UM SIMULADOR DISTRIBUÍDO BASEADO NO PARADIGMA ESPAÇO-TEMPORAL

Nahri Balesdent Moreano

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS.

Aprovada por:

Prof. Valmir Carneiro Barbosa, Ph.D.
(Presidente)

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Adriano Joaquim de Oliveira Cruz, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
FEVEREIRO DE 1994

MOREANO, NAHRI BALESDENT

Um Simulador Distribuído Baseado no Paradigma Espaço-Temporal

[Rio de Janeiro] 1994.

ix, 112 p., 29.7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas, 1994)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Simulação 2. Sistemas Distribuídos

I. COPPE/UFRJ II. Título (série)

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

Um Simulador Distribuído Baseado no Paradigma Espaço-Temporal

Nahri Balesdent Moreano

Fevereiro de 1994

Orientador : Valmir Carneiro Barbosa

Programa : Engenharia de Sistemas e Computação

Este trabalho descreve o projeto, a implementação e uma avaliação experimental de um simulador distribuído de eventos discretos baseado no paradigma espaço-temporal de Chandy e Sherman. O ponto central do projeto do simulador foi a total separação das funções pertinentes ao simulador daquelas pertinentes à aplicação sendo simulada, permitindo assim o tratamento de uma vasta classe de problemas. A implementação foi realizada em Occam 2 sobre um hipercubo de *transputers* com oito processadores. Uma avaliação experimental foi feita sobre a simulação de colisão de partículas em duas dimensões, um problema notoriamente difícil em termos de paralelização, e que ainda não havia sido tratado pelo paradigma espaço-temporal. Os resultados indicam valores de *speedup* compatíveis com os que têm sido obtidos para este problema com outros paradigmas.

Abstract of Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A Distributed Simulator Based on the Space-Time Paradigm

Nahri Balesdent Moreano

February 1994

Thesis Supervisor : Valmir Carneiro Barbosa
Department : Programa de Engenharia de Sistemas e Computação

In this work we describe the design, the implementation, and an experimental evaluation of a dicrete-event distributed simulator based on Chandy and Sherman's space-time paradigm. The key point in the design of the simulator has been the complete separation between simulator-related functions and those pertaining to the particular application being simulated, thereby allowing the treatment of a vast class of problems. The simulator was implemented in Occam 2 on an eight-node transputer hypercube. An experimental evaluation was carried out on the simulation of colliding particles in two dimensions, a notoriously difficult problem in terms of parallel processing amenability, and which had not yet been approached via the space-time paradigm. Results indicate speedups comparable to what has been obtained for this problem under different paradigms.

Índice

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

1.1 Motivação

Em diversas áreas da ciência, sistemas físicos são representados por modelos matemáticos cuja solução analítica é desconhecida ou muito custosa. Existem ainda sistemas para os quais a realização de experimentos reais é insegura ou dispendiosa. Nestes casos, a utilização da simulação por computador é uma abordagem alternativa que viabiliza a análise de tais sistemas.

O objetivo da simulação por computador é construir um sistema lógico que reproduza o comportamento de um determinado sistema físico real, cujas características se deseja analisar. Especificamente na simulação dirigida por eventos, o sistema físico precisa ser modelado como um conjunto de subsistemas independentes que interagem durante a simulação, através da troca de eventos. Para que uma simulação reproduza o comportamento do sistema físico de forma correta, as relações de causalidade do sistema físico devem ser respeitadas, o que corresponde a manter a ordem correta de execução dos eventos.

Aplicações de áreas como engenharia, ciência da computação, economia e militar podem ser estudadas através da simulação. Por exemplo, sistemas de computação, redes de comunicação, sistemas de combates militares, sistemas de dinâmica molecular e sistemas de dinâmica de populações são alguns sistemas que são tratados por simulação.

A simulação de alguns sistemas físicos mais complexos, no entanto, consome uma quantidade excessiva de tempo de processamento em computadores seqüenciais.

O advento da tecnologia de processamento paralelo e a estrutura intrinsecamente paralela do problema de simulação — um sistema físico dividido em vários subsistemas — sugerem a utilização de computadores paralelos, com o objetivo de reduzir o tempo de processamento destas simulações.

Alguns algoritmos têm sido propostos com a intenção de permitir a elaboração de simulações em ambientes paralelos, particularmente em sistemas distribuídos. A exploração do paralelismo nas simulações é realizada através da execução concorrente dos subsistemas que compõem o sistema físico, o que é denominado de simulação paralela ou, mais precisamente para ambientes distribuídos, simulação distribuída.

No entanto, em um sistema distribuído, a ausência de uma base de tempo global e o controle distribuído da simulação trazem dificuldades no que se refere ao respeito às relações de causalidade do sistema físico. Em tais ambientes, não é tão simples para o algoritmo de simulação manter essas relações, como o é em um ambiente seqüencial.

Portanto, é responsabilidade do algoritmo de simulação distribuída assegurar o andamento correto da simulação, garantindo que as relações de causalidade não sejam violadas quando a simulação é realizada em um computador paralelo distribuído, e permitir assim que seja explorado o alto grau de paralelismo apresentado pelas simulações.

Quanto à forma como mantêm o respeito às relações causais do sistema físico, os algoritmos de simulação distribuída podem ser classificados em duas categorias. Os métodos conservadores evitam completamente a possibilidade de ocorrer algum erro de causalidade, isto é, de algum evento ser executado fora de ordem, e assim garantem que a simulação sempre progrida na direção correta. Para isso, baseiam-se em alguma estratégia para determinar o momento correto de processar um evento. Ao contrário destes métodos, os mecanismos otimistas não evitam a ocorrência de erros de causalidade, pois permitem que a simulação prossiga sem ter certeza de que os eventos estão sendo executados na ordem correta. Quando um erro de causalidade ocorre, o mecanismo realiza a detecção e a recuperação do erro, cancelando todos os efeitos da computação prematura, para dar prosseguimento à simulação a partir de um ponto anterior ao erro.

1.2 Objetivos do Trabalho

Este trabalho tem como finalidade apresentar o projeto, a implementação e uma avaliação experimental do desempenho de um simulador distribuído otimista genérico, baseado no paradigma espaço-temporal proposto por K. M. Chandy e R. Sherman em [CS89a].

O ponto central do projeto do simulador foi a total separação das funções pertinentes ao simulador daquelas pertinentes à aplicação sendo simulada. O algoritmo de simulação distribuída otimista e as políticas e os mecanismos que ele utiliza são implementados de forma encapsulada no simulador, enquanto que a aplicação é responsável exclusivamente pela modelagem do sistema físico real. Tal separação permite ao simulador tratar uma vasta classe de sistemas físicos reais.

Uma avaliação experimental do desempenho do simulador distribuído é realizada empregando-se um sistema de colisão de partículas. O objetivo desta avaliação é, através da variação de alguns parâmetros e da análise de diversas medidas, concluir para que classe de aplicações o simulador pode oferecer um bom desempenho. Os resultados obtidos com esta análise indicam que o simulador se comporta muito bem se o volume de processamento que ele possui para realizar excede o volume de comunicação.

Neste trabalho são também apresentados os conceitos básicos da área de simulação e descritos alguns dos principais algoritmos de simulação distribuída conservadora e otimista existentes na literatura, com a intenção de situar o paradigma espaço-temporal e o simulador distribuído desenvolvido no contexto do problema de simulação.

1.3 Organização do Texto

Este texto é composto de seis capítulos. O primeiro apresenta esta introdução com o objetivo fornecer uma visão geral sobre o trabalho. O segundo capítulo introduz os principais conceitos da área de simulação, descreve como o processamento paralelo pode ser empregado para o desenvolvimento de algoritmos de simulação paralela, e discute as principais dificuldades na criação destes algoritmos para ambientes distribuídos. São descritos também alguns dos principais algoritmos para simulação

distribuída de eventos discretos.

No capítulo 3 é descrito o paradigma de simulação distribuída espaço-temporal, no qual se baseia o simulador desenvolvido. O quarto capítulo apresenta o projeto e a implementação do simulador, assim como as políticas e os mecanismos que ele implementa para a realização da simulação distribuída otimista. O capítulo ainda descreve o ambiente de desenvolvimento do simulador, composto pelo computador paralelo, a linguagem de programação e o modelo de comunicação utilizados. Também é apresentado o modelo de desenvolvimento de aplicações para este simulador.

O quinto capítulo apresenta uma avaliação experimental de desempenho do simulador. O sistema de colisão de partículas empregado como aplicação para esta avaliação é descrito, assim como a forma e as dificuldades de exploração de paralelismo neste sistema. Os resultados obtidos desta avaliação são apresentados e analisados. Por último, no capítulo 6 são apresentadas as conclusões assim como os possíveis trabalhos futuros.

Capítulo 2

Simulação Distribuída

O objetivo deste capítulo é introduzir os principais conceitos da área de simulação, que servem de base para todo o restante do trabalho. São descritos também alguns dos principais algoritmos para simulação existentes na literatura.

Este capítulo está organizado em cinco seções. Na primeira são apresentados os conceitos básicos de simulação, assim como o algoritmo para simulação seqüencial. A seção 2.2 descreve como o processamento paralelo pode ser empregado para o desenvolvimento de algoritmos de simulação paralela. Algumas das principais dificuldades na criação destes algoritmos são discutidas, com ênfase nos algoritmos de simulação paralela para ambientes distribuídos. Nas seções 2.3 e 2.4 são descritos os principais métodos de simulação distribuída conservadora e otimista, respectivamente. Por último, na quinta seção é descrita uma forma diferente de explorar paralelismo nas simulações, que é a decomposição temporal.

2.1 Simulação de Sistemas Físicos

Conforme apresentado na seção 1.1, em diversas áreas da ciência, sistemas reais são estudados e analisados através da simulação por computador. Esta seção busca então apresentar os conceitos fundamentais da área de simulação.

O sistema real que se deseja simular é chamado de *sistema físico*. Neste trabalho são considerados sistemas físicos que possam ser modelados como um conjunto de subsistemas denominados *processos físicos* (PFs), que comportam-se de forma independente, exceto quando interagem com outros PFs do sistema. Cada PF

representa um componente do sistema real a ser simulado.

As interações entre os processos físicos são modeladas por *eventos* trocados entre eles, que ocorrem no processo físico ao qual o evento se destina. A cada evento e está associado um rótulo de tempo t que corresponde ao instante de ocorrência da interação referente a e no sistema físico real.

O sistema físico é simulado através da construção de um *sistema lógico*, que consiste de um conjunto de *processos lógicos* (PLs). Cada PL do sistema lógico corresponde a um PF do sistema físico. O *estado* de um processo físico em um determinado instante corresponde a uma porção do estado do sistema físico. Assim, cada processo lógico possui *variáveis de estado* que armazenam o estado local do PF correspondente, no instante de tempo que o PL se encontra. As interações entre os PFs, isto é, os eventos trocados entre eles também são reproduzidos no sistema lógico. Assim, um algoritmo de simulação tem como objetivo reproduzir o comportamento de um sistema físico durante um intervalo de tempo, isto é, determinar os estados de todos os PFs e os eventos trocados entre eles durante este intervalo.

Um importante conceito na caracterização dos métodos de simulação é o *tempo de simulação*, que é definido como uma abstração do tempo real, no sentido em que um estado abstrato do sistema lógico em um dado instante de tempo de simulação corresponde a um estado do sistema físico no instante de tempo real correspondente. O rótulo de tempo t associado a cada evento e é o seu tempo de simulação, que corresponde ao instante de tempo real em que e ocorre no sistema físico.

Os métodos de simulação podem ser classificados em duas categorias quanto à forma como o tempo de simulação progride : *dirigidos por tempo* e *dirigidos por eventos*. Em ambas as categorias, a evolução da simulação é ditada pela progressão do tempo de simulação. Na simulação contínua dirigida por tempo, o tempo de simulação é avançado em fatias pequenas de tamanho constante, iterativamente. Esta quantia constante define um intervalo de simulação. Todas as mudanças de estado do sistema em um determinado intervalo devem ser simuladas antes de avançar o tempo de simulação para o próximo intervalo. Este método é aplicado a sistemas onde as mudanças de estado ocorrem continuamente no tempo.

Na simulação discreta dirigida por eventos, também chamada de simulação de eventos discretos, o tempo de simulação avança em saltos não constantes, para instantes de tempo discretos da simulação, nos quais há uma mudança de estado

do sistema, correspondendo à ocorrência de um evento. Entre a ocorrência de dois eventos consecutivos nada de importante acontece no sistema. Este método é aplicado a sistemas que são muito irregulares em seu comportamento temporal. Os métodos de simulação estudados nesse trabalho pertencem a esta última categoria.

Por permitirem uma forma de tratamento diferente, cada uma destas duas classes de métodos de simulação possui suas aplicações específicas, isto é, é indicada para a simulação de sistemas físicos com determinadas características. Um exemplo de aplicação para a simulação dirigida por tempo são os sistemas meteorológicos. Exemplos de aplicação para a simulação de eventos discretos são redes de comunicação, sistemas de computação e dinâmica molecular.

Os algoritmos de simulação descritos nesse trabalho aplicam-se a sistemas físicos que satisfaçam às propriedades de *realizabilidade* e *preditibilidade* ([M86]) descritas a seguir :

> Realizabilidade : Um evento enviado por um processo físico no tempo de simulação t é uma função do seu estado inicial, de t e dos eventos que ele recebeu até o instante t, inclusive.
>
> Preditibilidade : Para todo ciclo de n processos físicos $PF_0, \ldots, PF_{n-1}$, onde PF_i recebe eventos de PF_{i-1} e envia eventos para PF_{i+1} (dado que a aritmética dos subscritos é módulo n), existe um número real ϵ, $\epsilon > 0$, e pelo menos um PF no ciclo, tal que os eventos enviados pelo PF ao longo do ciclo até o instante $t + \epsilon$, podem ser determinados dados os eventos que este PF recebeu até o instante t, inclusive.

A propriedade de realizabilidade simplesmente garante que um PF não pode prever nenhum evento que ele receberá no futuro da simulação. A propriedade de preditibilidade evita a situação de uma definição circular, onde um evento recebido por um PF é função dele mesmo. A existência do número real ϵ não implica que seu valor seja conhecido ou possa ser determinado.

A título de ilustração destas propriedades, utiliza-se como exemplo um sistema físico modelado como uma rede de filas fechada, em que os *jobs* esperam em uma fila, são atendidos pelo servidor correspondente àquela fila, passam para a fila correspondente ao próximo servidor do ciclo, e assim permanecem circulando na rede. Neste sistema, cada servidor tem um tempo de serviço mínimo diferente de zero. Os

processos físicos representam cada par fila-servidor e os eventos trocados entre eles indicam os *jobs* saindo de um servidor e entrando na fila do próximo servidor.

Como um servidor só envia um *job* para o próximo servidor após tê-lo recebido do servidor anterior e tê-lo atendido, a propriedade de realizabilidade é satisfeita. Além disso, dado que um *job* começou a ser atendido por um determinado servidor da rede no instante t, ele só será enviado para um próximo servidor em um instante maior ou igual a $t+\epsilon$, onde ϵ é o tempo de serviço mínimo (diferente de zero) daquele servidor. Assim, é possivel predizer os *jobs* que sairão deste servidor até o instante $t+\epsilon$ pelo menos, e a propriedade de preditibilidade também é satisfeita. A validade da propriedade de preditibilidade assegura que não ocorre a situação em que um *job* percorre todo um ciclo de servidores da rede em um intervalo de tempo zero, isto é, ele tem tempo de serviço zero em todos os servidores do ciclo.

Caso se deseje simular sistemas em que tal situação pode ocorrer, uma solução é representar o tempo de simulação dos eventos como um par ordenado (t,k), onde t é o tempo de simulação do evento com o significado que tem sido tratado até agora, e k é o número de servidores da rede pelos quais aquele *job* já foi atendido. Com esta representação diferente do tempo de simulação, a propriedade de preditibilidade passa a ser satisfeita, mesmo que todos os servidores de um ciclo da rede tenham tempo de serviço zero. Por simplicidade então, assume-se que todos os sistemas físicos atendem à propriedade de preditibilidade.

Um sistema físico apresenta relações de causalidade que determinam uma relação de dependência entre todos os eventos do sistema. Esta relação de dependência entre os eventos está relacionada com a ordem em que os eventos ocorrem no sistema físico. Além disso, o tempo de simulação associado a um evento e' deve ser maior ou igual ao tempo associado a qualquer evento e, se e' depende de e.

A relação de dependência entre os eventos do sistema é definida da forma a seguir ([L78]) :

> A relação "$\rightarrow$" de dependência sobre o conjunto de eventos de um sistema físico é a menor relação satisfazendo as três seguintes condições :
>
> 1. Se e e e' são eventos que ocorrem no mesmo processo físico, e e ocorre antes de e', então $e \rightarrow e'$.
> 2. Se e e e' são eventos que ocorrem em processos físcos distintos, e a ocorrência de e resulta na geração de e', então $e \rightarrow e'$.

3. Se $e \rightarrow e'$ e $e' \rightarrow e''$, então $e \rightarrow e''$.

Esta relação de dependência define uma ordenação parcial irreflexiva sobre o conjunto de eventos do sistema físico. Esta ordenação é apenas parcial pois nem todos os eventos dependem uns dos outros. Se $e \not\rightarrow e'$ e $e' \not\rightarrow e$, então e e e' são chamados eventos concorrentes. Além disso, a ordenação é irreflexiva pois requere-se que a relação de dependência não seja cíclica, isto é, um evento não pode depender dele mesmo ($e \not\rightarrow e$), o que é garantido pela propriedade de preditibilidade.

Para que uma simulação reproduza o comportamento do sistema físico de forma correta, nenhum evento pode ser processado antes que todos os eventos dos quais ele depende já tenham sido processados. Assim, a relação de dependência define a ordem em que os eventos devem ser executados na simulação, isto é, os eventos devem ser processados de forma a seguir a ordenação parcial irreflexiva definida pela relação de dependência. O termo *sincronização* refere-se à propriedade de um algoritmo de simulação que garante que os eventos são executados na ordem correta.

Uma vez definidos um sistema físico e um sistema lógico, apresenta-se então o algoritmo básico de simulação seqüencial. Usualmente um simulador seqüencial utiliza duas estruturas de dados principais :

- uma variável *relógio* que armazena uma grandeza real indicando o tempo de simulação até o qual o sistema lógico já evoluiu, isto é, denotando o quanto a simulação já progrediu. Todos os eventos com tempo de simulação t, tal que $t <$ relógio, já foram executados; e
- uma *lista de eventos* ordenada de forma não-decrescente pelo tempo de simulação dos eventos, contendo os eventos que foram escalonados, mas ainda não foram executados. Portanto, todos os eventos pertencentes à lista possuem tempo de simulação t, tal que $t \geq$ relógio.

Além disso, conforme já apresentado, cada PL possui um conjunto de variáveis de estado que respresentam o estado local do PF correspondente em um determinado instante de tempo de simulação.

Cada evento é composto dos seguintes campos : PL que o originou, PL ao qual ele se destina, texto do evento (que contém as informações que representam a interação no sistema físico correspondente a este evento) e, conforme já mencionado, o tempo de simulação associado ao evento. Um evento usualmente denota

alguma mudança no estado do sistema físico, mais precisamente no estado do PF correspondente ao PL ao qual ele se destina.

No algoritmo de simulação seqüencial, inicialmente é atribuído o valor zero ao relógio. A lista de eventos começa com um determinado número de eventos iniciais. No laço de iteração principal do simulador, o evento com o menor tempo de simulação da lista de eventos é retirado dela e repassado para o PL ao qual ele se destina, para ser executado.

Um PL ao receber um evento, simula a ocorrência daquele evento no sistema físico, podendo ocasionar uma mudança no estado daquele daquele PL, e produz zero ou mais novos eventos destinados a ele mesmo ou a outros PLs, de forma a modelar relações de causalidade do sistema físico. Os eventos gerados são para o futuro da simulação, isto é, dado que o evento executado possui tempo de simulação t, os eventos gerados possuem tempo de simulação t', $t' \geq t$. Estes eventos produzidos são inseridos na lista de eventos.

Nem todos os eventos contidos na lista de eventos serão executados, pois a execução de um evento com tempo de simulação t, além de gerar novos eventos que são incluídos na lista, pode também determinar o cancelamento de eventos com tempo de simulação t', $t' < t$, já contidos na lista. Isto ocorre quando a expectativa do comportamento futuro da simulação, representada pelos eventos contidos na lista, é frustrada pela ocorrência de um determinado evento.

Após a execução do evento, é atribuído ao relógio o valor do tempo de simulação deste evento. O algoritmo segue, iterativamente, até que um critério de terminação da simulação seja satisfeito, que no caso corresponde à lista de eventos estar vazia, isto é, a não haver mais nenhum evento a ser executado. No algoritmo 2.1 é apresentado este algoritmo de simulação seqüencial.

Conforme mencionado, o tempo de simulação associado a um evento deve ser maior ou igual ao tempo associado a qualquer outro evento do qual o primeiro dependa. Assim, para atender à exigência de que nenhum evento deve ser processado antes que todos os eventos dos quais ele depende já o tenham sido, o que corresponde a respeitar as relações de causalidade do sistema físico, na simulação seqüencial os eventos são executados em ordem não-decrescente de tempo de simulação.

Portanto, neste algoritmo seqüencial é fundamental que, a cada iteração, sempre seja selecionado para execução o evento e com o menor tempo de simulação

Início
Relógio := 0
Lista de Eventos := $\{e_i \mid e_i$ é um evento inicial$\}$
Enquanto Lista de Eventos $\neq \emptyset$ **Faça**
Retire o evento e de menor tempo de simulação t da lista de eventos
Simule o efeito da ocorrência de e no tempo de simulação t no processo lógico PL_d ao qual e se destina
Se a execução de e em PL_d causou modificações na lista de eventos **Então**
Atualize a lista de eventos, inserindo ou cancelando eventos
Fim Se
Relógio := t
Fim Enquanto
Fim

Algoritmo 2.1: Algoritmo de simulação seqüencial

da lista de eventos. Se, em uma determinada iteração, algum outro evento e' com tempo de simulação maior que o de e fosse selecionado, seria possível que a execução de e' modificasse o estado do sistema. Quando o evento e fosse processado, ele encontraria um estado alterado por e', levando à situação incorreta em que um evento futuro pode afetar um evento passado.

2.2 Simulação Distribuída de Eventos Discretos

A simulação de alguns sistemas físicos mais complexos consome uma quantidade excessiva de tempo de processamento em computadores seqüenciais. A simulação seqüencial só é viável nos casos em que a lista de eventos é limitada, em relação à velocidade de processamento do computador.

A estrutura do problema de simulação descrita anteriormente – um sistema físico dividido em vários processos físicos que interagem entre si – é intrinsecamente paralela, o que sugere a utilização de computadores paralelos, com o objetivo de reduzir o tempo de processamento das simulações.

Simulação paralela de eventos discretos refere-se à execução de um único algo-

ritmo de simulação em um computador paralelo, através da decomposição do sistema físico sendo simulado em um conjunto de processos concorrentes. Mais precisamente, este trabalho aborda algoritmos para a *simulação distribuída assíncrona de eventos discretos*, que é a simulação paralela em um computador paralelo de controle e memória distribuídos, isto é, em um sistema distribuído.

Por sistema distribuído entende-se um conjunto de nós de processamento espacialmente espalhados que não compartilham memória e que estão interconectados por canais de comunicação segundo uma determinada topologia, através dos quais se comunicam pela troca de mensagens. Assim, em processamento distribuído não há processos de controle centralizados. Assume-se que as mensagens enviadas através dos canais de comunicação têm tempos de transmissão arbitrários, porém finitos.

Uma importante característica de um sistema distribuído é a ausência de uma base de tempo global, isto é, os diversos nós de processamento não compartilham um relógio global. Cada processador possui o seu relógio local independente dos demais. Esta questão traz algumas das principais dificuldades no desenvolvimento de algoritmos para sistemas distribuídos.

Como dito anteriormente, para que uma simulação reproduza o comportamento de um sistema físico de forma correta, nenhum evento pode ser processado antes que todos eventos dos quais ele depende já tenham sido processados. Manter a ordem correta de execução dos eventos no simulador equivale a respeitar as relações de causalidade do sistema físico. Quando isto não acontece, ocorrem erros que são denominados *erros de causalidade*. Pode ser garantido que as relações de depêndencia entre os eventos serão respeitadas se a simulação seguir a *restrição de causalidade local* ([F90]), descrita a seguir :

> Uma simulação de eventos discretos obedecerá à restrição de causalidade local se, e somente se, cada processo lógico executar seus eventos na ordem não-decrescente do tempo de simulação.

Na simulação seqüencial, esta restrição de causalidade local é respeitada simplesmente através da utilização da lista de eventos ordenada pelo tempo de simulação, e de sempre ser selecionado para execução o evento de menor tempo de simulação da lista.

Porém, nem todos os eventos dependem uns dos outros, e o paralelismo se torna

vantajoso justamente nos casos em que há independência entre eventos, permitindo o processamento destes em paralelo. Para isso, na simulação distribuída os processos lógicos associados aos processos físicos que compõem o sistema físico são distribuídos pelos nós de processamento do computador distribuído e executados em paralelo, e os eventos trocados entre os PLs são projetados como mensagens enviadas entre os processadores.

Como os processos lógicos ficam espalhados pelos nós de processamento, a lista de eventos, que na simulação seqüencial era mantida de forma centralizada, é, na simulação distribuída, particionada entre os processadores, de forma que em cada processador a lista conterá os eventos destinados aos PLs alocados àquele nó de processamento. Assim, na simulação distribuída, com a ausência de uma base de tempo global e com o controle distribuído da execução, surge a dificuldade de como determinar se um evento e pode ser processado, ou seja, como determinar se todos os eventos e' dos quais o evento e depende já foram processados. Ou ainda, como determinar se um evento e depende de outro evento e', sem de fato processar este evento e'.

É responsabilidade do mecanismo de simulação distribuída assegurar que as relações de causalidade não são violadas quando o programa de simulação é executado em um computador paralelo. Para isso, mensagens de controle e de sincronização são adicionadas ao simulador, criando no entanto problemas de *overhead* de comunicação, *deadlock*, gerenciamento de espaço de armazenamento, e outros.

Embora a simulação distribuída apresente um alto grau de paralelismo, a dificuldade está na identificação da relação de depêndencia entre os eventos do sistema, que determina a ordem em que eles devem ser executados, e que é, em geral, muito complexa e dependente da aplicação e dos dados.

Os métodos de simulação distribuída dirigida por eventos podem ser classificados em duas categorias, quanto à forma como mantêm o respeito às relações de causalidade : *conservadores* e *otimistas*. Os primeiros evitam a ocorrência de erros de causalidade através da determinação, utilizando alguma estratégia, de quando um evento pode ser executado, ou seja, quando todos os eventos dos quais ele depende já o foram. Os métodos otimistas não impedem a ocorrência de um erro de causalidade e baseiam-se em um mecanismo de detecção e recuperação destes erros.

Em [F90], Fujimoto faz um levantamento detalhado sobre os algoritmos pro-

postos para simulação distribuída tanto conservadora quanto otimista. São também apresentados os desempenhos obtidos com estes algoritmos assim como uma análise crítica às estratégias conservadoras e otimistas.

Fujimoto também cita outras formas existentes de explorar paralelismo em simulação. Uma delas propõe a utilização de unidades funcionais dedicadas para implementar funções específicas da simulação seqüencial, como por exemplo manipulação da lista de eventos, geração de números aleatórios, etc, que seriam executadas em paralelo com a simulação.

Uma outra alternativa propõe a execução de replicações independentes de simulação seqüencial em diferentes processadores. Esta alternativa é útil se a simulação é de natureza estocástica e deseja-se realizar várias execuções do mesmo programa, com diferentes condições iniciais.

2.3 Métodos Conservadores

Os algoritmos conservadores evitam completamente a possibilidade de ocorrer algum erro de causalidade. Para isso, baseiam-se em alguma estratégia para determinar quando é seguro processar um evento, isto é, determinar quando todos eventos que podiam afetar aquele evento já foram processados. Mais precisamente, se um processo lógico possuir um evento e ainda não processado, com tempo de simulação t (e não possuir nenhum outro evento com tempo de simulação menor que t), e este processo puder determinar que é impossível que ele mais tarde receba um outro evento com tempo de simulação menor que t, então este processo poderá seguramente processar e. A escolha de apenas eventos seguros para execução garante que a restrição de causalidade local é respeitada, ou seja, nenhuma relação de causalidade do sistema físico é violada.

Em alguns métodos conservadores, se um processo lógico não possuir nenhum evento seguro para executar, então ele se bloqueará, até poder determinar que algum evento destinado a ele é seguro. Estes bloqueios podem levar a simulação a situações de *deadlock* se as precauções adequadas não são tomadas. Alguns destes métodos, que são denominados bloqueantes, resolvem este problema evitando que o *deadlock* ocorra, através da utilização de um mecanismo de prevenção. Outros, se baseiam em alguma estratégia para a detecção da ocorrência do *deadlock* e a sua recuperação.

Chandy e Misra [CM79] e, de forma independente, Bryant [B77] desenvolveram alguns dos primeiros algoritmos de simulação distribuída dirigida por eventos que são conservadores e baseiam-se em um mecanismo de prevenção de *deadlock*. Estes métodos requerem que sejam especificados estaticamente quais processos lógicos podem se comunicar com quais outros. Pelo modelo, a seqüência de eventos enviados de um PL_i para outro PL_j está em ordem não-decrescente de tempo de simulação, isto é, é exigido que os meios de comunicação preservem a ordem de envio dos eventos. Isto garante que, se PL_j recebeu um evento e com tempo de simulação t de PL_i, então t é um limite inferior do tempo de simulação de qualquer evento subseqüente que PL_j venha a receber de PL_i, ou seja, PL_j tem certeza que recebeu de PL_i todos os eventos com tempo de simulação menor que t.

Os eventos recebidos por um PL_j são armazenados em filas, estando cada fila associada a cada PL_i que pode enviar eventos para PL_j. Estas filas mantêm a ordem de recebimento dos eventos, e por conseguinte a ordenação não-decrescente dos eventos pelo tempo de simulação. A cada canal de entrada em um determinado PL_j (isto é, a cada PL_i que pode enviar eventos para PL_j) está associado um relógio, cujo valor é o tempo de simulação do evento na frente da fila correspondente a PL_i, ou, caso esta fila esteja vazia, o tempo de simulação do último evento recebido oriundo de PL_i.

O algoritmo repetidamente seleciona o canal de entrada de algum PL cujo relógio possui menor valor dentre todos os canais de entrada de todos os PLs alocados a um mesmo nó de processamento, e, se existe algum evento na fila correspondente ao canal escolhido, executa-o. Se a fila selecionada está vazia, então o processo lógico bloqueia-se, à espera da chegada de eventos que o retirem deste estado. Este protocolo garante que cada processo lógico somente processa eventos na ordem não-decrescente de tempo de simulação, assegurando assim que a restrição de causalidade local seja respeitada.

Se houver no sistema um ciclo de canais, tal que o valor de seus relógios sejam mínimos para os PLs correspondentes, e tal que suas filas associadas estejam vazias, então todos os PLs do ciclo bloqueiam-se e tem-se uma situação de *deadlock*.

Para evitar estas situações de *deadlock*, o algoritmo utiliza mensagens nulas que não correspondem à interações entre os PFs no sistema físico, mas são incluídas para efeito de sincronização. Estas mensagens nulas possuem também rótulos de tempo e têm o seguinte significado : uma mensagem nula com tempo t enviada de PL_i para

PL_j é uma garantia de que PL_i não enviará mais para PL_j eventos com tempo de simulação menor que t.

O rótulo de tempo de uma mensagem nula enviada por um processo lógico é determinado utilizando o valor do relógio de cada canal de entrada deste PL. O rélogio $relógio_{mín}$ com o menor tempo dentre todos é um limite inferior do tempo de simulação do próximo evento a ser executado por este PL. Além disso é utilizado também algum conhecimento específico sobre o sistema físico que forneça o incremento mínimo no tempo de simulação de qualquer evento que seja executado por aquele PL, isto é, deseja-se conhecer o número ϵ, tal que a execução por este PL de um evento com tempo de simulação t produz eventos com tempo de simulação t', $t' \geq t + \epsilon$. Aliando estas duas informações, pode-se determinar um limite inferior do tempo de simulação do próximo evento a ser enviado por este PL que é $relógio_{mín} + \epsilon$. Este limite inferior é o rótulo de tempo da mensagem nula enviada por cada canal de saída do PL.

Sempre que um processo lógico termina de executar um evento, ele envia uma mensagem nula para cada um dos demais PLs com os quais ele pode comunicar-se, indicando o limite inferior determinado. Um PL, ao receber uma mensagem nula, pode então determinar novos limites inferiores do tempo de simulação do próximo evento a ser enviado por cada canal seu de saída, e enviar estas informações para os demais PLs com os quais ele se comunica, e assim por diante.

Em [CM79], os autores mostram que este mecanismo evita a ocorrência de *deadlock* para a simulação de sistemas físicos que satisfaçam a propriedade de preditibilidade, isto é, desde que não exista um ciclo de PLs tal que a soma dos incrementos mínimos ϵ destes PLs seja zero. Como para determinar o incremento mínimo ϵ de um processo lógico é necessário conhecimento específico sobre o comportamento da aplicação, é tarefa do programador da aplicação determinar o rótulo de tempo associado às mensagens nulas.

Se a quantidade de mensagens nulas em relação ao número total de mensagens da simulação é grande, há uma degradação no desempenho deste algoritmo. Assim, variações deste mecanismo de mensagens nulas foram propostas com o objetivo de diminuir o número destas mensagens que trafegam pelo sistema. Em [M86], Misra propõe que o envio das mensagens nulas seja feito sob demanda e não após a execução de cada evento. Sempre que um PL_j vai ficar bloqueado, pois o seu relógio com menor tempo corresponde a um canal de entrada cuja fila de eventos associada está

vazia, PL_j envia um pedido para PL_i, onde PL_i é o processo lógico que envia eventos para PL_j através do referido canal. Neste pedido, PL_j requisita uma nova mensagem, que pode ser nula ou um evento. Ao receber esta nova mensagem de PL_i, o processo PL_j retoma a execução.

Chandy e Misra ([CM81]) também desenvolveram um algoritmo conservador similar ao já descrito, exceto que não são utilizadas mensagens nulas. O mecanismo de determinação dos eventos seguros a serem processados é o mesmo, porém é permitido que ocorram *deadlocks*. O algoritmo implementa mecanismos para a detecção e recuperação do *deadlock*. A recuperação do *deadlock* é realizada determinando-se o evento ainda não processado com menor tempo de simulação de todo o sistema. Este evento, que é obviamente seguro, é então executado, quebrando a situação de *deadlock*.

Vários outros algoritmos conservadores para simulação paralela foram propostos. Algumas das principais idéias utilizadas por estes algoritmos são descritas no restante desta seção.

Um recurso utilizado nos algoritmos conservadores é o *lookahead*, que refere-se à capacidade de previsão, para um determinado sistema físico, do que acontecerá, ou mais importante, do que não acontecerá no futuro da simulação [F88]. *Lookahead* é definido da forma a seguir :

> Se um PL está no instante de simulação t, isto é, este PL já recebeu e executou todos os eventos com tempo de simulação menor ou igual a t e pode predizer com absoluta certeza todos os eventos que ele gerará com tempo de simulação t', $t' \leq t + L$, então este PL possui *lookahead* L.

Um processo lógico com *lookahead* L, ao receber um evento com tempo de simulação t, pode garantir que nenhum outro evento, a não ser aqueles que ele pode predizer, será gerado com tempo de simulação t', $t' \leq t + L$. Esta informação pode possibilitar aos outros PLs da simulação determinar que alguns dos seus eventos ainda não processados já são seguros para processar. Assim, o uso de *lookahead* pode melhorar o desempenho dos algoritmos conservadores, porém requer muito conhecimento específico sobre o comportamento do sistema físico.

Em [L89] encontra-se a descrição de um outro recurso aplicado a algoritmos conservadores, que é o mecanismo de janelas móveis de tempo de simulação. Este

mecanismo tem como objetivo reduzir o volume de processamento necessário para determinar se um evento é seguro para execução. Para isso, o mecanismo emprega uma estratégia para diminuir o conjunto de PLs cujas listas de eventos deve-se examinar a fim de determinar se o próximo evento de um determinado PL é seguro para processar. A largura w da janela é escolhida estaticamente. A extremidade inferior da janela em cada momento da simulação é o tempo de simulação do evento com menor tempo dentre todos os eventos ainda não processados do sistema. Somente os eventos ainda não processados cujos tempos de simulação se encontram dentro da janela podem ser eventos seguros. Assim, dois eventos e_1 e e_2 com tempos de simulação t_1 e t_2, respectivamente, são executados em paralelo somente se $|t_2 - t_1| \leq w$.

Uma importante questão é como determinar a largura da janela temporal. Se a janela for muito estreita, poucos eventos poderão ser executados em paralelo. Por outro lado, se a janela for muito larga, o algoritmo de simulação se comportará da mesma forma que faria se não utilizasse nenhuma janela de tempo, de forma que o *overhead* acarretado pelo gerenciamento do mecanismo da janela não se justificaria. A escolha da largura da janela requer informação dependente do sistema físico pois utiliza o incremento mínimo ϵ no tempo de simulação com a execução de um evento, já mencionado nesta subseção.

Chandy e Sherman apresentaram um algoritmo conservador para simulação distribuída [CS89b], que utiliza o conceito de conhecimento condicional para determinar quando os eventos são seguros para processar. Neste método, eventos que ocorrem apenas se alguma determinada condição é satisfeita são chamados de eventos condicionais. Eventos que sabe-se que ocorrem incondicionalmente são chamados de eventos definitivos.

Conforme visto, na simulação seqüencial os eventos na lista de eventos são condicionais, e o fato de um evento condicional possuir o menor tempo de simulação dentre todos os eventos ainda não processados do sistema, é suficiente para convertê-lo em um evento definitivo.

O algoritmo de eventos condicionais baseia-se no fato de que é sempre seguro executar eventos definitivos, e como o mesmo não é verdade para eventos condicionais, propõe um mecanismo para transformar o maior número possível de eventos condicionais em definitivos. Para isso, cada PL recebe mensagens de controle de todos os demais PLs do sistema com informações sobre o evento condicional de me-

nor tempo de simulação de cada um deles, e com base nestas informações determina se o seu evento condicional de menor tempo de simulação pode ser convertido em definitivo.

Alguns eventos são definitivos por natureza, isto é, de acordo com características específicas do sistema físico pode-se garantir que alguns eventos são incondicionais, e portanto podem ser executados independentemente do que ocorre nos outros PLs. Para isso, quando um processo lógico envia um evento, ele informa se o evento é condicional ou definitivo.

Este método também requer que sejam especificados estaticamente quais processos lógicos podem se comunicar com quais outros. Mas de maneira diferente dos algoritmos de Chandy e Misra descritos anteriormente, neste algoritmo a troca de mensagens de controle para fins de sincronização não é restrita entre apenas os processos lógicos que trocam eventos durante a simulação. Quaisquer dois PLs do sistema podem se comunicar com o objetivo de determinar que eventos são seguros, isto é, de converter eventos condicionais em definitivos, independente deles trocarem eventos na simulação ou não. O uso de eventos condicionais garante a ausência de *deadlocks*, tornando desnecessária a utilização de mensagens nulas. Este método também exige que os meios de comunicação preservem a ordem de envio dos eventos, pois pelo modelo, a seqüência de eventos enviados de um PL_i para outro PL_j está em ordem não-decrescente de tempo de simulação. Com isso garante-se que, se PL_j recebeu um evento e com tempo de simulação t de PL_i, PL_j tem certeza que recebeu de PL_i todos os eventos com tempo de simulação menor que t.

A principal desvantagem dos algoritmos conservadores de simulação distribuída é que eles não exploram completamente o paralelismo que pode ser extraído do sistema físico sendo simulado. Para obter um bom desempenho, a maior parte dos algoritmos conservadores requer que o sistema físico apresente características favoráveis, como por exemplo grandes incrementos mínimos de tempo de simulação ou um bom *lookahead*. Para isso, é necessário que muito conhecimento sobre o comportamento do sistema físico seja fornecido, Além disso, grande parte destes métodos exige que seja especificada estaticamente a configuração da rede de processos lógicos, isto é, quantos são os PLs e quais PLs podem se comunicar com quais outros. As tentativas de superar estes problemas criando processos "reserva" e definindo uma rede completamente conexa podem causar *overheads* excessivos.

2.4 Métodos Otimistas

Os algoritmos otimistas não evitam a ocorrência de erros de causalidade. Ao contrário dos algoritmos conservadores que precisam determinar se um evento é seguro para processar antes de realmente executá-lo, os métodos otimistas permitem que a simulação prossiga, sem ter certeza se os eventos processados são seguros ou não. Desta forma é possível que erros de causalidade ocorram, e quando isso acontece, o algoritmo realiza a detecção do erro e executa um mecanismo para recuperá-lo. Justamente por não precisarem ter certeza de que os eventos são seguros para processá-los, os métodos otimistas não envolvem mecanismos de bloqueio de processos lógicos.

O mecanismo de *time warp* proposto por Jefferson em [J85,J87] é o método otimista mais discutido na literatura, e baseia-se no paradigma de tempo virtual, que possui o mesmo significado de tempo de simulação. Este método será descrito a seguir, assim como algumas das diversas otimizações e variações propostas sobre o método original. Neste algoritmo, qualquer processo lógico pode trocar eventos com quaisquer outros PLs do sistema, não sendo necessário portanto especificar estaticamente quais PLs podem se comunicar. Além disso, também não é exigido que os canais de comunicação preservem a ordem de envio das mensagens.

Cada processo lógico do sistema possui associado a ele um conjunto de estruturas de dados composto por :

- um relógio local que indica o tempo de simulação do evento que está sendo executado por este PL ou o tempo de simulação do último evento executado por ele;
- variáveis de estado que armazenam o estado local atual do PL;
- uma fila de estados contendo cópias gravadas dos estados do PL em instantes de simulação recentes, ordenados de forma não-decrescente pelo tempo de simulação;
- uma fila de eventos de entrada contendo os eventos recebidos recentemente pelo PL, ordenados de forma não-decrescente pelo tempo de simulação; e
- uma fila de eventos de saída, contendo cópias dos eventos enviados recentemente pelo PL, ordenados de forma não-decrescente pelo tempo de simulação.

Um processo lógico PL_i segue iterativamente selecionando o evento ainda não processado de menor tempo de simulação da sua fila de eventos de entrada, atribuindo ao seu relógio o tempo de simulação deste evento, executando-o e gerando eventos destinados a ele mesmo ou a outros PLs. Estes eventos são enviados para os PLs aos quais eles se destinam e uma cópia de cada um dos eventos enviados é inserida na fila de eventos de saída de PL_i. Sempre que um PL recebe um evento, ele é inserido na sua fila de eventos de entrada.

Um erro de causalidade é detectado sempre que um processo lógico PL_i recebe um evento e com tempo de simulação t menor que o tempo do seu relógio local, o que significa que PL_i processou um ou alguns eventos prematuramente, isto é, desrespeitando a restrição de causalidade local. Para recuperar o erro, o algoritmo de *time warp* cancela todos os efeitos da computação incorreta, isto é, os efeitos de todos os eventos e' que foram processados prematuramente por PL_i. Estes eventos possuem tempo de simulação t', $t' > t$. Com isto, o algoritmo faz com que PL_i retorne para um instante de simulação anterior ao erro, no qual ele possa receber o evento e sem violar a restrição de causalidade local. Este mecanismo de recuperação é denominado *rollback* e o evento que o causa — no exemplo, o evento e com tempo de simulação t — é chamado de *straggler*.

A execução prematura de um evento e' por um processo lógico PL_i pode causar duas conseqüências que precisam ser canceladas quando o algoritmo conclui que ocorreu um erro de causalidade : o processamento de e' pode modificar o estado de PL_i e pode enviar um ou mais eventos para outros PLs do sistema.

Para ser possível cancelar uma alteração no estado de um PL, o algoritmo de *time warp* grava o estado local do PL periodicamente. Estes estados gravados são mantidos na lista de estados do PL. Quando é detectado um erro de causalidade, por PL_i haver recebido um evento e com tempo de simulação t menor que o valor de seu relógio local, o mecanismo de *rollback* busca na fila de estados o último estado gravado de PL_i com tempo de simulação menor que t e restaura-o. Os estados da lista de estados gravados com tempo de simulação maior que t podem ser descartados. O valor do relógio local de PL_i também é corrigido, recebendo o tempo de simulação t do *straggler* e.

Para cancelar o envio dos eventos gerados pela execução prematura de um evento e' por PL_i, o mecanismo de *rollback* envia, para cada evento gerado e'' destinado a um PL_j, uma cópia negativa de e'' destinada ao mesmo PL_j. Esta

cópia negativa, chamada de anti-mensagem e com o mesmo tempo de simulação que o evento e'' original, tem a função de, quando for recebida por PL_j, cancelar e''. Mensagens que correspondem aos eventos do sistema físico sendo simulado são chamadas de mensagens positivas. Para saber quais foram os eventos e'' enviados o mecanismo utiliza a lista de eventos de saída de PL_i, onde é mantida uma cópia de cada um dos eventos enviados por PL_i. Após o envio das anti-mensagens, os eventos cancelados podem ser descartados da lista de eventos de saída.

Após estes procedimentos, localmente em PL_i o mecanismo de *rollback* está completo e PL_i está em um ponto da simulação em que pode receber o *straggler* e corretamente e prosseguir executando os eventos na ordem não-decrescente de tempo de simulação novamente. Os eventos que haviam sido executados prematuramente são reprocessados, agora na ordem correta.

Quando um processo lógico PL_j recebe uma anti-mensagem correspondente a um evento e'' a ser cancelado, a anti-mensagem é tratada de forma semelhante a um evento. A lista de eventos de entrada de PL_j é percorrida, para a inserção da anti-mensagem. Como o tempo de simulação da anti-mensagem é o mesmo de e'', se e'' ainda não foi executado por PL_j, então a anti-mensagem não chegou atrasada a PL_j e tem o efeito de apenas cancelar e'', sem alterar o prosseguimento da execução de PL_j. Se, quando PL_j recebe a anti-mensagem, o evento e'' já havia sido executado, ele pode ter causado alterações no estado de PL_j e o envio de eventos para um outro grupo de PLs. Como a anti-mensagem foi recebida atrasada, isto é, seu tempo de simulação é menor que o relógio de PL_j, então ela causa *rollback* em PL_j também, para desfazer o efeito de ter sido processado um evento que vai ser cancelado. Pode ainda ocorrer uma terceira situação, na qual PL_j recebe a anti-mensagem antes de receber o evento e'' correspondente, pois o algoritmo não assume que os canais de comunicação preservam a ordem de envio das mensagens. Neste caso, a anti-mensagem é inserida na fila de eventos de entrada, como se fosse um evento, e causará o cancelamento de e'' quando ele for recebido.

Repetindo recursivamente este procedimento, isto é, propagando o *rollback* para outros PLs se necessário, todos os efeitos da computação incorreta serão cancelados. Segundo Jefferson, este protocolo de anti-mensagens é extremamente robusto e garante que a simulação por *time warp* sempre progride. Não há possibilidade de ocorrer *deadlock*, simplesmente porque não há bloqueio de processos lógicos. Também não há possibilidade de que a propagação do *rollback* se estenda indefi-

nidamente para o passado da simulação. No pior caso, todos os PLs do sistema fazem *rollback* para o mesmo tempo de simulação para o qual o PL onde o *rollback* se iniciou fez, isto é, para o tempo de simulação do *straggler*, e depois continuam a processar normalmente.

O *rollback* em um ambiente distribuído é custoso pois o processo lógico que recebe um *straggler* pode ter enviado vários eventos para outros PLs, causando efeitos neles e levando-os a enviar ainda mais eventos para mais outros PLs, e assim por diante. Algumas desses eventos podem inclusive estar fisicamente em trânsito e portanto fora do controle do sistema por um período de tempo arbitrário. Todos esses eventos porém, em trânsito ou não, e independentemente de quão longas sejam as cadeias causais envolvidas, devem ser efetivamente cancelados e seus efeitos, se houver, revertidos.

Conforme mencionado na seção 2.3, o evento ainda não processado de menor tempo de simulação dentre todos os eventos ainda não executados do sistema, é sempre seguro para ser executado. O tempo de simulação deste evento é denominado no algoritmo de *time warp* de tempo virtual global e corresponde ao instante até o qual a simulação já progrediu. Um evento com tempo de simulação menor que o tempo virtual global nunca será cancelado por um *rollback*. Assim, o espaço de armazenamento utilizado por tais eventos, na lista de eventos de entrada e saída e na lista de estados gravados, pode ser descartado. Os eventos e estados com tempo de simulação maior que o tempo virtual global ainda podem ser necessários, caso ocorra um *rollback*. Da mesma forma, operações que não podem ser canceladas (por exemplo, operações de entrada e saída) não podem ser realizadas até que o tempo virtual global alcance o tempo de simulação no qual a operação ocorreu. Este mecanismo de recuperação de espaço de armazenamento e consumação de operações irreversíveis é chamado de "coleta de fósseis". Vários algoritmos foram propostos para a determinação do tempo virtual global ([B90]), que é utilizado para este mecanismo e para a detecção de terminação da simulação.

Diversas otimizações foram propostas no sentido de diminuir o tempo gasto na recuperação de um erro de causalidade. No algoritmo original de *time warp*, conforme descrito, quando um PL_i sofre *rollback* para um tempo de simulação t, o mecanismo imediatamente envia anti-mensagens correspondentes aos eventos enviados, cujas cópias estão armazenadas na fila de eventos de saída, e que foram gerados pelos eventos executados prematuramente por PL_i. Após este procedimento, PL_i

prossegue sua execução, reprocessando estes eventos. Este mecanismo de cancelamento do envio dos eventos é denominado cancelamento agressivo. Basicamente, as otimizações propostas visam refazer a computação desfeita, isto é, refazer a execução dos eventos processados prematuramente, de uma forma menos custosa do que simplesmente reprocessá-los por completo.

Podem ocorrer situações em que o recebimento do *straggler* não altere suficientemente a execução dos eventos processados prematuramente, que foi desfeita em conseqüência deste recebimento, a ponto de modificar os eventos gerados pela execução destes eventos. Em [G88,RFBJ90], é apresentado um mecanismo para cancelar o envio dos eventos, alternativo ao cancelamento agressivo, denominado cancelamento preguiçoso. Neste mecanismo, quando um erro de causalidade é detectado, as anti-mensagens correspondentes aos eventos gerados não são enviadas imediatamente. Ao invés disso, o mecanismo espera para testar se o reprocessamento dos eventos desfeitos pelo *rollback* vai gerar os mesmos eventos. Isto é realizado comparando os eventos gerados com as cópias dos eventos já enviados, mantidas na fila de saída. Se um mesmo evento é gerado, não há necessidade de cancelar o envio inicial daquele evento, então o mecanismo não envia a anti-mensagem correspondente e também não envia o evento gerado pela segunda vez. São enviadas apenas as anti-mensagens correspondentes aos eventos que não foram gerados novamente.

O mecanismo de cancelamento preguiçoso requer um *overhead* adicional, pois algumas comparações de eventos são necessárias para determinar se são gerados os mesmos eventos enviados anteriormente. Além disso, a utilização deste mecanismo pode permitir que a computação incorreta prossiga por mais tempo do que prosseguiria se o cancelamento agressivo fosse utilizado. Por outro lado, o cancelamento preguiçoso pode permitir que a simulação seja executada em um tempo inferior ao tempo de execução do seu caminho crítico. A explicação para este fenômeno é que computações com entradas incorretas ou apenas parcialmente corretas podem mesmo assim gerar resultados corretos. Isto não é possível utilizando o cancelamento agressivo, pois as computações que sofrem *rollback* são imediatamente descartadas, mesmo se elas geraram o resultado correto. Assim, o cancelamento preguiçoso pode melhorar ou piorar o desempenho do simulador, dependendo de características da aplicação.

Outro mecanismo alternativo proposto é a reavaliação preguiçosa, também chamada de *jump forward*, que é similar ao cancelamento preguiçoso, exceto que

lida com o estado do processo lógico e não com eventos. Podem ocorrer situações em que, quando um PL recebe um *straggler*, sofre *rollback* e processa este *straggler*, o seu estado depois de executá-lo permanece o mesmo de antes de processá-lo. Se nenhum evento foi recebido, então a reexecução dos eventos desfeitos pelo *rollback* será idêntica à execução original. Portanto, não é necessário repetir a execução destes eventos desfeitos e o PL pode voltar para o instante da simulação em que se encontrava antes de receber o *straggler* e sofrer o *rollback*. Para determinar se o PL pode "pular" este trecho da simulação , o mecanismo compara o estado restaurado do PL antes dele executar o *straggler* com o estado resultante desta execução, para saber se houve alteração.

Esta estratégia de reavaliação preguiçosa pode ser eficiente nos casos em que o *straggler* é um evento de consulta, isto é, que envolve apenas leitura de dados e não acarreta nenhuma mudança no estado do PL que o executa. Por outro lado, este mecanismo requer um *overhead* para a comparação de estados do PL, que dependendo da aplicação, podem ser grandes. Além disso, a implementação desta estratégia pode complicar significantemente o código do simulador, dificultando a sua manutenção.

Em [F90], Fujimoto argumenta que as situações nas quais o cancelamento preguiçoso ou a reavaliação preguiçosa trazem ganhos de desempenho são as mesmas nas quais pode-se explorar *lookahead* através dos métodos conservadores. Só que ao contrário dos métodos conservadores que requerem que o *lookahead* seja especificado explicitamente, o mecanismo de cancelamento preguiçoso explora o *lookahead* de uma forma transparente para o programador da aplicação. Por outo lado, conforme já descrito, este mecanismo introduz um *overhead* maior para a comparação de eventos, que não existe quando o *lookahead* é explicitamente programado na aplicação.

Janelas temporais, não muito diferentes das propostas para métodos conservadores, também foram propostas para algoritmos otimistas ([SBW88]). Nestes métodos, as janelas temporais são utilizadas para evitar que as computações incorretas propaguem-se muito para o futuro da simulação. O mecanismo de janela temporal móvel usa uma janela de tempo de largura fixa w. Somente eventos com tempo de simulação no intervalo $[t, t + w]$ são elegíveis para processar, onde t é o tempo de simulação do evento ainda não processado de menor tempo de simulação do sistema. Assim, dois eventos só podem ser executados em paralelo se a diferença

entre os seus tempos de simulação for menor que a largura da janela. O principal objetivo deste mecanismo é limitar o tamanho e a freqüência dos *rollbacks*, e limitar também o espaço de armazenamento ocupado para manter informações (eventos e estados) utilizadas pelos mecanismos que mantêm o respeito à relação de causalidade local na simulação.

Uma questão neste método é como determinar a largura da janela. Como no caso conservador, se a janela for muito estreita, permitirá que pouco paralelismo seja explorado, e por outro lado, se for muito larga, permitirá que muitos erros de causalidade ocorram. Uma crítica feita a este mecanismo é que ele não distingue computações corretas de incorretas, e pode portanto, impedir desnecessariamente o progresso de computações corretas. Além disso, a estratégia utilizada pelo algoritmo de *time warp* de executar prioritariamente os eventos com menor tempo de simulação, já é uma forma de inibir o avanço excessivo de computações incorretas.

Em [MWM88] encontra-se a descrição de um algoritmo para a simulação otimista que implementa um mecanismo denominado *wolf call* cujo objetivo é cancelar rapidamente o efeito de uma computação incorreta para que ela se propague o mínimo possível, quando um erro de causalidade é detectado. Para isso, o algoritmo exige que sejam definidos estaticamente quais processos lógicos podem enviar eventos para quais outros, e precisa de informações específicas sobre o tempo real que uma mensagem leva para se propagar pelos PLs. Neste mecanismo, sempre que um *straggler* é recebido por um PL_i, mensagens de controle são enviadas através de um *broadcast* para todos os PLs que PL_i pode ter afetado com sua computação incorreta. Se, quando um PL_j recebe uma mensagem de controle, ele já havia processado eventos decorrentes da computação incorreta, então PL_j também vai fazer um *broadcast* para todos os PLs que podem ter sido afetados por ele, e assim sucessivamente. Estas mensagens de controle possuem informações sobre os eventos que devem ser cancelados e seu efeito é paralizar rapidamente a propagação da computação incorreta.

Estes *broadcasts* requerem suporte em software e hardware dedicado, para garantir que as mensagens de controle alcançem os PLs aos quais se destinam em tempos suficientemente pequenos. Além disso, é necessário o conhecimento da velocidade em tempo real da propagação dos eventos incorretos e da transmissão das mensagens de controle. A desvantagem desta estratégia é que, da mesma forma que as janelas temporais, ela também não diferencia computações corretas de incorretas.

As mensagens de controle são enviadas para todos os PL que podem ter sido afetados pela computação incorreta, mas este conjunto de PLs pode ser significantemente maior que o conjunto de PLs que foram realmente afetados.

Uma forma alternativa para cancelar o envio de eventos no método de *time warp* foi proposta por Fujimoto em [F89]. Este mecanismo, denominado cancelamento direto é aplicado a implementações do algoritmo de *time warp* para computadores paralelos de memória compartilhada. Sempre que um evento e gera um outro evento e', é criado um ponteiro de e para e', e assim os ponteiros criados formam estruturas que de certa maneira representam as relações de causalidade do sistema. Se mais adiante na simulação detectar-se que a execução do evento e foi prematura, e que todos os eventos por ele gerados devem ser cancelados (utilizando-se o cancelamento agressivo ou preguiçoso), estes ponteiros são utilizados para a identificação destes eventos e o seu cancelamento. Assim, este mecanismo busca reduzir o *overhead* associado ao cancelamento de eventos e, em conseqüência, rapidamente cancelar a computação incorreta, para evitar que ela se propague excessivamente.

Conforme visto, os métodos otimistas baseiam-se em mecanismos de *rollback* para o cancelamento de computações erradas. Portanto, a eficiência destes algoritmos está fortemente relacionada com o tempo gasto com a execução incorreta e o seu estorno através de algum mecanismo. Uma questão levantada é se a simulação apresentará um comportamento no qual a maior parte do tempo é gasta executando computações errôneas e cancelando-as, e não realmente realizando computações corretas.

Os métodos otimistas baseiam-se na crença de que a maioria dos eventos do sistema respeita um princípio de localidade temporal, ou seja, que a maior parte dos eventos enviados é recebida no futuro dos processos lógicos aos quais eles se destinam, e portanto a execução destes eventos não causa erros de causalidade. Além disso, o mecanismo de escalonamento de eventos do algoritmo de *time warp*, por exemplo, dá prioridade a eventos com menor tempo de simulação, o que inibe a propagação excessiva da computação incorreta para o futuro da simulação, pois os eventos gerados em conseqüência da computação incorreta tendem a possuir tempos de simulação maiores que os eventos corretos.

Um problema nos métodos otimistas é a necessidade de gravar os estados dos processos lógicos periodicamente. O tempo gasto para gravar estados é bastante grande e o *overhead* com estas gravações pode degradar o desempenho do simulador.

Em [FTG88] é proposta a utilização de um componente físico chamado *rollback chip* para minimizar o *overhead* com o gerenciamento de estados no algoritmo de *time warp*, implementando em *hardware* as funções de gravação e restauração de estados e recuperação de espaço de armazenamento de estados.

Os algoritmos otimistas demandam muito mais memória que os métodos conservadores e, além disso, possuem implementações muito mais complexas. Da mesma forma, a depuração destes algoritmos também é mais complexa.

A principal vantagem dos algoritmos otimistas é que eles permitem que o simulador explore o paralelismo em simulações de sistemas físicos nas quais é possível que erros de causalidade ocorram, porém na prática freqüentemente não ocorrem. Os algoritmos conservadores em geral não exploram tais situações. Além disso, utilizando-se mecanismos otimistas é possível a construção de simuladores de propósito geral, isto é, independentes de uma aplicação, que podem ser utilizados para simular diversos sistemas físicos. Isto é possivel por os mecanismos implementados pelos algoritmos otimistas não se basearem em informações específicas sobre características do sistema físico, como acontece com os métodos conservadores.

2.5 Decomposição Temporal

Todos os algoritmos de simulação distribuída descritos nas seções 2.3 e 2.4 exploram o paralelismo nas simulações da mesma forma : o sistema físico é dividido em subsistemas (processos físicos) que são simulados em paralelo. Esta estratégia é denominada decomposição espacial da aplicação. A cada processo físico é associado um processo lógico, responsável pela sua simulação ao longo de toda a duração da simulação.

No entanto, as aplicações também podem ser decompostas temporalmente. Cada processo físico pode ser dividido em fases que o representam em diferentes intervalos de tempo de simulação. Por exemplo, um determinado PF pode ser dividido em duas fases, de forma que a primeira fase o representa no intervalo $[0, t)$ e a segunda no intervalo $[t, H)$, onde H é o horizonte final da simulação. A cada fase é associado um processo lógico responsável pela simulação do processo físico ao longo do intervalo de tempo correspondente àquela fase. O paralelismo é explorado através da simulação em paralelo destas fases.

O PL associado a cada fase é responsável por processar os eventos destinados ao PF correspondente cujos tempos de simulação pertençam ao intervalo de tempo referente àquela fase. Existe uma importante conexão entre duas fases consecutivas de um PF : o estado final do PF na primeira fase deve ser igual ao estado inicial do PF na segunda fase. Para isso, sempre que a primeira fase completa a execução de todos os eventos recebidos, uma cópia do estado final do PF é enviada para a fase seguinte, onde este estado será utilizado como estado inicial.

Se, após haver enviado um estado para a fase seguinte, a primeira fase do PF receber um evento atrasado e sofrer *rollback*, provavelmente a execução desta fase será alterada. Neste caso, quando esta fase completar novamente a execução de todos os eventos recebidos, uma cópia do novo estado final do PL é enviada para a fase seguinte. Ao receber este novo estado, a fase seguinte pode já haver começado a sua execução baseando-se no primeiro estado recebido, e portanto também sofrerá *rollback*.

A estratégia de decomposição temporal da aplicação foi apresentada no paradigma espaço-temporal proposto por Chandy e Sherman em [CS89a] que será descrito no próximo capítulo. Este paradigma permite a decomposição tanto espacial quanto temporal da aplicação, e assim busca extrair um maior paralelismo da simulação. Neste sentido, este paradigma inovou em relação aos outros modelos otimistas propostos até então, como por exemplo o algoritmo de *time warp*. No entanto, este trabalho apenas introduz este novo conceito, não apresentando um projeto ou implementação nem resultados obtidos.

Em [RBJ91] é apresentada a adição do conceito de decomposição temporal ao algoritmo de *time warp*, com o objetivo de possibilitar um melhor desempenho, por extrair mais paralelismo da aplicação e por permitir que seja feito um melhor balanceamento estático de carga.

Os autores observam que o algoritmo de *time warp* na sua forma original pode fornecer um bom desempenho se há um bom balanceamento de carga, isto é, se os PLs são distribuídos pelos nós de processamento do computador paralelo de forma que cada nó possua aproximadamente o mesmo volume de trabalho. Quando isto acontece, é provável que os PLs evoluam no tempo de simulação com aproximadamente a mesma proporção e com isso os PLs tendem a não se adiantar muito com computações incorretas e a fazer menos *rollbacks*. Assim, a simulação tende a apresentar um melhor desempenho. Todavia, este método de balanceamento estático de

carga não considera a dinâmica da simulação, pois podem existir sistemas físicos que apresentem variações no volume de trabalho ao longo de sua simulação. Para tais sistemas, a distribuição estática da carga pelos processadores que parece adequada no estágio inicial da simulação pode não prover um bom balanceamento em estágios seguintes.

Em um exemplo apresentado, é considerada uma simulação com três PLs, PL_0, PL_1 e PL_2, realizada em um multiprocessador com dois nós de processamento. A simulação é composta por três estágios, cada um representando aproximadamente um terço do tempo de duração da simulação, tanto em tempo de execução real quanto em tempo de simulação. Suponha que PL_0 requer um grande volume de processamento nos primeiro e segundo estágios, porém não realiza nada no terceiro estágio. PL_1 por sua vez requer um grande volume de processamento nos primeiro e terceiro estágios, mas fica ocioso durante o segundo estágio. Por último, PL_2 requer um grande volume de processamento nos segundo e terceiro estágios e fica ocioso durante o primeiro estágio.

Não existe um bom balanceamento estático de carga para este exemplo. Dois PLs são alocados a um dos processadores e o terceiro PL é alocado ao outro processador. No processador ao qual dois PLs foram alocados, haverá um estágio em que os dois PLs exigirão um grande volume de processamento. Durante este mesmo estágio, o processador que possui apenas um PL ficará ocioso. Na verdade, o mecanismo de *time warp* permitirá que este PL continue seu processamento de forma otimista, avançando para o futuro da simulação. Porém, como o outro processador está sobrecarregado, muito provavelmente este PL sofrerá *rollback*.

Se a decomposição temporal é utilizada, um dos PLs, por exemplo PL_1, é dividido em duas fases, uma que corresponde aos primeiro e segundo estágios da simulação e outra ao terceiro. O balanceamento estático de carga pode ser feito de maneira que PL_0 e a segunda fase de PL_1 são alocados a um dos processadores, enquanto que PL_2 e a primeira fase de PL_1 são alocados ao outro processador. Com isso, os dois processadores terão, durante os três estágios da simulação, aproximadamente o mesmo volume de processamento a realizar.

Um ponto a ressaltar é que no exemplo descrito as duas fases de PL_1 são executadas em dois nós de processamento diferentes, mas não necessariamente em paralelo. A segunda fase de PL_1 não é iniciada no começo da simulação, e sim apenas quando a simulação atinge o terceiro estágio. Portanto, a decomposição

temporal não foi utilizada com o objetivo de extrair mais paralelismo da aplicação, mas de permitir que seja feito um melhor balanceamento estático de carga e assim possibilitar um melhor desempenho.

A decomposição temporal também pode ser utilizada por uma estratégia de balanceamento dinâmico de carga, baseada na divisão de um PL em fases durante a execução da simulação e na migração de fases de um nó de processamento para outro.

Quando a decomposição temporal é utilizada para explorar paralelismo da aplicação, surge uma difícil questão a ser resolvida : dado que as duas fases de um PF serão executadas em paralelo — a primeira referente ao intervalo $[0, t)$ e a segunda ao intervalo $[t, H)$ — a segunda fase vai começar o seu processamento utilizando como estado inicial uma estimtiva do estado do PF no instante inicial t do intervalo de tempo correspondente àquela fase. A questão é como determinar esta estimativa do estado inicial do PF para a segunda fase.

Qualquer estimativa utilizada que seja diferente do estado do PF no instante t enviado pela primeira fase para a segunda fará com que a segunda fase sofra *rollback* ao receber o estado migrado. Isto é, caso a estimativa não seja exatamente correta, toda a computação realizada pela segunda fase será cancelada e terá que ser reprocessada. Assim é muito difícil aproveitar a computação realizada pelas fases não iniciais dos PFs.

Uma situação na qual a decomposição temporal pode melhorar bastante o desempenho de uma simulação é quando existe uma grande proporção de eventos de consulta, isto é, eventos que não acarretam nenhuma mudança no estado dos PLs que os processam. Neste caso, o recebimento de um estado migrado não necessariamente causaria *rollback* na segunda fase de um PL, pois possivelmente os eventos não precisariam ser reprocessados. No entanto, a maioria dos sistemas físicos simulados não apresentam uma grande proporção de eventos de consulta.

Tem-se então que o recebimento de um estado quase sempre causa *rollback* na fase à qual ele se destina (supondo que os eventos processados não sejam de consulta), pois esta fase vinha trabalhando com uma estimativa incorreta do seu estado inicial. Para que a simulação distribuída tenha um ganho de paralelismo com a utilização da decomposição temporal da aplicação, é necessário um mecanismo para consertar a computação incorreta mais eficiente do que simplesmente cancelar e refazer tudo.

Em [LL91] e [BCL92] foram propostos mecanismos que realizam esse conserto de forma eficiente, eliminando a necessidade de *rollback*. Estes mecanismos de correção de estados, denominados funções de "fix-up", buscam, a partir do recebimento do estado inicial correto, consertar toda a computação realizada por uma fase não inicial de um PF enquanto ela ainda não conhecia este estado. Desta forma, a simulação paralela não é mais limitada pela restrição de causalidade local. Uma fase não inicial, responsável por simular um PF no intervalo $[t, t'')$, utiliza uma determinada estimativa como o seu estado inicial no instante t. Se esta fase se encontra em um instante t' de simulação, isto é, já simulou todos os eventos que recebeu com tempo de simulação em $[t, t']$, ela já determinou portanto o estado do PF no instante t', baseado no estado inicial estimado. Quando esta fase recebe um novo estado do PF para o instante t, o mecanismo de correção conserta a computação realizada por ela até então, mais precisamente os eventos gerados e o estado do PF no instante t'.

O algoritmo apresentado em [LL91] aplica-se a sistemas físicos parcialmente regenerativos, isto é, sistemas onde pelo menos uma parte do estado do sistema repete-se ciclicamente. De maneira diferente do paradigma espaço-temporal e do algoritmo de *time-warp* com decomposição temporal, que dividem o domínio de tempo estaticamente em intervalos de tempo específicos, o algoritmo apresentado em [LL91] particiona o domínio de tempo durante a simulação, através da repetição parcial de estados.

Os mecanismos propostos neste dois trabalhos exigem conhecimentos específicos sobre o sistema físico para a determinação da estimativa do estado inicial de uma fase e para a construção da função de correção. Estes mecanismos ainda são portanto específicos para as aplicações simuladas — que nos dois casos eram sistemas de rede de filas — e exploram a natureza estocástica deste tipo de aplicação para realizar a correção necessária. Porém, para simulações de sistemas físicos de natureza não estocástica, como por exemplo o sistema de colisão de partículas simulado neste trabalho, um mecanismo semelhante às funções de correção não parece viável.

Capítulo 3

O Paradigma Espaço-Temporal

Neste capítulo é descrito o paradigma de simulação distribuída espaço-temporal que foi apresentado em [CS89a] e mais tarde formulado com mais precisão em [BCL91]. Segundo os autores, este paradigma, além de propor um novo algoritmo de simulação distribuída, é também uma teoria unificadora de simulação no sentido em que pode servir de base para diferentes métodos de simulação.

Este capítulo é composto por duas seções. A primeira apresenta uma especificação formal do sistema físico, que é utilizada na seção seguinte, onde é descrito o algoritmo espaço-temporal.

3.1 Especificação do Sistema Físico

Como visto na seção 2.1, o sistema físico que se deseja simular é modelado por um conjunto de processos físicos que interagem entre si. Uma interação entre dois PFs, por exemplo PF_i e PF_j, é modelada por um evento enviado entre PF_i e PF_j. Deseja-se simular o comportamento deste sistema físico, isto é, determinar os estados dos PFs e os eventos trocados entre eles, no intervalo de tempo $(0, H]$, onde H é o tempo final da simulação (horizonte).

A notação utilizada para descrever o comportamento de um sistema físico é apresentada a seguir :

- $S(i,t)$ é o estado local de PF_i no instante t;
- $M(i,j,t)$ é a mensagem enviada por PF_i para PF_j no instante t — é uma

mensagem de saída de PF_i e uma mensagem de entrada de PF_j no instante t;

- $M(*, i, t)$ são todas as mensagens de entrada de PF_i no instante t;
- $M(i, *, t)$ são todas as mensagens de saída de PF_i no instante t;
- $M(i, j, (u, v])$ são as mensagens $M(i, j, t)$ para todo t no intervalo $(u, v]$, $u < v$; e
- $S(i, (u, v])$ e $S(*, (u, v])$ possuem significados análogos.

O algoritmo espaço-temporal aplica-se à simulação de sistemas físicos que atendam às propriedades de realizabilidade e preditibilidade descritas na seção 2.1. Pela propriedade de realizabilidade, os estados e as mensagens de saída de um processo físico em um intervalo de tempo são funções do seu estado no início do intervalo e das suas mensagens de entrada durante o intervalo. Assim, o comportamento de PF_i é definido por

$$S(i, (u, v]) = f_i(S(i, u), M(*, i, (u, v])) \tag{3.1}$$

e

$$M(i, *, (u, v]) = g_i(S(i, u), M(*, i, (u, v])), \tag{3.2}$$

onde f_i é a função de atualização de estados de PF_i e g_i é a função de determinação de mensagens a serem enviadas por PF_i.

O problema de simulação consiste em determinar $S(*, (0, H])$, que são os estados de todos os PFs do sistema físico em todos os instantes de simulação, e $M(*, *, (0, H])$, que são todas as mensagens trocadas entre os PFs durante a simulação, que satizfaçam as equações 3.1 e 3.2. Na maioria das simulações práticas, porém, deseja-se determinar os estados dos PFs e as mensagens trocadas entre eles em alguns instantes específicos de simulação.

O sistema físico pode ser representado por um diagrama onde a dimensão horizontal representa o espaço (isto é, diferentes PFs) e a dimensão vertical representa o tempo. Existe no diagrama uma linha vertical de tempo para cada processo físico, como na figura 3.1. Este diagrama de linhas de tempo é chamado de diagrama espaço-temporal, com os processos físicos ocupando a dimensão espaço.

Cada ponto do diagrama espaço-temporal representa uma porção do estado do sistema físico em um determinado instante. Dado o ponto (i, t) na altura t na

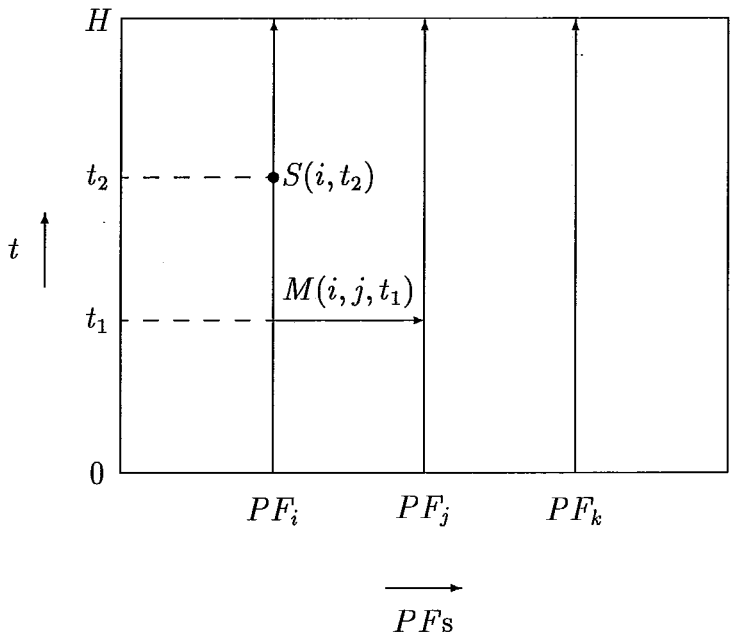


Figura 3.1: Diagrama espaço-temporal

linha de tempo de PF_i no diagrama (figura 3.1), a ele estão associados o estado $S(i,t)$, as mensagens de entrada $M(*,i,t)$ e as mensagens de saída $M(i,*,t)$ de PF_i no instante t. O ponto (i,t) representa o estado $S(i,t)$ de PF_i no instante t. A mensagem $M(i,j,t)$ enviada por PF_i para PF_j no instante t é representada por uma linha horizontal direcionada do ponto (i,t) para o ponto (j,t).

Um algoritmo de simulação é um método de "preencher" o diagrama espaço-temporal, isto é, de determinar o comportamento do sistema físico para todos os pontos (i,t) do diagrama. Para isso, o algoritmo precisa determinar, para cada ponto (i,t), o estado $S(i,t)$, as mensagens de entrada $M(*,i,t)$ e as mensagens de saída $M(i,*,t)$, que satizfaçam as equações 3.1 e 3.2.

O diagrama espaço-temporal pode ser dividido em regiões de número e formatos arbitrários, onde cada região representa uma porção do estado do sistema físico em um intervalo de tempo. Mais precisamente, uma região r do diagrama espaço-temporal representa a porção do estado do sistema físico correspondente aos PFs que estão incluídos em r, para o intervalo de tempo em que cada um desses PFs está contido em r. Uma região pode incluir mais de um PF e um PF pode estar em mais de uma região. Na figura 3.2, por exemplo, o diagrama foi dividido em três regiões, r_0, r_1 e r_2. A região r_0 representa o estado de PF_i para todos os

instantes no intervalo $(0, t']$; r_1 por sua vez representa os estados de PF_i e PF_j para o intervalo $(t', H]$; e, por último, r_2 representa os estados de PF_j para $(0, t']$ e de PF_k para $(0, H]$.

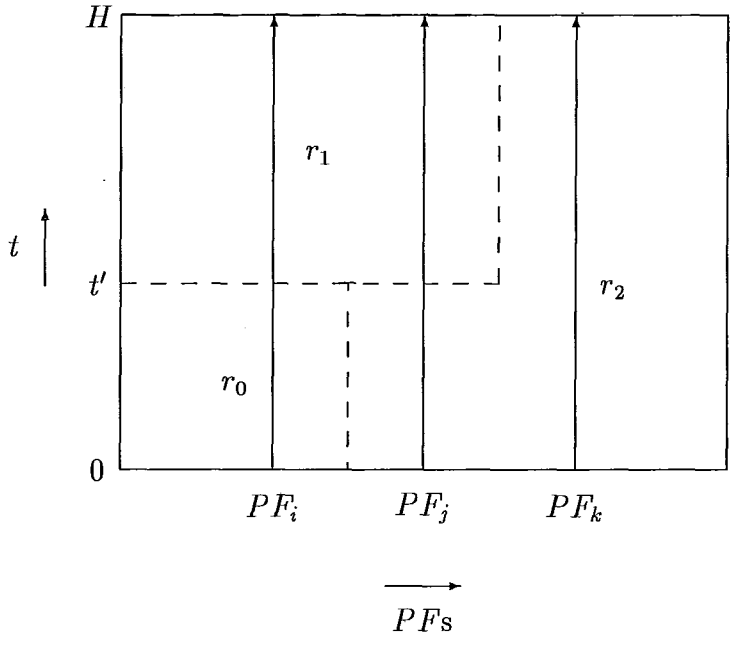


Figura 3.2: Divisão em regiões

Uma região r do diagrama espaço-temporal é definida como um conjunto de pontos (i, t) do diagrama. As entradas de uma região r do diagrama espaço-temporal são :

- $M(i, j, t)$, se o ponto (i, t) não está em r e o ponto (j, t) está em r; e
- $S(i, t)$, se o ponto (i, t) não está em r e o ponto (i, t_+) está em r, onde t_+ é o exato instante depois de t.[1]

As saídas da região r são :

- $M(i, j, t)$, se o ponto (i, t) está em r e o ponto (j, t) não está em r; e
- $S(i, t)$, se o ponto (i, t) está em r e o ponto (i, t_+) não está em r, onde t_+ é o exato instante depois de t.

[1]Esta definição de t_+ é imprecisa quando se considera que o tempo é uma grandeza contínua. Todavia, na prática este tempo é discretizado, e então a definição se aplica.

No diagrama espaço-temporal da figura 3.2, por exemplo, a saída de r_0 destinada a r_1 é $S(i,t')$ e as saídas de r_0 destinadas a r_2 são $M(i,j,(0,t'])$ e $M(i,k,(0,t'])$. As saídas de r_1 que são todas destinadas a r_2 são $M(i,k,(t',H])$ e $M(j,k,(t',H])$. As saídas de r_2 destinadas a r_0 são $M(j,i,(0,t'])$ e $M(k,i,(0,t'])$ e as destinadas a r_1 são $S(j,t')$, $M(k,i,(t',H])$ e $M(k,j,(t',H])$.

3.2 O Algoritmo Espaço-Temporal

Nesta seção o algoritmo espaço-temporal é descrito. O objetivo foi propor um algoritmo de simulação distribuída não-determinístico bastante genérico, de forma que fosse possível derivar diversos algoritmos específicos a partir dele.

O algoritmo espaço-temporal se baseia na idéia apresentada na seção 3.1, de que um algoritmo de simulação é um método de "preencher" o diagrama espaço-temporal, determinando o comportamento do sistema físico para todos os pontos do diagrama. Para isso, o algoritmo inicialmente divide o diagrama espaço-temporal em um número finito arbitrário de regiões de formatos também arbitrários. Em seguida o algoritmo determina o comportamento de cada região, baseando-se nos comportamentos das regiões vizinhas a ela. Duas regiões são vizinhas se compartilham uma fronteira. Assim, existe uma dependência cíclica entre os comportamentos de diferentes regiões, pois o comportamento de uma região r_0 depende do comportamento de uma região vizinha r_1, e o comportamento de r_1 depende do comportamento de r_0. Esta dependência cíclica é resolvida através da utilização de um método de relaxação.

A cada região do diagrama espaço-temporal é associado um processo de simulação (PS) responsável por simular a porção do sistema físico que aquela região representa, isto é, por determinar o estado do sistema físico nos pontos do diagrama espaço-temporal pertencentes àquela região. Mais precisamente, PS_r simula o comportamento de cada PF incluído na região r, para o intervalo de tempo em que cada PF está contido em r. Por exemplo, na figura 3.2 o processo associado à região r_0, PS_{r_0}, simula PF_i no intervalo $(0,t']$, PS_{r_1} simula PF_i e PF_j no intervalo $(t',H]$ e PS_{r_2} simula PF_j no intervalo $(0,t']$ e PF_k no intervalo $(0,H]$.

O conceito de processo de simulação é diferente do conceito de processo lógico. Um PL modela o comportamento de um único PF, enquanto que um PS modela uma região do diagrama espaço-temporal, que pode incluir vários PFs, cada um

deles para um intervalo de tempo diferente. Um único PF pode ser modelado por vários PSs, para um intervalo de tempo diferente em cada PS.

Inicialmente no algoritmo espaço-temporal, o valor de $M(i,j,t)$ é arbitrário, para todo PF_i e PF_j do sistema físico e todo t em $(0,H]$. O valor inicial de $S(i,0)$ para todo PF_i, é o estado inicial correto de PF_i dado pelo sistema físico, enquanto que para $t > 0$ $S(i,t)$ é arbitrário.

Pelo método de relaxação, a simulação de cada região r do diagrama espaço-temporal é realizada da seguinte maneira : da forma como S e M são iniciados, PS_r começa o algoritmo com estimativas das entradas de r, isto é, com estimativas das saídas, destinadas a r, das regiões vizinhas a r. As entradas e as saídas da região r são um conjunto de estados e mensagens, de acordo com a definição na seção 3.1. Assim, PS_r determina as estimativas dos estados $S(i,t)$ e das mensagens $M(i,*,t)$ para todos os pontos (i,t) de r, utilizando as estimativas das entradas de r, de forma a satisfazer as equações 3.1 e 3.2. PS_r envia então as estimativas das saídas de r que acabou de determinar, para os PSs associados às devidas regiões vizinhas a r.

Sempre que PS_r recebe uma nova estimativa de alguma entrada de r, ele determina novamente as estimativas dos estados $S(i,t)$ e das mensagens $M(i,*,t)$ para todos os pontos (i,t) de r, utilizando essas estimativas mais atuais das entradas de r, e satisfazendo as equações 3.1 e 3.2. Se a nova estimativa do estado $S(i,t)$ é diferente da estimativa anterior, para algum ponto (i,t) de r, PS_r envia as novas estimativas das saídas de r, que acabou de determinar, para os PSs associados às devidas regiões vizinhas a r. Caso contrário, PS_r não faz nada.

Com esse procedimento, a simulação de uma região r sempre atribui para as estimativas das saídas de r os valores que são corretos, isto é, que satisfazem as equações 3.1 e 3.2, de acordo com as últimas estimativas recebidas das entradas de r.

À medida que a simulação progride, isto é, que esse procedimento é repetido, as estimativas vão se corrigindo e convergindo para os valores corretos. O algoritmo termina quando um ponto fixo é alcançado. Um ponto fixo de uma computação é um estado que permanece inalterado ao evoluir a computação. No caso desse algoritmo, um ponto fixo é um comportamento para cada região do diagrama espaço-temporal que permanece inalterado ao evoluir a computação. Quando o ponto fixo é atingido, a simulação já convergiu para todos os pontos (i,t) do diagrama, isto é, $S(*,(0,H])$

e $M(*,*,(0,H])$ já convergiram para os valores corretos.

Quanto melhor for a estimativa inicial de $M(i,j,t)$, para todo PF_i, PF_j e todo t, mais rápido o algoritmo convergirá. Assim, $M(i,j,t)$ pode receber um valor nulo ou receber as melhores estimativas que se possui, da teoria ou de simulações anteriores de sistemas físicos similares.

Durante a simulação, cada PS_r só envia as saídas de r para as regiões vizinhas a r se algum estado de r foi alterado. Assim, se todos os PSs estão ociosos, isto é, não possuem novas estimativas das entradas de sua região para tratar, e se, para cada região r, todas as estimativas das saídas de r que foram enviadas já foram recebidas pelas regiões de destino, então a simulação está em um ponto fixo.

No algoritmo 3.1 é apresentado o algoritmo espaço-temporal executado pelo PS_r responsável pela simulação da região r do diagrama espaço-temporal. Todos os PSs executam este mesmo algoritmo, cada um simulando a sua região correspondente.

Início
  $S(*,(0,H])$ e $M(*,*,(0,H])$ são iniciados com valores dados ou arbitrários e $S(*,0)$ é iniciado com o estado inicial correto do sistema físico
  Determine as estimativas de $S(i,t)$ e $M(i,*,t)$ para todo ponto (i,t) de r, seguindo as equações 3.1 e 3.2
  Envie as estimativas das saídas de r para as regiões vizinhas a r
  Enquanto a computação não atingiu um ponto fixo **Faça**
    Espere receber novas estimativas das entradas de r de alguma região vizinha
    Redetermine as estimativas de $S(i,t)$ e $M(i,*,t)$ para todo ponto (i,t) de r, seguindo as equações 3.1 e 3.2
    Se o estado $S(i,t)$ para algum ponto (i,t) de r foi alterado **Então**
      Envie as novas estimativas das saídas de r para as regiões vizinhas a r
    Fim Se
  Fim Enquanto
Fim

Algoritmo 3.1: Algoritmo espaço-temporal executado por PS_r

Como neste algoritmo a dependência entre os comportamentos das regiões existe apenas entre regiões vizinhas, ou seja, a troca de informações ocorre ape-

nas entre regiões vizinhas, dois processos físicos que interajam entre si durante a simulação devem ser modelados pela mesma região ou por regiões vizinhas.

Os PSs não precisam ser iniciados todos ao mesmo tempo no inicio da simulação. De acordo com o algoritmo específico que se deseja derivar a partir do algoritmo espaço-temporal e com a divisão do diagrama espaço-temporal em regiões empregada, cada PS pode iniciar sua execução em um ponto diferente da simulação. É necessário apenas garantir que todos os PSs serão iniciados, ou seja, que todas as regiões serão simuladas.

O algoritmo espaço-temporal pode explorar paralelismo tanto pela decomposição espacial quanto temporal da aplicação, pois a divisão do diagrama espaço-temporal em regiões de forma arbitrária permite que a aplicação seja decomposta no domínio do tempo e do espaço. Em [BCL91] é apresentada a prova de corretude deste algoritmo, que se baseia na aciclicidade das dependências entre as variáveis a serem determinadas (estados e mensagens).

Capítulo 4

O Simulador

Este capítulo descreve o simulador distribuído projetado e implementado neste trabalho. O simulador desenvolvido implementa um algoritmo de simulação distribuída de eventos discretos otimista baseado no algoritmo espaço-temporal descrito no capítulo 3. Conforme visto naquele capítulo, o algoritmo espaço-temporal é extremamente abstrato, o que resultou na necessidade de um projeto bastante detalhado do simulador.

Este capítulo está organizado em três seções. A primeira seção descreve o ambiente de desenvolvimento do simulador, isto é, o computador paralelo, a linguagem de programação e o modelo de comunicação utilizados. A seção seguinte apresenta o simulador e as políticas e os mecanismos que ele implementa para a realização da simulação distribuída otimista. Por último, na terceira seção é descrito o modelo utilizado para o desenvolvimento de aplicações que modelem sistemas físicos para este simulador.

4.1 O Ambiente de Desenvolvimento

O simulador distribuído foi desenvolvido utilizando a linguagem de programação Occam 2 no computador paralelo de memória distribuída NCP I desenvolvido na COPPE/UFRJ. Esta seção apresenta três subseções. A primeira descreve este computador paralelo e também o elemento processador que constitui cada um de seus nós. A subseção seguinte apresenta os conceitos básicos da linguagem de programação Occam 2. A terceira subseção descreve o modelo de comunicação utilizado pelo simulador.

4.1.1 O Multiprocessador NCP I e o *Transputer*

O sistema NCP I é um computador paralelo que encontra-se em fase de desenvolvimento na COPPE/UFRJ. O simulador apresentado neste trabalho foi desenvolvido utilizando a implementação atual do NCP I, que será descrita mais adiante.

O projeto completo do NCP I, descrito em [ACSC91], prevê a construção de um computador paralelo com modelo de arquitetura MIMD que suporte o desenvolvimento de aplicações paralelas baseado tanto no modelo de programação de memória compartilhada quanto no de memória distribuída. Neste projeto, o multiprocessador é composto de 16 nós de processamento idênticos, onde em cada nó existem dois elementos de processamento. O elemento denominado *Private Memory Processing Element* (PME) é constituído de um microprocessador Inmos *transputer* T800, enquanto que o outro elemento de processamento, denominado *Shared Memory Processing Element* (SME) é constituído de um microprocessador Intel i860. Existem ainda em cada nó de processamento dois módulos de memória local, um que apenas o PME pode acessar e outro que é compartilhado pelo PME e pelo SME. Os PMEs são interconectados por canais físicos de comunicação ponto-a-ponto segundo a topologia de um hipercubo. Os SMEs estão ligados a um barramento, através do qual podem acessar a memória centralizada do computador paralelo e a controladora de memória secundária.

No estágio atual, o NCP I é constituído de oito nós de processamento, onde cada nó consiste do elemento de processamento PME. Mais precisamente, cada nó de processamento é composto de um microprocessador Inmos *transputer* T800 e um módulo de memória local de 4 *Mbytes* de capacidade de armazenamento. Os oito nós de processamento estão ligados por uma rede de interconexão segundo uma topologia de hipercubo e comunicam-se exclusivamente por troca de mensagens através desta rede.

Uma rede em topologia de um hipercubo de grau n possui 2^n processadores que são identificados por números de 0 a $2^n - 1$. As conexões entre os processadores são definidas de forma que cada processador tenha n processadores vizinhos, isto é, possua canais físicos de comunicação conectando-o a outros n processadores. A identificação de cada um dos vizinhos de um processador é determinada a partir da inversão de cada um dos n *bits* da representação binária da identificação deste processador.

O NCP I utiliza um microcomputador PC como hospedeiro, através do qual são realizadas as operações de E/S. Apenas um dos nós de processamento está conectado ao hospedeiro, sendo este nó portanto, o único capaz de realizar operações de E/S e denominado nó raiz ou processador raiz.

A figura 4.1 mostra a estrutura de interconexão do NCP I, que consiste de um hipercubo de oito nós de processamento, rotulados com a representação binária de suas identificações. O processador 000 é o processador raiz, possuindo portanto conexão com o hospedeiro.

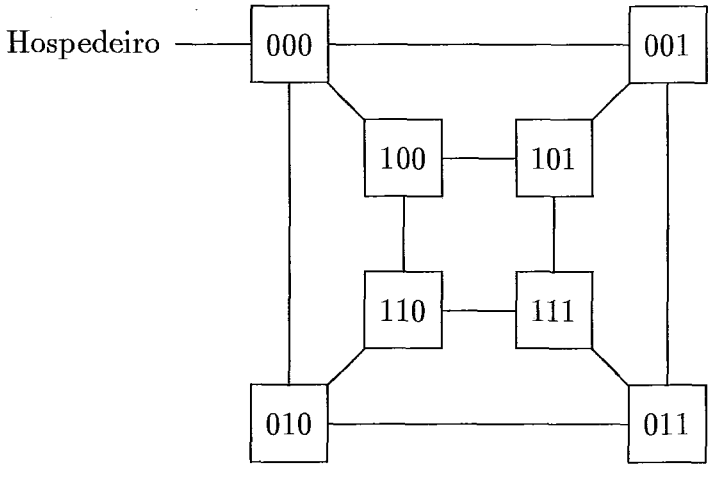


Figura 4.1: Hipercubo de oito nós

O Inmos *transputer* T800 é um microprocessador de 32 *bits* que apresenta na sua única pastilha uma memória interna SRAM de 4 *Kbytes*, uma unidade aritmética de ponto flutuante de 64 *bits* e 4 canais físicos bidirecionais de comunicação. O T800 opera a 20 *MHz* e os canais seriais de comunicação transferem dados na taxa de 20 *Mbits/s*. Uma especificação mais detalhada do microprocessador *transputer* T800 pode ser encontrada em [I88a].

O número de registradores do *transputer* é reduzido devido à existência da memória interna. Seu conjunto de instruções também é simples e foi projetado de forma a permitir a compilação eficiente de linguagens de alto nível.

Para a execução de um único processo seqüencial são utilizados os seis seguintes registradores : o ponteiro *WPtr* para o *workspace* deste processo (o *workspace* de um processo é a área de memória onde são armazenadas as suas variáveis locais);

o ponteiro $IPtr$ para a próxima instrução a ser executada por este processo; o registrador de operandos $OReg$ utilizado para a formação dos operandos das instruções; e os três registradores A, B e C que formam uma pilha de avaliação e são utilizados como fonte e destino da maioria das operações lógicas e aritméticas. É nesta pilha que as expressões são avaliadas, e as instruções referem-se a ela implicitamente.

Estes seis registradores são suficientes para a execução de um único processo seqüencial. No entanto o *transputer* também oferece suporte para a execução de vários processos concorrentemente. Para isso ele possui um escalonador de processos implementado em seu microcódigo (que permite que os processos sejam executados concorrentemente compartilhando o tempo do processador), apresenta dois níveis de prioridade de processos e utiliza mais dois pares de registradores, que serão descritos adiante.

Pelo escalonador um processo pode estar em três estados diferentes : executando, prontos para executar ou suspenso. Em um determinado instante, apenas um processo está executando. Um processo está suspenso quando está esperando por uma comunicação — para receber ou enviar uma mensagem — ou esperando que um intervalo de tempo se complete. Como existem dois níveis de prioridade de processos, cada processo pode ser de alta (prioridade 0) ou de baixa prioridade (prioridade 1).

Os processos prontos para executar são mantidos em duas filas de processos prontos, uma para os processos de alta prioridade e outra para os de baixa prioridade. O *transputer* possui dois pares de registradores que apontam para estas filas de processos : $FPtr0$ e $BPtr0$ apontam, respectivamente, para o início e o fim da fila de processos prontos de alta prioridade; e $FPtr1$ e $BPtr1$ têm funções análogas para a fila de processos prontos de baixa prioridade.

Os processos de alta prioridade são escalonados de acordo com uma política FIFO, ou seja, se há um ou mais processos de alta prioridade prontos para executar, então é selecionado o primeiro processo da lista de processos prontos correspondente. Este processo é executado até que ele tenha que esperar por uma comunicação ou por um intervalo de tempo, ou até que ele termine seu processamento. Um processo de baixa prioridade é selecionado para execução somente se não existe nenhum processo de alta prioridade pronto para executar, ou seja, todos estão suspensos à espera de uma comunicação ou de um intervalo de tempo. Sempre que um processo de baixa prioridade está executando e um processo de alta prioridade torna-se pronto para a

execução, pois a comunicação ou o intervalo de tempo que ele esperava se completou, o processo de baixa prioridade é suspenso pelo escalonador e o processo de alta prioridade passa à execução. Quando novamente não houver nenhum processo de alta prioridade para executar, o processo de baixa prioridade que havia sido interrompido retorna à execução.

Para o escalonamento dos processos de baixa prioridade uma política *round-robin* é utilizada, ou seja, se há um ou mais processos de baixa prioridade prontos para executar, então é selecionado o primeiro processo da lista de processos prontos correspondente. Este processo é executado até que ele tenha que esperar por uma comunicação ou por um intervalo de tempo, ou até que se complete a fatia de tempo de processamento designada a cada processo de baixa prioridade, ou ainda, até que um processo de alta prioridade que estava suspenso torne-se pronto para executar.

Conforme descrito, os processos prontos para executar são mantidos nas filas de processos prontos. Mais precisamente, estas filas armazenam os descritores dos processos prontos. Um processo suspenso esperando por uma comunicação ou por um intervalo de tempo tem seu descritor mantido em uma posição de memória associada ao canal pelo qual se dará a comunicação ou na fila associada ao relógio, respectivamente.

Para que a troca de contexto, quando um processo é suspenso e outro processo é selecionado para a execução, seja eficiente, o escalonador garante que os processos só são suspensos em determinados pontos do código onde a pilha de avaliação esteja vazia. Desta forma, não é necessário copiar o conteúdo desta pilha para posterior restauração.

Este mecanismo de escalonamento descrito foi ligeiramente alterado no simulador implementado para permitir a suspensão da execução de processos de baixa prioridade no simulador, conforme será descrito na subseção 4.2.6, que apresenta o mecanismo de "preempção".

Todo esse suporte em *hardware* e microcódigo para o processamento concorrente de processos oferecido pelo *transputer* permite que não seja necessário que um sistema operacional em *software* realize estas tarefas. Além disso, a existência dos canais físicos de comunicação e de instruções específicas para a troca de mensagens faz do *transputer* um microprocessador adequado para a construção de computadores paralelos distribuídos. Como será visto na subseção 4.1.2, este suporte permite

que a linguagem Occam 2 implemente de maneira direta e eficiente um modelo de concorrência e comunicação.

A configuração atual do NCP I — uma rede de *transputers* que se comunicam por canais físicos — faz dele um ambiente apropriado para o desenvolvimento de sistemas distribuídos, como é o caso do simulador distribuído desenvolvido neste trabalho.

4.1.2 A Linguagem de Programação Occam 2

Para a implementação de algoritmos paralelos é necessário que o ambiente de desenvolvimento apresente primitivas que possibilitem a criação e a execução de processos concorrentes e a comunicação entre estes processos. Esta comunicação pode ser realizada através de variáveis compartilhadas pelos processos concorrentes ou pela troca de mensagens entre eles.

A linguagem Occam 2 ([B88a]), também desenvolvida pela Inmos, tem como objetivo oferecer um ambiente de programação apropriado para o desenvolvimento de aplicações paralelas. Para isso, ela oferece formas simples de expressar paralelismo e primitivas de comunicação entre processos concorrentes, além de possuir as construções tradicionais das linguagens procedurais. As principais construções desta linguagem relacionadas com o desenvolvimento de programas paralelos são descritas nesta subseção.

Um programa em Occam 2 é basicamente uma coleção de processos concorrentes que se comunicam durante a execução através da troca de mensagens. Embora Occam 2 seja uma linguagem de programação de alto nível, seu desenvolvimento foi fortemente associado com o desenvolvimento do *transputer*. O modelo de concorrência de Occam 2 foi bastante influenciado pela necessidade de prover as mesmas técnicas de programação em um único *transputer* e em uma rede de *transputers*. Assim, as construções que expressam paralelismo e comunicação desta linguagem são implementadas diretamente pelas instruções e pelos mecanismos suportados pelo *transputer*.

No Occam 2 a criação de processos concorrentes é feita através da construção **PAR** que possui a estrutura a seguir :

PAR

P_1

$\vdots$

P_n

Os processos $P_1 \ldots P_n$ são executados concorrentemente e podem interagir entre si. Todos possuem o mesmo nível de prioridade. O processo que contém a construção **PAR** só termina quando todos os processos que o constituem terminam.

A construção **PRI PAR** permite a criação de processos concorrentes com diferentes níveis de prioridade. A cada processo da construção é associado um nível de prioridade diferente, sendo que a ordem textual é usada para indicar um nível de prioridade decrescente. Como o *transputer* suporta apenas dois níveis de prioridade, na prática a construção **PRI PAR** permite a criação de processos de alta e de baixa prioridade, e sua estrutura é a seguinte :

PRI PAR

P_1

P_2

A construção **PRI PAR** indica que os processos P_1 e P_2 são executados concorrentemente, sendo que o primeiro é de alta prioridade e o segundo de baixa prioridade. Para associar o mesmo nível de prioridade a um grupo de processos, eles precisam ser envolvidos por uma construção **PAR** interna à construção **PRI PAR**, como mostrado a seguir :

PRI PAR

PAR

P_{alta_1}

$\vdots$

P_{alta_n}

PAR

P_{baixa_1}

$\vdots$

P_{baixa_m}

Na linguagem Occam 2, os processos concorrentes comunicam-se através da troca de mensagens utilizando duas primitivas de comunicação, uma para o envio e outra para o recebimento de uma mensagem. O modelo de comunicação implementado nesta linguagem é bloqueante, isto é, o processo que envia a mensagem fica suspenso até que o processo para o qual a mensagem é destinada execute o comando de recebimento correspondente a esta mensagem.

A troca de mensagens é feita através de canais de comunicação, isto é, um processo envia uma mensagem por um canal, e outro processo recebe esta mensagem por este mesmo canal. Os canais são unidirecionais e conectam apenas dois processos, um deles a origem e o outro o destino das mensagens que transitam por aquele canal.

O Occam 2 oferece também a construção **ALT**, que possibilita que um processo escolha para executar, de forma arbitrária, um entre vários recebimentos de mensagens. A construção **ALT** possui a seguinte estrutura :

ALT
  G_1
    P_1
  ⋮
  G_n
    P_n

G_i é denominado um guarda e P_i é o processo correspondente a este guarda. Um guarda pode ser um comando de recebimento de mensagens acompanhada ou não por uma expressão *booleana*. Durante a execução da construção **ALT**, determina-se quais guardas estão "prontos", isto é, para cada guarda, se a mensagem já está disponível para ser recebida e a expressão *booleana* é verdadeira (se houver). Um dentre todos os guardas prontos é escolhido de forma não-determinística e executado, e o seu processo correspondente também é executado. Se nenhum guarda estiver pronto, o processo que contém a construção **ALT** é suspenso, até que pelo menos um guarda se torne pronto.

Esta construção **ALT** se baseia na noção de comandos guardados e é muito útil no desenvolvimento de algoritmos distribuídos, pois permite que eles sejam escritos e implementados de forma bastante não-determinística. O Occam 2 também oferece

a construção **PRI ALT**, que é semelhante à construção **ALT**, mudando apenas o critério de escolha do guarda a ser executado. Na construção **ALT** um guarda pronto é escolhido não-deterministicamente, enquanto que na **PRI ALT** é escolhido o primeiro guarda pronto de cima para baixo na ordem textual da construção.

O mapeamento de processos concorrentes em diferentes processadores é realizado estaticamente através da construção **PLACED PAR**. A comunicação entre dois processos concorrentes alocados em um único processador ou em dois processadores distintos é realizada através de canais lógicos ou físicos, respectivamente. Nos dois casos, as mesmas primitivas de envio e recebimento de mensagens são utilizadas.

Existem outras linguagens de programação disponíveis para o *transputer*, como o C Paralelo e o Fortran Paralelo. O Occam 2 foi utilizado para a implementação do simulador distribuído desenvolvido neste trabalho por traduzir diretamente os conceitos de desenvolvimento de algoritmos distribuídos. Além disso, devido à relação entre esta linguagem e o *transputer*, estes conceitos são implementados de forma bastante eficiente.

Os programas em Occam 2 são desenvolvidos e executados em um ambiente de trabalho, o *Transputer Development System* (TDS), apresentado em [I88b]. Este ambiente fornece as ferramentas básicas para o desenvolvimento de programas, que são o editor de programas, o compilador, o *link*-editor, o depurador e uma coleção de bibliotecas. O TDS é executado no processador raiz da rede de *transputers* e é responsável por, no início da execução de um programa, distribuir para os processadores da rede os processos a serem executados em cada um deles. Além disso, o TDS é encarregado pela "interface" entre o *transputer* raiz e o computador hospedeiro para a realização das operações de E/S requeridas durante a execução de um programa.

4.1.3 O Processador Virtual de Comunicação

Como o NCP I no seu estágio atual trata-se de um multiprocessador de memória distribuída, os diferentes nós de processamento comunicam-se exclusivamente através da troca de mensagens. Como esses nós estão interconectados por uma rede ponto-a-ponto com a topologia de um hipercubo, cada nó possui canais físicos de comunicação conectando-o a apenas alguns dos demais nós de processamento de rede. Mais precisamente, no hipercubo de grau n um nó possui canais físicos conectando-o somente

aos n nós vizinhos a ele.

Para que cada nó seja capaz de enviar mensagens para qualquer outro nó do hipercubo, seja este seu vizinho ou não, é necessário haver um mecanismo de roteamento de mensagens. Quando é requisitado o envio de uma mensagem de um processador para outro que não é seu vizinho, este mecanismo responsabiliza-se por enviar a mensagem para processadores intermediários, seguindo uma rota pré-estabelecida, até a mensagem alcançar o processador ao qual ela se destina.

Em alguns multiprocessadores este mecanismo já está presente no *hardware* (sob a forma de um processador dedicado de comunicação) ou em um módulo do sistema operacional. Como o ambiente NCP I/TDS-Occam 2 não oferece este mecanismo, fez-se necessária a utilização de um módulo de programa para realizar tal tarefa, chamado Processador Virtual de Comunicação (PVC) e apresentado em [D90].

Este módulo consiste de um processo em Occam 2, fornecido sob a forma de procedimentos em uma biblioteca, e tem como objetivo justamente simular um processador de comunicação em *hardware*. O PVC implementa algoritmos de retransmissão e difusão de mensagens para enviar uma mensagem para os processadores para os quais ela se destina. Entre estes algoritmos, os utilizados pelo simulador são o envio de uma mensagem de um processador para outro, sejam eles vizinhos ou não, podendo ser necessária a retransmissão da mensagem por processos intermediários; e a difusão de uma mensagem, a partir de um processador para todos os demais processadores da rede. O envio de mensagens realizado pelo PVC preserva a ordem de envio das mensagens, isto é, utiliza a política FIFO para a transmissão de mensagens.

A cada processador da rede de *transputers* é alocada uma réplica do PVC, que é executada concorrentemente com os demais processos que compõem o simulador, como será descrito na seção 4.2.

A utilização deste processador virtual de comunicação permite que o roteamento de mensagens na rede fique encapsulado neste módulo, de forma que o simulador usufrua deste mecanismo sem se ater à sua implementação ou execução. Salvo questões de desempenho, o simulador pode tratar a rede de transputers como uma rede completamente conectada num nível lógico, pois qualquer processador pode enviar mensagens para qualquer outro.

4.2 O Simulador Distribuído

O algoritmo espaço-temporal descrito no capítulo 3 foi utilizado como base para o desenvolvimento de um simulador distribuído otimista de eventos discretos, que é apresentado nesta seção. Este simulador foi projetado e implementado no ambiente de desenvolvimento já descrito na seção 4.1. Como o algoritmo espaço-temporal aplica-se à simulação de sistemas físicos que atendam às propriedades de realizabilidade e preditibilidade (ver seção 2.1), o simulador também aplica-se à simulação de tais sistemas.

Conforme visto na seção 3.1, o sistema físico sendo simulado é representado por um diagrama espaço-temporal, e cada região deste diagrama corresponde ao comportamento de um conjunto de processos físicos para intervalos de tempo específicos. O simulador desenvolvido neste trabalho explora paralelismo por decomposição apenas espacial da aplicação, logo o diagrama espaço-temporal é dividido apenas no domínio do espaço. Isto significa que cada região do diagrama corresponde ao comportamento de um conjunto de PFs para todo o intervalo de duração da simulação.

No algoritmo espaço-temporal, as regiões enviam estimativas das suas saídas para as regiões vizinhas e recebem destas regiões estimativas das suas entradas (ver seção 3.1). As entradas e saídas de uma região podem ser mensagens enviadas de um PF para outro e estados de um PF. Como neste simulador a aplicação é decomposta apenas espacialmente, as entradas e as saídas de uma região passam a ser apenas as mensagens trocadas entre os PFs, pois não existe o envio de estados de um PF de uma região para outra.

O simulador é replicado em todos os nós de processamento do computador paralelo. Cada uma dessas réplicas é responsável pela simulação de uma porção da aplicação. Para isso, o diagrama espaço-temporal da aplicação é divido em regiões de forma que o número de regiões seja igual ao número de processadores da máquina. Assim, a porção da aplicação associada a cada réplica do simulador corresponde a uma região do diagrama espaço-temporal. Esta porção da aplicação é implementada como um conjunto de processos lógicos, correspondentes aos processos físicos contidos naquela região. Cada nó de processamento executa então o mesmo algoritmo simulador, porém em cada um deles o simulador é responsável por um conjunto diferente de PLs.

O simulador é composto de três camadas de programa. A primeira camada é responsável pela troca de mensagens entre os nós de processamento da máquina distribuída. Esta camada, chamada de camada de comunicação, implementa os mecanismos de armazenamento temporário e roteamento de mensagens necessários em um sistema distribuído. A segunda camada é a que realmente implementa o algoritmo de simulação. Nela estão implementados os mecanismos necessários para a realização da simulação distribuída otimista, que são escalonamento de eventos, *rollback*, gravação e recuperação de estados, "preempção", detecção de convergência e recuperação de espaço de armazenamento. A terceira camada corresponde à aplicação, e contém os processos lógicos que modelam uma região específica do diagrama espaço-temporal referente ao sistema físico sendo simulado.

O simulador foi desenvolvido de forma a existir uma separação nítida entre as camadas de simulação e de aplicação. O algoritmo de simulação distribuída otimista e as políticas e os mecanismos que ele utiliza são implementados de forma encapsulada na camada de simulação. A camada de aplicação é responsável exclusivamente pela modelagem do sistema físico real.

A separação entre essas duas camadas permite que a aplicação seja desenvolvida de forma independente da camada de simulação, isto é, sem se ater aos mecanismos que implementam o modelo otimista. Mais precisamente, os processos lógicos que compõem a camada de aplicação tratam apenas de processar um evento recebido, isto é, realizar as alterações no estado que modelem a ocorrência daquele evento, e de gerar novos eventos, modelando as relações de causalidade do sistema físico. Cada PL sempre supõe que os eventos estão sendo processados na ordem correta, isto é, respeitando a restrição de causalidade local (ver seção 2.2). A utilização dos mecanismos de *rollback*, preempção, gravação e recuperação de estados e outras questões relacionadas com a simulação distribuída otimista são totalmente "transparentes" para a camada de aplicação.

O isolamento da camada de simulação também permite que sejam simulados diversos sistemas físicos diferentes, sem a necessidade de alterações nas camadas de comunicação e simulação. Este trabalho teve como objetivo projetar e implementar um simulador distribuído otimista genérico baseado no paradigma espaço-temporal, isto é, um simulador distribuído aplicável a uma vasta classe de sistemas físicos. O desenvolvimento de aplicações, isto é, a modelagem do sistema físico real em processos físicos e a construção dos processos lógicos correspondentes a eles é função

do programador da aplicação. A seção 4.3 apresenta o modelo utilizado para o desenvolvimento de aplicações para este simulador.

A figura 4.2 mostra a estrutura do simulador distribuído para o caso de serem utilizados apenas dois nós de processamento da máquina paralela. A troca de mensagens entre os dois processadores é feita através das camadas de comunicação. A camada de simulação recebe mensagens de e envia mensagens para as camadas de simulação residentes em outros processadores através das camadas de comunicação. Estas mensagens podem ser eventos ou mensagens de controle utilizadas pelos mecanismos da simulação distribuída otimista, conforme será visto mais adiante nesta seção. A camada de aplicação é composta pelos PLs que modelam o sistema físico. Conforme será descrito na subseção 4.2.2, que apresenta as estruturas de dados e de controle da camada de simulação, um PL recebe eventos de e envia eventos para outros PLs através da camada de simulação. Para o caso de serem utilizados oito nós de processamento, a estrutura do simulador distribuído permanece a mesma, sendo que são interligadas apenas as camadas de comunicação residentes em processadores vizinhos no hipercubo.

As camadas de comunicação, simulação e aplicação residentes em um mesmo nó de processamento são implementadas como processos concorrentes, sendo que as duas primeiras são processos de alta prioridade e cada processo lógico que compõe a camada de aplicação é um processo de baixa prioridade. Existem duas razões para a atribuição destes níveis de prioridades para os processos que compõem as três camadas. O primeiro motivo é que as camadas de comunicação e simulação são processos gerenciais fundamentais para a evolução da simulação distribuída. É a camada de comunicação que recebe as mensagens vindas de outros processadores e as repassa para a camada de simulação. Esta camada, por sua vez, faz os tratamentos necessários para implementar o modelo otimista e envia e recebe eventos para/dos processos lógicos. Portanto, é desejável que estas duas camadas tenham prioridade de execução, pois são elas que possibilitam que a camada de aplicação tenha trabalho para realizar (isto é, eventos para processar) e assim garantam a evolução da simulação.

O segundo motivo é que, conforme será explicado na subseção 4.2.6, que descreve o mecanismo de "preempção" utilizado pela camada de simulação, podem existir situações em que esta camada conclua que o processamento de um determinado evento pelo processo lógico ao qual ele se destina deve ser interrompido, isto

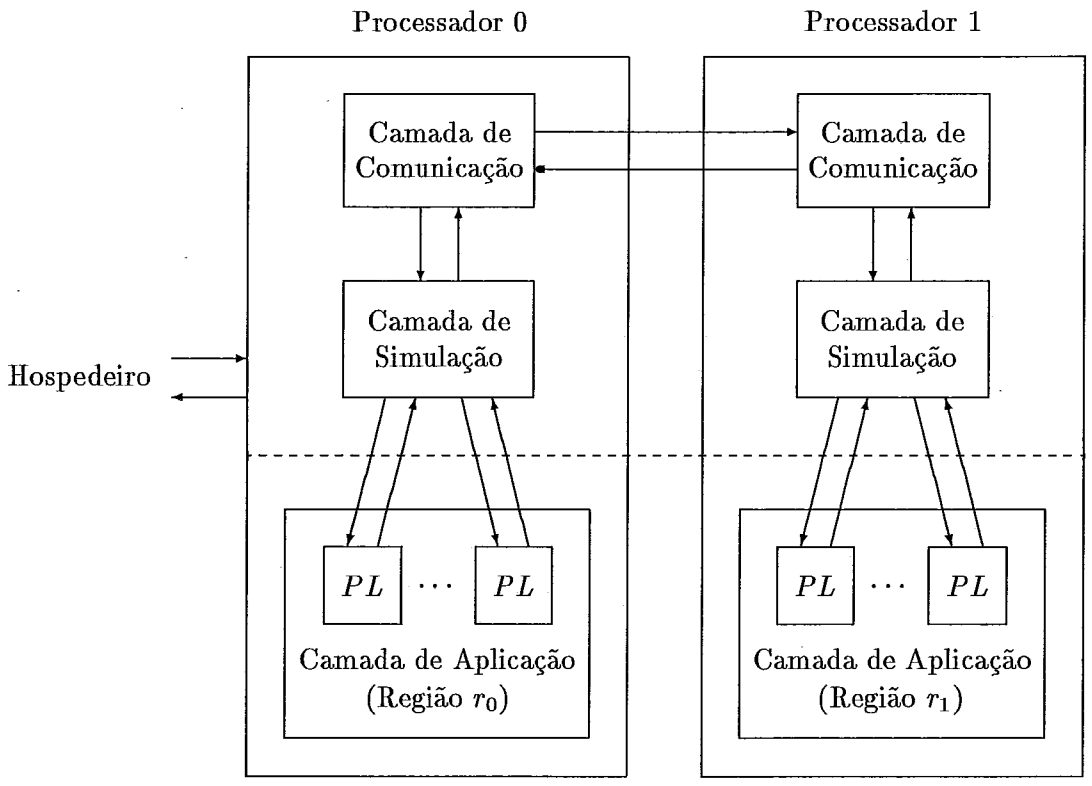


Figura 4.2: Simulador distribuído

é, o PL não deve continuar a execução daquele evento. A atribuição do nível de prioridade baixa aos PLs permite a detecção de tais situações e a realização deste tratamento.

As camadas de comunicação e simulação comunicam-se através de canais lógicos. As camadas de comunicação residentes em dois processadores vizinhos comunicam-se através dos canais físicos do computador paralelo. A cada processo lógico da camada de aplicação estão associados canais lógicos que permitem a ele comunicar-se com a camada de simulação. É através destes canais que o PL troca eventos com a camada de simulação. Por simplicidade, a camada de simulação será chamada apenas de simulador.

Esta seção possui oito subseções. A primeira descreve a camada de comunicação, enquanto que a segunda apresenta as estruturas de dados e de controle do simulador. As seis subseções seguintes descrevem os mecanismos utilizados pelo simulador para a realização da simulação distribuída otimista, que são escalonamento

de eventos, *rollback*, gravação e recuperação de estados, "preempção", detecção de convergência e recuperação de espaço de armazenamento.

4.2.1 Camada de Comunicação

Conforme visto na subseção 4.1.3, porque o NCP I é um computador de memória distribuída com os nós de processamento interconectados ponto-a-ponto segundo a topologia de um hipercubo, é necessária a utilização de uma camada de programa que se encarregue da troca de mensagens entre os processadores da máquina. A camada de comunicação é responsável pelo roteamento de mensagens quando for requisitada a comunicação entre dois processadores não vizinhos na rede. Para isso ela utiliza o processador virtual de comunicação, descrito na subseção 4.1.3, que realiza o roteamento das mensagens.

A camada de comunicação é implementada como um processo concorrente ao processo associado ao simulador. As mensagens vindas de outros processadores são recebidas pela camada de comunicação, que as envia para o simulador através de um canal lógico. Da mesma forma, quando o simulador deseja enviar uma mensagem para outro processador, ele a envia através de um canal lógico para a camada de comunicação, que por sua vez, envia a mensagem para o processador ao qual ela se destina. A troca de mensagens entre as camadas de comunicação alocadas a processadores vizinhos no hipercubo é realizada através dos canais físicos da rede.

Dado que esses canais físicos de comunicação têm capacidade de armazenamento zero, toda a comunicação entre dois simuladores alocados a dois processadores distintos da rede seria bloqueante. Como o simulador distribuído necessita de uma comunicação com maior capacidade de armazenamento para o seu correto funcionamento, tal situação é inviável. Portanto, a camada de comunicação também é responsável pelo armazenamento temporário das mensagens, com o objetivo de permitir que o simulador execute normalmente livre de bloqueios e de situações de *deadlock*.

Para isso são utilizados dois outros processos na camada de comunicação que são concorrentes ao processo associado ao PVC, logo são concorrentes ao simulador e aos processos associados à camada de aplicação. Estes dois processos funcionam como *buffers* para o armazenamento temporário de mensagens que chegam de outros nós de processamento destinadas ao simulador alocado a este nó. Uma mensagem

enviada por outro nó é recebida inicialmente pelo PVC que a envia para o primeiro processo de armazenamento temporário, *buffer* 1, que por sua vez a repassa para o segundo processo, *buffer* 2. É este último processo que enviará a mensagem para o simulador. A estrutura da camada de simulação com os três processos que a compõem é mostrada na figura 4.3. Nesta figura está exemplificada a camada de comunicação alocada ao processador 0, para o caso de serem utilizados apenas dois nós de processamento do computador paralelo.

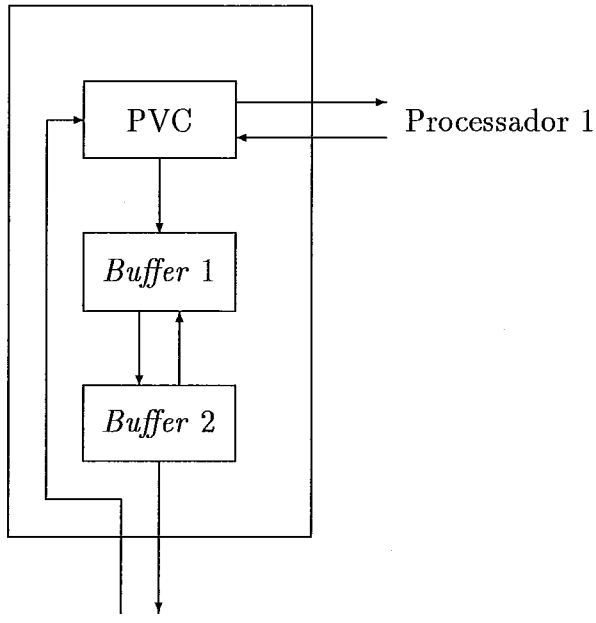


Figura 4.3: Camada de comunicação do processador 0

O processo *buffer* 1 é capaz de armazenar um grande número de mensagens e está sempre pronto para receber mensagens do PVC. Ele as repassa para o processo *buffer* 2, preservando a ordem de envio das mensagens (isto é, assim como o PVC também adota a política FIFO para a transmissão de mensagens). O processo *buffer* 2 é capaz de armazenar apenas uma mensagem. Ele continuamente recebe uma mensagem do processo *buffer* 1 e repassa-a para o simulador. Quando tenta enviar a mensagem para o simulador, o processo *buffer* 2 pode ficar suspenso, esperando o simulador estar pronto para recebê-la.

Após completar o envio da mensagem para o simulador, o processo *buffer* 2 envia um sinal para o processo *buffer* 1, indicando que está pronto para receber uma

nova mensagem. Ao receber este sinal, o processo *buffer* 1 envia uma nova mensagem (caso ele possua pelo menos uma mensagem armazenada) para o processo *buffer* 2, que já está pronto para recebê-la. Com este mecanismo, o processo *buffer* 1 nunca fica bloqueado esperando para enviar uma mensagem para o processo *buffer* 2, pois ele só o faz quando o segundo já sinalizou que está pronto para recebê-la. Assim, o processo *buffer* 1 está sempre pronto para receber mensagens vindas do PVC, desde que sua capacidade de armazenamento não se esgote.

Dado que o processo *buffer* 1 é capaz de armazenar n mensagens, estes dois processos de armazenamento temporário de mensagens juntos simulam um canal de capacidade $n + 1$ entre o simulador e algum dos demais simuladores da rede que envie mensagens para este. Assim, enquanto esta capacidade de armazenamento de mensagens do processo *buffer* 1 não for esgotada, os outros simuladores poderão enviar mensagens para este simulador livremente, isto é, sem ficar suspensos esperando o simulador estar pronto para receber as mensagens, e todos os problemas de degradação de desempenho e possibilidade de *deadlock* estão afastados. O processo *buffer* 2 é o único que pode ficar suspenso, à espera do simulador.

Estes processos para o armazenamento temporário de mensagens são utilizados apenas para o recebimento de mensagens oriundas de outros nós de processamento. Não há necessidade de processos como esses para o envio de mensagens para outros processadores, dado que as mensagens enviadas passarão pelos processos de armazenamento temporário de mensagens recebidas alocados ao processador ao qual elas se destinam. Em outras palavras, havendo processos para armazenamento temporário apenas para o recebimento de mensagens externas, já se garante que o ciclo de processos pelos quais existe fluxo de mensagens está livre de situações de *deadlock* (a menos que a capacidade de armazenamento destes processos seja esgotada).

4.2.2 Estruturas de Dados e Estruturas de Controle

Nesta subseção são apresentadas as principais estruturas de dados e de controle do simulador. Conforme já descrito, existe uma réplica do simulador executando em cada processador, sendo que cada réplica é responsável pela simulação de uma porção do sistema físico, modelada como um conjunto de processos lógicos.

Cada processo lógico do sistema possui variáveis de estado que representam o estado local, em um determinado instante de simulação, do processo físico corres-

pondente àquele PL. O estado local atual de um PL é o seu estado no instante correspondente ao tempo de simulação do último evento processado por aquele PL. Dado o estado local atual de um PL, após ele processar um evento (e possivelmente alterar este estado), as suas variáveis de estado conterão o seu novo estado local. Assim, o estado de um PL em um determinado instante de simulação é obtido a partir do estado inicial do PL e da execução de todos os eventos recebidos pelo PL até aquele instante.

Além das variáveis de estado, cada processo lógico também possui variáveis auxiliares. Estas variáveis são temporárias, isto é, cada vez que um PL recebe um evento para processar, ele atribui valores para estas variáveis e as utiliza. Porém após terminar o tratamento do evento, não é necessário que estes valores sejam mantidos para quando um próximo evento for processado por este PL.

Cada processo lógico do sistema possui uma identificação global única que o distingue dos demais e é utilizada pelos simuladores e pelos outros PLs para referenciá-lo. Um evento no simulador, tal como descrito na seção 2.1, é composto pelos seguintes campos : identificação do processo lógico que o originou, identificação do processo lógico ao qual ele se destina, tempo de simulação em que ocorre o evento e o texto do evento (que contém as informações que representam a interação no sistema físico correspondente a este evento).

De acordo com a filosofia adotada de manter uma separação entre o simulador e a camada de aplicação, sempre que um PL envia um evento destinado a outro PL, este evento é na verdade enviado para o simulador responsável pelo PL que originou o evento. É o simulador que encaminha o evento para o seu PL destino, esteja ele alocado a este mesmo processador ou não. Neste último caso, o simulador envia o evento, através da camada de comunicação, para o simulador responsável pelo PL destino do evento. Da mesma forma, quando um PL recebe um evento para processar oriundo de outro PL (residente neste mesmo processador ou não), na verdade foi o simulador que repassou este evento para ele. Assim, a comunicação entre os PLs do sistema é sempre realizada através dos simuladores, sendo "transparente" para os PLs o caminho seguido e o tratamento recebido pelos eventos.

Para cada PL sob sua responsabilidade, o simulador possui uma lista de eventos destinados àquele PL, ordenados pelo tempo de simulação de forma não-decrescente, e um relógio local que indica o instante de simulação em que o PL se encontra, isto é, o tempo de simulação do último evento processado ou em proces-

samento por aquele PL. Assim, o estado local deste PL (representado pelas suas variáveis de estado) após terminar de processar o evento corresponde ao instante de simulação indicado pelo seu relógio local.

A lista de eventos de um PL contém eventos destinados a ele já processados e ainda a processar. Os eventos já processados precisam ser mantidos na lista pois pode ser necessário cancelar os efeitos da sua computação e reprocessá-los, como será visto na subseção 4.2.4, que descreve o mecanismo de *rollback*. Porém, pode não ser necessário manter na lista de eventos todos os eventos já processados desde o início da simulação. Conforme será descrito na subseção 4.2.8, onde é apresentado o mecanismo de recuperação de espaço de armazenamento, pode bastar manter na lista apenas os eventos já processados mais recentes.

Conforme descrito no início desta subseção, cada PL possui variáveis de estado que representam seu estado local no instante de simulação atual daquele PL. O simulador possui, para cada PL sob sua gerência, uma lista de estados contendo cópias dos estados locais daquele PL em diferentes instantes de simulação. Periodicamente, o simulador precisa gravar o estado local de cada PL sob sua gerência. Estes estados gravados são mantidos nesta lista de estados ordenados de forma não-decrescente pelo tempo de simulação correspondente ao estado. Os estados locais iniciais dos PLs, isto é, os estados dos PLs no instante inicial da simulação, são definidos pela aplicação. O simulador também mantém uma cópia do estado inicial de cada PL na respectiva lista de estados.

Os estados antigos dos PLs são necessários para ser possível realizar o cancelamento da execução de uma seqüência de eventos, quando este PL sofre *rollback*. O mecanismo de gravação de estados é apresentado com detalhes na subseção 4.2.5. De forma similar à lista de eventos, pode não ser necessário manter na lista de estados todos os estados gravados desde o início da simulação, bastando manter apenas os estados mais recentes.

A figura 4.4 mostra as estruturas de dados do simulador associadas a um processo lógico PL_i. Na fila de eventos estão os eventos $e_0, \ldots, e_5$ destinados a PL_i, com tempos de simulação $t_0, \ldots, t_5$, respectivamente, sendo $t_0 < t_1 < \ldots < t_5$. Os eventos e_0, e_1 e e_2 já foram executados, enquanto que os demais ainda não o foram. Portanto, o relógio local de PL_i contém o tempo de simulação t_2 do último evento processado, e_2. Uma cópia do estado inicial de PL_i é mantida em s_{ini}. O estado gravado s_0 corresponde ao estado local de PL_i após executar o evento e_0, a partir do

estado inicial s_{ini}. Da mesma forma, s_1 corresponde ao estado de PL_i após executar e_1 a partir do estado e_0, ou seja, após executar e_0 e e_1 a partir do estado inicial, e assim por diante. Como e_3, e_4 e e_5 ainda não foram executados, s_3, s_4 e s_5 ainda não foram determinados. Neste exemplo é suposto que o estado local de PL_i é gravado após a execução de cada evento. Na subseção 4.2.5 esta questão é discutida.

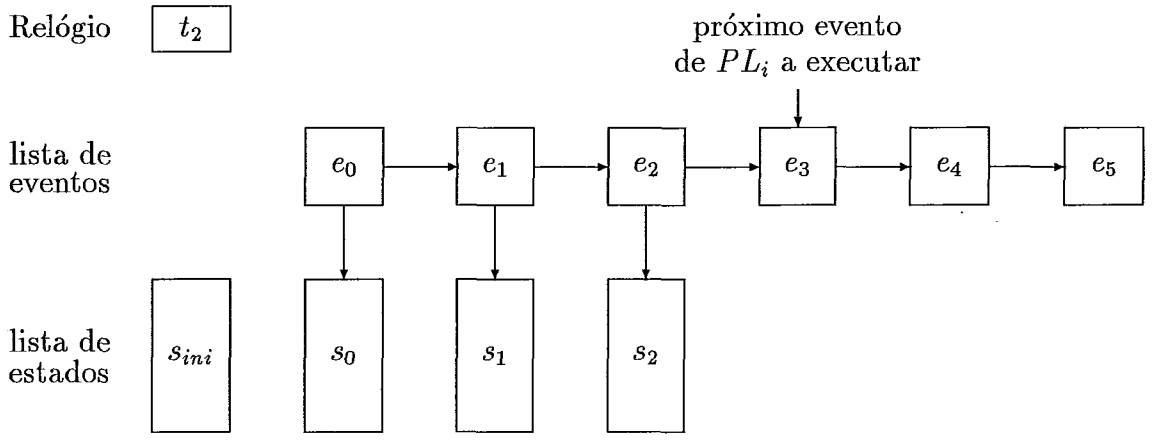


Figura 4.4: Relógio, fila de eventos e fila de estados de PL_i

O simulador possui três fases de processamento : fase de iniciação, de simulação e de finalização. Na primeira fase o simulador inicia as suas estruturas de dados, comunica-se com os simuladores alocados a outros nós de processamento para a troca de informações iniciais e realiza as demais iniciações necessárias para dar começo à simulação. Na segunda fase o simulador executa, continuamente até atingir o fim da simulação, um laço no qual controla a evolução da simulação, recebendo, tratando e enviando mensagens. Na fase de finalização, o simulador comunica aos demais processos concorrentes (das camadas de comunicação e aplicação) que a simulação terminou e realiza as demais tarefas de finalização.

Cada processo lógico do sistema também possui três fases de processamento que correspondem às três fases do simulador. Nas suas fases de iniciação e finalização, da mesma forma que o simulador, o PL realiza tarefas para dar começo à simulação e para finalizá-la, respectivamente. Na sua fase de simulação, um PL executa, continuamente até o fim da simulação, um laço em que espera receber algum evento para processar, trata este evento, isto é, simula a interação no sistema físico correspondente à ocorrência daquele evento, e produz zero ou mais novos eventos

destinados a ele mesmo ou a outros PLs, modelando assim as relações de causalidade do sistema físico. Quando trata o evento, o PL pode alterar as variáveis de estado, representando a mudança no estado local do seu PF correspondente, causada pela ocorrência daquele evento no sistema físico.

Na primeira das suas três fases, o simulador inicia as suas estruturas de dados e variáveis de controle. As listas de eventos e de estados de cada PL são iniciadas vazias e o relógio local de cada PL é iniciado com o instante inicial da simulação. Cada PL no início de sua primeira fase envia sua identificação global para o simulador ao qual está subordinado. Uma vez que cada simulador já tem conhecimento de todos os PLs que estão sob sua gerência, ele troca informações com os demais simuladores da rede, para que todos saibam a que simulador está relacionado cada PL do sistema.

Em seguida, é realizada a distribuição dos dados iniciais obtidos a partir de um arquivo lido da unidade de disco. Este arquivo contém os dados que definem o estado local inicial de cada processo lógico do sistema. Os estados locais iniciais de cada PL do sistema formam um estado inicial global do sistema físico e são fornecidos pelo programador da aplicação. (O conceito de estado global de um sistema distribuído será descrito mais adiante na subseção 4.2.7.) Por ser o processador raiz o único capaz de realizar operações de E/S, é o simulador alocado a este processador que lê da unidade de disco esses dados. Este simulador realiza então a distribuição destes dados, enviando os dados correspondentes aos PLs sob sua responsabilidade diretamente para eles, enquanto que os dados referentes aos PLs alocados a outros processadores são distribuídos pela rede. Este espalhamento é feito de forma que cada simulador da rede (exceto o simulador raiz) receba os dados referentes aos PLs sob sua responsabilidade. Cada um destes simuladores repassa então para cada PL sob sua gerência os dados iniciais referentes a ele.

De posse dos seu dados iniciais, cada PL pode determinar os seus eventos iniciais que dão início a simulação. Estes eventos são chamados de eventos fonte pois não são gerados como conseqüência da execução, isto é, da ocorrência de outros eventos. Cada PL gera zero ou mais eventos fonte, enviando-os para o seu simulador responsável. Ao receber um evento fonte, caso ele se destine a um PL alocado a este mesmo processador, o simulador simplesmente o insere na fila de eventos do seu PL destino. Caso contrário, o simulador encaminha-o para o simulador responsável.

Após receber todos os eventos fonte dos PLs sob sua gerência, o simulador

faz uma cópia do estado local inicial de cada um destes PLs e insere-a na lista de estados gravados do respectivo PL.

Como última tarefa desta fase de iniciação, o simulador precisa coletar certas informações de cada PL sob sua gerência, e para isso, cada um destes PLs envia as suas respectivas informações para o simulador. Estes dados serão utilizados pelo mecanismo de "preempção" executado pelo simulador para interromper o processamento de um evento em um PL. O conteúdo destes dados e este mecanismo serão descritos na subseção 4.2.6. Por ora, basta esclarecer que estes dados fornecem informações suficientes para que o simulador seja capaz de identificar o processo correspondente ao PL que deve ser interrompido e suspendê-lo.

Entre o término do espalhamento dos dados iniciais e o início do recebimento dos eventos fonte existe nos simuladores uma etapa de sincronização. Esta sincronização se dá entre todos os simuladores da rede e é necessária para evitar que um simulador receba eventos para algum PL sob sua gerência antes de receber os dados iniciais referentes àquele PL. O objetivo desta sincronização é definir uma separação entre as mensagens em trânsito da fase de iniciação e as da fase de simulação, pois os eventos só devem começar a ser executados na fase de simulação.

Quando começam a fase de simulação, alguns simuladores da rede já possuem eventos a processar, que são os eventos fonte. Nesta fase, cada simulador executa, continuamente até atingir o fim da simulação, um laço no qual controla a evolução da simulação. Dentro deste laço o simulador pode selecionar um evento para execução ou receber um evento vindo de algum PL sob sua responsabilidade ou alguma mensagem externa (oriunda de um simulador alocado a outro processador da rede).

O simulador seleciona um evento para execução, sempre que nenhum PL sob sua gerência está processando (ou seja, não há nenhum evento sendo executado) e existe algum evento a ser processado (isto é, existe algum evento ainda não executado destinado a um PL alocado a este simulador). O evento selecionado é aquele de menor tempo de simulação, dentre todos os eventos ainda não processados. Este critério é descrito com mais detalhes na subseção 4.2.3 que descreve a política de escalonamento implementada pelo simulador.

Após selecionar o evento e antes de enviá-lo para o seu PL destino, o simulador atualiza o relógio local deste PL, que recebe o tempo de simulação deste evento a ser processado. O simulador envia então o evento selecionado para o PL para o qual

ele se destina (que realizará a simulação deste evento) e continua executando o laço principal, esperando receber mensagens externas e eventos.

Ao receber o evento este PL trata-o e pode produzir novos eventos destinado a ele mesmo ou a outros PLs. Estes eventos são enviados um a um pelo PL para o simulador, que se encarrega de encaminhá-los ao destino correto. Se o PL ao qual o evento se destina também está alocado a este nó de processamento, o simulador o insere na lista de eventos deste PL destino. (Na verdade, é necessário um tratamento mais detalhado que será descrito na subseção 4.2.4.) Caso contrário, tendo sido o PL destino do evento alocado a outro processador, o simulador envia o evento através da camada de comunicação para o simulador responsável por este PL, residente neste outro nó.

Após enviar todos os eventos que gerou para o simulador, o PL em execução envia também um sinal de fim de evento, que tem a função de avisar o simulador que o PL terminou o processamento daquele evento que ele recebeu. O simulador nunca envia um evento para um PL processar antes de receber o sinal de fim de evento do PL que processou o último evento selecionado. Assim, o simulador garante que localmente, isto é, em cada processador, não existem dois PLs processando eventos ao mesmo tempo (mais precisamente, concorrentemente). Não haveria ganho algum com a execução concorrente de eventos destinados a dois ou mais PLs subordinados a um mesmo simulador, pois nenhum paralelismo poderia ser extraído.

Ao receber o sinal de fim de evento do PL que processou o último evento selecionado, o simulador pode gravar uma cópia do estado local deste PL. Essa cópia, se realizada, é inserida na lista de estados gravados daquele PL.

Durante essa fase de simulação, o simulador pode também receber mensagens oriundas de outro nó de processamento da rede. Estas mensagens podem ser eventos vindos de um PL origem alocado a este outro processador ou mensagens de controle que são trocadas entre os simuladores da rede. Quando o simulador recebe um evento, através da camada de comunicação, vindo de um PL alocado a outro processador, ele realiza um tratamento para o recebimento do evento. Na subseção 4.2.4 este tratamento é descrito por completo. Por ora, basta saber que o simulador insere o evento na lista de eventos correspondente ao PL ao qual ele se destina. Como será visto mais adiante, algumas mensagens de controle são utilizadas durante a simulação pelos mecanismos de *rollback* e de detecção de convergência descritos nas subseções 4.2.4 e 4.2.7, respectivamente. Quando o simulador recebe alguma

destas mensagens de controle, ele aciona o mecanismo correspondente para tratá-la.

À medida que os eventos são executados e as mensagens externas tratadas, a simulação vai progredindo. Quando o simulador determina que foi atingido o fim da simulação, ele termina a fase de simulação e passa para a última das suas três fases, a de finalização. O método utilizado pelo simulador para realizar esta detecção de terminação da simulação será descrito na subseção 4.2.7.

Na fase de finalização, o simulador envia para cada PL sob sua gerência um sinal avisando que a simulação chegou ao fim. Ao receber este sinal, cada PL envia os seus dados finais para o simulador e em seguida termina a sua execução. Estes dados representam o estado local do PL no instante final da simulação e assim, os estados locais finais de cada PL do sistema formam um estado final global do sistema físico. Cada simulador recebe então os dados finais de todos os PLs sob sua responsabilidade. Em seguida, é realizado o agrupamento dos dados finais de todos os PLs do sistema, para a sua gravação em um arquivo em disco. Para isso, todos os simuladores da rede, exceto o simulador alocado ao processador raiz, enviam os dados finais dos PLs sob sua gerência para o simulador raiz. Quando este simulador já recebeu os dados de todos os demais simuladores da rede, ele grava estes dados em um arquivo de saída na unidade de disco.

Por último na sua etapa de finalização, o simulador envia sinais para os processos da camada de comunicação informando que a simulação chegou ao fim. Cada processo, ao receber este sinal, termina sua execução.

Entre o término da gravação dos dados finais e o início do envio dos sinais de terminação para a camada de comunicação existe nos simuladores uma etapa de sincronização. Esta sincronização entre todos os simuladores da rede é necessária para sinalizar a todos os simuladores, exceto o raiz, que o simulador raiz já terminou de receber os dados finais de todos os demais simuladores. Com isso evita-se a situação em que em um determinado processador, o processo da camada de comunicação responsável pelo roteamento de mensagens encerre seu processamento e que após isso, algum simulador alocado em outro processador tente enviar seus dados finais para o processador raiz, precisando para isso que estas mensagens sigam uma rota que inclua o processador que já teve seu processo de comunicação terminado.

As três fases, iniciação, simulação e finalização, do algoritmo de simulação distribuída são apresentadas nos algoritmos 4.1, 4.2 e 4.3, respectivamente. Apenas

uma construção da linguagem Occam 2 — a construção **PRI ALT** — é utilizada (ver subseção 4.1.2). Os demais mecanismos utilizados pelo simulador são descritos detalhadamente nas subseções seguintes.

Início
  Para cada PL_i alocado a este simulador **Faça**
    Inicie fila de eventos e fila de estados de PL_i vazia
    Relógio de $PL_i := 0$
  Fim Para
  Espere receber identificação global dos seus PLs e envie para demais simuladores
  Espere receber identificação global dos PLs dos demais simuladores
  Se é o simulador raiz **Então**
    Leia dados iniciais de todos os PLs do arquivo de entrada
    Envie para os demais simuladores os dados iniciais dos seus PLs
  Senão
    Espere receber do simulador raiz dados iniciais dos seus PLs
  Fim Se
  Envie para cada um dos seus PLs os seus dados iniciais
  Faça sincronização com demais simuladores
  Enquanto não receber todos os eventos fonte dos seus PLs (se houver) **Faça**
    Espere receber um evento fonte de um dos seus PLs
    Se o PL destino do evento está alocado a este simulador **Então**
      Insira evento na lista de eventos do PL destino
    Senão
      Envie evento para o simulador responsável pelo PL destino
    Fim Se
  Fim Enquanto
  Grave estado local inicial dos seus PLs e insere nas listas de estados
  Espere receber dos seus PLs dados para "preempção"
Fim

Algoritmo 4.1: Fase de iniciação do simulador distribuído

Início
  Enquanto não chegou ao fim da simulação **Faça**
    PRI ALT
      Espere receber mensagem externa
        Caso a mensagem **Seja**
          Evento
            Insira evento na lista de eventos do PL destino
            Execute tratamento para recebimento do evento
          Mensagem de controle
            Execute mecanismo correspondente à mensagem de controle
        Fim Caso
      Há um PL executando e espere receber um evento deste PL
        Se o PL destino do evento está alocado a este simulador **Então**
          Insire evento na lista de eventos do PL destino
          Execute tratamento para recebimento do evento
        Senão
          Envie evento para o simulador responsável pelo PL destino
        Fim Se
      Há um PL executando e espere receber sinal de fim de evento deste PL
        Se necessário grave estado deste PL e insira na sua lista de estados
      Não há um PL executando e há eventos para processar
        Selecione para execução evento não processado de menor tempo
        Relógio do PL destino do evento := Tempo do evento
        Envie evento para o PL destino
  Fim Enquanto
Fim

Algoritmo 4.2: Fase de simulação do simulador distribuído

4.2.3 Política de Escalonamento

Sempre que nenhum dos PLs sob a sua gerência está em execução e existe algum evento ainda não processado destinado a um destes PLs, o simulador seleciona um evento para ser executado. Os eventos a serem processados são mantidos nas listas de eventos dos PLs aos quais eles se destinam, e lá continuam mesmo depois de

serem executados, pois estas listas também armazenam os eventos já processados.

Início
Envie sinal de terminação para seus PLs
Espere receber de cada um dos seus PLs os seus dados finais
Se não é o simulador raiz **Então**
Envie os dados finais dos seus PLs para o simulador raiz
Senão
Espere receber os dados finais dos PLs dos demais simuladores
Grave dados finais de todos os PLs no arquivo de saída
Fim Se
Faça sincronização com demais simuladores
Envie sinal de terminação para processos da camada de comunicação
Fim

Algoritmo 4.3: Fase de finalização do simulador distribuído

Para selecionar o evento a ser executado, o simulador utiliza o seguinte critério : o evento selecionado é aquele de menor tempo de simulação, dentre todos os eventos ainda não processados destinados aos PLs sob a responsabilidade deste simulador. Como os eventos são mantidos ordenados de forma não-decrescente pelo tempo de simulação nas listas de eventos dos PLs, a implementação deste critério é simples. Além disso, para que somente um dos PLs sob a sua gerência execute por vez, o simulador só seleciona um novo evento para executar, depois que já recebeu o sinal de fim de evento do PL que processou o último evento escalonado.

Utilizando esta política para o escalonamento de eventos, o simulador tenta respeitar a restrição de causalidade local (ver seção 2.2). Como ele não possui informações globais, isto é, sobre todos os PLs do sistema, esse respeito não é necessariamente garantido. O simulador não tem como determinar se o evento que ele selecionou para executar é o evento com menor tempo de simulação dentre todos os eventos ainda não processados do sistema. Ele apenas garante que selecionou o evento com menor tempo de simulação dentre todos os eventos ainda não processados destinados aos PLs sob sua gerência. O mecanismo de *rollback* descrito na subseção a seguir é utilizado para corrigir os eventuais erros causais.

4.2.4 *Rollback*

Por implementar um algoritmo otimista, o simulador processa eventos sem ter certeza de que eles são seguros e assim podem ocorrer erros de causalidade. Quando o simulador recebe um evento atrasado destinado a um PL sob sua gerência, é detectado um erro de causalidade. Para recuperar o erro, o simulador utiliza um mecanismo de *rollback*, que cancela os efeitos dos eventos que este PL processou prematuramente, fazendo-o retornar para o ponto correto onde ele possa receber o evento para processar na ordem correta. O mecanismo de *rollback* precisa cancelar as modificações no estado do PL e o envio dos eventos causados pela execução prematura de alguns eventos.

Para desfazer a alteração no estado de um processo lógico PL_{dest} que recebeu um *straggler* com tempo de simulação t_{str}, o mecanismo de *rollback* restaura para as variáveis de estado de PL_{dest} o seu estado gravado com maior tempo de simulação t_{rest} , tal que $t_{rest} < t_{str}$. É por esta razão que as variáveis de estado dos PLs precisam ser gravadas periodicamente e as cópias mantidas nas listas de estados.

O mecanismo de *rollback* cancela o envio de eventos gerados por PL_{dest} para outros PLs enviando uma mensagem de controle para os simuladores responsáveis pelas regiões vizinhas à região r que engloba PL_{dest}. A função desta mensagem, denominada *inf_rb*, é informar estes simuladores que PL_{dest} sofreu *rollback* para o tempo de simulação t_{str}, e que portanto todos os eventos que estes simuladores receberam oriundos de PL_{dest} com tempo de simulação maior que t_{str} eram estimativas prematuras.

O mecanismo de *rollback* atrasa o relógio local de PL_{dest} para o tempo de simulação t_{rest} do estado restaurado e descarta da lista de estados de PL_{dest} todos os estados gravados com o tempo de simulação maior que t_{rest}. Com isto, o mecanismo fez PL_{dest} retornar para um ponto da simulação no qual poderá receber o evento, sem que ocorra um erro de causalidade. O evento é inserido na fila de eventos de PL_{dest} e o próximo evento de PL_{dest} a executar passa a ser o evento seguinte ao evento correspondente ao estado restaurado.

A partir deste momento, o simulador pode continuar a executar PL_{dest} e os demais PLs sob sua gerência processando os eventos na ordem não-decrescente de tempo de simulação. Quando PL_{dest} receber seu próximo evento a executar, ele irá processá-lo como se nada houvesse acontecido pois este mecanismo de *rollback* é

isolado no simulador e completamente "transparente" para a camada de aplicação.

O algoritmo que implementa o mecanismo de *rollback* é apresentado no algoritmo 4.4. Sempre que o simulador recebe um evento com tempo de simulação t_{str} destinado a um processo lógico PL_{dest} sob sua gerência, e o relógio local de PL_{dest} é maior que t_{str}, este evento é um *straggler* e o mecanismo apresentado é executado.

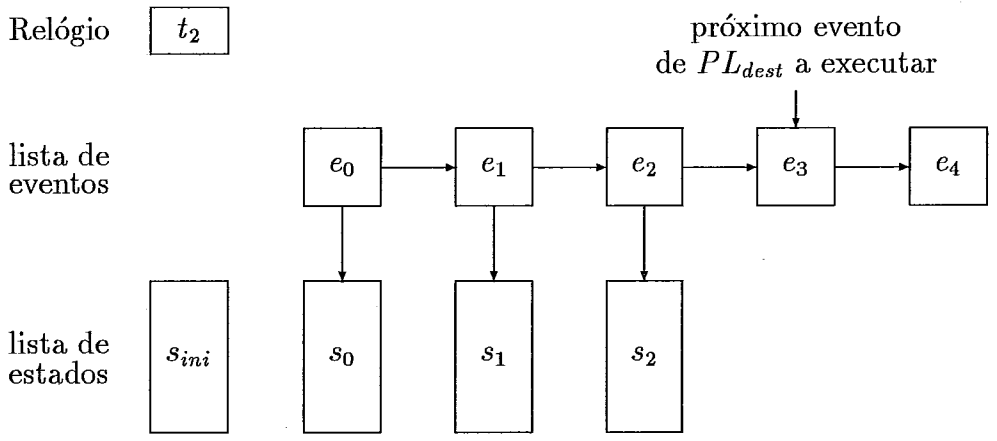
Procedimento *Rollback*(PL_{dest}, t_{str})
Início
Restaure estado s_{rest} de PL_{dest} com o maior tempo t_{rest}, $t_{rest} < t_{str}$
Envie mensagem *inf_rb* para regiões vizinhas a essa, informando
que PL_{dest} sofreu *rollback* para o tempo t_{str}
Cancele estados da fila de estados de PL_{dest} com tempo t, $t > t_{rest}$
Relógio de $PL_{dest} := t_{rest}$
Próximo evento de PL_{dest} a executar é o evento seguinte a s_{rest}
Fim

Algoritmo 4.4: Mecanismo de *rollback*

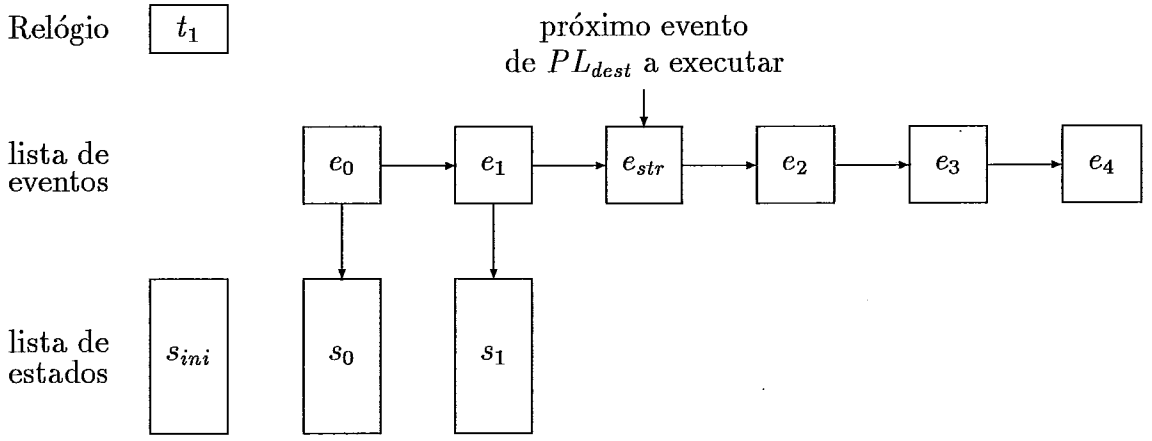
A figura 4.5 exemplifica a execução deste mecanismo de *rollback*. As estruturas de dados de PL_{dest}, antes de receber o *straggler*, são mostradas na figura 4.5(a). Os eventos $e_0, \ldots, e_4$ possuem tempos de simulação $t_0, \ldots, t_4$, respectivamente. Os eventos e_0, e_1 e e_2 já foram processados, enquanto que os demais ainda não o foram. Supõe-se que o estado local de PL_{dest} é gravado após a execução de cada evento. Ao receber o evento e_{str} com tempo de simulação t_{str}, tal que $t_1 < t_{str} < t_2$, o mecanismo de *rollback* é executado. As alterações causadas pelo mecanismo são mostradas na figura 4.5(b). O estado s_1 é restaurado para as variáveis de estado de PL_{dest}. Mensagens *inf_rb* são enviadas para regiões vizinhas à região que engloba PL_{dest}, informando que PL_{dest} sofreu *rollback* para o tempo t_1. O estado s_2 é descartado da fila de estados e o relógio local de PL_{dest} recebe t_1. O evento e_{str} é inserido na fila de eventos de PL_{dest} e o próximo evento de PL_{dest} a executar passa a ser e_{str}. A partir deste ponto, o simulador volta a processar os eventos na ordem não-decrescente de tempo de simulação.

O simulador de cada uma das regiões vizinhas à região que simula PL_{dest} receberá uma mensagem *inf_rb* informando que PL_{dest} sofreu *rollback* para o tempo

de simulação t_{str}. Quando um destes simuladores receber esta mensagem, algum dos processos lógicos sob sua gerência, por exemplo PL'_{dest}, pode já haver executado eventos oriundos de PL_{dest} com tempo de simulação maior que t_{str}. Neste caso, PL'_{dest} também processou eventos prematuramente, logo também sofrerá *rollback*. PL'_{dest} terá que retornar para o tempo de simulação do evento de menor tempo dentre os eventos processados prematuramente, isto é, para o tempo de simulação do evento processado por PL'_{dest} e oriundo de PL_{dest} com menor tempo t_{rb}, tal que $t_{rb} > t_{str}$.



(a) Antes de receber o *straggler* e_{str}



(b) Após receber o *straggler* e_{str}

Figura 4.5: Mecanismo de *rollback* em PL_{dest}

O tratamento deste *rollback* é o mesmo de quando o simulador recebe um

evento atrasado. O mecanismo vai restaurar um estado antigo de PL'_{dest}, enviar mensagens *inf_rb* para as regiões vizinhas à região que engloba PL'_{dest}, cancelar estados correspondentes a eventos processados prematuramente da fila de estados de PL'_{dest}, atrasar seu relógio local e determinar seu novo próximo evento a executar. Ou seja, o mesmo procedimento de *rollback* apresentado no algoritmo 4.4 é executado. Porém neste caso o mecanismo atua sobre PL'_{dest} para fazê-lo sofrer *rollback* para o tempo de simulação t_{rb}.

Realizando este procedimento sucessivamente, o simulador propaga o *rollback* para todas as regiões afetadas pela computação incorreta e todos os seus efeitos são cancelados. À medida que o *rollback* se propaga de um PL para outro, os PLs envolvidos retornam para tempos de simulação cada vez maiores, ou seja, cancelam um volume menor de processamento. Este protocolo garante que a simulação sempre progride, pois no pior caso todas as regiões fazem *rollback* para o tempo de simulação t_{str} do evento que deu início ao processo de *rollback* (*straggler*).

Se, quando um simulador receber a mensagem *inf_rb* informando que PL_{dest} sofreu *rollback* para um tempo de simulação t_{str}, nenhum dos processos lógicos sob sua gerência houver executado eventos oriundos de PL_{dest} com tempo de simulação maior que t_{str}, a mensagem *inf_rb* não causa *rollback* nesta região.

Cada simulador, após terminar o tratamento local do *rollback*, volta a selecionar para processamento os eventos destinados aos PLs sob sua gerência seguindo a ordem não-decrescente de tempo de simulação. Como os eventos destinados aos PLs sob a gerência de um simulador são processados segundo esta ordem de tempo, apenas eventos oriundos de PLs alocados a outros processadores podem dar início a um processo de *rollback*.

Ao receber um evento e com tempo de simulação t para processar, um processo lógico PL_i executa-o, atualizando o seu estado local para o tempo de simulação t, e gera os eventos correspondentes à proxima interação com menor tempo de simulação t', $t' \geq t$. É possível que após terminar de processar o evento e, PL_i receba um novo evento para processar com tempo de simulação t'', tal que $t \leq t'' \leq t'$. Neste caso, se aquela interação com tempo de simulação t' ainda for válida, os eventos correspondentes a ela serão gerados novamente. Caso contrário, PL_i gerará os eventos correspondentes à nova próxima interação com menor tempo de simulação maior ou igual a t''. Em ambos os casos, os eventos gerados inicialmente devem ser cancelados, pois serão repetidos ou inválidos.

Portanto, além do mecanismo de *rollback* descrito, também é necessário um tratamento para o cancelamento de eventos. Sempre que o simulador recebe um evento e com tempo de simulação t destinado a um PL sob sua gerência, por exemplo PL_i, que cause *rollback* ou não em PL_i, o simulador insere e na lista de eventos de PL_i. Além disso, o simulador cancela da lista de eventos de PL_i todos os eventos oriundos do mesmo PL que e com tempos de simulação maiores que t. Um tratamento semelhante a este é realizado quando o simulador recebe uma mensagem *inf_rb* que cause *rollback* ou não.

No algoritmo de relaxação aplicado pelo paradigma espaço-temporal o comportamento de uma região depende de e afeta apenas o comportamento das regiões vizinhas a ela. Por isso, quando um dos PLs sob a sua gerência sofre *rollback*, o simulador precisa enviar a mensagem *inf_rb* apenas para os simuladores responsáveis pelas regiões vizinhas à sua região.

Este mecanismo de *rollback* é um pouco diferente do utilizado pelo algoritmo de *time warp*, onde os eventos enviados prematuramente são cancelados um a um de forma explícita, através do envio de uma anti-mensagem para cada evento a ser cancelado. Para isso, o algoritmo de *time warp* utiliza a lista de eventos enviados (ver seção 2.4).

O mecanismo de *rollback* do simulador desenvolvido cancela os eventos enviados prematuramente através do envio de apenas uma mensagem *inf_rb* para cada região vizinha. Este mecanismo é bastante eficiente pois não necessita da lista de eventos enviados e principalmente porque há uma grande redução no volume de mensagens. No entanto, este protocolo requer que os canais de comunicação preservem a ordem de envio das mensagens. De fato, os canais físicos de comunicação do computador distribuído e o processador virtual de comunicação preservam esta ordem.

Este mecanismo de cancelamento do envio de eventos é agressivo, no sentido em que não espera para testar se o reprocessamento dos eventos desfeitos pelo *rollback* vai gerar os mesmos eventos, antes de cancelá-los.

4.2.5 Gravação e Recuperação de Estados

As variáveis de estado de um PL são mantidas numa área de memória de forma que elas sejam compartilhadas entre aquele PL e o seu simulador responsável. O PL, ao

processar um evento, utiliza as variáveis e pode alterá-las, indicando uma mudança no seu estado local causada pela ocorrência daquele evento.

Ao término do processamento de um evento pelo PL ao qual ele se destinava, o simulador pode acessar as variáveis de estado deste PL para fazer uma cópia do estado local atual do PL. Para isso, o simulador aloca espaço para um novo estado gravado, copia as variáveis de estado do PL para este estado alocado e o insere na lista de estados do PL.

Quando o simulador precisa fazer um dos PLs sob sua gerência retornar para um ponto anterior da simulação devido ao recebimento de um evento atrasado ou de uma mensagem *inf_rb* atrasada, o simulador restaura um estado antigo deste PL, no qual este recebimento não configure um erro de causalidade. Uma vez determinado qual estado gravado da lista de estados do PL deve ser restaurado, o simulador copia este estado para as variáveis de estado do PL. Assim, quando o PL executar novamente, seu estado local atual será o estado restaurado.

Como o simulador somente acessa as variáveis de estado de um PL quando ele já terminou de processar um evento, garante-se que as variáveis de estado, que são compartilhadas pelo PL e pelo simulador, são acessadas com exclusão mútua.

A freqüência com que o estado local de um PL é gravado pode ser variada, podendo ser a cada evento processado pelo PL ou adotando como espaçamento um número maior de eventos processados. A escolha desta freqüência é uma questão de compromisso, pois quando os estados são gravados muito freqüentemente, torna-se necessária a disponibilidade de bastante memória. Por outro lado, se a distância entre dois estados gravados é muito grande, isto é, muitos eventos foram processados por este PL desde a última vez em que ele teve um estado seu gravado, quando ocorre um *rollback* o PL pode precisar voltar atrás mais do que o necessário inicialmente para restaurar um estado antigo. Quando esta situação ocorre, é necessário reprocessar alguns eventos que não precisariam ser desfeitos, isto é, eventos com tempo de simulação menor que o tempo de simulação do *straggler* e maior que o tempo de simulação do estado restaurado.

4.2.6 Preempção

Quando o simulador recebe um evento atrasado ou uma mensagem *inf_rb* que cause *rollback* para o tempo de simulação t_{rb} em um dos PLs sob sua gerência, é possível

que este processo lógico, PL_i, esteja processando um evento e com tempo de simulação maior que t_{rb}. Neste caso, não faz sentido permitir que PL_i prossiga com o processamento de e, pois já é sabido que este processamento será desfeito e seus efeitos cancelados. Inclusive, o quanto antes o mecanismo de *rollback* for iniciado, mais rapidamente a computação incorreta será cancelada, propagando-se menos pelo sistema. Assim, sempre que essa situação descrita ocorre, o simulador utiliza um mecanismo de "preempção" para interromper PL_i e suspender o processamento do evento e.

O mecanismo de interrupção foi implementado com algumas alterações no mecanismo de escalonamento de processos do processador T800, descrito na subseção 4.1.1. Como os PLs são processos de baixa prioridade, um PL só é executado quando não há nenhum processo de alta prioridade pronto para executar, isto é, todos os processos de alta prioridade — que são os processos da camada de comunicação e o simulador — estão suspensos esperando pelo recebimento de uma mensagem.

Pelo mecanismo de escalonamento de processos original do T800, quando um PL está executando e chega uma mensagem qualquer, oriunda de outro nó de processamento da rede, este PL é suspenso temporariamente, para que os processos da camada de simulação recebam a mensagem e repassem-na para o simulador. Quando estes processos ficarem suspensos novamente, o PL prosseguirá com sua execução.

Ao receber a mensagem externa e detectar que o PL que estava executando não deve retomar sua execução, o simulador executa o mecanismo de "preempção". Este mecanismo altera o estado do processo associado ao PL, que deixará de estar suspenso esperando poder retomar sua execução, e passará a estar esperando receber do simulador um novo evento para executar, como os demais PLs sob a gerência deste simulador.

Para isso, o simulador precisa conhecer, para cada PL sob sua gerência, o descritor do processo associado ao PL, e o endereço da instrução de recebimento pelo PL de um evento vindo do simulador. São essas as informações que os PLs enviam para o simulador ao final da fase de iniciação. O mecanismo de "preempção" foi implementado utilizando instruções do T800 ([I88c]) que permitem alterações na fila de processos prontos de baixa prioridade.

4.2.7 Detecção de Convergência

Uma tarefa do simulador é também determinar o quanto a simulação já progrediu, isto é, a partir do instante 0 inicial da simulação, até que instante T as estimativas sobre os comportamentos de todas as regiões já estão corretas. Quando isso é determinado, diz-se que a simulação já convergiu para o intervalo $(0, T]$. O mecanismo de detecção de convergência é utilizado pelo simulador para determinar a terminação da simulação e para a recuperação de espaço de armazenamento, conforme será descrito na subseção 4.2.8.

Este mecanismo de detecção de convergência baseia-se na noção de um estado global de um sistema distribuído e na detecção de propriedades estáveis deste sistema distribuído [CL85]. Um estado conjunto de um sistema distribuído é dado por um estado local de cada processo e um conjunto de mensagens em trânsito em cada canal de comunicação do sistema. Um estado global é um estado conjunto consistente, pois devido à ausência de uma base de tempo global, nem todos os estados conjuntos são realizáveis.

Uma propriedade estável de uma computação distribuída é tal que, uma vez que ela se torne verdadeira em um certo ponto da computação, ela permanecerá verdadeira em todos os pontos seguintes desta computação. Um exemplo de uma propriedade estável é a terminação da computação. A detecção de terminação de uma computação seqüencial é simples : a computação termina quando o processo fica ocioso. Porém, em um sistema distribuído não se pode concluir que a computação terminou quando os processos ficam ociosos. Pode haver alguma mensagem em trânsito a ser recebida por um processo que voltará a ficar ativo na computação em conseqüência deste recebimento.

Especificamente, pode-se argumentar que se tem um estado global de um sistema distribuído sempre que todos os canais de comunicação estão vazios, isto é, não há mensagens em trânsito. Em particular, se neste estado global os processos estão ociosos, então uma condição de terminação foi atingida. Ao restringir estes conceitos a intervalos de tempo de simulação, esta condição de terminação se traduz em uma condição de convergência dentro daqueles intervalos. Detectar a terminação da simulação corresponde a detectar a convergência da simulação para todo o intervalo $(0, H]$ da simulação, onde H é o horizonte final da simulação.

Para realizar a detecção de convergência foi implementado um algoritmo de

controle centralizado baseado na idéia proposta em [CS89a]. O problema consiste em determinar o instante T de simulação para o qual não existam mensagens em trânsito com tempos menores ou iguais a T, e os simuladores estejam ociosos com relação a eventos naquele intervalo (isto é, não possuam eventos ainda não processados com tempos de simulação menores ou iguais a T).

O algoritmo utiliza um processo chamado detector, existente em apenas um processador da rede, que pode se comunicar com os processos simuladores residentes em todos os processadores da rede. O simulador responsável pela região r do diagrama espaço-temporal possui contadores $rec[r', t]$ e $env[r', t]$ que são, respectivamente, os números de mensagens com tempo de simulação t que ele recebeu de e enviou para a região r', para cada região r' vizinha a r e cada tempo de simulação t em $(0, H]$. Quando o simulador fica ocioso, isto é, quando ele não tem nenhum evento para processar e todos os PLs sob sua responsabilidade estão ociosos também, ele envia estes contadores para o detector.

O detector recebe estes contadores de cada região, e os armazena nas variáveis $rec[r, r', t]$ e $env[r', r, t]$, que são os números de mensagens com tempo t que a região r recebeu de e enviou para a região r', respectivamente. O detector determina que a simulação convergiu até o tempo T, se para todos tempos $t \leq T$ e todas as regiões vizinhas r e r', $rec[r, r', t] = env[r', r, t]$.

A igualdade de $rec[r, r', t]$ e $env[r', r, t]$, para todo $t \leq T$ e todas as regiões r e r' vizinhas, garante que os canais de comunicação estão vazios, com respeito ao intervalo $(0, T]$. Aliado a isso, o fato de que cada simulador envia seus contadores para o detector quando está ocioso garante que a simulação convergiu para o intervalo de tempo $(0, T]$. Isto é, garante-se que não será realizado mais nenhum tipo de processamento com respeito a tempos de simulação menores ou iguais a T, dado que a convergência é uma propriedade estável.

Sempre que determina uma convergência para um tempo de simulação maior, o detector avisa todos os simuladores. Ao receber o aviso da convergência da simulação até um instante T, o simulador toma providências em relação ao gerenciamento de espaço de armazenamento, conforme será descrito na subseção a seguir. Ao receber o aviso da convergência da simulação até o instante H final da simulação — o que corresponde à terminação da simulação — o simulador deixa o laço principal da sua seção de simulação e passa para a sua seção de terminação.

Na verdade, a implementação dos contadores $rec[r', t]$ e $env[r', t]$ do simulador e dos contadores correspondentes do detector é um pouco diferente. Como o tempo de simulação é uma grandeza contínua, não é possível definir os contadores da forma como foi feita, onde $rec[r', t]$ contém o número de mensagens que a região r recebeu de uma região r' vizinha, com tempo de simulação t. Para a implementação ser factível, o domínio de tempo $(0, H]$ da simulação é dividido em intervalos, o que corresponde a discretizar o tempo de simulação. Dado que $(0, H]$ é dividido em intervalos, o contador $rec[r', i]$ conterá o número de mensagens que a região r recebeu de uma região r' vizinha, com tempo de simulação t, tal que t esteja contido no i-ésimo intervalo de $(0, H]$.

4.2.8 Recuperação de Espaço de Armazenamento

Conforme já descrito, o simulador possui para cada PL sob sua responsabilidade uma lista de eventos e uma de estados. A primeira armazena os eventos já processados e ainda a processar destinados ao PL, enquanto que a segunda armazena cópias do estado local do PL correspondentes a tempos de simulação do passado do PL.

Como o espaço de armazenamento na memória em cada processador do computador paralelo é limitado, é desejável manter nestas listas apenas os eventos e estados realmente necessários para o prosseguimento da simulação.

Os eventos já processados e os estados antigos são necessários para realizar um *rollback* quando ocorre um erro de causalidade. Porém não é necessário manter todos os eventos já processados e os estados gravados desde o início da simulação. Quando a simulação convergiu até o instante T, não existem mais mensagens em trânsito com tempos de simulação menores ou iguais a T (o que equivale a dizer que não existem mais estimativas incorretas para instantes menores ou iguais a T). Logo, não é possível que ocorra *rollback* para um tempo de simulação menor que T. Assim, é necessário manter apenas os eventos já processados e estados gravados com tempos de simulação maiores que T.

Quando recebe do detector o aviso de que a simulação convergiu até o tempo de simulação T, o simulador executa um mecanismo de recuperação de espaço de armazenamento. Este mecanismo, chamado de "coleta de fósseis", descarta todos os eventos e estados com tempo de simulação t, $t < T$, liberando o espaço por eles ocupado.

O termo "coleta de fósseis" vem do inglês *fossil collection*, um jargão da área que se refere à recuperação de espaço de memória através da liberação de estruturas que não são mais necessárias, sendo portanto um processo distinto do que se costuma chamar de *garbage collection.*

Com a liberação de espaço ocupado por eventos e estados, este espaço de armazenamento pode ser reutilizado, atenuando as restrições impostas pela limitação da memória disponível. Por este motivo, na simulação distribuída otimista, é mais interessante utilizar um mecanismo de detecção de convergência do que simplesmente a detecção de terminação.

4.3 Desenvolvimento de Aplicações

Nesta seção é apresentado o modelo utilizado para o desenvolvimento de aplicações que modelem sistemas físicos para o simulador distribuído desenvolvido. Qualquer sistema físico que satizfaça às propriedades de realizabilidade e preditibilidade (ver seção 2.1) pode ser tratado pelo simulador.

A modelagem do sistema físico como um conjunto de processos físicos e da interação entre eles através de eventos, assim como a implementação destes PFs como processos lógicos, é responsabilidade do programador da aplicação.

O formato do diagrama espaço-temporal é definido pelo sistema físico a ser simulado. Como o paradigma espaço-temporal utiliza a noção de vizinhança em seu algoritmo de relaxação, a divisão do diagrama em regiões deve ser feita de forma que dois PFs que interajam entre si sejam alocados à mesma região ou a duas regiões vizinhas no diagrama.

Por simplicidade, a cada procesador do computador paralelo é alocada uma região do diagrama espaço-temporal, isto é, um conjunto de processos lógicos. Assim, o diagrama é dividido em regiões de forma que o número de regiões seja igual ao número de processadores utilizados da máquina.

A divisão do diagrama espaço-temporal em regiões e a alocação delas aos processadores também deve ter como objetivo a distribuição equilibrada do volume de computação e a redução do volume de comunicação. Para isso, o programador da aplicação deve alocar os PLs aos processadores utilizando um balanceamento estático de carga.

Cada processo lógico modela um processo físico do sistema físico e interage com outros PLs recebendo e enviando eventos. Conforme descrito na subseção 4.2.2, cada PL possui três fases de processamento : iniciação, simulação e finalização. Estas fases foram descritas na subseção citada, juntamente com a descrição do simulador e são apresentadas no algoritmo 4.5.

```
Declaração das variáveis de estado
Declaração das variáveis temporárias
Fase de iniciação
Início
    Envie sua identificação global para o simulador
    Espere receber do simulador seus dados iniciais
    Envie seus eventos fonte para o simulador
    Envie dados para "preempção" para o simulador
Fim
Fase de simulação
Início
    Enquanto não chegou ao fim da simulação Faça
        PRI ALT
            Espere receber do simulador evento
                Processe evento podendo alterar variáveis de estado
                Envie eventos para o simulador
                Envie sinal de fim de evento para o simulador
            Espere receber do simulador sinal de fim de simulação
    Fim Enquanto
Fim
Fase de finalização
Início
    Envie seus dados finais para o simulador
Fim
```

Algoritmo 4.5: Algoritmo executado por um processo lógico

O simulador fornece para o programador da aplicação um módulo de rotinas que implementam a "interface" entre ele e o processo lógico. Existem por exemplo

rotinas que permitem ao PL receber um evento do simulador, enviar um evento para o simulador, enviar sua identificação, receber os dados iniciais, enviar os dados finais e enviar o sinal de fim de evento. Assim, o programador da aplicação deve desenvolver os processos lógicos seguindo a estrutura apresentada no algoritmo 4.5 e utilizando as rotinas deste módulo fornecido pelo simulador.

Capítulo 5

Avaliação de Desempenho

Este capítulo apresenta uma avaliação experimental de desempenho do simulador distribuído desenvolvido baseado no paradigma espaço-temporal. Para realizar esta avaliação foi desenvolvida uma aplicação que modela um sistema de colisão de partículas em duas dimensões. Através da simulação desta aplicação, vários experimentos foram realizados com o simulador distribuído, com o objetivo de medir diversos critérios de desempenho. Estes experimentos foram realizados no computador paralelo descrito na seção 4.1. Através das medidas de desempenho uma avaliação do comportamento do simulador é realizada.

Duas seções compõem este capítulo. A primeira descreve o sistema físico de colisão de partículas. Também são apresentadas a estratégia utilizada para a modelagem deste sistema físico como um conjunto de processos físicos e das interações entre estes PFs, e a forma de exploração de paralelismo deste sistema físico. A segunda seção descreve os experimentos realizados com o simulador, através da apresentação das medidas de desempenho tomadas e dos resultados obtidos. Uma avaliação destes resultados com o objetivo de concluir o comportamento geral do simulador distribuído é apresentada.

5.1 Sistema de Colisão de Partículas

Em um sistema de colisão de partículas em duas dimensões, um determinado número de partículas esféricas (ou melhor, discos) estão em movimento dentro de um domínio de duas dimensões, limitado por bordas (figura 5.1). As partículas colidem umas com as outras e com as bordas do domínio.

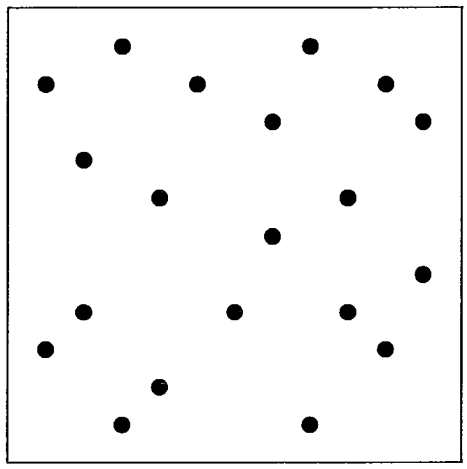


Figura 5.1: Sistema de colisão de partículas

Apesar do sistema de colisão de partículas ser de difícil tratamento, seu estudo é bastante interessante, pois o sistema possui diversas aplicações práticas e sua simulação é computacionalmente custosa. Além disso, o sistema de colisão de partículas ainda não havia sido tratado pelo paradigma espaço-temporal. Dado que os algoritmos distribuídos conservadores não são apropriados para tratarem este sistema, pois ele não apresenta nenhum *lookahead*, uma estratégia otimista deve ser empregada.

Um estado de um sistema de colisão de partículas em um determinado instante é dado pela posição e a velocidade de cada partícula naquele instante. Dado um estado inicial, deseja-se simular as colisões que ocorrem entre as partículas e entre as partículas e as bordas, durante um determinado período de tempo, e com isso determinar o estado do sistema ao final deste período.

As partículas possuem velocidades constantes. Assume-se que as colisões entre duas partículas e entre uma partícula e uma borda são perfeitamente elásticas, isto é, não há perda de energia cinética. Ao sofrerem uma colisão as partículas têm suas velocidades alteradas.

Sejam duas partículas, p_0 e p_1, com o mesmo raio r, em um determinado estado do sistema referente a um instante de tempo t_i, no qual p_0 e p_1 possuem posições $\vec{s}_{0i}$ e $\vec{s}_{1i}$, e velocidades $\vec{v}_{0i}$ e $\vec{v}_{1i}$, respectivamente, dadas por :

$$\vec{s}_{0i} = (s_{0ix}, s_{0iy}),$$

$$\vec{s}_{1i} = (s_{1ix}, s_{1iy}),$$

$$\vec{v}_{0i} = (v_{0ix}, v_{0iy})$$

e

$$\vec{v}_{1i} = (v_{1ix}, v_{1iy}).$$

A partir deste estado, uma condição necessária e não suficiente para que p_0 e p_1 colidam uma com a outra, é :

$$\vec{s}_{01i} \cdot \vec{v}_{01i} < 0, \tag{5.1}$$

onde $\vec{s}_{01i}$ e $\vec{v}_{01i}$ são a posição e a velocidade relativa de p_0 e p_1, dadas por :

$$\vec{s}_{01i} = \vec{s}_{1i} - \vec{s}_{0i}$$

e

$$\vec{v}_{01i} = \vec{v}_{1i} - \vec{v}_{0i}.$$

Esta condição testa se as trajetórias das partículas p_0 e p_1 estão convergindo.

Dado que a condição 5.1 é satisfeita, é necessário determinar o instante t_f em que as partículas p_0 e p_1 colidem, ou melhor, o intervalo de tempo Δt, tal que

$$\Delta t = t_f - t_i.$$

Como p_0 e p_1 possuem velocidades constantes, as suas posições $\vec{s}_{0f}$ e $\vec{s}_{1f}$ no instante t_f são dadas respectivamente por :

$$\vec{s}_{0f} = \vec{s}_{0i} + \vec{v}_{0i}\Delta t \tag{5.2}$$

e

$$\vec{s}_{1f} = \vec{s}_{1i} + \vec{v}_{1i}\Delta t. \tag{5.3}$$

Subtraindo-se a equação 5.2 da 5.3, obtém-se

$$\vec{s}_{01f} = \vec{s}_{01i} + \vec{v}_{01i}\Delta t, \tag{5.4}$$

que descreve a posição relativa das partículas no instante t_f da colisão. Como neste exato instante p_0 e p_1 estão encostadas, o módulo da posição relativa delas é igual à soma dos raios de p_0 e p_1, isto é, tem-se que

$$|\vec{s}_{01f}| = 2r. \tag{5.5}$$

TE

Substituindo a equação 5.4 na 5.5,

$$|\vec{s}_{01i} + \vec{v}_{01i}\Delta t| = 2r.$$

Desenvolvendo esta equação chega-se a

$$|\vec{s}_{01i}|^2 + 2(\vec{s}_{01i} \cdot \vec{v}_{01i})\Delta t + |\vec{v}_{01i}|^2\Delta t^2 = 4r^2,$$

que pode ser reescrita sob a forma $a\,\Delta t^2 + b\,\Delta t + c = 0$ de uma equação do segundo grau da seguinte maneira :

$$|\vec{v}_{01i}|^2\Delta t^2 + 2(\vec{s}_{01i} \cdot \vec{v}_{01i})\Delta t + |\vec{s}_{01i}|^2 - 4r^2 = 0. \tag{5.6}$$

Resolvendo a equação 5.6 obtém-se Δt :

$$\Delta t = \frac{-2(\vec{s}_{01i} \cdot \vec{v}_{01i}) \pm \sqrt{4(\vec{s}_{01i} \cdot \vec{v}_{01i})^2 - 4|\vec{v}_{01i}|^2(|\vec{s}_{01i}|^2 - 4r^2)}}{2|\vec{v}_{01i}|^2} \tag{5.7}$$

É possível que a equação 5.6 resulte em dois valores $\Delta t'$ e $\Delta t''$. Neste caso, o instante correto da colisão será :

$$t_f = t_i + \min(\Delta t', \Delta t'').$$

Por simplicidade, assume-se que as colisões entre duas partículas são sempre frontais, isto é, que o parâmetro de impacto é sempre nulo. Assim, ao se colidirem as partículas p_0 e p_1 trocam de velocidades, isto é, as velocidades $\vec{v}_{0f}$ e $\vec{v}_{1f}$ após o choque de p_0 e p_1 respectivamente, são :

$$\vec{v}_{0f} = \vec{v}_{1i}$$

e

$$\vec{v}_{1f} = \vec{v}_{0i}.$$

Supondo que o domínio do sistema de colisão de partículas em duas dimensões é quadrado, ele possuirá quatro bordas, borda esquerda, direita, inferior e superior. Sejam $s_{esq\,x}$, $s_{dir\,x}$, $s_{inf\,y}$ e $s_{sup\,y}$ as coordenadas destas quatro bordas respectivamente.

Dada uma partícula p_0, no instante t_i p_0 possui posição $\vec{s}_{0i} = (s_{0ix}, s_{0iy})$ e velocidade $\vec{v}_{0i} = (v_{0ix}, v_{0iy})$. A partir deste estado é possível que p_0 colida, por exemplo, com a borda direita, se a seguinte condição for satisfeita :

$$v_{0ix} > 0. \tag{5.8}$$

Como p_0 possui velocidade constante, a sua posição $\vec{s}_{0f}$ no instante t_f é dada por :

$$s_{0fx} = s_{0ix} + v_{0ix}\Delta t \tag{5.9}$$

e

$$s_{0fy} = s_{0iy} + v_{0iy}\Delta t,$$

onde $\Delta t = t_f - t_i$.

Se a colisão entre a partícula p_0 e a borda direita realmente ocorrer, no exato instante t_f da colisão, p_0 estará encostada na borda direita, e portanto a coordenada x de sua posição s_{0fx} neste instante será :

$$s_{0fx} = s_{dirx} - r. \tag{5.10}$$

Substituindo a equação 5.10 na 5.9 :

$$s_{dirx} - r = s_{0ix} + v_{0ix}\Delta t,$$

determina-se Δt :

$$\Delta t = \frac{s_{dirx} - r - s_{0ix}}{v_{0ix}}, \tag{5.11}$$

e portanto o instante t_f da colisão.

Para colisões com as três demais bordas, o tratamento é análogo. A condição 5.8 para cada uma das demais bordas é :

$$\begin{aligned} \text{borda esquerda} &: v_{0ix} < 0 \\ \text{borda inferior} &: v_{0iy} < 0 \\ \text{borda superior} &: v_{0iy} > 0. \end{aligned} \tag{5.12}$$

A equação 5.10 para cada uma destas bordas é da forma :

$$\begin{aligned} \text{borda esquerda} &: s_{0fx} = s_{esqx} + r \\ \text{borda inferior} &: s_{0fy} = s_{infy} + r \\ \text{borda superior} &: s_{0fy} = s_{supy} - r. \end{aligned} \tag{5.13}$$

Se a colisão da partícula p_0 com a borda esquerda ou direita realmente ocorrer, a sua velocidade $\vec{v}_{0f}$ após o choque será :

$$v_{0fx} = -v_{0ix}$$

e

$$v_{0fy} = v_{0iy}.$$

Se a colisão ocorrer com a borda inferior ou superior, $\vec{v}_{0f}$ será :

$$v_{0fx} = v_{0ix}$$

e

$$v_{0fy} = -v_{0iy}.$$

Supondo que no instante t_i a condição 5.1 para colisão entre as partículas p_0 e p_1 é verdadeira, a colisão ocorrerá no instante t_f (determinado pela equação 5.7) se não ocorrer nenhuma outra colisão envolvendo p_0 ou p_1 com alguma outra partícula ou alguma borda, em um instante t tal que $t < t_f$. Se existir esta colisão no instante t, a velocidade da partícula p_0 ou p_1 envolvida será modificada neste instante, e portanto o resultado da equação 5.7 não será mais válido.

Da mesma forma, supondo que no instante t_i uma das condições 5.8 e 5.13 para colisão entre a partícula p_0 com alguma borda é verdadeira, a colisão ocorrerá no instante t_f (determinado pela equação 5.11 ou análoga) se não ocorrer nenhuma outra colisão de p_0 com alguma partícula ou alguma outra borda, em um instante t tal que $t < t_f$.

Portanto, para determinar corretamente a seqüência de colisões que ocorrem no sistema de colisão de partículas, é necessário determinar todas as possíveis colisões a acontecer. Dado o estado inicial do sistema, é preciso determinar para cada partícula, se é possível que ela colida com cada outra partícula e com cada borda do domínio. Dentre todas as colisões determinadas, somente aquela de menor tempo de ocorrência certamente acontecerá. As demais poderão ocorrer ou não dependendo da ocorrência de colisões com tempos menores envolvendo aquelas partículas.

Assim, a simulação de um sistema de colisão de partículas tem as seguintes linhas gerais : dado o estado inicial do sistema no instante t_i, determinam-se todas as possíveis colisões entre duas partículas e entre uma partícula e uma borda, que são mantidas em uma estrutura de dados. Com isso é possível determinar a colisão de menor tempo t_c que certamente ocorrerá. Pode-se então determinar o estado do sistema neste instante t_c atualizando a posição de todas as partículas e simular a colisão neste instante, atualizando a velocidade da(s) partícula(s) envolvida(s) na colisão.

Todas as demais colisões com alguma partícula também envolvida nesta colisão anterior certamente não ocorrerão mais. Portanto tais colisões podem ser removidas da estrutura de dados.

Dado este estado atual do sistema no instante t_c, é necessário determinar novamente todas as possíveis colisões que podem ocorrer. No entanto agora basta determinar as novas possíveis colisões com a(s) partícula(s) envolvida(s) na colisão anterior. As demais colisões (que envolvem as demais partículas) continuam válidas e já se encontram na estrutura de dados. É necessário então determinar para cada partícula envolvida na colisão anterior, as suas possíveis colisões com cada outra partícula e com cada borda, e inseri-las na estrutura de dados.

Com isto, é possível determinar a nova colisão de menor tempo t'_c que certamente ocorrerá e o procedimento descrito se repete, até que sejam simuladas todas as colisões com tempo menor que o tempo final da simulação.

A estrutura de dados que armazena as possíveis colisões pode ser implementada por uma matriz bidimensional com uma linha e uma coluna para cada partícula, ou como um vetor (através de um mapeamento), dado que apenas a parte triangular inferior ou superior da matriz seria utilizada. No entanto, apesar de ser necessário determinar as possíveis colisões de cada partícula com cada outra do sistema e com cada borda, uma proporção bem menor de possíveis colisões é obtida. Isto ocorre porque muitos pares de partículas não satisfazem à condição 5.1. Assim, a utilização de uma lista encadeada de colisões ordenada pelo tempo de ocorrência das colisões é bem mais eficiente, tendo sido portanto adotada.

O sistema físico de colisão de partículas foi modelado como um conjunto de processos físicos, através da divisão do domínio em setores. Cada PF corresponde a um setor do domínio e o PL correspondente é responsável pela simulação do comportamento das partículas que estão posicionadas dentro daquele setor. A figura 5.2 mostra um sistema de colisão de partículas em que o domínio foi dividido em quatro setores.

Em cada setor a simulação é realizada da mesma forma descrita anteriormente. É necessário entretanto acrescentar a modelagem da passagem de uma partícula de um setor para outro. Uma colisão entre duas partículas ou entre uma partícula e uma borda do domínio é modelada como um evento oriundo do PL referente ao setor onde ocorre a colisão e destinado a ele mesmo. A passagem de uma partícula

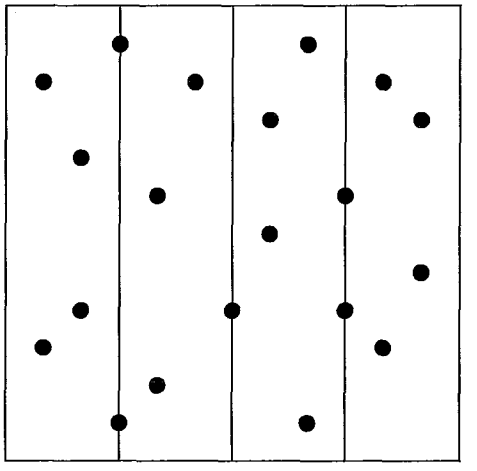


Figura 5.2: Domínio dividido em quatro setores

de um setor para outro é modelada por dois eventos : um oriundo do PL referente ao setor que contém a partícula originalmente e destinado ao PL referente ao setor no qual a partícula está entrando, no instante em que a partícula começa a entrar neste setor; e outro evento oriundo do PL referente ao setor origem da partícula destinado a ele mesmo, para o instante em que a partícula termina de sair deste setor. Estes dois eventos são necessários pois, dado que a partícula possui dimensão, ela pode sofrer uma colisão enquanto estiver na fronteira entre dois setores, isto é, enquanto pertencer aos dois setores. Neste caso, o PL referente ao setor no qual ocorre a colisão envia um evento para o PL referente ao outro setor que também possui a partícula, informando a alteração na velocidade da partícula em conseqüência do choque.

As variáveis de estado de cada PL contêm a identificação das partículas contidas naquele instante no setor correspondente, a posição e a velocidade dessas partículas e a lista de possíveis colisões a ocorrer, que agora armazena também as possíveis saídas de partículas.

Um PL_i referente a um setor s_i se comporta da seguinte maneira : inicialmente PL_i determina todas as possíveis colisões entre as partículas contidas em s_i e destas partículas com as bordas e todas as possíveis saídas de partículas de s_i. PL_i envia então o seu evento fonte para o simulador, que corresponde à colisão ou à saída de menor tempo.

Quando recebe um evento com o tempo t do simulador, que pode ser uma colisão ou a entrada de uma partícula em s_i, PL_i atualiza a posição das partículas em s_i para o tempo t e trata o evento de forma adequada. No primeiro caso, PL_i determina a nova velocidade da(s) partícula(s) envolvidas na colisão, e no segundo, insere aquela partícula no conjunto de partículas de s_i.

Em seguida PL_i determina as possíveis colisões ou saídas da(s) partícula(s) envolvida(s) no evento que acabou de processar. PL_i determina então o próximo evento a ocorrer com alguma partícula de s_i, selecionando a colisão ou saída de menor tempo de sua lista, e envia este evento para o simulador.

O sistema de colisão de partículas foi simulado no simulador distribuído desenvolvido e também em um simulador seqüencial, desenvolvido para a comparação de seu desempenho com o do simulador distribuído. Na simulação seqüencial, o domínio também pode ser dividido em setores.

Esta estratégia de divisão do domínio em setores permite que o processo lógico responsável pela simulação de um setor s_i determine apenas as possíveis colisões de partículas em s_i com outras partículas também contidas em s_i ou com bordas, e não as possíveis colisões entre todas as partículas do sistema.

Caso um domínio com n partículas não seja dividido em setores, o número de possíveis colisões entre duas partículas que devem ser determinadas inicialmente é :

$$\frac{n(n-1)}{2} = \frac{n^2 - n}{2},$$

pois é necessário testar a possível colisão de cada partícula com cada outra do domínio. O número de possíveis colisões entre partículas e bordas que devem ser determinadas inicialmente é $4n$. Portanto o número total de possíveis colisões a serem determinadas é :

$$\frac{n^2 - n}{2} + 4n. \tag{5.14}$$

Se este domínio é dividido em s setores, e supondo que as partículas estão distribuídas espacialmente de maneira uniforme, cada setor possuirá aproximadamente n/s partículas. O número de possíveis colisões entre duas partículas que devem ser determinadas inicialmente em cada setor é :

$$\frac{\frac{n}{s}\left(\frac{n}{s} - 1\right)}{2},$$

pois é necessário testar a possível colisão de cada partícula deste setor com cada outra do mesmo setor. O número de possíveis colisões entre partículas e bordas que

devem ser determinadas inicialmente em cada setor é $4n/s$. Portanto o número total de possíveis colisões a serem determinadas em cada setor é :

$$\frac{\frac{n}{s}\left(\frac{n}{s}-1\right)}{2}+\frac{4n}{s}.$$

Como existem s setores, o número total de possíveis colisões a determinar inicialmente em todos os setores é :

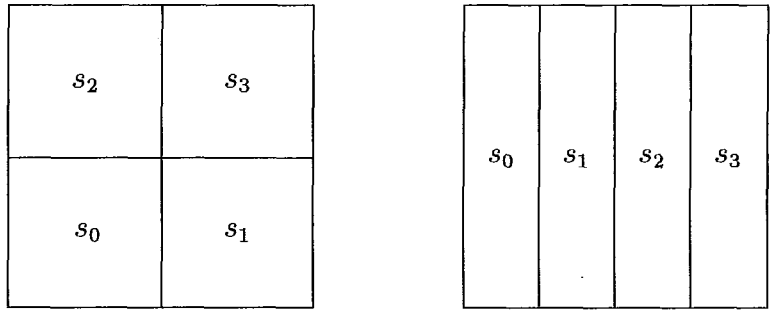
$$s\left(\frac{\frac{n}{s}\left(\frac{n}{s}-1\right)}{2}+\frac{4n}{s}\right)=\frac{\frac{n^2}{s}-n}{2}+4n. \tag{5.15}$$

Embora os números obtidos nas equações 5.14 e 5.15 sejam ambos da ordem de n^2, o número alcançado com a divisão do domínio em setores é sempre menor que o obtido sem a divisão. Com a divisão do domínio em s setores, o número de possíveis colisões entre duas partículas é sempre reduzido aproximadamente na razão de s, enquanto que o número de possíveis colisões entre partículas e bordas mantém-se constante. Para a determinação apenas das possíveis colisões (entre partículas ou entre partícula e borda) com as partículas envolvidas no último evento, o resultado é análogo.

Por este motivo, a simulação seqüencial também pode beneficiar-se em desempenho com a divisão do domínio do sistema de colisão de partículas em setores. Este benefício é alcançado quando o ganho obtido com a redução do número de possíveis colisões a determinar excede o custo do tratamento da passagem de partículas de um setor para outro. Na seção 5.2 esta observação é confirmada através de resultados obtidos.

Na simulação distribuída deste sistema de colisão de partículas o domínio foi dividido de forma a haver um setor (ou um PF) por região do diagrama espaço-temporal e, em conseqüência, por processador do computador paralelo. Note que para um sistema de colisão de partículas em duas dimensões o diagrama espaço-temporal deve ser visto em três dimensões, sendo a figura 5.2 a parte "espacial" do diagrama.

Foram utilizadas duas formas diferentes de dividir o domínio em setores. A figura 5.3 mostra estas duas formas para o caso da divisão do domínio em quatro setores. Com a forma de divisão da figura 5.3(a) com 4 ou 8 setores, cada setor terá de 3 a 5 setores vizinhos, enquanto que com a forma de divisão apresentada em 5.3(b) com 2, 4 ou 8 setores, cada setor terá no máximo dois setores vizinhos.



(a) (b)

Figura 5.3: Formas de divisão do domínio

Conforme já descrito, o paradigma espaço-temporal utiliza a noção de vizinhança em seu algoritmo de relaxação, e portanto as interações (troca de eventos) entre os processos físicos ocorrem apenas entre PFs situados na mesma região ou em regiões vizinhas no diagrama espaço-temporal. Além disso, ao realizar um *rollback*, o simulador de uma região envia mensagens *inf_rb* de propagação de *rollback* para apenas os simuladores das regiões vizinhas à sua região.

Assim, como cada região do diagrama espaço-temporal corresponde a um setor do domínio, a redução do número de setores vizinhos a cada setor parece reduzir consideravelmente o volume do fluxo de mensagens de controle na simulação distribuída.

Além disso, as regiões foram mapeadas na rede de processadores interconectados sob a topologia de um hipercubo de forma que regiões vizinhas (e portanto setores vizinhos) fossem alocadas a processadores vizinhos no hipercubo. Desta forma a troca de eventos e mensagens *inf_rb* ocorre apenas entre processadores que se comunicam diretamente, sem a necessidade de roteamento.

Estas características fizeram com que a divisão do domínio na forma apresentada na figura 5.3(b) fornecesse os melhores resultados, apesar de aumentar a fronteira dos setores em relação à forma da figura 5.3(a). Portanto todos os experimentos apresentados na seção a seguir utilizam este tipo de divisão.

5.2 Resultados e Análise de Desempenho

O sistema de colisão de partículas descrito na seção anterior foi simulado com o objetivo de permitir a análise do desempenho do simulador distribuído. O desempenho deste simulador é avaliado através da sua comparação com o desempenho de um simulador seqüencial e também da medição e análise de diversos parâmetros na simulação distribuída. O objetivo principal é, a partir dos resultados obtidos, concluir o comportamento genérico do simulador distribuído.

Diferentes testes foram realizados, variando-se o número n de partículas no domínio para os valores 25, 50, 75 e 100, enquanto que o tamanho do domínio e o raio das partículas foram mantidos constantes. Para estes testes, as posições das partículas no estado inicial do sistema foram determinadas aleatoriamente de forma que elas ficassem distribuídas de maneira uniforme pelo domínio. Todos os testes gerados foram executados nos simuladores seqüencial e distribuído. Neste último, os testes foram realizados utilizando 2, 4 e 8 processadores, isto é, um hipercubo de grau 1, 2 e 3, respectivamente.

Nas simulações realizadas os estados dos processos lógicos foram gravados a cada evento executado. Os resultados apresentados na literatura mostram que esta é a freqüência de gravação que fornece melhor desempenho. A aplicação de colisão de partículas é bastante custosa em relação à demanda de espaço de armazenamento para a gravação de estados, pois os estados dos PLs são bastante grandes. O estado de um PL engloba a identificação, a posicão e a velocidade de cada partícula incluída no setor que ele simula e a lista de possíveis colisões ou saídas de partículas. Assim, por limitações de memória não foi possível realizar simulações com o número de partículas acima de 100.

O tempo de execução das simulações, isto é, sua duração, foi medido nos simuladores seqüencial e distribuído. Este tempo representa somente o tempo dispendido processando a simulação. O tempo gasto com a leitura dos dados iniciais a partir do disco, assim como com a escrita dos dados finais, não foram considerados. O objetivo desta medida é avaliar somente o desempenho do algoritmo do simulador, independente dos recursos oferecidos para a entrada e a saída de dados e o seu custo.

A figura 5.4 mostra o gráfico do tempo de execução da simulação seqüencial, com a variação do número de setores em que o domínio foi dividido e do número de partículas no domínio. Conforme mencionado na seção 5.1 o desempenho da

simulação seqüencial pode melhorar com a divisão do domínio em setores. Neste gráfico isto pode ser constatado observando-se que o tempo de execução da simulação seqüencial diminui bastante à medida que o número de setores aumenta, para o caso de 100 partículas. Nos demais casos, como o número de partículas é pequeno em relação ao número de setores, o ganho obtido com a redução do número de possíveis colisões a determinar nem sempre compensa o *overhead* com o tratamento da passagem de partículas de um setor para outro. À medida que o domínio é dividido em um número cada vez maior de setores, o número de eventos de entrada e saída de partículas cresce, aumentando assim o número total de eventos a serem processados.

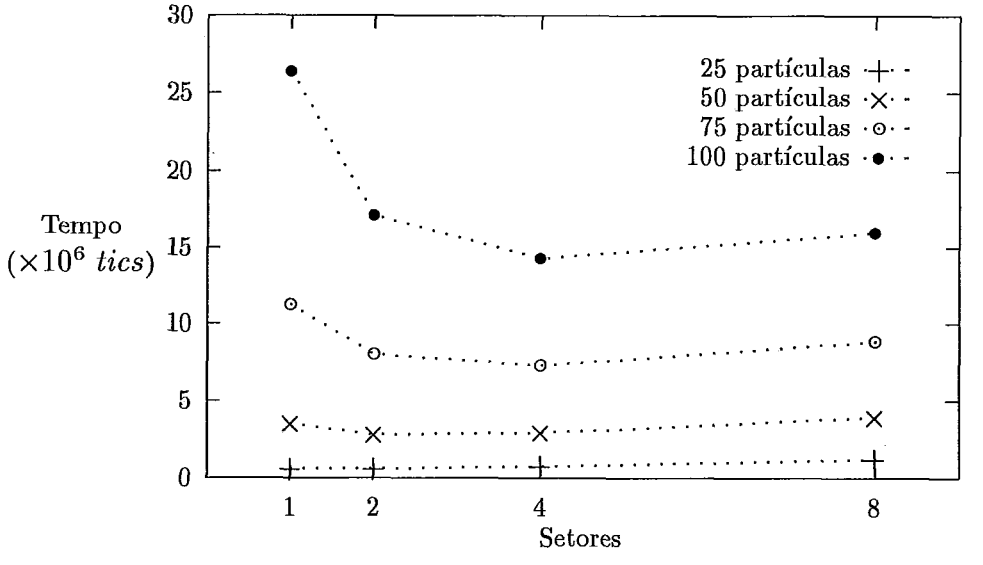


Figura 5.4: Tempo de execução da simulação seqüencial

O gráfico do tempo de execução da simulação distribuída é apresentado na figura 5.5, onde o número de setores corresponde ao número de processadores utilizados. O tempo de execução para 100 partículas em dois processadores requereria, para ser obtido, uma simulação com uso de memória além da disponível, não sendo portanto apresentado.

Da mesma forma que no gráfico da figura 5.4, neste gráfico pode ser observado que o aumento do número de setores, que agora corresponde ao aumento de processadores, é vantajoso apenas quando o número de partículas não é muito pequeno em relação ao número de setores. Na simulação distribuída, o aumento do número

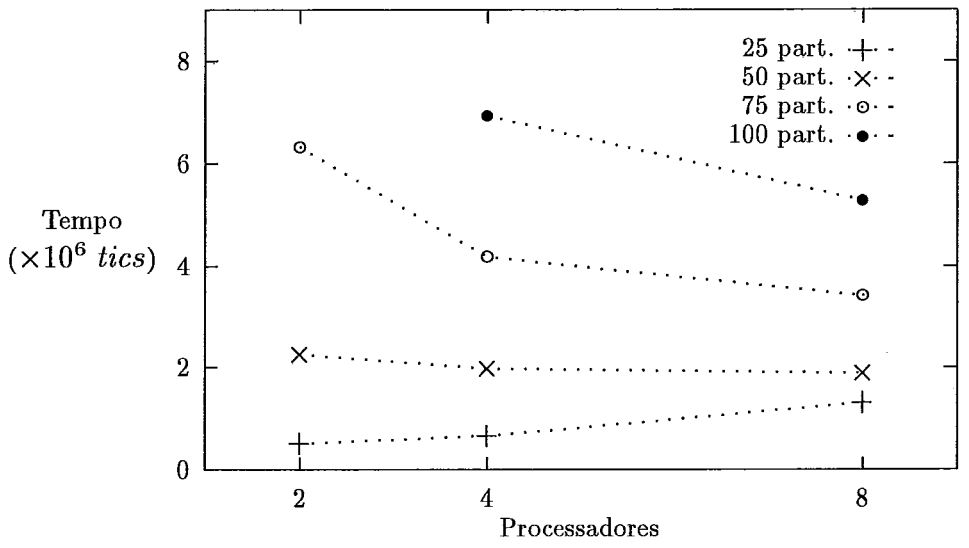


Figura 5.5: Tempo de execução da simulação paralela

de setores, além de causar o aumento do número de eventos de entrada e saída de partículas de um setor para outro, acarreta também um aumento no volume de mensagens enviadas entre os processadores. Como o simulador em cada processador é responsável por apenas um setor do domínio, os eventos de entrada e saída de partículas demandam o envio de mensagens entre processadores, enquanto que os eventos de colisão são eventos de processamento interno a um processador.

Obtidos o tempo de execução das simulações seqüencial e distribuída, uma importante medida de desempenho é o *speedup* alcançado pelo simulador distribuído. O *speedup* é definido da seguinte forma :

$$speedup = \frac{tempo\ de\ execucão\ do\ melhor\ algoritmo\ seqüencial}{tempo\ de\ execucão\ do\ algoritmo\ paralelo},$$

para uma mesma entrada de dados. Supondo que o algoritmo paralelo foi executado utilizando p processadores, o *speedup* ótimo é p. Este resultado corresponde à situação em que a paralelização do algoritmo não introduz nenhum *overhead* como por exemplo a comunicação ou a sincronização entre os processadores.

No cálculo dos *speedups* do simulador distribuído para o sistema de colisão de partículas, a definição de "melhor" algoritmo seqüencial não diz respeito apenas ao algoritmo mais rápido, mas sim a um algoritmo que reproduza os mesmos resultados do algoritmo paralelo. No caso da aplicação específica em questão, há uma grande

sensibilidade numérica, e para isso adotou-se no caso seqüencial a mesma divisão do domínio em setores utilizada no caso paralelo.

O gráfico da figura 5.6 mostra os *speedups* alcançados pelo simulador distribuído em relação ao número de processadores utilizados. Conforme esperado, o *speedup* melhora à medida que o número de partículas (ou seja, o tamanho da entrada do simulador) cresce. Quando este número aumenta, cresce também a densidade de partículas, isto é, o número de partículas por área no domínio. Com isso a proporção de eventos de colisão em relação aos eventos de entrada e saída de partículas aumenta. O melhor *speedup* é o obtido para 100 partículas e oito processadores.

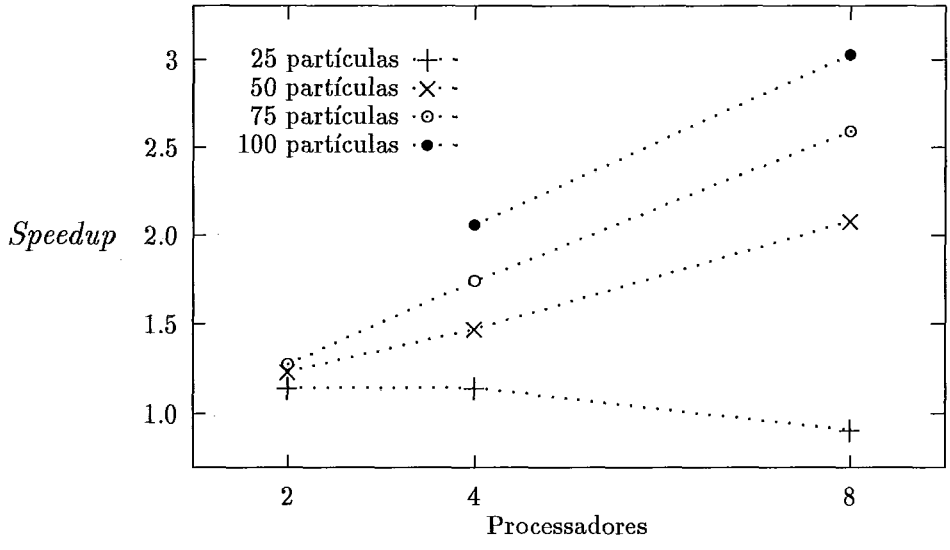


Figura 5.6: *Speedup*

Pelo gráfico é possível concluir que se o número de partículas fosse aumentado, o desempenho do simulador distribuído seria ainda melhor. Da mesma forma, caso o número de processadores utilizados fosse aumentado, por exemplo para 16, provavelmente ainda seria otido um melhor desempenho. Contudo, com o aumento progressivo do número de processadores, certamente um limite seria alcançado, no qual o aumento do número de processadores degradaria o desempenho do simulador distribuído, pois o volume de comunicação excederia o volume de computação interna aos processadores.

Os resultados obtidos podem ser considerados muito bons no contexto da simulação de colisão de partículas. Este problema já havia sido submetido ao método

de simulação distribuída otimista de *time warp* ([B88,H89]) e os *speedups* obtidos foram semelhantes.

Os gráficos das figuras 5.7 e 5.8 apresentam as porcentagens de eventos de colisão e de eventos de entrada e saída (E/S) de partículas, respectivamente, nas simulações distribuídas. Por eventos de E/S entende-se eventos que modelam a passagem de uma partícula de um setor para outro, demandando comunicação entre processadores.

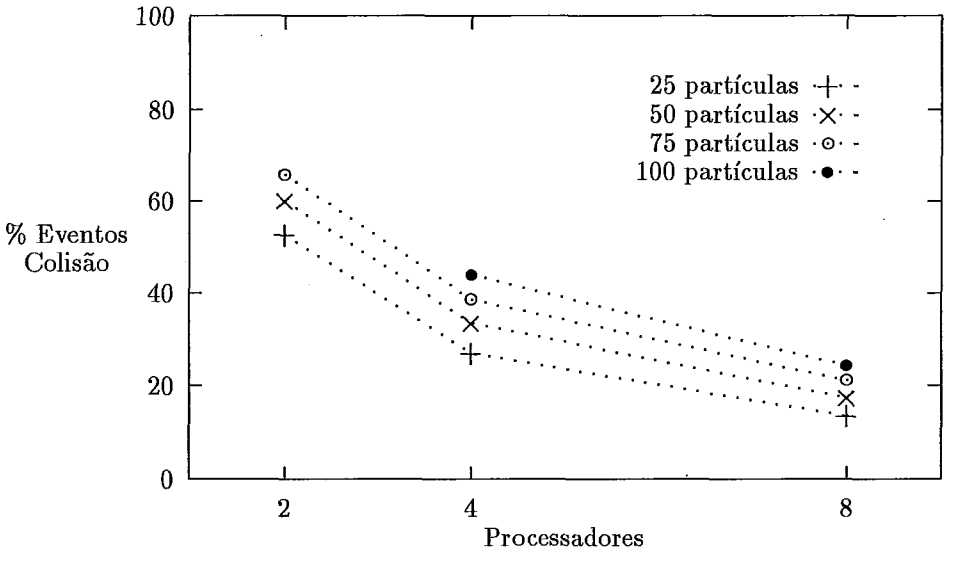


Figura 5.7: % de eventos de colisão

À medida que o número de processadores aumenta (e o número de setores também), o número de eventos de E/S cresce. Como o número de eventos de colisão permanece constante, a porcentagem de eventos de colisão diminui. Como os eventos de colisão são de processamento interno aos processadores, o ideal seria ter a maior proporção possível de eventos de colisão, sem no entanto deixar de explorar todo o paralelismo oferecido, isto é, utilizar todos os processadores do hipercubo.

Em [S87], a taxa R/C de uma computação paralela é definida como a razão entre o tempo de processamento da tarefa (R) e o tempo de comunicação (C). Esta taxa expressa quanto *overhead* é introduzido por unidade de computação. A razão entre a porcentagem de eventos de colisão e a de eventos de E/S pode representar uma taxa R/C da simulação distribuída do sistema de colisão de partículas.

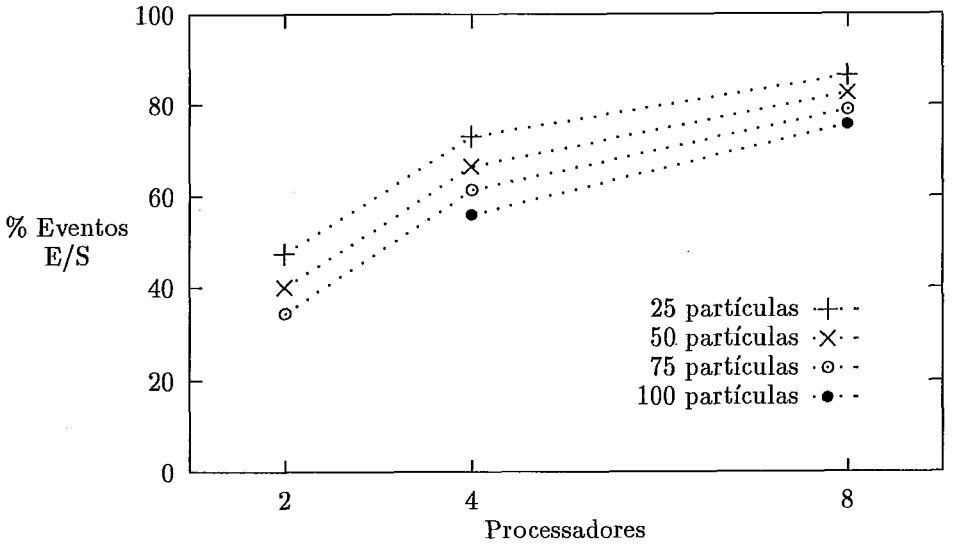


Figura 5.8: % de eventos de entrada/saída

Para um grande número de setores e um pequeno número de partículas, a taxa R/C é muito baixa, pois a porcentagem de eventos de colisão é bem menor que a porcentagem de eventos de E/S. Esta situação pode ser interpretada como uma computação paralela de granularidade bem fina, dado que existem poucas partículas por setor. O tempo de processamento R cresce com o aumento do número de partículas, enquanto que o volume de comunicação C cresce com o aumento do número de setores. O objetivo é obter a maior taxa R/C possível, porém explorando ao máximo o paralelismo oferecido. Por exemplo, caso não sejam utilizados todos os processadores, a taxa R/C aumentará, dado que o volume de comunicação diminuirá. No entanto, o paralelismo disponível não estará sendo totalmente explorado, o que não é desejável. Portanto, o objetivo é encontrar um ponto de compromisso de forma a balancear paralelismo e *overhead* de comunicação.

Por se tratar de um algoritmo de simulação otimista, os processos lógicos podem prosseguir executando e produzindo eventos mesmo que os eventos executados não sejam seguros. Quando um erro de causalidade é detectado, o simulador cancela os eventos enviados prematuramente pelo PL que sofrerá *rollback*. Quando estes eventos são cancelados, eles podem já ter sido processados ou não. Portanto, o número total de eventos produzidos pelos PLs na simulação distribuída otimista é maior que o número total de eventos executados pelos PLs, dado que alguns

eventos são cancelados antes de serem executados. Da mesma forma, o número total de eventos executados pelos *PLs* é maior que o número total de eventos corretos da simulação, pois alguns eventos produzidos e executados prematuramente são cancelados. Estes eventos são gerados pela execução otimista dos *PLs* e não necessariamente correspondem a eventos corretos da simulação da aplicação.

Os gráficos das figuras 5.9, 5.10 e 5.11 mostram, respectivamente, o número de eventos gerados, executados e corretos da simulação distribuída. É óbvio que estes três números crescem com o aumento do número de partículas, pois, conforme descrito na seção 5.1, o número de colisões é aproximadamente proporcional ao quadrado do número de partículas. Os números de eventos gerados, executados e corretos também crescem com o aumento do número de setores, pois a quantidade de eventos de E/S aumenta.

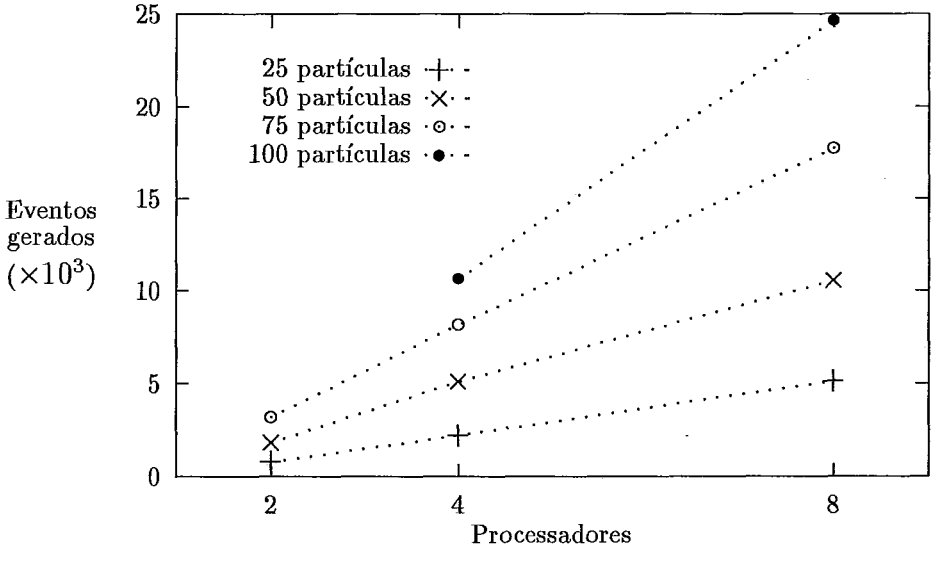


Figura 5.9: Número de eventos gerados

Comparando os três gráficos é possível observar que o número de eventos gerados e executados são muito maiores que o número de eventos realmente corretos. Esta comparação expressa o quão otimista o simulador distribuído foi. Vale lembrar que na simulação conservadora os números de eventos executados e corretos são iguais. O fato do número de eventos executados ser muito maior que o de eventos corretos pode levar à conclusão de que grande parte do tempo de execução da simulação distribuída foi gasto com o processamento de eventos que seriam cance-

lados mais tarde. Contudo, em [F90] Fujimoto argumenta que o tempo que um algoritmo otimista gasta com a execução de eventos que serão cancelados é o mesmo tempo que, em um algoritmo conservador, os PLs gastariam com bloqueios por não possuírem eventos seguros para processar.

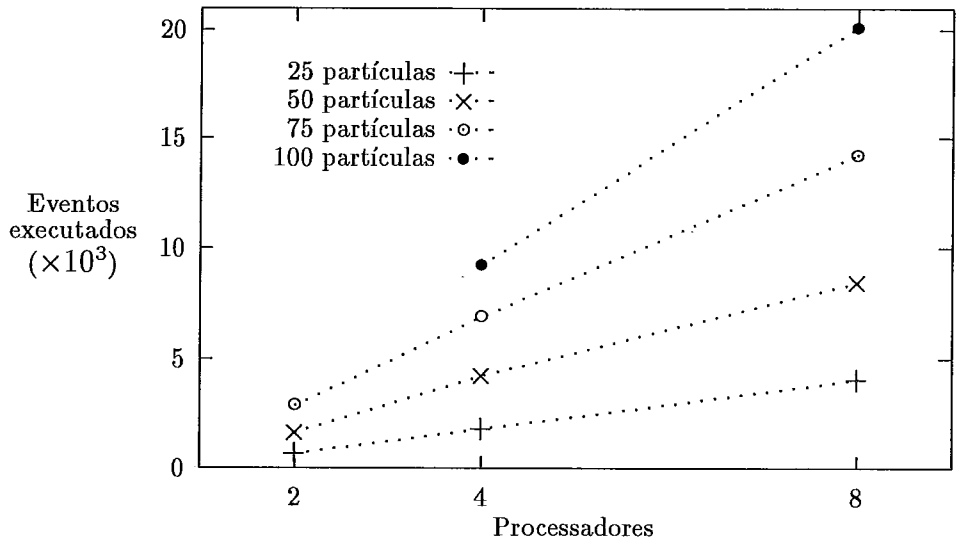


Figura 5.10: Número de eventos executados

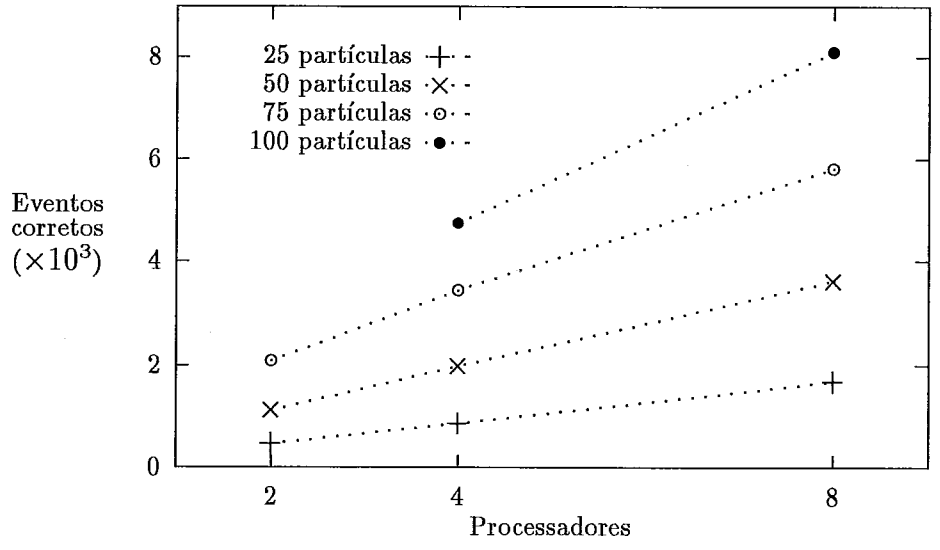


Figura 5.11: Número de eventos corretos

Os gráficos das figuras 5.12 e 5.13 apresentam duas medidas importantes dos algoritmos de simulação distribuída : o número de *rollbacks* realizados pelo simulador e o tamanho médio destes *rollbacks*. O número de *rollbacks* corresponde ao número total de vezes que o mecanismo de *rollback* foi executado, causado pelo recebimento de uma mensagem *inf_rb* ou de um evento atrasado, em todos os processadores.

O número de eventos processados por um processo lógico que foram desfeitos por um *rollback* é o que é denominado de tamanho do *rollback*. O tamanho do *rollback* mede o quanto a simulação incorreta e prematura desfeita pelo *rollback* havia avançado em um PL.

Apesar de o número de *rollbacks* crescer com o aumento do número de processadores (e portanto de setores), o tamanho médio destes *rollbacks* diminui. Isto permite que o *overhead* com *rollbacks* não tenha um impacto negativo tão grande no desempenho do simulador distribuído, à medida que são utilizados mais processadores. De fato, à medida que o número de setores aumenta, o número de eventos de E/S também cresce, proporcionando uma maior e mais freqüente interação entre os setores. Os setores estão sempre recebendo eventos de E/S oriundos de outros setores. Assim, a simulação incorreta e prematura avança pouco até ser desfeita, levando a uma diminuição no tamanho médio dos *rollbacks*. No entanto, justamente por esta interação ser mais freqüente, o número de *rollbacks* aumenta.

Conforme apresentado, quando o simulador detecta um erro de causalidade e executa o mecanismo de *rollback*, ele envia uma mensagem *inf_rb* para cada região vizinha à região que ele simula. Dada a forma com que o domínio do sistema de colisão de partículas foi dividido em setores, cada setor possui no máximo dois setores vizinhos, e portanto cada região possui no máximo duas regiões vizinhas. Assim, o número de mensagens *inf_rb* enviadas em uma simulação distribuída corresponde aproximadamente ao dobro do número de *rollbacks* daquela simulação.

Como o número de *rollbacks* aumenta à medida que o número de processadores cresce, o número de mensagens *inf_rb* enviadas aumenta também. Assim, constata-se mais uma origem para o aumento do volume de comunicação da simulação à medida que o número de processadores cresce (além do número de eventos de E/S). O volume total de comunicação da simulação distribuída é dado pelos eventos de E/S gerados, pelas mensagens *inf_rb* enviadas e ainda pelas mensagens utilizadas pelo mecanismo de detecção de convergência.

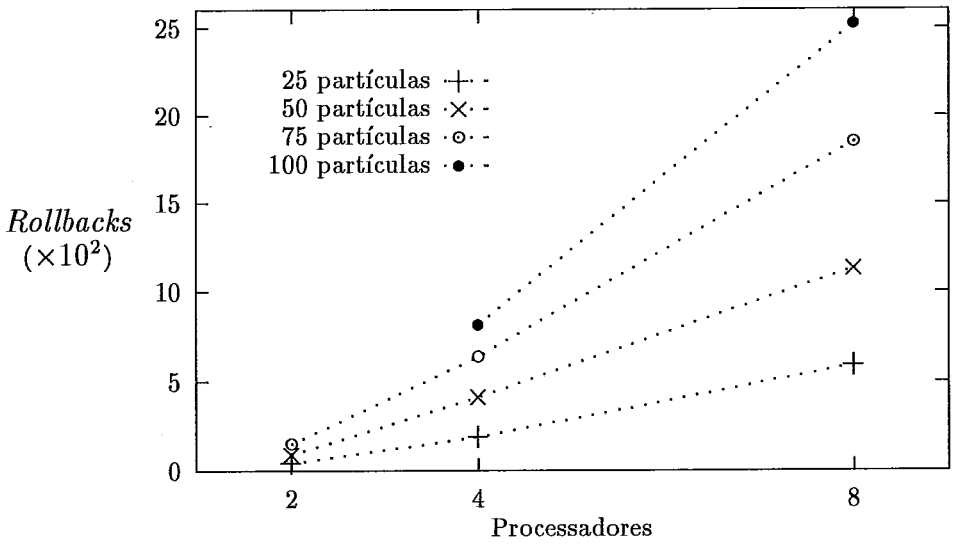


Figura 5.12: Número de *rollbacks*

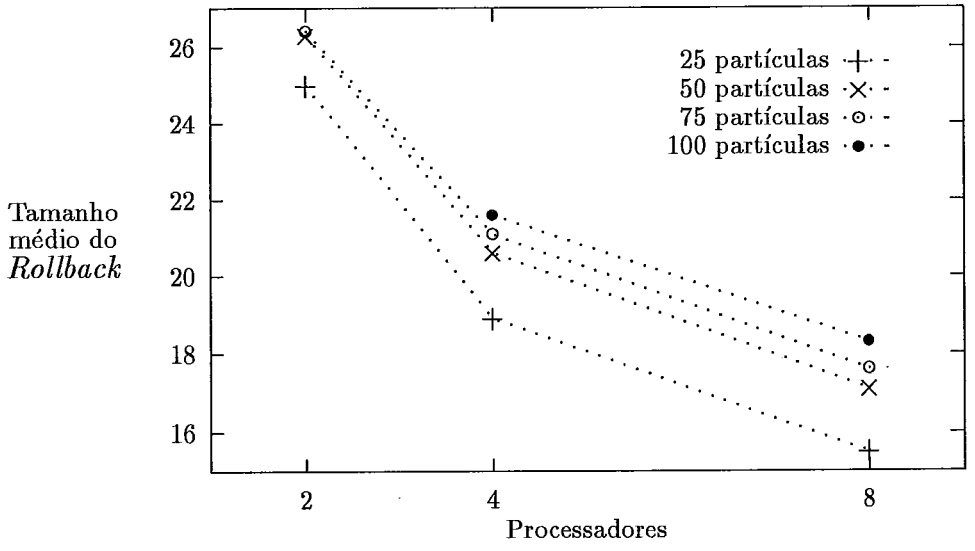


Figura 5.13: Tamanho médio dos *rollbacks*

Como o simulador distribuído é otimista, este volume de comunicação pode ser muito grande, devido a um grande número de eventos de E/S gerados não corretos e de mensagens *inf_rb*. No entanto, nos algoritmos conservadores também há um *overhead* de comunicação muito grande devido às mensagens de controle para a determinação de eventos seguros.

Os gráficos das figuras 5.12 e 5.13 não ilustram muito bem o comportamento do simulador em relação ao número e tamanho médio dos *rollbacks* com o aumento do número de partículas. É claro que quando o número de partículas cresce, o número total de eventos (tanto de colisão quanto de E/S) aumenta. Assim, é esperado que o número e o tamanho médio dos *rollbacks* também cresça, dado que o tempo de execução da simulação aumentou.

Portanto são desejáveis medidas relativas do número de *rollbacks* e do tamanho médio dos *rollbacks* da simulação distribuída. Os gráficos da figura 5.14 e 5.15 apresentam, respectivamente a taxa de *rollbacks* por partícula e a taxa de tamanho dos *rollbacks* por partícula. Com estas medidas relativas, observa-se que a razão entre o número de *rollbacks* e o número de partículas permanece quase constante à medida que o número de partículas cresce, enquanto que a razão entre o tamanho médio dos *rollbacks* e o número de partículas diminui um pouco.

Através de várias medidas apresentadas e principalmente do *speedup*, fica claro que o simulador distribuído apresenta um bom desempenho se o tamanho da entrada de dados da simulação é suficientemente grande. Neste caso, o volume de processamento interno aos processadores excede o volume de comunicação. Nas simulações com entradas pequenas, a quantidade de mensagens *inf_rb* torna-se muito grande em relação ao volume de processamento interno aos processadores. Neste caso, a simulação progride de forma muito lenta, pois o simulador passa grande parte de seu tempo cancelando computações prematuras e reprocessando-as.

Para o sistema de colisão de partículas, este tamanho da entrada corresponde ao número de partículas no domínio. Devido à grande demanda de espaço de armazenamento para os estados dos processos lógicos nesta aplicação, a maior entrada possui 100 partículas. Esta entrada é considerada pequena, pois para a simulação com oito setores cada processador possuirá aproximadamente $100/8 = 12.5$ partículas, o que é uma carga muito baixa.

Para aplicações nas quais os estados dos PLs são muito grandes, o *overhead* com a gravação e recuperação de estados pode prejudicar o desempenho do simulador. Na simulação distribuída, o número de eventos executados corresponde aproximadamente ao número de vezes que um estado foi gravado, dado que os estados são gravados a cada evento executado. O número de *rollbacks* fornece o número de vezes que um estado foi restaurado. Com estas duas medidas é possível dimensionar este *overhead*. Conforme descrito na seção 2.4, uma solução para este problema foi

proposta em [FTG88] e prevê a utilização de um componente de *hardware* dedicado que implemente os mecanismos de gravação e recuperação de estados.

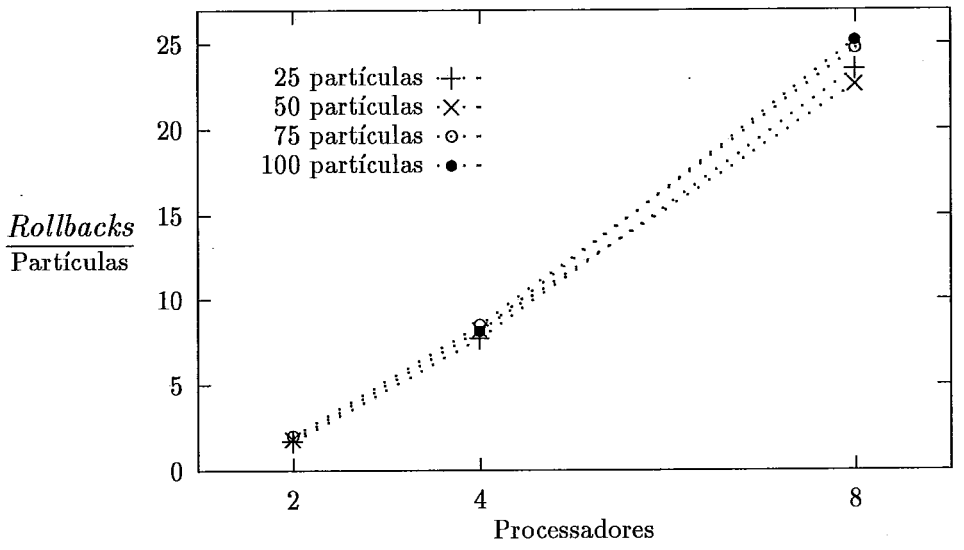


Figura 5.14: Número de *rollbacks* por partícula

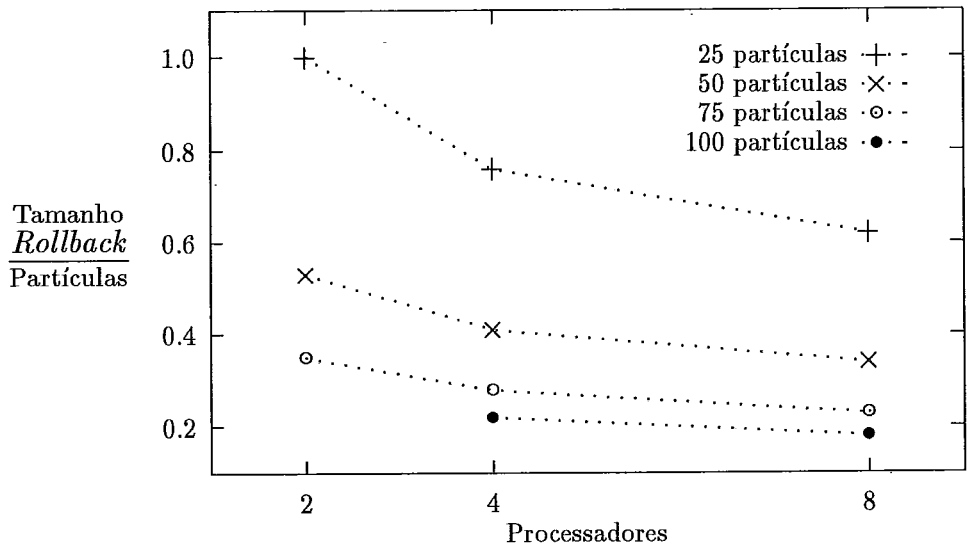


Figura 5.15: Tamanho médio dos *rollbacks* por partícula

Pode ser observado a partir desta análise que o volume de comunicação na simulação distribuída é bastante grande. O fato do processador virtual de comu-

nicação (PVC) ser um processo concorrente ao simulador (ver subseção 4.1.3) prejudica o desempenho do simulador distribuído. O tempo de execução da simulação distribuída será maior pois o PVC ocupa o processador para realizar o gerenciamento da comunicação, atrasando o simulador. Existem multiprocessadores nos quais há um processador real de comunicação em *hardware*, deixando o processador principal dedicado totalmente à computação em questão.

Capítulo 6

Conclusões

Este trabalho apresentou o projeto e a implementação de um simulador distribuído otimista genérico baseado no paradigma espaço-temporal proposto por Chandy e Sherman em [CS89a]. O simulador foi desenvolvido utilizando a linguagem de programação Occam 2 em um hipercubo de *transputers* com oito processadores.

O principal objetivo do trabalho foi oferecer uma ferramenta eficiente que permita o tratamento de uma vasta classe de sistemas físicos, através da simulação distribuída de tais sistemas. Em particular, o simulador distribuído viabiliza o estudo de sistemas físicos mais complexos, cuja simulação consumiria um tempo excessivo de processamento em computadores seqüenciais.

O ponto central do projeto do simulador foi a total separação das funções pertinentes ao simulador daquelas pertinentes à aplicação sendo simulada. Tal separação tornou o simulador genérico, isto é, independente da aplicação, permitindo o tratamento de uma vasta classe de problemas.

Inicialmente neste texto foram apresentados os conceitos básicos da área de simulação e as principais dificuldades da simulação distribuída. Foi realizado um levantamento dos principais algoritmos de simulação distribuída conservadora e otimista encontrados na literatura e do comportamento destes algoritmos. A descrição de alguns destes mecanismos serviu para comparação com o simulador desenvolvido ou para sugestões de otimizações ao simulador.

O paradigma espaço-temporal no qual se baseia o simulador desenvolvido foi descrito. Também foram descritos o ambiente de desenvolvimento deste simulador e o modelo para o desenvolvimento de aplicações que modelem sistemas físicos para

o simulador.

O simulador distribuído foi apresentado através da descrição dos seus principais módulos — camadas de comunicação, de simulação e de aplicação — e dos mecanismos que ele implementa para a realização da simulação distribuída otimista — escalonamento de eventos, *rollback*, gravação e recuperação de estados, preempção, detecção de convergência e recuperação de espaço de armazenamento.

Uma avaliação experimental de desempenho do simulador foi realizada, utilizando como aplicação um sistema físico de colisão de partículas em duas dimensões. Este sistema foi descrito e também foram apresentadas a estratégia utilizada para a sua modelagem como um conjunto de processos físicos e das interações entre eles, e a forma de exploração de paralelismo deste sistema.

Os experimentos realizados com o simulador e os resultados obtidos foram apresentados, juntamente com uma análise com o objetivo de concluir o comportamento geral do simulador distribuído.

Este problema de colisão de partículas é notoriamente difícil em termos de paralelização e ainda não havia sido tratado pelo paradigma espaço-temporal. Os resultados indicaram valores de *speedup* muito bons e compatíveis com os que têm sido obtidos para este problema com outros paradigmas. Além disso, um desempenho ainda melhor seria obtido se fosse possível aumentar o número de partículas no domínio e o número de processadores.

Os experimentos realizados e as medidas apresentadas para a avaliação de desempenho do simulador distribuído permitiram concluir que o simulador apresenta um desempenho muito bom se o tamanho da entrada de dados da simulação é suficientemente grande. Neste caso, o volume de processamento interno aos processadores excede o volume de comunicação entre eles causado pela paralelização.

Nas simulações com entradas de dados pequenas, a quantidade de mensagens para a propagação de *rollbacks* torna-se muito grande em relação ao volume de processamento interno aos processadores. Neste caso, a simulação progride de forma muito lenta, pois o simulador passa grande parte de seu tempo cancelando computações prematuras e reprocessando-as, o que leva a um baixo desempenho do simulador.

Para aplicações nas quais os estados dos processos lógicos são muito grandes,

como é o caso do sistema de colisão de partículas, a simulação distribuída otimista pode ser muito custosa em relação ao espaço de armazenamento. Além disso, o *overhead* com a gravação e a recuperação de estados pode ter um impacto negativo no desempenho do simulador.

O bom desempenho dos algoritmos de simulação distribuída otimista provém da capacidade deles utilizarem ao máximo o tempo de processamento dos processadores. Através da execução de computações apenas possivelmente corretas, evita-se que os processadores fiquem ociosos, mesmo quando a aplicação não oferece condições de *lookahead.*

Este estudo deixou claro que as principais fontes de *overhead* dos algoritmos de simulação distribuída otimista são o grande volume de mensagens enviadas para a propagação de *rollbacks* e o tempo de processamento gasto com o cancelamento e reprocessamento de computações incorretas e com a gravação e recuperação de estados.

Algumas das sugestões para minimizar estas degradações são manter uma carga de processamento interno aos processadores grande, realizar mapeamentos dos processos lógicos nos processadores que forneçam um bom balanceamento de carga e diminuam a necessidade de roteamento e retransmissão de mensagens, e utilizar componentes de *hardware* dedicados para a comunicação e para os mecanismos de gravação e recuperação de estados.

Uma característica atraente do modelo espaço-temporal é que as trocas de eventos se dão apenas entre regiões vizinhas, o que diminui o volume de mensagens enviadas e evita a necessidade de roteamento, se um mapeamento adequado das regiões nos processadores pode ser realizado.

Um interessante trabalho futuro é a utilização da decomposição temporal além da espacial da aplicação, com o objetivo de extrair mais paralelismo da simulação e assim possibilitar um melhor desempenho. Em específico, é desejável o estudo de mecanismos eficientes de correção de estados para uma classe mais abrangente de aplicações.

Outro importante trabalho futuro é a introdução de mecanismos de balanceamento de carga nos algoritmos de simulação distribuída, dado que uma boa distribuição da carga é fundamental para o bom desempenho do simulador. Estes mecanismos podem ser estáticos ou dinâmicos e podem utilizar a decomposição

temporal da aplicação. Especificamente para o balanceamento dinâmico de carga, é necessário o estudo da migração de processos lógicos de um processador para outro, e da divisão de um PL em dois e composição de dois PLs em um único, tanto espacial quanto temporalmente.

Referências Bibliográficas

[ACSC91] Amorim, C. L.; Citro, R.; de Souza, A. F. & Chaves Filho, E. M., *The NCP I Parallel Computer System.* COPPE/UFRJ, Programa de Engenharia de Sistemas e Computação, Relátorio Técnico ES-241/91, 1991.

[BCL91] Bagrodia, R.; Chandy, K. M. & Liao, W. T., "A unifying framework for distributed simulation", *ACM Transactions on Modeling and Computer Simulation* **1** (4), 348-385, 1991.

[BCL92] Bagrodia, R.; Chandy, K. M. & Liao, W. T., "An experimental study on the performance of the space-time simulation algorithm", *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, 159-168, 1992.

[B88] Beckman, B.; DiLoreto, M.; Sturdevant, K.; Hontalas, P.; Van Warren, L.; Blume, L.; Jefferson, D. & Bellenot, S., "Distributed simulation and Time Warp — part 1 : design of colliding pucks", *Proceedings of the SCS Multiconference on Distributed Simulation*, 56-60, 1988.

[B90] Bellenot, S., "Global virtual time algorithms", *Proceedings of the SCS Multiconference on Distributed Simulation*, 122-127, 1990.

[B77] Bryant, R. E., *Simulation of Packet Communications Architecture Computer Systems.* Massachusetts Institute of Technology, MIT/LCS/TR-188, 1977.

[B88a] Burns, A., *Programming in Occam 2.* Addison-Wesley, Wokingham, Inglaterra, 1988.

[CL85] Chandy, K. M. & Lamport, L., "Distributed snapshots : determining global states of distributed systems", *ACM Transactions on Computer Systems* **3** (1), 63-75, 1985.

[CM79] Chandy, K. M. & Misra, J., "Distributed simulation : a case study in design and verification of distributed programs", *IEEE Transaction on Software Enginering* **SE-5** (5), 440-452, 1979.

[CM81] Chandy, K. M. & Misra, J., "Asynchronous distributed simulation via a sequence of parallel computations", *Communications of the ACM* **24** (4), 198-206, 1981.

[CS89a] Chandy, K. M. & Sherman, R., "Space-time and simulation", *Proceedings of the SCS Multiconference on Distributed Simulation*, 53-57, 1989.

[CS89b] Chandy, K. M. & Sherman, R., "The conditional event approach to distributed simulation", *Proceedings of the SCS Multiconference on Distributed Simulation*, 93-99, 1989.

[D90] Drummond, L. M. de A., *Projeto e Implementação de um Processador Virtual de Comunicação.* COPPE/UFRJ, Programa de Engenharia de Sistemas e Computa,c ao, Tese de Mestado, 1990.

[F88] Fujimoto, R. M., "Lookahead in parallel discrete event simulation", *Proceedings of the International Conference on Parallel Processing*, 34-41, 1988.

[F89] Fujimoto, R. M., "Time Warp on a shared memory multiprocessor", *Proceedings of the International Conference on Parallel Processing*, 242-249, 1989.

[F90] Fujimoto, R. M., "Parallel discrete event simulation", *Communications of the ACM* **33** (10), 30-53, 1990.

[FTG88] Fujimoto, R. M.; Tsai, J. & Gopalakrishnan, G., "The roll back chip : hardware support for distributed simulation using Time Warp", *Proceedings of the SCS Multiconference on Distributed Simulation*, 81-86, 1988.

[G88] Gafni, A., "Rollback mechanisms for optimistic distributed simulation system", *Proceedings of the SCS Multiconference on Distributed Simulation*, 61-67, 1988.

[H89] Hontalas, P.; Beckman, B.; DiLoreto, M.; Blume, L.; Reiher, P.; Sturdevant, K.; Van Warren, L.; Wedel, J.; Wieland, F. & Jefferson, D., "Performance of the colliding pucks simulation on the Time Warp operating systems (part 1 : asynchronous behavior and sectoring)", *Proceedings of the SCS Multiconference on Distributed Simulation*, 3-7, 1989.

[I88a] Inmos, *IMS T800 Transputer Preliminary Data Sheet.* Inmos Ltd., 1988.

[I88b] Inmos, *Transputer Development System.* Prentice-Hall, 1988.

[I88c] Inmos, *The Transputer Instruction Set - A Compiler Writers' Guide.* Prentice-Hall, 1988.

[J85] Jefferson, D., "Virtual time", *ACM Transactions on Programming Languages and Systems* **7** (3), 404-425, 1985.

[J87] Jefferson, D.; Beckman, B.; Wieland, F.; Blume, L.; DiLoreto, M.; Hontalas, P.; Laroche, P.; Sturdevant, K.; Tupman, J.; Warren, V.; Wedel, J.; Younger, H. & Bellenot, S., "Distributed simulation and the Time Warp operating system", *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, 77-93, 1987.

[L78] Lamport, L., "Time, clocks, and the ordering of events in a distributed system", *Communications of the ACM* **21** (7), 558-565, 1978.

[LL91] Lin, Y. B. & Lazowska, E. D., "A time-division algorithm for parallel simulation", *ACM Transactions on Modeling and Computer Simulation* **1** (1), 1991.

[L89] Lubachevsky, B. D., "Efficient distributed event-driven simulations of multiple-loop networks", *Communications of the ACM* **32** (1), 111-123, 1989.

[MWM88] Madisetti, V; Walrand, J. & Messerschmitt, D., "Wolf : a rollback algorithm for optimistic distributed simulation systems", *Proceedings of the Winter Simulation Conference*, 296-305, 1988.

[M86] Misra, J., "Distributed discrete-event simulation", *ACM Computing Surveys* **18** (1), 39-65, 1986.

[RBJ91] Reiher, P.; Bellenot, S. & Jefferson, D., "Temporal decomposition of simulations under the Time Warp operating system", *Proceedings of the 5th Workshop on Parallel and Distributed Simulation*, 47-54, 1991.

[RFBJ90] Reiher, P.; Fujimoto, R.; Bellenot, S. & Jefferson, D., "Cancellation strategies in optimistic execution systems", *Proceedings of the SCS Multiconference on Distributed Simulation*, 112-121, 1990.

[SBW88] Sokol, L.; Briscoe, D. & Wieland, A., "MTW : a strategy for scheduling discrete simulation events for concurrent execution", *Proceedings of the SCS Multiconference on Distributed Simulation*, 34-42, 1988.

[S87] Stone, H. S., *High-Performance Computer Architecture.* Addison-Wesley, Reading, MA, 1987.