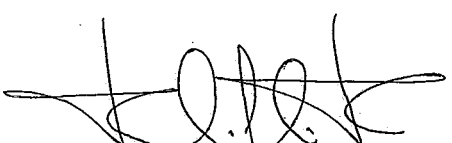
Simulação Distribuída de Sistemas com Evolução Temporal Híbrida

Roseli Suzi Wedemann

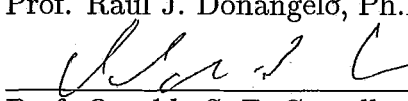
TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

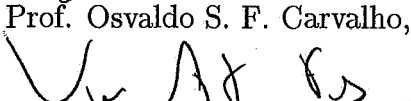


Prof. Felipe M. G. França, Ph.D.
(presidente)

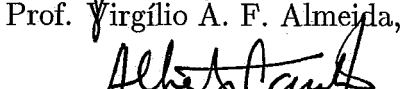
Prof. Raul J. Donangelo, Ph.D.



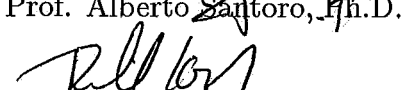
Prof. Osvaldo S. F. Carvalho, Dr. d' État



Prof. Virgílio A. F. Almeida, Ph.D.



Prof. Alberto Santoro, Ph.D.



Prof. Belita Koiller, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
DEZEMBRO DE 1995

Wedemann, Roseli Suzi

Simulação Distribuída de Sistemas com Evolução Temporal Híbrida [Rio de Janeiro] 1995

xviii, 156 p., 29.7 cm, (COPPE/UFRJ, D. Sc., ENGENHARIA DE SISTEMAS E COMPUTAÇÃO, 1995)

TESE – Universidade Federal do Rio de Janeiro, COPPE

1 – Simulação Distribuída 2 – Dirigida por Tempo e Eventos 3 – Otimista 4 – Sistemas Distribuídos

I. COPPE/UFRJ II. Título(Série).

Agradecimentos

Durante os anos em que cursei o doutorado, muita vida foi vivida. Vida vivida com muita gente. Hoje, termino de escrever a tese, com consciência clara de que as pessoas que *conviveram* comigo contribuíram, de alguma forma, em algum momento, para a realização deste trabalho, tornando o processo mais interessante. Nestas páginas de agradecimentos, relato fragmentos da memória desta convivência. A parte científica da tese começa logo em seguida. Aqui ficam lembranças, pedaços da vida, que escrevo com muito prazer, porque é o que eu sinto.

As pessoas mais diretamente responsáveis pela tese são o Valmir e o Raul, os orientadores. O Valmir cumpriu o papel de orientador direitinho. Sempre me cobrou trabalho bem feito, de boa qualidade. Ele me orientou na escolha dos cursos (que foram muitos, porque esta é uma tese em ciência da computação e eu me formei em física). Sugeriu assuntos interessantes e soube avaliar o que é importante.

O Raul, como já tinha exercido esse papel no trabalho de mestrado, me passou as palavras sábias de sua mãe: "Roseli, a gente faz o nosso melhor e depois fica contente com o que foi possível fazer." Ele também sugeriu questões interessantes a serem estudadas, discutiu, esclareceu dúvidas e se preocupou com o andamento do trabalho.

Os dois têm várias características em comum. Como mestres, não se

preocuparam tanto em me ensinar a fazer as coisas. Eles me incentivaram a buscar problemas, questões ainda não abordadas, a identificar o que era importante e ainda não tinha sido feito. Depois, buscar soluções, padrões. Ajudaram-me a ter iniciativa própria e independência. Confiaram na minha capacidade.

Muito importante para mim é o fato de que os dois entendem que a vida, durante um doutorado, não tem só doutorado. Várias vezes aconteceram coisas que não eram doutorado e me exigiram um gasto de energia bastante grande. Em todas estas situações, os dois tiveram bastante sensibilidade e puderam me apoiar. Sem esta compreensão nestes vários momentos, eu talvez tivesse abandonado o curso e esta tese não teria sido escrita. Tenho grande admiração pelos dois, como cientistas e como pessoas humanas. Tem sido um trio bem sucedido.

Ainda com contribuições diretas ao trabalho, gostaria de agradecer ao Hartmut Schultz pelas sugestões de bibliografia e discussões iniciais sobre problemas modelados matematicamente por equações de BUU. Importante também foi o simulador da máquina paralela que usei (o iPSC/860), desenvolvido e cedido, gentilmente, por Loic Prylli da ENS em Lyon, França. O simulador, que roda numa estação de trabalho Sun, facilitou muito o desenvolvimento dos programas. O Ian Thompson e o Laboratório de Daresbury na Inglaterra nos forneceram acesso ao iPSC/860 deste laboratório, o que possibilitou a realização das medidas para a validação do algoritmo.

Pai e mãe sempre marcam muito a gente. Meu pai é uma pessoa muito inteligente que sabe usar muito bem a racionalidade. Sempre foi inconformado, ativo, questionou. Ele me levou a muitas bibliotecas. Numa delas,

muito grande, que tinha uma múmia egípcia, a gente fez as cópias da tese de doutorado dele. Ele tem muitos livros e discos com boa música. A gente conversou, discutiu, analisou, questionou, brigou, concordou e discordou sobre muita coisa. Vivemos crises...

Minha mãe não é tão racional. Ela é muito sensível e intuitiva e dizia para não exagerar na auto-cobrança, para "não esquentar tanto a cabeça com as coisas". Um pouco como a mãe do Raul. Foi exigente para ela também fazer o mestrado, trabalhar e cuidar de quatro filhas e casa. Nossa casa vivia cheia de gente. Nós nos falávamos muito em torno da mesa, durante o lanche, já tarde da noite (todo mundo é bem noturno), comendo um bolo que minha mãe tinha feito. Nossa mesa era freqüentada por muitos amigos interessantes, alunos do pai, meus tios, tias, primos e ali discutíamos religião, filosofia, ética, ciência e a vida alheia também. Tenho três irmãs mais novas a Suzana, a Elen e a Nana, que deram trabalho e se preocupam comigo.

O Nelson conviveu comigo durante muitos anos e pôde continuar a me ajudar, mesmo depois que a convivência não foi mais possível. Sem essa ajuda, teria sido muito difícil terminar a tese. Foi boa a convivência e a ajuda.

Existem também muitos amigos. A Regina sempre viveu intensamente. A Vera (minha tia) conversou muito comigo e me ajudou a lidar com coisas complicadas. O Nelson, além de ser um amigo muito solidário, me emprestou o micro que possibilitou o trabalho de escrita em casa. A Gláucia é pisciana, muito sensível, solidária e irreverente. Ela dá aulas de canto maravilhosas e tem uma atitude muito positiva diante da vida. A Anna é afetiva e solidária. A Ita é muito afetiva e a Ana Lúcia é uma pessoa única. O Marcello tomou

água de coco comigo em Copacabana, foi atencioso e carinhoso. O Adauto soube compreender vários momentos, mesmo sem saber direito o que estava acontecendo, e ser solidário. O Serginho é agitado e se preocupa comigo também. O Rescala é sensível e cuidadoso. A Ondina é minha amiga desde a graduação e é muito afetiva. O encontro com o Alexandre e a Flávia tem sido muito bom.

Na COPPE/Sistemas a Maria Cristina, a Maria Claudia e a Lúcia Maria estudaram muito comigo e são muito amigas. Tem também a Clícia, o Wamberto, a Claudia e o Ricardo. A gente já estudou, comeu, viajou e esquentou muito a cabeça juntos. São todos amigos queridos. O Eliseu me ajudou em diversas situações. O Edil e o Claudio também me orientaram em várias ocasiões e deram cursos interessantes. Tive outros bons professores também, como a Sheila, o Veloso e o Jaime. A Claudia, a Rose e a Ana Paula resolveram muita burocracia na secretaria. O Mota me ajudou pacientemente no Laboratório de Computação Paralela da COPPE.

No Departamento de Física Nuclear, o pessoal me cedeu uma estação de trabalho Sun que eu dominei durante quase três anos. Lá tem o Valmar, que, *no fundo*, é uma pessoa carinhosa, sensível e das mais prestativas que conheço. Ele me ajudou a montar os gráficos com os resultados e também reclama bastante. O Tulião fez a figura 5.1 que ficou belíssima. A Deise, o Marcos, a Ana Maria, o Valmar, o Felipe, o Maurício, a Neide e o Miguel, o Tulião, o Rui, o Itamar, o Hélio, o Kodama, a Marta, e o Bertulani me deram muitas caronas e tornaram as viagens pelo trânsito do Rio de Janeiro mais agradáveis, pelo bom papo. Tem também o Brinati, o Armandinho, o Carlos, o Sérgio, o Lula, o Márcio, o Raphael, o Paulinho, a Maria Helena, o

Curt, o Núbio, o Leandro, a Luciana, o Felipe Coelho, o Jojô, o Marechal, o Odair, o Stênio, o César, o Sidney e a Aninha. Durante o dia de trabalho não se fala muita coisa séria, especialmente na sala de computação, onde a gente tenta trabalhar. Os passeios de barco com o Nelson, a Ginette, o Gustavo, o Hugo e a Daniele tornaram o período do final do último verão em que eu trabalhei na tese mais agradáveis. Estas pessoas acompanharam de perto o dia a dia do trabalho e as dificuldades que apareceram pelo caminho. Sou muito muito grata pelo apoio, solidariedade e amizade que cada um pôde me oferecer.

Ainda no Instituto de Física, tem a Luciane que é minha amiga desde a graduação e com quem eu converso muito. Tem também a Dora que eu conheci durante este último ano. Elas também foram solidárias e muito amigas. E muito mais gente que nem dá para citar aqui...

Tem toda a gente do Canto em Canto e a Elza. Com eles, eu tive o prazer de viver o gozo que é cantar, durante dez anos. Foram muitos acordes, harmonias, melodias, fugas, canções, motetos, concertos, palcos, luzes, dissonâncias, viagens, ensaios, discussões, cansaços, aprendizados, contatos, concursos, festivais e técnicas vocais vividos juntos. Muito SOM BOM! Tudo com sensibilidade, expressividade e controle de qualidade, características do trabalho da Elza.

A Helena talvez tenha conhecido comigo o maior espectro das minhas emoções. Ela me ajudou a identificar e encarar meus fantasmas e a fazer coisas importantes e boas para mim. Muitas questões foram bem elaboradas com ela, que é muito boa profissional e uma pessoa sensível. Foi um grande encontro.

A Carolina, a Lali, a Anoca e a Elisoca, a Elisa, a Letícia e o Guilherme, a Ana e o Caio e a Mariana também participaram com espontaneidade e alegria de criança. Alguns nem são mais tão criança.

O José Antônio é o médico gastroenterologista que descobriu as bactérias que me causavam uma gastrite há oito anos e está me tratando. A Graciela é a homeopata que trata disso com homeopatia e cuidado há muitos anos.

E teve muita boa música. Ouvi muitos instrumentistas, orquestras, conjuntos vocais, cantores, cantoras e compositores, enquanto programava, depurava e escrevia trabalhos de qualificação e a tese. Entre eles alguns amigos como o Maurício, que é excelente violonista e gravou ótimos discos de choro. Todas essas horas de som tornaram o trabalho mais agradável.

Por último e com muita importância, gostaria de agradecer os órgãos de financiamento à pesquisa brasileiros, que remuneraram o meu trabalho durante a tese: o CNPq, a CAPES e a FAPERJ. Mesmo com todas as dificuldades encontradas para se financiar pesquisa em países subdesenvolvidos como o nosso, a tese só se realizou porque pude receber este sustento. De outra forma, eu teria que parar o trabalho e procurar emprego. Espero que muitos outros estudantes possam continuar a ter esta oportunidade no futuro.

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

Simulação Distribuída de Sistemas com Evolução Temporal Híbrida

Roseli Suzi Wedemann

dezembro de 1995

Orientadores: Valmir C. Barbosa e Raul J. Donangelo

Programa: Engenharia de Sistemas e Computação

Alguns sistemas físicos precisam ser modelados para simulação de forma que evoluam no tempo dirigidos por tempo e por eventos. Um exemplo é um sistema de partículas em colisão, evoluindo em um campo de potencial que varia dinamicamente. Este tipo de comportamento pode ser observado na modelagem de sistemas físicos reais, tais como colisões nucleares e o transporte de neutrinos em uma explosão de uma supernova.

Neste trabalho, propomos um algoritmo para simular sistemas que progridem em passos de tempo fixos, com uma atualização de variáveis de estado ao final de cada intervalo, tal como a abordagem dirigida por tempo, mas que progridem de acordo com o instante de ocorrência de eventos discretos dentro de cada intervalo, caracterizando uma evolução dirigida por eventos. Dizemos que ele é um algoritmo híbrido, já que sincroniza a computação de acordo com os aspectos de tempo contínuo e eventos discretos do modelo

do sistema físico. O método é otimista, no sentido de que não proíbe que a simulação evolua localmente devido a restrições rígidas de sincronização. Esta evolução otimista é baseada na suposição de que toda a informação disponível localmente está correta. Se a computação e a recepção de mensagens no futuro invalidarem esta suposição, o algoritmo utiliza técnicas de *rollback* para corrigir as estimativas erradas. O algoritmo também é útil como um mecanismo escalável para limitar otimismo em simulações dirigidas por eventos.

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

Distributed Simulation of Systems with a Hybrid Time Evolution Nature

Roseli Suzi Wedemann

December, 1995

Thesis Supervisors: Valmir C. Barbosa and Raul J. Donangelo

Department: Programa de Engenharia de Sistemas e Computação

Some physical systems need to be modeled for simulation so that they evolve in time in a time-driven (time-stepped) and event-driven manner. An example is a system of colliding particles that move in a dynamically changing potential field. Such a behavior can be observed in the modeling of real physical systems, such as nuclear collisions and the transport of neutrinos in a supernova explosion.

We have developed a distributed algorithm for the simulation of systems of this type, which evolve according to time steps of known duration, but also require simulation time to evolve in an event-driven manner within each step, in order to reproduce the occurrence of discrete events. We refer to it as a hybrid simulation algorithm, since it synchronizes the computation according to both the time- and event-driven aspects of the physical system model. It is an optimistic simulation method, in the sense that it does

not prevent the simulation from evolving locally because of strict synchronization requirements. This optimistic evolution is based on the supposition that all locally available information is correct. If this proves wrong by future computation and message passing, then the algorithm relies on rollback mechanisms to correct the wrong estimates. The algorithm is also useful as a scalable mechanism for limiting optimism in pure event-driven simulations.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

A disponibilidade de processadores paralelos de custo relativamente baixo tem oferecido à comunidade científica uma velocidade de processamento cada vez mais alta, com um aumento também na capacidade de armazenamento. Atualmente, a maior dificuldade na utilização de *multicomputadores* (máquinas paralelas com memória distribuída e programadas sem variáveis compartilhadas entre os processadores) está no desenvolvimento de algoritmos que permitam um aproveitamento máximo do desempenho potencial oferecido pelo hardware. Desenvolver, implementar, e testar os algoritmos em problemas reais, típicos da computação científica, torna-se necessário para validar o uso dos métodos e máquinas paralelas, na solução destes problemas.

O desenvolvimento de algoritmos para simulação distribuída é um problema ainda muito discutido e de grande interesse para a comunidade científica. Muitos problemas que não têm solução analítica podem ser modelados para simulação em computador. Também, em casos nos quais a realização de experiências reais é insegura ou muito dispendiosa, a simulação em computador pode fornecer uma boa aproximação para o comportamento do sistema

real. A maior parte desses problemas são computacionalmente intensos e requerem o uso de máquinas de alto desempenho (multicomputadores, multiprocessadores ou máquinas vetoriais). Alguns exemplos são: simulações de sistemas meteorológicos, linhas de montagem, sistemas de dinâmica molecular, sistemas de combate militar, redes de comunicação, sistemas de computação, sistemas econômicos, sistemas de dinâmica de populações e colisões nucleares.

O ponto de partida da simulação feita em computadores é construir um *sistema lógico*, que reproduza o comportamento de um determinado *sistema físico*, cujas características se deseja analisar. Consideraremos sistemas físicos que podem ser modelados como uma rede de subsistemas, aos quais chamaremos de *processos físicos* (*PF*s) e que agem de forma autônoma exceto para interagir com outros *PF*s do sistema. A evolução de cada *PF* é descrita por um conjunto de eventos, onde cada evento estará associado a um instante de ocorrência. Por exemplo, um sistema de computação pode ser modelado como um conjunto de *PF*s independentes, que são os recursos do sistema: CPU, memória, discos, terminais, impressoras, etc. e que interagem entre si para servir os *jobs* que competem pelo uso destes recursos. Os outros exemplos de sistemas físicos que citamos no parágrafo anterior também podem ser modelados desta forma.

Os passos típicos na construção e uso de um programa de simulação são [Mi86]:

1. Ter um sistema físico natural com características conhecidas ou previstas;

2. A partir do sistema natural construir um modelo onde aspectos relevantes para a simulação são mantidos e aspectos irrelevantes são descartados;

3. Construir um programa de simulação que possa ser executado em um computador, ou seja, um sistema lógico;

4. Analisar os resultados da simulação para entender e prever o comportamento do sistema físico real.

O modelo ao qual nos referimos no passo 2 é o que chamaremos de *sistema físico*. Neste trabalho estudaremos métodos para se construir um programa de computador para a simulação de um sistema físico (passo 3).

O sistema lógico que simula o comportamento do sistema físico que se quer estudar é constituído de *processos lógicos* (*PLs*), onde cada *PL* corresponde à parte do programa que simula o *PF* respectivo. Referimo-nos ao *estado* de um *PL*, em um determinado instante de tempo de simulação, como sendo o valor das *variáveis de estado*, que armazenam os valores das grandezas físicas do *PF* respectivo, no tempo real correspondente.

Há vários aspectos a serem considerados na caracterização de métodos de simulação. Um dos mais relevantes tem a ver com o *tempo de simulação*, que pode ser definido como: o valor do relógio do simulador, no qual ocorre qualquer estado abstrato do sistema simulado, correspondente a um estado do sistema real que ocorre em um determinado tempo real, t. O tempo de simulação (tempo no sistema lógico) é então uma abstração de t (tempo no sistema físico). O termo *sincronização* se refere à propriedade do simulador, que garante que os eventos na simulação ocorrem na seqüência correta de

tempo de simulação.

1.1 Simulações Dirigidas por Tempo e por Eventos

Quanto ao tempo de simulação, os métodos podem ser classificados em *dirigidos por tempo* e *dirigidos por eventos*. Em programas de simulação, uma variável *relógio* armazema o instante de tempo até o qual o sistema físico já foi simulado (i.e. o tempo de simulação). A simulação *dirigida por tempo* é aplicada a modelos de sistemas físicos, nos quais as mudanças de estado ocorrem continuamente no tempo e, por isto, é também chamada de *simulação de tempo contínuo*. Neste caso, a simulação progride, incrementando-se o tempo de simulação por uma quantia fixa, que define um intervalo de simulação e atualizando-se o estado do sistema ao final de cada intervalo. O estado de um *PL* local, no final de um determinado intervalo de tempo, é função do estado ao final do intervalo anterior e de toda a informação recebida de *PLs* vizinhos que se referem ao intervalo presente, recebidos antes do final deste intervalo. Um exemplo de sistema físico que pode ser simulado desta maneira é um sistema meteorológico. Em geral, o comportamento destes sistemas contínuos é modelado matematicamente, por sistemas de equações diferenciais parciais acopladas.

Na simulação dirigida por eventos, as mudanças no estado do sistema são efetuadas instantaneamente quando um evento ocorre e o relógio da simulação é atualizado para o valor do tempo de ocorrência do evento. A seqüência dos tempos de simulação (que corresponde à seqüência de tempos

de eventos) é discreta e não decrescente. Por isto, este tipo de simulação é também chamado de simulação de *sistemas de eventos discretos* (SED). Alguns exemplos de SED são: redes de comunicação, sistemas de computação, dinâmica molecular, etc.

No que diz respeito ao sistema lógico correspondente ao sistema físico sendo simulado, o tempo de simulação sempre é discreto. No caso de simulações dirigidas por tempo, sabe-se a priori em que instantes ocorrerão as mudanças de estado do sistema (no final de cada intervalo), ou seja, qual serão os tempos discretos marcados pelo relógio do simulador. Já em simulações dirigidas por eventos, os instantes de ocorrência dos eventos não são conhecidos a priori. No caso geral, o instante de ocorrência do próximo evento só é conhecido quando o evento anterior a ele é executado pois, quando um evento é processado, ele pode gerar e cancelar outros eventos um dos quais pode ser o próximo.

Em SED, toda interação entre processos físicos é modelada como o escalonamento e/ou cancelamento de eventos entre processos lógicos. Veremos mais tarde que em simulação distribuída, essa interação entre *PF*s deve ser efetuada através da troca de mensagens entre os *PL*s respectivos.

Nas simulações seqüenciais de SED, uma estrutura de dados chamada de *lista de eventos* mantém um conjunto de mensagens, cada uma associada a um tempo de simulação. A lista de eventos é um conjunto de pares ordenados (t, e), onde t é o tempo de ocorrência do evento e. Nem todos os eventos da lista serão executados, pois a execução de um evento pode causar o cancelamento e/ou a inclusão de outros eventos na lista. Isto acontece, por exemplo, no caso da simulação de um *PF* associado a um servidor que pode ser in-

Início;
relógio := t_0;
{Inicialmente existe uma entrada na lista para cada
PF que pode gerar eventos.}
lista_de_eventos := $\{(t_i, e_i)$ | o evento e_i será processado em t_i,
a não ser que o PF onde e_i será executado
execute um evento e_j, tal que $t_j < t_i$ e
e_j cancele e_i.$\}$;
Enquanto o critério de término não for satisfeito **Faça**
retire o par (t, e) com menor t da lista;
simule o efeito da execução de e em t;
Se houver modificações na lista de eventos como conseqüência
da execução de e **Então**
atualize a lista (estas modificações terão $t' > t$);
relógio := t;
Fim_Enquanto;
Fim;

Figura 1.1: Algoritmo para simulação seqüencial.

terrompido (preempção). Há uma *relação de dependência* entre os eventos do sistema. Nenhum evento e pode ocorrer antes que todos os eventos, e', dos quais ele depende, tenham ocorrido. Esta exigência mantém a ordem cronológica de ocorrência de eventos do sistema físico, no caso seqüencial. Um sistema lógico simula um sistema físico corretamente se é possível ao sistema lógico reproduzir a ocorrência de eventos do sistema físico de forma exata.

Na figura 1.1, apresentamos o algoritmo seqüencial para a simulação de SED [Mi86]. Na nossa notação, que será usada em todos os algoritmos apresentados neste trabalho, o texto que aparece entre {} é comentário. Misra prova que este algoritmo simula o sistema físico de forma correta em [Mi86].

1.2 Porque Fazer Simulação Distribuída?

A simulação seqüencial só é viável nos casos em que o tamanho da lista de eventos é limitado em relação à velocidade de processamento da máquina. Problemas típicos das diversas áreas de engenharia, ciência de computação, química, física, economia, etc. consomem uma quantidade enorme de tempo de processamento em máquinas seqüenciais. A estrutura de problemas de simulação descrita anteriormente (um sistema físico dividido em vários *PF*s interagindo entre si) é intrinsecamente paralela, o que sugere a utilização de máquinas paralelas de baixo custo e alto desempenho. Embora a simulação distribuída seja um tipo de problema com um grau bastante alto de paralelismo, paradoxalmente na prática ele é bastante difícil de paralelizar. Espera-se obter um conhecimento maior sobre processamento paralelo e distribuído com o estudo de simulações.

Neste trabalho, estudamos métodos para sincronizar um programa de simulação, composto de um conjunto de processos concorrentes, para execução em um computador paralelo. Há outras alternativas de paralelização. Uma delas consiste em processar *funções de suporte* em paralelo com a simulação propriamente dita. Estas funções de suporte incluem entrada e saída de dados, rotinas para tratar a lista de eventos, coleta de dados para estatística e geradores de números aleatórios, entre outras. Esta abordagem oferece um *speedup* limitado quando comparado com os métodos que estudamos [Fu90, Ka87].

A outra forma de paralelização explora a natureza estocástica da maior parte das aplicações de simulação, a qual torna necessária a execução do

mesmo programa muitas vezes (repetições independentes), com diferentes condições iniciais (por exemplo, sementes diferentes para o gerador de números aleatórios). Este método propõe a execução de cada repetição independente (ou grupo de repetições independentes) em um processador distinto e se torna atraente por evitar problemas de sincronização e *deadlock*, além de não introduzir *overhead* de comunicação. Estes problemas todos aparecem na paralelização do programa simulador, como veremos mais adiante. A abordagem é útil se a simulação é altamente estocástica e se deseja fazer um número muito grande de repetições não acopladas entre si, para diminuir a variância, ou seja, se é necessário simular um problema sobre muitos parâmetros iniciais diferentes. Ela apresenta problemas se cada processador não possui memória suficiente para armazenar o simulador inteiro. Também em um ambiente de desenvolvimento de projetos, onde os resultados de uma experiência (repetição) são usados para decidir quais os parâmetros de entrada da experiência seguinte, este método não pode ser utilizado. O desempenho dependerá também da eficácia da alocação das repetições independentes aos processadores.

1.3 Modelo de Processamento Distribuído Assíncrono

Um *sistema distribuído* consiste de um número finito de *PLs* e de *canais lógicos direcionados* que conectam alguns pares de *PLs*. Idealmente, estes *PLs* e a comunicação entre eles são executados em um conjunto de processadores conectados entre si, por uma rede de *canais físicos* muito rápida, que define uma *topologia de interconexão*.

Cada *PL* executa código seqüencial e dois comandos, *send* e *receive*. Em um comando *send*, um *PL* especifica um canal e uma mensagem a ser enviada. A execução do *send* resulta na colocação da mensagem no canal especificado, após o que, o *PL* remetente continua executando seu código. Cada mensagem leva um tempo indeterminado, porém finito, para atingir o *PL* destino. Isto resulta da incapacidade de se determinar os atrasos de transmissão na rede de comunicação de forma exata e impossibilita uma sincronização global dos relógios dos processadores da rede.

Devido a esta última característica de sistemas distribuídos reais, supomos um modelo de processamento distribuído *assíncrono*, isto é, sistemas cuja principal característica é a ausência completa de uma base de tempo comum a todos os processadores que o compõem. Cada processador possui um relógio local, independente dos demais. Em uma rede heterogênea, os processadores e canais podem ser fisicamente diferentes uns dos outros. Toda a comunicação entre processos se dá através da troca de mensagens. Não há variáveis compartilhadas ou processos de controle centrais, o controle é distribuído.

Em um comando *receive*, um *PL* especifica um ou mais canais, através dos quais ele deseja receber uma mensagem. Um *PL* receptor pode ficar bloqueado, até que uma das mensagens pelas quais ele espera tenha chegado por um dos canais de entrada.

Através deste modelo de processamento distribuído assíncrono, podemos implementar a simulação distribuída como descrevemos anteriormente: um conjunto de *PL*s trocando mensagens simula um conjunto de *PF*s interagindo entre si.

1.4 Dificuldades da Simulação Distribuída

Deseja-se construir um sistema lógico simulador, onde as mudanças de estado muito provavelmente não ocorrerão na mesma velocidade em que ocorrem no sistema físico simulado. O sistema lógico será executado em uma máquina, com um ou mais processadores e canais de comunicação que executarão com velocidades arbitrárias. Em outras palavras, deseja-se reproduzir o comportamento de um sistema físico, com o uso de componentes lógicos assíncronos.

No caso de simulação dirigida por tempo, a dificuldade aparece quando se quer avançar o relógio dos processadores para o próximo intervalo. Como os processadores independentes podem saber se já receberam todas as mensagens que deveriam receber no intervalo atual?

Manter a ordem correta de execução dos eventos equivale a respeitar as relações de causalidade do sistema físico. Em simulação dirigida por eventos, a lista de eventos do algoritmo seqüencial é uma estrutura intrinsecamente seqüencial. A execução de um evento no instante t pode causar o cancelamento ou inclusão de outros eventos na lista, a serem processados em instantes $t' > t$, ou ainda mudar variáveis de estado manipuladas durante a execução. Mas nem todos os eventos dependem uns dos outros e o paralelismo se torna vantajoso justamente nos casos em que há independência entre eventos, permitindo a execução destes eventos independentes em paralelo.

A dificuldade reside em como determinar se um evento e pode ser executado, ou seja, como determinar se todos os eventos e' executados por outros *PLs*, dos quais e depende, já foram executados; ou ainda, como saber se e'

afeta e sem de fato simular e'? A chave para a solução do problema consiste em simular toda a interação entre *PF*s como troca de mensagens entre *PL*s e na codificação do tempo físico de ocorrência do evento como parte da mensagem trocada.

Embora a simulação distribuída apresente um alto grau de paralelismo, as relações de causalidade do sistema físico que determinam a ordem na qual os eventos devem ser processados é, em geral, muito complexa e apresenta grande dependência dos dados. Em muitas outras áreas (por exemplo, operações matriciais) a maior parte da estrutura do processamento é conhecida em tempo de compilação. Ao contrário, a natureza dinâmica do comportamento de problemas de simulação distribuída é a principal dificuldade na formulação de uma solução geral. Na tentativa de respeitar as relações de causalidade nos *PL*s, mensagens de sincronização são adicionadas ao simulador, criando problemas de *overhead* de comunicação, *deadlock*, gerência de espaço de armazenamento em memória, além de outros que veremos mais adiante.

1.5 Objetivos

Vários métodos foram propostos para simular sistemas físicos em sistemas de computação paralelos e distribuídos. No entanto, todos os métodos apresentados se destinam a reproduzir o comportamento de sistemas que evoluem no tempo ou dirigidos por tempo ou dirigidos por eventos. Verificamos que existem alguns tipos de aplicações que progridem no tempo de forma híbrida, ou seja, dirigidas por tempo em dadas circunstâncias e dirigidas por eventos

em outras.

Neste trabalho, propomos um algoritmo para simular sistemas que progridem em passos de tempo fixos, com uma atualização de variáveis de estado ao final de cada intervalo, tal como a abordagem dirigida por tempo, mas que progridem de acordo com o instante de ocorrência de eventos discretos dentro de cada intervalo, caracterizando uma evolução temporal dirigida por eventos. Até o momento, não existe na literatura uma proposta para a solução de paralelização de simulações de sistemas que se comportam desta forma híbrida.

Veremos mais adiante que os métodos para simulação distribuída dirigida por eventos podem ser classificados em conservadores e otimistas, dependendo da forma como impõem a sincronização entre os *PLs*. Os métodos conservadores impõem vínculos de sincronização mais rígidos que os otimistas, sendo ineficientes como método de simulação de propósito geral. Os algoritmos otimistas, por sua vez, necessitam de mais recursos como memória e às vezes hardware dedicado, para efetuar a sincronização de forma eficiente e, mesmo assim, a eficiência nem sempre pode ser garantida.

Nosso método permite que o sistema evolua em passos temporais de forma otimista, supondo que toda a informação disponível a um *PL* e seus vizinhos no grafo que representa o sistema físico, ao final de um intervalo, está correta. Se a execução futura mostrar que esta hipótese estava errada, a execução volta para um intervalo de tempo anterior, no qual o estado do sistema ainda é considerado correto (a decisão de progredir para este intervalo é *revogada*) e continua a partir daí novamente na direção de tempo crescente. Este método de *intervalos temporais revogáveis* utiliza um mecanismo de *roll-*

back de estados para desfazer simulação incorreta de intervalos. A simulação dentro de um intervalo pode ser feita usando um método para SED otimista bem estabelecido, como o *Time Warp* ou o algoritmo de Espaço-Tempo.

No momento, a pesquisa em simulação distribuída busca algoritmos e ambientes de simulação que ofereçam ao usuário a possibilidade de executar um conjunto maior de aplicações, com comportamentos dinâmicos de execução cada vez mais variados [St94, Jh94, Ha94, St94a]. Algumas destas abordagens propõem oferecer ambientes onde algoritmos otimistas e conservadores podem ser utilizados, dependendo das necessidades de cada *PL* do sistema. O método de intervalos temporais revogáveis pode também ser utilizado para simulações distribuídas dirigidas por eventos, como uma forma de impor uma sincronização mais rígida a uma simulação otimista, sem no entanto torná-la conservadora. Ele difere dos demais métodos já propostos para fazer este tipo de sincronização por fazê-la de forma distribuída e não criando barreiras de sincronização centralizadas para o sistema inteiro, tornando-o um método verdadeiramente distribuído e, portanto, escalável.

Atualmente, o desenvolvimento de tecnologia de hardware para computação paralela e distribuída é bastante bem dominada. Os multicomputadores já são produzidos industrialmente e já têm um mercado de consumo grande e crescente em estabelecimentos comerciais, além de universidades e instituições de pesquisa científica, onde foram inicialmente desenvolvidos e utilizados. O grande desafio para a difusão do uso de máquinas paralelas como computadores de propósito geral é o desenvolvimento de algoritmos, linguagens e ambientes de programação que permitam um uso eficiente desta tecnologia, com um esforço mínimo para o usuário. Para isto, é necessário ter

um conhecimento grande das aplicações que se beneficiariam do uso destas máquinas. Nosso trabalho é uma contribuição nesta área, pois foi desenvolvido a partir da pesquisa e estudo de aplicações que são computacionalmente intensas e apresentam um alto grau de paralelismo intrínseco.

Verificamos que existe uma distância grande entre a comunidade de cientistas da computação, que desenvolvem as máquinas, os métodos e o software para computação paralela e a comunidade de usuários potenciais (cientistas, engenheiros, indústrias, estabelecimentos comerciais, etc.), que conhecem a estrutura e os desafios dos problemas e sistemas que poderiam se beneficiar desta tecnologia. Na última reunião da *Workshop on Parallel and Distributed Simulation* de 1994, PADS'94 (principal forum internacional de discussão sobre o assunto), a principal questão colocada na mesa redonda sobre o estado atual da área foi conhecer e estudar aplicações (modelos de sistemas) reais, que poderiam ter um ganho de desempenho sensível, ou cuja solução computacional se tornasse possível, com o uso de simulação paralela e distribuída.

Mais especificamente, os sistemas utilizados com mais freqüência para verificar mecanismos de sincronização de simulações distribuídas são sistemas de filas estocásticos [Ch89, Mi86, Fu88, Ma89, Fu90, Ch91, St91], onde a simulação do sistema é repetida muitas vezes independentemente e, no final, é feita uma média sobre as quantidades físicas que se quer estudar. Ora, sabemos que, devido aos custos de comunicação entre *PLs* na paralelização de uma aplicação para execução distribuída, é mais eficiente executar cada uma das repetições independentes do sistema físico inteiro, em um processador independente (supondo que o número de repetições é igual ou maior que o número de processadores disponíveis). Desta forma, o custo de comunicação

é mínimo (apenas a distribuição inicial dos dados e a coleta da informação desejada no final) e o *speedup* é máximo (proporcional ao número de processadores disponíveis). O uso destes sistemas para testar o desempenho obtido com algoritmos para simulação paralela e distribuída é, portanto, um mero exercício acadêmico.

O modelo matemático de descrição de uma colisão nuclear, para o qual detectamos a necessidade de desenvolver o algoritmo para simulação paralela híbrido, também tem um caráter estocástico. No entanto, as repetições estocásticas são acopladas entre si e, como veremos no capítulo 4, a melhor forma de paralelização é dividir o sistema físico em *PLs* e executar todas as repetições referentes a um *PL* em um processador simultaneamente. Desta forma, detectamos uma classe de sistemas físicos para a qual a simulação paralela e distribuída é a melhor forma de obter ganho de desempenho, com o uso de máquinas paralelas. Além de nos fornecer um conhecimento maior sobre características de sistemas físicos que se benefiam de simulação paralela, este estudo nos forneceu dados de desempenho para um sistema físico real e não para uma carga de eventos sintética [Fu90] ou um sistema de filas já citado. Isto nos dá uma avaliação mais convincente do desempenho oferecido pelo método que propomos e da eficiência que se pode obter com a utilização de um multicomputador.

1.6 Organização do Texto

No próximo capítulo, apresentamos alguns dos principais algoritmos para a simulação distribuída dirigida por eventos (ou de eventos discretos). Discu-

timos as principais características destes métodos, assim como as características das aplicações que determinam a escolha de um método específico para cada problema.

No capítulo 3, descrevemos os principais métodos para simulação distribuída dirigida por tempo. Apresentamos os sincronizadores de algoritmos, que se aplicam a este tipo de problema e nos quais nos baseamos para desenvolver parte de nosso algoritmo híbrido.

Apresentamos o algoritmo que desenvolvemos para a simulação distribuída de sistemas com evolução temporal híbrida no capítulo 4. Especificamos o modelo de problema ao qual ele se aplica e apresentamos uma prova analítica de que ele simula o sistema físico corretamente e uma discussão de suas principais propriedades. Neste capítulo, também discutimos outros trabalhos relacionados ao assunto que estamos abordando.

Como não é possível demonstrar a eficiência do método analiticamente, no capítulo 5, descrevemos nossa experiência de implementação. Mostramos os resultados de desempenho obtidos para a aplicação que nos motivou inicialmente: uma colisão nuclear modelada matematicamente por uma equação de Boltzmann-Uehling-Uhlenbeck para a função distribuição de partícula única, resolvida pelo método de partículas teste. Finalmente, apresentamos nossas conclusões e análises, assim como possíveis trabalhos futuros, no capítulo 6.

Capítulo 2

Simulação Distribuída Dirigida por Eventos (SDDE)

O procedimento de simulação seqüencial dirigida por eventos consiste em executar eventos, retirando-os de uma lista em ordem cronológica estrita, garantindo que eventos gerados ou cancelados sejam também, por sua vez, executados na ordem correta, para que a simulação reproduza a ordem cronológica de ocorrência de eventos do sistema físico real. Este procedimento é intrinsecamente seqüencial e sua paralelização não é direta. Segundo Chandy, Holmes e Misra [Ch79], "o paralelismo só pode ser aproveitado através da mudança da estrutura da lista de eventos, de forma a reconhecer a independência, além da interdependência, entre os processos sendo simulados".

Em todos os métodos que veremos, o sistema físico é modelado como um grafo direcionado, onde cada nó corresponde a um *PL* que gera, cancela e processa eventos simulando um *PF*. Se um nó PL_i produz eventos que serão processados pelo nó PL_j, então existe um arco direcionado (canal) de PL_i para PL_j. Cada PL_i mantém uma variável $relógio_i$ que armazena o valor do

tempo de simulação até o qual PL_i já evoluiu. Em SDDE, quando um PL_i processa um evento, o valor de $relógio_i$ é instantaneamente avançado para o tempo de ocorrência, ou *timestamp*, deste evento.

Dizemos que um sistema lógico simula corretamente um determinado sistema físico se, para cada interação entre PF_i e PF_j no instante t, uma mensagem m com *timestamp* t, simbolizada por (t, m), é enviada por PL_i a PL_j em algum ponto da simulação e os eventos são processados pelos *PL*s efetuando mudanças de estado, correspondentes àquelas que ocorrem no sistema físico. Um evento (t, e) é uma computação seqüencial determinística comum, executada inteiramente por um PL_i, que envolve uma ou mais das seguintes operações [Je85, Na81]:

1. Recepção de uma ou mais mensagens destinadas a PL_i;
2. Leitura do relógio local;
3. Atualização de variáveis de estado locais;
4. Envio de um número qualquer de mensagens.

Para que uma simulação reproduza o comportamento de um sistema físico de forma correta, nenhum evento pode ser processado, antes que todos os eventos dos quais ele depende já tenham sido processados. Em termos das mensagens enviadas entre *PL*s para simular a interação entre *PF*s (alterações de estado, escalonamentos ou cancelamentos de eventos), este vínculo de causalidade implica em que uma mensagem enviada por um *PL* no instante t seja uma função do seu estado inicial e de todas as mensagens que ele recebeu

até t inclusive. Em outras palavras, é necessário que cada PL processe eventos em ordem não decrescente de *timestamps*.

Os métodos de SDDE podem ser classificados em *conservadores* e *otimistas*. Em métodos conservadores, as relações de causalidade *nunca* são violadas. Um PL só pode executar um evento escalonado para o instante t, quando ele tem absoluta certeza de que não receberá mensagem alguma com *timestamp* $t' < t$, ou seja, antes que todos os eventos do sistema com $t' < t$ já tenham sido executados. Já os métodos otimistas se baseiam na detecção e recuperação de erros de causalidade. Os PLs continuam executando eventos, sempre supondo que não receberão mensagens com *timestamps* anteriores ao valor do relógio local. Se isto acontecer, o método se recupera de todos os efeitos do erro e passa a executar a partir do último instante no qual a ordem temporal local está correta.

É importante notar que um retrato do sistema simulado, em um determinado instante de tempo real, não necessariamente corresponde a um retrato possível do sistema físico. Isto se deve ao fato de que os PLs não estão sincronizados, ou seja, cada um se encontra em um tempo de simulação diferente.

No restante deste capítulo descreveremos alguns dos principais métodos conservadores e otimistas, os conceitos nos quais eles se baseiam, bem como algumas análises de desempenho e comparações. Um relato mais completo dos algoritmos para SDDE propostos na literatura pode ser encontrado em [Fu90, We91].

2.1 Métodos Conservadores

Historicamente, os primeiros métodos de SDDE foram conservadores. O problema a ser resolvido por estes métodos é determinar se um evento (t, e) é *seguro* para execução, ou seja, garantir que todos os eventos (t', e') tais que $t' < t$ do mesmo PL já foram executados e que nenhuma mensagem com $t' < t$ será recebida no futuro. Processos que não têm pelo menos um evento seguro ficam bloqueados, podendo gerar situações de *deadlock*. Apresentaremos na próxima subseção um dos primeiros algoritmos conservadores, que ilustra muito bem as características, vantagens e limitações comuns a estes métodos. Outros algorimtos importantes podem ser encontrados em [Pe79, Lu89, Pr88, Ch89].

2.1.1 Algoritmos de Chandy e Misra

Chandy e Misra [Ch79, Ch79a, Ch81, Mi86], e na mesma época, mas de forma independente, Bryant [Br77], foram os primeiros a desenvolver algoritmos distribuídos para simulação. Nestes algoritmos, os processos e os canais de comunicação são definidos estaticamente, antes do início da simulação. A diferença entre os dois algoritmos básicos de Chandy e Misra reside no tratamento de *deadlock*. Um dos mecanismos evita *deadlock* (impede que ele ocorra), e o outro permite que se chegue a uma situação de *deadlock*, detecta-a e recupera o sistema intercomunicando informação global.

O método usado para garantir que um evento é seguro exige que a seqüência de tempos das mensagens enviadas sobre um canal seja não decrescente. As mensagens recebidas através de cada canal de entrada são armazenadas

em ordem FIFO (*F*irst *I*n *F*irst *O*ut), que também corresponde à ordenação cronológica por *timestamps* das mensagens, por causa da restrição anterior. Isto implica em que, se um PL_i recebe uma mensagem (t, m) de outro PL_j, então PL_i conhece *todas* as mensagens que PF_j enviou para PF_i até t inclusive, porque nenhuma mensagem será recebida no futuro com *timestamp* menor que t.

Há um *relógio* associado a cada canal, cujo valor é definido como a componente t da primeira mensagem na fila do canal se a fila tiver mensagens, ou como a componente temporal da última mensagem recebida através do canal, se a fila estiver vazia. Logo, PL_i conhece todas as mensagens recebidas pelo PF_i respectivo até o instante:

$$T_i = \min_{canais(j,i)} \{t_j\},$$

onde os t_js são os valores dos relógios dos canais incidentes sobre PL_i, e o mínimo é tomado sobre todos os canais de entrada. O valor do relógio de PL_i é definido como sendo igual a T_i. Pela definição de T_i e pela restrição do parágrafo anterior, PL_i pode simular PF_i até T_i, isto é, PL_i pode deduzir todas as mensagens transmitidas por PF_i até o instante T_i, e enviá-las.

O procedimento de cada PL consiste em selecionar o canal com o menor valor de relógio e, se houver uma mensagem na fila deste canal, processá-la. Se o canal selecionado estiver vazio, o processo é bloqueado. Este procedimento garante que cada processo só executa eventos em ordem não decrescente de tempo, respeitando as relações de causalidade locais. O algoritmo apresentado na figura 2.1 descreve os passos básicos executados por cada PL.

Na figura 2.2, há duas situações em que o sistema fica em *deadlock* por

Início;
  $relógio_i := 0$;
  {Todas as mensagens recebidas por PF_i até $relógio_i$
    são conhecidas por PL_i.}
  Enquanto o critério de término não for satisfeito **Faça**
    {Simule PF_i até $relógio_i$.}
    para cada canal de saída, calcule a seqüência de
      mensagens $[(t_1, m_1), (t_2, m_2), \ldots, (t_r, m_r)]$, onde
      $t_1 < t_2 < \cdots < t_r$, enviadas por PF_i através deste
      canal; {Estas mensagens devem ser compatíveis
      com a informação recebida até $relógio_i$.}
    envie cada mensagem em ordem através do canal correto;
    $T_i := relógio_i$;
    Enquanto $T_i = relógio_i$ **Faça**
      {Receba mensagens e atualize $relógio_i$ até que seu
        valor mude.}
      espere para receber mensagens sobre todos os canais
        de entrada;
      ao receber uma mensagem, atualize o estado de PL_i e
        recalcule $relógio_i$, o tempo mínimo sobre todos os
        relógios dos canais de entrada;
    Fim_Enquanto;
  Fim_Enquanto;
Fim;

Figura 2.1: Algoritmo básico de Chandy e Misra.

causa de processos bloqueados. O processo D da figura 2.2-a não pode prosseguir, mesmo tendo uma fila com mensagens a serem processadas, porque não recebeu mensagem sobre o canal com $t_{min} = 5$ e, portanto, não sabe se receberá alguma mensagem antes de $t = 12$, que é o *timestamp* da mensagem recebida através do outro canal. O processo C por sua vez também está bloqueado, por não ter recebido mais mensagens do processo A. Se o processo A não enviar mais mensagens para C, a simulação não terminará.

A figura 2.2-b mostra uma situação na qual o deadlock é conseqüência de um ciclo de filas vazias, onde os tempos dos canais são tais que, cada fila do ciclo espera por uma mensagem de outro canal cuja fila também está vazia. Estas situações ocorrem com maior probabilidade se o número de mensagens não processadas é pequeno quando comparado com o número de canais da rede, ou se os eventos não processados se concentram em parte localizada da rede.

Chandy e Misra [Ch79, Ch79a, Mi86] apresentaram duas maneiras de resolver este problema de *deadlock*. Uma delas evita que situações como as da figura 2.2 ocorram. O segundo método para a resolução de *deadlock* é deixá-lo acontecer, detectá-lo e recuperar o sistema [Ch81, Mi86]. Vários mecanismos foram propostos para a detecção de *deadlock* [Mi86, Di80]. Em ambos os casos, mensagens adicionais circulam pelo sistema para garantir o avanço do sistema simulado no tempo, no caso de evitar o *deadlock*, ou para dectectá-lo e resolvê-lo, no caso em que se permite que ele aconteça.

Vários autores fizeram análises de desempenho dos algoritmos de Chandy e Misra [Fu88, Ha88, Re88]. Os próprios autores relatam experiências para o algoritmo que evita *deadlock*, onde eles atingem um *speedup* linear para uma

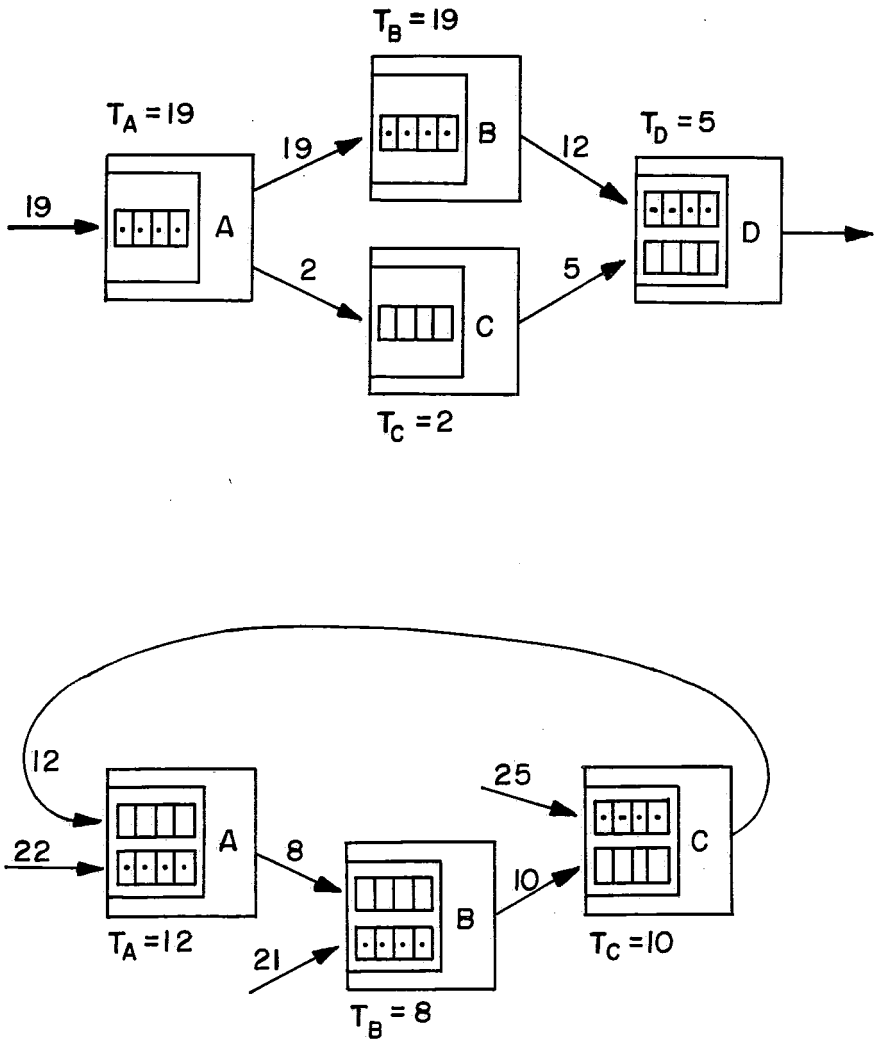


Figura 2.2: Situações em que podem ocorrer *deadlocks* com o uso do algoritmo da figura 2.1.

rede *tandem*, e um *speedup* de aproximadamente 30% do ótimo para uma rede com bifurcação e *merge* usando 6 processadores.

Reed et al. [Re88] implementaram os algoritmos de Chandy e Misra em uma máquina Sequent com memória compartilhada. A implementação usa mecanismos de sincronização que garantem exclusão mútua no uso de memória compartilhada, para implementar comunicação via troca de mensagens, alocação de *PLs* aos processadores e detecção de *deadlock*. Estudos extensos foram feitos para várias topologias de redes diferentes.

A menos de redes sem ciclos, os *speedups* obtidos foram insignificantes (próximos de 1). Os autores chamam a atenção para o fato de que os resultados são extremamente sensíveis à suposições e técnicas de implementação. Sua implementação não supõe nenhum conhecimento da topologia da rede, e previsões sobre o comportamento dos servidores (*lookahead*) limitado, mantendo porém a generalidade da solução. Seus resultados prevêm o comportamento de sistemas cuja estrutura e complexidade proíbem muito conhecimento prévio e, portanto, não permitem o ajuste do mecanismo de simulação subjacente.

Em [Fu88], Fujimoto define *lookahead* da seguinte forma: "se um processo tem conhecimento de todos os eventos que ocorreram até o instante de simulação T, e pode prever todos os eventos que ele gerará com *timestamp* menor ou igual a $T + L$, então diz-se que o processo tem *lookahead* L" (veja também [Ch79a]). Um processo pode escalonar um evento para o futuro, se o *timestamp* deste evento for menor ou igual ao valor atual de seu relógio mais o seu *lookahead*. Mais ainda, um processo com *lookahead* L pode garantir que nenhum evento, a não ser aqueles que ele pode prever, poderá ser gerado

até o instante $relógio_i + L$, possibilitando a outros processos a execução de mensagens de eventos pendentes que eles já tenham recebido.

Fujimoto argumenta e mostra empiricamente em [Fu88] que o uso de look-ahead pode melhorar em muito o desempenho de métodos conservadores. Em seu estudo dos algoritmos de Chandy e Misra, Fujimoto simulou redes de filas usando 16 processadores no BBN Butterfly, com memória compartilhada. Ele reproduziu os resultados de Reed et al. com *lookahead* zero e mostrou como estes resultados podem ser melhorados para sistemas com bom *lookahead*, especialmente o sistema de servidores (mesmo com ciclos) com disciplina de escalonamento FCFS (*F*irst *C*ome, *F*irst *S*erved). Muitos sistemas físicos porém não oferecem bom *lookahead*; um exemplo é uma rede de servidores com mensagens com prioridades e possibilidade de preempção (mesmo que uma mensagem esteja sendo servida, não se pode garantir o tempo do final do serviço, pois outra mensagem poderá chegar e interromper o serviço). Fujimoto então conclui que os algoritmos conservadores só terão um bom desempenho se as propriedades de *lookahead* forem boas, porque "estes algoritmos devem continuamente prever o que *não* acontecerá para continuar executando de forma correta".

Ghosh et al. propõem uma implementação específica do algoritmo que evita *deadlock* [Gh88], para a simulação de modelos que representam componentes digitais complexos de VLSI. Ele relata *speedups* de até 12 com 16 processadores de um hipercubo com 64 processadores do Bell Labs. O modelo usa um valor de *lookahead* correspondente ao tempo real de atraso de propagação do sinal em cada componente.

2.1.2 Críticas a Métodos Conservadores

Em métodos conservadores, o simples fato de que existe a possibilidade de um evento e_1 afetar outro evento e_2 é condição suficiente para que e_1 e e_2 sejam executados seqüencialmente. Se a simulação é tal que e_1 raramente afeta e_2, muito do paralelismo disponível não é aproveitado. De uma forma geral, se a determinação de eventos seguros para execução sempre supõe o pior caso, os métodos conservadores são excessivamente pessimistas forçando execução seqüencial quando ela não é necessária. O paralelismo possível é mal explorado.

A maior parte dos métodos conservadores requerem que muito conhecimento sobre o comportamento do sistema lógico seja previamente fornecido pelo programador. Por exemplo, deve-se fornecer incrementos mínimos de *timestamps* ou tamanhos de janelas temporais [Lu89]. É necessário que o programador da simulação esteja bem familiarizado com o mecanismo de sincronização, para obter bom desempenho. Isto pode levar à geração de código frágil, dificil de manter e modificar.

Outro problema para métodos conservadores é que pequenas mudanças na aplicação, como a introdução de poucas mensagens com prioridade, podem causar uma grande degradação no desempenho. Isto é ruim, porque muitas vezes não se tem muito conhecimento prévio sobre todo o comportamento possível dos sistemas simulados.

É importante notar que os métodos conservadores dependem da existência de boas condições de *lookahead* da aplicação, para ter um bom desempenho [Fu88, Fu90]. Dificilmente, um problema com *lookahead* limitado seria

simulado em paralelo, por um método conservador, apresentando *speedups* satisfatórios.

Além disso, a maior parte dos métodos conservadores requerem uma configuração estática dos processos e canais entre eles. A tentativa de solucionar este problema, criando processos *reserva* e definindo uma rede fortemente conectada, ou seja, com um canal entre todos os pares possíveis de *PL*s, pode causar *overheads* excessivos.

2.2 Métodos Otimistas

Como já foi mencionado, os métodos otimistas não evitam a ocorrência de erros de causalidade. Eles permitem que os erros ocorram, realizam a sua detecção e se recuperam deles, voltando a um estado no passado do processo local que esteja correto. Isto é chamado de *rollback*. Estes métodos *apostam* no fato de que a maior parte dos eventos respeitam um princípio de localidade temporal, ou seja, que a grande maioria das mensagens enviadas serão recebidas no futuro do *PL* receptor, e sua execução não causará erros de causalidade (e *rollbacks*) na maior parte das vezes. Os métodos otimistas, portanto, não envolvem mecanismos de bloqueio de processos. A maior vantagem desta abordagem é que ela explora paralelismo em situações em que um erro de causalidade *é possível*, mas de fato não ocorre. Isto é um ganho em relação aos métodos conservadores. Os métodos otimistas também permitem a criação dinâmica de processos.

2.2.1 O Mecanismo de "Time Warp"

O mecanismo de *Time Warp* se baseia no paradigma de *tempo virtual* e, além de ser um dos primeiros métodos otimistas a ser desenvolvido, é dos mais discutidos na literatura sobre SDDE [Je85]. *Tempo Virtual* é usado como sinônimo daquilo que definimos como tempo de simulação. Como em todos os métodos que vimos até agora, cada *PL* mantém um *relógio* que marca *T*empo *V*irtual *L*ocal (*TVL*). Qualquer par de *PL*s pode se comunicar entre si trocando mensagens, sem a necessidade de se estabelecer previamente os canais de comunicação. Cada mensagem necessariamente contém a seguinte informação:

1. A identidade do *PL* que a envia;
2. O *TVL* do *PL* que a envia no instante do envio, chamado de *T*empo *V*irtual de *E*nvio (*TVE*);
3. A identidade do *PL* receptor;
4. O tempo virtual em que ela deve ser recebida ou *T*empo *V*irtual de *R*ecebimento (*TVR*), que será denominado o *timestamp* da mensagem.

Os sistemas de tempo virtual estão sujeitos a duas regras:

a) O *TVE* de uma mensagem deve ser menor que seu *TVR*;

b) O tempo virtual de cada evento em um processo deve ser menor que o tempo virtual do próximo evento no mesmo processo.

Estas regras implicam em que todas as mensagens enviadas por um *PL* são transmitidas, em ordem não decrescente de *TVE* e todas as mensagens recebidas por um *PL* são executadas em ordem não decrescente de *TVR*.

O algoritmo de *Time Warp* tem dois níveis de controle, um é um mecanismo de controle local e outro um mecanismo de controle global. No mecanismo de controle local, o relógio local do *PL* sempre avança, após a execução de um evento, para o *TVR* da próxima mensagem da fila de entrada do *PL*. A fila de entrada armazena mensagens em ordem não decrescente de *TVR*. Em casos normais, cada *PL* ciclicamente recebe mensagens e executa eventos em ordem não decrescente de *TVL*. Este procedimento continua enquanto o *PL* não recebe uma mensagem com *timestamp* no passado do relógio local. Uma tal mensagem é chamada de *straggler*. Um *PL* detecta um erro de causalidade quando recebe um *straggler*. Uma vez detectado o erro, a recuperação é feita, obrigando o receptor do *straggler* a atrasar o seu relógio local para o valor do *timestamp* do *straggler*, cancelar todos os efeitos da computação errada e continuar executando a partir daí em ordem não decrescente de tempo virtual. Este procedimento é chamado de *rollback*.

Durante a execução de um evento, duas coisas podem acontecer que precisam ser desfeitas em caso de *rollback*. Um evento pode modificar o estado do *PL* no qual ele foi executado. Para fazer um *rollback* de estado é necessário que os estados dos *PLs* sejam armazenados periodicamente. Durante o *rollback*, restaura-se um estado antigo do sistema. É possível também que, durante a execução do evento, uma ou mais mensagens tenham sido enviadas. O cancelamento do envio de uma mensagem é feito enviando-se uma cópia negativa, chamada de *antimensagem*, para o processo que recebeu a

mensagem original, chamada mensagem positiva.

Para implementar o mecanismo de *rollback*, é necessário que os processos tenham a seguinte estrutura:

1. Um nome (identidade) para o processo que seja único no sistema;
2. Uma variável *relógio* que armazena *TVL*;
3. Um estado constituído, em geral, pelo espaço de dados do processo;
4. Uma *fila de estados* que contém cópias de estados recentes do processo;
5. Uma *fila de entrada* que contém todas as mensagens recebidas recentemente, algumas delas já processadas, ordenadas em ordem não decrescente de *TVR*;
6. Uma *fila de saída*, contendo cópias negativas de mensagens enviadas recentemente, mantidas em ordem de *TVE*.

Cada mensagem, correspondendo a eventos da simulação, transmitida de um PL_i para a fila de entrada de outro PL_j, é enviada com um *sinal* positivo. Para cada uma destas mensagens, uma outra com conteúdo idêntico, porém com sinal negativo (a antimensagem) é armazenada na fila de saída de PL_i. As antimensagens são usadas para anular (aniquilar) o efeito da mensagem positiva original em caso de *rollback*.

Quando um *straggler* é recebido, as antimensagens com *TVE* maior que o *timestamp* do *straggler* são enviadas para os *PLs* que receberam as mensagens positivas respectivas. Se uma antimensagem ocorre em uma fila juntamente com sua mensagem correspondente, as duas se anulam e o efeito resultante

é como se o processo nunca tivesse recebido nenhuma das duas. Se um *PL* recebe uma antimensagem, correspondente a uma mensagem positiva que ele já processou, a antimensagem é recebida no passado do relógio local e causa um *rollback* no *PL* receptor, para um tempo anterior ou igual ao do recebimento da mensagem positiva, que não deveria ter sido processada. Se a antimensagem for recebida antes da mensagem positiva correspondente, ela é colocada na fila e é aniquilada assim que a positiva chegue. Procedendo-se desta forma, recursivamente, todos os efeitos da computação errada são desfeitos.

Pode-se mostrar que neste mecanismo o sistema sempre progride, assumindo-se as seguintes condições: cada evento termina normalmente; mensagens enviadas são recebidas em um tempo arbitrário, porém finito; nenhum processo é indefinidamente atrasado por uma política de escalonamento em um processador; e há memória suficiente. Estas condições são necessárias para o funcionamento correto de qualquer um dos algoritmos apresentados neste trabalho.

Uma boa política de escalonamento é sempre escalonar o *PL* com menor *TVL*, em um determinado processador, para execução. Não há possibilidade de *deadlock*, pois não há bloqueio de processos. Também não é possível haver um efeito dominó (muitos *rollbacks* para um passado indefinidamente longínquo dos *PLs*), pois no pior caso todos voltam para o mesmo tempo virtual para o qual o processo original voltou.

Já vimos que, em simulação, o evento não processado com menor *timestamp* no sistema é seguro para execução. No mecanismo de *Time Warp*, define-se como *tempo virtual global*, *TVG*, no instante de tempo real, r, o

valor mínimo entre: (1) todos os *TVL*s dos relógios locais, e (2) os *TVE*s de todas as mensagens que foram enviadas mas ainda não processadas no instante real r. O *TVG* nunca decresce e eventualmente cresce se as condições dos parágrafos anteriores são satisfeitas. A razão para isto é que o *TVG* só poderia decrescer, se algum processo retornasse a um valor de *TVL* inferior ao *TVG*. Basta, então, mostrar que quando o *TVL* de um processo regride, ele o faz para um tempo virtual igual ou posterior ao *TVG*. O *TVL* só regride quando há um *rollback*. No pior caso, todos os processos voltam para o mesmo tempo virtual, T, para o qual o processo que provocou inicialmente o *rollback* retornou. Contudo, T é o *TVR* da mensagem não processada, que por ser posterior ao *TVE* da mesma, é igual ou posterior ao *TVG*, pela definição de *TVG*.

Nenhum evento com *timestamp* menor que *TVG* pode ser desfeito por *rollback*. Logo, o *TVG* serve como um limite de tempo, anterior ao qual toda a computação já feita pode ser considerada definitiva. Só é necessário armazenar um estado anterior ao *TVG*, e conseqüentemente todos os outros anteriores podem ser descartados. Mensagens das filas de entrada e saída com *timestamps* menores que *TVG* podem ser descartadas também. Este processo de destruição de informação mais antiga que *TVG* é chamado de *coleta de fósseis*.

Operações definitivas como as de entrada e saída de dados, detecção de término, e determinação de erros da computação, só são efetuadas quando o *TVG* do sistema for maior que o tempo de ocorrência destes eventos. Em [Je85, Fu90], há sugestões de vários autores que propõem algoritmos para a determinação de *TVG*. A freqüência com que isto deve ser feito depende

de um compromisso: freqüência maior implica em maior tempo de processamento gasto e maior número de mensagens transmitidas na rede para o cálculo de *TVG*, mas por outro lado, resulta em um tempo de resposta melhor e em uma utilização mais eficiente da memória. O cálculo de *TVG* é uma computação distribuída executada concorrentemente com a simulação.

Várias propostas foram feitas no sentido de otimizar o tempo gasto na recuperação de um erro de causalidade como o *cancelamento preguiçoso* [Ga88] e a *reavaliação preguiçosa* [We88, Je87]. Estas variações do mecanismo de *Time Warp* apresentam melhoras no desempenho, para alguns tipos de aplicações com características específicas [Fu90].

Existem alguns modelos analíticos para a avaliação de desempenho do mecanismo de *Time Warp* [La83, Mi84]. Em [Fu90], Fujimoto indica vários outros autores que fizeram este tipo de estudo e descreve alguns deles. Todos os modelos fazem muitas suposições simplificadoras. Por exemplo, alguns custos para armazenar estados e *overhead* de comunicação são desprezados.

Segundo Fujimoto, a suposição mais crítica diz respeito ao custo associado ao *rollback* de um *PL*. Como custo de um *rollback*, entende-se qualquer computação que não é feita durante o progresso normal da execução, na direção de tempo de simulação crescente. Alguns autores de modelos analíticos assumem custo de *rollback* nulo, outros assumem custo constante. Ainda outros assumem um tempo real para *rollback* proporcional à duração do *rollback* em tempo de simulação, ou seja, a quanto o *TVL* foi atrasado.

Observa-se que os modelos podem prever resultados que abrangem desde desempenhos excelentes até desempenhos extremamente pobres (eventual-

mente piores que o algoritmo seqüencial) dependendo do custo atribuído ao *rollback*. Fujimoto propõe que o custo de *rollback* seja proporcional à duração do *rollback* em tempo de simulação, baseado no argumento de que quanto maior a duração do *rollback*, mais antimensagens devem ser enviadas, e que este é o maior *overhead* associado ao *rollback*. Porém, a constante de proporcionalidade deve ser menor do que 1 (em geral, menor que 0.1 de acordo com experiências). Isto porque a simulação na direção de tempo crescente envolve mais computação [1] do que o envio das antimensagens para desfazer as mensagens positivas. Para eventos com granularidade grande, a constante pode ser ordens de grandeza menor que 0.1, o que justificaria o uso de modelos que assumem custo zero para *rollbacks*. Muitos modelos indicam um desempenho para o *Time Warp* igual ou melhor que para os métodos conservadores.

Vários resultados de bom desempenho foram relatados com o uso de *Time Warp* para simular sistemas físicos reais. Jefferson et al. [Be88, Je88, Ho89] relatam *speedups* da ordem de 12 com o uso de 32 processadores, na simulação de um conjunto de discos colidindo elasticamente sobre uma superfície bidimensional. Simulações de cenários de campo de batalha forneceram *speedups* de até 35, com 100 processadores.

Fujimoto também relata experiências [Fu89] em uma máquina BBN Butterfly com memória compartilhada, usando uma versão do *Time Warp* independentemente desenvolvida por ele. As experiências envolvem a simulação de redes de filas fechadas, com interconexões em forma de uma topologia hipercúbica e alcançam *speedups* de até 57 usando 64 processadores. É feita também uma comparação do *Time Warp* com métodos conservadores. Esta

[1] *Overhead* de escalonamento, processamento de mensagens sendo recebidas, armazenamento de estados e a computação da simulaçao propriamente dita

mostra que, diferentemente de métodos conservadores, bom *lookahead* não é condição necessária para se obter bom desempenho com *Time Warp*, embora melhore os resultados consideravelmente. Também foram realizadas experiências que medissem efeitos devidos à distribuição do incremento de *timestamps*, à topologia da rede e granularidade computacional sobre o desempenho. Estas revelam que "o *Time Warp* foi capaz de atingir *speedup* em proporção à quantidade de paralelismo disponível na carga computacional." Estes resultados apoiam o argumento de defensores de métodos otimistas de que "o *Time Warp* pode explorar qualquer paralelismo disponível, sem exigir informações extensas específicas à aplicação do usuário".

Embora estes resultados sejam muito encorajadores, Fujimoto chama a atenção para o fato de que o *overhead* associado ao armazenamento de estados pode causar sensível degradação do desempenho. Ele relata que o desempenho caiu à metade ao aumentar-se o tamanho do vetor de estado de 100 para 2000 bytes, nas simulações de redes de filas mencionadas anteriormente. Uma estratégia possível é diminuir a freqüência na qual os estados são armazenados. A desvantagem desta diminuição seria o fato de que, possivelmente, alguns *rollbacks* teriam que ser feitos para instantes mais longínquos no passado do que o *timestamp* do *straggler*. Como vimos no caso dos estudos analíticos, isto implicaria em um custo maior para *rollback*, o que também degrada o desempenho. Soluções que implicam em um aumento do custo de *rollback* devem ser cuidadosamente pesadas contra os possíveis benefícios ganhos. Jefferson et al. e Fujimoto [Fu90] têm usado a estratégia de armazenar o estado inteiro de um processo após a execução de cada evento.

2.2.2 Janelas Temporais Otimistas

O uso de janelas temporais é um tema emergente na pesquisa de algoritmos de sincronização para simulação paralela e distribuída [Ni92]. Estes protocolos restringem toda a atividade de simulação concorrente a uma janela de tempo, determinada por um mecanismo de sincronização global. Existem protocolos de janelas *conservadores* como aqueles encontrados em [Ch89, Lu87, Lu88, Lu89, St91, Ni93]. Estes algoritmos todos calculam um tempo mínimo, que todos os processadores devem alcançar, antes que qualquer um deles avance para a próxima fase, ou janela. Os eventos dentro da janela podem ser executados em paralelo, sem que isso gere erros de causalidade.

Em métodos *otimistas*, as janelas temporais são usadas para evitar que as computações erradas se propaguem longe demais, na direção de tempos de simulação futuros. O mecanismo chamado MTW (*M*oving *T*ime *W*indows) é apresentado por Sokol, Briscoe e Wieland em [So88, So89]. Nele, escolhe-se um tamanho fixo W para a janela. Somente os eventos com *timestamps* no intervalo $[T, T+W)$, onde T é o menor *timestamp* entre todos os eventos da simulação, podem ser escalonados para execução. Em outras palavras, dois eventos podem ser executados em paralelo se e somente se a diferença entre seus *timestamps* for menor que o tamanho da janela.

A determinação de T é feita em paralelo com a execução dos eventos. Um processo dedicado faz um levantamento entre todos os outros *PL*s. Este algoritmo é centralizado e está descrito informalmente em [So88].

O controle do tamanho da janela limita o assincronismo potencial entre os *PL*s. Como no caso conservador, se o tamanho da janela for muito pequeno,

pouco paralelismo será explorado. Por outro lado, se o tamanho da janela for muito grande, existirá maior probabilidade de ocorrer erros de causalidade. Neste sentido, MTW é um método híbrido, sendo mais ou menos otimista conforme o tamanho da janela.

O principal objetivo do MTW é limitar o tamanho e freqüência de *rollbacks*, e também o volume de informação sobre a história da simulação que precisa ser armazenado, para manter causalidade na execução da simulação.

Permitir que apenas os eventos com *timestamps* pertencentes ao intervalo $[T, T+W)$ sejam executados não garante que estes eventos serão executados em ordem de tempo de simulação correta. Em [So89], os autores apresentam um conjunto de regras que precisam ser impostas ao sistema, para que a simulação respeite os vínculos de causalidade dentro dos limites da janela. Os erros de causalidade (eventos anômalos) serão causados pela chegada de *stragglers* com *timestamps* anteriores ao tempo local do *PL*, porém posteriores ao tempo que marca o início da janela.

Cada *PL* usa o conjunto de regras que dizem respeito às informações que classificam as mensagens, a fim de determinar quais os eventos que podem ser escalonados para execução em paralelo, e quais devem ser executados seqüencialmente. Para tanto, é necessário que algumas informações especiais sobre eventos sejam codificadas nas mensagens. Estas regras preservam as relações de causalidade entre eventos localmente, mas não garantem que os *PL*s nunca recebam *stragglers*. O mecanismo de MTW tenta resolver o erro de causalidade provocado pela chegada de um *straggler*, de várias maneiras. Se o erro não se enquadrar em determinados tipos, faz-se um *rollback* como no *Time Warp*.

Os autores argumentam que métodos otimistas produzem vários tipos de erros temporais, e diferentes técnicas podem ser usadas para corrigir diferentes tipos de erros. Estas outras técnicas visam a redução do número de *rollbacks* efetuados. O MTW escolhe uma estratégia apropriada para resolver cada tipo de erro. Além de *rollback*, o MTW usa *atualização de filas de eventos* e *recuperação de dados de estados passados* para fazer a sincronização. Os autores dão exemplos e argumentam que a aplicação correta deste procedimento pode eliminar um número sensível de *rollbacks* que seriam usados no *Time Warp*.

Os autores relatam uma experiência em [So89], de simulação de campo de batalha, que confirma as seguintes hipóteses. Conforme o tamanho da janela aumenta, o paralelismo aumenta e o tempo de execução diminui. As estratégias de correção temporal alternativas corrigem um grande número de anomalias temporais. Ampliando-se muito o tamanho da janela, há menos chance de poder corrigir anomalias temporais com outra estratégia que não seja *rollback*, ou seja, MTW com janelas muito grandes é equivalente ao *Time Warp*. Por outro lado, conforme o tamanho da janela diminui, maior a freqüência com a qual se deve determinar seu limite inferior e maior o *overhead* associado a este cálculo.

Faltam ainda dados de *speedup* e comparações com *Time Warp*. É importante também analisar o *overhead* associado à determinação do limite inferior da janela, o menor *timestamp* de eventos não processados no sistema. Os autores calculam este limite de forma centralizada, ou seja, há uma sincronização global do sistema para a determinação do próximo limite inferior da janela. As implementações relatadas foram feitas em máquinas

com memória compartilhada e usam variáveis compartilhadas globais para fazer esta estimativa. Esta abordagem não é eficiente em um ambiente distribuído com muitos processadores, ou seja, não é escalável.

Um problema do método está na determinação do tamanho da janela. Outra crítica é que as janelas não distinguem computações corretas de incorretas, e podem, portanto, impedir desnecessariamente o progresso de computações corretas. Além disso, a estratégia do *Time Warp* de escalonar prioritariamente as atividades com menores *timestamps* já inibe o avanço exagerado de alguns processos em relação a outros. A utilidade de MTW é hoje um assunto bastante discutido.

2.2.3 Simulação do Espaço-Tempo

Chandy e Sherman também desenvolveram um método otimista para simulação baseado em técnicas de relaxamento [Ch89a]. No trabalho [Ba91], os autores apresentam uma *teoria de simulação unificada*, que trata tanto simulações dirigidas por tempo como simulações dirigidas por eventos, da qual os algoritmos conhecidos podem ser derivados (inclusive o algoritmo de espaço-tempo). O objetivo de uma simulação é calcular valores de variáveis de estado dos *PLs* que representam o sistema físico, em diferentes instantes de tempo. Por exemplo, a simulação de um sistema de partículas interagindo entre si, em geral, requer a especificação da posição e velocidade de cada partícula em vários instantes diferentes.

Deste ponto de vista, a simulação pode ser encarada como o preenchimento de um gráfico espaço-tempo, onde um eixo representa o tempo de

simulação e o outro representa as variáveis que caracterizam o estado do sistema. Na representação gráfica, cada PF é representado por uma linha de tempo vertical perpendicular ao eixo associado às variáveis de estado (veja a figura 2.3). Seja um ponto (i,t) no gráfico, na altura t da linha de tempo do PF_i. Associa-se a cada ponto (i,t), as mensagens que devem ser recebidas e enviadas pelo PF_i no instante t e os valores das variáveis de estado de PF_i em t. A tarefa do simulador é construir este gráfico, a partir de condições iniciais e de contorno. Este método não diferencia a variável temporal da variável espacial. Inclusive, pode-se evoluir de um ponto temporal qualquer, na direção de tempo crescente ou decrescente.

Na proposta de Chandy e Sherman, o gráfico espaço-tempo é particionado em um número arbitrário de regiões, de formato também arbitrário (veja a figura 2.3). O cálculo, para uma determinada região, depende do cálculo nas regiões vizinhas. A dificuldade reside justamente nesta dependência cíclica. Se PL_i e PL_j representam duas regiões vizinhas, o estado de PL_i depende do estado de PL_j e vice-versa. Os autores propõem o uso de relaxamento para resolver o problema.

Cada processo calcula os valores das varáveis de estado para a sua região, usando *estimativas* dos valores das variáveis das regiões vizinhas. Os PLs continuamente recebem nova informação sobre os estados das regiões vizinhas, calculam valores de variáveis para a sua região e enviam informação sobre seu novo estado para os vizinhos. Para se calcular o comportamento de cada região, pode-se usar simulação seqüencial com a lista de eventos. Se no término da simulação de uma determinada região, a informação de entrada recebida dos vizinhos não corresponde às estimativas usadas no início da

simulação da região, recorre-se a um mecanismo de *rollback* e a região é reexecutada, a partir de um estado anterior correto. A computação termina quando os processos atingem um ponto estável, ou seja, quando os estados dos processos não mais variam enquanto a computação progride. Dizemos que neste ponto a computação convergiu. As estimativas iniciais que cada *PL* faz sobre o comportamento dos vizinhos são arbitrárias, a menos das condições iniciais e de contorno dadas pela especificação do problema.

O diagrama espaço-tempo para um problema qualquer consiste de uma linha de tempo vertical para cada processo físico (veja a figura 2.3). A linha é rotulada com o estado do processo. Uma mensagem enviada de PF_i para PF_j, no instante t, é representada por um arco horizontal na altura t, direcionado da linha de tempo de PF_i para a linha de tempo de PF_j. Um arco vertical, direcionado para uma região, representa o estado de um *PF* em um determinado instante. Os arcos direcionados para fora de uma região são funções dos arcos direcionados para dentro da região. A partir destas definições, usa-se o método de relaxamento descrito anteriormente, onde a informação recebida e enviada entre vizinhos é representada pelos arcos.

Os autores apresentam um algoritmo que detecta convergência para instantes de tempo crescentes, de forma a reduzir o intervalo de tempo de simulação em que se deve simular o sistema durante a execução. Há uma versão síncrona e outra assíncrona para a detecção de convergência. Há também uma prova em [Ch89a] de que a computação sempre converge para os valores corretos das variáveis de estado, se algumas condições são satisfeitas.

Dependendo de como o gráfico de espaço-tempo é particionado e de como se faz o escalonamento de eventos em uma região, o algoritmo de espaço-

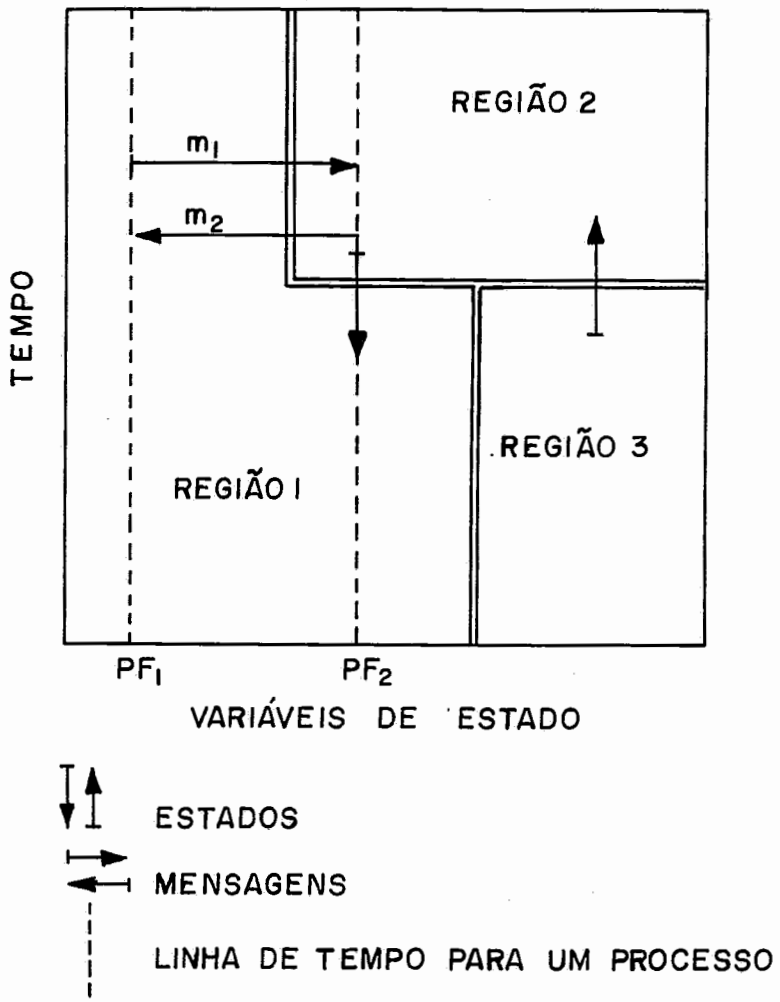


Figura 2.3: Exemplo de partição da região espaço-tempo a ser preenchida pelos *PLs* e de arcos que representam troca de mensagens.

tempo se comporta como os diversos algoritmos que apresentamos (simulação dirigida por tempo, dirigida por eventos seqüencial, métodos paralelos conservadores, *Time Warp*, etc...) [Ba91]. Por exemplo, se dividimos o gráfico de espaço-tempo em tiras verticais, de forma que cada região corresponde à linha de tempo de um *PF* e permitimos um escalonamento de eventos otimista, com um mecanismo de correção de computações erradas, temos o *Time Warp*. Neste, o estado de uma região é uma estimativa do comportamento do gráfico de espaço-tempo, supondo que o fluxo de entrada seja correto. As mensagens novas que continuam chegando, indicam mudanças nas estimativas do comportamento das regiões vizinhas, que fazem com que seja necessário recalcular o estado da região, como em um *rollback*. Se o fluxo de mensagens gerado pelo novo cálculo, for diferente do que tinha sido calculado antes, novas mensagens são enviadas, de forma parecida com o cancelamento por antimensagens.

O algoritmo inova, quando permite que as regiões do gráfico de espaço-tempo sejam divididas, de forma que *PLs* diferentes podem simular o mesmo *PF* em intervalos de tempo diferentes e executados concorrentemente, possibilitando um paralelismo temporal. Esse tipo de paralelismo é eficiente quando os processos físicos apresentam um comportamento cíclico, de forma que se possa fazer boas estimativas das variáveis de estado e das entradas iniciais dos *PLs* que simulam intervalos de tempo diferentes.

Em [Ba91], são fornecidos vários resultados de desempenho para diferentes implementações do mecanismo de espaço-tempo. Dependendo da granularidade dos processos lógicos, da quantidade de memória disponível, do algoritmo de detecção de convergência (síncrono ou assíncrono), do tipo de

paralelismo (espacial ou temporal) e de características da aplicação (por exemplo *lookahead* e comportamento cíclico no caso de paralelismo temporal), foram obtidos *speedups* entre 15% e 88% do número de processadores usados.

2.2.4 Crítica a Métodos Otimistas

Uma questão que logo se coloca sobre métodos otimistas é: será que se consegue evitar que o sistema passe a maior parte de seu tempo executando computação errada e corrigindo-a, às custas de possível computação correta? Isto provavelmente acontecerá se a aplicação apresentar pouco paralelismo, em relação ao número de processadores disponível. Fujimoto argumenta [Fu90] que este comportamento não deve ser comum, porque o mecanismo de escalonamento do *Time Warp*, que dá prioridade para eventos com *timestamps* menores, inibe a propagação de erros para um futuro muito distante, em detrimento de computações corretas.

Um problema muito sério do *Time Warp*, segundo Fujimoto, é o *overhead* associado ao armazenamento de estados. O tempo gasto para armazenar estados é bastante grande, mesmo para tamanhos de estados modestos. Isto parece limitar a utilidade do *Time Warp* a aplicações onde a quantidade de computação gasta para processar um evento é bem maior do que o custo associado ao armazenamento de estados. Fujimoto sugere o uso de hardware especializado para resolver este problema [Fu88a, Fu89a].

Os algoritmos otimistas também usam várias vezes mais memória que os conservadores. Pode-se executar *Time Warp* com pouca memória, usando *rollbacks* para retomar memória. Isto é feito às custas de uma degradação no

desempenho. Tratamentos de erros de execução não são facilmente realizados por algoritmos otimistas, devido a complicações introduzidas pelo uso do mecanismo de *rollback*. Os *rollbacks* também tornam difícil a depuração de programas pelo usuário.

Argumenta-se que a implementação do mecanismo de *Time Warp* é muito mais complexa do que a dos algoritmos conservadores. A depuração do sistema também é mais complexa. Defensores de métodos otimistas argumentam que o esforço de implementação se justifica, uma vez que será empreendido apenas no desenvolvimento do núcleo do sistema.

2.3 Conclusões Sobre Métodos Para SDDE

Os métodos conservadores parecem fornecer bons resultados em casos nos quais se tem bastante informação prévia sobre a aplicação (bom *lookahead*). Por outro lado, o desempenho obtido com métodos otimistas indica a possibilidade de que eles ofereçam um desempenho razoável como método geral para simulação, se os custos com armazenamento de estados puderem ser controlados e o problema oferecer um grau de paralelismo razoável. Se houver propriedades de *lookahead* pobres e o custo de armazenamento de estados for alto, nenhum método já proposto oferecerá bom desempenho.

Faltam também mais pesquisas na área de aplicações em tempo real. Há grande desafio na tentativa de se usar este tipo de método para resolver o problema da computação paralela de "propósito geral".

Capítulo 3

Simulação Distribuída Dirigida por Tempo (SDDT)

Como já foi dito no Capítulo 1, em simulação dirigida por tempo, o tempo é incrementado de uma quantia fixa que define um intervalo de simulação. As mudanças de estado que ocorrem durante o intervalo de tempo são consideradas simultâneas e independentes e são processadas no final do intervalo. Todas as mudanças de estado do sistema que devem ser processadas no intervalo de tempo presente são executadas no final do intervalo, e o sistema então passa para o intervalo de tempo seguinte. As mudanças no estado de um *PL* durante um determinado intervalo vão influenciar o comportamento do *PL* e de seus vizinhos no intervalo seguinte. Estes métodos são também chamados de algoritmos de *tempo contínuo* ou de *intervalos temporais*.

Em [Pe79, Pe80], Peacock, Wong e Manning apresentaram alguns algoritmos para SDDT. Os algoritmos apresentados por eles ou sincronizam a rede toda para avançar de um intervalo para outro ou então avançam de forma assíncrona. Os algoritmos síncronos fazem a sincronização de forma

distribuída mas exploram pouco o paralelismo. Já o algoritmo assíncrono usa um mecanismo de controle do avanço dos *PLs* que é centralizado, envolvendo broadcasts de um processo central para todos os *PLs* e a convergência de informação de todos os nós da rede para o processo central. Isto implica num custo relativamente alto de comunicação na rede para uma rede grande, ou seja, não é escalável. Os autores esboçam um algoritmo assíncrono com controle distribuído parecido com o algoritmo para SDDE de Chandy-Misra, mas não explicam como resolver problemas como possíveis *deadlocks* em ciclos de *PLs* existentes na rede.

Apresentamos a seguir os sincronizadores de algoritmos propostos por Awerbuch, que permitem que a rede evolua de forma assíncrona e fazem um controle distribuído de sincronização da rede, sendo portanto escaláveis. O algoritmo para simulação de sistemas com evolução temporal híbrida, que apresentaremos no capítulo seguinte, usa um mecanismo parecido com o Sincronizador α de Awerbuch, para realizar a sincronização entre os intervalos de tempo.

3.1 Sincronizadores de Algoritmos

Estes são métodos para executar algoritmos síncronos em uma rede de processadores assíncrona. Eles foram propostos por Awerbuch em [Aw85]. Como os algoritmos síncronos são menos eficientes, porém menos complicados que os algoritmos assíncronos, esta é uma ferramenta bastante útil. Ela permite que o usuário escreva um algoritmo como se ele fosse ser executado em uma rede síncrona.

Em uma rede síncrona, um relógio global, ao qual todos os *PLs* têm acesso, avança segundo um intervalo de tempo fixo, predeterminado. No caso de simulações, o relógio global marca o tempo de simulação. As mensagens só podem ser enviadas no início de um intervalo de tempo síncrono e o atraso de propagação da mensagem é, no máximo, igual ao intervalo. Todas as mensagens enviadas no início de um intervalo são recebidas antes do início do próximo intervalo.

Os sincronizadores propostos reproduzem esta propriedade de algoritmos síncronos da seguinte forma. Um novo *pulso* (intervalo de tempo) do relógio local é gerado em um *PL* somente depois que ele já tenha recebido todas as mensagens do algoritmo síncrono, enviadas a ele pelos seus vizinhos, no intervalo presente. Para garantir esta propriedade, é necessário enviar mensagens adicionais para fins de sincronização.

Na análise de complexidade dos algoritmos será usada a seguinte notação:

Sinc: um sincronizador;

$C(Sinc)$: número de mensagens do sincronizador *Sinc* para cada intervalo de tempo síncrono;

$T(Sinc)$: tempo do sincronizador *Sinc* para cada intervalo de tempo síncrono;

$C(S)$: número de mensagens do algoritmo síncrono S;

$T(S)$: número de unidades de tempo síncrono para o algoritmo síncrono S;

A: O algoritmo assíncrono resultante da combinação de *Sinc* e S;

$C(A)$: complexidade de mensagens do algoritmo assíncrono A;

$T(A)$: complexidade de tempo do algoritmo assíncrono A;

$C_{inic}(Sinc)$: complexidade de mensagens da fase de inicialização do sincronizador;

$T_{inic}(Sinc)$: complexidade de tempo da fase de inicialização do sincronizador;

Dado um sincronizador $Sinc$ e os algoritmos S e A, tem-se que:

$$C(A) = C(S) + T(S)C(Sinc) + C_{inic}(Sinc)$$

$$T(A) = T(S)T(Sinc) + T_{inic}(Sinc).$$

Um sincronizador é eficiente se os parâmetros $C(Sinc)$, $C_{inic}(Sinc)$, $T(Sinc)$ e $T_{inic}(Sinc)$ são suficientemente pequenos. O primeiro e o terceiro parâmetros são determinantes já que representam o *overhead* por intervalo de tempo síncrono.

O autor propõe o uso de um sincronizador chamado Sincronizador γ, que é uma combinação de dois sincronizadores mais simples: o Sincronizador α e o Sincronizador β. O Sincronizador γ apresenta baixa complexidade de tempo, que é uma propriedade do Sincronizador α, e baixa complexidade de comunicação, propriedade do Sincronizador β.

Diz-se que um PL está *seguro* em relação a um determinado pulso do relógio, se todas as mensagens do algoritmo síncrono, enviadas por aquele PL naquele pulso do relógio, já chegaram aos seus destinos. Após a execução de um determinado pulso, cada PL eventualmente se torna seguro. Para um PL_i saber se está seguro, exige-se que cada PL_j vizinho de PL_i que recebeu

uma mensagem sua, envie uma confirmação da mensagem para PL_i. Estas mensagens de confirmação não aumentam a complexidade de comunicação assintótica do algoritmo.

Um novo pulso pode ser gerado em um PL_i quando é garantido que nenhuma mensagem, enviada em um pulso anterior do algoritmo síncrono, chegará a este nó no futuro. Com a definição do parágrafo acima, isto acontece quando todos os vizinhos de PL_i estiverem seguros em relação ao pulso anterior.

O primeiro sincronizador apresentado é o Sincronizador α. Cada PL_i executa o ciclo $k \geq 1$ conforme o algoritmo apresentado na figura 3.1. Os custos do Sincronizador α são:

$$T_{inic}(\alpha) = C_{inic}(\alpha) = 0$$

$$C(\alpha) = \mathcal{O}(\mid E \mid)$$

$$T(\alpha) = \mathcal{O}(1),$$

onde E é o conjunto dos arcos do grafo que representa o sistema físico sendo simulado. O valor de $T(\alpha)$ reflete o fato de que toda a comunicação ocorre entre vizinhos. Para o Sincronizador α, $C(\alpha) \geq T(\alpha)$, ou seja, o custo de comunicação é maior que o custo temporal.

O sincronizador β precisa de uma fase de inicialização na qual um líder entre os PLs, PL_L, é eleito e uma *spanning tree* com raiz em PL_L é construída. Cada PL_i executa os passos do algoritmo apresentado na figura 3.2. Os custos do sincronizador β são:

$$C(\beta) = \mathcal{O}(n)$$

$$T(\beta) = \mathcal{O}(n),$$

onde n é o número de nós do grafo que representa o sistema físico. Os valores dos custos são estes porque todo o processo ocorre na *spanning tree*. Os custos, $T_{inic}(\beta)$ e $C_{inic}(\beta)$, dependem do algoritmo escolhido para construir a árvore. Há vários algoritmos na literatura, em particular o autor apresenta um em [Aw85]. O Sincronizador β é tão caro em comunicação quanto em tempo, $C(\beta) = T(\beta) = \mathcal{O}(n)$. Comparando os dois sincronizadores observa-se que $C(\alpha) \geq C(\beta)$ e $T(\beta) \geq T(\alpha)$.

O Sincronizador γ é uma forma híbrida dos dois sincronizadores já apresentados, e tem uma fase de inicialização na qual a rede é subdividida em *clusters*. A subdivisão é definida por uma *spanning forest* da rede física simulada. Cada árvore da floresta define um cluster de nós chamada de *árvore intracluster*, com um líder. Entre cada par de árvores intracluster vizinhas é escolhido um arco preferencial (veja a figura 3.3).

O Sincronizador γ funciona em duas fases. Na primeira fase, o Sincronizador β é aplicado separadamente em cada cluster. Quando o líder de um cluster sabe que seu cluster está seguro, ele notifica o fato para todos os nós do cluster e também para os líderes de clusters vizinhos, através de um caminho que inclui o arco preferencial entre clusters. Os nós do cluster iniciam então a segunda fase, na qual eles esperam até saber que todos os clusters vizinhos estão seguros e inicializam uma aplicação do Sincronizador α entre clusters.

O Sincronizador γ engloba uma família de sincronizadores parametrizados por $2 \leq k < n$. Pode-se encontrar uma descrição mais formal do Sincronizador γ, assim como do algoritmo de geração da *spanning forest* em

[Aw85]. Este algoritmo gera uma floresta, que tem a soma do número de arcos das árvores com o número de arcos preferenciais entre clusters, E_p, e altura máxima entre todas as árvores da floresta, H_p, que satisfazem:

$$E_p \leq kn$$

$$H_p \leq \frac{\log_2 n}{\log_2 k}.$$

A análise de complexidade do Sincronizador γ é apresentada em [Aw85] e resulta em:

$$C_{inic}(\gamma) = \mathcal{O}(kn^2)$$

$$T_{inic}(\gamma) = \mathcal{O}(n\frac{\log_2 n}{\log_2 k})$$

$$C(\gamma) = \mathcal{O}(E_p) = \mathcal{O}(kn)$$

$$T(\gamma) = \mathcal{O}(H_p) = \mathcal{O}(\frac{\log_2 n}{\log_2 k}).$$

Comparando os três sincronizadores temos:

$$C(\alpha) > C(\gamma) > C(\beta) \quad e$$

$$T(\alpha) < T(\gamma) < T(\beta).$$

A variação de k no intervalo $[2, n)$ faz o Sincronizador γ ser mais parecido com o Sincronizador α ou com o Sincronizador β.

A apresentação feita nesta seção se baseia em [Aw85] e na seção 4.9 de [Am88].

Tanto os métodos de sincronização de Peacock como os sincronizadores requerem um conhecimento prévio dos canais de comunicação entre *PL*s. Isto ocorre nos Sincronizadores α e γ, porque eles requerem que cada *PL* aguarde

até saber que todos os vizinhos estão seguros, para iniciarem um novo pulso do relógio local. Isto só é possível, se cada nó já sabe previamente quem são os seus vizinhos na rede física.

Em [Aw88], Awerbuch e Sipser apresentam um sincronizador dinâmico para redes, onde os processos e canais podem falhar e se recuperar durante a execução do algoritmo. O objetivo é reproduzir o comportamento do algoritmo síncrono que executaria em uma rede síncrona estática, executando-o em uma rede assíncrona dinâmica. Quando um canal se recupera ou falha, o sincronizador reinicia a execução do algoritmo a partir do pulso zero (tempo de simulação zero). As partes da rede que se estabilizarem se comportarão como a rede estática síncrona. Este algoritmo pode ser adaptado para se ter um sincronizador para um simulador, onde a rede física sendo simulada não é estática.

Início;
  $relógio_i := t_0$;
  Enquanto $relógio_i \leq$ tempo de término **Faça**
    simule novo intervalo de tempo $[relógio_i, relógio_i + \Delta t]$;
    Se confirmações para todas as mensagens enviadas por PL_i
      no intervalo $[relógio_i, relógio_i + \Delta t]$ já foram recebidas
      Então Faça
      PL_i está seguro;
      informe todos os vizinhos que está seguro;
    Fim_Se;
    Se já recebeu mensagens de todos os vizinhos avisando
      que estão seguros em relação ao intervalo atual **Então**
      $relógio_i := relógio_i + \Delta t$;
  Fim_Enquanto;
Fim;

Figura 3.1: Sincronizador α

Início;
  $relógio_i := t_0$;
  Enquanto $relógio_i \leq$ tempo de término **Faça**
    simule novo intervalo de tempo $[relógio_i, relógio_i + \Delta t]$;
    Se $PL_i = PL_L$ **Então Faça**
      Espere para receber mensagens confirmando que todos os nós estão seguros em relação ao intervalo $[relógio_i, relógio_i + \Delta t]$;
      Avise todos os filhos na árvore que todos estão seguros;
    Senão {PL_i não é o líder}
      Espere saber que está seguro e receber mensagens avisando que todos os filhos na árvore também estão seguros;
      Informe o pai na árvore que está seguro;
      Espere receber mensagem do pai avisando que pode iniciar novo intervalo de tempo;
      Passe esta mensagem para todos os filhos na árvore;
    Fim_Se;
    $relógio_i := relógio_i + \Delta t$;
  Fim_Enquanto;
Fim;

Figura 3.2: Sincronizador β

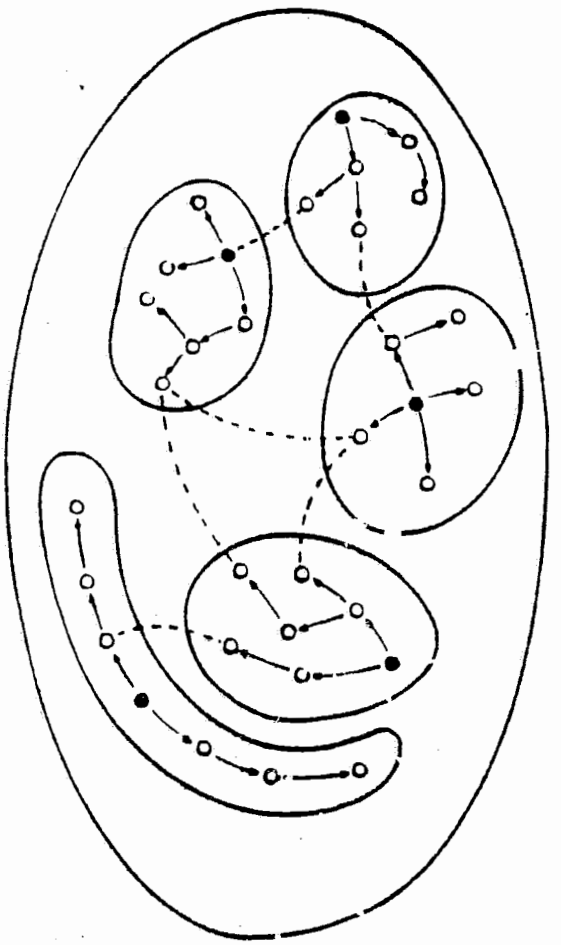


Figura 3.3: Subdivisão de uma rede física em uma *spanning forest*.

Capítulo 4

O Mecanismo de Intervalos Temporais Revogáveis

Como vimos até agora, muitos métodos foram propostos para realizar a simulação de sistemas físicos em máquinas paralelas e distribuídas [Ch89, Fu90, Je85, Mi86, Ba91, Lu89]. Todos estes algoritmos distribuídos se destinam a reproduzir o comportamento de sistemas físicos que podem ser modelados para simulação ou por uma abordagem dirigida por tempo ou dirigida por eventos. No entanto, não existe uma proposta na literatura para a simulação de sistemas que apresentam uma evolução temporal híbrida.

Os métodos dirigidos por tempo devem ser utilizados para simular sistemas físicos nos quais as mudanças de estado ocorrem continuamente no tempo e, portanto, podem ser simulados progredindo-se em intervalos de tempo fixos e atualizando-se o estado do sistema ao final de cada intervalo. Nos algoritmos de tempo contínuo, o estado de um processo físico (PF) ao final de um determinado intervalo de tempo é função do seu estado ao final do intervalo anterior, e também de todas as entradas recebidas de PFs vizi-

nhos, referentes ao intervalo atual, recebidas antes do final deste intervalo. Já vimos que a maior parte dos sistemas físicos que podem ser simulados desta maneira são modelados matematicamente por sistemas de equações diferenciais parciais.

Vimos no capítulo anterior que, por definição, em um algoritmo síncrono baseado em troca de mensagens, todas as mensagens enviadas em um determinado intervalo de tempo (ou iteração) devem alcançar seus destinos, antes do final deste intervalo. Também foram apresentados os três sincronizadores de Awerbuch [Aw85] que, quando acoplados a um algoritmo síncrono, fornecem um algoritmo assíncrono que executará em um máquina assíncrona respeitando a definição anterior. Estes mecanismos podem ser usados para sincronizar simulações distribuídas dirigidas por tempo.

Já em algoritmos de simulação dirigidos por eventos, quando um evento é executado, o relógio da simulação é atualizado para o instante de ocorrência deste evento. No algoritmo seqüencial, os eventos são armazenados em uma lista de eventos e são tratados em ordem não decrescente de tempo de simulação [Mi86]. Quando um evento é executado, variáveis de estado são atualizadas e novos eventos futuros podem ser escalonados, sendo inseridos na lista de eventos. Vimos que os algoritmos dirigidos por eventos, também conhecidos como algoritmos distribuídos de eventos discretos, podem ser classificados em conservadores e otimistas, dependendo da imposição (ou não) da condição de que os processos devem esperar pela recepção de toda a informação necessária de outros processos, antes de prosseguir. Alguns destes algoritmos são encontrados em [Ch79, Je85, Ba91].

Em nossos estudos de aplicações que se beneficiariam de simulação para-

lela e distribuída, verificamos que existe uma classe de sistemas físicos que não podem ser modelados nem por uma abordagem dirigida por eventos, nem por uma abordagem dirigida por tempo, mas sim por uma abordagem híbrida. Neste trabalho, apresentamos um método para simular modelos de sistemas físicos que evoluem no tempo segundo intervalos de tempo pré-estabelecidos, com uma atualização de variáveis de estado ao final de cada intervalo, tal como no caso dirigido por tempo, mas que evoluem segundo o instante de ocorrência de eventos discretos, dentro de cada intervalo, como na abordagem dirigida por eventos.

Nosso método utiliza um mecanismo de sincronização entre intervalos de tempo parecido com o sincronizador α de Awerbuch [Aw85]. Relaxamos a condição de que um *PL* deve satisfazer para que esteja *seguro*, de forma que possa avançar para o intervalo seguinte. A principal característica do nosso método é que permitimos que os *PLs* avancem de forma otimista para o próximo intervalo, supondo que toda a informação disponível no final do intervalo atual, a ele e seus vizinhos mais próximos, está correta. Caso esta hipótese esteja errada, exigimos que o *PL* retorne a um estado anterior correto (faça um rollback de estado), e prossiga considerando toda a informação já recebida até então. Por esta característica, o método foi denominado de *algoritmo de intervalos temporais revogáveis*.

O algoritmo de intervalos temporais revogáveis pode ser acoplado a um método de simulação otimista como o *Time Warp* ou o Espaço-Tempo (para fazer a evolução de eventos discretos dentro do intervalo) para realizar uma simulação temporal híbrida, tal como descrevemos anteriormente.

Ele também pode ser aplicado a uma simulação de eventos discretos

otimista pura, como forma de limitar o otimismo do método escolhido, sem no entanto, criar barreiras de sincronização globais, centralizadas. Neste sentido, ele funciona como um mecanismo de janela temporal [So89, So90, Ni92a, Tu92, Ba90, St91, St94a] diferente dos demais, por determinar os limites da janela de forma otimista e verdadeiramente distribuída, sendo portanto um método *escalável*. Isto também reduz o *overhead* associado ao cálculo de Tempo Virtual Global (TVG), já que, como veremos mais adiante, os limites inferiores das janelas e o diâmetro do grafo de *PLs* que representa o sistema físico dão um limite superior para os atrasos dos relógios locais dos *PLs*, podendo reduzir a freqüência na qual se torna necessário calcular TVG.

Mesmo tendo um custo de comunicação mais alto do que os demais, escolhemos um mecanismo parecido com o sincronizador α, porque na simulação de tempo contínuo e na forma híbrida, os *PLs* vizinhos precisam se comunicar entre si ao final de cada intervalo, para trocarem variáveis de estado. A mensagem de que um *PL* está seguro vai junto com a mensagem que contém as variáveis de estado, sem causar mais tráfego de mensagens na rede.

Na próxima seção, apresentamos uma descrição das características de evolução temporal híbrida de um sistema físico que pode ser simulado pelo mecanismo de passos temporais revogáveis, dando especial atenção ao sistema físico que nos inspirou inicialmente a desenvolver o método. Na seção 4.2, apresentamos o método formalmente e em seguida a prova de corretude. Na seção 4.4, discutimos o uso do algoritmo para limitar otimismo em SDDEs puras. Na seção 4.5 revisamos alguns dos trabalhos relacionados ao assunto do qual tratamos, e explicamos por que estes métodos ou não se prestam a resolver este tipo de simulação ou não o fazem da forma mais eficiente.

4.1 Especificação da Aplicação

Nesta seção, descrevemos as principais características do tipo de sistema físico que pode ser simulado pelo algoritmo que propomos. Consideramos sistemas físicos que podem ser modelados por um conjunto de processos físicos (*PF*s) que executam de forma independente, exceto para interagir com outros *PF*s. Estes sistemas podem ser representados por um grafo, com um nó associado a cada *PF* e uma aresta (representando um canal de comunicação) correspondendo a cada par de *PF*s que interagem entre si.

O algoritmo de evolução temporal híbrida que propomos pode ser usado para simular sistemas físicos que progridem no tempo, tanto de forma contínua como de forma discreta. Em outras palavras, suas variáveis de estado evoluem continuamente no tempo, com alterações instantâneas causadas pela ocorrência de eventos discretos. Durante a evolução, o tempo de simulação t progride segundo intervalos fixos, Δt, tais que o início do $(i+1)$-ésimo intervalo é dado a partir do início do i-ésimo intervalo por $t_{i+1} = t_i + \Delta t$. Além disso, dentro de cada intervalo, $[t_i, t_i + \Delta t)$, o relógio da simulação é atualizado para o instante de ocorrência do próximo evento discreto no intervalo.

Para simular um sistema deste tipo seqüencialmente, deve-se manter a lista de eventos discretos, do algoritmo dirigido por eventos seqüencial, e começar a simular a partir do instante inicial, t_0. Somente os eventos da lista que ocorrem antes do final do intervalo devem ser escalonados para execução. Quando não houver mais eventos a serem escalonados no intervalo atual, uma mudança de estado global ocorre correspondendo ao instante de simulação

$t_0 + \Delta t$, o relógio da simulação é atualizado para este valor e novos eventos são escalonados para o próximo intervalo. O algoritmo para simulação híbrido seqüencial está descrito mais formalmente na figura 4.1.

Uma característica importante das aplicações que podem ser simuladas com nosso algoritmo é que, para realizar as mudanças de estado que ocorrem no final de um intervalo de tempo, cada *PF* precisa de informações sobre valores das variáveis de estado de outros *PF*s vizinhos, no final do intervalo (antes que eles executem a mudança de suas variáveis). No algoritmo híbrido, isto significa que cada *PF* precisa conhecer todos os outros *PF*s aos quais ele está diretamente conectado (i.e. os outros *PF*s que podem afetar sua evolução temporal diretamente).

4.1.1 Um Sistema Físico Real

A única forma confiável de validar e testar máquinas e algoritmos para computação paralela e distribuída é testá-los em problemas reais [Be92]. Vimos no Capítulo 1 que o desenvolvimento destes algoritmos e sua validação é o grande desafio atual para a difusão do uso de multicomputadores como máquinas de próposito geral. Desenvolvemos um algoritmo para simular sistemas físicos com a estrutura de evolução temporal híbrida porque verificamos que um dos modelos matemáticos usados para estudar reações entre íons pesados se comporta desta forma. Implementamos o simulador testando-o neste problema de física nuclear, em um multicomputador com topologia hipercúbica, o iPSC/860 da Intel.

Início;
int := t_0;
Enquanto *int* $\leq$ tempo de término **Faça**
relógio := *int*;
Atualize o estado global do sistema no instante *relógio*;
{Inicialmente existem entradas na lista de eventos, para
cada *PF* que pode gerar eventos neste intervalo.}
lista_de_eventos := $\{(t_i, e_i)$ | o evento e_i será processado em t_i, $int \leq t_i < int + \Delta t$, a não ser que o *PF* onde e_i será executado execute um outro evento e_j tal que $t_j < t_i$ e e_j cancele e_i. Os eventos são armazenados em ordem não decrescente de t_i. A lista contém um evento (∞, e_∞) que sinaliza o término.$\}$;
próximo_tempo := instante de ocorrência do próximo evento na *lista_de_eventos*, i.e. o menor valor de t_i da lista;
Enquanto *próximo_tempo* $\leq$ tempo de término e *próximo_tempo* $< int + \Delta t$ **Faça**
retire o par (*próximo_tempo*, e) com menor timestamp da lista de eventos;
simule o efeito da execução de e em *próximo_tempo*;
Se houverem modificações na lista de eventos como conseqüência da execução de e **Então**
atualize a lista (estas modificações corresponderão a eventos com timestamps t', tais que *próximo_tempo* $\leq t' < int + \Delta t$);
relógio := *próximo_tempo*;
próximo_tempo := instante de ocorrência do próximo evento na *lista_de_eventos*;
int := *int* + Δt;
Fim;

Figura 4.1: Algoritmo Seqüencial Para Simulação Híbrida.

O sistema físico que nos interessou é uma colisão entre íons pesados, a energias intermediárias (entre $\approx$ 20 e $\approx$ 200 MeV por nucleon do projétil, no sistema de referência do laboratório), por exemplo, $^{93}Nb +^{93} Nb$ a 60 MeV/nucleon [Ba92]. A evolução de tais sistemas pode ser descrita por uma teoria baseada na equação de Boltzmann para a função distribuição de partícula única [Ai85, Be88a]. Esta equação contém um termo correspondente à força, dado pelo gradiente do campo médio, que trata efeitos coletivos no movimento dos nucleons individuais e uma integral de colisão com a forma de Uehling-Uhlenbeck, que trata colisões microscópicas entre pares de partículas. A equação resultante 4.1 é chamada de equação de Boltzmann-Uehling-Uhlenbeck (BUU).

$$\frac{\partial}{\partial t} f(\mathbf{x},\mathbf{p},t) + \nabla_r f(\mathbf{x},\mathbf{p},t) \cdot \nabla_p h(\mathbf{x},\mathbf{p},t) - \nabla_p f(\mathbf{x},\mathbf{p},t) \cdot \nabla_r h(\mathbf{x},\mathbf{p},t) = I_{coll} \tag{4.1}$$

Nesta equação, $f(\mathbf{x},\mathbf{p},t)$ é o número de partículas no ponto $(\mathbf{x},\mathbf{p})$ do espaço de fase no instante de tempo t. O termo $h(\mathbf{x},\mathbf{p},t) = p^2/2m + V(\mathbf{x},t)$ é a hamiltoniana a 1–corpo e $V(\mathbf{x},t)$ é o potencial médio auto-consistente gerado pelos nucleons. O termo I_{coll} é responsável pelo efeito médio da variação da ocupação do espaço de fase devido às colisões nucleon-nucleon e é dado por:

$$I_{coll} = \frac{1}{\pi^3 m^2 \hbar^3} \int d^3\mathbf{p_2} d^3\mathbf{p_3} d^3\mathbf{p_4}\ \delta(\epsilon + \epsilon_2 - \epsilon_3 - \epsilon_4)\ \delta(\mathbf{p} + \mathbf{p_2} - \mathbf{p_3} - \mathbf{p_4}) \frac{d\sigma}{d\Omega}[f_3 f_4 (1-f)(1-f_2) - f f_2 (1-f_3)(1-f_4)]\ ,$$

sendo que $\epsilon = p^2/2m$, $d\sigma/d\Omega$ é a seção de choque nucleon-nucleon, $\hbar$ é a constante de Planck dividida por 2π e δ representa a função delta de Dirac.

Neste trabalho, estamos interessados apenas em apresentar o suficiente da teoria e do modelo matemático para compreender o algoritmo que o resolve e suas possibilidades de paralelização. Sugerimos as referências [Ai85, Be88a] para uma descrição da equação de BUU, assim como maiores detalhes sobre o sistema físico, que não estão apresentados aqui.

O algoritmo numérico que descreveremos para tratar os efeitos coletivos no movimento de cada nucleon (evolução temporal dirigida por tempo) é equivalente a usar o método de *leap frog* para resolver as equações:

$$\begin{aligned} \frac{d\mathbf{x}(t)}{dt} &= \frac{\mathbf{p}(t)}{m}; \\ \frac{d\mathbf{p}(t)}{dt} &= -\nabla U(\rho(\mathbf{x}(t))), \end{aligned}$$

onde $\mathbf{x}(t)$ e $\mathbf{p}(t)$ representam a posição e o momento respectivamente de cada nucleon, $\rho(\mathbf{x}(t))$ é a densidade de nucleons no ponto $\mathbf{x}(t)$ e $U(\rho(\mathbf{x}(t)))$ é o campo (ou potencial) médio nuclear, tudo no instante de tempo t. De fato, realizar os seguintes passos iterativos:

$$\begin{aligned} \mathbf{x_i}(t) &= \mathbf{x_i}(t-\Delta t) + \mathbf{p_i}(t-\frac{\Delta t}{2})\frac{\Delta t}{m}; \\ \mathbf{p_i}(t+\frac{\Delta t}{2}) &= \mathbf{p_i}(t-\frac{\Delta t}{2}) - \nabla U\left(\rho(\mathbf{x_i}(t))\right)\Delta t, \end{aligned}$$

é equivalente a usar um método de segunda ordem para resolver as equações.

A equação de BUU (equação 4.1) não pode ser resolvida analiticamente e é necessário utilizar um método numérico computacionalmente viável. Agora, descrevemos uma solução numérica da equação de BUU, que é uma heurística, chamada de *método de partículas teste* [Ai85, Be88a]. O método é estocástico, de forma que é necessário repetir o procedimento um número grande de vezes (repetições independentes do programa) para obter valores médios das quantidades físicas que interessam. Para resolver as equações de BUU com o

método de partículas teste, o valor médio da densidade de partículas no instante t, $\rho(t, \mathbf{r})$, onde a média é tomada sobre todas as repetições, é usado para obter o potencial (campo médio), a força, e os momenta das partículas no instante $t + \Delta t/2$. Estes valores dos momenta das partículas, por sua vez, são usados para atualizar suas posições no intervalo $[t, t + \Delta t)$. Portanto, é necessário fazer o tratamento estatístico de forma concorrente. As *repetições* do programa são executadas simultaneamente (em paralelo) e compartilham resultados sobre as densidades das partículas, $\rho(t, \mathbf{r})$, obtidas em cada repetição, durante a execução e são, de fato, acopladas entre si.

O paralelismo inerente ao problema o torna especialmente adequado para a execução em um multicomputador. Esta característica também faz com que o problema seja especialmente interessante e diferente dos sistemas normalmente usados para testar algoritmos para simulação paralela e distribuída que, em geral, são modelos estocásticos com repetições independentes entre si. Já mencionamos que no caso de haver muitas repetições não acopladas, como acontece nos modelos de teoria de filas, é melhor executar cada repetição independente em um processador diferente. Isto se justifica porque os custos de comunicação envolvidos na paralelização de cada repetição do sistema não estão presentes no caso da execução independente de cada repetição em um processador diferente.

Para executar as N repetições individuais em paralelo, cada partícula no projétil e no alvo é representada por um conjunto de N *partículas teste*. Portanto, se existem A_p nucleons no projétil e A_t nucleons no alvo (em ambos os casos, A é o número de massa atômica dos núcleos respectivos), o número total de partículas na simulação será $N(A_p + A_t)$. Os principais

passos executados durante a simulação são dados a seguir;

1. A configuração inicial (posição e momento de cada partícula teste do alvo e do projétil) é atribuída por sorteios do tipo Monte Carlo, de acordo com as distribuições adequadas. Os nucleons do alvo são uniformemente distribuídos sobre uma esfera de raio $R_t = 1.15A_t^{1/3}$ fm. O mesmo procedimento é realizado para as partículas do projétil sobre uma esfera de raio $R_p = 1.15A_p^{1/3}$ fm. O momento de cada nucleon é sorteado de acordo com uma distribuição de Fermi.

2. O projétil é lançado contra o alvo, através da atribuição da velocidade e parâmetro de impacto adequados ao feixe (nucleons do projétil).

3. O valor do relógio da simulação é atualizado para $t = t_0$.

4. O valor de $\rho\,(t_0, \mathbf{x})$ é determinado, contando-se todas as partículas teste, $N'(t_0)$, em um elemento de volume $(\delta x)^3$, em torno do ponto $\mathbf{x}$ e calculando: $\rho(t_0, \mathbf{x}) = N'(t_0)/[(\delta x)^3 N]$. O potencial nuclear médio dependente da densidade, U, é então atualizado em cada ponto, de acordo com a seguinte parametrização (potencial de Skyrme):

$$U(\rho) = C\rho/\rho_0 + D(\rho/\rho_0)^{7/6}(\mathrm{MeV}), \tag{4.2}$$

onde $C(\rho/\rho_0)$ é atrativo (i.e. C é uma constante negativa), $D(\rho/\rho_0)^{7/6}$ é repulsivo (i.e. D é uma constante positiva) e ρ_0 é a densidade nuclear normal (densidade do estado fundamental).

O momento de cada partícula é então calculado de acordo com as equações de Hamilton:

$$\mathbf{p_i}(t + \Delta t/2) = \mathbf{p_i}(t - \Delta t/2) - \nabla U\left(\rho(\mathbf{x_i}(t))\right)\Delta t. \tag{4.3}$$

5. Dentro de cada intervalo de tempo $[int, int + \Delta t)$, a simulação progride dirigida por eventos. As partículas teste evoluem em um potencial constante, de acordo com a mecânica Newtoniana, seguindo trajetórias retilíneas. Se $(\mathbf{x_i}, \mathbf{p_i})$ são a posição e o momento da partícula teste i, respectivamente, m é a massa do nucleon e δt é o intervalo de tempo desde a ocorrência do último evento, então a outra equação de Hamilton dá a forma como as posições evoluem:

$$\mathbf{x_i}(t + \delta t) = \mathbf{x_i}(t) + \frac{\mathbf{p_i}(t)}{m}\delta t, \tag{4.4}$$

As colisões nucleares (eventos), que ocorrem neste intervalo de tempo, são escalonadas em uma lista de eventos. Os eventos da lista são executados, em ordem não decrescente do tempo de simulação em que devem ocorrer (ordem não decrescente de *timestamps*). Como conseqüência da execução de cada colisão, possivelmente novos eventos serão escalonados e eventos já escalonados podem ser retirados da lista. Isto porque após uma colisão, é necessário recalcular o instante em que o par que colidiu possivelmente colidirá com todas as outras partículas em uma determinada vizinhança. A equação 4.4 é usada para calcular a posição de todas as partículas teste, no instante de ocorrência do evento novo em $t + \delta t$, dado que o último evento ocorreu no instante t.

O instante de ocorrência de uma colisão entre duas partículas é calculado determinando-se o instante no qual o par alcançará a distância de máxima aproximação de suas trajetórias clássicas. Se esta distância for menor que $\sqrt{\sigma_{NN}/\pi}$ (σ_{NN} é a seção de choque nucleon-nucleon determinada experimentalmente), a colisão pode ocorrer e o evento é escalonado para execução. Supomos que estas colisões são instantâneas

e que o efeito do evento sobre o estado do sistema é alterar os momenta das partículas que colidiram. O ângulo de espalhamento do par, no referencial do seu centro de massa, é gerado aleatoriamente de acordo com a seção de choque diferencial nucleon-nucleon experimental. Só existem colisões entre partículas pertencentes à mesma repetição Monte Carlo.

6. Quando não houver mais eventos no intervalo $[int, int + \Delta t)$, todas as partículas evoluem, de acordo com a equação 4.4, para as posições $\mathbf{x_i}(int+\Delta t)$. Um novo valor para a densidade $\rho(int+\Delta t, \mathbf{x})$, é calculado, contando-se todas as partículas teste N' em um elemento de volume $(\delta x)^3$, em torno do ponto $\mathbf{x}$ e dividindo-se $\rho(\mathbf{x}) = N'/[(\delta x)^3 N]$. O potencial nuclear médio dependente da densidade, U, é então atualizado em cada ponto, usando-se a equação 4.2.

   O momento de cada partícula é calculado agora, de acordo com a equação 4.3. Dados os novos momenta e posições das partículas (o estado do sistema), no final do intervalo $[int, int + \Delta t)$, int é atualizado, $int := int + \Delta t$, e o procedimento é repetido a partir do passo 5.

4.1.2 Paralelismo Inerente à Aplicação

A primeira forma que se pode pensar em paralelizar o problema que apresentamos nesta seção poderia ser executar cada uma das repetições individuais, em um processador diferente. No entanto, as repetições não são realmente independentes porque, como vimos, elas precisam compartilhar informações a respeito da densidade de partículas no final de cada intervalo de tempo. Este tipo de paralelização causaria um *overhead* de comunicação muito alto,

porque cada processador teria que trocar informação com todos os outros ao final de cada intervalo.

Como só é necessário saber o número de partículas na vizinhança de um ponto para calcular a densidade, é melhor fazer o seguinte. O espaço ocupado pelo sistema é dividido em regiões adjacentes (por exemplo, uma malha tri-dimensional). Um processo lógico (PL) é associado a cada região e simula todas as N repetições em paralelo, usando o método de partículas teste descrito na seção anterior, ou seja, um PL é responsável por simular o movimento e as colisões de todas as partículas teste pertencentes à sua região. Ao final de cada intervalo, cada região (PL) só precisa comunicar aos seus *vizinhos mais próximos* na rede o número de partículas em um elemento de volume, próximo de sua fronteira comum. Dentro de um intervalo, no caso paralelo, os eventos são colisões entre pares de partículas e a passagem de partículas entre regiões vizinhas. Esta abordagem reduz enormemente o custo da comunicação na rede de processadores por mantê-la apenas entre vizinhos mais próximos.

Acreditamos que esta aplicação é um bom teste para algoritmos de simulação paralela. O sistema é grande e apresenta uma carga computacional bastante alta, com muito paralelismo intrínseco, representando um problema real e não uma carga de trabalho sintética. Este problema não apresenta *lookahead* e, portanto, é mais adequado tratá-lo com métodos otimistas.

4.2 Apresentação do Algoritmo

A idéia básica por trás do algoritmo que estamos propondo, para resolver problemas de simulação com uma estrutura de evolução temporal do tipo que descrevemos na seção 4.1, é usar o *sincronizador* α [Aw85], acrescido de um mecanismo otimista para a sincronização entre intervalos de tempo, e usar um método para SDDE otimista dentro de cada intervalo.

A razão pela qual escolhemos o sincronizador α é que cada *PL* tem mesmo que se comunicar com seus vizinhos, ao final de cada intervalo, para trocar informações sobre variáveis de estado. As mensagens de sincronização podem ser representadas por estas mensagens de estado. Na próxima seção, provamos que o algoritmo está correto se as seguintes condições são garantidas.

1. Cada *PL* conhece os outros *PL*s vizinhos, aos quais ele está conectado, no grafo que representa o modelo do sistema físico.

2. Os canais lógicos que conectam os *PL*s são FIFO (First In, First Out). As mensagens enviadas entre um par de *PL*s serão recebidas, na ordem de tempo real em que foram enviadas. Esta condição pode ser relaxada, se introduzimos um mecanismo de antimensagens como no *Time Warp*.

3. Cada evento é executado em tempo finito.

4. O tempo de transmissão de mensagens é arbitrário, porém finito.

5. Existe uma constante positiva ϵ, tal que, para cada ciclo de *PF*s no modelo do sistema físico, existe pelo menos um *PF* que somente gerará mensagens m', causadas pela recepção de mensagens m no instante t, com *timestamps* $t + \epsilon$. Não é necessário conhecer o valor de ϵ.

Estas condições garantem que o algoritmo é livre de *deadlock* e termina corretamente.

Dizemos que um *PL* está SEGURO em relação a um intervalo de tempo $[t, t + \Delta t)$ quando o próximo evento a ser processado na sua lista de eventos tem um *timestamp* maior que $t + \Delta t$. Em outras palavras, um *PL* é considerado SEGURO em relação ao intervalo de tempo atual se todas as mensagens que ele poderia ter enviado dentro deste intervalo já foram enviadas. Esta definição de um *PL* SEGURO é diferente da de Awerbuch e é possível porque estamos considerando canais lógicos de comunicação FIFO e também porque usamos um mecanismo de *rollback* para corrigir estimativas erradas.

Tal como no mecanismo de *Time Warp*, consideramos que cada mensagem contém a seguinte informação:

i) Identidade do *PL* que a envia;

ii) Identidade do *PL* que deve receber a mensagem;

iii) Tempo Virtual de Envio, TVE, que é o valor do tempo de simulação que o relógio do *PL* que envia a mensagem está marcando, no instante em que ele a envia;

iv) Tempo Virtual de Recebimento, TVR, o tempo virtual em que o *PL* destino deve receber a mensagem;

v) Conteúdo da mensagem.

Cada mensagem deve satisfazer TVR $\geq$ TVE. Os eventos escalonados por um *PL* para ele mesmo também devem conter esta informação.

Apresentamos nosso algoritmo de forma modular, onde um processo chamado de *simulador*, que executa em cada processador, é responsável pelo mecanismo de sincronização. Os *PLs* são apenas responsáveis por gerar a configuração inicial, gerar e executar os eventos, atualizar as variáveis de estado e preparar os dados de saída do programa ao terminar. Usamos o conceito de *Tempo Virtual Global* (*TVG*), cuja definição está na seção 2.2.1 [Je85], para a detecção de progresso global do sistema.

Uma instância do simulador executa em cada processador e mantém as seguintes variáveis para cada PL_i alocado a seu processador.

relógio$_i$: Marca *Tempo Virtual Local* (*TVL*), relógio local de PL_i.

t_i: Armazena o valor do limite inferior do intervalo de tempo virtual sendo simulado.

pronto$_i$: Variável booleana, que é VERDADEIRA se PL_i está ocioso e portanto pronto, para receber novas instruções sobre como proceder.

*vizinho*_SEGURO$_i(j,t)$: Variável booleana, que indica que PL_j está SEGURO em relação ao intervalo $[t - \Delta t, t)$, $\forall$ PL_j que é vizinho de PL_i e para todos os intervalos de tempo t da simulação.

*todos_os_vizinhos*_SEGUROS$_i(t)$: Variável booleana, que é VERDADEIRA se todos os vizinhos de PL_i estão SEGUROS em relação ao intervalo $[t - \Delta t, t)$.

$\rho_i(t)$: Representa as variáveis de estado que PL_i precisa enviar a seus vizinhos, no final do intervalo $[t - \Delta t, t)$.

lista_de_estados$_i$: Lista dos estados passados de PL_i, que serão usados em caso de *rollback*.

lista_de_msgs_recebidas$_i$: Lista de mensagens de eventos recebidas, para serem executadas por PL_i, mantidas em ordem não-decrescente de TVR.

estados_dos_vizinhos$_i(t)$: Armazena as variáveis de estado, $\rho_j(t)$, de vizinhos PL_j de PL_i, usadas para calcular mudanças de estado, que ocorrem antes de iniciar o intervalo de simulação $[t, t + \Delta t)$.

Todos os protocolos para recepção de mensagens no algoritmo devem ser executados *atomicamente*. Listamos agora, todas as mensagens usadas pelos protocolos de sincronização.

INICIALIZE_(i): Enviada pelo simulador para cada PL_i local, para dar partida ao procedimento de inicialização do PL.

TERMINE_(i): Enviada pelo simulador para cada PL_i local, para dar partida ao procedimento de término.

COMPLETE_INTERVALO_ATUAL_(i, t): Enviada pelo simulador para PL_i, avisando-o que ele já terminou a execução de todos os eventos no intervalo de tempo atual e que deve calcular seu estado, correspondente ao final do intervalo $[t - \Delta t, t)$, e enviar $\rho_i(t)$ que será enviado a seus vizinhos.

INICIE_NOVO_INTERVALO_$(i, t,$*estados_dos_vizinhos*$_i(t))$: Notifica o PL_i local que ele deve calcular o estado correspondente ao início do intervalo de tempo $[t, t + \Delta t)$, a partir da informação recebida de seus vizinhos,

correspondente ao final do intervalo $[t - \Delta t, t)$. Novos eventos serão escalonados para execução neste novo intervalo de tempo.

SEGURO_$(j, i, t, \rho_j(t))$: Mensagem enviada pelo simulador local de PL_j para o simulador local de PL_i, notificando-o que PL_j está SEGURO em relação ao intervalo $[t - \Delta t, t)$ e fornecendo $\rho_j(t)$ para o cálculo do novo estado de PL_i, correspondente ao início do intervalo $[t, t + \Delta t)$.

NOTIFIQUE_ROLLBACK_(j, i, t): Notificação do simulador de PL_j para o simulador de PL_i, informando que o primeiro fez um *rollback* para o instante de tempo virtual t.

EVENTO_(j, i): Enviada por PL_j para seu simulador local, que então a envia para o simulador de PL_i. Quando o simulador que executa no processador que executa PL_i receber a mensagem, ele a enviará para PL_i. A mensagem causa a execução do evento correspondente ao seu conteúdo, que reproduz o comportamento do PF sendo simulado. Dizemos que o PL_j escalona o evento para ser executado por PL_i. Chamaremos um evento de STRAGGLER se ele atingir seu destino no passado do relógio local, TVR $\leq$ *relógio*$_i$.

PRONTO_$(i, t, estado_i(t))$: Mensagem enviada por PL_i ao simulador local, avisando-o que PL_i está ocioso e pronto para receber instruções, sobre como proceder para continuar executando ou terminar. Representamos por $estado_i(t)$ todas as variáveis do PL_i no instante t, que caracterizam o estado do PF_i respectivo, no mesmo instante. O estado atualizado após a execução do evento é enviado para o simulador local, que o armazena na lista de estados (procedimento chamado de *checkpointing*).

ESTADO_PRONTO_$(i, t, \rho_i(t), estado_i(t))$: Mensagem de PL_i para o simulador local, avisando-o que PL_i terminou a atualização de suas variáveis de estado, correspondentes ao final do intervalo $[t - \Delta t, t)$, e fornecendo as variáveis $\rho_i(t)$ que devem ser enviadas a seus vizinhos.

Cada processador tem um processo simulador que executa o algoritmo da figura 4.2. O comportamento deste simulador consiste em enviar uma mensagem para inicializar cada *PL* local, receber mensagens até que TVG seja maior que o tempo de término e depois enviar uma mensagem para cada *PL* local avisando-o que a condição de término foi satisfeita e que ele pode executar o procedimento de finalização de sua computação. As figuras 4.3–4.7 mostram os protocolos a serem executados pelo simulador ao receber cada tipo de mensagem.

As mensagens PRONTO_$(i, t, estado_i(t))$ são enviadas ao simulador pelos *PL*s locais após a execução da fase de inicialização, de eventos e de inicialização de novos intervalos (veja figuras 4.10, 4.11 e 4.13). Uma mensagem deste tipo sinaliza ao simulador que o *PL* que a enviou está ocioso e pronto para receber e executar novos eventos ou iniciar um novo intervalo. Estas mensagens também trazem o estado local de PL_i (após a execução do evento) a ser inserido na lista de estados de PL_i.

O protocolo da figura 4.4 especifica as ações a serem seguidas na recepção de mensagens referentes a eventos. Se o evento foi enviado por um *PL* local, destinado a um *PL* remoto, o simulador local o envia para o simulador do *PL* destino. Se a mensagem for destinada a um *PL* local, primeiro o simulador verifica se o *PL* que a enviou já havia sido marcado seguro em relação ao intervalo ao qual a mensagem pertence, e o marca não seguro (a hipótese de

que os canais lógicos entre *PL*s são FIFO garante que o *PL* origem sofreu *rollback* e enviou este evento depois que enviou mensagens SEGURO referentes a intervalos posteriores ao *timestamp* do evento). Ou seja, se o *PL* que enviou este evento já havia dado permissão para PL_i avançar para o próximo intervalo esta permissão é *revogada*. Se o evento chegar no passado do *PL* destino (é um STRAGGLER), este *PL* sofre um *rollback* de estado. O evento é então inserido na lista de eventos a serem processados pelo *PL* destino.

As mensagens ESTADO_PRONTO_$(i, t, \rho_i(t), estado_i(t))$ são recebidas segundo o protocolo da figura 4.5. Elas sinalizam ao simulador local que PL_i acabou de finalizar o intervalo $[t - \Delta t, t)$. As variáveis de estado local, $\rho_i(t)$, a serem enviadas para os vizinhos de PL_i são fornecidas, assim como o estado local do PL_i no final do intervalo. O simulador então envia mensagens SEGURO_$(i, j, t, \rho_i(t))$ para os vizinhos, PL_j, de PL_i. Se o simulador local de PL_i já recebeu notificações de que seus vizinhos estão seguros em relação a esse intervalo, ele envia uma mensagem para PL_i disparando o início do intervalo seguinte.

A recepção de uma mensagem SEGURO_$(j, i, t, \rho_j(t))$ fornece ao PL_i as variáveis de estado do vizinho PL_j, referentes ao final do intervalo $[t - \Delta t, t)$. O simulador local marca PL_j como estando SEGURO em relação a este intervalo. Se essa mensagem chegar no passado do relógio de PL_i e o valor de $\rho_j(t)$ for diferente de um valor recebido anteriormente, ela causa um *rollback*. Isto porque a notificação de *rollback* que PL_j enviou após ter enviado a mensagem SEGURO antiga, não necessariamente causou um *rollback* em PL_i, que continuou executando com uma estimativa possivelmente errada de $\rho_j(t)$. A hipótese de que os canais lógicos entre *PL*s são FIFO garante

que esta mensagem SEGURO_$(j, i, t, \rho_j(t))$ foi enviada mais recentemente que a recebida anteriormente. Se ao receber esta mensagem, também já foram recebidas mensagens SEGURO de todos os outros vizinhos de PL_i referentes a este intervalo, o simulador envia mensagem para PL_i disparando o início de um novo intervalo.

O último tipo de mensagem a ser tratado pelo simulador local são mensagens NOTIFIQUE_ROLLBACK_(j, i, t). Estas são tratadas segundo o protocolo da figura 4.7. Se existem eventos de PL_j com *timestamps* posteriores a t, já processados, na lista de eventos de PL_i, estes foram enviados antes do rollback de PL_j (isto é garantido pela hipótese dos canais lógicos entre *PL*s serem FIFO), são retirados da lista e PL_i sofre *rollback* para desfazer os efeitos do processamento destes eventos errados.

O procedimento de envio de eventos da figura 4.8 simplesmente verifica se o próximo evento da lista pertence ao intervalo atual do *PL* destino. Caso isto seja verdade, o evento é enviado para ser executado. Caso contrário, o simulador envia mensagem ao *PL* destino disparando a inicialização de um novo intervalo. O relógio de cada *PL* local é atualizado por este procedimento.

A figura 4.9 mostra o procedimento de *rollback* para um instante T. Ao inicializar um *rollback* de estado de um PL_i, o simulador envia notificações deste *rollback* para todos os seus *PL*s vizinhos. Se o *rollback* foi causado por uma mensagem SEGURO, é restaurado o estado de PL_i referente ao final do intervalo ao qual a mensagem se refere. Caso contrário, é restaurado o estado mais recente de PL_i anterior a T. Eventos escalonados por PL_i ou pelo *PL* que causou o *rollback*, com *timestamps* posteriores a T são retirados da lista de eventos (eles estão errados porque, pela hipótese FIFO dos canais lógicos,

foram enviados antes da recepção da mensagem que causou o *rollback*). O PL que causou o *rollback* é marcado como INSEGURO em relação ao intervalo ao qual T pertence e os posteriores. O relógio local é atualizado. Finalmente, se o *rollback* foi causado por uma mensagem SEGURO, o simulador envia mensagem disparando o início do intervalo $[T, T + \Delta t)$ em PL_i. Caso contrário, se PL_i está ocioso, o simulador lhe envia seu próximo evento.

O algoritmo de simulação, executado em cada processador é apresentado a seguir. Logo depois, a partir da figura 4.10, apresentamos os procedimentos muito simples seguidos por cada PL_i ao receber mensagens. Lembramos que o texto que aparece entre {} é comentário. Usamos a notação t^- para indicar o instante de tempo no final de um intervalo, porém antes de receber as variáveis de estado dos vizinhos necessárias para iniciar o novo intervalo. Por exemplo, na figura 4.5, $estado_i(t^-)$ se refere ao estado do PL_i local ao final do intervalo $[t - \Delta t, t)$, antes dele usar as variáveis $\rho_j(t)$ enviadas pelos vizinhos, PL_j, para calcular $estado_i(t)$, seu estado no início do intervalo $[t, t + \Delta t)$.

Início;
  Para cada PL_i local **Faça**
    $relógio_i := t_i := t_0$;
    $vizinho\_\text{SEGURO}_i(j,t)$:= falso, $\forall j,t$ tal que PL_j é um vizinho de PL_i e t são limites inferiores de intervalos;
    $todos\_os\_vizinhos\_\text{SEGUROS}_i(t)$:= falso, $\forall t$;
    {A lista de eventos de PL_i é inicializada com um evento que indica o término da simulação escalonado para $t = \infty$.}
    $lista\_de\_msgs\_recebidas_i := \{(\infty, e_\infty)\}$;
    $pronto_i$:= falso;
    Envie INICIALIZE_(i) para PL_i.;
  Enquanto TVG $\leq$ tempo de término **Faça**
    Receba mensagens e trate-as de acordo com os protocolos adequados.;
    {Todos os protocolos para recepção de mensagens devem ser executados atomicamente!};
  Envie TERMINE_(i) para todos os PLs locais.;
Fim;

Figura 4.2: Algoritmo simulador executado por cada processador.

{Protocolos executados na recepção de mensagens.}
Se uma mensagem PRONTO_$(i, t, estado_i(t))$ for recebida **Então**
  Se $relógio_i = t$ **Então** {PL_i não sofreu rollback.}
    Insira o novo estado de PL_i, $estado_i(t)$ na $lista\_de\_estados_i$.;
  $pronto_i$:= verdadeiro;
  Envie_Proximo_Evento_Para_o_PL_i;

Figura 4.3: Protocolo executado na recepção de mensagens PRONTO_$(i, t, estado_i(t))$.

Se uma mensagem EVENTO_(j, i) for recebida **Então**
  Se PL_i não está alocado a este processador **Então**
    Envie EVENTO_(j, i) para o processador ao qual PL_i está alocado.
  Senão
    t_{sup} := limite superior do intervalo ao qual o TVE de
        EVENTO_(j, i) pertence, $t_{sup} - \Delta t \leq \text{VST} < t_{sup}$;
    Enquanto *vizinho*_SEGURO$_i(j, t_{sup})$ **Faça**
      *vizinho*_SEGURO$_i(j, t_{sup})$:= falso;
      *todos_os_vizinhos*_SEGUROS$_i(t_{sup})$:= falso;
      $t_{sup} := t_{sup} + \Delta t$;
    Se EVENTO_(j, i) é um STRAGGLER **Então** {TVR < *relógio*$_i$}
      Faça_um_Rollback_Para_$(TVR, i, j, falso)$;
    Insira EVENTO_(j, i) na *lista_de_msgs_recebidas*$_i$.;

Figura 4.4: Protocolo executado na recepção de mensagens EVENTO_(j, i).

Se uma mensagem ESTADO_PRONTO_$(i, t, \rho_i(t), estado_i(t^-))$
  for recebida **Então**
  Se t = *relógio*$_i$ **Então**
    {PL_i não sofreu rollback enquanto calculava $\rho_i(t)$.}
    Insira $estado_i(t^-)$ na *lista_de_estados*$_i$.;
    Envie SEGURO_$(i, j, t, \rho_i(t))$ para todos os vizinhos PL_j.;
    Se *todos_os_vizinhos*_SEGUROS$_i(t)$ **Então**
      Envie INICIE_NOVO_INTERVALO_$(i, t, estados_dos_vizinhos_i(t))$.;
    Senão *pronto*$_i$:= verdadeiro;
  Senão {PL_i sofreu um rollback.}
      pronto$_i$:= verdadeiro;
      Envie_Proximo_Evento_Para_o_PL_i;

Figura 4.5: Protocolo executado na recepção de mensagens
ESTADO_PRONTO_$(i, t, \rho_i(t), estado_i(t^-))$.

$\{PL_j$ está SEGURO em relação ao intervalo $[t - \Delta t, t).\}$
Se uma mensagem SEGURO_$(j, i, t, \rho_j(t))$ for recebida **Então**
  *vizinho*_SEGURO$_i(j, t)$:= verdadeiro;
  Inclua $\rho_j(t)$ em *estados_dos_vizinhos*$_i(t)$. Se o valor
    já existia, substitua-o pelo valor novo.;
  Se $t <$*relógio*$_i$ e $\rho_j(t)$ é muito diferente do
      último $\rho_j(t)$ recebido **Então**
    Faça_um_Rollback_Para_$(t, i, j,$ *verdadeiro*$)$;
  Senão {O SEGURO de PL_j estava bloqueando PL_i.}
    Se *vizinho*_SEGURO$_i(j, t)$, $\forall PL_j$ vizinho de PL_i **Então**
      *todos_os_vizinhos*_SEGUROS$_i(t)$:= verdadeiro;
      Se *pronto*$_i$ e $t =$ *relógio*$_i$ **Então**
        $\{PL_i$ já completou o intervalo atual e não sofreu rollback.$\}$
        Envie INICIE_NOVO_INTERVALO_$(i, t,$*estados_dos_vizinhos*$_i(t))$.;
        pronto$_i$:= falso;

Figura 4.6: Protocolo executado na recepção de mensagens SEGURO_$(j, i, t, \rho_j(t))$.

$\{PL_j$ sofreu rollback para o instante $T.\}$
Se uma mensagem NOTIFIQUE_ROLLBACK_(j, i, T) for recebida **Então**
  Se existem mensagens de PL_j na *lista_de_msgs_recebidas*$_i$
      já processadas e tais que TVE $> T$ **Então**
    $t :=$ TVE da mensagem de PL_j com menor
      TVE tal que TVE $> T$;
    Retire esta mensagem com TVE $= t$ da *lista_de_msgs_recebidas*$_i$.;
    Faça_um_Rollback_Para_$(t, i, j, falso)$;
  Senão
    Retire mensagens escalonadas por PL_j com TVE $> T$ de
      lista_de_msgs_recebidas$_i$.;

Figura 4.7: Protocolo executado na recepção de mensagens NOTIFIQUE_ROLLBACK_(j, i, T).

{Este procedimento tenta enviar o próximo evento para o PL_i local
respeitando as regras de sincronização. Ele é responsável pela
atualização dos relógios.}
Envie_Proximo_Evento_Para_o_PL_i
Início;
$prox_$TVR := TVR do próximo EVENTO na
$lista\_de\_msgs\_recebidas_i$;
t_{inf} := limite inferior do intervalo ao qual $prox_$TVR pertence,
$t_{inf} \leq prox_$TVR$< t_{inf} + \Delta t$;
Se $t_{inf} \leq t_i + \Delta t$ **Então**
Se $prox_$TVR $>$ tempo de término **Então**
$relógio_i := \infty$;
Senão
$relógio_i := prox_$TVR;
Envie próximo EVENTO para PL_i;
$pronto_i$:= falso;
Senão $\{t_{inf} \geq t_i + \Delta t\}$
Se $t_i + \Delta t >$ tempo de término **Então**
$relógio_i := \infty$;
Senão {Avance para o próximo intervalo.}
$t_i := t_i + \Delta t$;
$relógio_i := t_i$;
Envie COMPLETE_INTERVALO_ATUAL_(i, t_i) para PL_i.;
$pronto_i$:= falso;
Fim;

Figura 4.8: Procedimento de despacho de eventos para os PLs locais.

{Procedimento de rollback. Implementa um rollback do PL_i para
o instante T, causado por PL_j.}
Faça_um_Rollback_Para_$(T, i, j, Inicie_Intervalo)$
Início;
Envie NOTIFIQUE_ROLLBACK_(i, k, T) para todos os vizinhos PL_k.;
Se $Inicie\_Intervalo$ **Então** Restaure $estado_i(T^-)$ na $lista\_de\_estados_i$.;
Senão Restaure o estado mais recente da $lista\_de\_estados_i$ que
corresponde a um tempo virtual local T', que satisfaz $T' \leq T$.;
Retire mensagens do tipo EVENTO escalonadas por PL_i
ou PL_j com TVE $> T$. Não reenvie mensagens de PL_i
com TVE no intervalo $[T', T]$.
t_{sup} := limite superior do intervalo ao qual T pertence,
$t_{sup} - \Delta t \leq T < t_{sup}$;
Enquanto $vizinho\_SEGURO_i(j, t_{sup})$ **Faça**
$vizinho\_SEGURO_i(j, t_{sup})$:= falso;
$todos\_os\_vizinhos\_SEGUROS_i(t_{sup})$:= falso;
$t_{sup} := t_{sup} + \Delta t$;
Se $T' < t_i$ **Então**
$t_i := t_{inf}$, onde t_{inf} é o limite inferior do intervalo
ao qual T' pertence, $t_{inf} \leq T' < t_{inf} + \Delta t$;
$relógio_i := T'$;
Se $Inicie\_Intervalo$ **Então**
Envie INICIE_NOVO_INTERVALO_$(i, t_i, estados_dos_vizinhos_i(t_i))$.;
Senão Se $pronto_i$ **Então** Envie_Proximo_Evento_Para_o_PL_i;
Fim;

Figura 4.9: Procedimento de Rollback.

{Algoritmo executado por cada PL_i.}
Se uma mensagem INICIALIZE_(i) for recebida **Então**
Calcule configuração inicial.;
Escalone eventos que ocorrerão no primeiro intervalo de tempo (TVE $< t_0 + \Delta t$) e envie-os para o simulador.;
Envie PRONTO_$(i, t_0, estado_i(t_0))$ para o simulador local.;

Figura 4.10: Protocolo executado por PL_i ao receber mensagem INICIALIZE_(i).

Se uma mensagem EVENTO_(j, i) for recebida **Então**
Execute este evento, possivelmente alterando variáveis de estado.;
Se necessário, escalone novas mensagens EVENTO_(i, j) e envie-as para o simulador que as enviará aos PL_j.
Somente escalone eventos que satisfazem TVE $< t_i + \Delta t$.;
Envie PRONTO_$(i, relógio_i, estado_i(relógio_i))$ ao simulador local.;

Figura 4.11: Protocolo executado por PL_i ao receber mensagens EVENTO_(j, i).

Se uma mensagem COMPLETE_INTERVALO_ATUAL_(i, t) for recebida **Então**
Calcule o valor das variáveis de estado no final do intervalo $[t - \Delta t, t)$.;
Envie ESTADO_PRONTO_$(i, t, \rho_i(t), estado_i(t^-))$ com as variáveis de estado $\rho_i(t)$, que devem ser enviadas aos vizinhos nas mensagens SEGURO_$(i, j, t, \rho_i(t))$.;

Figura 4.12: Protocolo executado por PL_i ao receber mensagens COMPLETE_INTERVALO_ATUAL_(i, t).

Se uma mensagem INICIE_NOVO_INTERVALO_$(i, t, estados_dos_vizinhos_i(t))$ for recebida **Então**
Calcule o novo estado do sistema no início do intervalo $[t, t + \Delta t)$, baseado nas mensagens $estados\_dos\_vizinhos_i(t)$ recebidas dos vizinhos.;
Se necessário, escalone novas mensagens EVENTO_(i, j) e envie-as ao simulador local, que as enviará para os PL_j.
Somente escalone eventos que satisfazem TVE $< t + \Delta t$.;
Envie PRONTO_$(i, t, estado_i(t))$ para o simulador local.;

Figura 4.13: Protocolo executado por PL_i ao receber mensagens INICIE_NOVO_INTERVALO_$(i, t, estados_dos_vizinhos_i(t))$.

Se uma mensagem TERMINE_(i) for recebida **Então**
Execute o procedimento de término apropriado sobre variáveis de estado, tais como estatística, etc. preparando-as para output.

Figura 4.14: Protocolo executado por PL_i ao receber mensagem TERMINE_(i).

O algoritmo de passos temporais revogáveis permite que os *PLs* progridam no tempo de forma otimista dentro de cada intervalo. Cada *PL* deve receber mensagens do tipo SEGURO de todos os seus vizinhos, relativas ao intervalo de tempo presente, para poder avançar para o próximo intervalo. Como os *PLs* enviam mensagens tipo SEGURO de forma otimista (eles não podem ter certeza absoluta de que não receberão STRAGGLERs de algum outro *PL*), estes vizinhos podem sofrer *rollback* para intervalos anteriores, se tornando INSEGUROs novamente. Quando um PL_j, anteriormente SEGURO, sofre *rollback* se tornando INSEGURO, seus vizinhos PL_i receberão mensagens NOTIFIQUE_ROLLBACK_(j, i, T), mas não sofrerão *rollback* imediatamente, a não ser que algum deles já tenha processado mensagens incorretas de PL_j. O PL_i sofrerá *rollback* mais adiante, se ele receber um novo STRAGGLER de PL_j

após o seu *rollback* ou se o novo estado de PL_j, $\rho_j(t)$, que será enviado com a nova mensagem SEGURO for muito diferente (dentro de um intervalo de tolerância dado pela aplicação) do valor enviado com a mensagem SEGURO anterior.

É possível adaptar o algoritmo para intervalos de tempo de tamanho variável, se estes tamanhos são conhecidos antes da execução ou, por exemplo, enviando os valores dos tamanhos dos intervalos, juntamente com as mensagens SEGURO. Na seção 4.4, apresentaremos a prova do algoritmo e algumas de suas principais propriedades.

4.3 Um Mecanismo Distribuído Para Limitar Otimismo

Em métodos otimistas para simulação de eventos discretos comuns, não há restrições sobre a diferença do valor dos relógios dos *PLs*. Pode acontecer, por exemplo, uma situação na qual um dos *PLs* atrase muito mais, em tempo virtual, do que os outros. Um PL_i muito atrasado provavelmente acarretará *rollbacks* sucessivos nos seus vizinhos, que se propagarão pela rede. Neste caso, a maior parte do trabalho realizado pelos outros *PLs* será desfeito por *rollback*, sendo portanto inútil. O PL_i lento será um gargalo para o avanço temporal do sistema como um todo e a simulação paralela será muito ineficiente. Este tipo de situação pode ser causada, por exemplo, quando a carga computacional não está igualmente distribuída entre os *PLs*.

Em simulação distribuída de eventos discretos otimistas, há uma série de *overheads* associados aos processos de *rollback*. A necessidade de gravar o

estado dos *PLs* (*checkpointing*) freqüentemente implica em gastos de tempo real e memória consideráveis, usados apenas para garantir o mecanismo de sincronização. O cálculo de Tempo Virtual Global (TVG), que é uma medida do avanço do sistema como um todo, e que deve ser realizado em paralelo com a simulação propriamente dita, também implica em custos que podem onerar excessivamente a computação distribuída.

É fácil ver que o algoritmo de Intervalos Temporais Revogáveis que apresentamos pode ser utilizado para realizar simulações de eventos discretos de forma semi-otimista. Mesmo não sendo necessária a troca de informação sobre variáveis de estado ao final de intervalos de tempo fixos (evolução de tempo contínuo), o mecanismo de trocar mensagens SEGURO pode ser utilizado para impedir que os relógios dos *PLs* se distanciem muito uns dos outros.

Um PL_i só pode avançar de um intervalo para o seguinte, depois que todos os seus vizinhos estejam SEGUROS em relação ao intervalo presente. Suponhamos que PL_i já havia recebido mensagens SEGURO_$(j, i, t, \rho_j(t))$ de todos os seus vizinhos e avançado para o intervalo $[t, t + \Delta t)$. Se um PL_j vizinho que já havia enviado uma mensagem SEGURO_$(j, i, t, \rho_j(t))$ para PL_i receber um straggler de outro *PL*, fizer um *rollback* para um intervalo anterior a t e enviar um ou mais STRAGGLERS com TVR pertencente a um intervalo anterior, PL_i fará um *rollback* para um intervalo anterior, *revogando* a decisão de avançar para $[t, t + \Delta t)$.

Veremos na próxima seção que este mecanismo garante um limite superior para o valor do maior atraso possível entre os relógios dos *PLs*, proporcional ao *diâmetro* do grafo de *PLs* que representa o sistema físico e ao tamanho

do intervalo de tempo. Também veremos que isto fornece um limite inferior para o valor de TVG (TVG_{inf}) do sistema em qualquer instante de tempo real, que os *PLs* podem calcular a partir do valor dos seus relógios locais.

Esta última propriedade limita a freqüência com a qual TVG deve ser calculado e pode até tornar o cálculo desnecessário, se a quantidade de memória disponível for suficiente para armazenar listas de estados e eventos, necessárias ao mecanismo de sincronização, entre o valor do relógio do *PL* e TVG_{inf}. Todos os estados e eventos com *timestamps* anteriores a TVG_{inf} podem ser descartados (procedimento chamado de *fossil collection* na literatura). O tamanho do intervalo de tempo pode ser escolhido, de forma a compatibilizar a quantidade de paralelismo oferecido pela aplicação, o número médio de eventos que ocorrem em um intervalo e a quantidade de memória necessária para armazenar o estado do sistema simulado (ou seja, a quantidade de memória necessária ao mecanismo de sincronização). O paralelismo intrínseco da aplicação pode ser medido indiretamente pela freqüência de *rollbacks* e só se torna necessário fazer cálculos de TVG quando a aplicação ou o sistema requisitam memória e esta não está mais disponível.

As quantidades que determinam o tamanho ideal do intervalo (freqüência de *rollbacks*, número de eventos por intervalo, e memória necessária para armazenar estados) são dependentes do problema que se está simulando, mas podem ser medidas localmente durante a execução, pelo sistema simulador. Isto possibilita um ajuste, em tempo de execução, do tamanho do intervalo, permitindo que o mecanismo possa ser usado como um método para simulação paralela e distribuída de *propósito geral*.

4.4 Corretude do Algoritmo

Nesta seção apresentamos a prova de que o algorimo de intervalos temporais revogáveis apresenta as propriedades necessárias para executar a simulação corretamente. Estas propriedades básicas são: ausência de *deadlock*, a execução progride no tempo e termina em tempo finito e a execução reproduz o comportamento do modelo do sistema físico corretamente.

Além destas propriedades, comuns a todos os mecanismos de sincronização de simulações paralelas e distribuídas, mostramos uma característica particular de nosso algoritmo, que faz com que os *PLs* evoluam no tempo mais uniformemente. O algoritmo garante que o maior atraso entre os relógios locais dos *PLs*, em qualquer instante de tempo real durante a execução, nunca é maior que $(d+1)\Delta t$, onde d é uma medida do *diâmetro* do grafo (veja a definição 3 da subseção 4.4.3) que representa o sistema físico sendo simulado e Δt é o tamanho do intervalo de tempo fixo, segundo o qual o sistema evolui de forma dirigida por tempo.

4.4.1 Prova de Ausência de "Deadlock"

Para progredir no tempo (avançar o relógio), um PL_i deve receber um de três tipos de mensagens (veja protocolos 4.11, 4.12 e 4.13, executados pelos *PLs*): EVENTO_(j,i), INICIE_NOVO_INTERVALO_$(i,t,estados_dos_vizinhos_i(t))$ ou COMPLETE_INTERVALO_ATUAL_(i,t). O envio de mensagens EVENTO_(j,i) e COMPLETE_INTERVALO_ATUAL_(i,t) depende do próprio PL_i, pois os eventos lhe são despachados assim que ele envia PRONTO_$(i,t,estado_i(t))$ para o simulador. As mensagens COMPLETE_INTERVALO_ATUAL_(i,t) são envi-

adas ao PL_i , quando a próxima mensagem na sua *lista_de_msgs_recebidas*$_i$ tiver *timestamp* pertencente ao intervalo seguinte. Desta forma, o envio destes dois tipos de mensagem não pode ser bloqueado por outros *PL*s.

O envio de INICIE_NOVO_INTERVALO_$(i, t, estados_dos_vizinhos_i(t))$ para PL_i seria bloqueado se o simulador local de PL_i não recebesse mensagens SEGURO_$(j, i, t, \rho_j(t))$ de todos os seus vizinhos PL_j. Mostraremos a seguir que o algoritmo impede que esta situação aconteça. É importante notar que, como a *lista_de_msgs_recebidas*$_i$ de cada PL_i é inicializada com um evento (∞, e_∞), quando ele termina a execução de seus eventos no intervalo atual, ele termina o intervalo e fica PRONTO, ou para receber um STRAGGLER ou para esperar receber seguros dos vizinhos e continuar a executar eventos (procedimento Envie_Proximo_Evento_Para_o_PL_i). Agora provamos o seguinte teorema.

Teorema 1 *Todos os* PL*s receberão mensagens* SEGURO *de seus vizinhos referentes a todos os intervalos de tempo, de forma que nenhum* PL_i *ficará bloqueado por não receber mensagens* SEGURO *de seus vizinhos.*

Prova: Há três situações possíveis para os PL_j vizinhos de PL_i:

i) Há vizinhos PL_j no *futuro* de PL_i com $t_j > t_i$.

Ao terminar os intervalos anteriores a $t_i + \Delta t$, PL_j enviou uma mensagem ESTADO_PRONTO_$(j, t^w, \rho_j(t^w), estado_j(t^{w-}))$ para seu simulador, que enviou mensagens SEGURO_$(j, i, t^w, \rho_j(t^w))$, $\forall t^w \leq t_j$ para todos os vizinhos de PL_j, incluindo PL_i. O simulador local de PL_i, por sua vez, recebeu estas mensagens de SEGURO e tratou-as segundo

o protocolo da figura 4.6, marcando estes vizinhos PL_j como SEGUROs em relação aos intervalos anteriores a $t_i + \Delta t$. Se estes PL_j não sofrerem *rollback* para um intervalo anterior a $t_i + \Delta t$, caindo no caso iii) que trataremos adiante, eles continuarão seguros em relação a qualquer $t^w \leq t_i + \Delta t$, (o valor de $vizinho\_\text{SEGURO}_i(j, t^w)$ será verdadeiro $\forall t^w$), e portanto, estes PLs não bloqueiam PL_i.

ii) Há vizinhos PL_j de PL_i no *presente*, com $t_j = t_i$.

Se algum destes sofrer *rollback* para um intervalo anterior, caem no caso iii) que trataremos a seguir.

Os PL_j que não sofrerem *rollback* para um intervalo anterior, executarão suas listas de eventos e, em um tempo finito, seu próximo evento estará no intervalo seguinte. Quando isto acontecer, cada PL_j se tornará SEGURO e enviará uma mensagem $\text{SEGURO\_}(j, i, t_i + \Delta t, \rho_j(t_i + \Delta t))$ (veja o protocolo na figura 4.5) para PL_i desbloqueando-o.

iii) Há vizinhos PL_j de PL_i no *passado* de PL_i, com $t_j < t_i$.

Estes vizinhos devem ter sofrido *rollback* e não estão bloqueados por PL_i que está no futuro (não há perigo de ciclo de bloqueio $PL_i \Leftrightarrow PL_j$).

Se PL_j sofreu *rollback*, mas não enviou eventos para PL_i, posteriores ao *rollback*, PL_i considerará que PL_j continua seguro em relação a $t^w = t_i$, pois o envio de uma mensagem $\text{NOTIFIQUE\_ROLLBACK\_}(j, i, t^w)$ não implica em que PL_j não será mais considerado SEGURO por PL_i (veja o protocolo na figura 4.7) e, portanto, PL_j não está bloqueando PL_i em intervalos anteriores a t_i.

Caso PL_j tenha sofrido *rollback* e enviado um STRAGGLER para PL_i com TVR no intervalo $[t^w, t^w + \Delta t)$, $t^w < t_i$, este STRAGGLER, além

de causar um *rollback* em PL_i (veja o protocolo na figura 4.4), causará a marcação de PL_j como não seguro a partir do intervalo $[t^w, t^w + \Delta t)$ em PL_i (ver procedimento de *rollback* 4.9). O PL_i então estará no mesmo intervalo que PL_j estava quando enviou o STRAGGLER e terá que esperar que PL_j esteja seguro em relação a este novo intervalo $[t_i = t^w, t^w + \Delta t)$ (isto é equivalente ao item ii) que já analisamos).

Se PL_j sofreu *rollback* para um intervalo anterior a $[t_i - \Delta t, t_i)$ antes de enviar SEGURO_$(j, i, t_i + \Delta t, \rho_j(t_i + \Delta t))$ e não causou *rollback* em PL_i, PL_i ficará bloqueado no intervalo $[t_i, t_i + \Delta t)$ até que PL_j volte a executar este intervalo novamente e fique seguro em relação a ele.

O *PL* mais atrasado na rede PL_{strag}, a partir de um certo instante, já terá recebido todas as mensagens enviadas para ele com TVR $<$ $relógio_{strag}$ e não mais poderá sofrer *rollback*. Ele então progredirá, liberando seus vizinhos para o próximo intervalo $[t_{strag} + \Delta t, t_{strag} + 2\Delta t)$, através do envio de SEGURO_$(strag, k, t_{strag} + \Delta t, \rho_{strag}(t_{strag} + \Delta t))$. Estes vizinhos PL_k, por sua vez, progredirão dentro do próximo intervalo e, se houver novo PL_{strag}, o processo se repetirá liberando novos vizinhos, até que os PL_j que bloqueiam PL_i estejam SEGUROS em relação ao intervalo $[t_i - \Delta t, t_i)$, passem para $[t_i, t_i + \Delta t)$, caindo no caso ii) que já analisamos, e desbloqueiam PL_i em tempo finito.

4.4.2 Prova de que o Algoritmo Progride no Tempo

Mostramos na subseção anterior que o mecanismo de bloqueio parcial ao final de cada intervalo não causa *deadlock* (todos os *PLs* acabam avançando para o próximo intervalo). Precisamos então mostrar que os *PLs* progridem dentro

de cada intervalo, ou seja que eles não ficam ciclicamente sofrendo *rollback*.

Definição 1 *O Tempo Virtual Global (TVG), em um determinado instante de tempo real, é o mínimo entre todos os valores de relógio$_i$ $\forall i$ e o TVE de qualquer mensagem* EVENTO_(j,i), NOTIFIQUE_ROLLBACK_(j,i,t) *ou* SEGURO_$(j,i,t,\rho_j(t))$, *para qualquer j, i ou t, enviada mas ainda não recebida.*

As mensagens EVENTO_(j,i), NOTIFIQUE_ROLLBACK_(j,i,t) e SEGURO_$(j,i,t,\rho_j(t))$ são as únicas que podem causar *rollbacks*. O TVG corresponde a uma espécie de relógio global que marca o menor tempo de evento possível no sistema. Provamos agora a seguinte propriedade de TVG, tal como ele foi definido acima.

Teorema 2 *O Tempo Virtual Global (TVG) do sistema físico, simulado pelo algoritmo de Intervalos Temporais Revogáveis, nunca decresce.*

Prova: O TVG só poderia retroceder se *relógio$_i$* de algum *PL* retornasse a um valor inferior a TVG, por causa de uma mensagem de um dos três tipos que citamos acima, ou se algum *PL* enviasse uma mensagem com TVE no passado do seu relógio local, o que é proibido. Basta mostrar que quando o *relógio$_i$* de um PL_i regride, ele o faz para um instante de tempo virtual igual ou posterior a TVG.

Um *relógio$_i$* só regride quando há um *rollback*. No pior caso, todos os processos voltam para o mesmo tempo virtual, para o qual o processo que provocou inicialmente o *rollback* (PL_{inic}) voltou. Mas, o instante para o qual um PL_i volta, como conseqüência da recepção de um STRAGGLER de

PL_{inic}, é o TVR da mensagem que causou o *rollback*, que é maior ou igual ao seu TVE. Mas, pela definição de TVG, este TVE é maior ou igual a TVG no instante de tempo real em que o *rollback* acontece. Logo, nenhum PL_i pode voltar $relógio_i$ para um tempo virtual menor que TVG e TVG nunca decresce.

Apresentamos agora a prova do seguinte teorema que garante o progresso do algoritmo.

Teorema 3 *O Tempo Virtual Global (TVG) do sistema físico, simulado pelo algoritmo de Intervalos Temporais Revogáveis, cresce e o algoritmo termina.*

i) Se TVG $= t_i + \Delta t$ de uma mensagem SEGURO_$(i, j, t_i + \Delta t, \rho_i(t_i + \Delta t))$, enviada por um PL_i, esta mensagem poderia ou não estar bloqueando PL_j.

   a) A mensagem não estava bloqueando PL_j ($vizinho\_SEGURO_j(i, t + \Delta t)$ já é verdadeira). Esta mensagem alcançará seu destino e, ou não causa nada, ou causa um *rollback* em PL_j (veja a figura 4.6). Se ela causar um *rollback* em PL_j, este ou outro PL eventualmente escalonarão um evento futuro para PL_j, ou ele progredirá para o próximo intervalo de tempo quando todos os vizinhos estiverem seguros. Se a mensagem não causa nada, TVG eventualmente assumirá um valor maior ou igual a $t_i + \Delta t$, através da recepção de outra mensagem que não um SEGURO e caímos em um dos outros casos, que analisaremos a seguir.

   b) A mensagem estava bloqueando PL_j. Assim que a mensagem chega, ela desbloqueia PL_j, que passa então a simular o intervalo $[t_i +$

$\Delta t, t_i + 2\Delta t$), e TVG assumirá um novo valor maior que $t_i + \Delta t$, como conseqüência desta evolução do relógio de PL_j.

ii) Se TVG $= T$ de uma mensagem NOTIFIQUE_ROLLBACK_(i, j, T), esta mensagem será recebida, causando ou não um *rollback*.

a) A mensagem não causa *rollback*. Um novo mínimo entre *timestamps* de mensagens e relógios locais é feito, através da recepção de outras mensagens futuras e de eventos gerados pelo próprio PL_j e TVG acabará progredindo.

b) A mensagem causa *rollback*. Ao receber a mensagem, PL_j acerta seu relógio $relógio_j = T$. Neste instante, TVG $= relógio_j$, situação que analisamos no item iii).

iii) Se TVG $= relógio_i$, para algum PL_i, este PL_i não poderá sofrer *rollback* (TVG não decresce). Se PL_i recebe mensagens com TVR $= relógio_i$, nada acontece. Como a computação é otimista e devido à hipótese 5 que fizemos no início da seção 4.2, PL_i eventualmente escalonará um evento com $TVR > relógio_i$, ou receberá SEGURO_$(j, i, t_i + \Delta t, \rho_j(t_i + \Delta t))$, de todos os vizinhos PL_j (estes não podem sofrer *rollback* para antes de TVG $= relógio_i$), fazendo com que $relógio_i$ avance e TVG progrida.

iv) Se TVG $=$ TVE de uma mensagem EVENTO_(j, i), com TVE $=$ TE e TVR $=$ TR, esta mensagem será recebida e eventualmente processada, pois o PL que a recebe não poderá sofrer *rollback* para um instante anterior a TE, de forma a desfazê-la. Como conseqüência de sua recepção e processamento por PL_i, $relógio_i :=$ TVR $\geq$ TVE e, ou TVG progride, ou caímos no caso iii) de TVG $= relógio_i$, onde TVG também acaba progredindo.

Resumindo, enquanto o simulador processa *rollbacks*, estes ocorrem para um instante maior ou igual a TVG e TVG possivelmente não mudará (certamente não diminuirá). Eventualmente, a mensagem com menor TVE > TVG corresponderá a um evento, cujo processamento implicará na atualização do relógio de um *PL* com menor valor de $relógio_i$. Devido à hipótese 5 da seção 4.2, algum *PL* gerará um evento com *timestamp* maior que TVE = TVG ou TVG será igual ao *timestamp* de uma mensagem SEGURO, do fim do intervalo ao qual $relógio_i$ pertence, e o TVG do sistema então aumentará para outro valor, maior que o anterior, progredindo.

Quando TVG $\geq$ *tempo_de_termino*, o algoritmo terminará, devido à condição do loop principal do simulador (veja o algoritmo na figura 4.2).

4.4.3 Prova de que Existe um Atraso Máximo Entre *PL*s

Nesta subseção, mostramos que o algoritmo de Intervalos Temporais Revogáveis sincroniza a rede de *PL*s parcialmente, não permitindo que os *PL*s se distanciem muito uns dos outros no tempo sem, no entanto, criar barreiras de sincronização globais. O mais interessante é que esta propriedade é garantida por um mecanismo distribuído, diferentemente dos mecanismos que encontramos na literatura (veja a seção seguinte). Primeiramente precisamos fazer duas definições.

Definição 2 *Em um grafo de PLs que representa o sistema físico sendo simulado, chamaremos de* **distância** *entre dois PLs quaisquer, interligados de forma que a computação de um deles possa afetar o outro, ao tamanho do*

menor caminho, em número de arestas, entre os PLs do par.

Definição 3 *Simbolizamos por d, a* **maior** *distância entre PLs, para qualquer par de PLs, no grafo de PLs que representa o sistema físico sendo simulado. Podemos também chamar d de* **diâmetro** *do grafo.*

O seguinte teorema garante um limite superior para o atraso entre os relógios dos *PLs*.

Teorema 4 *Se a relação de dependência entre os PLs é mútua, ou seja se existe o canal de comunicação* $\mathrm{PL}_i \Longrightarrow \mathrm{PL}_j$ *e o canal* $\mathrm{PL}_i \Longleftarrow \mathrm{PL}_j$*, o atraso máximo no valor da variável* t_j*, que marca o intervalo corrente de* PL_j*, em relação ao valor de* t_i *de outro* PL_i*, para qualquer par (*PL_i*,* PL_j*) no grafo de PLs que representa o sistema físico sendo simulado, em qualquer instante de tempo real, é igual a* $(d+1)\Delta t$*.*

Aresentamos agora a prova indutiva deste teorema.

Base: O teorema vale para $d = 1$**.** Esta distância d corresponde a uma rede com dois *PLs* (supondo que os canais são bidirecionais). Denotaremos por t^w, onde w é um inteiro, ao w-ésimo intervalo de tempo da simulação sendo executada. Seja $t_i = t_j = t^{w-1}$, de forma que ambos PL_i e PL_j executam o passo $[t^{w-1}, t^w)$.

O protocolo de recepção de mensagens seguro, mostrado na figura 4.6, só permite que PL_j inicialize a simulação de um intervalo $[t^w, t^{w+1})$ novo, quando PL_i estiver seguro em relação ao intervalo $[t^{w-1}, t^w)$. O

PL_i pode ficar seguro e enviar uma mensagem SEGURO_$(i, j, t^w, \rho_i(t^w))$ para PL_j, que por sua vez, causa um *rollback* em PL_i para $[t^{w-1}, t^w)$, de forma que $t_i = t^{w-1}$. Se, logo em seguida, PL_j fica seguro, ele avança para $[t^w, t^{w+1})$. Se ele não sofrer *rollback* enquanto executa este intervalo, assim que ele fica seguro em relação a $[t^w, t^{w+1})$, ele avança o relógio para $t_j = t^{w+1}$ (veja o protocolo de despacho de eventos na figura 4.8). Agora, PL_j espera que PL_i fique seguro em relação ao intervalo $[t^w, t^{w+1})$, e envie SEGURO_$(i, j, t^{w+1}, \rho_i(t^{w+1}))$. Neste instante de tempo real, $t_j - t_i = t^{w+1} - t^{w-1} = (t^w + \Delta t) - (t^w - \Delta t) = 2\Delta t = (d+1)\Delta t$. Mas agora, PL_j só poderá simular o intervalo $[t^{w+1}, t^{w+2})$ e avançar seu relógio para $t_j = t^{w+2}$, quando PL_i estiver seguro em relação a $[t^w, t^{w+1})$, e não poderá, sem que ele próprio sofra *rollback* para um intervalo anterior e atrase t_j, causar *rollback* de PL_i para o intervalo $[t^{w-2}, t^{w-1})$. Logo, o atraso máximo entre t_j e t_i é $t_j - t_i = 2\Delta t = (d+1)\Delta t$.

É importante notar que este resultado só vale, se a relação de dependência é mútua, ou seja se existe o canal $PL_i \Longrightarrow PL_j$ e o canal $PL_i \Longleftarrow PL_j$. Se, por exemplo, não existisse o canal $PL_i \Longleftarrow PL_j$, PL_i enviaria mensagens SEGURO e avançaria sem restrições.

Hipótese Indutiva: Supomos que o teorema vale para um grafo de *PL*s que representa o sistema físico sendo simulado, com maior distância $d = n$.

Passo Indutivo: O Teorema vale para $d = n + 1$. Em um caminho que represente a maior distância d em um grafo de *PL*s, como aquele mostrado na figura 4.15, para que PL_j esteja simulando o intervalo $[t^w, t^{w+1})$, é necessário que PL_j tenha recebido SEGURO_$(k, j, t^w, \rho_k(t^w))$

de PL_k. No pior caso, PL_j pode ter enviado um STRAGGLER com TVR $< t^w$ para PL_k, enquanto SEGURO_$(k, j, t^w, \rho_k(t^w))$ estava em trânsito e, logo em seguida, avançado para $[t^w, t^{w+1})$, com $t_j = t^w$, enquanto PL_k continuou no intervalo $[t^{w-1}, t^w)$, com $t_k = t^{w-1}$. Se PL_j ficar seguro em relação a $[t^w, t^{w+1})$, ele avança t_j para $t_j = t^{w+1}$ e fica aguardando uma mensagem SEGURO_$(k, j, t^{w+1}, \rho_k(t^{w+1}))$ de PL_k. Neste instante de tempo real, o atraso $t_j - t_k$ é igual a $t^{w+1} - t^{w-1} = 2\Delta t$. Mas, pela hipótese indutiva, no pior caso, PL_k pode ter acabado de simular $[t^{w-1}, t^w)$, enquanto PL_i está simulando $[t^{w-(n+1)}, t^{w-n})$, de forma que $t_k - t_i = t^w - t^{w-(n+1)} = t^w - (t^w - (n+1)\Delta t) = (n+1)\Delta t$.

Se, ainda no pior caso, PL_i causa um *rollback* em cascata, que faz com que PL_k tenha que sofrer *rollback* para $[t^{w-(n+1)}, t^{w-n})$ e PL_j não precise sofrer *rollback* (PL_k não enviou mensagens para PL_j anteriores a t^w), t_k terá o novo valor $t_k = t_w - (n+1)\Delta t$, enquanto $t_j = t^{w+1}$. De forma que, no pior caso, $t_j - t_k = t_j - t_i = t^{w+1} - (t^w - (n+1)\Delta t) = (n+2)\Delta t$.

Além disto, PL_j não poderá avançar para $[t^{w+1}, t^{w+2})$, antes que PL_k esteja seguro em relação a $[t^w, t^{w+1})$, e isto só ocorrerá, pela hipótese indutiva, quando todos os *PLs* de quem ele depende, até PL_i, estejam seguros pelo menos em relação a $[t^{w-n}, t^{w-(n-1)})$ (lembramos que, neste instante de tempo real, $t_k - t_i = t^{w+1} - t^{w-n} = (n+1)\Delta t$). Portanto, quando PL_k envia SEGURO_$(k, j, t^{w+1}, \rho_k(t^{w+1}))$ para PL_j, t_j poderá ser igual a t^{w+2} e $t_j - t_l$, onde PL_l é qualquer PL no caminho entre PL_j e PL_i, será *no máximo* igual a $t_j - t_i = t^{w+2} - t^{w-n} = (n+2)\Delta t$.

É importante que a *distância* entre dois *PLs* seja definida como o menor caminho entre eles, pois, por exemplo, para o grafo da figura 4.16, se o atraso máximo fosse medido pelo caminho 1, $\Delta t = 1$ e $t_4 = 5$, $t_4 - t_1$

poderia ser igual a $4\Delta t$, de forma que t_1 pode ser igual a 1. Mas, pelo caminho 2, para que $t_4 = 5$, é necessário que, no mínimo, $t_1 = 3$ o que é uma contradição.

Figura 4.15: Caminho que representa a maior distância d em um grafo de *PLs*.

Esta última propriedade do atraso máximo entre *PLs*, dada pelo teorema 4, é uma propriedade importante do algoritmo de intervalos temporais revogáveis, porque fornece um piso inferior para o valor do Tempo Virtual Global (TVG), que pode ser calculado a partir do relógio local de cada *PL*. Se o atraso máximo no valor da variável t_i de PL_i, em relação a t_j de um outro PL_j qualquer no grafo de *PLs*, tem um limite superior dado por $(d+1)\Delta t$, então PL_i pode calcular que TVG nunca é menor que $t_i - (d+1)\Delta t$. Se o processador que executa PL_i tem memória suficiente, para armazenar todos os estados do sistema em um intervalo de tempo de tamanho $(d+1)\Delta t$, o cálculo de TVG se torna desnecessário, porque todos os estados e mensagens com *timestamps* menores que $t_i - (d+1)\Delta t$ podem ser descartados. Também todas as ações definitivas, como entrada e saída, com *timestamps* menores que $t_i - (d+1)\Delta t$, podem ser realizadas.

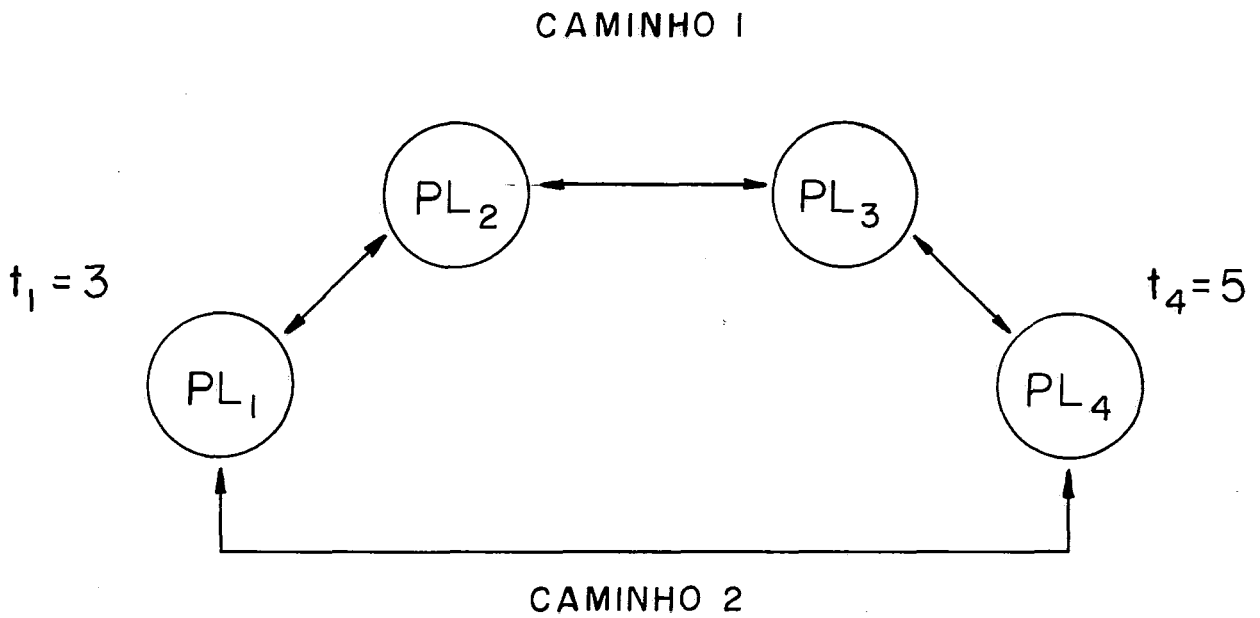


Figura 4.16: Exemplo de atrasos máximos para dois caminhos entre *PLs*.

4.4.4 Prova de que o Algoritmo Simula o Sistema Físico Corretamente

Cada PL_i executa a simulação local, usando o algoritmo de lista de eventos seqüencial, com eventos correspondentes às mudanças de estado ao final de cada intervalo de tempo Δt. O PL_i executa na direção de tempo crescente, supondo que toda a informação de que ele dispõe localmente está correta, ou seja, que ele já recebeu todas as mensagens que deveria receber até $relógio_i$.

Quando se confirma que esta suposição estava errada [1], todo o efeito da computação realizada com bases em informação errada é desfeito: um estado correto é restaurado, mensagens erradas são descartadas, $relógio_i$ é atrasado para um tempo correspondente ao estado restaurado correto (veja o procedimento da figura 4.9), e a execução prossegue a partir da nova informação correta. Como isto é feito sempre que uma estimativa errada foi feita, o algoritmo não fica em *deadlock* e progride no tempo (como demonstramos nas subseções anteriores), e o algoritmo seqüencial simula o sistema físico corretamente, o algoritmo paralelo de intervalos temporais revogáveis também o faz.

4.5 Trabalhos Relacionados

Como mencionamos nos capítulos anteriores, vários algoritmos já foram propostos para a simulação paralela de eventos discretos [Ch79, Ba91, Je85, So89]. Além disso, Awerbuch apresentou seus sincronizadores, com os quais se pode obter um algoritmo assíncrono, que executa em um computador paralelo assíncrono, a partir de um algoritmo síncrono [Aw85]. Estes sincronizadores podem ser usados para realizar simulação distribuída dirigida por tempo. Nenhum destes algoritmos tem a característica de sincronizar aplicações, que apresentam a abordagem de evolução temporal híbrida (dirigida por tempo e eventos). Embora nosso objetivo inicial tenha sido o desenvolvimento de um algoritmo para a simulação de sistemas com evolução

[1] Isto acontece quando PL_i recebe um STRAGGLER, uma mensagem SEGURO_$(j, i, t_j, \rho_j(t_j))$ tal que $t_j < t_i$ e $\rho_j(t_j)$ é muito diferente do valor de $\rho_j(t_j)$ que PL_i já havia recebido anteriormente, ou uma mensagem NOTIFIQUE_ROLLBACK_(j, i, t) com $t < relógio_i$, sendo que PL_i já havia executado eventos recebidos de PL_j com *timestamps* posteriores a t.

temporal híbrida, verificamos que o método se presta muito bem à função de limitar o otimismo de mecanismos otimistas, alvo que tem sido perseguido pela comunidade nos últimos anos. Apresentamos a seguir algumas das principais propostas de mesclar métodos conservadores com otimistas, para obter melhores desempenhos, diminuindo o tempo real de simulação.

No algoritmo de Moving Time Windows [Br87, So89, So90], uma janela de tamanho W, $[T, T+W)$, é escolhida e todos os eventos com *timestamps* neste intervalo são executados, de forma otimista. Eventos fora da janela não são escalonados para execução e o limite inferior da janela se move constantemente, na direção de tempo crescente, para o instante de ocorrência do evento com menor *timestamp* da simulação. Este método poderia ser adaptado ao nosso problema se, por exemplo, o limite inferior da janela T só se movesse depois que todos os eventos do sistema com *timestamps* menores que $T+W$ já tivessem sido executados. Ele não é conveniente, porque o método usado para detectar o limite inferior T (no nosso caso, o instante em que todos os eventos no intervalo atual já foram executados) usa variáveis globais, compartilhadas e, portanto, só é adequado para máquinas paralelas de memória compartilhada e não é escalável.

Alguns outros trabalhos se propõem a limitar o otimismo de mecanismos otimistas para simulações de eventos discretos. Em [Ni92a], Nicol apresenta um mecanismo para criar uma barreira de sincronização global a todos os processadores, no final de cada intervalo de tempo de determinado tamanho. Em seu mecanismo, os processadores *entram* na barreira de forma otimista, ou seja, eles podem sofrer *rollback* para fora da barreira, mas cada processador só pode avançar para o próximo intervalo (*sair* da barreira na direção de

tempo crescente), quando tem certeza de que todos os outros processadores já alcançaram a barreira. Não há *rollback* de um intervalo para outro anterior. O algoritmo apresenta uma complexidade de tempo de $\mathcal{O}(\log^2 P)$, onde P é o número de processadores, para criar a barreira. Nossa abordagem é mais escalável, por criar *barreiras* localizadas (cada *PL* só precisa de informação de seus vizinhos imediatos para avançar) e permitir o avanço otimista de uma janela de tempo para outra. A complexidade de tempo para criar a barreira em nosso algoritmo é $\mathcal{O}(1)$.

Em [Tu92], Turner e Xu apresentam o algoritmo de *Bounded Time Warp*, onde também se busca uma sincronização global do sistema, ao final de intervalos de tempo, com um mecanismo de passagem de token por um anel que envolve todos os processadores. Como no algoritmo de Nicol, cada *PL* só pode avançar para o intervalo seguinte quando todos os outros *PLs* já terminaram o intervalo atual (alcançaram a barreira). No algoritmo de Nicol, a sincronização é feita percorrendo-se uma estrutura de árvore binária balanceada que envolve todos os *PLs*. Já no algoritmo de Turner e Xu, os *PLs* são sincronizados por uma estrutura em anel.

O método apresentado em [Ba90] se baseia na premissa de que, durante uma simulação paralela e distribuída, um ou mais processadores avançam mais do que os outros em tempo de simulação e, como conseqüência, sofrem mais *rollbacks* (recebem mais eventos com *timestamps* no passado do relógio local). Do ponto de vista dos processadores, isto significa que os erros de causalidade ocorrem em clusters. Os autores se baseiam nestes fatos, para desenvolver um mecanismo de escalonamento de *PLs*, que busca reduzir a freqüência de ocorrência destes clusters, detectando-os. O método propõe

o uso de uma *Janela de Bloqueio*, que é um intervalo de tempo durante o qual o *PL* fica bloqueado sem executar eventos e avançar seu relógio. O tamanho da janela é calculado usando a história recente do *PL* para estimar o comportamento futuro. Se o *PL* passou uma parte razoável de tempo real, δt, recente executando erros de causalidade, ele fica bloqueado por um tempo proporcional a δt.

Em todos os mecanismos que buscam limitar o otimismo de mecanismos otimistas, se busca melhorar o desempenho de simulações onde pode ocorrer *cascatas* de *rollbacks* (o número de *PLs* participantes do *rollback* aumenta indefinidamente). Também existe um padrão de comportamento indesejado de simulações puramente otimistas chamado de *eco*, no qual a amplitude (tamanho) do *rollback* aumenta indefinidamente, que são evitados por estes mecanismos limitadores de otimismo. Em simulações onde estes comportamentos se manifestam de forma sensível, os métodos citados apresentam bons resultados de desempenho, quando comparados ao mecanismo de *Time Warp* puro, para as aplicações estudadas.

Outro mecanismo para limitar otimismo foi proposto por Steinman em [St91]. Neste trabalho, ele define ciclos de tempo (janelas), cujo tamanho depende dos tamanhos dos intervalos de tempo entre o instante de ocorrência de um evento e os *timestamps* dos eventos que ele pode gerar. Este intervalo tem tamanho variável e definido em tempo de execução. Para definir o limite inferior da janela, cada *PL* executa seu intervalo baseado em estimativas locais e, ao final do intervalo estimado localmente, bloqueia e faz um broadcast de sua estimativa para todos os outros *PLs* da rede. Caso um *PL* tenha executado além do limite inferior global para a janela, ele sofre um *rollback*. O

método envolve um mecanismo *preguiçoso* de envio de mensagens. As mensagens que foram geradas em um determinado intervalo só são enviadas após o cálculo do limite inferior global para a janela e apenas se foram geradas por eventos com *timestamps* inferiores ao novo limite calculado. Este mecanismo também envolve uma sincronização global da rede ao final de cada intervalo, com mensagens enviadas entre todos os possíveis pares de processadores. O autor sugere o uso de hardware dedicado para fazer esta sincronização. O mecanismo é eficiente para aplicações nas quais os eventos gerados, como conseqüência da execução de outros eventos, se distanciam no tempo. Em [St94a], Steinman verifica esta última afirmação desenvolvendo um modelo analítico aproximado de desempenho, para um problema de sistemas de filas. Um bom desempenho também foi obtido para sistemas de simulação militares relatado em [St94].

O mecanismo apresentado em [Ha94] propõe um esquema de bloqueio, com uma janela temporal calculada apenas com informação disponível localmente, mas específica de sistemas de filas. O método usa parâmetros de sistemas de filas, como o tempo (de simulação e real) médio entre chegadas de eventos por um determinado canal, para decidir se deve bloquear ou executar de forma otimista, um *PL* que não dispõe de toda a informação necessária para prosseguir. Desta forma, o método ajusta dinamicamente a política de avanço temporal (otimista ou pessimista) e obtem bons resultados para aplicações, nas quais se conhece as distribuições dos tempos médios entre *timestamps* de eventos.

Bagrodia e Jha apresentam uma proposta de integração dos mecanismos otimista e conservador em um único protocolo de propósito geral, permitindo

que partes diferentes do sistema possam ser simuladas usando protocolos diferentes e que estes protocolos sejam trocados dinamicamente, durante a execução [Jh94]. Eles formalizam o conceito de Mecanismo de Controle Local (MCL) (bloqueante em protocolos conservadores e não bloqueante em mecanismos otimistas) e Mecanismo de Controle Global (MCG) (cálculo de TVG em métodos otimistas e uso de mensagens nulas em métodos conservadores) e permitem combinações diferentes destes mecanismos para gerar novos protocolos. Os autores citam situações, nas quais seriam mais favoráveis o uso de um ou outro protocolo, mas dizem que ainda estão no processo de desenvolvimento de uma heurística, para decidir dinamicamente qual o mecanismo a ser usado em cada instante da simulação, por cada parte do sistema. Eles sugerem, por exemplo, a monitoração da freqüência de *rollbacks* de *PLs* otimistas e do tempo de bloqueio de *PLs* conservadores e a troca do modo, caso estes valores ultrapassem um determinado valor. Jha e Bagrodia ainda estão implementando o mecanismo e não apresentaram resultados de desempenho.

Quanto ao estudo da paralelização de simulações de sistemas físicos modelados matematicamente pela equação de BUU, encontramos as referências [St93] e [Sc92]. Ambos os trabalhos usam a equação de BUU para modelar dinâmica de fluidos, por exemplo, para calcular o fluxo em torno de uma nave espacial, durante a entrada na atmosfera. Os autores propõem o uso do método de partículas teste, mas fazem uma aproximação mais simplificadora, supondo que todas as colisões que ocorrem em um determinado intervalo ocorrem no mesmo instante de tempo, tornando a solução dirigida apenas por tempo, do ponto de vista de sincronização. Para executar a simulação em paralelo, eles usam um mecanismo de sincronização global do tipo barreira ao final de cada intervalo, com troca de mensagens entre todos os pares de

processadores do sistema. A computação é síncrona e não otimista.

Em nosso trabalho, apresentamos uma abordagem com um tipo de janela temporal, que verifica quando a janela de um determinado processador pode progredir para o próximo intervalo, de forma otimista e distribuída. A janela temporal se move em um determinado *PL*, baseado apenas em informação localizada de vizinhos mais próximos. Não sincronizamos a rede inteira ao final de cada intervalo, o que reduz o tráfego de mensagens na rede e permitimos que os *PL*s avancem de forma otimista de um intervalo para outro, de forma que se na maior parte das vezes, a informação disponível localmente para cada *PL* está correta, o tempo durante o qual ele ficaria bloqueado é gasto fazendo computação útil. Se a suposição otimista de que o *PL* poderia se mover para o próximo intervalo se mostrar errada, o *PL* local realiza um *rollback* para um intervalo anterior correto.

Capítulo 5

Resultados de Experiências

5.1 A Implementação

Como mencionamos na seção 4.1.1, a aplicação que nos motivou a desenvolver o algoritmo foi uma colisão nuclear modelada matematicamente pela equação de BUU resolvida pelo método de partículas teste. Esta aplicação pertence a uma classe de problemas de N-corpos interagindo através de uma força de curto alcance. Após a ocorrência de um evento, envolvendo uma ou mais partículas e ao final de cada intervalo, é necessário recalcular a possibilidade e os instantes de ocorrência de colisões destas partículas cujo momento mudou com todas as outras de uma mesma repetição Monte Carlo, em uma vizinhança limitada pelo alcance da força.

É possível obter bons resultados na paralelização desta aplicação, como consequência do fato de que a força nuclear é de curto alcance. Em nosso modelo da colisão nuclear, não consideramos a força de repulsão coulombiana entre os prótons. Para aplicações onde a interação é de longo alcance, as

partículas sempre interagem com todas as outras do sistema e a complexidade para calcular estas interações é $\mathcal{O}(M^2)$, onde M é o número de partículas. Se as interações são de curto alcance e dividimos o espaço físico em r regiões, a complexidade para calcular as interações em cada região cai para $\mathcal{O}((M/r)^2)$, porque só é necessário recalcular os instantes das colisões entre cada partícula e as outras que pertencem à mesma região.

Mais especificamente, realizamos simulações da aplicação de partículas colidindo entre si no simulador distribuído que desenvolvemos (usando o método de intervalos temporais revogáveis) e também em um simulador seqüencial que implementa o algoritmo para simulação híbrida da figura 4.1. Comparamos então o desempenho do simulador seqüencial com o do simulador distribuído. De acordo com o que mencionamos no parágrafo anterior sobre o paralelismo da aplicação, na simulação seqüencial também é vantajoso dividir o espaço físico em regiões. Isto permite que um PL responsável pela simulação da região r_i determine apenas as possíveis colisões entre partículas pertencentes à r_i e não envolvendo todas as partículas do sistema.

Se estamos simulando um sistema com A nucleons, usando N partículas teste por nucleon (N repetições Monte Carlo do sistema) e $tcol$ é o tempo necessário para estimar a possibilidade de ocorrer uma colisão entre um par de nucleons, o tempo gasto para calcular as possíveis colisões entre pares de nucleons que são determinadas a cada início de intervalo, caso o domínio não seja dividido em regiões é

$$N\frac{A(A-1)}{2}tcol = N\frac{A^2-A}{2}tcol. \tag{5.1}$$

Isto se verifica porque é necessário testar a possível colisão de cada nucleon com todos os outros do domínio e apenas nucleons pertencentes à mesma

repetição Monte Carlo colidem entre si.

Se o domínio for dividido em r regiões, cada região possuirá aproximadamente NA/r partículas teste, já que as partículas são distribuídas uniformemente pelo domínio. O tempo para estimar as possíveis colisões entre dois nucleons que são determinadas a cada início de intervalo em cada região é (testando a possibilidade de colisão de cada nucleon de uma repetição com todos os outros da mesma repetição em uma região):

$$N\left(\frac{\frac{A}{r}(\frac{A}{r}-1)}{2}\right)tcol. \tag{5.2}$$

Além das colisões, deve-se testar a passagem de cada partícula de uma região para cada uma das regiões vizinhas. Para uma divisão em até $r = 8$ regiões mapeadas em um grid 3-dimensional (veja a figura 5.1), cada região tem $(r-1)$ vizinhos. Chamando de *tpass* o tempo para estimar a passagem de uma partícula de uma região para a outra, o tempo total para estimar eventos por região é então:

$$N\left(\frac{\frac{A}{r}(\frac{A}{r}-1)}{2}tcol + (r-1)\frac{A}{r}tpass\right). \tag{5.3}$$

O tempo total necessário para testar possíveis colisões e passagens de partículas entre regiões, para as r regiões, é:

$$Nr\left(\frac{\frac{A}{r}(\frac{A}{r}-1)}{2}tcol + (r-1)\frac{A}{r}tpass\right) = N\left(\frac{(\frac{A^2}{r}-A)}{2}tcol + A(r-1)tpass\right). \tag{5.4}$$

Para que o tempo da simulação seqüencial com uma divisão em r regiões seja menor que o tempo da mesma simulação sem divisões, é necessário que

a expressão 5.4 seja menor que a expressão 5.1, ou seja:

$$N\left(\frac{(\frac{A^2}{r}-A)}{2}tcol + A(r-1)tpass\right) < N\frac{A^2-A}{2}tcol. \qquad (5.5)$$

Esta desigualdade é satisfeita se $r < (A/2)(tcol/tpass)$. Embora ambos os lados da desigualdade 5.5 sejam da ordem de A^2, isto significa que, para $r < (A/2)(tcol/tpass)$, mesmo na simulação seqüencial, é vantajoso dividir o domínio físico da colisão em r regiões. O número de colisões possíveis entre dois nucleons fica proporcional a $1/r$, de forma que o ganho obtido com a redução do número de colisões a serem determinadas é maior que o custo do tratamento da passagem dos nucleons de uma região para outra.

Em uma colisão entre dois núcleos pesados, os nucleons dos núcleos são lançados uns contra os outros. Neste sistema, a densidade de partículas varia dinamicamente em cada região. Por exemplo, no instante em que os dois núcleos colidem, há uma forte concentração dos nucleons em torno do ponto onde eles se chocam [We92]. Como a carga computacional (complexidade de tempo) de um *PL* é proporcional ao quadrado do número de partículas na região pela qual ele é responsável, uma divisão do espaço tridimensional que envolve a colisão em regiões de tamanhos iguais, implica em uma divisão desigual da carga computacional entre os *PLs* e, portanto, entre os procesadores. Mas vimos que para obter bom desempenho com os algoritmos de simulação paralela e distribuída, é necessário que a carga computacional esteja bem balanceada entre processadores. O problema se agrava ainda mais, devido ao fato de que a densidade de partículas e portanto a carga variam durante a evolução temporal do sistema.

Para obtermos bons resultados de desempenho nas simulações destas co-

lisões nucleares, é necessário acoplar ao algoritmo de simulação, um algoritmo de balanceamento dinâmico de carga. Esta foi a característica do sistema que chamou nossa atenção inicialmente. O problema de realizar balanceamento dinâmico de carga de forma eficiente em computação paralela e distribuída é extremamente difícil, havendo poucas propostas na literatura.

Como nesta etapa estamos interessados em testar apenas o mecanismo de sincronização da simulação, testamos o algoritmo na simulação de um núcleo parado e aquecido, que por isto expande na direção radial de forma homogênea. Este processo acontece, por exemplo, após a colisão de um próton ou antipróton com energia cinética elevada, com um núcleo [Do92]. Dividimos o espaço que envolve a colisão em um grid tridimensional. Cada região do grid corresponde a um *PF* que foi simulado por um *PL* diferente e, como os processadores do iPSC/860 não são multiprogramados, alocamos um *PL* a cada processador.

Até oito processadores (*PLs*), dividimos a forma esférica do núcleo em um grid com até oito regiões cúbicas. A figura 5.1 mostra a divisão em regiões para 2, 4 e 8 *PLs*. Esta divisão tem uma geometria simples e minimiza a relação superfície / volume de cada região, o que também minimiza o número de eventos de passagem de partículas de uma região para a outra, em relação ao número de eventos (colisões) internos a cada região. Conseqüentemente, esta divisão também minimiza a comunicação entre *PLs*. Como o alcance da interacao entre os nucleons é relativamente grande quando comparado com o tamanho do núcleo, não usamos mais que oito divisões porque isto implicaria, por exemplo, em que uma grande parte dos nucleons pertencessem a várias regiões simultaneamente (reduzindo a independência entre os *PLs* e

seqüencializando a simulação).

Por enquanto, ainda não temos uma aplicação verdadeiramente escalável. Em uma segunda etapa, quando incluirmos balanceamento dinâmico de carga e calcularmos uma colisão entre dois núcleos, poderemos aumentar o número de *PLs* em pelo menos o dobro (estaremos simulando dois núcleos).

Nos cálculos que fizemos, usamos um intervalo de tempo $\Delta t = 1 fm/c$. Este valor é adequado para a integração da equação 4.3. O alcance máximo da interação nucleon-nucleon que utilizamos é de 0.6 fm. Na próxima seção, mostramos resultados de desempenho para as simulações que implementamos nesta etapa para testar o método de intervalos temporais revogáveis.

É importante salientar que as funções do simulador são implementadas separadamente da aplicação (veja [Mo94]). O algoritmo de intervalos temporais revogáveis pode ser implementado de forma que ele possa fornecer as funções de sincronização a qualquer aplicação que apresente o comportamento de evolução temporal especificado no início da seção 4.1. De fato, estas funções podem fazer parte de um sistema operacional distribuído (veja [Je88]). A aplicação somente é reponsável por gerar e executar eventos e enviar eventos e estados ao simulador. Embora nosso objetivo principal não tenha sido gerar um ambiente de simulação, procuramos manter esta independência entre simulador e aplicação ao máximo, de forma que possamos usar nossa implementação do algoritmo em outras aplicações no futuro.

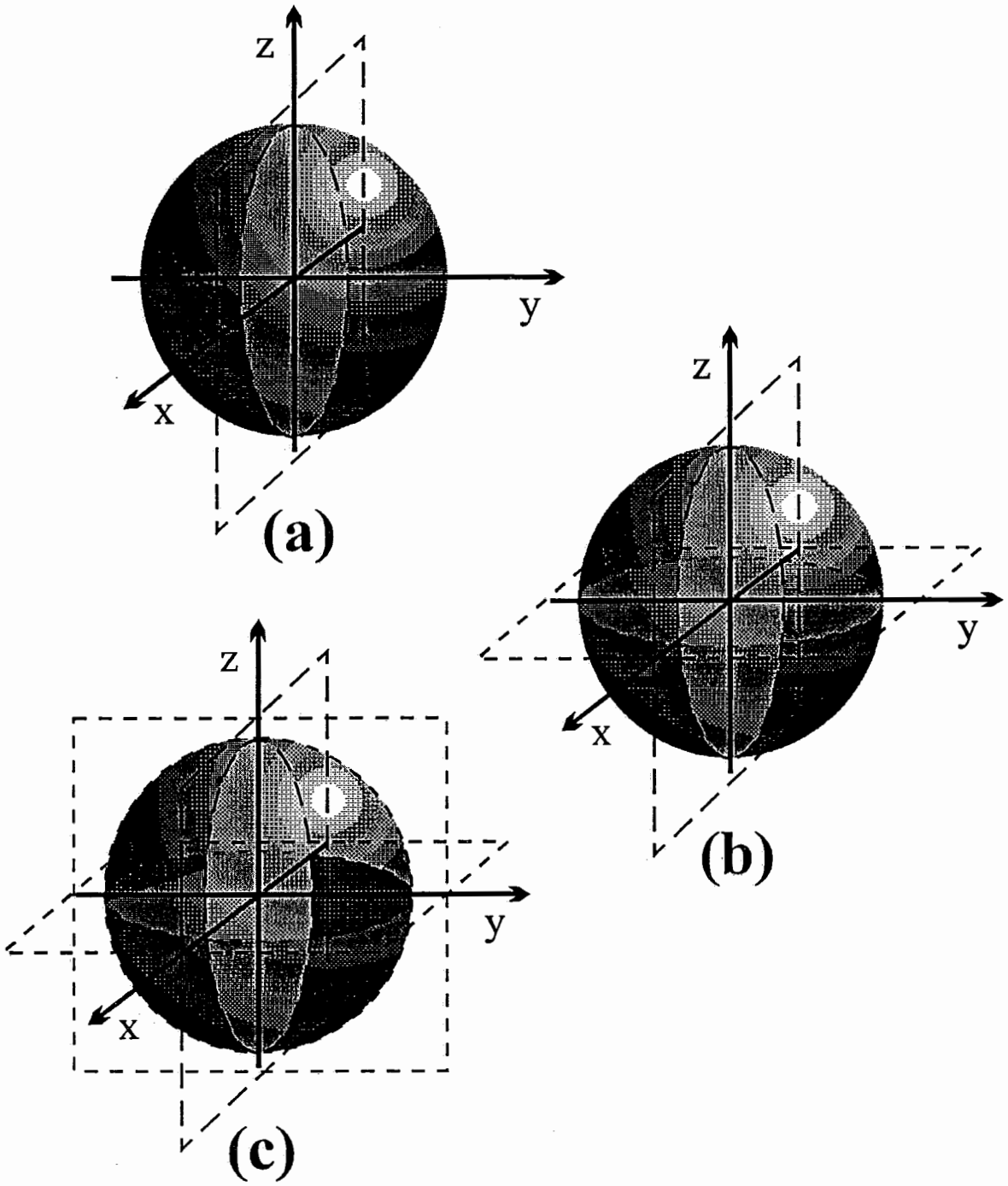


Figura 5.1: Divisão do domínio físico em 2, 4 e 8 regiões.

5.2 Desempenho

Nesta seção, apresentamos comparações dos desempenhos obtidos com o simulador distribuído que desenvolvemos (usando o método de intervalos temporais revogáveis) com os desempenhos obtidos usando um simulador seqüencial que implementa o algoritmo para simulação híbrida da figura 4.1. As medidas de tempo de execução foram realizadas em simulações de um núcleo quente em expansão, como explicamos na seção anterior e cujo comportamento descrevemos na seção 4.1.1. Também medimos alguns parâmetros do simulador distribuído como: o número de *rollbacks*, o tamanho médio dos *rollbacks*, a proporção de eventos executados que são corretos e outros que também refletem sua eficiência. Com estes resultados, tentamos entender o comportamento genérico do simulador distribuído.

Como vimos na seção anterior que o termo dominante da complexidade de tempo da simulação nuclear é proporcional ao quadrado do número de nucleons pertencentes ao núcleo, A^2, mantivemos o número de partículas teste por nucleon N fixo e medimos as grandezas que caracterizam o desempenho para diferentes valores de A. Usamos os seguintes valores de A: 35, 70, 100, 160, 208 e 230, correspondendo respectivamente aos núcleos: Cl, Ge, Sn, Gd, Pb e Th. O número de repetições Monte Carlo foi mantido fixo em $N = 10$. Este valor é muito baixo para obter estatísiticas razoáveis do sistema físico, mas foi o maior valor que conseguimos simular com a memória de 16Mb por processador de que dispomos. Veremos que a memória necessária para armazenar estados limita o tamanho máximo do sistema que conseguimos simular. Todos os testes foram executados nos simuladores seqüencial (rodando em um processador da máquina) e distribuído.

As simulações distribuídas foram executadas em 2, 4 e 8 processadores de um hipercubo iPSC/860 da Intel (de dimensão 1, 2 e 3 respectivamente). As posições e momenta de cada partícula teste foram sorteados segundo distribuições uniformes dentro de uma esfera de raio $R = 1.15A^{1/3}$ fm. Este procedimento está descrito na seção 4.1.1 e foi realizado por um dos processadores, que depois enviou as partículas para os *PLs* aos quais elas pertenciam. O espaço físico (a esfera) que contém o núcleo foi dividido em regiões, como mostra a figura 5.1 e cada região foi simulada por um *PL*. O mapeamento de *PLs* aos processadores foi feito de forma que cada processador executasse um *PL* e *PLs* responsáveis por regiões vizinhas foram alocados a processadores vizinhos no hipercubo.

Em nossos testes, armazenamos o estado de cada *PL* após a execução de cada evento, ao término de cada intervalo antes de enviar as variáveis de estado para os vizinhos e também imediatamente após receber as variáveis de estado dos vizinhos (o estado do *PL* ao iniciar o novo intervalo). É importante notar que estes dois últimos tipos de estado se referem ao mesmo instante de tempo virtual e, por isto denotamos o tempo do final do intervalo com um sinal $(-)$, por exemplo (t^-), na apresentação do algoritmo. Este procedimento de armazenar estados é chamado de *checkpointing*. No final da seção 2.2.1, mencionamos que esta tem sido a freqüência de armazenamento que fornece os melhores desempenhos, segundo autores como Jefferson e Fujimoto [Fu90].

Os estados dos *PLs* são bastante grandes nesta aplicação e demandam muita memória. O estado de um *PL* é constituído da identificação, posição e momento de cada partícula pertencente ao *PL*, além da lista de eventos

locais de colisão e passagem entre fronteiras previstos pelo *PL*. Os estados referentes ao final dos intervalos (marcados com um "superscrito" negativo) ainda devem conter uma matriz que armazena a distribuição espacial da densidade na região simulada pelo *PL*. A memória gasta com o armazenamento de estados é o que determina o tamanho máximo do sistema físico para um dado tamanho de memória dos processadores e por isto usamos um valor de $N = 10$ e maior valor de $A = 230$. O armazenamento de estados após cada evento também implica em um *overhead* de tempo relativamente grande necessário para fazer essas cópias.

Usamos o Teorema 4 da seção 4.4.3 para estimar o limite inferior de TVG e usamos este valor para retomar memória. Até 8 processadores, temos um grafo totalmente conectado com diâmetro $d = 1$, de forma que foi necessário armazenar estados pertencentes ao intervalo presente e anterior do *PL*. É possível acoplar ao simulador um mecanismo distribuído para calcular TVG que tente determinar valores de TVG mais próximos do valor do relógio do *PL*, permitindo a liberação de mais estados passados e portanto o armazenamento de estados maiores (isto permitiria a simulação de sistemas maiores). Este mecanismo acrescentaria um *overhead* de tempo de processamento para a determinação distribuída de TVG.

Devemos também mencionar que as medidas de tempo de execução, tanto do simulador seqüencial quanto do simulador distribuído, se referem apenas ao tempo de processamento da simulação. Tempos gastos com leitura e escrita de dados em disco (feitas apenas uma vez no início e final) e com a inicialização seqüencial do sistema não foram considerados. Isto se justifica pelo fato de que estamos interessados apenas na avaliação do mecanismo de

sincronização do algoritmo simulador. Passamos agora à apresentação dos gráficos de desempenho que obtivemos.

Nossa análise tem uma estrutura muito parecida com aquela apresentada em [Mo94] por Moreano, na avaliação do algoritmo de Simulação do Espaço-Tempo. Como a classe de aplicações que podem ser tratadas por aquele algoritmo (simulações dirigidas por eventos) podem também ser tratadas pelo algoritmo de Intervalos Temporais Revogáveis e a autora tratou uma aplicação (um sistema de discos colidindo elasticamente em uma superfície bidimensional) que apresenta uma estrutura muito parecida com a estrutura da nossa, é interessante fazer algumas comparações.

Na seção 5.1, mostramos analiticamente que o desempenho da simulação seqüencial também melhora com a divisão do espaço físico em regiões. Para mostrar isto experimentalmente, apresentamos medidas da variação do tempo de execução da simulação seqüencial conforme aumentamos o número de divisões em regiões para os diversos valores de A, na figura 5.2. Para os sistemas menores ($A = 35$ e 70), o ganho obtido com a divisão em regiões e conseqüente avaliação de possíveis colisões apenas em uma vizinhança de cada partícula, não compensa o *overhead* introduzido para calcular e processar as passagens dos nucleons de uma região para outra. Nestes casos, a divisão em regiões piora os tempos de execução.

Conforme o tamanho do sistema aumenta, vemos que a curva de tempo de execução cai (o primeiro termo da expressão 5.4 indica um comportamento proporcional a $1/r$), passa por um mínimo e que esse mínimo se desloca lentamente para a direita no gráfico, ou seja, para um número de regiões cada vez maior. Para um dado valor de A, conforme o número de regiões aumenta

ainda mais, o termo referente à passagem de partículas entre regiões passa a dominar e o tempo de execução aumenta com o número de regiões. Podemos então deduzir que conforme o tamanho do sistema físico aumenta, passa a ser vantajoso subdividir o processamento entre um número cada vez maior de *PLs*. Isto mostra que esta aplicação apresenta um grau de paralelismo intrínseco escalável.

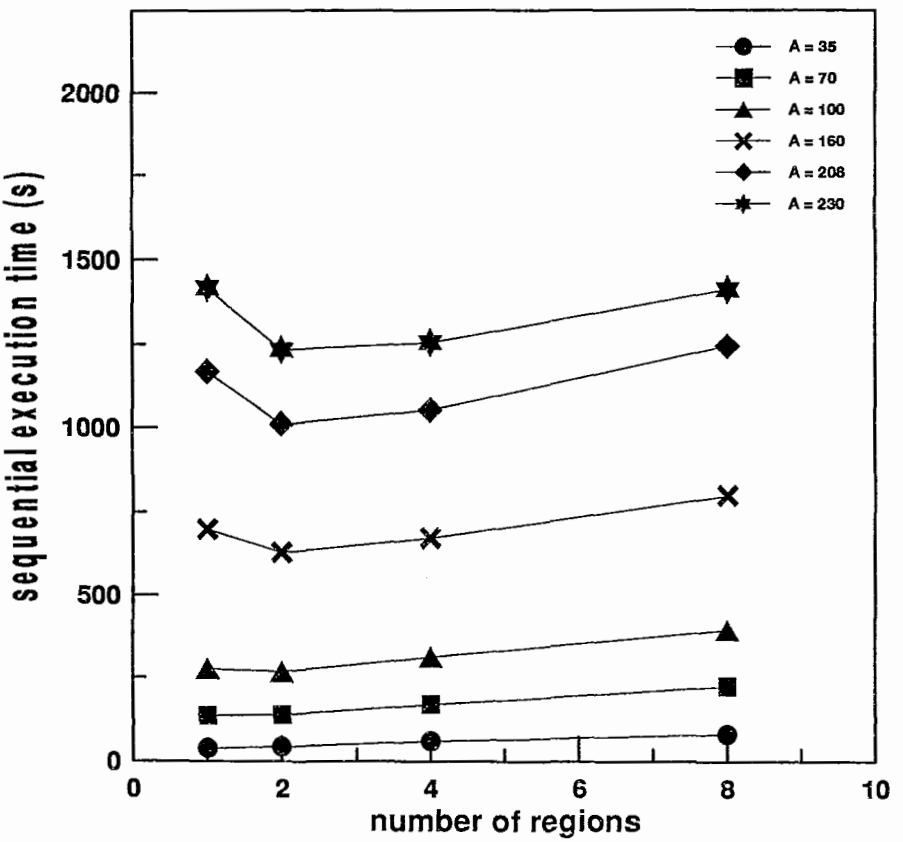


Figura 5.2: Tempo de execução da simulação seqüencial.

Em seguida, na figura 5.3, mostramos como o tempo de execução se comporta quando executamos a simulação de cada região em um processador

diferente. Não foi possível obter o tempo referente à execução da simulação de $A = 160$ com dois processadores nem os tempos para $A = 208$ e $A = 230$ com dois e quatro processadores, porque estas simulações requerem mais memória do que dispomos. Achamos interessante mostrar os pontos referentes a $A = 208$ e $A = 230$, mesmo não sendo possível apresentar as curvas completas, porque estes pontos estão na região em que computação paralela começa a ser interessante (sistemas maiores em mais processadores), onde o sistema físico apresenta maior paralelismo intrínseco.

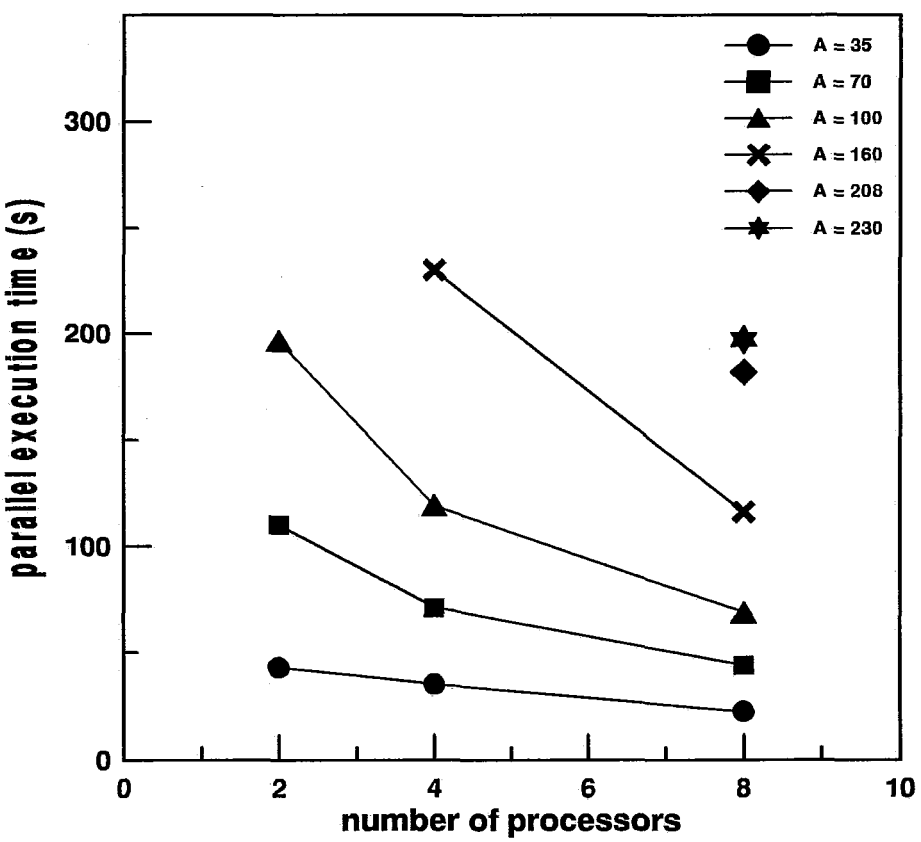


Figura 5.3: Tempo de execução da simulação paralela.

Vemos que para os sistemas menores quase não há ganho ao se dividir o problema entre os diversos processadores. Para um sistema com poucos nucleons, quando dividimos o espaço físico em regiões, aumentamos muito o número de eventos de entrada e saída de partículas de uma região para outra, quando comparado com o número de eventos de colisão. No caso da computação distribuída, com uma região alocada a cada processador, os eventos de entrada e saída de partículas entre as regiões requerem comunicação entre os processadores (envio de mensagens), além do processamento destes eventos. Nos sistemas com menos nucleons divididos em várias regiões, a carga de mensagens na rede fica grande em relação ao processamento interno de cada processador e a computação paralela não é eficiente.

O gráfico da figura 5.3 mostra claramente que, para um tamanho fixo do sistema, o tempo de execução cai inicialmente com a divisão do trabalho entre os processadores, mas esta queda se dá a uma taxa menor conforme o número de processadores (regiões) aumenta. Esta tendência, analisada junto com o gráfico de tempo de execução seqüencial sugere que, para um determinado tamanho de sistema, o desempenho melhora com o aumento do número de divisões em regiões, satura e eventualmente começa a degradar e que o desempenho ótimo se dá para um número de divisões cada vez maior conforme o tamanho do sistema aumenta.

Para mostrar o ganho de velocidade de processamento que se pode ter com o uso de processamento paralelo e distribuído, é útil mostrar medidas de *speedup*. Esta grandeza é definida da seguinte forma:

$$speedup = \frac{tempo\ de\ execução\ do\ melhor\ algoritmo\ seqüencial}{tempo\ de\ execução\ do\ algoritmo\ paralelo}, \qquad (5.6)$$

para uma mesma entrada de dados. Nesta definição, o *melhor algoritmo*

seqüencial não significa apenas o mais rápido. É necessário que os algoritmos seqüencial e paralelo produzam os mesmos resultados (nesta aplicação, isto significa a mesma posição e momento para cada partícula ao final da simulação). Como esta aplicação apresenta uma grande sensibilidade numérica, a inclusão de eventos diferentes em uma das duas ou ambas as simulações pode gerar saídas bastante diferentes (pelo menos do ponto de vista microscópico, o sistema é caótico). Portanto, os tempos de execução que aparecem na equação para o *speedup* devem ser medidos para uma mesma divisão em regiões, de forma que ambos os simuladores executem a mesma seqüência de *eventos corretos* (não considerando aqueles que são desfeitos por *rollback* no simulador distribuído).

Mostramos os *speedups* que obtivemos na figura 5.4. O *speedup* ótimo é obtido quando a paralelização não introduz nenhum *overhead* como comunicação ou sincronização em relação ao algoritmo seqüencial. Para uma computação paralela usando p processadores, o *speedup* ótimo é p. Conforme o tamanho do sistema aumenta, o número de partículas teste por região aumenta, a razão entre a superfície e o volume de cada região diminui e a proporção de eventos de colisão em relação aos eventos de passagens pelas fronteiras aumenta, i.e. a granularidade da computação paralela aumenta e o *speedup* melhora.

O melhor *speedup* que obtivemos foi de aproximadamente 7 para um sistema de $A = 230$ nucleons em oito processadores. Mesmo se comparamos o o tempo de execução em oito processadores deste sistema com a simulação seqüencial de menor tempo (com uma divisão em duas regiões) ainda obtemos um *speedup* superior a 6. Estes valores são muito próximos do ótimo e

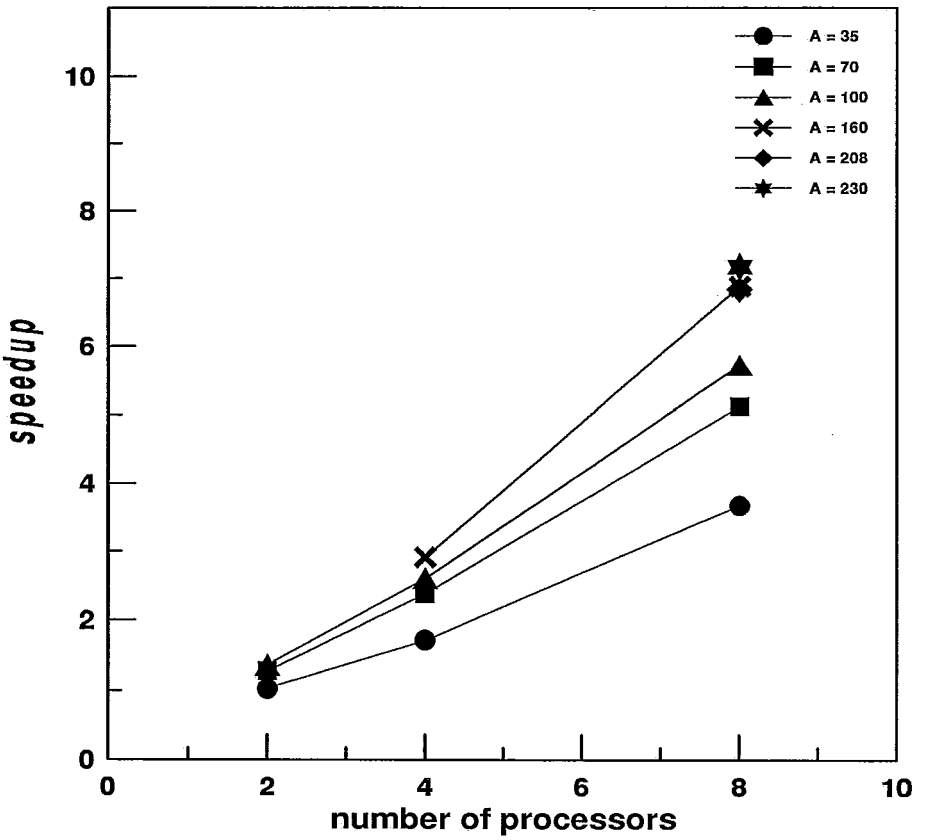


Figura 5.4: *Speedup*

melhores do que os resultados para este tipo de sistema encontrados na literatura em [Be88, Ho89, Je88, Mo94]. Nestes trabalhos, os *speedups* alcançados em geral são menores que 50% do número de processadores usados. Como nos outros trabalhos, nossos resultados indicam que o aumento do tamanho do sistema implicaria em *speedups* ainda melhores.

Podemos apontar alguns fatores que favorecem nossos resultados. Em primeiro lugar, conseguimos realizar testes com sistemas maiores do que os trabalhos citados. No caso de [Mo94], sabemos que isso foi possível porque

dispomos de mais memória por processador. Além disto, em [Mo94], a máquina usada por Moreano não dispunha de um processador de comunicação implementado em *hardware*. O gerenciamento de comunicação foi realizado por um *processador virtual de comunicação*, implementado em *software* que era um processo executado pelo processador local de forma concorrente ao simulador, degradando desempenho. O iPSC/860 tem um processador real de comunicação em *hardware*.

Outro fator é que o tempo gasto para calcular as possíveis colisões e também para processá-las (granularidade do cálculo) em nossa aplicação é maior. Nossa simulação é tridimensional e envolve cálculos de transformadas de Lorentz para mudanças relativísticas entre referenciais que não são necessários nos outros trabalhos. Isto aumenta a importância da divisão do trabalho local a cada processador, ou seja, favorece o argumento de que devemos distribuir a carga computacional, porque aumenta a razão entre o tempo de cálculo e o tempo de comunicação. Neste sentido é interessante ver como o sistema de discos em colisão serve como um modelo físico simplificado que fornece um limite inferior de desempenho para esta classe de aplicações [Be88]. Os sistemas reais, mais complexos, se beneficiam ainda mais do processamento distribuído.

Por último, os trabalhos citados se referem a aplicações dirigidas exclusivamente por eventos, não apresentando as barreiras de sincronização revogáveis e localizadas ao final de intervalos de tempo fixos. Este mecanismo, presente em nosso simulador, impede que os *PLs* se distanciem muito uns dos outros durante a execução reduzindo o número e *tamanho* dos *rollbacks*, como veremos mais adiante. Além disto, com os intervalos temporais

revogáveis, não foi necessário inserir um mecanismo para o cálculo de TVG (devido à propriedade garantida pelo Teorema 4 da seção 4.4.3), para os tamanhos de sistemas que tratamos.

Outros parâmetros que medem a vantagem de se dividir a simulação em regiões são as porcentagens dos eventos totais da simulação que são colisões (de processamento local aos *PLs*) e que implicam em comunicação entre regiões. Estas duas grandezas estão apresentadas nas figuras 5.5 e 5.6. Por eventos de comunicação, denotamos os eventos que modelam a passagem de partículas de uma região para outra que, no caso paralelo, implicam em troca de mensagens entre os processadores. Estas grandezas mostram mais diretamente o grau de paralelismo desta aplicação, ou seja, mostra como que ao dividirmos o problema inserimos comunicação em relação ao processamento interno de cada *PL*.

À medida que o número de regiões aumenta, o número de eventos de comunicação também aumenta. Como os eventos de colisão são processados localmente pelos *PLs*, o ideal é ter a maior proporção de eventos de colisão, sem deixar de explorar o paralelismo oferecido pela aplicação e pela máquina paralela. Já que o número de eventos de colisões permanece constante e a porcentagem de eventos de colisão diminui com o aumento do número de regiões, a forma de minimizar comunicação em relação a computação é escolher um particionamento do problema que favoreça este objetivo. Para este tipo de aplicação, isto significa escolher uma divisão entre regiões que minimize o tamanho da superfície entre regiões em relação ao volume das mesmas. Desta forma, a proporção de partículas dentro das regiões (e portanto de eventos de colisão) em relação ao número de partículas nas regiões

próximas das superfícies (e portanto de eventos de comunicação) é máximo. Por isto escolhemos a divisão apresentada na figura 5.1. É interessante notar que os sistemas com menor número de partículas apresentam maior razão superfície / volume e, conseqüentemente, maior porcentagem de eventos de comunicação para um dado número de regiões.

Se escolhermos uma configuração com apenas uma região, a comunicação é nula e a proporção de eventos de colisão é máxima. No entanto, não estaremos utilizando todo o paralelismo disponível. Para um determinado tipo de particionamento e tamanho do sistema, é necessário balancear *overhead* de comunicação e paralelismo através da escolha de um número de regiões que proporcione um bom desempenho.

É importante também notar que, mesmo que a proporção de eventos de colisão seja baixa para divisões em 4 e 8 *PLs*, para os sistemas maiores o *speedup* é bom. Isto é conseqüência do fato, já mencionado, de que medimos o *speedup* da computação paralela em relação à simulação seqüencial com a mesma divisão em regiões. As simulações com grande proporção de eventos de colisão em relação à comunicação são as que fornecem os mínimos dos gráficos da figura 5.2. Nestes sistemas maiores, há paralelismo a ser explorado através da divisão em mais regiões, mesmo com o *overhead* mais alto de calcular passagens entre mais regiões.

O algoritmo de intervalos temporais revogáveis é um algoritmo otimista. Cada *PL* supõe que toda a informação da qual ele dispõe em um determinado instante de tempo real está correta e prossegue gerando e executando eventos com base nesta informação. Quando um erro de causalidade é detectado pela chegada de uma mensagem com *timestamp* no passado do relógio local do

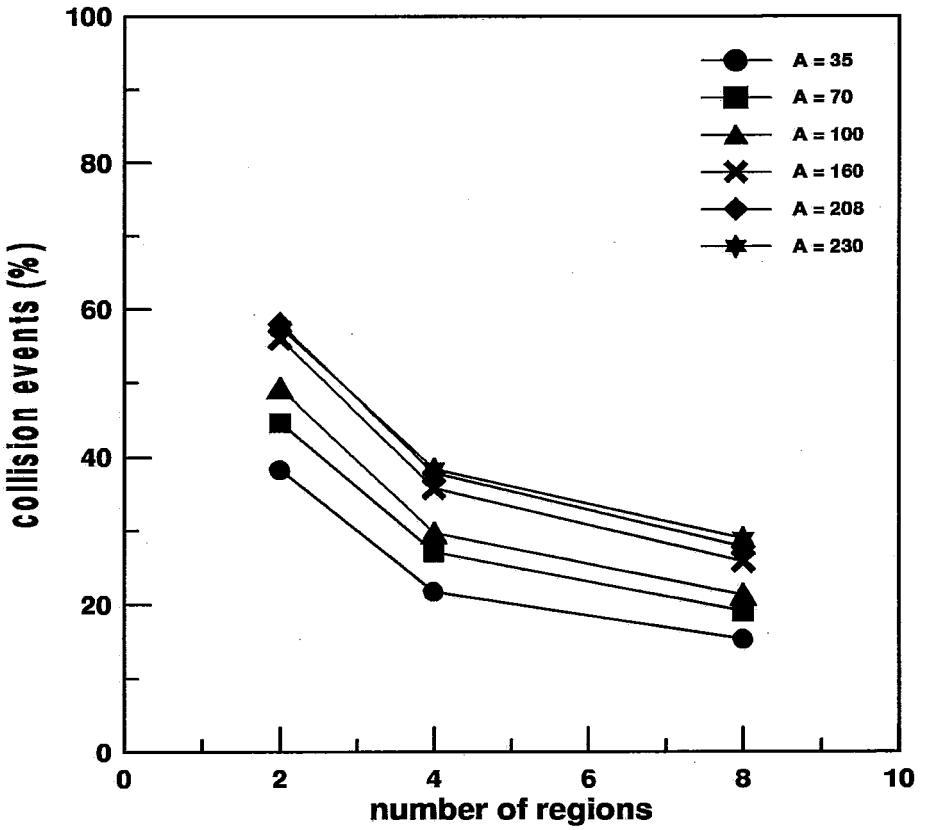


Figura 5.5: % de eventos de colisão

PL, o simulador realiza um *rollback* de estado, desfazendo o efeito de eventos errados já processados e cancelando outros que foram gerados porém ainda não executados. Portanto, o número total de eventos gerados e executados pelos *PL*s durante a simulação distribuída otimista é maior do que o número total de eventos corretos da aplicação. Como alguns eventos gerados (errados) são cancelados antes mesmo de serem executados, o número de eventos gerados é maior que o número de eventos executados. Uma comparação de medidas do número de eventos gerados, executados e corretos da simulação

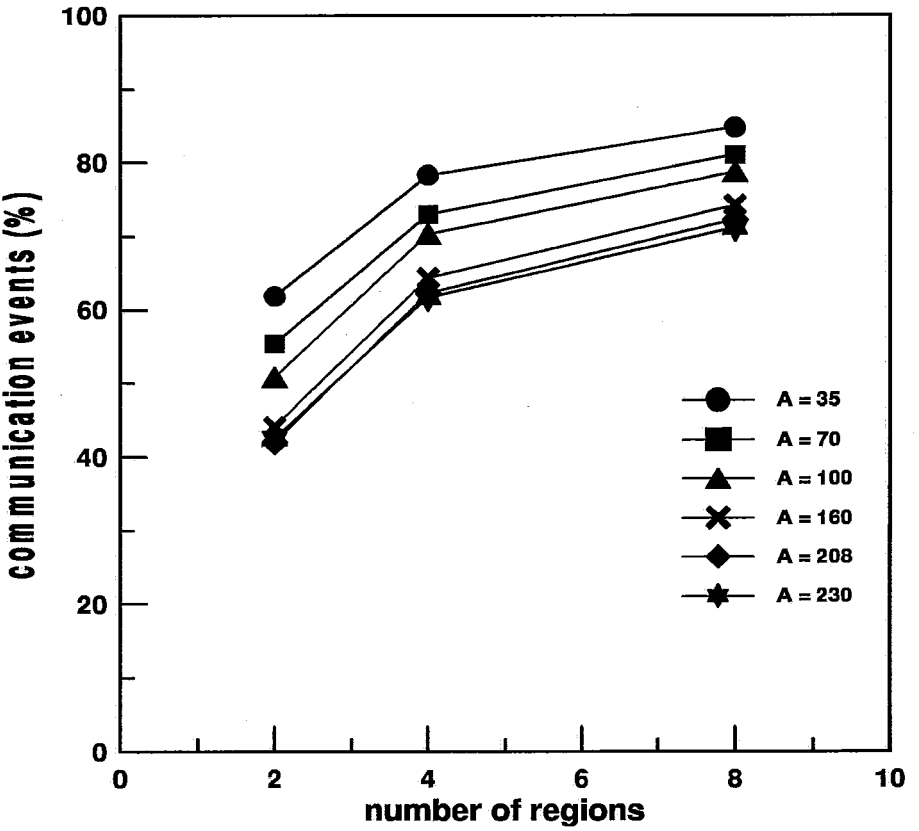


Figura 5.6: % de eventos de comunicação

otimista reflete o quão otimista o simulador distribuído é.

Apresentamos medidas destas grandezas nos gráficos das figuras 5.7, 5.8 e 5.9. As três grandezas crescem com o aumento de A, pois vimos na seção 5.1 que o número de colisões é proporcional a A^2. Elas também devem crescer com o aumento do número de processadores porque, ao aumentar o número de divisões em regiões, o número de eventos de passagens entre fronteiras aumenta. Vemos que o número de eventos gerados e executados é bem maior que o número de eventos corretos.

É interessante notar que em métodos conservadores todos os eventos gerados e executados são corretos. Podería-se argumentar que, como em métodos otimistas o número de eventos executados é muito maior que o número de eventos corretos, a maior parte do tempo de processamento da simulação distribuída foi gasto com trabalho inútil diferentemente das simulações conservadoras. No entanto, Fujimoto argumenta em [Fu90] que o tempo gasto gerando e executando eventos incorretos na simulação distribuída é o mesmo que seria gasto com bloqueios em simulações conservadoras. As medidas de *speedup* sugerem que esta computação incorreta não compromete severamente o desempenho do simulador distribuído.

Duas grandezas importantes na avaliação de algoritmos para simulação paralela e distribuída otimistas são o número de *rollbacks* que foram executados e o *tamanho médio dos rollbacks*. O número de vezes que o procedimento Faça_um_Rollback_Para_$(T, i, j, Inicie_Intervalo)$ da figura 4.9 foi executado corresponde ao número de *rollbacks*. Estes *rollbacks* são disparados pela recepção de mensagens do tipo EVENTO_(j, i) (figura 4.4), SEGURO_$(j, i, t, \rho_j(t))$ (figura 4.6) e NOTIFIQUE_ROLLBACK_(j, i, T) (figura 4.7) que chegam no passado do relógio local de PL_i. O número de *rollbacks*, assim como a comparação entre o número de eventos gerados, executados e corretos, também indica o quão otimista foi a simulação. O gráfico de medidas do número de *rollbacks* está apresentado na figura 5.10 para os diversos valores de A.

É de se esperar que ocorram mais *rollbacks* conforme o número de partículas aumenta, já que há mais eventos sendo processados e, portanto, maior probabilidade de chegar informação no passado do relógio local de um *PL*.

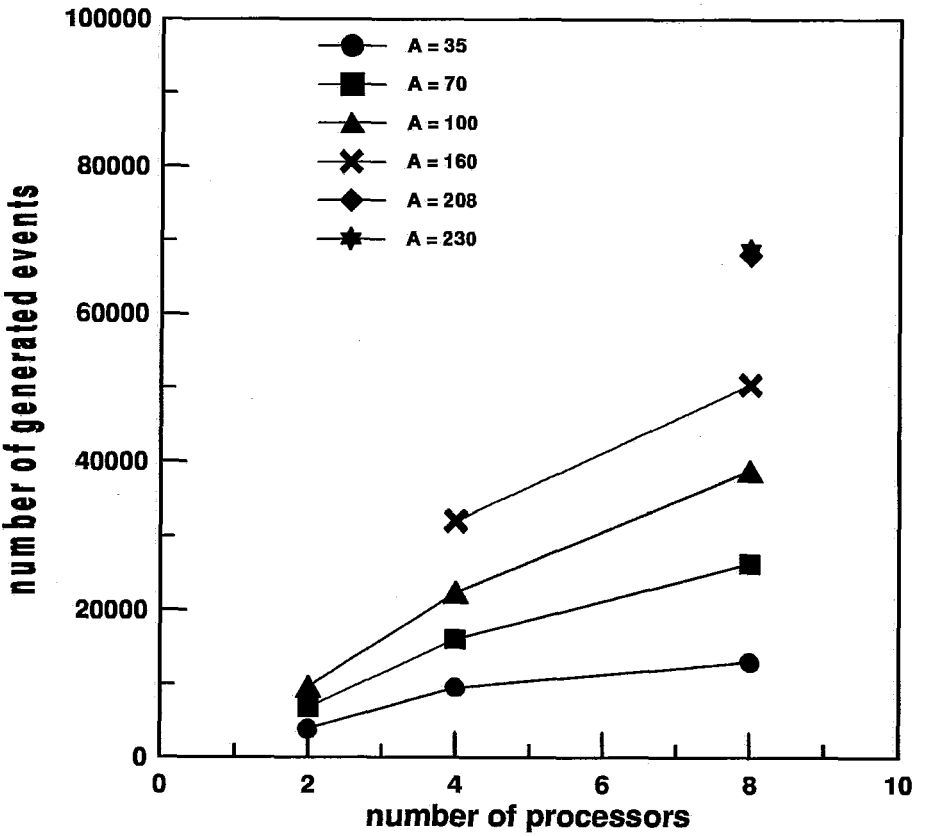


Figura 5.7: Número de eventos gerados na simulação paralela.

O aumento com o número de divisões em regiões (processadores) também se justifica por haver mais *PL*s sendo sincronizados (sofrendo *rollbacks*).

Chamamos o número de eventos já processados por um *PL* que são desfeitos durante um *rollback* de *tamanho do rollback*. Esta grandeza indica o quanto a simulação incorreta desfeita pelo *rollback* em um *PL* havia avançado. Como vemos na figura 5.11, no algoritmo de intervalos temporais revogáveis, o tamanho médio dos *rollbacks* é praticamente constante e pequeno (varia entre os valores 2 e 3). Esta característica é muito interessante e desejável,

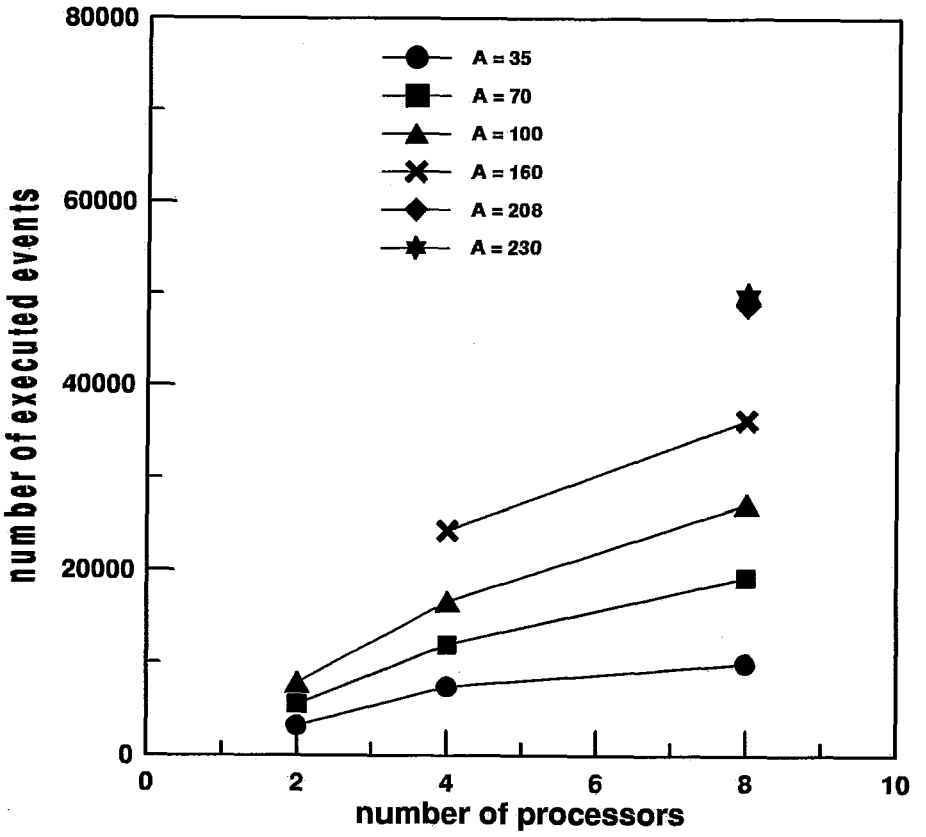


Figura 5.8: Número de eventos executados na simulação paralela.

pois reflete o fato de que as "barreiras" de sincronização revogáveis ao final de cada intervalo impedem que os *PLs* sofram *rollbacks* para um passado muito longínquo, mantendo os relógios dos *PLs* mais próximos entre si e reduzindo a propagação de *rollbacks*. Além disto, o fato desta grandeza se manter constante em um valor baixo indica que o algoritmo é escalável em relação a ela. Em nossos testes com até 8 processadores, o diâmetro d do grafo de *PLs* que representam o sistema físico (veja a Definição 3 na seção 4.4.3) não aumenta com o aumento do número de divisões em regiões. Na

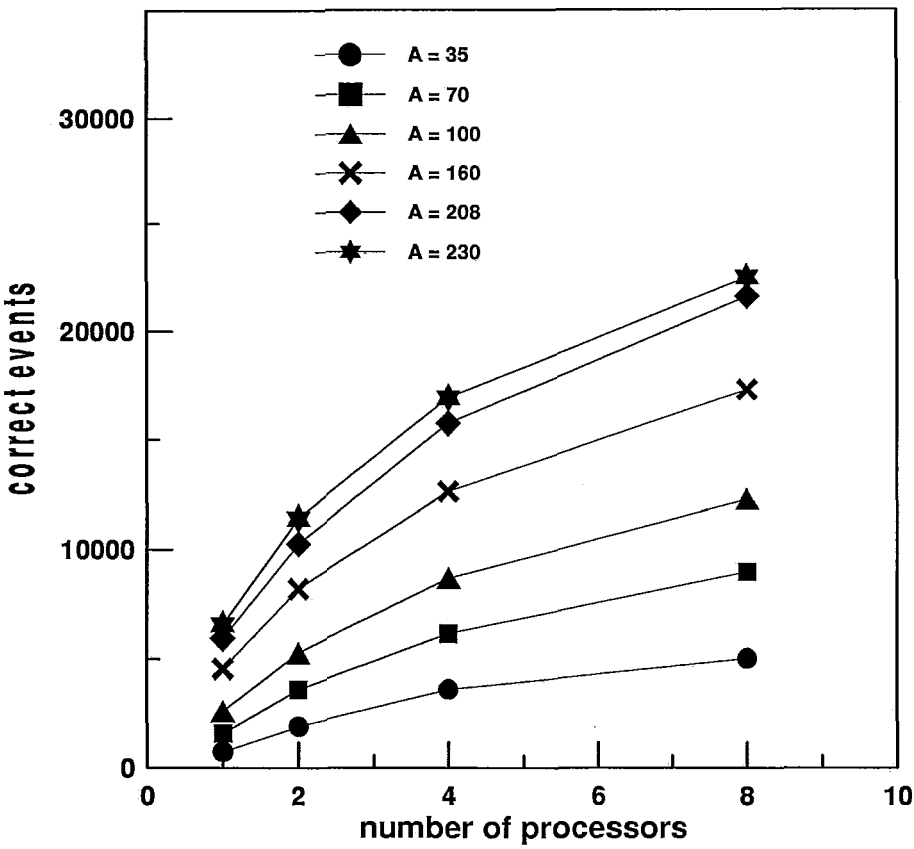


Figura 5.9: Número de eventos corretos.

verdade, segundo o Teorema 4 da seção 4.4.3, o tamanho médio dos *rollbacks* deve aumentar com o aumento do diâmetro d, mas será sempre limitado por este.

Quando um PL_j sofre um *rollback*, ele envia para cada um dos seus vizinhos PL_i mensagens NOTIFIQUE_ROLLBACK_(i, j, t) (veja as figuras 4.7 e 4.9). Portanto, o aumento do número de *rollbacks* com o número de regiões tem como conseqüência o aumento do tráfego absoluto de mensagens na rede. No entanto, um particionamento de carga que distribui bem os *PLs*

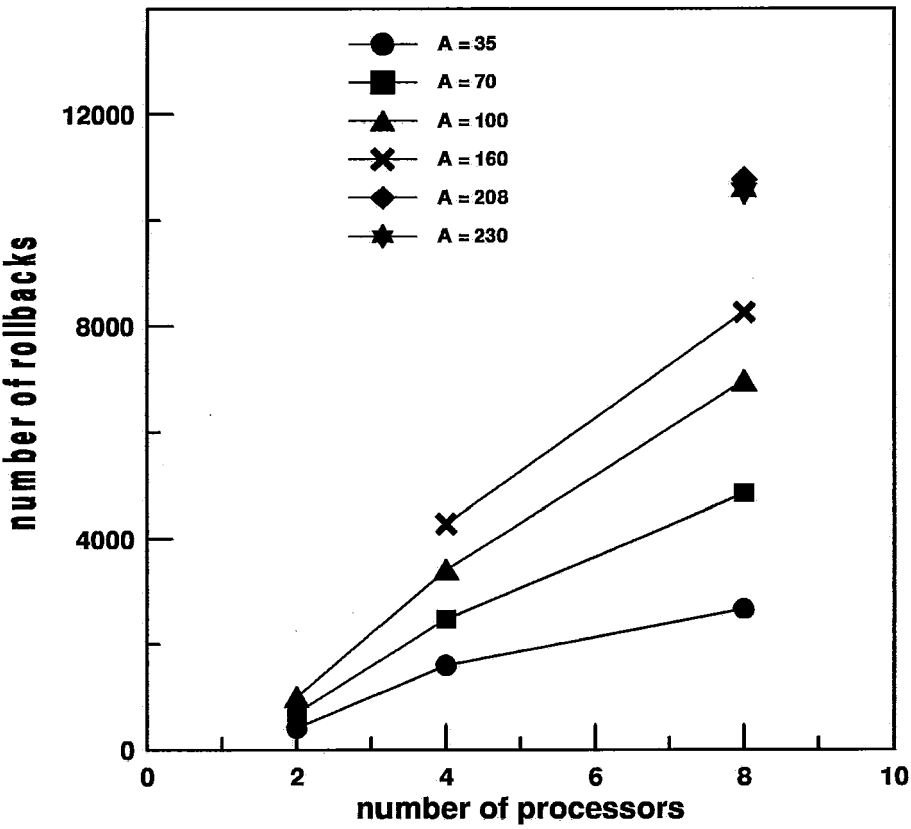


Figura 5.10: Número de *rollbacks*.

e mantém localidade, de forma que vizinhos no grafo de *PLs* são alocados a processadores vizinhos, distribui este aumento de tráfego de mensagens uniformente entre os processadores e o mantém localizado, aumentando o tráfego total mas não o tráfego por canal de comunicação. O volume total de comunicação da simulação distribuída é dado pelas mensagens referentes a eventos de passagem de partículas entre regiões, notificações de SEGURO e notificações de *rollbacks*. Quanto mais bem feito o particionamento do problema, menor será a carga de mensagens por canal de comunicação físico,

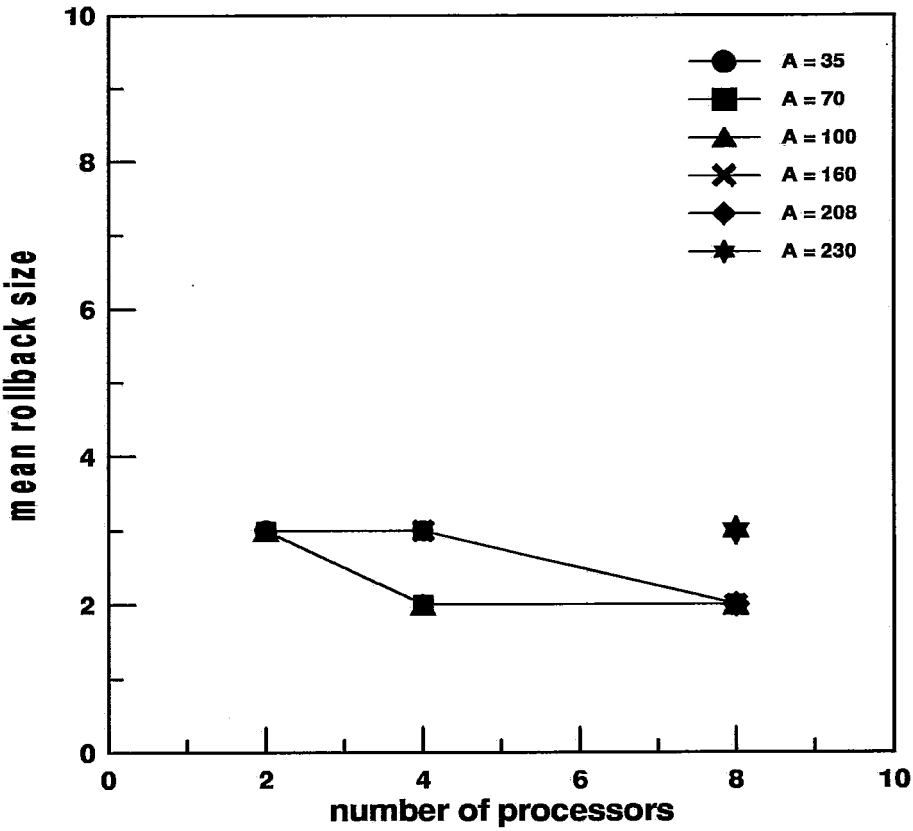


Figura 5.11: Tamanho médio dos *rollbacks*.

já que o algoritmo tem como princípio explorar a localidade da computação e comunicação ao máximo. Cada *PL* interage apenas com seus vizinhos na rede. Em métodos conservadores, há um *overhead* grande causado pelo tráfego de mensagens de controle para a determinação de quais eventos são seguros para serem executados.

É interessante observarmos o comportamento do número de *rollbacks* com o aumento do número de partículas. Quando dividimos cada valor mapeado no gráfico 5.10 pelo número de nucleons A respectivo (não dividimos pelo

número de partículas teste por nucleon N) obtemos o gráfico da figura 5.12. Este gráfico mostra que o número de *rollbacks* cresce a uma taxa ligeiramente menor conforme a valor de A aumenta. Isto significa que o *overhead* de sincronização adicionado à simulação diminui conforme o número de partículas aumenta, ou seja, o *overhead* aumenta com o aumento do tamanho do sistema mas a uma taxa menor. Se consideramos o número de *rollbacks* por partícula teste (é só dividir a escala do eixo das ordeanadas por $N = 10$), obtemos medidas de número de *rollbacks* por partícula ainda menores que aquelas obtidas por Moreano em [Mo94].

Devido à quantidade de memória de que dispomos não foi possível obter medidas para núcleos com mais de 230 nucleons e mais que 10 partículas teste por nucleon. Para uma simulação em 8 processadores, isto equivale a aproximadamente $2300/8 \approx 288$ partículas teste por processador. Nossos resultados reforçam as conclusões de outros autores que trataram aplicações semelhantes [Be88, Ho89, Je88, Mo94], de que o simulador distribuído otimista apresenta melhor desempenho conforme o tamanho do sistema simulado (tamanho da entrada da simulação) aumenta. No entanto, para sistemas maiores, o *overhead* associado ao armazenamento de estados pode introduzir uma degradação de desempenho. Após a execução de cada evento é feita uma cópia de todo o estado da aplicação e a cada *rollback* um estado é restaurado. Em [Fu88a], Fujimoto, Tsai e Gopalakrishnan propõem o uso de *hardware* dedicado para realizar o armazenamento de estados e diminuir este *overhead*.

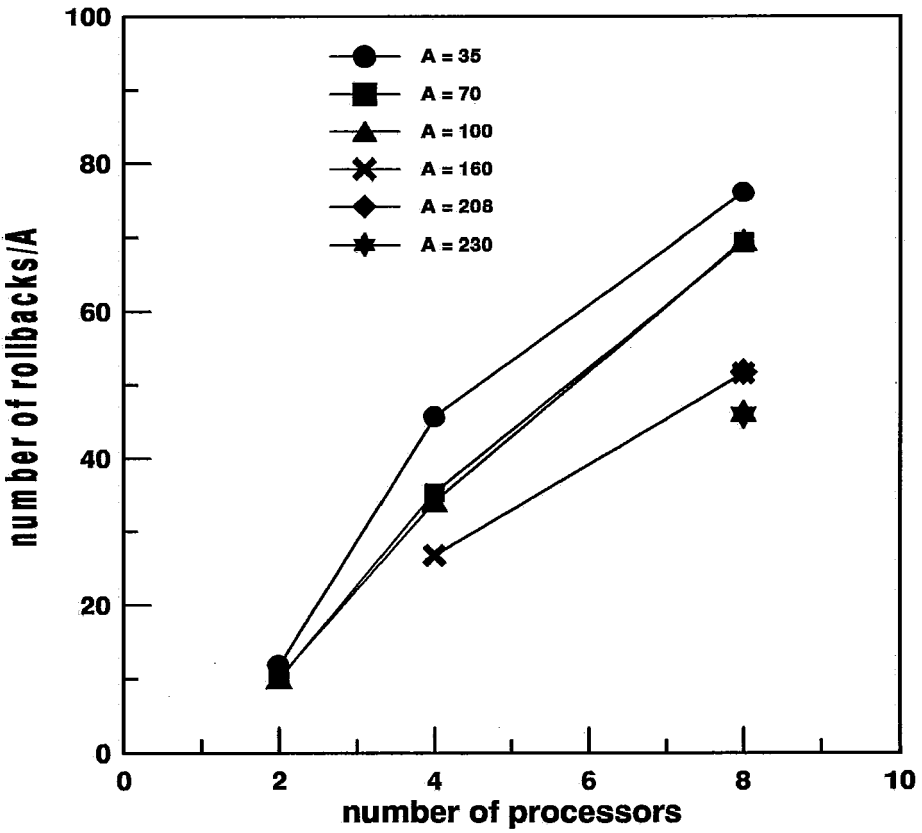


Figura 5.12: Número de *rollbacks* por nucleon.

Capítulo 6

Conclusões

Neste trabalho, propomos um algoritmo para a simulação de sistemas que evoluem no tempo dirigidos por tempo e por eventos, uma forma de evolução temporal que chamamos de híbrida. Até o presente momento, ainda não foi proposto nenhum outro algoritmo para a simulação de sistemas que apresentam este tipo de comportamento. Detectamos a necessidade de desenvolver este algoritmo a partir de um estudo do comportamento de problemas científicos que potencialmente se utilizam de computação de alto desempenho para serem resolvidos. Vimos que este tipo de estudo com o desenvolvimento de algoritmos, linguagens e ambientes de programação que permitam o uso eficiente das máquinas paralelas, com um esforço mínimo para o usuário, formam um dos principais desafios atuais para a difusão destas máquinas como computadores de propósito geral.

Nosso principal objetivo neste trabalho foi apresentar o algoritmo de intervalos temporais revogáveis, assim como os resultados da implementação e testes do método em uma aplicação de física nuclear.

Inicialmente, apresentamos os conceitos básicos e as dificuldades da área de simulação distribuída. Mostramos os principais algoritmos propostos na literatura para simulação paralela e distribuída dirigidas por tempo e por eventos (estes podem ser conservadores ou otimistas). Descrevemos as principais características destes algoritmos, as condições nas quais cada um apresenta melhor desempenho e resultados de experiências de implementações já realizadas.

Em seguida, no capítulo 4, especificamos as classes de problemas de simulação que podem ser simuladas pelo algoritmo de intervalos temporais revogáveis. Descrevemos o sistema físico que nos motivou inicialmente: uma colisão nuclear modelada matematicamente por uma equação de Boltzmann-Uehling-Uhlenbeck para a função distribuição de partícula única, resolvida pelo método de partículas teste. Argumentamos que esta aplicação é especialmente adequada para simulação paralela porque, além de apresentar um alto grau de paralelismo, apresenta uma granularidade de computação grande em relação à comunicação (para um bom particionamento do problema) e, embora seja uma modelagem estocástica, as repetições Monte Carlo do problema são acopladas entre si, de forma que a melhor solução paralela é dividir o sistema físico espacialmente em *PL*s e executar cada *PL* em um processador diferente. Neste sentido, esta aplicação é muito interessante porque a maior parte dos problemas usados para testar algoritmos de simulação paralela e distribuída ou são modelos idealizados, muitas vezes com cargas computacionais sintéticas, ou sistemas estocásticos onde a melhor forma de paralelizar seria rodar cada simulação independente do problema em um processador diferente. Nossos resultados de desempenho se referem a uma aplicação real com a melhor solução possível de paralelização.

Ainda no capítulo 4, fizemos a apresentação formal do algoritmo e em seguida, da prova de corretude onde aparecem as propriedades básicas do método. Mostramos que, embora o algoritmo não tenha sido desenvolvido originalmente com este objetivo, ele pode ser usado muito bem como uma forma de limitar o otimismo de simulações dirigidas exclusivamente por eventos otimistas. Em seguida, fizemos um levantamento dos trabalhos de simulação paralela que tentam limitar otimismo mais relacionados com o nosso. Argumentamos que o algoritmo de intervalos temporais revogáveis tem a vantagem de limitar o otimismo sem criar barreiras de sincronização centralizadas. O algoritmo decide que um *PL* pode avançar para o próximo intervalo baseado apenas em informação própria e dos vizinhos mais próximos. Se esta decisão se mostrar errada no futuro através da recepção de mensagens STRAGGLERS, ela é revogada e o *PL* sofre um *rollback* voltando para um intervalo anterior. Ou seja, o avanço de uma janela (intervalo) para outra é feito de forma otimista e distribuída.

Finalmente, apresentamos uma avaliação experimental do desempenho do simulador executando a simulação do sistema nuclear que descrevemos. Esta análise permite induzir o comportamento genérico do algoritmo. Obtivemos resultados de *speedup* muito bons, em geral melhores que os já apresentados na literatura para simulações dirigidas por eventos de sistemas de discos colidindo em uma superfície bi-dimensional. Vimos que conforme o tamanho do sistema simulado aumenta, o desempenho melhora para um particionamento cada vez maior do problema. Este é o comportamento esperado para problemas com bom paralelismo intrínseco, escaláveis. Os algoritmos otimistas apresentam bom desempenho porque utilizam ao máximo o tempo de processamento dos processadores. Os processadores nunca ficam ociosos como

nos algoritmos conservadores, pois se permite que eles trabalhem, mesmo que não tenham certeza de que têm toda a informcação necessária, executando computação possivelmente correta.

Para tamanhos de sistema muito pequenos, o *overhead* causado pelas mensagens de sincronização e *rollbacks* se torna grande quando comparado com o volume de processamento interno aos processadores e a simulação paralela não é eficiente. Nestes casos, o simulador passa grande parte do seu tempo cancelando computações prematuras e reprocessando-as.

As experiências mostram que as principais fontes de *overhead* em simulações distribuídas que utilizam nosso algoritmo são: o tempo gasto processando *rollbacks*, as mensagens de notificação de *rollbacks* que os propagam pela rede, as mensagens de sincronização ao final de cada intervalo (mensagens SEGURO), o tempo e memória gastos com a gravação e recuperação de estados e o aumento de comunicação de eventos físicos inserida pelo particionamento cada vez maior do problema.

Para minimizar estes *overheads*, é importante manter a carga de processamento interno aos processadores grande e realizar particionamentos e mapeamentos do sistema físico que forneçam bom balanceamento de carga e diminuam a necessidade de roteamento e retransmissão de mensagens. Também existem propostas de utilização de *hardware* dedicado para realizar o armazenamento e recuperação de estados.

Uma propriedade importante do método de intervalos temporais revogáveis é que ele mantém os valores dos relógios dos *PLs* mais próximos uns dos outros. Isto permite que a freqüência de determinação de TVG seja menor,

às vezes até mesmo dispensando seu cálculo e também reduz o tamanho médio dos *rollbacks*, impedindo que os *PLs* fiquem voltando para um passado longínquo.

6.1 Trabalhos Futuros

Ainda é necessário continuar investigando várias características do algoritmo que propomos para poder validá-lo como um *método de simulação distribuída de propósito geral.* É necessário ajustar o tamanho ideal do intervalo Δt, em cada aplicação para usar o método como uma forma de limitar otimismo em simulações distribuídas dirigidas por eventos (SDDE). Como determinar o valor de Δt que mantém os relógios dos *PLs* mais próximos, impedindo *rollbacks* excessivos e para um passado longínquo sem, no entanto, impedir que eles avancem no tempo usando ao máximo o paralelismo inerente à aplicação? Em nosso trabalho, o valor de Δt foi fixado pela precisão numérica necessária para integrar as equações integro-diferenciais de BUU.

É também muito importante comparar o algoritmo de intervalos temporais revogáveis com os outros métodos propostos para restringir o otimismo de simulações distribuídas, que mencionamos na seção 4.5. Uma comparação com métodos otimistas puros, como *Time Warp* e Espaço-Tempo, é fundamental para verificar o quanto SDDEs se beneficiam de uma limitação do otimismo. Para isto, é necessário acoplarmos um algoritmo para determinação de TVG à nossa implementação.

No que diz respeito à modelagem de uma colisão nuclear que nos inspirou inicialmente, precisamos acoplar ao método de simulação que propomos um

algoritmo para balanceamento dinâmico de carga durante a simulação. Esta foi a característica do problema que nos interessou inicialmente e este tipo de mecanismo é necessário, para que qualquer algoritmo de simulação distribuída possa realmente ser usado como mecanismo de propósito geral. O desenvolvimento de algoritmos eficientes para compartilhamento dinâmico de carga em computação paralela e distribuída é um problema de difícil solução e que ainda requer propostas viáveis [Fo88, Ni92, Re90, We91a]. Acreditamos que podemos usar a propriedade de evolução temporal contínua e em parte previsível, da carga computacional em sistemas como a colisão nuclear, para fazer um balanceamento dinâmico durante a simulação.

Uma vez que tivermos um ambiente para simulação distribuída com evolução temporal híbrida e balanceamento dinâmico de carga, poderemos realizar simulações de colisões nucleares, obtendo resultados interessantes para a física destas colisões em energias intermediárias. Nesse ponto, teremos também uma aplicação maior, com mais *PLs* executando em mais processadores, podendo validar ainda mais os resultados de desempenho obtidos neste trabalho.

Finalmente, é importante testar o método na simulação de outros sistemas físicos que apresentam a propriedade de evolução temporal híbrida ou que tenham evolução temporal dirigida exclusivamente por eventos. Sistemas reais de grandes dimensões, como a solução da equação de transporte de neutrinos em uma explosão de super nova (também modelada por uma equação de BUU e resolvida pelo método de partículas teste) ou sistemas de dinâmica molecular, fornecerão mais informações sobre o desempenho e escalabilidade do método.

Referências

[Ai85] Aichelin, J. and Bertsch, G., *Numerical Simulation of Medium Energy Heavy Ion Reactions*, Physical Review C 31, 5(May 1985), 1730–1738.

[Am88] Amorim, C., Barbosa, V. C. and Fernandes, E. S. T., *Uma Introdução à Computação Paralela e Distribuída*, Unicamp-IMECC, Campinas, S. P., 1988.

[Aw85] Awerbuch, B., *Complexity of Network Synchronization*, J. of the ACM 32, 4(Oct. 1985), 804–823.

[Aw88] Awerbuch, B. and Sipser, M., *Dynamic Networks are as Fast as Static Networks*, Proceedings of the 29^{th} Annual Symposium on Foundations of Computer Science, (Oct. 24–26, 1988), 206–219, White Plains, N. Y.

[Ba90] Ball, D. and Hoyt, S., *The Adaptive Time Warp Concurrency Control Algorithm*, Proceedings of the SCS Multiconference on on Distributed Simulation, Vol. 22(1) (Jan. 1990), 174–177.

[Ba91] Bagrodia, R., Chandy, K. M. and Liao, W. T., *A Unifying Framework for Distributed Simulation*, ACM Transaction on Modeling and Computer Simulations 1, 4(Oct. 1991), 348–385.

[Ba92] Bauer, W., Bertsch, G.F. and Schulz, H., *Bubble and Ring Formation in Nuclear Fragmentation*, Technical Report MSUCL-840, (Jun. 1992), 1–9, Michigan State University - National Superconducting Cyclotron Laboratory.

[Be88] Beckman, B., DiLoreto, M., Sturdevant, K., Hontalas, P., Van Warren, L., Blume, L., Jefferson, D. R. and Bellenot, S., *Distributed Simulation and Time Warp Part 1: Design of Colliding Pucks*, Proceedings of the SCS Multiconference on Distributed Simulation 19, 3(Jul. 1988), 56–60.

[Be88a] Bertsch, G.F. and Gupta, S.D., *A Guide to Microscopic Models For Intermediate Energy Heavy Ion Collisions*, Physics Reports 160, 4(Mar. 1988), 190–233, North Holland.

[Be92] Bell, G., *Ultracomputers: A Teraflop Before Its Time*, Comm. of the ACM 35, 8(Aug. 1992), 27–47.

[Br77] Bryant, R. E., *Simulation of Packet Communication Architecture Computer Systems*, MIT/ LCS/ TR-188, Massachusetts Institute of Technology, Cambridge, Massachusetts, (Nov. 1977).

[Br87] Briscoe, D.P., Sokol, L.M. and Wieland, A.P., *Object-Oriented Simulations on a Parallel Arqchitecture*, Technical Report MTR-87W00172, (Sept. 1987), 1–27, The MITRE Corporation.

[Ch79] Chandy, K. M., Misra, J. and Holmes, V., *Distributed Simulation of Networks*, Computer Networks 3, (1979), 105–113.

[Ch79a] Chandy, K. M. and Misra, J., *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs*, IEEE Trans. on Software Eng., SE-5, 5(Sept. 1979), 440–452.

[Ch81] Chandy, K. M. and Misra, J., *Asynchronous Distributed Simulation via a Sequence of Parallel Computations*, Commun. of the ACM 24, 4(Apr. 1981), 198–206.

[Ch85] Chandy, K. M. and Lamport, L., *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Trans. on Computer Systems 3, 1(Feb. 1985), 63–75.

[Ch87] Chandy, K. M. and Misra, J., *Conditional Knowledge as a Basis for Distributed Simulation*, Technical Report 5251:TR:87, Computer Science Department, California Institute of Technology (1987).

[Ch89] Chandy, K. M. and Sherman, R., *The Conditional Event Approach to Distributed Simulation*, Proceedings of the SCS Multiconference on Distributed Simulation 21, 2(Mar. 1989), 93–99.

[Ch89a] Chandy, K. M. and Sherman, R., *Space-Time and Simulation*, Proceedings of the SCS Multiconference on Distributed Simulation 21, 2(Mar. 1989), 53–57.

[Di80] Dijkstra, E. W. and Scholten, C. S. *Termination Detection for Diffusing Computations*, Inf. Proc. Letters 11, 1(Aug. 1980), 1–4.

[Do92] Donangelo, R. and Wedemann, R. S., *Thermal Equilibration in Fragmentation Induced by Proton-Nucleus Collisions*, Nuclear Physics A541 (1992), 641–650.

[Du87] Duda A., Harus, G., Haddad, Y. and Bernard, G., *Estimating Global Time in Distributed Systems*, Proceedings of the 7^{th} International Conference on Distributed Computing Systems, (Sep. 1987), 299–306.

[Fo88] Fox, G. C. and Furmanski, W., *Load Balancing Loosely Synchronous Problems with a Neural Network*, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Vol. I, (Jan. 19–20, 1988), 241–278, ACM Press.

[Fu88] Fujimoto, R. M., *Lookahead in Parallel Discrete Event Simulation*, Proceedings of the 1988 International Conference on Parallel Processing Vol. III, (Aug. 15–19, 1988), 34–41, The Pennsylvania State University Press.

[Fu88a] Fujimoto, R. M., Tsai, J. J. and Gopalakrishnan, G., *Design and Performance of Special Purpose Hardware for Time Warp*, Proceedings of the 15^{th} Annual International Symposium on Computer Architecture (May 30–Jun. 2, 1988), Honolulu, Hawaii, 401–408.

[Fu89] Fujimoto, R. M., *Time Warp on a Shared Memory Multiprocessor*, Proceedings of the 1989 International Conference on Parallel Processing, (Aug 8–12, 1989), III-242–III-249.

[Fu89a] Fujimoto, R. M., *The Virtual Time Machine*, International Symposium on Parallel Algorithms and Architectures, (Jun. 1989), 211–239.

[Fu90] Fujimoto, R. M., *Parallel Discrete Event Simulation*, Commun. of the ACM 33, 10(Oct. 1990), 30–53.

[Ga88] Gafni, A., *Rollback Mechanisms for Optimistic Distributed Simulation Systems*, Proceedings of the SCS Multiconference on Distributed Simulation 19, 3(Jul. 1988), 61–67.

[Gh88] Ghosh, S. and Yu, M., *An Asynchronous Distributed Approach for the Simulation of Behavior-Level Models on Parallel Processors*, Proceedings of the 1988 International Conference on Parallel Processing Vol. I, (Aug. 15–19, 1988), 74–77, The Pennsylvania State University Press.

[Ha88] Hartrum, T. C. and Donlan, B. J., *HYPERSIM: Distributed Discrete-Event Simulation on an iPSC*, 3^{rd} Conference on Hypercube Concurrent Computers and Applications Vol. 1, (Jan. 19–20, 1988), Pasadena Ca., 745–747, ACM Press.

[Ha94] Hamnes, D. O. and Tripathi, A., *Investigations in Adaptive Distributed Simulation*, Proceedings of the Eighth Workshop on Parallel and Distributed Simulation (PADS'94), Edinburgh, Scotland, U.K., (July 6–8, 1994), 20–23, SCS, ACM–SIGSIM, IEEE Computer Society.

[Ho89] Hontalas, P., Beckman, B., DiLoreto, M., Blume, L., Reiher, P., Sturdevant, K., Van Warren, L., Wedel, J., Wieland, F. and Jeffer-

son, D. R., *Performance of the Colliding Pucks Simulation on the Time Warp Operating System*, Proceedings of the SCS Multiconference on Distributed Simulation 21, 2(Mar. 1989), 3–7.

[Je85] Jefferson, D. R., *Virtual Time*, ACM Trans. on Prog. Lang. and Syst. 7, 3(Jul. 1985), 404–425.

[Je87] Jefferson, D. R., Beckman, B., Wieland F., Blume, L., DiLorento, M., Hontalas, P., Reiher, P., Sturdevant, K., Tupman, J., Wedel, J. and Younger, H., *The Time Warp Operating System*, 11^{th} Symposium on Operating Systems Principles 21, 5(Nov. 1987), 77-93.

[Je88] Jefferson, D. R. *et al*, *The Status of the Time Warp Operating System*, 3^{rd} Conference on Hypercube Concurrent Computers and Applications Vol. 1, (Jan. 19–20, 1988), Pasadena Ca., 738–744, ACM Press.

[Jh94] Jha, V. and Bagrodia, R., *A Unified Framework for Conservative and Optimistic Distributed Simulation*, Proceedings of the Eighth Workshop on Parallel and Distributed Simulation (PADS'94), Edinburgh, Scotland, U.K., (July 6–8, 1994), 12–19, SCS, ACM–SIGSIM, IEEE Computer Society.

[Ka87] Kaudel, F. J., *A Literature Survey on Distributed Discrete Event Simulation*, ACM SIMULETTER 18, 2(Jun. 1987), 11–21.

[La83] Lavenberg, S., Muntz, R. and Samadi, B., *Performance Analysis of a Rollback Method for Distributed Simulation*, Performance'83, 117–132, Amsterdam: North Holland.

[Lu87] Lubachevsky, B.D., *Efficient Parallel Simulations of Asynchronous Cellular Arrays*, Complex Systems 1, 6(Dec. 1987), 1099–1123.

[Lu88] Lubachevsky, B.D., *Efficient Parallel Simulations of Dynamic Ising Spin Systems*, J. Comp. Physics 75, 1(Mar. 1988), 103–122.

[Lu89] Lubachevsky, B.D., *Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks*, Commun. of the ACM 32, 1(Jan. 1989), 111–123.

[Ma88] Madisetti, V., Walrand, J., and Messerschmitt, D., *Wolf: A Rollback Algorithm for Optimistic Distributed Simulation Systems*, Proceedings of the 1988 Winter Simulation Conference, San Diego, Ca.

[Ma89] Madisetti, V., Walrand, J., and Messerschmitt, D., *Efficient Distributed Simulation*, Proceedings of the 22^{nd} Annual Simulation Symposium Tampa, Florida, (March 28–31, 1989), 5–21, IEEE Computer Society Press.

[Mi86] Misra, J., *Distributed Discrete-Event Simulation*, Comp. Surveys 18, 1(Mar. 1986), 39–65.

[Mi84] Mitra, D. and Mitrani, I., *Analysis and Optimum Performance of Two Message-Passing Parallel Processors Synchronized by Rollback*, Performance'84, 35–50, New York: Elsevier.

[Mo94] Moreano, N. B., *Um Simulador Distribuído Baseado no Paradigma Espaço-Temporal*, Tese de Mestrado, COPPE - Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ (Feb. 1994).

[Na81] Nance, R. E. and Tech, V., *The Time and State Relationships in Simulation Modeling*, Commun. of the ACM 24, 4(Apr. 1981), 173–179.

[Ni92] Nicol, D. M. and Fujimoto, R., *Parallel Simulation Today*, Technical Report, ICASE–92-62, NASA-CR–18, (1992), 1–34.

[Ni92a] Nicol, D. M., *Optimistic Barrier Synchronization*, Technical Report, NASA, (1992), 1–20.

[Ni93] Nicol, D. M., *The Cost of Conservative Synchronization in Parallel Discrete-Event Simulation*, Journal of the ACM (Apr. 1993).

[Pe79] Peacock, J. K., Wong, J. W. and Manning, E. G., *Distributed Simulation Using a Network of Processors*, Computer Networks 3, (1979), 44–56.

[Pe80] Peacock, J. K., Manning, E. and Wong, J. W., *Synchronization of Distributed Simulation Using Broadcast Algorithms*, Computer Networks 4, (1980), 3–10.

[Pr88] Prakash, A. and Ramamoorthy, C. V., *Hierarchical Distributed Simulations*, Proceedings of the 8^{th} International Conference of Distributed Computing Systems, (Jun. 13–17, 1988), San José, Ca, Computer Society Press.

[Re88] Reed, D. A., Malony, A. D. and McCredie, B. D., *Parallel Discrete Event Simulation Using Shared Memory*, IEEE Trans. on Software Eng. 14, 4(Apr. 1988), 541–553.

[Re90] Reiher, P. L. and Jefferson, D., *Dynamic Load Management in the Time Warp Operating System*, Transactions of the Society for Computer Simulation, 7(2), (Jun. 1990), 91–120.

[Sc92] Schüler, A., *Parallelizing Particle Simulations Based on the Boltzmann Equation*, Parallel Computing 18(1992), 269–279.

[So88] Sokol, L. M., Briscoe, D. P. and Wieland, A. P., *MTW: A Strategy for Scheduling Discrete Event Simulation Events for Concurrent Execution*, Proceedings of the SCS Multiconference on Distributed Simulation 19, 3(Jul. 1988), 34–42.

[So89] Sokol, L. M., Brian, K. S. and Vincent, S. H., *MTW: A Control Mechanism for Parallel Discrete Simulation*, Proceedings of the 1989 International Conference on Parallel Processing 3, (Aug. 1989), III-250–III-254, The Pennsylvania State University Press.

[So90] Sokol, L.M. and Stucky, B.K., *MTW: Experimental Results for a Constrained Optimistic Scheduling Algorithm*, 1990 SCS Conference on Distributed Simulation.

[St91] Steinman, J., *SPEEDES: Synchronous Parallel Environment for Emulation and Discrete-Event Simulation*, Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation, Vol. 23 (Jan. 1991), 95–103, SCS Simulation Series.

[St93] Struckmeier, J. and Pfreundt, F. J., *On the Efficiency of Simulation Methods for the Boltzmann Equation on Parallel Computers*, Parallel Computing 19(1993), 103–119.

[St94] Steinman, J., *Parallel Proximity Detection and the Distribution List Algorithm*, Proceedings of the Eighth Workshop on Parallel and Distributed Simulation (PADS'94), Edinburgh, Scotland, U.K., (July 6–8, 1994), 3–11, SCS, ACM–SIGSIM, IEEE Computer Society.

[St94a] Steinman, J., *Discrete Event Simulation and the Event Horizon*, Proceedings of the Eighth Workshop on Parallel and Distributed Simulation (PADS'94), Edinburgh, Scotland, U.K., (July 6–8, 1994), 39–49, SCS, ACM–SIGSIM, IEEE Computer Society.

[Tu92] Turner, S. J. and Xu, M. Q., *Performance Evaluation of the Bounded Time Warp Algorithm*, Proceedings of the Sixth Workshop on Parallel and Distributed Simulation (PADS'92), 24 (1992), 117–126, SCS Simulation Series.

[We88] West, D., *Optimizing Time Warp: Lazy Rollback and Lazy Reevaluation*, M.S. Thesis, University of California, (Jan. 1989).

[We91] Wedemann, R. S., *Simulação Distribuída*, Relatório Interno, COPPE - Universidade Federal do Rio de Janeiro, (May 1991), Rio de Janeiro, RJ, Brasil.

[We91a] Wedemann, R. S., *Alocação Dinâmica de Tarefas em Sistemas Distribuídos*, Relatório Interno, COPPE - Universidade Federal do Rio de Janeiro, (Dec. 1991), Rio de Janeiro, RJ, Brasil.

[We92] Wedemann, R. S., *Simulação Distribuída com Carga Computacional Dinâmica e não Homogênea*, Relatório Interno, COPPE - Universidade Federal do Rio de Janeiro, (May 1992), Rio de Janeiro, RJ, Brasil.

[Wi89] Wieland, F. and Jefferson, D., *Case Studies in Serial and Parallel Simulation*, Proceedings of the 1989 International Conference on Parallel Processing, (Aug. 8–12, 1989), III-255–III-258. The Pennsylvania State University Press.