	$0{,}149 \pm 0,012$	$1{,}062 \pm 0,098$
4	$0{,}154 \pm 0,016$	$1{,}089 \pm 0,082$
5	$0{,}158 \pm 0,013$	$1{,}127 \pm 0,102$
6	$0{,}163 \pm 0,011$	$1{,}171 \pm 0,130$
7	$0{,}170 \pm 0,017$	$1{,}220 \pm 0,099$
8	$0{,}182 \pm 0,019$	$1{,}268 \pm 0,121$
9	$0{,}194 \pm 0,018$	$1{,}325 \pm 0,118$
10	$0{,}201 \pm 0,023$	$1{,}387 \pm 0,146$

Tabela 2: Influência do mecanismo de migração individual no número de mensagens *stragglers* e anti-mensagens ($\overline{x} \pm DP,\ n = 12$)

mensagens *stragglers* para ambos os mecanismos. Quanto ao número de anti-mensagens, vê-se que há um aumento mais pronunciado, particularmente no mecanismo de migração individual.

Esses resultados podem ser explicados pelo fato de que, quando os processos passam por migração, há um distanciamento de seus LVTs. Esse distanciamento leva à ocorrência de mensagens *stragglers* assim que os processos são reiniciados e trocam mensagens entre si. Esse efeito é mais intenso no mecanismo de migração individual, pois apenas alguns processos são migrados, tendo assim seus LVTs atrasados em relação aos processos que permanecem simulando.

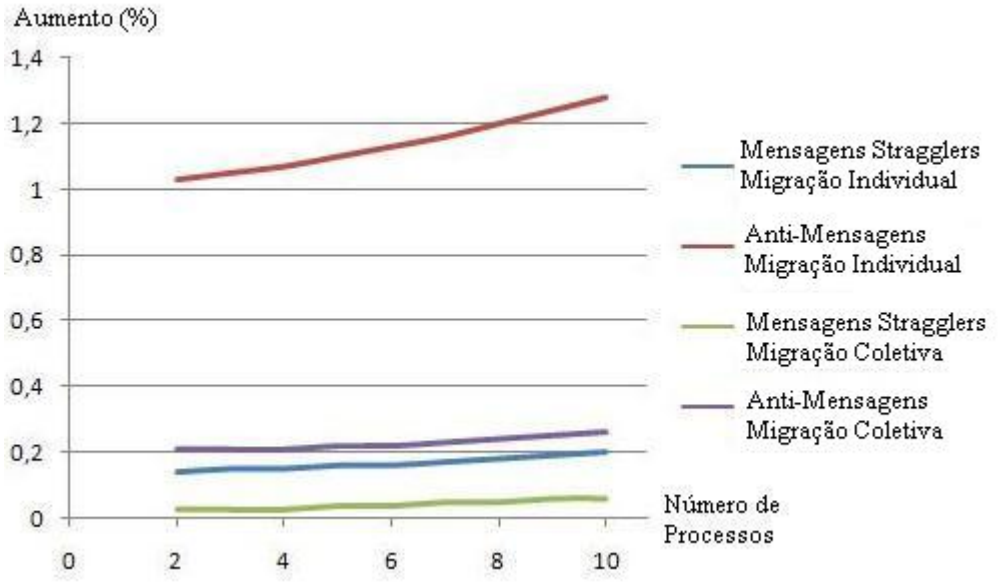

Figura 17: Influência dos mecanismos de migração no número de mensagem *stragglers* e anti-mensagem

Dos resultados conclui-se também que os mecanismos de migração propostos causam pequena influência no aumento de mensagens *stragglers* e anti-mensagens. Além disso, os mecanismos são escaláveis já que, à medida em que se aumenta o número de processos, observa-se um pequeno aumento no número de mensagens *stragglers* e anti-mensagens.

7.4 Tempo para Reinicialização dos Processos

Esta seção apresenta um fator que tem influência no desempenho de todo o mecanismo, ou seja, o tempo necessário para que seja realizada a reinicialização de um processo. O tempo em que o processo está sendo reinicializado corresponde a um período em que não há simulação de fato e, portanto, representa uma perda na simulação distribuída com migração.

A partir do momento em que algum processo entra no estado de pré-migração (independente do mecanismo de migração utilizado), este para de realizar a simu-

lação e apenas volta a simular após a recuperação de seus dados. Assim, o tempo para reinicialização considerado foi o período entre a entrada no estado de pré-migração até o reinício da simulação, sem levar em conta o tempo para salvamento e recuperação das variáveis.

O custo total de migração pode ser obtido pela soma dos tempos de reinicialização de processos (abordado nesta seção), recuperação e salvamento de variáveis (abordados na seção 7.5). Este procedimento pode ser adotado para ambos os mecanismos de migração propostos neste trabalho.

Testes experimentais foram realizados para avaliar o tempo de reinicialização e os resultados destes testes são apresentados nas próximas seções.

7.4.1 Mecanismo de Migração Coletiva

O desempenho do mecanismo de migração coletiva é muito afetado pelo tempo de reinicialização de um processo e isso se deve a sua característica básica: a migração simultânea de todos os processos da simulação.

O número de processos na simulação afeta o tempo de reinicialização dos processos, pois há um aumento de mensagens recebidas e enviadas no estado de pré-migração.

Além disso, outro fator de influência no tempo de reinicialização é o desempenho da função do MPI utilizada para a criação de processos (neste caso a função `MPI_Comm_spawn_multiple`). Essa função tem seu tempo de execução diretamente afetado pelo número de processos que estão sendo criados e, portanto, influencia no tempo para reinicialização dos processos em função do número deles na simulação. Neste contexto, o tempo de reinicialização será avaliado em função do número de processos.

O experimento consistiu em migrar os processos e medir o tempo de reinicialização variando o número de processos na simulação. Foram realizados dez

experimentos para permitir obter a média e desvio padrão para cada situação. A tabela 3 resume os resultados experimentais obtidos.

Número de Processos	Tempo de Reinicialização (s)
2	$0{,}157 \pm 0{,}032$
3	$0{,}610 \pm 0{,}035$
4	$0{,}903 \pm 0{,}034$
5	$1{,}023 \pm 0{,}061$
6	$1{,}156 \pm 0{,}041$
7	$1{,}184 \pm 0{,}043$
8	$1{,}251 \pm 0{,}025$
9	$1{,}286 \pm 0{,}057$
10	$1{,}302 \pm 0{,}059$

Tabela 3: Tempo de reinicialização de processos no mecanismo de migração coletiva ($\overline{x} \pm DP,\ n = 10$)

A figura 18 apresenta um gráfico em que o eixo das abscissas representa o número de processos e o eixo das ordenadas o tempo gasto na reinicialização.

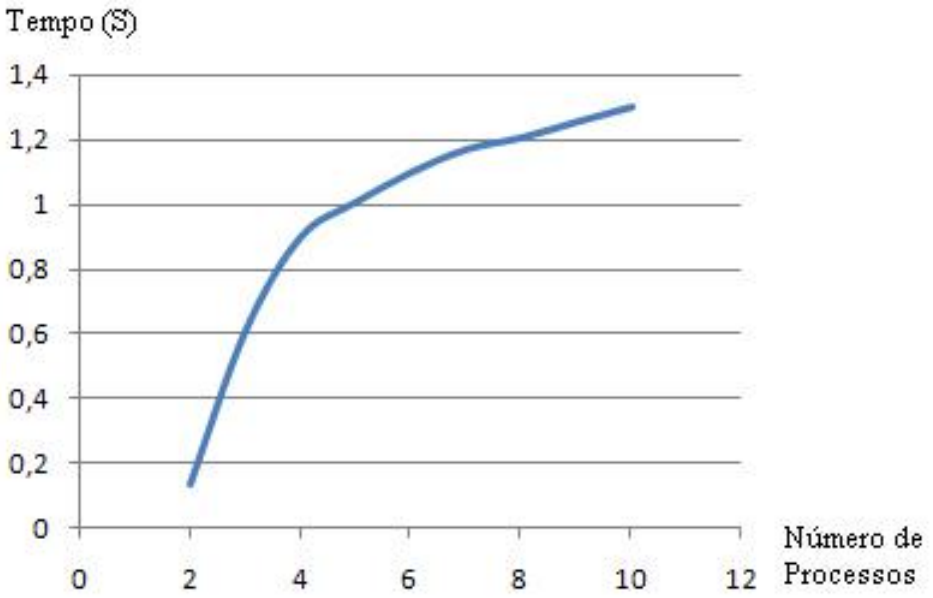

Figura 18: Tempo para reinicialização no mecanismo de migração coletiva

Pode-se verificar, nestes resultados, que o tempo de reinicialização aumenta à

medida em que aumenta o número de processos na simulação. Foi possível verificar experimentalmente que o tempo para a execução da chamada da função para criação dinâmica de processos (`MPI_Comm_spawn_multiple`) foi a maior responsável pelo tempo de reinicialização dos processos.

7.4.2 Mecanismo de Migração Individual

Ao contrário do mecanismo de migração coletiva, o desempenho do mecanismo de migração individual não é tão sensível ao tempo de reinicialização, pois realiza a migração apenas dos processos que necessitam realizar o balanceamento de carga. Entretanto, essa característica só poderá ser observada se, de fato, houver a necessidade de migração de apenas alguns processos da simulação.

Da mesma forma que o mecanismo de migração coletiva, o número de processos afeta o tempo de reinicialização, pois nessa implementação, a função para criação de processos utilizada, ou seja, `MPI_Comm_spawn`, é executada n vezes, n sendo número de processos que migraram. A figura 19 apresenta o resultado experimental obtido, em que o eixo das abscissas representa o número de processos e o eixo das ordenadas representa o tempo de reinicialização. A tabela 4 apresenta os dados experimentais obtidos quando se varia o número de processos que migram (todos os processos migraram apesar de se tratar do mecanismo de migração individual).

Um outro tipo de análise ainda pode ser realizado neste mecanismo. A análise consiste em verificar qual é o acréscimo de tempo que ocorre quando apenas um processo é migrado e o número de total de processos na simulação é alterado. Esse acréscimo ocorre devido à necessidade de recuperar os identificadores de um número maior de processos, bem como, a uma maior troca de mensagens associada aos avisos de estado de pré-migração entre os processos.

A tabela 5 apresenta os dados obtidos quando apenas um processo é migrado e o número de processos na simulação é alterado. Verifica-se que há um aumento

Número de Processos	Tempo de Reinicialização (s)
2	$0{,}490 \pm 0{,}030$
3	$0{,}626 \pm 0{,}045$
4	$0{,}728 \pm 0{,}023$
5	$0{,}808 \pm 0{,}070$
6	$1{,}089 \pm 0{,}066$
7	$1{,}125 \pm 0{,}089$
8	$1{,}339 \pm 0{,}100$
9	$1{,}400 \pm 0{,}067$
10	$1{,}602 \pm 0{,}097$

Tabela 4: Tempo de reinicialização de processos no mecanismo de migração individual ($\overline{x} \pm DP,\ n = 10$)

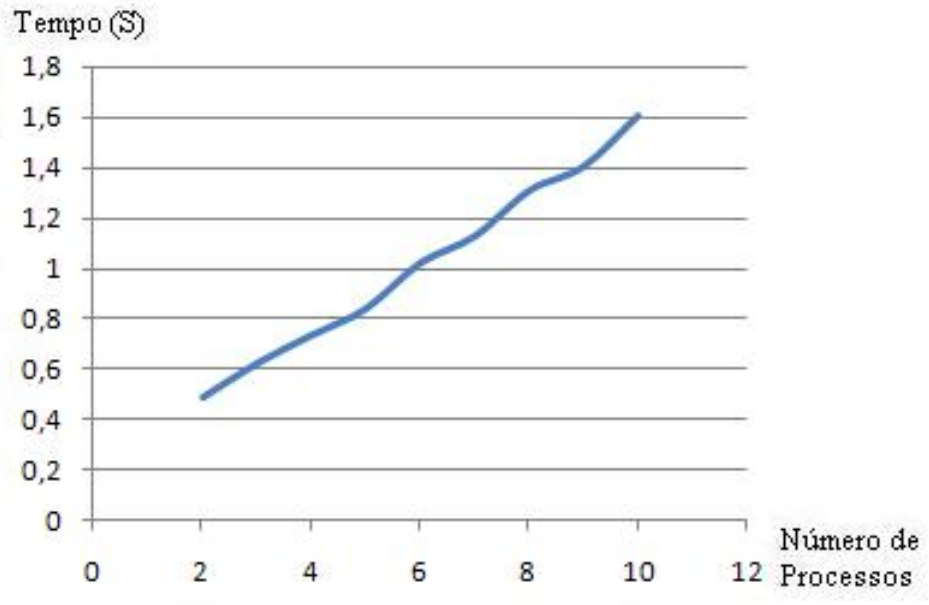

Figura 19: Tempo de reinicialização de processos no mecanismo de migração individual

do tempo de reinicialização para um único processo à medida em que se aumenta o número de processos na simulação. Experimentalmente, verificou-se que esse tempo era, em sua maior parte, associado com o tempo de conexões entre os processos, ou seja, o tempo para a recuperação dos identificadores entre eles. A figura 20 apresenta um gráfico com os resultados deste experimento.

Número de Processos	Tempo de Reinicialização (s)
2	$0{,}433 \pm 0,032$
3	$0{,}450 \pm 0,040$
4	$0{,}461 \pm 0,081$
5	$0{,}472 \pm 0,019$
6	$0{,}499 \pm 0,057$
7	$0{,}520 \pm 0,055$
8	$0{,}528 \pm 0,063$
9	$0{,}535 \pm 0,074$
10	$0{,}556 \pm 0,044$

Tabela 5: Tempo de reinicialização de um único processo no mecanismo de migração individual ($\overline{x} \pm DP,\ n = 10$)

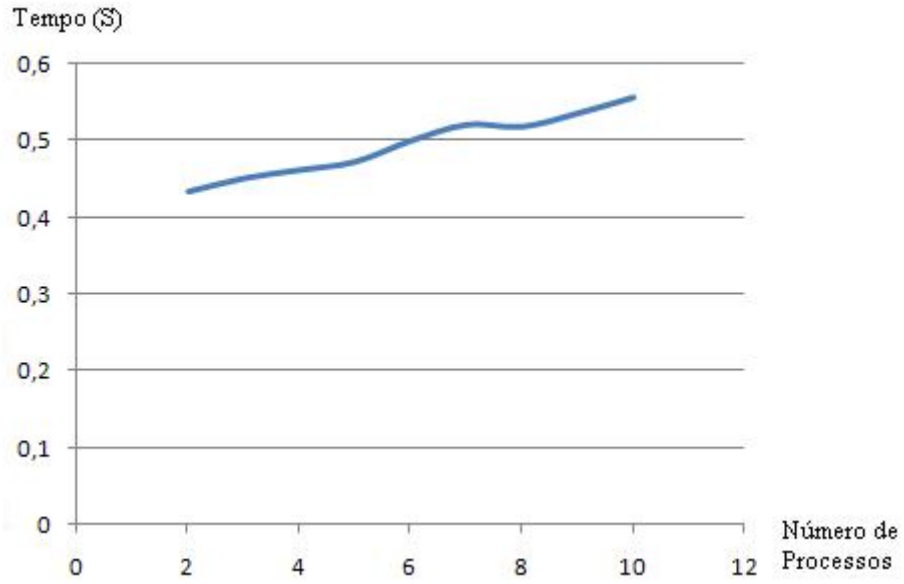

Figura 20: Tempo para reinicialização de um único processo no mecanismo de migração individual

7.5 Tempo para Salvamento e Recuperação das Variáveis

Neste trabalho, considera-se como tempo para recuperação das variáveis (LVT, lista de eventos futuros, de mensagens enviadas e de estados locais armazenados), o tempo gasto para obter os dados do servidor de arquivos (utilizando o NFS -

network file system), alocar memória e atribuir os seus valores.

A implementação utilizada foi a mesma para ambos os mecanismos, portanto, não há uma avaliação individual sobre esse aspecto em cada um deles. A figura 21 ilustra os resultados obtidos, em que o eixo das abscissas corresponde à quantidade de dados recuperados e o eixo das ordenadas, o tempo gasto.

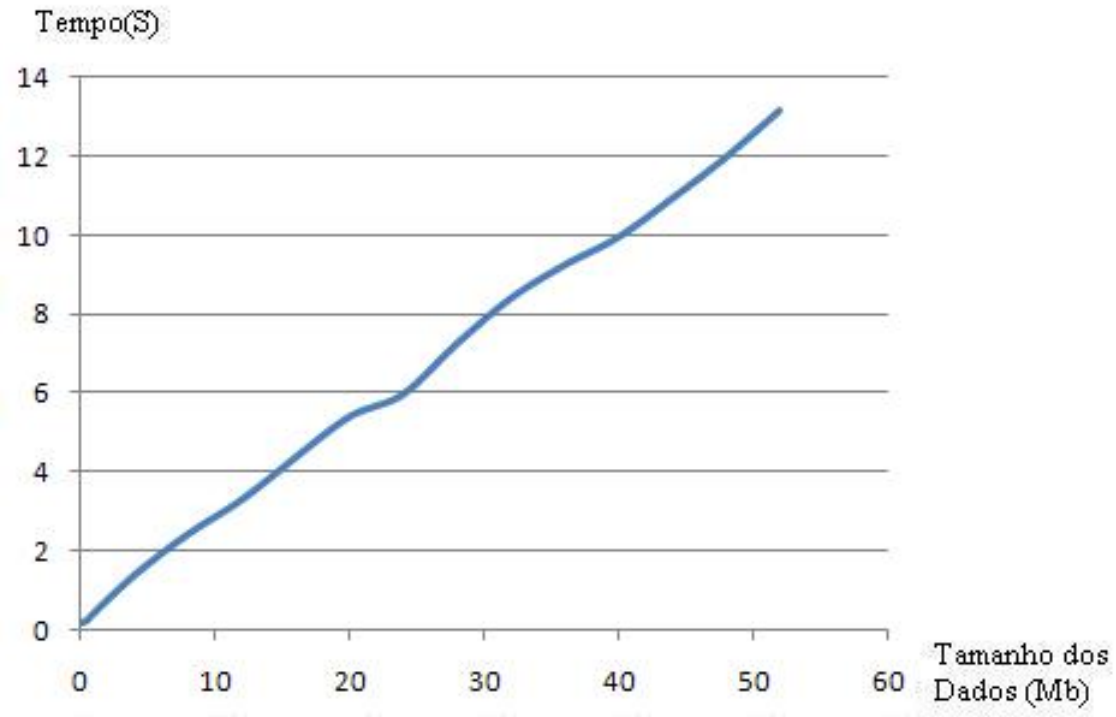

Figura 21: Tempo para recuperação das variáveis

Também foi realizado experimentos com a intenção de avaliar o tempo para o salvamento das variáveis. A figura 22 apresenta os resultados obtidos.

Para obter os resultados, diferentes tamanhos de dados foram considerados, onde, para cada tamanho, foi medido o tempo de salvamento e recuperação das variáveis (10 replicações do experimento foram realizadas para cada tamanho, ou seja, $n = 10$). O coeficiente de variação ($DP/\overline{x}$) para a recuperação de dados foi sempre menor que 6%. Para o salvamento de dados, o coeficiente de variação foi

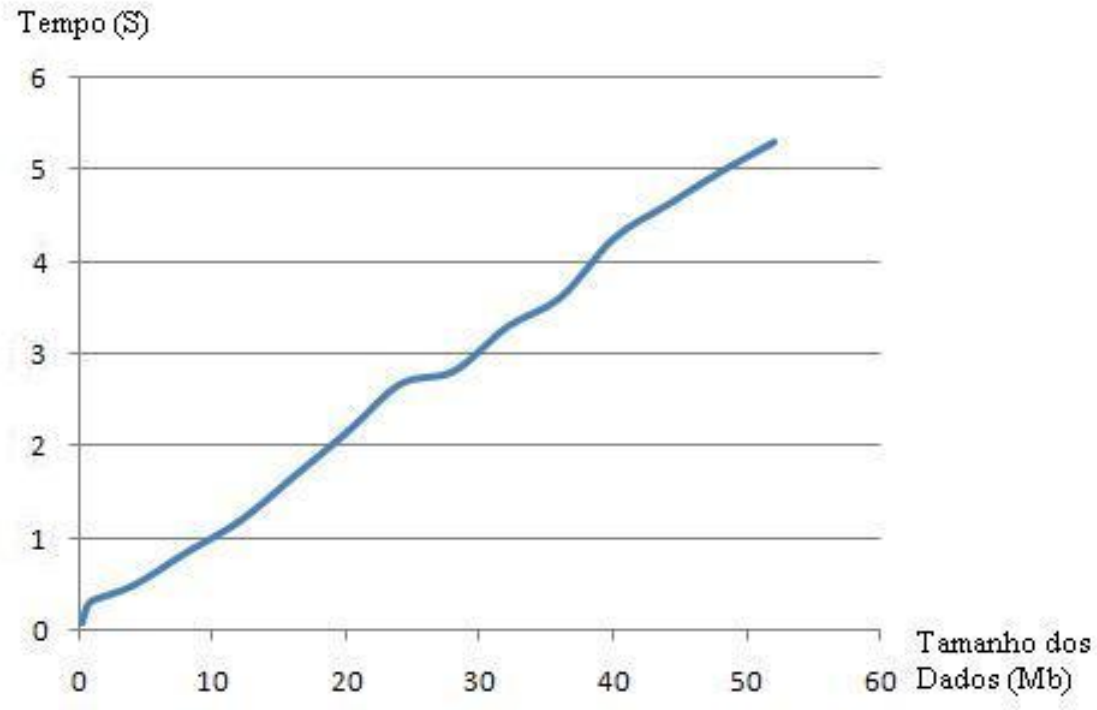

Figura 22: Tempo para salvamento das variáveis

sempre menor que 5%.

Desses resultados verifica-se que o tempo para recuperação das variáveis é maior do que o tempo para salvamento das mesmas. Em ambos os casos, o custo está associado com a troca de mensagens pela rede de comunicação. Entretanto, além do tempo de troca dos dados pela rede, o tempo de recuperação das variáveis é afetado pelo tempo de alocação dinâmica de memória. Experimentalmente, este tempo foi algumas vezes superior ao tempo de transferência de dados e é a principal razão para a diferença observada entre os tempos de recuperação e de salvamento.

Ressalta-se que ambos os tempos contribuem para o custo de migração de processos e, portanto, devem ser considerados. Como pode ser verificado, os tempos de salvamento e recuperação juntos podem chegar a valores maiores que 15 segundos, para estruturas maiores que 40 Mb. Esse tempo é muito superior ao tempo de reinicialização dos processos, como mostrado na seção 7.4.

Dessa forma, a importância da implementação de um mecanismo de *Garbage Collection* fica ressaltada, pois o tempo de recuperação e salvamento de dados é diretamente proporcional ao tamanho dos dados. Observa-se que esses dados consistem na lista de eventos futuros, lista de mensagens enviadas e lista de estados armazenados, sendo que este último corresponde à maior quantidade de dados associados e que cresce rapidamente, caso não seja implementado um mecanismo de liberação de memória.

Realizando a implementação de um mecanismo de *Garbage Collection*, verificou-se que o tempo de salvamento e recuperação de variáveis foi sempre menor do que 0,1 segundos para o como modelo de 10 processos, conforme descrito na seção 7.3.

O mecanismo de *Garbage Colletion* implementado consistiu em manter na lista de estados salvos um total de 100 estados, de forma que o total de dados sempre foi menor que 0,5 Mb. Este número de estados foi suficiente para garantir que não houvesse *rollbacks* para LVTs menores que o estado do menor LVT armazenado.

7.6 Desempenho e Custo Associados aos Mecanismos de Migração

O custo associado à implementação, neste trabalho, representa o percentual de processamento computacional necessário para o funcionamento dos mecanismos de migração.

Para fazer essa avaliação, foi medido o tempo para se executar 10.000 iterações da simulação, em duas situações. Na primeira, foi executada a simulação sem o uso de mecanismos de migração, ou seja, apenas o *Time Warp* em sua forma original. Na segunda, foram executados ambos os mecanismos de migração.

Assim, foi medido o tempo para execução em cada situação e o custo da implementação de cada mecanismo pode ser obtido de acordo com a expressão a

seguir:

$$Custo(\%) = 100 - \frac{100 * T_{tw}}{t_m}$$

Onde: t_m é o tempo gasto por um determinado mecanismo, e

T_{tw} é o tempo gasto pelo *Time Warp*.

Pode-se obter o desempenho associado à implementação da seguinte forma:

$$Desempenho(\%) = \frac{100 * T_{tw}}{t_m} = 100 - Custo$$

A tabela 6 apresenta os resultados experimentais obtidos para o custo e o desempenho da implementação em cada um dos mecanismos propostos. Este experimento foi replicado 10 vezes com a finalidade de obtenção dos dados estatíscos média e desvio padrão.

Mecanismo de Migração	Custo (%)	Desempenho (%)
Coletiva	4,2 $\pm$ 1,2	95,8 $\pm$ 1,2
Individual	6,9 $\pm$ 0,9	93,1 $\pm$ 0,9

Tabela 6: Custo e desempenho das implementações dos mecanismos de migração ($\overline{x} \pm DP, n = 10$)

Deve ser levado em consideração que a implementação em si, e não apenas o projeto do mecanismo, tem influência nos resultados obtidos. Entretanto, de alguma forma, a codificação extra é inevitável e inerente ao funcionamento do mecanismo, cujo custo pode ter impacto no desempenho e pode ser avaliado de acordo com o disposto neste trabalho.

Fica claro nesses resultados o desempenho superior do mecanismo de migração coletiva. Isto se deve a sua maior simplicidade de implementação, principalmente no que se refere ao uso do MPI. Experimentos demonstraram que a maior perda no mecanismo de migração individual ocorre na verificação da chegada de mensagens em diferentes *intercommunicators* (já que há um *intercommunicator* para cada processo).

Uma possibilidade de minimizar essa diferença seria fazer uso de funções do MPI que unem os *intercommunicators* associados a cada processo da simulação em um único *intracommunicator*.

7.7 Influência das Mensagens dos Mecanismos de Migração

O funcionamento dos mecanismos de migração faz uso de mensagens trocadas entre o processo mestre e a *thread* de sincronismo, bem como de mensagens trocadas no estado de pré-migração.

Para verificar o consumo de largura de banda que estas mensagens impõem, foi comparado o total de dados com o total de dados enviados referentes às mensagens necessárias ao funcionamento do mecanismo de migração.

O experimento para avaliar a influência destas mensagens consistiu em simular modelos com um número variável de processos e igual probabilidade de criação de eventos entre si. O temporizador do processo mestre foi de 60 segundos.

Nº de Processos	Mens. do Mec. Coletiva (%)	Mens. do Mec. Individual (%)
2	$0{,}098 \pm 0{,}018$	$0{,}113 \pm 0{,}019$
3	$0{,}103 \pm 0{,}015$	$0{,}118 \pm 0{,}012$
4	$0{,}109 \pm 0{,}021$	$0{,}121 \pm 0{,}017$
5	$0{,}114 \pm 0{,}028$	$0{,}129 \pm 0{,}023$
6	$0{,}121 \pm 0{,}019$	$0{,}134 \pm 0{,}020$
7	$0{,}124 \pm 0{,}023$	$0{,}136 \pm 0{,}031$
8	$0{,}127 \pm 0{,}034$	$0{,}142 \pm 0{,}028$
9	$0{,}132 \pm 0{,}032$	$0{,}147 \pm 0{,}027$
10	$0{,}143 \pm 0{,}030$	$0{,}153 \pm 0{,}040$

Tabela 7: Influência dos mecanismos de migração no uso da rede de comunicação ($\overline{x} \pm DP, n = 10$)

A figura 23 ilustra a influência dos mecanismos de migração sobre as mensagens trocadas na rede de comunicação.

Os resultados mostram que há um uso muito pequeno da rede de comunicação

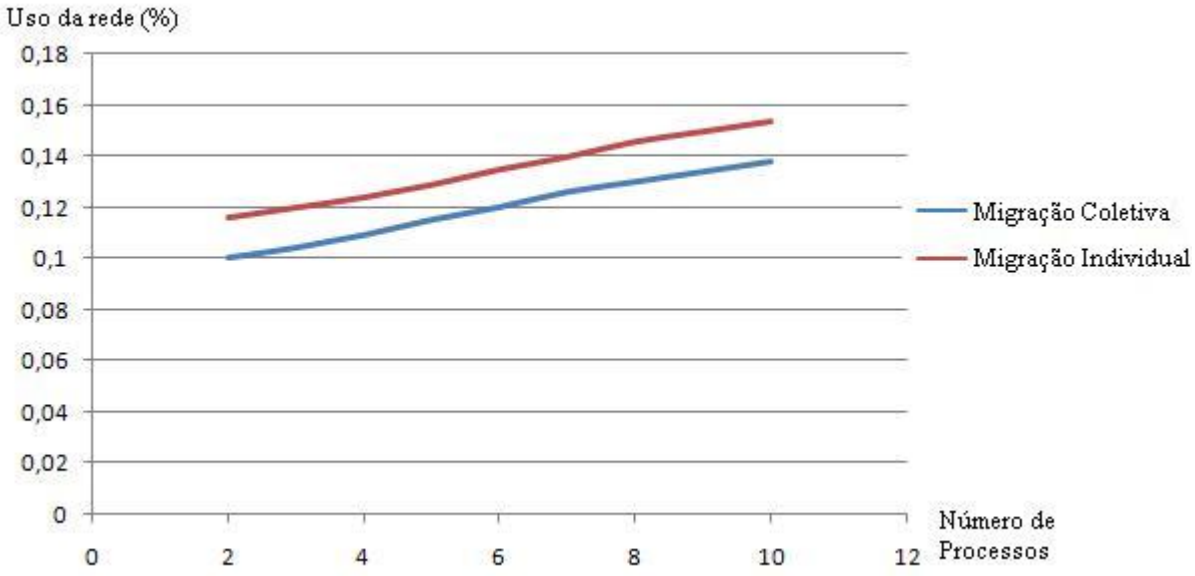

Figura 23: Influência dos mecanismos de migração no uso da rede de comunicação

para implementar os mecanismos de migração. Ademais, pode-se verificar a escalabilidade no uso da rede, pois o aumento do número de processos no modelo pouco afetou seu uso.

7.8 Influência do Temporizador do Processo Mestre

Ambos os mecanismos utilizam, no processo mestre, um temporizador que permite que a simulação ocorra durante um determinado período sem influências. Experimentalmente, verificou-se que a modificação do intervalo pode afetar:

- O número de mensagens *stragglers* e anti-mensagens: a redução desse tempo leva a uma maior frequência de erros de causalidade com o consequente aumento de *rollbacks* secundários.

- O uso da rede: um intervalo pequeno temporizador leva a uma maior frequência de troca de mensagens entre o processo mestre e os processos da simulação, bem como de mensagens de estado de pré-migração. Há também

maior frequência de migração, o que leva à maior frequência de salvamento e recuperação de dados.

A decisão de qual é o melhor intervalo para o temporizador deve ser tomada baseando-se nos itens mencionados, mas também, por quão rápida a frequência de criação de eventos se modifica ao longo da simulação. Caso se modifique rapidamente, um tempo curto para o temporizador é mais adequado (pois a falta de balanceamento na simulação surgirá mais rápido). Por outro lado, na situação em que a frequência na geração se altera lentamente, um tempo maior é mais indicado.

7.9 Previsão Teórica para o Ganho de Desempenho

Esta seção procura prever o ganho que pode ser obtido com o uso dos mecanismos de migração propostos, levando em consideração o uso de algoritmos de balanceamento desenvolvidos para a simulação distribuída.

Um parâmetro que influencia nesse sistema é o período do temporizador no processo mestre, cujo valor adotado foi de 60 segundos.

Considerando os resultados experimentais apresentados nesse capítulo e o resultado de possíveis ganhos que podem ser obtidos com o uso de algoritmos de balanceamento apresentados no capítulo 4, procurou-se prever o desempenho que pode ser obtido em cada um dos mecanismos de migração desenvolvidos.

Considerou-se que, a cada ciclo de migração e reinicialização, os processos se encontravam totalmente desbalanceados. Dessa forma, considerou-se que o ganho ao aplicar o algoritmo é a metade daquele obtido na literatura (referências citadas no capítulo 4), devido a sua progressiva perda de balanceamento ao longo do ciclo. Além disso, foi considerado o modelo de 10 processos com probabilidades (%9) de geração de eventos iguais entre si. Também foram utilizados os resultados dos experimentos para a composição do tempo de migração.

Equações são propostas para a eficiência final dos mecanismos de migração

com relação ao *Time Warp* sem migração.

7.9.1 Mecanismo de Migração Coletiva

A eficiência do mecanismo de migração coletiva, comparado com o *Time Warp* sem migração, pode ser previsto por:

$$e = \frac{d \cdot timer}{timer + t_m} \tag{7.1}$$

Onde: d é o desempenho da implementação do mecanismo de migração;

timer é o intervalo do temporizador do processo mestre (60 segundos);

t_m é o tempo de migração.

Com os dados experimentais, e usando a equação 7.1, obtém-se:

$$e = \frac{95,8 \cdot 60}{60 + 1,4} = 93,62\%$$

Com o valor do desempenho pode-se abordar qual seria o ganho final com a seguinte equação:

$$g = 100 - 100 \cdot \frac{1 - g_p/2}{e} \tag{7.2}$$

Onde: g_p é o ganho percentual do algoritmo utilizado em termos de redução no tempo de execução da simulação.

Segundo o trabalho de Carothers e Fujimoto (2000), é possível obter uma redução do tempo de execução em torno de 27% com o uso do algoritmo de balanceamento *Background Execution* (BGE), descrito no capítulo 4. Com o uso da equação 7.2, obtém-se:

$$g = 100 - 100 \cdot \frac{100 - 27/2}{93,62} = 7,61\%$$

Neste contexto, é possível obter uma redução de 7,61% no tempo total de

simulação com o uso do mecanismo de migração coletiva de processos juntamente com o algoritmo BGE.

7.9.2 Mecanismo de Migração Individual

No mecanismo de migração individual, a eficiência, comparada com o *Time Warp* sem migração, pode ser dada por:

$$e = \frac{d \cdot [timer + f \cdot t_m \cdot (1 - p)]}{timer + t_m} \tag{7.3}$$

Onde: d é o desempenho da implementação do mecanismo de migração;

timer é o intervalo do temporizador do processo mestre;

p é a porcentagem média de processos que migram na simulação;

t_m é o tempo para a migração no mecanismo de migração individual;

f é um fator percentual associado à frequência de envio e recebimento de mensagens pelos processos que migram.

O fator f representa a porcentagem média do tempo de processamento (durante a migração), dos processos que não migraram, que não foi perdido devido à mensagens *stragglers* provindas de processos migrados que pararam sua simulação (e tiveram seus LVTs atrasados).

É interessante notar que, se o fator f é nulo, a equação 7.3 se reduz a equação 7.1. Portanto, o benefício que poderia ser obtido pela continuidade da simulação pelos processos que não migram é perdido caso sofram *rollbacks* de processos que tiveram seus LVTs atrasados devido à migração assim que estes retornem à simulação.

Também nota-se que, se todos os processos migram ($p = 1$), a equação 7.3 se reduz a 7.1, o que é de se esperar, já que todos os processos migram simultaneamente.

Uma forma de inferir sobre o fator f é calcular a razão entre a frequência de eventos recebidos pela frequência de eventos enviados para cada processo migrado e, em seguida, realizar a média destes valores. A razão mencionada está relacionada com a probabilidade de um processo migrado de enviar ou receber mensagens imediatamente após seu retorno à simulação. Caso envie uma mensagem, esta terá maior probabilidade de ocasionar um erro de causa e efeito (pois o LVT do processo se atrasa durante a migração). Por outro lado, se este processo receber uma mensagem de um processo não migrado, seu LVT poderá ser adiantado.

Com os dados experimentais e considerando uma média de migração de 50% (p = 0,5) e o fator $f = 0,5$, usando a equação 7.3, tem-se:

$$e = \frac{93,1 \cdot [60 + 0,5 \cdot 1,2 \cdot (1 - 0,5)]}{60 + 1,2} = 91,73\%$$

Assim, usando a equação 7.2, o ganho final neste contexto será:

$$g = 100 - 100 \cdot \frac{100 - 27/2}{91,73} = 5,70\%$$

Esse resultado indica que há uma redução de 5,70% no tempo de execução da simulação com uso do mecanismo de migração individual juntamente com o algoritmo BGE.

7.10 Decisão de Qual Mecanismo de Migração Utilizar

Os resultados experimentais obtidos com relação ao tempo de reinicialização de processos (seção 7.4), a influência no número de mensagens *stragglers* e anti-mensagens (seção 7.3) e o custo associado ao mecanismo (seção 7.6), permitem inferir que o mecanismo de migração coletiva apresenta melhor desempenho em várias condições.

A vantagem existente no mecanismo de migração individual consiste em permitir que os processos que não migram continuem simulando. No entanto, para

que se tire proveito dessa característica, é necessário que o processamento realizado por estes processos durante a migração de outros processos não seja perdido devido a mensagens *stragglers* que podem surgir após a migração considerando o distanciamento dos LVTs (que é mais acentuado no mecanismo de migração individual). A probabilidade de surgirem tais mensagens é proporcional à frequência de envio de mensagens pelos processos que passam por migração.

Assim, pode-se afirmar que o mecanismo de migração individual será mais indicado quando as seguintes características de simulação forem satisfeitas:

1. Uma porcentagem pequena do número de processos necessita de migração;
2. Os processos que necessitam de migração são os mesmos que apresentam baixa frequência de envio de mensagens em relação à frequência de mensagens recebidas.
3. O comportamento do modelo muda rapidamente de forma que o tempo do temporizador do processo mestre é pequeno em relação ao tempo gasto para todo o procedimento de migração (no mecanismo de migração coletiva).

Caso qualquer uma destas características não seja satisfeita, o mecanismo mais indicado para realização da migração é o coletivo.

As equações 7.1 e 7.3 propostas para o desempenho global dos mecanismos de migração podem auxiliar quantitativamente na decisão de qual mecanismo de migração utilizar.

Além disso, com as equações 7.1 e 7.3, é possível especificar um algoritmo que tome a decisão de qual mecanismo de migração utilizar em tempo de execução. Uma questão a ser resolvida, nesse contexto, é como realizar a mudança de mecanismo de migração dinamicamente. Uma possível abordagem é, antes da reinicialização dos processos pelo processo mestre, verificar qual mecanismo utilizar. Assim, se o ideal é usar o mecanismo coletivo, utilizam-se os procedimentos

envolvidos neste após a reinicialização dos processos. Caso contrário, utilizam-se os procedimentos do mecanismo de migração individual.

7.11 Considerações Finais

Resultados experimentais avaliando os mecanismos de migração propostos foram apresentados neste capítulo. Os resultados mostraram que o mecanismo de migração coletiva foi superior ao mecanismo de migração individual.

Uma importante conclusão obtida na análise dos experimentos, medindo o tempo de salvamento e recuperação de variáveis, foi a verificação da necessidade de implementação de um mecanismo de *Garbage Collection* em ambos os mecanismos, para evitar uma elevada perda de desempenho com o salvamento e a recuperação dos estados dos processos.

Foi apresentada uma previsão teórica para o ganho de desempenho que cada mecanismo pode oferecer. Supondo algumas hipóteses, ambos os mecanismos demonstraram a possibilidade de ganho de desempenho.

Por fim, foram explicitadas as condições do modelo de simulação sob as quais o mecanismo de migração individual é o mais indicado. Se essas condições não forem satisfeitas, o mecanismo de migração coletiva torna-se o mais indicado. As equações 7.1 e 7.3 permitem definir quantitativamente qual mecanismo usar.

8 Conclusões

Este trabalho apresentou dois mecanismos de migração de processos para Simulação Distribuída com base no protocolo otimista *Time Warp*. Ambos os mecanismos se baseiam no uso de um processo mestre que gerencia os processos da simulação. Estes, por sua vez, apresentam duas *threads*: a principal, que realiza a simulação, e a de sincronismo, que se comunica com o processo mestre verificando a cada momento qual procedimento o processo da simulação deve realizar.

O mecanismo de migração coletiva consiste na migração de todos os processos simultaneamente, ou seja, todos os processos são encerrados e reinicializados simultaneamente. Já o mecanismo de migração individual consiste na migração de um ou mais processos sem a necessidade de migrar todos ao mesmo tempo.

Uma comparação entre os mecanismos de migração propostos foi realizada e verificou-se que o mecanismo coletivo apresenta maior facilidade de implementação (levando em consideração o uso da biblioteca de comunicação MPI), menor tempo de migração e menor influência sobre o número de mensagens *stragglers* e anti-mensagens. Por outro lado, o mecanismo de migração individual pode apresentar maior desempenho quando apenas alguns processos da simulação necessitam passar por migração, já que, esse mecanismo permite que os processos que não migram continuem simulando. Além disso, este mecanismo é mais flexível, considerando-se a possibilidade de construção de mecanismos de mapeamento dinâmico dos processos da simulação.

Foi possível prever teoricamente o ganho de desempenho que cada mecanismo

de migração pode proporcionar fazendo uso de informações sobre o ganho que um determinado algoritmo de escalonamento pode oferecer. Em ambos os casos foi possível verificar ganhos de desempenho.

Por fim, salienta-se que os testes realizados para ambos os mecanismos propostos se baseiam no uso da biblioteca de comunicação MPI e os resultados foram afetados pelos recursos oferecidos por essa biblioteca.

8.1 Contribuições deste Trabalho

A maior contribuição deste trabalho é a proposta de dois mecanismos de migração de processos para a simulação distribuída baseada no uso do *Time Warp*, com a capacidade de obter, em tempo de execução, informações sobre o andamento da simulação que podem ser utilizadas por um algoritmo de escalonamento.

Uma das possíveis implementações para cada um dos mecanismos de migração foi realizada, baseando-se nos recursos disponíveis na biblioteca MPI.

Outra contribuição deste trabalho são os resultados experimentais obtidos, fornecendo informações sobre o funcionamento dos mecanismos propostos. Uma importante conclusão obtida desses resultados é a inviabilidade da implementação da migração sem o uso de um mecanismo de liberação de memória ou *Garbage Collection*, devido ao elevado tempo associado ao salvamento e restauração dos estados dos processos.

Por fim, esse trabalho permitiu prever que há a possibilidade de ganho de desempenho de ambos os mecanismos de migração.

8.2 Sugestões para Trabalhos Futuros

Por se tratar de mecanismos novos para migração de processos, várias questões podem ser exploradas como continuidade deste projeto. Entre essas questões cita-

se:

1. A implementação de algoritmos de escalonamento de forma a utilizá-los em conjunto com os mecanismos de migração propostos e verificar, experimentalmente, o ganho que pode ser obtido.

2. A otimização dos mecanismos propostos de modo a obter maior desempenho.

3. O projeto de mecanismos de migração de processos para simulação distribuída baseada no protocolo *Rollback* Solidário.

4. Ampliação dos testes realizados considerando outras configurações arquiteturais e diferentes modelos de simulação.

5. O estudo do impacto das técnicas de gerenciamento de memória nos mecanismos de migração de processos.

Referências

ANGELO, D. G. Parallel and distributed simulation from many cores to the public cloud. p. 14–23, 2011.

BABAOGLU, O.; MARZULLO, K. Consistent global states of distributed systems: Fundamental concepts and mechanisms. *In: MULLENDER, S. (Ed.). Distributed Systems.*, Addison-Wesley, New York, p. 55–96, 1993.

BANKS, J.; CARSON, J. S.; NELSON, B. L. *Discrete Event-System Simulation.* 2. ed. NJ: Prenticce-Hall, Englewood Cliffs, 1996.

BOUKERCHE, A.; DAS, S. K. Dynamic load balancing strategies for conservative parallel simulations. *Proceedings of the 11th workshop on Parallel and distributed simulation*, p. 20–28, 1997.

BRUSCHI, S. M. *ASDA - Um Ambiente de Simulação Distribuída Automático.* Tese (Doutorado) — ICMC - USP, 2003.

BRYANT, R. E. Simulation of packet communication architecture computer systems. *MIT-LCS-TR-188*, Massachussets, v. 7, n. 3, p. 404–425, 1977.

BURDORF, C.; MARTI, J. Load balancing strategies for time warp on multiuser workstations. *The Computer Journal*, v. 36, n. 2, p. 168–176, 1993.

CAI, W.; TURNER, S. J. An algorithm for distributed discrete-event simulation: the "carrier null message" approach. *Proceedings of the SCS Multiconference on Distributed Simulation*, v. 22, n. 1, p. 3–8, 1990.

CAO, J.; LI, Y.; GUO, M. Process migration for mpi applications based on coordinated checkpoint. *11th International Conference on Parallel and Distributed Systems*, v. 1, p. 306–312, 2005.

CARDOSO, J.; VALLETE, R. *Redes de Petri.* Florianópolis: Editora da UFSC, 1997.

CAROTHERS, C. D.; FUJIMOTO, R. M. Efficient execution of time warp programs on heterogeneous now platforms. *IEEE Transactions on Parallel and Distributed Systems*, v. 11, n. 3, 2000.

CARVALHO, O. A. de J. *Avaliação de políticas de escalonamento para execução de simulações distribuídas*. Dissertação (Mestrado) — ICMC - USP, 2008.

CHANDY, K. M.; MISRA, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, v. 5, n. 5, p. 440–452, 1979.

CHANDY, K. M.; MISRA, J. Assynchronous distributed simulation via a sequence of parallel computations. *Communications of ACM*, v. 24, n. 11, p. 198–205, 1981.

CHAUDHARY, V.; WALTERS, J. P. N.; JIANG, H. *Computation Checkpointing and Migration*. NY, USA: New Science Publishers, Inc. Commack, 2010.

CHWIF, L.; PAUL, R. J.; BARRETTO, M. R. P. Discrete event simulation model reduction: A causal approach. *Simulation Modelling Practice and Theory*, EUA, v. 4, p. 930–944, 2006.

CORTELLESSA, V.; QUAGLIA, F. An analysis of the efficiency of optimistically synchronized parallel simulators. *First Conference on Simulation Modeling and Applications*, 1998.

COULOURES, G.; DOLLIMORE, J.; KINDBERG, T. *Sistemas Distribuídos - Conceitos e Projeto*. [S.l.]: Bookman, 2007.

DEELMAN, E.; SZYMANSKI, B. K. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems. *Proceedings of the 12th workshop on Parallel and distributed simulation - PADS*, p. 46–53, 1998.

DONGARRA, J. et al. *MPI: The Complete Reference*. 2. ed. New York: The MIT Press, 1998.

DOUGLIS, F.; OUSTERHOUT, J. Transparent process migration: Desing alternatives and the sprite implentation. v. 21, n. 8, p. 757–785, 1991.

EWING, G. C.; PAWLIKOWSKI, K.; MCNICKLE, D. Akaroa-2: Exploiting network computing by distributing stochastic simulation. *Proceeding of European Simulation Multiconference (ESM)*, Warson - Poland, p. 175–181, 1999.

FERRARI, D.; ZHOU, S. An empirical investigation of load indices for load balancing applications. *Proceedings of the 12th Symposium on Computer Performance Modeling, Measurement, and Evaluation (Performance'87)*, p. 515–528, 1987.

FERSCHA, A. Parallel and distributed simulation of discrete event systems. *In: Handbook of Parallel and Distributed Computing*, McGraw-Hill, 1995.

FUJIMOTO, R. M. Parallel discrete event simulation. Communications of the ACM, v. 33, p. 30–53, 1990.

FUJIMOTO, R. M. *Parallel and Distributed Simulation Systems*. New York: John Wiley & Sons, 2000.

FUJIMOTO, R. M.; HYBINETTE, M. Computing global virtual time in shared-memory multiprocessors. *ACM Trans. Model. Comput. Simul.*, v. 7, n. 4, 1997.

FUJIMOTO, R. M.; MALIK, A. W.; PARK, A. J. Parallel and distributed simulation in the cloud. *SCS MS Magazine*, 2010.

GABRIEL, E.; FAGG, G. E.; BOSILCA, G. Open mpi: Goals, concept, and design of a next generation mpi implementation. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary: [s.n.], 2004. p. 97–104.

GEROFI, B.; FUJITA, H.; ISHIKAWA, Y. An efficient process live migration mechanism for load balanced distributed virtual environments. *IEEE International Conference on Cluster Computing (CLUSTER)*, p. 197–206, 2010.

GIOIOSA, R. et al. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2005.

GLAZER, D. W.; TROPPER, C. On process migration and load balancig in time warp. *IEEE Transactions on Parallel and Distributed Systems*, p. 318–327, 1993.

GLYNN, P. W.; HEIDELBERGER, P. Analysis of initial transient deletion for parallel steady-state simulations. *SIAM J. Scientific Stat. Computing*, v. 13, n. 4, p. 904–922, 1992.

HAKIM, M.; JAIS, J.; SALWA, S. Mosix: Implementation, trend and benchmark in malaysia. *International Symposium on Information Technology. ITSim 2008*, p. 1–6, 2008.

HAREL, D. et al. The formal semantics of statecharts. *Proceeding of the 2nd IEEE Symposium on Logic in Computer Science*, Nova York, p. 54–64, 1987.

IBRAHIM, K. Z. et al. Optimized precopy live migration for memory intensive applications. *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

JANKOWSKI, G. et al. Towards grid checkpoint architecture. *PPAM 2005 Poznan*, 2005.

JEFFERSON, D. R. Virtual time. *ACM Transactions on Programming Languages and Systems*, v. 7, n. 3, p. 404–425, 1985.

KURVE, A.; GRIFFIN, C.; KESIDIS, G. Iterative partitioning scheme for distributed simulation of dynamic networks. *16th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, p. 92–96, June 2011.

LIU, Q.; WAINER, G. Lightweight time warp: A novel protocol for parallel optimistic simulation of large-scale devs and cell-devs models. *Symposium on Distributed Simulation and Real-Time Applications. 12th IEEE-ACM International*, p. 131–138, 2008.

LIU, T.; MA, Z.; OU, Z. A novel process migration method for mpi applications. *15th IEEE Pacific Rim International Symposium on Dependable Computing. PRDC'09*, p. 247–251, 2009.

LOBATO, R. S.; ULSON, R. S.; SANTANA, M. J. A revised taxonomy for time warp based distributed synchronization protocols. *Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications. DS-RT 2004*, p. 226–229, 2004.

LOTTIAUX, R. et al. Openmosix, openssi and kerrighed: a comparative study. *IEEE International Symposium on Cluster Computing and the Grid. CCGrid*, p. 1016–1023, 2005.

LOW, Y. et al. Survey of languages and runtime libraries for parallel discrete event simulation. *Technical Article Simulation*, v. 72, n. 3, p. 170–186, 1999.

MANIVANNAN, D.; SINGHAL, M. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, v. 10, n. 7, p. 703–713, July 1999.

MATTERN, F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, v. 18, n. 4, p. 423–434, 1993.

MELLO, R. F. de; SENGER, L. J. A new migration model based on the evaluation of processes load and lifetime on heterogeneous computing environments. *16th Brazilian Symposium on Computer Architecture and High Performance Computing. SBAC-PAD*, p. 222–227, 2004.

MILOJICIC, D.; DOUGLIS, F.; PAINDAVEINE, Y. Process migration. *ACM Computing Surveys*, 2000.

MISRA, J. Distributed discrete-event simulation. *Computer Surveys*, v. 18, p. 39–65, 1986.

MOREIRA, E. M. *Rollback Solidário: Um Novo Protocolo Otimista para a Simulação Distribuída*. Tese (Doutorado) — Universidade de São Paulo - USP, Instituto de Ciências Matemáticas e de Computação, São Carlos, 2005.

MPI-FORUM, M. P. I. *MPI: Message Passing Interface Standard Version 2.1*. Knoxville, Tennessee, 2008.

NICOL, D. M.; REYNOLDS, P. F. Problem-oriented protocol design. *Proceedings of the 16th Conference on Winter Simulation*, p. 471–474, 1984.

PERIN, C. F. *Introdução à Simulação de Sistemas*. Nova York: Editora da Unicamp, 1995.

PESCHLOW, P.; HONECKER, T.; MARTINI, P. A flexible dynamic partitioning algorithm for optimistic distributed simulation. *International Workshop on Principles of Advanced and Distributed Simulation - PADS*, p. 219–228, 2007.

PRADO, D. *Teoria das Filas e da Simulação*. 4. ed. Nova Lima, MG - Brasil: INDG-TECS, 1999.

RAJAEI, H. Time warp: An implementation and performance analysis. *21st International Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2007.

RAVASSI, J. F. *JUMP: Uma política de escalonamento unificada com migração de processos*. Dissertação (Mestrado) — ICMC - USP, 2009.

REGO, V. J.; SUNDERAN, V. S. Concurrent stochastic simulation: Experiments with eclipse. *Proc. Int. Conf. Perf. of Distributed Systems and Integrated Communication*, p. 253–271, 1991.

REITHER, P. L.; JEFFERSON, D. Dynamic load management in the time warp operating system. *Transation Society for Computer Simulation*, v. 7, n. 2, p. 91–120, 1990.

REYNOLDS, P. F. A. A shared resource algorithm for distributed simulation. *Proceedings of the 9th Annual Symposium on Computer Architecture*, v. 24, n. 11, p. 259–266, 1982.

RICHMOND, M.; HITCHENS, M. A new process migration algorithm. *ACM Sigops Operating Systems Review*, v. 31, p. 31–42, 1997.

RODRIGUES, G.; CORES, I.; RODRIGUES, G. An application level approach for proactive process migration in mpi applications. *12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, p. 400–405, 2011.

ROMAN, E. *A Survey of Checkpoint/Restart Implementations*. Berkeley, CA: Lawrence Berkeley National Laboratory, 2002.

SCHRIBER, T. J.; BRUNNER, T. D. Inside discrete-event simulation software: How it works and why it matters. *Proceedings of the 34th Winter Simulation Conference*, v. 1, p. 97–107, 2002.

SHANNON, R. E.; SADOWSKI, R. P.; PEGDEN, C. D. *Introduction to Simulation Using SIMAN*. 2. ed. Nova York: McGraw-Hill, 1990.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating System Concepts*. 8. ed. Massachusetts: John Wiley, 2009.

SKOLD, S.; RONNGRENN, R. Event sensitive state saving in time warp parallel discrete event simulation. *Proceedings of the 28th conference on Winter simulation*, p. 653–660, december 1996.

SMITH, J. A survey of process migration mechanisms. *Operating Systems Review*, 1988.

SOM, T. K.; SARGENT, R. G. A probabilistic event scheduling policy for optmistic parallel discrete event simulation. *Proceeding of the 12th Wokshop on Parallel and Distributed Simulation*, p. 56–63, 1998.

SOM, T. K.; SARGENT, R. G. Model structure and load balancing in optimistic parallel discrete event simulation. *Proceeding of 14th Workshop on Parallel and Distributed Simulation*, p. 147–154, 2000.

STALLINGS, W. *Operating systems: internals and design principles*. 6. ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2009. Web Site for Operating Systems: internals and design principles - WilliamStallings.com/OS/OS6e.html.

STEINMAN, J. S. Incremental state saving in speedes using c++. *Proceedings of the 1993 Winter Simulation Conference*, p. 687–696, December 1993.

STELLNER, G. Cocheck: checkpointing and process migration for mpi. *The 10th International on Parallel Processing Symposium, Proceedings of IPPS'96*, p. 526–531, 1996.

TEO, Y. M.; NG, Y. K. Spades/java: Object-oriented parallel discrete-event simulation. *Proceedings of the 35th Annual Simulation Symposium*, p. 245–252, 2002.

VOORSLOUYS, B. L. *Utilização de Políticas de Escalonamento para a execução de simulação distribuída*. Dissertação (Mestrado) — ICMC - USP, 2006.

WANG, J.; TROPPER, C. Optimizing time warp simulation with reinforcement learning techniques. *Proceedings of the 2007 Winter Simulation Conference*, p. 577–584, 2007.

WANG, J.; TROPPER, C. Using genetic algorithms to limit the optimism in time warp. *Proceedings of the Winter Simulation Conference (WSC)*, p. 1180–1188, 2009.

WANG, Y. M. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, April 1997.

ZENG, Y.; CAI, W.; TURNER, S. J. Batch based cancellation: A rollback optimal cancellation scheme in time warp simulations. *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, p. 78–86, 2004.

ZIMMERMAN, A.; KNOKE, M.; HOMMEL, G. Complete event ordering for time warp simulation of stochastic discrete event systems. *4th Symposium on Design, Analysis, and Simulation of Distributed Systems (DASD)*, n. 4, 2006.

APÊNDICE A – Exemplos de Funções do MPI

Este anexo apresenta algumas funções do MPI que poderão ser uttilizadas na implementação dos mecanismos de migração a serem projetados.

A.1 Comunicação Ponto a Ponto

- `MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`

  Parâmetros de Entrada:

  1.buf: endereço inicial do *buffer* de dados a ser enviado;

  2.count: inteiro não negativo indicando o número de dados a ser enviado;

  3.datatype: tipo de estrutura de dados a ser enviada;

  4.dest: *rank* do destinatário;

  5.tag: rótulo de identificação da mensagem.

  Esta função não apresenta parâmetros de saída.

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`

  Parâmetros de Entrada:

  1.count: número máximo de elementos a receber;

2.datatype: tipo de estrutura de dados a receber;

3.source: *rank* identificador do emissor;

4.tag: rótulo da mensagem;

5.comm: Comunicador.

Parâmetros de Saída:

1.buf: endereço inicial do *buffer* para receber os dados;

2.status: estado do recebimento da mensagem.

- `int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)`

Parâmetros de Entrada:

1.source: *rank* do processo emissor ou MPI_ANY_SOURCE para qualquer emissor;

2.tag: rótulo da mensagem ou MPI_ANY_TAG para qualquer rótulo;

3.comm: comunicador da mensagem.

Parâmetros de Saída:

1.flag: indicação de recebimento de mensagem. Valor 0 indica que nenhuma foi recebida. Valor 1 indica que foi recebida uma mensagem;

2.status: estado do recebimento da mensagem.

A.2 Criação de Processos

- `int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])`

Parâmetros de Entrada:

1.command: nome do programa a ser executado no novo processo;

2.argv: argumentos de comando para o processo criado;

3.maxprocs: número de processos a serem criados;

4.info: um par chave-valor dizendo onde e como iniciar os processos;

5.root: *rank* do processo em que os argumentos anteriores são examinados;

6.comm: *intracommunicator* contendo o grupo do processo que cria os processos.

Parâmetro de Saída:

1.intercomm: *intercomunicator* entre o grupo do processo criador e o grupo dos processos criados;

2.array_of_errcodes: um código de erro para cada processo.

- `int MPI_Comm_spawn_multiple(int count, char *array_of_commands[], char **array_of_argv[], int array_of_maxprocs[], MPI_Info array_of_info[], int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])`

Parâmetros de entrada:

1.count: número de diferentes programas criados;

2.array_of_commands: programas (comandos) a serem executados;

3.array_of_argv: argumentos para os programas criados;

4.array_of_maxprocs: número máximo de processos a serem criados por comando;

5.array_of_info: informações de onde e como criar processos de cada comando;

6.root: *rank* do processo em que os argumentos anteriores são examinados;

7.comm: *intracommunicator* contendo o grupo do processo que cria os processos.

Parâmetros de saída:

1.intercomm: *intercomunicator* entre o grupo do processo criador e o grupo dos processos criados;

2.array_of_errcodes: um código de erro para cada processo.

A.3 Criação de Portas

- `int MPI_Open_port(MPI_Info info, char *port_name)`

Parâmetro de Entrada:

1.info: opções de como estabelecer um endereço. Não suportado na versão corrente do MPI. Pode ser usado o parâmetro pré-definido MPI_INFO_NULL para indicar a opção padrão.

Parâmetro de saída:

1.port_name: identificador da porta atribuído pelo sistema MPI.

- `int MPI_Publish_name(char *service_name, MPI_Info info, char *port_name)`

Parâmetros de entrada:

1.service_name: nome de serviço para uma porta;

2.info: opções para as funções para o nome de serviço;

3.port_name: porta na qual será atribuído o nome de serviço.

Esta função não apresenta parâmetros de saída.

- `int MPI_Comm_accept(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)`

Parâmetros de entrada:

1.port_name: identificador de porta;

2.info: opções para a conexão. Não suportado nesta versão do MPI;

3.root: *rank* do processo raiz no comunicador comm;

4.comm: *intracommunicator* sobre o qual a chamada é coletiva.

Parâmetro de saída:

1.port_name: identificador associado com o nome de serviço.

- `int MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name)`

Parâmetros de entrada:

1.service_name: um nome de serviço conhecido;

2.info: opções para o nome de serviço.

Parâmetro de saída:

1.port_name: identificador associado com o nome de serviço.

- `int MPI_Comm_connect(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)`

Parâmetros de entrada:

1.port_name: identificador MPI da porta a ser localizada para conexão;

2.info: opções para a conexão. Não suportado na versão corrente do MPI;

3.root: *rank* raiz no comunicador comm.

Parâmetro de saída:

1.newcomm: *intercommunicator* retornado após a conexão.