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

Simulação paralela de eventos discretos com uso de memória compartilhada distribuída

por

MARCELO TRINDADE REBONATTO

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Prof. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, dezembro de 2000.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Rebonatto, Marcelo Trindade

Simulação paralela de eventos discretos com uso de memória compartilhada distribuída / Marcelo Trindade Rebonatto. - Porto Alegre: PPGC da UFRGS, 2000.

100f.:il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, 2000. Orientador: Cláudio Fernando Resin Geyer.

Agradecimentos

Agradeço em primeiro lugar a Deus, pelo dom da vida.

Aos meus pais, Alberto e Lourdes, por todo o apoio e incentivo, pela amizade e presença constante em todas as etapas da minha vida; aos meus irmãos Carlos Alberto, Rita, Márcia e Ana pela parceria e solidariedade, agradeço por conviverem e entenderem uma pessoa que em determinados momentos só falava de simulação paralela, DSM ... e, muitas vezes nem presente estava para falar sobre esses assuntos totalmente estranhos.

A Cíntia, que junto a mim iniciou esta jornada, pelo companheirismo e compreensão em todos os momentos difíceis, suportando as minhas ausências e me apoiando para que eu pudesse completar este trabalho. Pela paciência de conversar comigo sobre assuntos que não tinha a mínima idéia e por certo pouco interesse em conhecer.

Ao professor Cláudio Geyer: orientador mas acima de tudo um amigo. Sua experiência e sugestões foram decisivas na realização deste trabalho. A todos os integrantes do CPPD pelas contribuições de todos, sem as quais, este trabalho não seria o mesmo. Agradecimento especial ao colega Adenauer Yamim, por produtivas conversas e sugestões de trabalho.

À Universidade de Passo Fundo pela liberação da bolsa de estudo, possibilitando assim o desenvolvimento deste trabalho. Ao Laboratório Computacional de Pesquisa do Curso de Computação da UPF pela cedência do uso dos computadores para a implementação e testes dos programas. À Biblioteca Central da UPF pelos empréstimos dos livros e materiais didáticos.

Aos colegas da turma de mestrado pelas experiências compartilhadas e solidariedade nos momentos críticos da jornada de estudo e trabalho, os quais, de uma forma ou outra, marcaram nossas vidas.

Aos meus amigos Adriano Pasqualotti, Marcos José Brusso e Alexandre Zanatta pelas valiosas discussões nos corredores da UPF sobre o assunto estudado. Grande parte do trabalho não pertence somente a mim, mas também a vocês.

A todos que conviveram comigo durante o período que estava engajado nessa jornada, direta ou indiretamente, quero registrar meus agradecimentos.

Sumário

Lista de Abreviaturas

ANSI	*American Nacional Standard Institute*
ASIM	*Arbeitsgemeinschaft Simulation*
ATW	*Adaptative Time Window*
CI	Interface de Comunicação
CMB	*Chandy, Misra e Byron protocols*
CPU	*Central Processing Unit*
CTW	*Conservative Time Window*
DSM	*Distributed Shared Memory*
EVL	*Event List*
FIFO	*First In First Out*
GPPD	Grupo de Processamento Paralelo e Distribuído
GTW	*Georgia Tech Time Warp*
GVT	*Global Virtual Time*
I/O	*Input/Output*
IQ	*Input Queue*
LAM/MPI	*Local Area Multicomputer – Messaging Passing Interface*
LAN	*Local Area Network*
LCC	*Local Causality Constraint*
LMC-IMAG	*Laboratoire de Modélisation et Calcul - Institut d'Informatique et de Mathematiques Appliquees de Grenoble*
LP	*Logical Process*
LVT	*Local Virtual Time*
MIMD	*Multiple Instruction Flow, Multiple Data Flow*
MISD	*Multiple Instruction Flow, Single Data Flow*
MP	*Messaging Passing*
MS	Máquina de Simulação
PPGC	Programa de Pós-graduação em Computação
PVM	*Parallel Virtual Machine*
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computer*
SCS	*The Society for Computer Simulation International*
SIGARCH	*Special Interest Group on Computer Architecture*
SIMD	*Single Instruction Flow, Multiple Data Flow*
SIMS	*Scandinavian Simulation Society*
SISD	*Single Instruction Flow, Single Data Flow*
SM	*Shared Memory*
SO	Sistema Operacional
SPMD	*Single Program Multiple Data*
SRADS	*Shared Resource Algorithm for Distributed Simulation*
TW	*Time Warp*
UFRGS	Universidade Federal do Rio Grande do Sul

Lista de Figuras

Lista de Tabelas

Resumo

A simulação paralela de eventos é uma área da computação que congrega grande volume de pesquisas, pela importância em facilitar o estudo de novas soluções nas mais diferentes áreas da ciência e tecnologia, sem a necessidade da construção de onerosos protótipos. Diversos protocolos de simulação paralela podem ser encontrados, divididos em dois grandes grupos de acordo com o algoritmo empregado para a execução em ordem dos eventos: os conservadores e os otimistas; contudo, ambos os grupos utilizam trocas de mensagens para a sincronização e comunicação. Neste trabalho, foi desenvolvido um novo protocolo de simulação paralela, fazendo uso de memória compartilhada, o qual foi implementado e testado sobre um ambiente de estações de trabalho, realizando, assim, simulação paralela com uso de memória compartilhada distribuída. O protocolo foi desenvolvido tendo como base de funcionamento os protocolos conservadores; utilizou diversas características dos mesmos, mas introduziu várias mudanças em seu funcionamento. Sua execução assemelha-se às dos protocolos de execução síncrona, utilizando conceitos como o *lookahead* e janelas de tempo para execução de eventos. A principal mudança que o novo protocolo sofreu foi proporcionada pelo acesso remoto à memória de um LP por outro, produzindo diversas outras nas funções relativas à sincronização dos processos, como o avanço local da simulação e o agendamento de novos eventos oriundos de outro LP. Um ganho adicional obtido foi a fácil resolução do *deadlock*, um dos grandes problemas dos protocolos conservadores de simulação paralela. A construção de uma interface de comunicação eficiente com uso de memória compartilhada é o principal enfoque do protocolo, sendo, ao final da execução de uma simulação, disponibilizado o tempo de simulação e o tempo de processamento ocioso (quantia utilizada em comunicação e sincronização). Além de uma implementação facilitada, propiciada pelo uso de memória compartilhada ao invés de trocas de mensagens, o protocolo oferece a possibilidade de melhor ocupar o tempo ocioso dos processadores, originado por esperas cada vez que um LP chega a uma barreira de sincronização. Em nenhum momento as modificações efetuadas infringiram o princípio operacional dos protocolos conservadores, que é não possibilitar a ocorrência de erros de causalidade local. O novo protocolo de simulação foi implementado e testado sobre um ambiente multicomputador de memória distribuída, e seus resultados foram comparados com dois outros simuladores, os quais adotaram as mesmas estratégias, com idênticas ferramentas e testados em um mesmo ambiente de execução. Um simulador implementado não utilizou paralelismo, tendo seus resultados sido utilizados como base para medir o *speedup* e a eficiência do novo protocolo. O outro simulador implementado utilizou um protocolo conservador tradicional, descrito na literatura, realizando as funções de comunicação e sincronização através de trocas de mensagens; serviu para uma comparação direta do desempenho do novo protocolo proposto, cujos resultados foram comparados e analisados.

Palavras-chave: simulação, simulação paralela, DSM, simulação paralela com memória compartilhada distribuída

Title: “Parallel simulation of discrete events with use of distributed shared memory”

Abstract

The parallel simulation of events is an area of computation that congregates a large volume of researches, for its importance in facilitating the study of new solutions in different areas of science and technology, without the necessity of the construction of onerous prototypes. Various protocols of parallel simulation may be found, divided into two groups according to the algorithm used for the execution in order of the events: the conservative and the optimistic; however both groups use messaging passing for communication and synchronization. In this study, it was developed a new protocol of parallel simulation, making use of shared memory, which was implemented and tested on an environment of workstations, making this way, a parallel simulation with use of distributed shared memory. The protocol was developed having as functioning basis the conservative protocols, it used various characteristics of them, but introduced several changes in their functioning. Its execution is similar to those of the protocols of synchronous execution, using concepts as lookahead and time windows for the execution of the events. The main change that the new protocol suffered, was given by the remote access to the memory of one LP by another, producing various others in the functions relative to synchronization of processes, as the local advance of the simulation and the scheduling of new events from other LP. An additional gain obtained was the easy resolution of the deadlock, one of the biggest problems of the conservative protocols of parallel simulation. The construction of an efficient communication interface with use of shared memory is the main focus of the protocol, being, at the end of the execution of a simulation, disposed the time of simulation and the time of idle processing (quantity used in communication and synchronization). Besides an easy implementation, propitiated by the use of shared memory instead of the messaging passing, the protocol offers the possibility of better use the idle time of the processors, originated by waits every time an LP gets to a barrier of synchronization. At no moment, the performed modifications infringed the operational principle of the conservative protocols, what means not to make possible the occurrence of local causality constrains. The new protocol of simulation was implemented and tested on a multicomputer environment of distributed memory, and its results were compared to two other simulators, which adopted the same strategies, with identical tools and tested in an equal environment of execution. One implemented simulator didn’t use the parallelism, having its results being used as basis to measure the speedup and the efficiency of the new protocol. The other implemented simulator used a traditional conservative protocol, described in the literature, making the communication and synchronization functions through messaging passing, it served for a direct comparison of the performance of the new proposed protocol, whose results were compared and analysed.

Keywords: simulation, parallel simulation, DSM, parallel simulation with use of distributed shared memory

1 Introdução

O presente texto consiste em um estudo sobre os protocolos de simulação paralela de eventos, tanto conservadores quanto otimistas. Além deste, detalha-se a especificação de um novo protocolo de simulação, o qual é baseado nos protocolos conservadores de simulação paralela, porém difere de todos os demais pelo fato de sua CI (Interface de Comunicação) utilizar variáveis compartilhadas ao invés de trocas de mensagens (MP - *Messaging Passing*).

1.1 Motivação

Simulações são amplamente utilizadas para analisar o funcionamento de sistemas reais ou futuras modificações nesses, possibilitando visualizar o comportamento de sistemas sem a necessidade de realização efetiva das modificações ou a construção, por vezes, de protótipos caros. A simulação computacional é empregada com grande sucesso como elemento auxiliar na tomada de decisões, principalmente no planejamento a médio e longo prazos e em situações que envolvem custos e riscos elevados. Os modelos de simulação são muito eficazes e versáteis no estudo dos mais diferentes problemas, permitindo o exame de detalhes específicos com grande precisão. Dessa forma, segundo Shannon [SHA 92], a aplicação de modelos de simulação busca:

- descrever o comportamento dos sistemas;
- construir teorias ou hipóteses que retratem o comportamento observado;
- usar o modelo para prever o futuro, isto é, os efeitos que serão produzidos pelas mudanças no sistema ou no método de operação.

Com o passar do tempo, as novas aplicações tendem a consumir cada vez maior poder computacional, porém seus tempos de execução não podem crescer em escala semelhante. Dessa forma, a simulação seqüencial pode ser insuficiente por causa das restrições impostas pelo tempo de processamento. Como alternativa, em alguns casos, tem-se a possibilidade de melhorar os algoritmos; já, em outros, algum tipo de paralelismo pode ser necessário [IKO 99], fazendo-se uso de vários processadores na simulação.

A fim de explorar o paralelismo, a carga computacional da simulação é dividida entre diversos processadores, transformando a aplicação em uma simulação paralela. A paralelização pode ser conseguida pela utilização de máquinas multiprocessadas ou pelo uso de várias máquinas monoprocessadas conectadas através de uma rede de comunicação [POR 97a].

Fujimoto [FUJ 2000] salienta que a utilização de vários processadores, distribuídos geograficamente ou não, proporciona diversos benefícios, entre os quais a redução do tempo de execução. Um menor tempo de processamento pode ser alcançado com a divisão de um processo de simulação em vários subprocessos, os quais podem ser executados paralelamente. Além disso, outros benefícios também são apontados, como a distribuição geográfica e a tolerância a falhas.

Diversos protocolos de simulação paralela de propósito geral foram desenvolvidos durante as duas últimas décadas, os quais podem ser divididos em dois grandes grupos, os conservadores e os otimistas, dependendo do algoritmo empregado para preservar a correta ordem de execução dos eventos [POR 97b]. Protocolos que não se encaixam em somente um princípio operacional para execução ordenada de eventos também podem ser encontrados. Conhecidos como protocolos híbridos, esses tentam aliar os benefícios dos conservadores e dos otimistas [VAC 99].

Durante a execução em paralelo de uma simulação, os processadores que a executam necessitam comunicar-se para que suas informações possam ser acessíveis aos demais. Os dados que devem ser trocados dizem respeito ao avanço da simulação, a requisições oriundas de dependência de dados e a solicitações de processamento de informações. A forma de comunicação usada tradicionalmente nos protocolos de simulação paralela é a troca de mensagens com um *timestamp* [FER 96], não existindo dados compartilhados entre os processadores. Todos os protocolos de simulação estudados comunicam-se por MP.

A programação no paradigma de memória compartilhada é considerada mais simples por evitar que o programador tenha de se preocupar com a comunicação entre os processos através de trocas explícitas de mensagens [ARA 99]. A proposição de um novo protocolo de simulação, baseado em variáveis compartilhadas, facilitaria a implementação de simulações paralelas; por conseqüência, o programador ficaria com maior tempo disponível para se dedicar às questões relativas ao sistema a ser simulado.

Para a criação de um novo protocolo de simulação fazendo uso de memória compartilhada, vários problemas têm de ser resolvidos, entre eles, a correta execução ordenada dos eventos e a falta de desempenho, sobretudo quando a memória a ser compartilhada estiver distribuída. Os ambientes com memória fisicamente distribuída são os mais facilmente encontrados e, quando conectados com uma rede de alta velocidade, fazem grande concorrência aos caros computadores paralelos [GEO 99].

1.2 A contribuição do autor

O principal objetivo do presente trabalho é a realização de um processo de simulação paralela em um ambiente com memória distribuída usando variáveis compartilhadas. Para que a simulação pudesse ser realizada, um novo protocolo de simulação paralela foi desenvolvido, usando-se em sua CI (Interface de Comunicação) variáveis compartilhadas. Essas variáveis foram utilizadas para controlar o avanço dos processadores engajados na simulação e para que uma unidade processadora pudesse criar novos eventos a serem executados em outra.

O novo protocolo de simulação baseou-se nos protocolos conservadores de simulação paralela. Dessa forma, os problemas pertinentes a esses protocolos tiveram de ser solucionados, entre eles, o baixo paralelismo na execução e o controle sobre as possíveis situações de *deadlock*. Como os estudos da simulação concentraram-se na CI, as operações realizadas pela MS (Máquina de Simulação) foram tratadas de forma superficial; em outras palavras, nenhuma aplicação específica foi simulada, sendo extraídos apenas valores relativos ao tempo de processamento e à ociosidade das unidades processadoras.

A implementação da simulação foi baseada em estações de trabalho ligadas através de uma rede de comunicação. Em tais ambientes, aplicações que utilizam DSM (*Distributed Shared Memory*) são geralmente mais lentas do que as que fazem uso de trocas de mensagens. A fim de proporcionar uma comparação entre a simulação por MP e a simulação por memória compartilhada, também se implementou um protocolo conservador de simulação utilizando trocas de mensagens, com o que, então, tal comparação pôde ser realizada.

Ainda, pode ser considerada como outra contribuição o domínio da programação paralela através de memória compartilhada distribuída utilizando a ferramenta *Athapascan0*. Esta biblioteca de comunicação é amplamente dominada pelos integrantes do GPPD (Grupo de Processamento Paralelo e Distribuído) da UFRGS (Universidade Federal do Rio Grande do Sul) na programação com trocas de mensagens, contudo ainda era desconhecida sua programação usando DSM.

1.3 Organização do texto

Tomando por base os fundamentos da simulação paralela e as possibilidades de programação de aplicações paralelas, apresenta-se uma revisão dos modelos de simulação paralela e dos paradigmas da programação paralela. Circunstanciado por essa revisão crítica, apresenta-se o protocolo proposto para simulação paralela usando variáveis compartilhadas, cuja descrição é enriquecida pela discussão dos aspectos pertinentes a sua implementação e pelos resultados de testes envolvendo parâmetros de simulações reais.

A dissertação está dividida em duas partes. A parte 1, “o contexto”, compõe-se dos capítulos 2 e 3. No capítulo 2, realiza-se um estudo sobre a simulação paralela de eventos discretos, analisando-se o processo de simulação paralela e seus componentes, os protocolos conservadores e otimistas, com suas principais vantagens e problemas. No capítulo 3, abordam-se as questões relativas a implementações de aplicações paralelas, tais como ambientes, formas de exploração do paralelismo e paradigmas utilizados na programação.

A parte 2, “a proposta”, registra a contribuição central do autor, estando composta pelos capítulos 4, 5, 6 e 7. No capítulo 4, descreve-se o modelo do protocolo de simulação paralela com uso de variáveis compartilhas, enfatizando suas estruturas de memória e o algoritmo de funcionamento; no capítulo a seguir, detalha-se a implementação realizada do protocolo proposto e de outros dois, um seqüencial e um com uso de MP, ambos utilizados para comparações. Os resultados obtidos por todos os simuladores implementados são discutidos no capítulo 6; por fim, o capítulo 7 registra as conclusões do trabalho.

2 Simulação paralela

Este capítulo apresenta um resumo sobre simulação paralela de eventos discretos. Inicialmente, analisam-se o processo de simulação em geral, o processo de simulação paralela e seus componentes. Após uma exposição inicial dos principais conceitos, abordam-se os protocolos conservadores e otimistas, cada um com suas principais variações, problemas e benefícios. Ao final, realiza-se uma breve comparação dos protocolos, apontando suas principais características, pontos positivos e negativos.

2.1 Introdução

A simulação baseada em computadores é amplamente usada nos dias de hoje em sistemas de predição e análise de desempenho. Entretanto, com o aumento da demanda de grandes modelos e cada vez mais complexos, a simulação seqüencial vem se tornando menos atrativa em virtude do tempo de simulação consumido. Na tentativa de reduzir o tempo de simulação e aumentar o tamanho dos sistemas simulados, diversos protocolos de simulação paralela foram desenvolvidos nas últimas duas décadas [PHA 99a].

Os protocolos utilizados para a simulação paralela dividem-se em dois grandes grupos: conservadores e otimistas. Os protocolos conservadores, propostos inicialmente por Chandy e Misra, processam os eventos seguindo estritamente a sua ordem seqüencial de acontecimento. Por sua vez, a ênfase dos protocolos otimistas, pela primeira vez propostos por Jefferson, está na detecção e recuperação de erros de causalidade local. Ambos possuem diversas variações, nas quais lhes foram acrescidas melhorias [FER 96], e são intensamente usados na simulação paralela de eventos discretos [POR 99].

2.2 Simulação de eventos

Simular significa reproduzir o funcionamento de um sistema do mundo real com o auxílio de um modelo, o que permite testar hipóteses sobre o valor de alguns componentes controlados [SIL 98]. A simulação baseada em computadores é uma das ferramentas mais poderosas disponíveis para projetar, planejar, controlar e avaliar novas alternativas e/ou mudanças de estratégia em sistemas do mundo real [SHA 92].

A representação desse funcionamento deve ser realizada no decorrer de um tempo. Durante o funcionamento, é realizado o avanço do tempo, de maneira discreta ou continua, o qual é chamado de tempo de simulação ou tempo simulado. O tempo simulado pode ou não coincidir com o tempo físico (de relógio) que a aplicação computacional consome durante sua execução. É comum o uso de escalas de tempo em processos de simulação discreta com o objetivo de acelerar o processo [FUJ 2000].

Os simuladores utilizam estratégias para representar as visões do mundo ou o sistema a ser simulado [FUJ 2000]. Na realização da simulação, podem ser utilizadas três perspectivas para mimetizar a mudança de estados do sistema real: simulação por escalonamento de eventos, por exame de atividades e por interação de processos

[COP 96]. Um evento acontece num ponto isolado do tempo, no qual decisões devem ser tomadas de forma a iniciar ou terminar uma atividade; um processo é uma seqüência ordenada de eventos que pode englobar várias atividades ([SOA 90], [OVE 91], [NOG 98]).

A simulação computacional baseada em eventos possui, ainda, uma divisão com relação aos tipos de eventos, os quais podem ser discretos ou contínuos. Contínuos são aqueles eventos que proporcionam uma mudança do tempo de maneira suave, sem descontinuidade ou atualizações abruptas; por sua vez, discretos são os eventos cuja variação do tempo se dá de maneira descontínua, ocorrendo somente em momentos predeterminados [NOG 98].

A simulação de eventos discretos possui, para o seu funcionamento, uma lista ordenada (por tempo de execução) de eventos a serem simulados, conhecida como *Event List* (EVL). O processo de simulação ocorre pela seleção do evento com o menor tempo para execução (*timestamp*) e sua posterior execução, atualizando, dessa forma, o tempo da simulação. Na execução de um evento, este pode agendar um novo evento a ser executado em um tempo futuro ([FUJ 90], [FER 96], [MUS 99]).

2.3 Componentes da simulação paralela

Sendo a predição um dos principais objetivos da utilização de modelos de simulação, é desejável que tais tarefas sejam executadas o mais rapidamente possível. A utilização de ambientes paralelos para a execução de simulações possibilita que se agilize a obtenção dos resultados [VAC 99].

Grandes simulações nas mais diversas áreas consomem significativos volumes de recursos computacionais e elevado tempo de execução em máquinas seqüenciais. Uma tendência natural é tentar utilizar vários processadores para acelerar o processo de simulação [REB 99a], contudo a maior desvantagem desse procedimento é a complexidade inerente a esse tipo de simulação, uma vez que a idéia do “tempo global” não é facilmente manipulável na computação paralela [OVE 91].

Na simulação paralela de eventos, o sistema a ser simulado é dividido em subsistemas a serem executados de forma paralela nos processadores, criando em cada um LP (*Logical Process*). Cada LP pode trabalhar com a evolução de seu tempo de processamento independentemente de outros LPs, mantendo, assim, seu próprio LVT (*Local Virtual Time*). O conjunto de LVTs de todos os LPs irá determinar o valor do GVT (*Global Virtual Time*), que representa o avanço geral da simulação. A execução da simulação num LP pode utilizar ou alterar dados pertencentes a outros LPs [FER 96], necessitando de sofisticados algoritmos de comunicação para sincronização de relógios locais e intercâmbio de informações [OVE 91].

2.3.1 Erros de causalidade local

Um dos principais problemas dos modelos de simulação paralela é a forte correlação entre os seus componentes. Para que a execução em paralelo de um modelo de simulação seja correta, um pré-requisito a ser atendido é o da garantia da não-ocorrência de Erros de Causalidade Local (ou *Local Causality Constraint* – LCC) ([FUJ 90], [FER 96]).

Erros de causalidade local são provocados pelo tratamento de eventos em ordem temporal não crescente durante a simulação. Dessa forma, eventos "futuros" podem, erroneamente, afetar o comportamento do modelo quando da execução de fatos "passados". Para evitar tais erros, é necessária a utilização de protocolos de sincronização, de modo a certificar-se de que o modelo está sendo executado corretamente com relação à ordenação temporal dos eventos [VAC 99].

Seqüências de processamento de eventos em que existe dependência entre dados devem ser executadas mantendo sua ordenação a fim de evitar a contradição de que ações no futuro possam influenciar as do passado. Por outro lado, eventos que não possuam relação de causalidade podem ser executados em qualquer ordem, sem que isso comprometa a veracidade dos dados oriundos do modelo ([REB 99a], [FUJ 2000]).

2.3.2 Processo de simulação paralela

Pode-se visualizar a simulação paralela como a cooperação de um conjunto de LPs interagindo, cada um deles simulando uma parte ou região do sistema do mundo real. Geralmente, uma região é representada pelo conjunto de todos os eventos a ela relacionados no mundo real [FER 96].

Um LP possui duas funções distintas: a execução de eventos e a comunicação com outros LPs. A execução de eventos é realizada pela máquina de simulação (MS), que executa somente os eventos locais, podendo, em sua execução, agendar um novo evento a ser inserido em sua própria lista ou em outra. O controle do avanço local da simulação também é controlado pela MS. A comunicação é realizada pela interface de comunicação, que utiliza um sistema para possibilitar a troca de informações, bem como a sincronização dos LPs. A interface de comunicação, ligada à máquina de simulação, cuida, ainda, da propagação de efeitos causais em eventos a serem executados por LP remotos. A arquitetura básica de uma simulação paralela é mostrada na Figura 2.1.

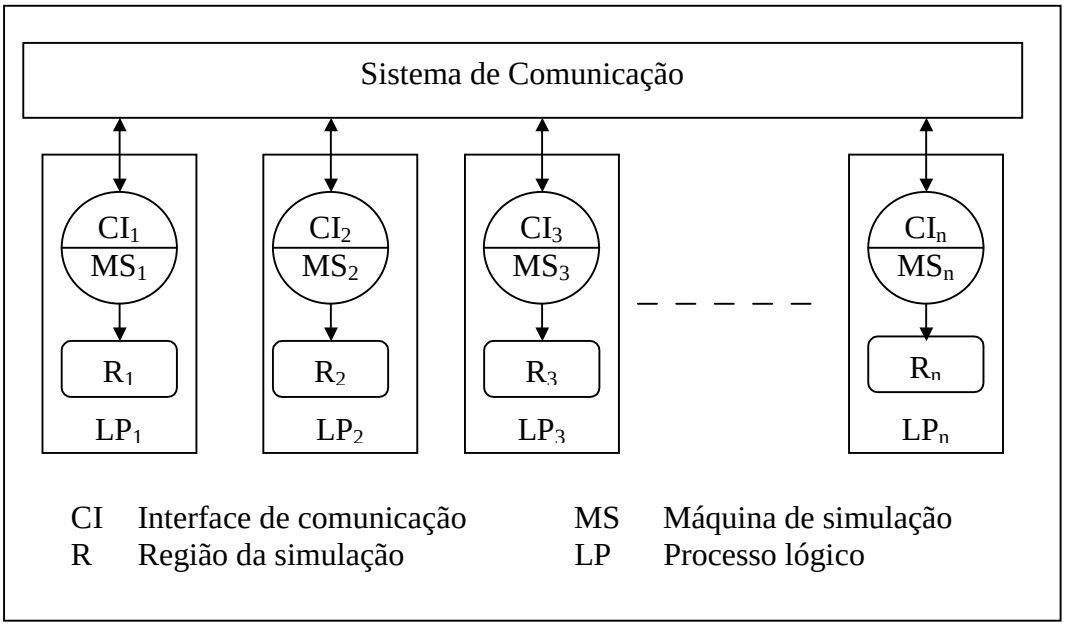


FIGURA 2.1 – Processo de simulação paralela

Foram propostos diversos protocolos de simulação paralela com o objetivo de tratar a aderência aos LCC. O desempenho de tais protocolos é intrinsecamente dependente do ambiente, da finalidade e do tipo de modelagem utilizado no processo de simulação, não tendo sentido sua classificação. Porém, duas linhas claras de abordagem podem ser determinadas:

- *protocolos conservadores*: inicialmente propostos por Chandy e Misra (1979) e Bryant (1984) apud ([FER 96] [FUJ 2000]), são assim denominados por se aproximarem muito dos modelos de execução tradicional de simulação. Neste tipo de protocolo, um evento somente é executado quando todos os eventos que podem afetá-lo já o foram. Dessa forma, sua funcionalidade inibe a ocorrência de LCC;

- *protocolos otimistas*: buscam explorar a violação aos LCC sendo baseados na observação empírica de que a ocorrência de erros de causalidade tende a ser minimizada pelo próprio processo de simulação. Assim, é permitida a execução de eventos mesmo que não se tenha a garantia de que seja temporalmente segura, tratando os erros de causalidade através de processos de *rollback* [MUS 99].

2.4 Protocolos conservadores

Os protocolos conservadores, também conhecidos como CMB (Chandy, Misra e Byron) por causa de seus criadores, não permitem a ocorrência de LCC. O princípio de funcionamento desses protocolos consiste em determinar quando é seguro executar um evento. Mais precisamente, se determinado LP possui um evento não processado E_1, com *timestamp* T_1, não existindo nenhum outro com tempo para execução inferior ao seu, e o LP puder determinar que é impossível o recebimento posterior de evento com

tempo menor que o T_1 a ser executado, o evento E_1 é um evento seguro ([FUJ 90], [REB 99a]).

Os protocolos CMB necessitam da especificação dos *links* de comunicação, indicando, para cada LP, todos os outros com os quais este pode se comunicar. Para determinar quando um evento pode ser executado, é necessário que todos os outros LPs enviem mensagens aos demais, transmitindo-as através do *link* de comunicação em ordem crescente de tempo para execução. Esse procedimento garante que a última mensagem recebida em um *link* tenha o maior *timestamp* entre as já recebidas [VAC 99].

As mensagens que chegam por um determinado *link* são armazenadas em ordem FIFO, que é a mesma ordem dos tempos das mensagens a serem executadas. Cada *link* possui um relógio, com valor igual ao da primeira mensagem da fila, caso a mesma contenha mensagens; ou igual ao da última mensagem recebida, se estiver vazia.

Em seu funcionamento, o LP, repetidamente, seleciona, dentre os eventos seguros a serem processados, o de menor *timestamp*, executando-o. Os LPs contendo eventos a processar não seguros devem ser bloqueados enquanto permanecerem nessa situação. A situação de bloqueio pode ser originada pela inexistência de eventos a serem processados na lista com o menor relógio, mas existirem eventos em outras oriundas de outros *links* de comunicação; dessa forma, ocasionam-se situações de *deadlock* que devem ser tratadas ([FUJ 90], [OVE 91], [FER 96]).

Os protocolos conservadores diferem, ainda, na forma como suas principais variações são implementadas. Abordagens síncronas, como “janela de tempo conservadora” e *lookahead*, podem explorar os eficientes mecanismos de sincronização em *hardware* encontrados em ambientes multiprocessados. Por outro lado, abordagens assíncronas, como “prevenção” e “detecção e recuperação” de *deadlock*, adaptam-se melhor a ambientes de memória distribuída ([REB 99a], [VAC 99], [FUJ 2000]).

2.4.1 Prevenção de *deadlock*

A situação de *deadlock* ocorre num ciclo de processos lógicos quando esses estão bloqueados. Fujimoto [FUJ 90] aponta que *deadlocks* podem ser freqüentes em situações em que existe um baixo número de eventos em relação ao número de ligações entre processos lógicos. Por exemplo, ele pode ser originado pela existência de filas vazias, quando essas possuem valor de relógio com tempo inferior às que possuem eventos. A Figura 2.2 demonstra essa situação, observando-se que isso ocorre porque a fila com o menor *timestamp* está vazia. Por exemplo, o primeiro processo possui uma fila vazia com tempo 12 e possui uma fila com eventos, cujo relógio vale 22. Uma situação semelhante ocorre com os demais processos; assim, nenhum processo pode seguir adiante, pois é incapaz de determinar um evento seguro, mesmo possuindo eventos não executados.

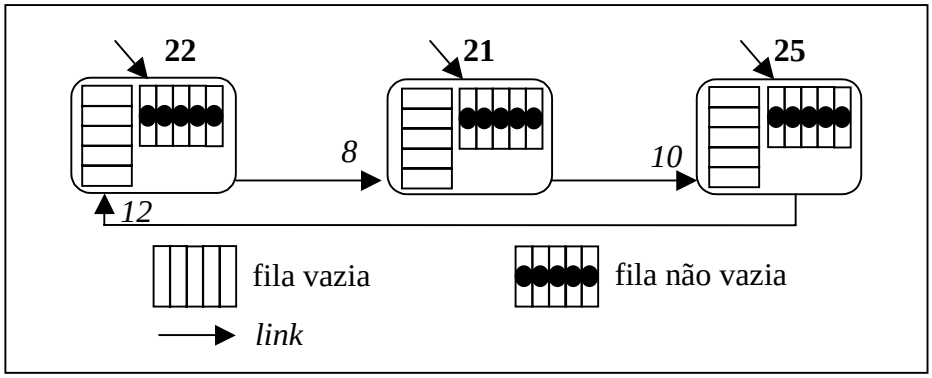


FIGURA 2.2 – Exemplo de *deadlock*

Mensagens nulas (que denotam eventos sem efeito) podem ser utilizadas para evitar as situações de *deadlock*. As mensagens nulas não têm relação com o modelo a ser simulado e servem apenas para sincronização entre os LPs. Uma mensagem nula possui um *timestamp* nulo (T_{null}), que é enviado do LP_A para o LP_B, dizendo para LP_B que LP_A não irá enviar qualquer outra mensagem com um *timestamp* menor no futuro. Elas foram introduzidas pelos criadores dos protocolos conservadores e eram enviadas sempre que a execução de um evento não gerava mensagens para um LP [BAR 96].

Determinar o *timestamp* das mensagens nulas a serem enviadas é fator decisivo para o funcionamento dessa abordagem. Uma forma para fazer isso é usar o menor valor das filas de chegadas de mensagens, que é o tempo do próximo evento a ser executado. Após encontrar o *timestamp* T_{null}, mensagens contendo essa informação devem ser enviadas aos demais LPs, os quais, ao recebê-las, atualizam seus relógios, repassam a informação adiante e podem executar seus eventos seguros [FUJ 2000].

Um problema que a abordagem de mensagens nulas enfrenta é o *overhead* trazido pela grande quantidade de mensagens. Melhorias no processo de envio definido na abordagem inicial foram propostas, entre elas: (a) sob-demanda: quando um processo fica bloqueado, começa a enviar mensagens nulas; (b) por requisição: o LP faz requisições de mensagens nulas aos processos vizinhos apenas quando fica bloqueado; (c) por tempo: após bloqueado, o processo espera um tempo para enviar/requisitar mensagens nulas ([VAC 99], [REB 99a], [FER 96]).

2.4.2 Detecção e recuperação de *deadlock*

Esta técnica permite a ocorrência de *deadlock*, mas fornece mecanismos para a detecção e recuperação da paralisação e tem como vantagem um menor número de mensagens nulas. O algoritmo executa duas fases: uma processando eventos até a ocorrência de *deadlock* (fase paralela) e outra que, após detectada a paralisação dos LPs, recupera o sistema elegendo um evento a ser processado (fase de interface). A técnica demonstra a existência na fase paralela de, pelo menos, um evento que pode ser processado, gerando, no mínimo, uma mensagem de evento, a qual poderá ser propagada antes do próximo *deadlock* [FER 96].

Uma forma simples de recuperação é utilizar o evento com menor *timestamp* nos *buffers* das filas de entrada e executá-lo. Sua localização é uma tarefa relativamente

direta, pois a execução está bloqueada (*deadlock*) e novos eventos não estão sendo criados [FUJ 2000].

O mecanismo de detecção baseia-se na falta de comunicação entre os processos para indicar a ocorrência de *deadlock*. Um controlador central deve ser implementado, o que viola princípios da computação distribuída. Para evitar que um único recurso torne a comunicação um gargalo na detecção do *deadlock*, pode ser utilizado um controlador distribuído [CHA 83]. Outras alternativas, como mensagens *marker* e pré-processamento para detecção de *deadlock* local, são encontradas. Informações mais completas, referências e explicações podem ser encontradas em ([FUJ 90], [FER 96], [VAC 99]).

2.4.3 Janela de tempo conservadora

Esta variação ao protocolo introduz uma janela lógica no tempo dos eventos, na qual os que são por ela compreendidos possuem livre execução (são considerados seguros), sem necessitar de nenhum tipo de verificação; podem, assim, ser executados paralelamente [AYA 92]. O algoritmo divide-se em duas fases: a primeira determina o tamanho da janela, delimitando os eventos compreendidos como seguros, e a segunda é responsável pelo tratamento dos eventos contidos na janela em ordem cronológica.

É importante notar que, ao final de cada fase, é necessária uma sincronização de todos os processos lógicos. Como as sincronizações podem ser freqüentes, o desempenho do algoritmo está diretamente relacionado aos recursos de sincronização da arquitetura alvo. O mecanismo que determina o tamanho da janela possui papel crítico no desempenho, uma vez que janelas muito pequenas ou grandes em demasia podem trazer ciclos de inatividade na simulação [FUJ 90]. Um exemplo de funcionamento da janela de tempo conservadora pode ser visualizado na Figura 2.3, adaptada de [MUK 97].

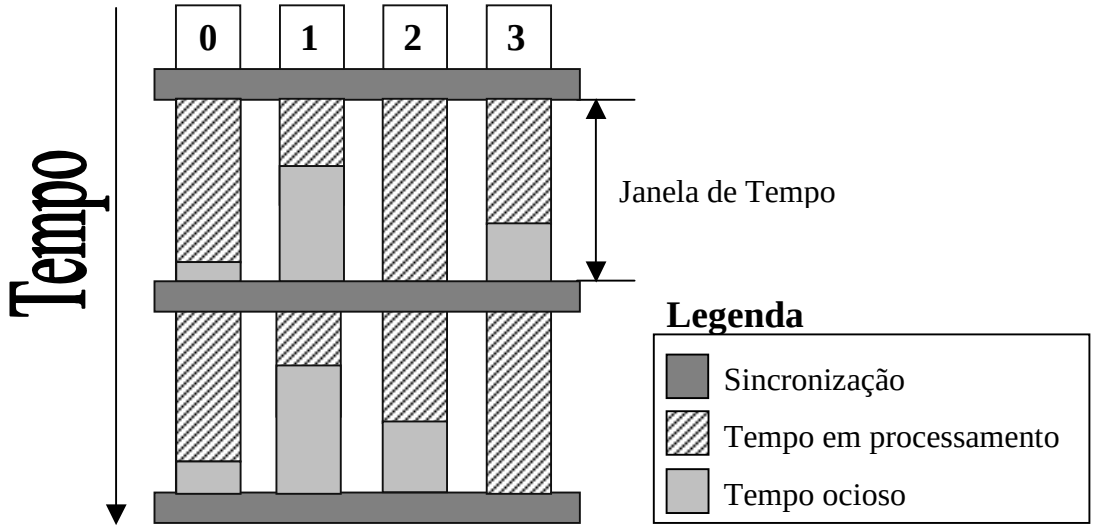


FIGURA 2.3 – *Conservative Time Windows*

Nem todos os modelos de simulação podem ser implementados usando-se a abordagem *conservative time-windows* (CTW). Os sistemas do mundo real que não

possuam conjuntos de eventos onde possa ser determinada uma janela para a execução não se adaptam a essa técnica.

2.4.4 *Lookahead*

Lookahead (olhar a frente) é a habilidade de um processo em predizer quais serão os eventos gerados no futuro. Se um processo no tempo simulado "T_{atual}" puder determinar que todos eventos serão gerados acima do tempo "$T_{atual} + L$", diz-se que o processo tem um *lookahead* L. Esse é utilizado principalmente nos protocolos de prevenção de *deadlock*, nos quais essa informação é adicionada à mensagem nula, garantindo, assim, que nenhum evento seja enviado com marca de tempo menor que "$T_{atual} + L$" [FUJ 2000].

Nos modelos em que a geração de eventos segue uma distribuição de probabilidades qualquer, muitas vezes calculada em tempo de execução, Nicol propõe a pré-computação de eventos, a qual possibilita que *lookaheads* possam ser calculados em tempo de simulação, aumentando o desempenho do protocolo. Porém, esta metodologia somente pode ser aplicada quando a geração de eventos é independente de informações contidas nas mensagens [MUS 99]. Por exemplo, se o tempo de serviço de evento é determinado pelo tamanho das informações, nenhuma pré-computação pode ser realizada até a chegada efetiva da mensagem.

A utilização de *lookahead* fixo e igual em todos os LPs durante o processo de simulação alivia o LP do custo de pesados cálculos para determinar o valor do *lookahead*. Este valor deve ser informado no início da simulação e varia drasticamente de um sistema para outro. Com o uso de *lookahead* não calculado em tempo de simulação, a máquina de simulação do LP ocupa-se apenas com a execução dos eventos pertinentes ao modelo ([POR 97a], [POR 97b], [NKE 98], [POR 98]).

2.4.5 Críticas aos protocolos conservadores

Um dos grandes problemas das técnicas conservadoras é que não podem explorar de forma completa o paralelismo nas aplicações de simulação. Caso o evento E_A afete E_B, direta ou indiretamente, as técnicas conservadoras devem executar E_A e E_B de forma seqüencial. No caso em que E_A raramente afete E_B, esses eventos poderiam ser processados de forma paralela na maioria das vezes. Geralmente, as técnicas conservadoras forçam uma execução seqüencial quando essa não é obrigatoriamente necessária, o que ocasiona um fraco paralelismo nas aplicações, principal problema dos protocolos conservadores na questão de desempenho.

Os protocolos conservadores também sofrem, invariavelmente, da possibilidade de ocorrência do *deadlock*. Algumas de suas variações evitam-no e outras permitem sua ocorrência, tratando-o após. De qualquer forma, ele é um problema a ser solucionado na construção de uma aplicação de simulação paralela conservadora.

Um recurso passível de ser empregado para melhorar o desempenho é a utilização de *lookahead*, a qual depende muito do conhecimento do modelador para que determine um valor correto para o sistema. Porém, um novo problema surge, que é

exatamente a determinação desse valor, visto que um valor errado pode acarretar os seguintes problemas: (a) pessimismo exagerado: valor pequeno, que não corresponde ao paralelismo presente no modelo, conseqüentemente, não obtém o desempenho máximo; (b) otimismo exagerado: valor alto, também não condizente com o modelo, podendo ocasionar erros nos resultados da simulação.

A falta de flexibilidade dos protocolos conservadores, por necessitarem de configurações estáticas (dos *links* de comunicação) quando do início dos processos, é outro ponto negativo; também a necessidade de um conhecimento profundo sobre o comportamento dos processos a serem simulados, a fim de incluir a sincronização de forma explícita nos programas, não facilita sua implementação. Dessa forma, uma das maiores desvantagens é que o programador tem de se concentrar nos detalhes do mecanismo de sincronização para obter um bom desempenho [FUJ 90].

2.5 Protocolos otimistas

Apresentados, inicialmente, por Jefferson e Zowizral em 1985, apud ([FER 96], [FUJ 2000]), os protocolos otimistas detectam e recuperam erros de causalidade, mas não impedem que esses ocorram. Diferindo dos conservadores, eles não necessitam determinar quando é seguro processar um evento; permitem a ocorrência de erros LCC e, quando da ocorrência de um, chamam um procedimento para recuperar o processamento perdido [BAR 96].

Uma das vantagens desses mecanismos é que permitem ao simulador explorar o paralelismo em situações nas quais é possível a ocorrência de erros de causalidade, mas que, de fato, não ocorrem. Em observações empíricas da evolução de experimentos de simulação, foi constatado que, em grande parte das situações em que havia risco de erros LCC, eles não ocorriam. Dessa forma, ao tratar a ocorrência de erros de causalidade como uma exceção, e não como uma regra, pode-se explorar naturalmente o paralelismo intrínseco dos modelos e apresentar melhores desempenhos. A criação dinâmica de novos processos lógicos pode ser facilmente implementada, sendo também um adicional desses protocolos [FUJ 90].

O principal problema dos protocolos otimistas está no momento da ocorrência de erros LCC, situação em que os LPs devem retornar a um estado seguro, anterior ao tempo atual simulado, para, só então, reexecutarem os eventos. O problema pode tornar-se ainda maior quando o retrocesso de eventos inclui comunicações entre LPs. Nesse caso, todos os LPs vão retornando até encontrarem um estado seguro, podendo sofrer forte efeito dominó. Exemplos de estratégias para diminuir o efeito dominó podem ser encontrados em ([JOH 93], [XU 93], [BAL 95], [CON 97]). Resta salientar que, para que os processos retornem a um estado seguro, eles devem ser armazenados em memória estável. O mecanismo de *rollback* exige que os LPs registrem os estados da simulação, acumulando informações de seus eventos internos e externos de forma cronológica. Os externos dizem respeito às filas de recebimento e envio de mensagens, ao passo que os internos tratam do conjunto de variáveis pertencentes ao processo. Aos estados salvos dá-se o nome de *checkpoint* ou ponto de recuperação ([JAL 94], [CAM 96], [LUM 97]).

O cálculo do avanço global da simulação (GVT) é outra atribuição dos LPs a ser realizada. Diferentemente dos protocolos conservadores, nos quais o menor LVT pode ser considerado como GVT, os otimistas empregam complicados cálculos para determinar o GVT, que serve, entre outros motivos, como um dos quesitos para a liberação de memória de estados passados que não mais serão utilizados (*fossil collection*) [FER 96].

2.5.1 Cancelamento agressivo (*Time Warp*)

Time Warp ou *Aggressive Cancellation* (TW) foi o primeiro protocolo otimista criado, sendo também o mais conhecido. De forma semelhante aos protocolos CMB, utiliza a troca de mensagens para a sincronização. Este protocolo determina como seguros os eventos com o menor *timestamp* entre os não processados nos LPs. Dessa forma, a execução da simulação pode avançar mais em um LP do que em outro, tornando o processo de *rollback* obrigatório de ser implementado.

Este protocolo utiliza um mecanismo para refazer o processamento quando erros LCC ocorrem, os quais são detectados ao ser recebida qualquer mensagem contendo um *timestamp* menor que o LVT atual. Essas mensagens, denominadas *stragglers*, acionam o mecanismo de recuperação, destruindo todas as alterações realizadas por processos prematuramente processados, voltando com o relógio local a um ponto consistente da simulação, com *timestamp* inferior à mensagem recebida ([REB 99a], [MUS 99]).

Existem dois tipos básicos de mensagens para implementar na CI: (a) mensagens positivas: referentes ao modelo que está sendo simulado; (b) mensagens negativas ou antimensagens: indicam quando eventos foram prematuramente processados [FER 96]. O processo de *rollback* é implementado usando as mensagens negativas (m-) para anular as positivas (m+); antes, porém, os efeitos da mensagem positiva são convenientemente tratados. A Tabela 2.1 ilustra as ações do algoritmo para anular as mensagens.

Tabela 2.1 – Tratamentos das mensagens em TW

Instante (m)	Chegada de m+	Chegada de m-
>= LVT (Futuro local)	**Se**: m- já se encontra na IQ(*) **Então**: aniquila m+	**Se**: par m+ encontra-se na IQ(*) **Então:** aniquila m+
	Se: m- não se encontra na IQ(*) **Então:** insere em ordem m+	**Se:** m+ não se encontra na IQ(*) **Então:** insere em ordem m-
< LVT (Passado local)	**Se**: m- já se encontra na IQ(*) **Então**: aniquila m-	**Se**: par m+ já processada **Então**: realiza *rollback* e aniquila m+
	Se: m- não se encontra na IQ(*) **Então**: realiza *rollback* e insere em ordem m+	**Se**: m+ não se encontra na IQ(*) **Então**: insere em ordem m-

(*) *input queue*: fila de entrada local
Fonte: VACCARO. Simulação paralela e distribuída com vistas ao *co-design*. p.29

2.5.2 *Lazy cancellation* e *Lazy reevaluation*

Como uma alternativa ao cancelamento agressivo, Gafni, em 1998, apud ([FER 96], [FUJ 2000]), propôs o "cancelamento preguiçoso" (*Lazy Cancellation*), cuja estratégia é de, no momento da chegada de um *straggler*, não enviar imediatamente as mensagens negativas. Esse protocolo faz uma comparação entre as mensagens enviadas e as novas geradas, enviando antimensagens apenas para as que são diferentes [VAC 99].

Dependendo da aplicação, a técnica de retardar o envio das antimensagens pode melhorar ou piorar o desempenho, gerando um *overhead* adicional, quando do processamento de uma mensagem, para verificar se existe a antimensagem correspondente, sendo necessária uma ou mais mensagens para realizar essa tarefa. Em contrapartida, possibilita que os estados do sistema sejam salvos com menor freqüência, necessitando de menos memória que o cancelamento agressivo [OVE 91].

Outra alternativa ao TW é uma técnica baseada numa forma muito semelhante ao atraso no envio das mensagens, chamada de "reavaliação preguiçosa" (*Lazy Reevaluation*) proposta por West em 1988, apud ([FER 96], [FUJ 2000]). Essa técnica difere da *Lazy cancellation* por operar com vetores de estado no lugar de mensagens, consistindo em avaliar se o estado do sistema é igual antes e depois da chegada e processamento de um *straggler*. Caso não tenham chegado novas mensagens, outra execução irá produzir os mesmos valores da original. Pode-se, após essa avaliação, "saltar" (*Jump Forward*) sobre os eventos, dependendo do exame do vetor de estados ([FER 96], [VAC 99]).

Um exemplo de que a *Lazy Reevaluation* pode representar uma grande melhoria no desempenho ocorre quando um evento denota uma tarefa apenas de leitura nos processos. Neste caso, não são necessárias novas reexecuções resultantes de consultas ([FER 96], [REB 99a]).

Estes protocolos exploram a invariância de certos eventos em relação à ocorrência de *stragglers*, mesma propriedade em que se baseia o processo de *lookahead* dos protocolos conservadores. A vantagem dos otimistas é o fato de serem intrinsecamente capazes de explorar o *lookahead* de um problema, desobrigando o analista de se preocupar com tais aspectos no momento da implementação do modelo. Isso ocorre em virtude da realização de *rollbacks* somente quando os efeitos de um *straggler* provocarem alterações de estados significativas [FUJ 90].

2.5.3 *Optimistc time windows*

Janelas de tempo implementadas de maneira semelhante à dos protocolos pessimistas também são propostas nos otimistas. Neste caso, as janelas de tempo são utilizadas para permitir a execução somente dos eventos compreendidos nelas, evitando que algum LP avance em demasia e tenha de refazer muitos eventos. Esta variação utiliza uma quantia tempo "W", que pode ser dinamicamente modificada para estabelecer a janela. Somente os eventos que irão ocorrer dentro do intervalo de tempo [T .. T + W] são passíveis de serem executados. O tempo T é associado ao menor *timestamp* dentre os eventos não processados ([REB 99a], [FUJ 2000]).

Este protocolo, originalmente proposto por Sokol, Briscoe e Wieland em 1988, apud ([FER 96], [FUJ 2000]), carece de processos com eficiência cientificamente comprovada para a geração do tamanho da janela. A principal crítica feita a ele é de que as janelas de tempo não fazem distinção entre processos seguros e não seguros, impedindo o progresso em paralelo de corretas computações [FUJ 90].

Uma versão mais recente, denominada *Adaptative Time Warp concurrency control algorithm* – ATW, desenvolvida por Ball e Hyot em 1990, apud ([FER 96], [FUJ 2000]), traz melhoramentos ao mecanismo de janela de tempo otimista, permitindo que o seu tamanho seja adaptado dinamicamente através da suspensão temporária do processamento de eventos nos LPs que excederem um número de *rollbacks* predeterminado [VAC 99].

2.5.4 *Wolf calls* e *Direct cancellation*

Proposto por Madisetti, Walrand & Messerschmitt em 1988, apud ([FER 96], [FUJ 2000]), o mecanismo denominado *Wolf calls* representa uma das primeiras tentativas de redução do efeito dominó causado pelos *stragglers*. Neste protocolo, *stragglers* provocam o envio de mensagens de controle especiais de alta prioridade (*wolf calls*) que param os processos afetados.

A literatura aponta desvantagens desta abordagem: (a) alguns processos podem ser desnecessariamente afetados, nos casos em que protocolos *Lazy Cancellation* e *Lazy Reevaluation* tiram vantagem; (b) o conjunto de processos que são afetados é significativamente menor do que os que poderiam ser afetados. Além disso, o protocolo requer que se saiba qual é o tempo real de envio de mensagens de modo a calibrar a velocidade das *wolf calls*, tarefa que pode não ser trivial, dependendo do meio físico disponível e da taxa efetiva de uso [FER 96].

Outra alternativa é o *Direct cancellation*, proposto por Fujimoto em 1990 [FUJ 90]. Este protocolo tem por objetivo evitar a geração de grandes cascatas de *rollbacks* em razão da velocidade de transmissão de mensagens contaminadas por informações inconsistentes. Ao invés de propor mensagens de alta prioridade, como no caso das *Wolf Calls*, Fujimoto propôs a manutenção de ponteiros para cada mensagem gerada. Assim, no caso de ocorrência de *stragglers*, o algoritmo somente tem de eliminar as mensagens através dos ponteiros e restabelecer os estados consistentes.

As vantagens apresentadas por este protocolo são a redução de *overheads* para geração e envio de antimensagens e boa eficiência no restabelecimento de estados consistentes para o sistema. Como desvantagem, há a necessidade da utilização de uma linguagem com suporte a ponteiros para entidades temporárias [FUJ 90].

2.5.5 Críticas aos protocolos otimistas

Os protocolos otimistas necessitam evitar um comportamento de obstrução ao processamento, isto é, impedir que seja gasta a maior parte do tempo executando-se computações incorretas. Dessa forma, eles devem manter a quantidade de computações incorretas num número bem menor do que as corretas. Caso a aplicação contenha um

limitado paralelismo sobre a quantidade de processadores disponíveis, um elevado número de *rollbacks* é aceitável. Tal situação é oriunda do avanço indevido dos processadores na execução em aplicações nas quais a execução simultânea não poderia ser realizada.

Em relação ao uso de memória, os protocolos otimistas sofrem restrições. Um dos principais problemas é a obrigatoriedade de, periodicamente, realizarem o armazenamento do estado dos processos [FUJ 92]. Esse procedimento pode causar um *overhead* capaz de levar a uma degradação séria de desempenho, tendo as aplicações que necessitam de alocação dinâmica a maior desvantagem. O consumo de memória também é levantado como problema. Os protocolos otimistas podem ser implementados com a mesma memória de algoritmos seqüenciais, porém com baixo desempenho; eles tendem a usar mais memória que os protocolos conservadores.

Outro aspecto a ser levado em conta é a preocupação com possíveis erros arbitrários, os quais podem ser causadores de novas execuções. Tais incorreções computacionais podem entrar em laços infinitos, necessitando de mecanismos de controle para uma intervenção a fim de interromper o sistema. Uma alternativa que existe em quase todos os sistemas é a implantação de uma tarefa para analisar seqüências de execuções incorretas escritas pelo programador. Essa tarefa poderia sempre verificar os índices de acesso a uma matriz em tempo de execução e, explicitamente, testar se um laço está com seu término assegurado.

Os proponentes de métodos conservadores asseguram que os protocolos otimistas são mais complexos de implementar do que os conservadores, particularmente se tentarem encontrar erros arbitrários [FUJ 90]. Um fator que ameniza a complexidade da implementação é a não-obrigatoriedade de conhecimento profundo da aplicação para construir uma eficiente CI.

2.6 Protocolos híbridos

Como tentativa de aproveitar o que há de melhor entre os protocolos conservadores e otimistas, diversos protocolos híbridos foram desenvolvidos. O *Probabilist optimism* foi proposto por Fersha [FER 96] e tem por objetivo explorar o comportamento dinâmico da simulação. Em cada passo, um grau justificável de otimismo é atribuído através de funções de verossimilhança calculadas sobre a probabilidade de que um evento tenha relação com outros. O protocolo, então, decide se um evento deve ser executado de forma otimista (imediatamente) ou conservadora (sendo retardado). Como benefícios, o protocolo apresenta uma forma eficiente de controle do pessimismo (CMB), bem como do otimismo (TW).

O protocolo *Bounded Lag Algorithm with Filtered Rollbacks*, proposto por Lubachevsky, Shwartz e Weiss em 1989, apud ([FER 96], [FUJ 2000]), usa a noção de distância entre processos para determinar se um processo é seguro. Ocasionalmente, pode ser quebrada a restrição de uma distância mínima de segurança, levando a possíveis erros de causalidade. Nessas situações, o protocolo faz uso de um processo de *rollback* para restaurar os estados consistentes [FER 96].

Dickens e Reynolds, apud ([FER 96], [FUJ 2000]), propuseram, em 1988 e 1990, um protocolo híbrido baseado em um algoritmo conservador denominado SRADS (*Shared Resource Algorithm for Distributed Simulation*). Este protocolo sugere a utilização de um protocolo conservador para processar eventos seguros e de outro otimista para processar os demais, sem a capacidade de propagação de subeventos. A justificativa para essa abordagem é o confinamento de possíveis *stragglers* a processos locais, evitando-se a necessidade de geração de antimensagens. O principal problema neste protocolo é o fato de eventos processados de maneira otimista não poderem ser utilizados pelo protocolo de sincronização para a determinação de novos eventos seguros [VAC 99].

2.7 Protocolos otimistas *versus* conservadores

Questões relativas à qualidade dos protocolos otimistas e conservadores são freqüentemente levantadas. Regras gerais de superioridade de um ou de outro mecanismo não podem ser formuladas por causa da grande diversidade dos processos a serem simulados; assim, devem-se evitar comparações diretas entre eles.

O desempenho pode sofrer influências: (1) do *hardware* da implementação (a razão da velocidade das computações); (2) do modelo de comunicação (FIFO, topologia da rede de comunicação, possibilidades operacionais); (3) dos modelos de sincronização (unidade de controle global, variáveis compartilhadas). Essas influências tornam difícil a comparação dos protocolos. Maiores informações podem ser extraídas em [FER 96]. A Tabela 2.2 separa e resume as características dos protocolos sem especificar o melhor ou o pior.

TABELA 2.2 – Características dos protocolos de simulação paralela

	Conservador (CMB)	Otimista (*Time Warp*)
Princípio Operacional	Erros de causalidade são estritamente proibidos. Somente os eventos seguros são processados.	Permite a ocorrência de erros de causalidade, mas detecta-os e recupera-os (imediatamente ou no futuro). Processa bons e maus eventos.
Sincronização	Utiliza um mecanismo de bloqueio, podendo levar o sistema a um *deadlock*. Pode se utilizar de mensagens nulas para sair do *deadlock*, causando um *overhead* de comunicação, ou, ainda, com mecanismos centralizados de detecção e recuperação de *deadlock*.	Utiliza um mecanismo de volta de execuções (dentro do tempo simulado). Deve utilizar mecanismos para aniquilar eventos já processados. Está sujeito a *overhead* de comunicação e efeito dominó.
Exploração do Paralelismo	O paralelismo não pode ser completamente explorado.	O paralelismo pode ser completamente explorado.
Lookahead	Explicitamente necessário para garantir desempenho.	Não necessário, mas útil para a otimização do protocolo.

	Conservador (CMB)	Otimista (*Time Warp*)
GVT	Não exige computações extras para seu cômputo.	Necessita de computações extras. Algoritmos centralizados podem causar gargalos, enquanto que os distribuídos impõem *overhead* de comunicação.
Tamanho dos estados	Enfrenta, arbitrariamente, modelos que consomem grandes quantidades de memória para armazenar seu estado.	Trabalha melhor quando o espaço de memória necessário para armazenar o estado do modelo é pequeno.
Memória	Consumo moderado de memória.	Agressivo consumo de memória, com *overhead* no salvamento de estados.
Mensagens e Comunicação	A ordem de chegada é segundo o *timestamp*, com a execução na mesma ordem sendo obrigatória. Separação dos canais de entrada e estática definição da topologia de comunicação.	Podem chegar fora de ordem cronológica, mas devem ser executadas em ordem. Utilizam uma única fila para as entradas, não necessitando de definição estática da topologia de comunicação.
Implementação	Direta de implementar. Controle e estruturas de dados simples.	Difícil de implementar e depurar. Estruturas de dados simples, porém complexas manipulações de dados e estruturas de controle.

Fonte: FERSCHA. Parallel and Distributed Simulation of Discrete Event Systems.

2.8 Considerações finais

A simulação computacional é uma ferramenta essencial para testar e analisar o desempenho de sistemas do mundo real. Em razão do aumento da complexidade dos sistemas, a simulação paralela acelera o processo e proporciona ganhos em relação ao tamanho do modelo simulado.

Os protocolos conservadores possuem como vantagem, em relação aos otimistas, o fato de serem menos complexos de implementar. Tendo um consumo moderado de memória, adaptam-se melhor a grandes modelos de simulação. Como ponto negativo, possuem um fraco paralelismo de execução, porém, com o correto uso de facilidades como *lookaehad* e janelas de tempo, seu desempenho pode ser melhorado de forma significativa.

Protocolos baseados em TW levam vantagem ao proporcionarem uma melhor exploração do paralelismo dos modelos. Particularidades como a granularidade do

ambiente computacional, topologia e localidade temporal de eventos não necessitam ser conhecidas a fundo para uma eficiente CI. Os pontos negativos traduzem-se na complexidade de implementar, no consumo agressivo de memória e na difícil obtenção do GVT.

3 Programação paralela

Para a implementação dos protocolos de simulação paralela descritos, devem ser utilizadas linguagens de programação ou compiladores especializados que permitam a exploração do paralelismo, além de que se deve efetuar a sincronização entre os processos lógicos de simulação.

Neste capítulo, descrevem-se os ambientes paralelos[1] disponíveis e sua classificação; após, as possibilidades disponíveis para se construir programas que explorem o paralelismo, juntamente com as principais formas de interação entre os processos. Ao final, discute-se a arquitetura baseada em memória compartilhada distribuída, juntamente com suas principais vantagens e desvantagens.

3.1 Introdução

A demanda crescente por processamento tem motivado a evolução dos computadores, viabilizando implementações de aplicações que envolvem uma elevada taxa de computação e grandes volumes de dados. Como tentativa para diminuir o tempo de resposta, têm-se as alternativas de aumentar o desempenho do processador ou de utilizar vários processadores [SAT 96].

Para conseguir um aumento de desempenho do processador, pode-se: (a) aumentar a velocidade do relógio: esta alternativa envolve melhorias na tecnologia de confecção de circuitos integrados, trazendo, entre outros problemas, o aumento de temperatura; (b) melhorias na arquitetura: motivou o surgimento dos processadores RISC, vetoriais e superescalares; (c) melhorias no acesso à memória: por exemplo, através da exploração da hierarquia de memória [SAT 96].

Vários processadores podem também ser utilizados, distribuindo-se entre si a carga de trabalho do programa, configurando um ambiente paralelo, conhecidos como multicomputadores e multiprocessadores. Os multiprocessadores possuem um espaço para endereçamento de memória único e compartilhado, podendo diferir na distribuição física da memória [SIL 99]. Os multicomputadores são computadores, freqüentemente denominados “nós”, interconectados por uma rede. Cada nó é um computador autônomo, consistindo de um processador, memória local e periféricos de I/O [CAL 99].

3.2 Arquiteturas seqüenciais e paralelas

A busca do alto desempenho visando atender à demanda crescente de processamento motivou o surgimento de vários modelos de arquiteturas paralelas. Diversas taxonomias foram propostas, entre elas a elaborada por Michael Flynn em 1972, citada em [CAL 99] e [SIL 99], e a construída por Gordon Bell em 1994, citada em [SAT 96]. As classificações foram elaboradas principalmente segundo o fluxo de instruções e dados nos processadores. Outras diferenças também são encontradas, como organização da memória (SM - *Shared Memory versus* MP - *Messaging Passing*), rede de interconexão

[1] Engloba plataformas de *hardware* e *software*.

e poder de execução de instruções dos processadores ([KUM 94] ,[CUL 99], [HWA 93], [HWA 98]).

Levando em consideração apenas o fluxo de instruções e de dados, tem-se a seguinte taxonomia:

- SISD (*Single Instruction Flow, Single Data Flow*): fluxo único de dados e instruções. Uma aplicação roda sobre o único processador, sendo obtida uma instrução do programa por vez, atuando sobre um único dado cada vez. Estão incluídas nesta categoria os sistemas uniprocessadores;

- MISD (*Multiple Instruction Flow, Single Data Flow*): fluxo múltiplo de instruções sobre um fluxo único de dados. Representa um processamento diversificado simultâneo sobre um mesmo conjunto de dados. Esta classe não possui representação devido a suas características;

- SIMD (*Single Instruction Flow, Multiple Data Flow*): fluxo único de instruções com múltiplo fluxo de dados. Engloba as máquinas que executam, simultaneamente, um mesmo trecho programa sobre conjuntos distintos de dados;

- MIMD (*Multiple Instruction Flow, Multiple Data Flow*): fluxo múltiplo de instruções e dados. Englobam as máquinas que executam porções de código distintas sobre diferentes conjuntos de dados. São o modelo mais geral de paralelismo e bastaste flexíveis, pois aceitam uma grande variedade de paradigmas de programação. Podem ser construídas por conjuntos de máquinas SISD (multicomputadores) ou máquinas com memória compartilhada (multiprocessadores). No segundo caso, dividem-se, ainda, quanto ao acesso à memória, podendo ser uniformes ou não ([ZOM 96], [CAL 99], [SIL 99]).

3.3 Programação paralela

A implementação de algoritmos em ambientes paralelos requisita recursos de programação paralela que permitam expressar o paralelismo e incluir mecanismos para sincronização e comunicação. A programação paralela é a programação concorrente orientada para ambientes paralelos, incorporando os seus requisitos de sincronização e comunicação; busca a utilização adequada dos recursos de processamento para otimizar o seu desempenho [SAT 96].

A programação paralela não é uma tarefa trivial. O programador deve preocupar-se em explorar o paralelismo da aplicação e estabelecer a interação entre os processos. O paradigma de programação paralela determina como essas tarefas são realizadas. Trocas de mensagens e memória compartilhada são os principais paradigmas de programação paralela existentes, diferindo, sobretudo, na forma como os processos interagem ([SAT 96], [SEI 98], [CAL 99]).

Na programação por trocas de mensagens, a interação entre os processos é realizada de forma explícita através de primitivas para envio e recebimento de

mensagens. O paradigma de memória compartilhada assume a existência de um espaço único para endereçamento global. Dessa forma, em SM, a interação entre os processos é realizada através de leituras e escritas em estruturas de dados compartilhadas [SEI 98].

3.3.1 Escalabilidade

Com o objetivo de aumentar o desempenho de um sistema, mais processadores podem ser inseridos. Inicialmente, poder-se-ia pensar que, a cada processador incluído, o desempenho melhoraria em proporção linear com o número de processadores; o sistema teria seu desempenho melhorado em escala com o número de processadores. Desse modo, um sistema escalável seria aquele cujo desempenho variasse linearmente com o custo do sistema. Contudo, formal e realisticamente, pode-se definir um sistema escalável como aquele que suporta uma razão custo desempenho constante [SIL 99]. Convém lembrar que há um limite no qual o aumento do número de processadores não melhora mais o desempenho.

A escalabilidade depende tanto de *hardware* como de *software*. No primeiro caso, o custo de comunicação, determinado principalmente pelas latências da rede de interconexão, pode diminuí-la. O *software* influencia a escalabilidade porque, para obtê-la de forma linear, seriam necessários algoritmos perfeitamente paralelos. Tais programas deveriam manter constante o acréscimo nas comunicações independentemente do número de processares, entretanto nenhum algoritmo enquadra-se totalmente nessa exigência ([RIC 97], [SIL 99]).

Multiprocessadores de memória distribuída são bastante escaláveis e seu modelo de programação é o de memória compartilhada, necessitando de controle explícito do paralelismo [CAL 99]. Outra forma de obter ambientes paralelos escaláveis é o uso de estações de trabalho conectadas a uma rede local. Dessa forma, podem ser utilizadas até algumas dezenas de estações para executar um único trabalho [SAT 96].

3.3.2 Linguagens de programação paralela

Dada a diversidade de arquiteturas paralelas, linguagens, compiladores e bibliotecas especiais foram propostos com o objetivo de gerenciar o paralelismo, que pode ser explorado de forma implícita, através de compiladores paralelizantes, ou explícita, com o uso de diretivas opcionais de compilação ou elementos na própria linguagem. ([SAT 96], [RIC 97]).

A utilização de compiladores que paralelizam automaticamente os programas minimiza as dificuldades da programação paralela, permitindo o reaproveitamento de programas seqüenciais já implementados. Por outro lado, a programação paralela, explicitamente assinalando o paralelismo, proporciona que fontes de paralelismo não passíveis de detecção por um sistema paralelizante possam ser exploradas. Aplicações nas quais formas de paralelismo específicas são requisitadas são de difícil paralelização automática [SAT 96].

Entre as maneiras de explorar o paralelismo em programas, destacam-se:

- linguagem seqüencial com paralelismo implícito: trata-se de linguagens nas quais o compilador gera o código paralelo; a eficiência está relacionada à qualidade do código paralelo gerado. Por mais "inteligente" que seja o compilador, raramente o paralelismo máximo ideal pode ser explorado. *Parafrase2*[2] e *Suif*[3] podem ser considerados exemplos [SAT 96];

- linguagem seqüencial com chamadas de sistema (*system calls*): embora seqüenciais, estas linguagens usam chamadas de sistema para construir e gerenciar o paralelismo. Um exemplo é a utilização da primitiva *fork* em programas escritos em C [CAL 99];

- linguagem seqüencial com chamadas de bibliotecas (*library calls*): estas linguagens usam chamadas de bibliotecas, que, por sua vez, usam *system calls* para explorar o paralelismo. A utilização de dois níveis pode acarretar maior ineficiência. Exemplos de bibliotecas que possuem tais características são PVM[4], *TreadMarks*[5] e *Athapascan0*[6] [SAT 96];

- linguagem paralela: a própria linguagem, ou suas extensões, possui construtores para criar e gerenciar o paralelismo, sendo exemplos *Occam*[7], SR[8] e C/C++ com extensões específicas ([WIL 96], [SIV 94]) , entre outros ([GEY 98], [CAL 99]).

3.4 Programação com trocas de mensagens

Os sistemas de computação paralela que não possuem memória compartilhada necessitam implementar a troca de mensagens e rotinas de sua sincronização para a comunicação entre os processadores [TIB 96]. Na programação com troca de mensagens, os processadores estão ligados a uma interface de conexão que permite o seu envio/recebimento. Cada processador possui sua própria memória local, que só pode ser acessada diretamente por ele [KUM 94]. Além disso, a programação por MP também pode ser implementada e utilizada em multiprocessadores de memória compartilhada.

Uma troca de mensagem envolve, pelo menos, dois processos: o transmissor, que envia a mensagem, e o receptor, que a recebe. Geralmente, isso é feito através de primitivas do tipo *Send* e *Receive* em duas principais modalidades: a comunicação síncrona e a comunicação assíncrona ([SAT 96], [HOA 78]). A Figura 3.1 mostra uma comunicação por trocas de mensagens.

Na comunicação com troca de mensagens síncrona, o transmissor envia a mensagem para o receptor e aguarda até que este sinalize o seu recebimento. Se o receptor não estiver pronto para receber a mensagem, o transmissor é bloqueado

[2] http://www.csrd.uiuc.edu/parafrase2/
[3] http://suif.stanford.edu/
[4] http://www.epm.ornl.gov/pvm/
[5] http://www.cs.rice.edu/~willy/TreadMarks/overview.html
[6] http://www-apache.imag.fr/software/ath0/
[7] http://www.ics.uci.edu/~mlearn/Occam.html
[8] http://www.cs.arizona.edu/sr/www/index.html

temporariamente; caso o receptor esteja pronto para receber a mensagem antes que ela seja enviada, também será bloqueado temporariamente. Após receber a mensagem, o receptor envia de volta um sinal confirmando seu recebimento com o que, imediatamente, os dois são desbloqueados para seguir seu fluxo de execução. Salienta-se que o sinal de confirmação do recebimento é enviado sem que o programador necessite explicitamente fazê-lo, ou seja, ele acontece a cargo do sistema, ocorrendo de maneira implícita ao codificador. Essa modalidade de comunicação não exige a presença de um *buffer* para armazenamento temporário de mensagens.

Na comunicação assíncrona, o transmissor envia a mensagem e prossegue em seu fluxo normal de execução sem sofrer nenhum tipo de bloqueio. Caso o receptor ainda não esteja pronto para receber a mensagem, esta deve permanecer armazenada temporariamente em um *buffer*; se estiver pronto para receber uma mensagem que ainda não foi enviada, deverá permanecer bloqueado temporariamente. Os sistemas que implementam esta modalidade geralmente oferecem primitivas adicionais que verificam se alguma mensagem chegou ou não, sem bloquear o processo receptor ([SAT 96], [GEY 98]).

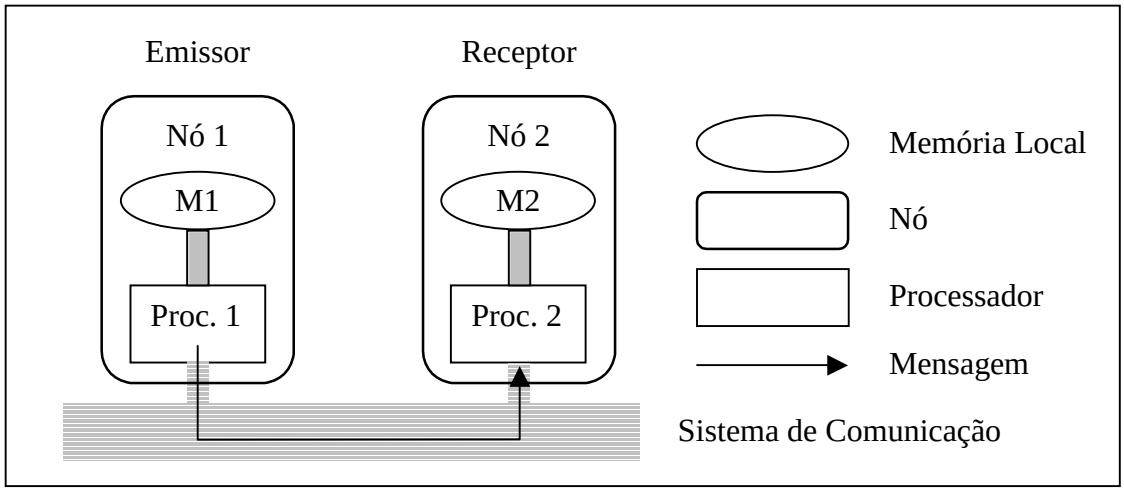


FIGURA 3.1 – Comunicação por trocas de mensagens

A Figura 3.1 exemplifica uma comunicação entre dois processos. O “Emissor” envia uma mensagem para o “Receptor” através do sistema de comunicação. Para essa operação, a primitiva Send é utilizada no primeiro processo, ao passo que o segundo utiliza um Receive.

3.5 Programação com memória compartilhada

Um sistema com memória compartilhada consiste em um grupo de processadores que se comunicam realizando operações sobre variáveis compartilhadas ao invés de enviar e receber mensagens ([KUM 94], [LYN 96]). Este é o modelo usual para programar em máquinas de memória centralizada, o que, necessariamente, não significa que possa ser aplicado somente nesse ambiente. Os processadores comunicam-se da mesma maneira que múltiplas *threads* ou processos o fazem na programação seqüencial, podendo obter ou liberar *locks* e usar semáforos. A Figura 3.2 mostra uma comunicação através de memória compartilhada.

Uma vantagem desse modelo é que o programador não precisa mover dados, facilitando o processo de depuração, pois o dado está imediatamente acessível. A portabilidade é outra vantagem visto que aplicações que utilizam o modelo de memória compartilhada são, a princípio, mais fáceis de serem transportadas para ambientes paralelos. O paradigma de programação de memória compartilhada é considerado mais simples do que MP, uma vez que o programador não precisa ficar controlando a comunicação entre os processos através de mensagens explícitas [CAL 99]. Alguns autores, porém, a consideram menos segura que a programação via trocas de mensagens.

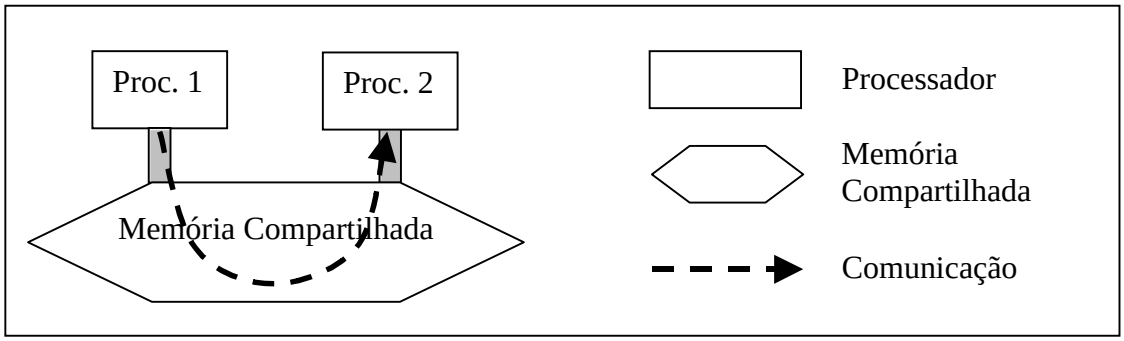


FIGURA 3.2 – Comunicação através de memória compartilhada

Na Figura 3.2, ocorre a comunicação entre dois processadores através de uma memória compartilhada. Um exemplo poderia ser a atualização de uma porção da memória por “Proc.1”, com sua posterior leitura por “Proc. 2”. Nenhuma primitiva adicional é necessária, exceto as de controle de acesso.

Durante muito tempo, o uso do paradigma de programação com memória compartilhada esteve associado a multiprocessadores com memória centralizada. Entretanto, arquiteturas desse tipo apresentam contenção no acesso ao barramento da memória comum, ainda que existam variações que não fazem uso de barramento simples, mas, sim, um conjunto de *switchs*, limitando sua escalabilidade. Sistemas com memória compartilhada distribuída (Figura 3.3), ou DSM (*Distributed Shared Memory*), unem a facilidade de programação do paradigma de SM com a escalabilidade de ambientes distribuídos [SEI 98].

3.6 Memória compartilhada distribuída

Em razão da menor complexidade da programação com o uso de memória compartilhada em relação a modelos com passagem de mensagem, há cerca de uma década e meia, estão em desenvolvimento pesquisas que buscam oferecer mecanismos que suportem variáveis compartilhadas em multicomputadores [ARA 99].

A arquitetura DSM pressupõe a existência de processadores com memórias locais, mas compartilhando um espaço de endereçamento único [SIL 99]. Em sistemas multicomputadores, cópias nas memórias locais dos dados compartilhados permitem que seus acessos sejam efetuados eficientemente. Entretanto, essa abordagem, chamada *caching*, cria o problema de consistência da *cache*, que ocorre quando um processador atualiza dados compartilhados. Como cópias desses dados podem estar presentes em outros nós, estes devem ser mantidos consistentes, não permitindo que um processador

obtenha um valor não atualizado. Para resolver esse problema, modelos de consistência foram propostos.

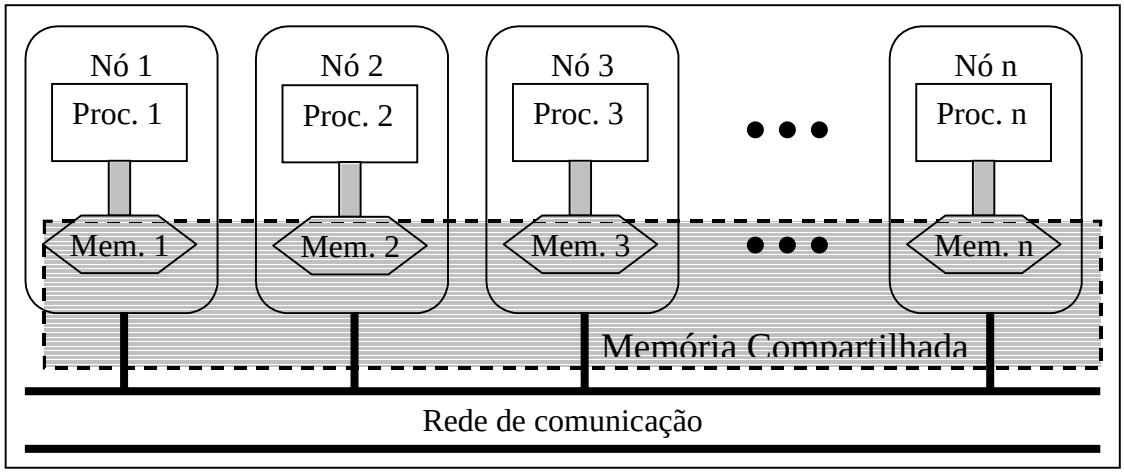


FIGURA 3.3 – Estrutura da memória compartilhada distribuída

Três abordagens têm sido utilizadas na implementação de sistemas DSM: (a) implementação por *hardware*: estendem técnicas tradicionais de *caching* para arquiteturas escaláveis; (b) implementação de bibliotecas pelo sistema operacional: o compartilhamento e a coerência são obtidos através de mecanismos do gerenciamento de memória virtual; (c) implementação pelo compilador e bibliotecas: acessos compartilhados são automaticamente convertidos em primitivas de coerência e sincronização ([SAT 96], [SIL 99]).

3.6.1 Protocolos de coerência

Levando em conta que muitos programas paralelos definem seus próprios requisitos de consistência de mais alto nível, requisitos de consistência de memória podem ser relaxados. Considerando a visão para o programador, as semânticas de coerência de memória podem ser apresentadas em modelos de consistência. Para a construção correta de um programa em um sistema com memória compartilhada distribuída, o programador deve conhecer como as atualizações são propagadas no sistema ([ARA 99] [ADV 99]).

Os modelos de consistência propostos e apresentados na literatura podem ser listados como segue:

- consistência estrita: qualquer leitura em uma posição de memória “x” retorna o valor mais recentemente escrito em “x”;

- consistência seqüencial: o resultado de uma execução é semelhante ao obtido pelo entrelaçamento de operações dos nós individuais quando executado em uma máquina seqüencial *multithreaded*. Os acessos devem ser ordenados considerando os acessos de todos os processadores;

- consistência causal: escritas que são relacionadas com outras devem ser vistas em todos os processos na mesma ordem. Escritas concorrentes podem ser vistas com diferentes ordens em processadores distintos;

- consistência de processador: escritas de um único processador são recebidas em ordem nos demais processadores; já escritas de diferentes processadores podem ser vistas em diferente ordem por outros processadores;

- consistência fraca: os acessos aos dados são tratados separadamente dos acessos de sincronização, mas requerem que todos os acessos aos dados anteriores sejam feitos antes que o acesso de uma sincronização seja obtido. Cargas e armazenagens entre acessos de sincronizações são livres de ordenação;

- consistência *release*: é uma consistência fraca com dois tipos de operadores: *acquire* e *release*. Um operador *acquire* é usado no início de uma seção crítica, para adquirir o direito exclusivo à sua execução, e o operador *release* para liberá-la e exportar os dados atualizados;

- consistência *lazy release*: é um tipo de consistência *release* que busca reduzir o número de mensagens e a quantidade de dados exportados por acessos remotos. Neste modelo, as modificações são exportadas apenas quando ocorre um acesso aos dados, através do operador *acquire*. Essa consistência garante apenas que um processador veja todas as modificações que precedem o acesso *acquire*. Uma modificação precede um *acquire* se ocorre antes de algum *release*, de modo que exista uma cadeia de operações *release-acquire* sobre a mesma variável de sincronização *lock*, terminando no *acquire* corrente;

- consistência *entry*: é utilizada a relação entre variáveis de sincronização específicas que protegem as seções críticas e os acessos aos dados compartilhados nelas efetuados. Uma seção crítica é delimitada por um par de acessos de sincronização a uma variável de sincronização “s”. Um acesso *acquire* a “s” é usado para obter o acesso a um conjunto de dados compartilhados, obtendo os dados compartilhados consistentes. Um acesso *release* à variável “s” no final da seção crítica providencia a sua liberação. São permitidos múltiplos acessos a dados compartilhados para leitura, através dos acessos de sincronização, que podem ser especificados como exclusivos ou não-exclusivos.

A semântica para coerência de memória mais intuitiva é a apresentada pela consistência estrita (*strict consistency*), na qual uma leitura retorna o valor mais recente do dado. Como “o mais recente” é um conceito ambíguo em um sistema distribuído, alguns sistemas DSM providenciam uma forma reduzida de coerência de memória ([TAN 95], [SAT 96]).

Modelos com consistência forte, porém menos restritivos que a estrita, podem ser utilizados, variando a forma como são implementados e o seu desempenho, porém o problema desses é o fraco desempenho. Para melhorá-lo, modelos de coerência fraca podem ser a solução, entretanto, se a aplicação necessita de uma forma de coerência

forte, não deve utilizar os modelos que oferecem coerência fraca. Trabalhos com o objetivo de reduzir o *overhead* das barreiras de sincronização e melhorar mecanismos de coerência fraca podem ser encontrados em ([MIL 96], [KON 97], [LEE 98], [SEI 98]).

3.6.2 Vantagens e desvantagens da DSM

Uma vantagem do modelo baseado em DSM em relação ao de trocas de mensagens é a forma lógica da programação, simples e já dominada pelo programador. O protocolo de acesso usado é consistente com o modo como as aplicações seqüenciais acessam os dados, permitindo uma transição natural para aplicações distribuídas. Em princípio, aplicações paralelas escritas para multiprocessadores podem ser executadas em ambientes de DSM sem alterações.

O sistema de DSM esconde o mecanismo de comunicação dos processos e permite que estruturas complexas possam ser passadas por referência, simplificando a programação. Por essa razão, o código de aplicações paralelas escritas para ambientes com memória compartilhada distribuída é menor e mais compreensível que os equivalentes com trocas de mensagens [FEL 94].

Uma grande preocupação dos modelos de programação baseados em memória compartilhada é a sincronização entre processadores no momento de acessos a variáveis compartilhadas, a qual pode ser efetuada usando, por exemplo, *locks* ou barreiras. A programação com MP também apresenta a necessidade de sincronização, a qual, contudo, é atingida implicitamente através das primitivas de programação [SEI 98].

Outro ponto negativo dos modelos com DSM é o desempenho. Em ambientes distribuídos, com cada processador tendo acesso somente à sua memória local e a DSM sendo implementada através de *software* (ambientes de estações de trabalho ligadas em rede), a comunicação, de uma forma ou outra, ocorre através de trocas de mensagens. Os *softwares* que proporcionam a SM apenas escondem este mecanismo do programador. Dessa forma, o uso de, no mínimo, duas camadas, deteriora o desempenho. Estudos realizados por Lu ([LU 95], [LU 97]) e Rodman [ROD 99] mostram claramente as diferenças entre MP e DSM.

3.7 Considerações finais

O uso de uma rede com computadores seguindo a arquitetura MIMD é uma solução flexível para o processamento paralelo. Multicomputadores são facilmente encontrados pela simples disposição de máquinas SISD trabalhando em conjunto. Por outro lado, essa flexibilidade adiciona uma complexidade extra aos programas paralelos uma vez que passa a existir a necessidade de codificar explicitamente a sincronização entre os processadores.

Os modelos de programação baseados em variáveis compartilhadas permitem implementações com menor complexidade em relação aos modelos com trocas de mensagem. Entretanto, as leituras e escritas dessas variáveis devem ser feitas considerando algumas restrições. Controles de acesso a uma região crítica envolvendo

variáveis compartilhadas devem ser implementados. Um exemplo são os mecanismos que implementam exclusão mútua, garantindo que uma seqüência de comandos seja executada exclusivamente por um processo.

A programação de arquiteturas MIMD usando variáveis compartilhadas através da técnica de memória compartilhada distribuída proporciona uma transição menos abrupta da programação seqüencial para a paralela. Da mesma forma que os ambientes paralelos baseados em multicomputadores, as técnicas que proporcionam uma DSM implementada via *software* são mais facilmente encontradas, porém desafios em relação ao desempenho, como a quantidade de comunicação para manter a coerência dos dados, ainda estão em aberto.

4 Protocolo de simulação paralela com variáveis compartilhadas

Apresentaram-se, anteriormente, diversos protocolos de simulação paralela, alguns dos quais utilizam estratégias otimistas e outros estratégias conservadoras no avanço do tempo local de simulação. Em um ponto, porém, eles convergem: utilizam, indistintamente, a comunicação através de trocas de mensagens.

Neste capítulo, apresenta-se um novo protocolo para simulação paralela, adotando para a sincronização o paradigma de memória compartilhada, baseado no funcionamento dos protocolos conservadores de execução síncrona e utilizando diversas das suas características. Inicialmente, relacionam-se as características herdadas de outros protocolos, assim como as mudanças efetuadas; após, descreve-se seu funcionamento, bem como as principais estruturas de dados; ao final, mostram-se os resultados passíveis de serem extraídos.

4.1 Introdução

A simulação paralela vem sendo usada nas mais diversas áreas há mais de duas décadas, tendo como principais objetivos obter ganhos em desempenho e aumentar a precisão dos modelos simulados. Ganhos em desempenho são originados pela redução do tempo de simulação, o que pode ser obtido com vários processadores executando o processo de simulação. A maior precisão dos modelos é alcançada com maior quantidade de memória disponível para a simulação, proporcionando modelos mais completos, que se aproximam do que representam na realidade.

Encontram-se diversos trabalhos sobre aplicações de simulação paralela, nos quais se utilizam tanto protocolos conservadores como otimistas. Exemplos da utilização de protocolos otimistas podem ser encontrados, por exemplo, em ([DAS 94], [PEN 98]). A primeira referência apresenta o simulador GTW (*Georgia Tech Time Warp*) versão 2.0, desenvolvido pelo *Georgia Institute of Technology*, que utiliza o protocolo *Time Warp* como base de seu mecanismo de sincronização. Na segunda, foi apresentado o *kernel* de simulação intitulado Warped, desenvolvido pela Universidade de Cincinatti, juntamente com sua especificação formal.

Aplicações envolvendo protocolos conservadores também podem ser encontradas em ([KEL 96], [POR 98]). Na primeira referência, os autores utilizam a simulação paralela conservadora para avaliar circuitos lógicos de multiprocessadores com memória compartilhada; na segunda, uma variação do protocolo CMB é utilizada na simulação de uma rede de telefonia celular. Uma simulação utilizando um protocolo conservador e variáveis compartilhadas pode ser encontrada em [MAC 96], o qual simula bolas de bilhar sobre uma mesa, dividindo-a em regiões, cada uma delas simulada independentemente em um processador. Os limites de cada uma das regiões e a área que os cercam são chamados de regiões críticas, onde são utilizadas variáveis compartilhadas para auxiliar a simulação, fazendo uso de um algoritmo para serialização dos eventos. Um problema é que o algoritmo de simulação empregado permite a ocorrência de erros LCC.

Com o passar do tempo, novos ambientes computacionais têm sido utilizados em processos de simulação paralela. O uso de vários computadores ligados através de uma rede de alta velocidade proporciona um novo e escalável ambiente para aplicações paralelas [GEO 99]. Com o uso de *clusters*, estão sendo criados novos protocolos baseados em outros já existentes, buscando melhor desempenho, melhor modelagem e maior facilidade de programação. CongDuc Pham elaborou um estudo sobre simulação paralela de eventos discretos sobre *clusters* de alto desempenho, considerando o ambiente como muito promissor para a execução de grandes simulações [PHA 99a]. Os possíveis impactos do ambiente distribuído para aplicações de simulação paralela podem ser encontrados em [IKO 98], que aborda questões relativas à simulação distribuída, como *software*, S.O. (Sistema Operacional), *hardware* e redes de conexão. Na sua execução foi utilizada uma variação do protocolo CMB, que reduz o número de canais lógicos entre os LPs, proporcionando um melhor desempenho.

A diversidade de aplicações, ambientes e protocolos utilizados na simulação paralela é bastante grande, tendo sido usada praticamente em todas as áreas. Todos os protocolos de simulação paralela citados usam trocas de mensagens para comunicação entre os LPs.

A programação no paradigma de memória compartilhada é considerada mais simples por evitar que o programador tenha de se preocupar com a comunicação entre os processos através da troca explícita de mensagens. O objetivo principal dos mecanismos DSM é ocultar a comunicação do programador e fornecer um modelo de programação baseado em dados compartilhados ao invés de trocas de mensagens. Um fator negativo é que os mecanismos que fornecem um ambiente DSM, tanto em *hardware* como em *software*, ainda necessitam de desenvolvimento para que tenham um desempenho semelhante ao das aplicações baseadas em trocas de mensagens [ARA 99].

O protocolo de simulação paralela a ser descrito neste capítulo faz uso de variáveis compartilhadas, as quais são utilizadas para construir uma Interface de Comunicação de fácil programação, buscando um desempenho satisfatório. O protocolo pode também ser utilizado em ambientes multicomputadores, através de mecanismos de DSM, com o que se aliam as vantagens da memória compartilhada com os benefícios dos sistemas distribuídos.

4.2 Princípios do protocolo

Os protocolos de simulação paralela usam trocas de mensagens para a comunicação entre os LPs. Nesse contexto, dois tipos de mensagens podem ser encontradas: as oriundas do mecanismo de sincronização e as provenientes da manipulação de variáveis que são necessárias em vários LPs ([REB 99b], [FER 96]). No primeiro caso, as mensagens podem ser encaradas como sinais que indicam o avanço de cada LP; no segundo, elas são utilizadas na manipulação de dados acessíveis por diversos LPs, emulando um ambiente de informações compartilhadas. No caso, o mundo real seria representado com maior naturalidade se fossem utilizadas técnicas de variáveis compartilhadas [HIR 97], as quais proveriam um modelo mais fiel ao mundo real. Essa qualidade, a velocidade e flexibilidade constituem as principais características de um bom modelo de simulação paralela [RIC 95].

O conceito de variáveis compartilhadas é muito semelhante à lógica empregada nos protocolos conservadores de simulação, nos quais os acessos devem ser realizados em ordem a fim de não incorrer em erros LCC, com o que não haveria a possibilidade de nova execução de ações. Algoritmos para simulação paralela utilizando variáveis compartilhadas foram propostos em [MEH 93], porém apenas conceitos gerais foram abordados, como controlador central de variáveis e técnicas para replicação, sendo aplicados em protocolos conservadores e otimistas.

O protocolo de simulação proposto neste trabalho utiliza características dos protocolos conservadores de simulação, mas apresenta modificações em alguns pontos, como no gerenciamento da memória, no avanço do tempo local de simulação e na sincronização dos processos lógicos. Sua funcionalidade assemelha-se muito à dos protocolos conservadores com execução síncrona.

4.2.1 Objetivos

O objetivo deste trabalho é a construção de um protocolo de simulação paralela utilizando variáveis compartilhadas em ambiente de memória distribuída. As técnicas DSM, além de proporcionarem uma programação mais próxima do mundo real, podem ser de extrema utilidade na construção de um protocolo de simulação paralela baseado nas técnicas conservadoras.

O protocolo de simulação descrito não leva em conta particularidades de aplicações específicas, mas tenta criar um ambiente parametrizável que possa servir a diversos tipos de aplicações. A ênfase é dada à “Interface de Comunicação - CI”, parte indispensável a todos os protocolos de simulação paralela. A CI foi construída usando variáveis compartilhadas como forma de comunicação entre os LPs.

A Figura 4.1 mostra o escopo deste trabalho, que é a construção de uma interface de comunicação utilizando variáveis compartilhadas em memória distribuída. As regiões sombreadas indicam em que partes do modelo de simulação paralela tradicional mais se concentraram os estudos.

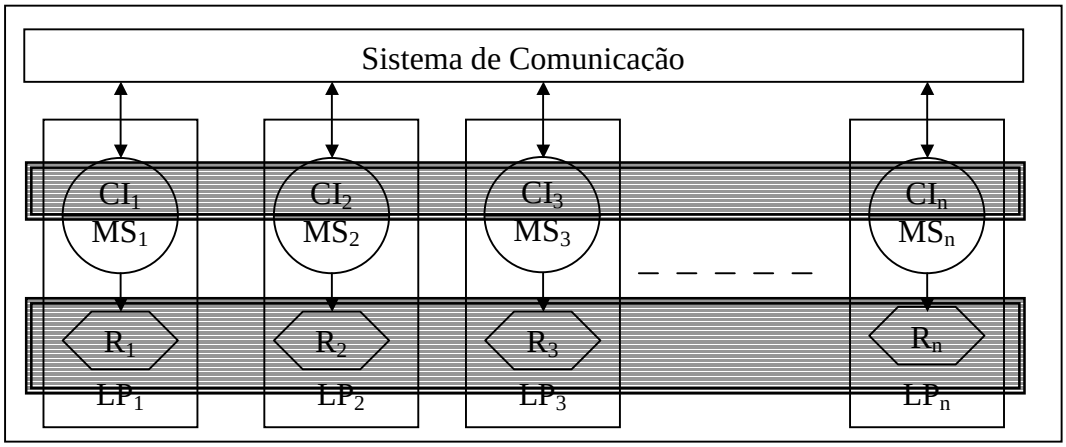


FIGURA 4.1 – Escopo do trabalho

No contexto da figura anterior, destacam-se como principais objetivos do modelo de simulação com variáveis compartilhadas:

a) estabelecer um algoritmo para sincronização dos LPs que resolva de maneira implícita os problemas relacionados ao *deadlock*;

b) manter, durante o maior tempo possível, os processadores trabalhando na simulação;

c) diminuir o tempo ocioso dos protocolos conservadores;

d) tentar diminuir o tempo gasto nos processos de simulação paralela.

Os objetivos que não se encaixam de forma direta na Interface de Comunicação podem ser alcançados de forma indireta. Com a construção de uma CI eficiente, maior tempo de processamento poderá estar disponível para a MS, podendo diminuir o tempo total da simulação.

4.2.2 Conceitos utilizados

Considerado como conservador, o protocolo a ser apresentado traz os benefícios e problemas de seus antecessores. Um dos principais problemas dos protocolos conservadores diz respeito ao seu baixo desempenho, ocasionado pelo fraco paralelismo, se comparados aos protocolos otimistas ([FUJ 90], [FER 96]). Porém, testes descritos em ([RIC 95], [POR 97a], [IKO 98]) informam um bom desempenho das técnicas conservadoras, em alguns casos até suplantando os protocolos otimistas. Ressalta-se que estes trabalhos foram realizados sobre aplicações específicas, as quais obtiveram bom desempenho.

O fraco paralelismo dos protocolos conservadores, evidenciado como problema desde a sua concepção, foi amenizado com a utilização de conceitos como *lookahead* e janelas de tempo. O protocolo a ser descrito neste capítulo faz uso desses componentes, característicos dos algoritmos de simulação paralela conservadora de execução síncrona, com o objetivo de melhorar o desempenho.

As vantagens dos protocolos CMB, como maior facilidade na implementação, baixo consumo de memória e fácil obtenção do GVT, são mantidas. Ao consumo moderado de memória, acrescenta-se o benefício de se adaptarem melhor às grandes simulações. Ainda que haja uma extensa lista das características dos protocolos conservadores utilizadas, mostram-se na Tabela 4.1 apenas as principais.

TABELA 4.1 – Características dos protocolos conservadores utilizadas

Conceito	Descrição
Princípio operacional	Não-ocorrência de LCC
Sincronização dos LPs	A cada barreira definida pela janela de tempo
Deadlock	Não ocorre
GVT	Obtido de forma direta
Lookahead	Utilizado de forma constante
Número de processadores	Fixado ao início do processo
Estrutura do LP	Mantém a estrutura de dois subsistemas

4.2.3 Modificações realizadas

O protocolo de simulação com variáveis compartilhadas apresenta diversas modificações em relação ao CMB inicialmente proposto, algumas das quais não são exclusividade sua visto que, ao longo do tempo, novas variações foram introduzidas nos protocolos CMB. Ele apresenta uma mudança em nível conceitual, ocasionando, dessa forma, diversas outras no funcionamento do processo de simulação, tanto em nível de sincronização dos LPs quanto no agendamento de novos eventos.

A principal modificação do protocolo ocorre em nível conceitual, proporcionada pelo acesso direto de um LP à parte da memória local pertencente a outro LP. Uma das características de ambos os grupos de protocolos de simulação paralela originalmente descritos é que um LP somente podia acessar diretamente sua memória local [FUJ 2000]; logo, o acesso a memórias não locais tinha de ser efetuado através de trocas de mensagens. A Figura 4.2 apresenta esta modificação.

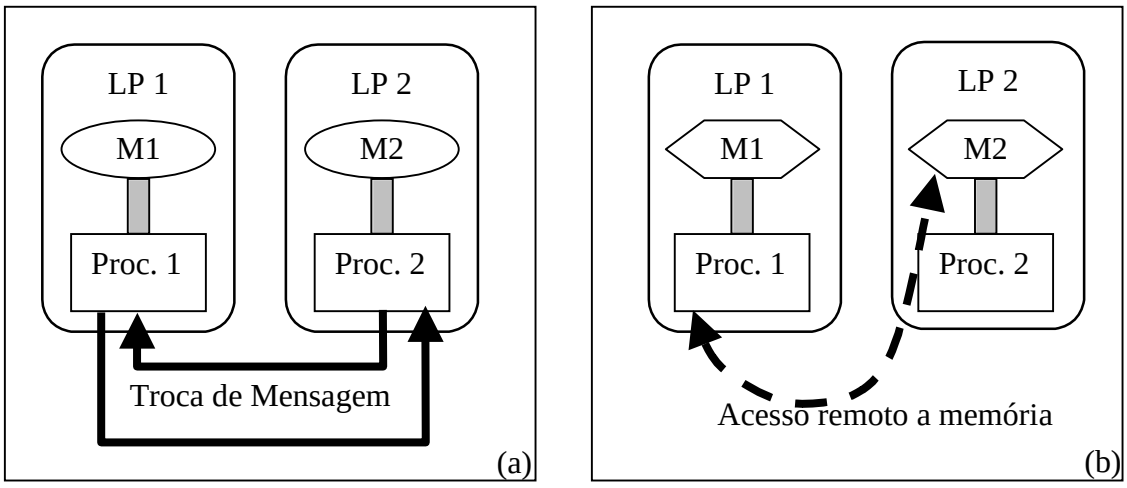


FIGURA 4.2 – Comunicações entre 2 LPs no modelo com DSM

A Figura 4.2a indica como, tradicionalmente, é realizado o acesso à memória de outro LP, havendo uma troca de mensagens entre os processos envolvidos. Já a

Figura 4.2b mostra o acesso à memória utilizado neste protocolo, realizado de forma remota. Assim, LP_1 pode remotamente acessar a memória compartilhada de LP_2 sem necessitar envolver LP_2 em processos de trocas de mensagens.

Cabe salientar que nem toda a memória local dos LPs necessita ser acessada pelos demais, apenas uma parte dela. Desse modo, havendo somente algumas regiões de memória compartilhadas, todo um novo mecanismo de sincronização pode ser construído. Com a possibilidade de um LP acessar remotamente parte da memória local de outro, diversas mudanças podem ser realizadas no protocolo de simulação paralela, entre elas:

- tratamento do *deadlock*: o novo protocolo, virtualmente, nunca entra em *deadlock* visto que há a prevenção para que tal não aconteça. O mecanismo de prevenção utiliza-se da memória compartilhada nos diversos LPs para seu funcionamento;

- avanço do tempo local de simulação: por assemelhar-se a um protocolo de execução síncrona, o avanço do LVT pode ocorrer de forma livre até a chegada a uma barreira de sincronização. Dessa forma, a utilização de variáveis compartilhadas nos LPs proporciona um avanço no tempo de simulação local;

- mecanismo de sincronização: a sincronização é efetuada usando estruturas de dados compartilhadas nos LPs;

- agendamento de novos eventos oriundos de outros LPs: é realizado quando o LP chega a uma barreira de sincronização. Nesse momento, pode verificar se existem novos eventos para ele em porções de memória compartilhada, sem necessitar esperar por mensagens.

4.3 Gerador de listas de eventos

Os processos de simulação de eventos, tanto seqüenciais como paralelos, utilizam uma lista ordenada para armazenar os eventos que compõem a simulação, a qual é mantida em ordem crescente de tempo de execução. Tais listas são oriundas das aplicações simuladas. Todavia, no protocolo de simulação com uso de variáveis compartilhadas descrito neste capítulo, nenhuma porção do mundo real está sendo simulada; assim, a lista de eventos deve ser criada para que o processo de simulação aconteça.

Dessa forma, um gerador de listas de eventos se faz necessário para criar uma EVL a ser executada. As seqüências de eventos geradas são armazenadas em memória estável para que possam serem reutilizadas em outros processos de simulação e, eventualmente, em outros protocolos de simulação para comparações.

O protocolo descrito é utilizado para simulação paralela de eventos; portanto, a lista de eventos deve ser dividida para que porções suas possam ser executadas em diversos processadores. O processo de divisão da lista e direcionamento para um ou outro LP, conhecido também como balanceamento de carga inicial, deve ser realizado.

No caso do presente protocolo, essa divisão é realizada tomando como base apenas o número de processadores disponíveis. A Figura 4.3 ilustra uma divisão de eventos.

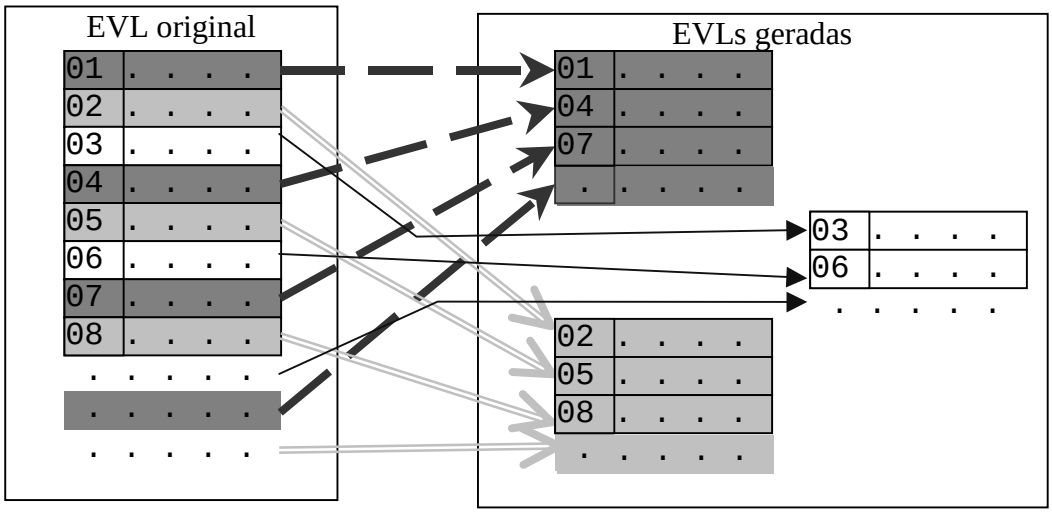


FIGURA 4.3 – Estratégia de divisão da EVL

A política de divisão empregada pode ser visualizada na Figura 4.3, que ilustra parte da divisão de uma lista de eventos em três outras. É conveniente salientar que o número dos eventos não é o *timestamp* desses, mas apenas uma numeração seqüencial.

Nenhuma computação é realizada a fim de analisar a melhor ou a pior estratégia para divisão, decisão que se deveu especialmente a dois motivos:

- como a simulação não representa nenhuma aplicação do mundo real, não há possibilidade de analisar dependências de dados e seqüências de eventos com relação causa-efeito;

- a área de balanceamento de carga é ampla e complexa, exigindo outro trabalho para seu aprofundamento, o que não é objeto do presente.

4.3.1 Parâmetros de entrada

Para que as listas de eventos possam ser geradas, parâmetros de entrada devem ser informados, os quais possibilitam diversos testes, com várias listas representando múltiplas simulações. Esses parâmetros são utilizados para que as listas possam assemelhar-se ao máximo com simulações reais. Os parâmetros que devem ser informados para a geração de uma lista são:

- nome do arquivo: nome dado à estrutura de memória estável que irá armazenar a seqüência de eventos;

- quantidade de eventos: indica o número de eventos que deverão ser gerados;

- incremento mínimo de *timestamp*: informa a diferença mínima de tempo para execução que um evento pode ter em relação ao seu subseqüente;

- incremento máximo de *timestamp*: indica a diferença máxima de tempo para execução que um evento pode ter em relação ao próximo evento;

- tempo mínimo de execução: valor mínimo de tempo de execução utilizado por um evento;

- tempo máximo de execução: valor máximo de tempo de execução utilizado por um evento;

- percentual de agendamento de eventos na própria EVL: probabilidade, em percentual, de cada evento agendar um novo em sua própria lista de eventos;

- percentual de agendamento de eventos em outra EVL: probabilidade, em percentual, de cada evento agendar um novo em uma lista que não seja a sua.

Na criação de um evento, é gerado um número, que obedece aos valores mínimos e máximos de tempo de execução, para ser o tempo gasto pelo evento. Em seguida, é gerado o dado que indicará se o evento irá ou não agendar um novo em sua EVL, obedecendo à probabilidade informada. Caso não seja criado evento na própria lista, é verificado se o evento não irá gerar um novo em outra EVL, obedecendo também a sua probabilidade. Se o evento que está sendo gerado criar um novo, será sorteado um tempo para a execução do novo evento dentro da faixa de valores informados para tempo de execução. Por último, é gerado um número, obedecendo ao mínimo e ao máximo incremento de *timestamp*, o qual será a diferença de tempo entre o evento atual e o próximo.

Todas as gerações de valores utilizadas são aleatórias obedecendo uma distribuição uniforme dos mesmos. Cada evento somente pode criar um novo em sua própria EVL ou em outra; somente eventos oriundos do gerador de listas terão a possibilidade de criar outros. Essa decisão foi tomada para padronizar a forma dos eventos gerados. Como ponto inicial para o tempo em que os eventos deverão ocorrer, foi utilizado o tempo um (1).

Após a geração do primeiro evento, o processo descrito se repete. Dessa forma, caso o incremento de *timestamp* gerado para o próximo evento seja zero (0), o tempo em que esse deverá ser executado será o mesmo do evento anterior. A Tabela 4.2 ilustra a geração do *timestamp* de eventos, utilizando valores de incremento do mesmo na faixa de [0..2].

TABELA 4.2 –*Timestamp* dos eventos obtidos do gerador de listas

Timestamp atual	Incremento gerado	*Timestamp* do próximo evento
1	0	1
1	2	3
3	1	4
4	1	5
5	0	5
5	2	7
7	1	8

Com os dados da Tabela 4.2, pode-se observar que, em alguns tempos, não há ocorrência de eventos; de forma semelhante, em alguns tempos, deve acontecer a execução de dois eventos simultaneamente.

4.3.2 Arquivos gerados

Diversos arquivos, contendo cada um uma lista de eventos, podem ser criados. Por mais que divirjam os valores de tempo de execução e criação ou não dos novos eventos, todos os arquivos mantêm uma mesma estrutura, contendo para todos os eventos da lista as mesmas informações.

Os arquivos são armazenados em formato texto, tendo cada informação gerada separada das outras por sinais de tabulação e um caracter "|" (*pipe*). Cada evento distinto é armazenado em um única linha. A Figura 4.4 mostra um arquivo com dez eventos gerados, com tempo de execução variando entre cinco e dez ([5..10]). O percentual de agendamento de novos eventos na própria lista e em outra é de 30% e o incremento de *timestamp* localiza-se na faixa entre zero e um ([0..1]).

```
1     1     |     9     |     N     |     S     |     5
2     2     |     5     |     N     |     N     |     0
3     2     |     7     |     N     |     N     |     0
4     2     |     8     |     N     |     N     |     0
5     3     |     6     |     S     |     N     |     8
6     4     |     8     |     N     |     N     |     0
7     5     |     8     |     S     |     N     |     7
8     6     |     9     |     N     |     N     |     0
9     7     |     9     |     S     |     N     |     9
10    7     |     5     |     S     |     N     |     9
```

FIGURA 4.4 – Lista de eventos em memória estável

O primeiro valor de cada linha não diz respeito à simulação, sendo apenas um contador de eventos gerado para controle; o segundo representa o tempo em que o evento deverá ocorrer; logo após, encontra-se a quantidade de tempo de execução gasta pelo evento. Os dois campos seguintes dizem respeito ao agendamento de novos eventos, o primeiro indicando que o novo evento deverá ser inserido na própria lista e o segundo em outra lista; o último campo, indica a quantidade de tempo de execução que o novo evento, se existir, irá consumir.

Para a execução de um processo de simulação, o nome de um arquivo de lista de eventos deve ser informado. Como a simulação é paralela, esse nome serve apenas como base para que os diversos arquivos gerados na divisão da EVL possam ser carregados. Dentro dos simuladores utilizados neste trabalho, uma estrutura de memória é utilizada para o processamento dos eventos, podendo ser visualizada na Figura 4.5.

```
estrutura eventos
Inicio
   tempo_evento : inteiro; { Tempo em que o evento será executado    }
   consumo      : inteiro; { Quantia de tempo despendida pelo evento }
   novo_agenda  : inteiro; { Se ira criar um novo                    }
   novo_consumo : inteiro; { Tempo consumido pelo novo evento        }
Fim.
```

FIGURA 4.5 – Estrutura para carga da lista de eventos

A estrutura descrita na Figura 4.5 indica os atributos de cada evento. Uma lista de variáveis com essa estrutura deve ser implementada para armazenar todos os eventos a serem processados, os quais devem estar ordenados pelo "`tempo do evento`", devendo as inserções realizadas por novos eventos agendados mantê-la ordenada.

4.4 Estrutura geral da memória

O protocolo descrito neste capítulo faz uso de variáveis compartilhadas para seu funcionamento, porém nem toda a memória dos LPs necessita ser compartilhada. Grande parte da memória dos LPs é utilizada com a manipulação de sua EVL, que não necessita ser compartilhada. A parte que necessita estar acessível a outros LPs são as informações relativas à sincronização dos processos.

Encontram-se diversos mecanismos de sincronização baseados em trocas de mensagens. No conteúdo das mensagens, encontram-se informações relativas ao avanço do tempo local de simulação e agendamento de novos eventos nos outros LPs. Outros tipos de informações também podem ser encontrados, como mensagens nulas (T_{null}) e antimensagens (m-), as quais, porém, também se referem, de uma maneira ou outra, ao avanço do LVT.

A fim de reduzir o tempo de ociosidade dos processadores, apenas o LVT de cada LP necessitaria ser compartilhado. Dessa forma, quando um LP alcançasse uma barreira de sincronização, poderia "espiar" os demais LVTs e atualizar o seu; porém, o LP não poderia seguir seu processamento sem as informações relativas aos eventos a serem nele criados por outros LPs. Assim, não apenas o LVT necessita ser compartilhado, como também regiões de memória compartilhada devem ser fornecidas para o controle da criação de novos eventos oriundos de outros LPs.

Portanto, são utilizadas estruturas de memória compartilhada para controlar o LVT e a criação de novos eventos. Porém, para o uso dessas variáveis, outras de acesso local são necessárias a fim de que os mecanismos descritos possam ser corretamente implementados. A Figura 4.6 apresenta um esquema das principais estruturas de dados utilizadas pelo modelo.

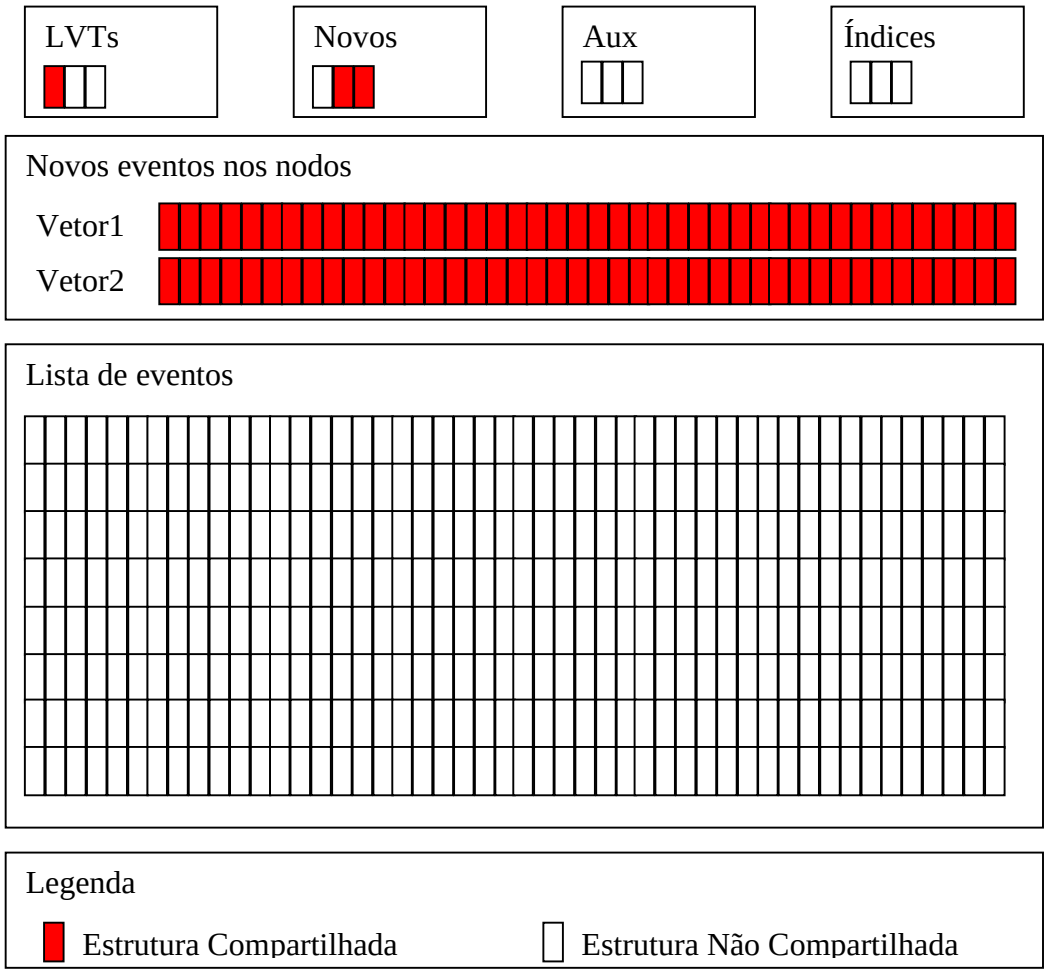


FIGURA 4.6 – Estrutura de memória de um LP

A Figura 4.6 apresenta a fotografia da memória de um LP do protocolo descrito, a qual foi extraída de LP_0 numa simulação envolvendo três LPs. Assim, as primeiras estruturas estão representadas com três posições. Ressalta-se que, no protocolo, não há limite em relação ao número de processadores.

A estrutura “LVTs” é a responsável pelo avanço da simulação nos LPs. Os vetores “Novos” e “Índices” são responsáveis pelo correto armazenamento das informações relativas à criação de novos eventos em outros nodos. A área de memória denominada “Aux” auxilia na leitura remota de eventos contidos nas estruturas “Vetor 1” e “Vetor 2”. A região identificada por “Lista de Eventos” armazena a EVL do LP.

4.4.1 Estruturas de dados compartilhadas

Antes de detalhar as estruturas de dados compartilhadas, descrevem-se as formas dos compartilhamentos, juntamente com os tipos de acessos possíveis. Na Figura 4.6, encontram-se diversas estruturas de dados compartilhadas, algumas com compartilhamento total e outras com parcial.

Os tipos de acessos à memória compartilhada previstos são de leitura e escrita para o LP que contém, fisicamente, a região compartilhada e apenas de leitura para os demais. Dessa forma, o funcionamento ocorre com o LP responsável pela variável

compartilhada fornecendo os valores que os demais LPs podem ler. A escolha de acessos remotos à memória somente de leitura justifica-se para não onerar o protocolo com acessos remotos de escrita, que necessitam de maior controle para serem realizados e, conseqüentemente, de maior *overhead* de processamento.

Duas formas de compartilhamentos são permitidas: com acesso a todos os LPs e com acesso a um só. Para exemplificar essa divisão, tomam-se por base, em um LP qualquer, seu LVT e a quantidade de eventos a serem criados em LP_x. No primeiro caso, todos os LPs devem enxergar o valor; já, no segundo, somente LP_x necessita acessar remotamente a variável.

A estrutura descrita como "**LVTs**" é um vetor de inteiros que armazena o LVT de cada um dos nodos. Apenas a posição relativa ao número do nodo em questão necessita ser compartilhada, sendo visível a todos os LPs. No exemplo da Figura 4.6, apenas a posição zero do vetor é compartilhada; as demais posições armazenam o LVT de outros nodos, atualizados por acessos remotos e utilizados para o controle do avanço local da simulação. A manutenção em apenas uma estrutura de dados dos LVTs da simulação proporciona o cálculo do GVT de forma facilitada: ele será o menor valor do vetor em questão.

A estrutura descrita como "**Novos**" também é um vetor de inteiros, que armazena a quantidade de novos eventos que um LP irá criar em outro dentro de uma barreira de sincronização. Diferentemente de "LVTs", suas porções compartilhadas são aquelas que não possuem correspondência com o nodo atual. Cada posição desse vetor somente é acessível de forma remota pelo nodo correspondente a sua posição, ou seja, a posição de número um somente poderá se acessada remotamente por LP_1.

As estruturas de dados descritas em "**Novos eventos nos nodos**" são vetores de inteiros que trabalham em conjunto com cada uma das posições do vetor "Novos". Nelas são guardados os dados referentes aos eventos propriamente ditos, cada um deles ocupando duas posições do vetor: uma com o tempo em que o evento irá ocorrer e outra com o tempo gasto em sua execução. Cada "Vetor" é compartilhado por inteiro e acessível de forma remota somente pelo nodo que referencia, da mesma forma que uma posição de "Novos".

4.4.2 Estruturas de dados com acesso local

Diversas estruturas de dados com acesso local são necessárias no protocolo de simulação paralela descrito. Uma extensa lista contendo variáveis de controle interno dos LPs, como contadores, *flags* e acumuladores, poderia ser elaborada, porém detalham-se apenas as mais relevantes e que dizem respeito diretamente ao processo de simulação. Assim, destacam-se a estrutura que armazena a lista de eventos e outras auxiliares, as quais armazenam informações complementares às variáveis compartilhadas.

Na Figura 4.6, encontra-se uma grande estrutura de dados referenciada por "**Lista de Eventos**", a qual pode ser encarada como um vetor, onde cada posição armazena a estrutura de dados descrita na Figura 4.5. A EVL é mantida de forma ordenada, com o tempo de execução do evento Ev_i sempre menor ou igual ao do evento

Ev_{i+1}. Todas as operações relativas à lista de eventos são realizadas localmente sobre essa estrutura.

A estrutura descrita como "**Índices**" é uma estrutura auxiliar às variáveis compartilhadas "Vetor n". Nela, em cada posição, é armazenado o índice do vetor em que o próximo evento a ser agendado em outro LP deverá ser inserido. Por exemplo, se, em LP_0, a variável `INDICES[1]` armazenar o valor 20, estará indicando que o próximo evento agendado por LP_0 em LP_1 deverá ser colocado a partir da vigésima posição na estrutura compartilhada "Vetor 1". Resta salientar que a posição referente ao LP atual não é utilizada.

A estrutura "**Aux**" também é uma estrutura auxiliar no uso de variáveis compartilhadas, armazenando em cada posição a quantidade de eventos já lidos remotamente no nodo referente a sua posição; sua utilização é indispensável para que os acessos à estrutura "Vetor n" possam ser realizados somente como leitura. A cada acesso remoto, é efetuado um cálculo para saber qual é a posição em "Vetor n" que contém os eventos a serem agendados. A quantidade de eventos a serem armazenados é obtida da estrutura compartilhada "Novos". De forma semelhante à estrutura de dados anterior, a posição referente ao próprio LP não é utilizada. A Figura 4.7 ilustra a utilização da estrutura "aux".

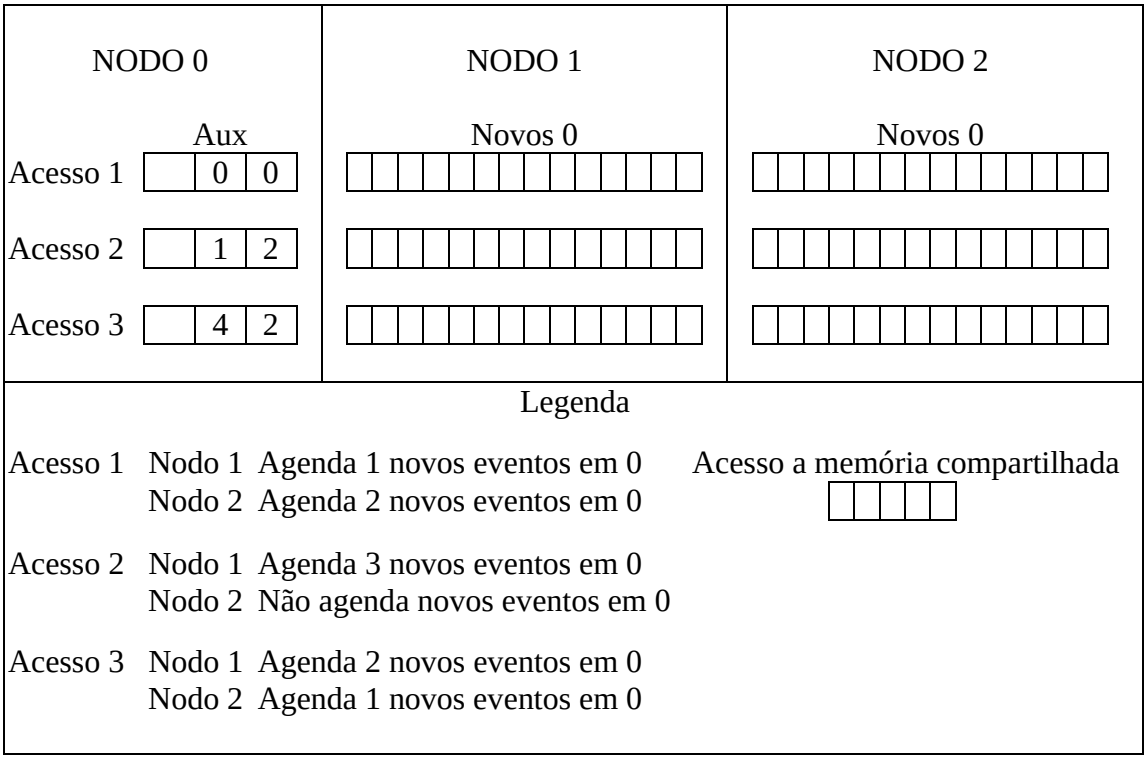


FIGURA 4.7 – Utilização da estrutura de dados "Aux"

A Figura 4.7 ilustra a leitura remota em três acessos consecutivos num ambiente com o mesmos número de LPs, demostrando as leituras realizadas por LP_0 dos eventos a serem agendados nele por LP_1 e LP_2. Sem a utilização da estrutura "Aux", seria necessário um algoritmo para acesso exclusivo aos vetores onde os eventos a serem agendados em outros nodos são armazenados.

4.5 Funcionamento do modelo

O protocolo de simulação paralela com uso de variáveis compartilhadas para ambientes de memória distribuída possui um funcionamento semelhante ao dos protocolos conservadores de simulação paralela de execução síncrona. Características como o uso *lookahead* e de janelas de tempo foram utilizadas com o objetivo de melhorar o desempenho.

A carga dos eventos em memória é realizada uma única vez a fim de não serem necessárias operações de I/O durante o processo de simulação. Os eventos são carregados na memória local de cada LP obedecendo à divisão da EVL realizada. Cada EVL local contém somente os eventos pertencentes a ele na simulação.

De forma semelhante aos protocolos conservadores de execução síncrona, o funcionamento do modelo é dividido em duas partes: a execução de eventos e a sincronização.

A parte de execução está inserida na “Máquina de Simulação”, ao passo que a de sincronização faz parte da “Interface de comunicação” (*vide* seção 2.3.2, Figura 2.1). A parte relativa à execução de eventos inicia a simulação e executa todos os eventos até a chegada a uma barreira de sincronização. Nesse ponto, a parte referente à sincronização assume o controle, executando tarefas como a definição de uma nova barreira de sincronização e a atualização de novos eventos agendados.

Após serem realizadas as tarefas de sincronização, o LP volta a executar eventos. O processo de simulação segue alternando entre as partes enquanto existirem eventos a serem simulados. A Figura 4.8 ilustra um algoritmo em alto nível do protocolo proposto, indicando cada uma de suas partes.

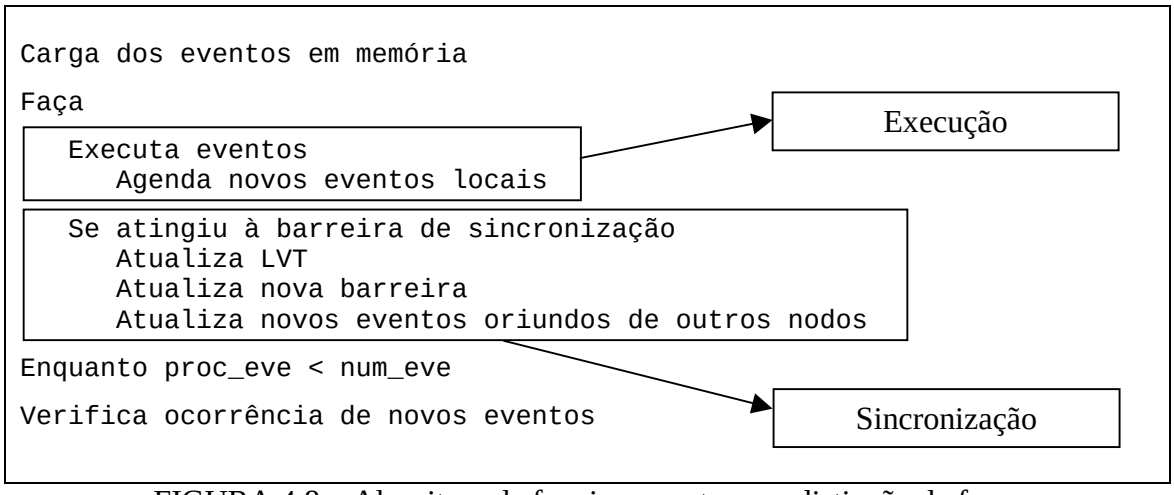


FIGURA 4.8 – Algoritmo de funcionamento com distinção de fases

Para finalizar o processo de simulação, deve ser realizada uma verificação com a finalidade de saber se todos os eventos que seriam agendados nos LPs realmente o foram; caso algum evento não tenha sido agendado, esse não será executado. Essa situação pode ocorrer porque, ao invés de os LPs receberem mensagens com os novos eventos a serem criados, eles buscam essas informações nos demais LPs. Assim, após

um LP encerrar sua execução, um novo evento a ser executado pode ser criado por outro LP que ainda não concluiu suas atividades.

4.5.1 Execução dos eventos

A parte de execução dos eventos é a parte mais simples do presente protocolo, pois nenhuma aplicação do mundo real é simulada; dessa forma, poucas particularidades são encontradas. Essa parte pode ser dividida em três fases:

i. uso de CPU;

ii. agendamento de novos eventos;

iii. atualização do LVT.

Na fase de uso da CPU, a execução dos eventos apenas força o LP a "gastar" o tempo que seria necessário para a execução de um evento, passando, após, para o próximo. Algumas distinções podem ser realizadas em relação aos eventos executados, podendo-se dividi-los em dois conjuntos: os iniciais e os criados dinamicamente.

Os eventos iniciais são aqueles originários do gerador de listas de eventos; durante sua execução, existe a possibilidade de agendamento de novos eventos na própria EVL ou em outra. Os eventos criados dinamicamente são aqueles cuja execução só é possível em razão de um evento precedente, sendo divididos entre eventos internos e externos: os internos dizem respeito àqueles que foram agendados pelo próprio LP, ao passo que os externos foram criados por outros. A divisão desses eventos se faz presente apenas por uma questão de verificação da exatidão do modelo que está sendo simulado; portanto, nenhuma diferença na computação de eventos internos ou externos é realizada no modelo de simulação. Os eventos criados dinamicamente possuem a característica de não agendarem novos eventos.

Quando um evento agenda um outro, a fase de agendamento de novos eventos assume o controle do LP. Computações relativas ao tempo em que o novo evento deverá ocorrer são necessárias, levando em consideração o valor do *lookahead* da simulação. Além de calcular o tempo em que o novo evento a ser agendado deverá ocorrer, deve-se localizar onde o evento será criado. Essa definição é obtida da lista de eventos, que é gerada indicando se e onde cada evento será criado.

Se o evento deve ser agendado na própria EVL, o processo de simulação pára até que ele seja incluído. A inclusão consiste em localizar a posição da lista em que o evento será inserido, abrir um espaço para ele e inseri-lo de maneira efetiva. No caso de o evento ser criado em uma EVL que não seja a própria, apenas ações preparatórias são realizadas. Deve ser efetuada a escolha de qual LP irá receber o evento, tarefa que é executada de forma aleatória, semelhante às rotinas de criação das listas de eventos. Definido o LP, o evento é inserido na estrutura de dados respectiva a partir da posição indicada na variável "Índices" (*vide* seção 4.4.2).

A fase de atualização do LVT atualiza a variável compartilhada que controla o avanço local da simulação. A cada evento executado, o valor de seu *timestamp* é

comparado com o valor atual do LVT; caso seja maior, o LVT é atualizado com o *timestamp* do último evento.

O conceito de janelas de tempo é empregado para possibilitar que o processo avance de forma segura pelos eventos compreendidos dentro de uma janela. A execução de eventos alterna-se dentro de uma janela de tempo entre as três fases descritas, até o encontro de uma barreira de sincronização, ou seja, o LVT alcança o final de uma janela de tempo. Nesse ponto, os eventos param de ser executados e as tarefas relativas à sincronização são computadas.

4.5.2 *Lookahead*

O protocolo que está sendo descrito utiliza o conceito de *lookahead* com o objetivo de diminuir o tempo de simulação. O *lookahead* é tratado como um parâmetro de entrada, podendo ser modificado de uma execução para outra. Dentro do protocolo com uso de variáveis compartilhadas, é utilizado para fins de cálculo do *timestamp* de novos eventos e como fator determinante no cômputo da janela de tempo.

Como os sistemas do mundo real podem ser totalmente diferentes em comportamento, o valor do *lookahead* deve variar de acordo com o sistema a ser simulado. Além disso, o presente trabalho também objetiva criar um ambiente parametrizável para análises de processos de simulação; dessa forma, podem-se testar diversos valores de *lookahead* numa única lista de eventos. Pode-se verificar qual é o seu impacto no tempo total da simulação e na quantia de tempo em que os processadores ficam ociosos, variando seu valor e mantendo as características das seqüências de eventos.

O valor a ser utilizado deve ser informado pelo usuário ao início do processo de simulação, sendo mantido sem alterações durante toda a execução. Uma opção em relação ao *lookahead* constante na simulação é o seu cômputo de maneira dinâmica, podendo ser diferente em vários pontos da simulação. Porém, conforme descrito em ([RIC 95], [POR 98], [NKE 98]), a simulação é uma representação de um único ambiente do mundo real, podendo permanecer com um valor igual na predição do tempo para criação de novos eventos.

Ao início de uma simulação, o *lookahead* (L) do modelo é conhecido e representa qual é a quantia de tempo em que o LP poderá “olhar a frente” em relação à criação de novos eventos. Quando a execução de um evento determina que ele crie um novo, o tempo em que o novo irá ocorrer deve ser calculado. Para esse cálculo, é gerado um valor aleatório dentro do intervalo [1 .. L], o qual representa o adicional a ser incluído no T_{atual} para a execução do novo evento. Porém, se um sistema possui *lookahead* L, diz-se que não pode receber novos eventos com tempo inferior a $T_{atual} + L$. Caso o valor aleatório seja diferente de L, o modelo agendará um novo evento, com uma diferença de tempo de execução em relação ao atual menor que L. Conforme descrito, a única possibilidade de não infringir o conceito do *lookahead* do sistema seria a geração do valor aleatório igual a L, fato que ocorreria com pouca probabilidade à medida que o valor de L aumentasse.

Para a solução desse problema, ao valor gerado sempre é adicionado o valor de L, transformando o intervalo para a geração do valor aleatório em [0 .. (L-1)]. O primeiro valor adicionado a T_{atual} corresponde ao valor do *lookahead* presente no modelo, ao passo que o valor aleatório é o incremento do mesmo, podendo não ocorrer nenhum incremento ou, no máximo, o valor de L menos 1. A subtração de uma unidade no cálculo do valor aleatório é efetuada para evitar que um novo evento seja criado com um tempo T_{atual} + (2 * L), o que pode não corresponder aos sistemas simulados. A fórmula abaixo exemplifica o cálculo do *timestamp* de um novo evento.

$$T_{novo} := T_{atual} + L + random(0, (L-1))$$

A visão do cálculo pode ser exemplificada quando analisado o agendamento de um novo evento na própria EVL. Se o sistema assume que não pode receber novos eventos com tempo inferior a T_{atual} + L, não pode, sob hipótese nenhuma, gerar um tempo inferior a esse para que um novo evento seja criado em sua EVL. Tal regra deve ser mantida tanto para o agendamento de novos eventos na própria EVL quanto em outra, pois, em ambas, estão sendo gerados novos eventos de um mesmo sistema simulado.

O valor do *lookahead* também é utilizado para a determinação das barreiras de sincronização, delimitadoras das janelas de tempo. O protocolo utiliza o valor de L para definir a primeira barreira de sincronização, possibilitando, assim, ao final do tempo em que o primeiro evento pode utilizar o *lookahead*, que uma sincronização seja efetuada para receber os novos eventos de outros LPs. A Figura 4.9a indica um modelo sendo simulado com o *lookahead* de 50, no qual todos os LPs podem avançar de maneira segura até o tempo 50, quando encontram uma barreira de sincronização.

4.5.3 Barreiras de sincronização

O novo protocolo utiliza janelas de tempo para a execução dos eventos a fim de diminuir a comunicação entre os LPs em sua sincronização. Dentro das janelas, o avanço do LVT é livre, devendo o processo, ao chegar à barreira delimitadora do final da janela, efetuar a sincronização. Nesse ponto, o protocolo conservador com uso de janelas de tempo (CTW) obriga os LPs que alcançaram primeiro a barreira de sincronização a esperar pelos demais. Após todos os LPs terem alcançado a barreira, o processo segue adiante, com a nova barreira sendo delimitada (*vide* seção 2.4.3, Figura 2.3).

O presente protocolo faz uso de uma variável compartilhada para o controle do LVT dos LPs, o que possibilita que, ao chegarem à sua barreira de sincronização, os LPs possam “espiar” quais são os valores dos demais e avançar sua barreira, diminuindo, assim, o tempo ocioso. Isso é mostrado na Figura 4.9b, onde se observa que a barreira inicialmente definida para LP_2 era 50 e, ao alcançá-la, foi redefinida para 80; já as demais permaneceram inalteradas.

Para efetuar o avanço da barreira de sincronização de LP_2, foi utilizado o menor LVT entre os LPs. A esse valor foi adicionado o valor constante do *lookahead*, possibilitando que LP_2 avançasse até o tempo de simulação 80. Com esse avanço, LP_2 não necessita esperar pelos demais para prosseguir com a simulação. Nesse ponto, os

eventos agendados nos outros nodos a serem executados em LP_2 devem ser efetivados. Para exemplificar, se LP_0, no tempo de simulação um (1), agendar um evento em LP_2, o tempo em que ele deverá ocorrer poderá ser 51; assim, o processo pode executá-lo sem que aconteçam problemas de erros LCC.

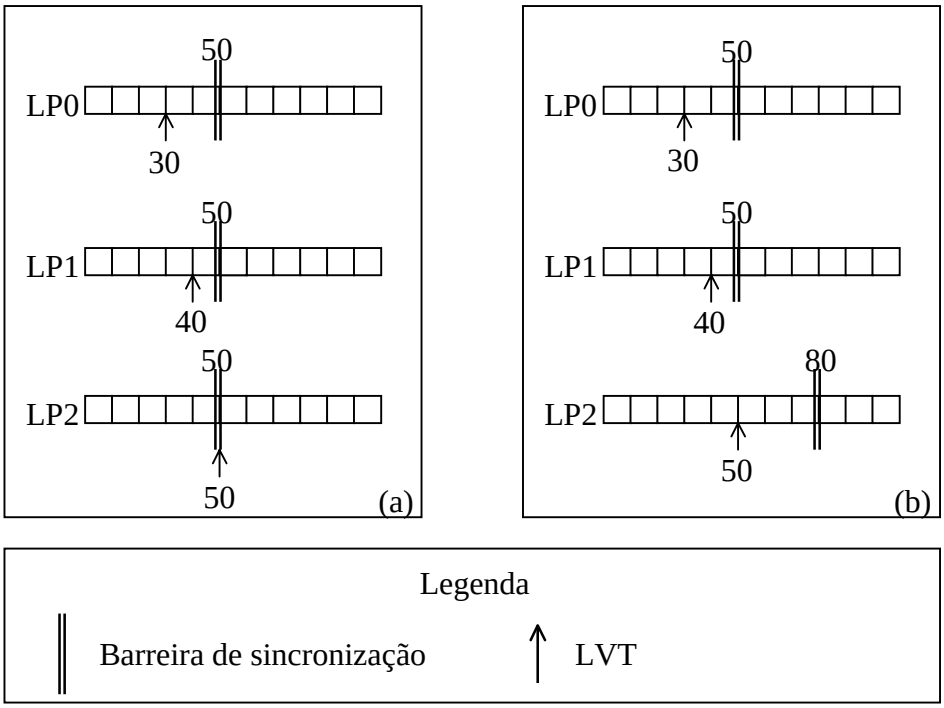


FIGURA 4.9 – Definição da primeira barreira

O LP_2 não pode avançar sua barreira para 100, pois eventos que aconteceriam entre o tempo 80 e 100 ainda podem ser agendados em LP_2 por outros LPs. Como o avanço dentro da janela de tempo é livre, ele correria o risco de alcançar novamente sua barreira de sincronização, no caso de um processador mais rápido que os demais ou com poucos eventos a executar. Dessa forma, eventos que deveriam ocorrer no intervalo de tempo entre 80 e 100 somente seriam percebidos em LP_2 no tempo de simulação 100, ocasionando erros LCC.

A prática da escolha do menor LVT entre os LPs que estão na fase de execução de eventos para calcular a próxima barreira de sincronização nem sempre pode ser utilizada. Caso um processador alcance a sua barreira de sincronização, mas tenha "atrasado" a simulação anteriormente, os LVTs dos demais LPs serão maiores que o dele. Assim, se for utilizado o menor LVT dos LPs em execução para o cômputo da nova barreira, erros LCC poderiam acontecer e o princípio operacional dos protocolos conservadores seria infringido. É o caso representado na Figura 4.10c.

Na Figura 410c, LP_0 e LP_1 alcançaram sua barreira de sincronização, tendo LP_2 um LVT superior a ela. Assim, a nova barreira seria definida em 120, o que poderia causar uma violação do princípio do *lookahead* e, possivelmente, erros LCC. Então, a nova barreira é definida em 100, utilizando o valor do próprio LVT somado ao valor do *lookahead*. É importante salientar que LP_2 não pára seu processamento para que se procedam às atualizações em LP_0 e LP_1. O acesso à memória compartilhada deve ocorrer sem obstrução ou qualquer outro tipo de interferência ao LP onde essa está localizada fisicamente.

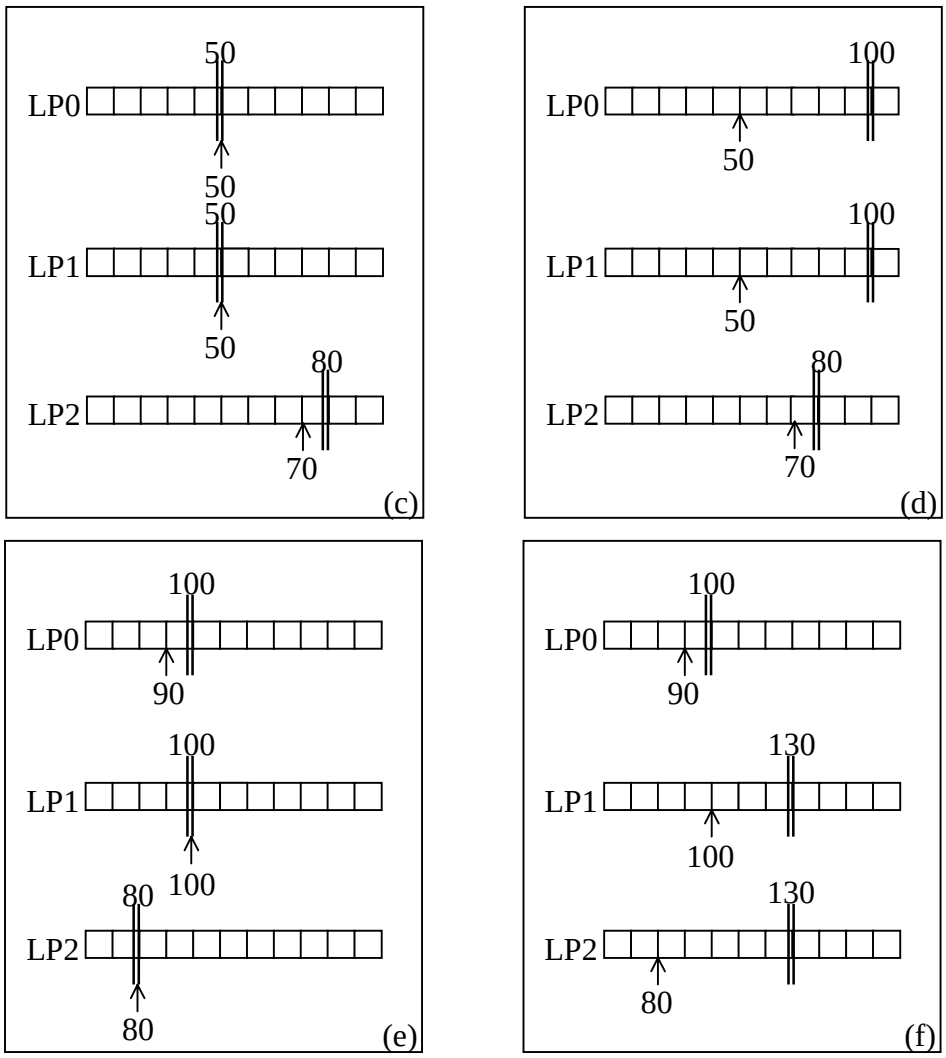


FIGURA 4.10 – Atualização da barreira de sincronização

Deve-se ressaltar que a definição das barreiras de sincronização leva em conta o valor do *lookahead*, contudo, enquanto ele é constante nos LPs, as barreiras são obrigatoriamente iguais apenas no início da simulação. O descompasso entre a barreira de um LP em relação à de outro pode ocorrer desde que os princípios da simulação sejam preservados.

Cada LP deve controlar de forma independente sua barreira de sincronização e seu LVT, porém a diferença entre a menor barreira presente no sistema em relação à maior nunca deverá ultrapassar o valor do *lookahead*. Mantendo essa condição sempre verdadeira, o princípio do não acontecimento de erros LCC é preservado. Essa política propicia situações nas quais a chegada à barreira de sincronização ocorre em diferentes LPs com valores de LVT distintos. A cada situação, a mesma rotina de atualização deve ser efetuada. É o caso da Figura 4.10e, onde dois LPs chegam a sua barreira, utilizando, para a geração do novo valor de sincronização, diferentes estratégias de cálculo em cada uma, porém estabelecendo um mesmo valor para a barreira.

A prática de avançar a barreira tomando por base o menor LVT existente na simulação oferece um novo período em que o LP pode continuar a executar seus

eventos. Saliente-se que esse período nunca é igual ao *lookahead* do modelo, porém pode beneficiar o processo de simulação. O LP pode executar eventos que o fariam atrasar a execução de outros contidos nas janelas de tempo seguintes. Dessa forma, além de diminuir o tempo ocioso, pode contribuir de forma efetiva na redução do tempo total da simulação.

4.5.4 Tratamento de *deadlock*

Um dos problemas a serem resolvidos nos protocolos conservadores de simulação paralela é o gerenciamento de situações de *deadlock*. O presente protocolo também observa essa preocupação com a utilização de janelas de tempo e *lookahead*. Como o protocolo se divide em duas partes, a de execução e a de sincronização, uma das tarefas da sincronização é evitar que o *deadlock* aconteça.

Em uma execução normal, o modelo não entra em *deadlock* visto que sempre existe um evento sendo executado ou a simulação chegou ao final. Os protocolos que se utilizam de duas fases, geralmente na sincronização, podem eleger um evento que pode acabar com uma situação de *deadlock*. No presente protocolo, isso não é necessário visto que, com o uso do menor LVT para o avanço da barreira local de sincronização, poderá ser encontrado um incremento mínimo de barreira de sincronização. Por menor que seja o avanço, proporciona novos eventos seguros para execução.

Se o menor LVT do modelo somado ao *lookahead* resultar em valor igual ao LVT atual, a barreira não pode avançar. Nesse caso, o LP deverá ficar aguardando até que o LP que está “atrasando” a simulação prossiga, liberando os demais. Um valor de *lookahead* muito pequeno ou eventos com diferenças de tempo de execução muito grandes podem ocasionar essa situação. Nesse caso, tanto no presente protocolo quanto em outros conservadores, não há como escapar da espera.

Salienta-se que, mesmo esperando pelo LP com o menor LVT, os processos não se encontram em *deadlock*, que, na simulação paralela conservadora, ocorre quando nenhum evento está sendo executado e não é possível determinar um seguro. Tal situação não acontece no presente protocolo, pois o sistema está executando um evento; assim, descaracteriza o *deadlock*.

4.5.5 Controle dos novos eventos agendados

Outra tarefa que um LP deve executar no momento da sincronização é a atualização dos eventos que lhe foram agendados por outros. Um LP somente pode avançar sua barreira e liberar a execução dos eventos se tiver certeza de que não receberá eventos a executar com *timestamp* inferior ao LVT.

O controle dos novos eventos agendados pelo próprio LP é realizado ainda no momento da sua execução, os quais são inseridos na EVL logo após a execução dos que os criaram, antes da execução do evento seguinte. Dessa forma, como o agendamento de eventos na própria EVL funciona de forma semelhante a uma simulação seqüencial, deve-se apenas calcular o *timestamp*, encontrar seu lugar na EVL e inseri-lo em ordem.

Por outro lado, o agendamento de eventos oriundos de outros LPs é mais complexo, sendo efetuado a cada sincronização, pois não serão gerados eventos com *timestamp* inferior à barreira de um LP enquanto esse estiver com o LVT dentro de uma janela que tenha ao seu final a referida barreira. Somente os eventos já agendados até o momento em que um LP realiza a sincronização serão criados; os demais o serão na próxima sincronização. Essa estratégia não ocasionará problemas, pois a barreira não será acrescida com um valor em que possam ocorrer erros LCC.

O processo de agendamento de eventos oriundos de outros LPs é dividido em duas etapas: a preparação e a efetivação. Na preparação, é realizado o cálculo do *timestamp* do evento; logo após, esse é armazenado na estrutura de memória compartilhada referente ao LP em que deve ser criado. Na efetivação, os eventos propriamente ditos são obtidos e inseridos na EVL.

Uma distinção do presente protocolo em relação aos baseados em trocas de mensagens é que um LP não recebe dos outros os eventos a serem executados; ao contrário, ele os lê remotamente nos LPs correspondentes. Essa ação só é possível com o uso da memória compartilhada, fundamento em que se baseia o presente protocolo. A vantagem dessa abordagem é que o LP não precisa esperar pela informação, mas quando dela necessita pode buscá-la diretamente em sua origem. Dessa forma, o LP pode avançar seu processamento sem depender de que os demais se comuniquem com ele.

Foi criado um algoritmo para a leitura das informações nos LPs. A cada sincronização, um ou dois acessos às estruturas de dados compartilhadas que tratam dos eventos são realizados: um para conhecer a quantidade de eventos e outro para, efetivamente, pegá-los. Assim, é sempre realizado um acesso à estrutura “Novos”, na posição referente ao LP atual. Esse valor é comparado com o armazenado no vetor “Aux” na posição referente ao LP lido; caso retorne um valor maior, um acesso é realizado à estrutura “Vetor n”[9] para obter informações relativas ao *timestamp* e tempo gasto com a execução dos eventos. O segundo acesso é realizado sempre em diferentes posições do “Vetor n”. Esse algoritmo de leitura deve ser realizado em todos os demais LPs presentes na simulação. A definição da posição do “Vetor n” em que será realizada a segunda leitura remota é efetuada utilizando a variável local “Aux”, que tem seu valor atualizado a cada leitura remota realizada (*vide* seção 4.4.2, Figura 4.7).

Resta salientar que todos os eventos lidos remotamente já foram armazenados na sua respectiva estrutura antes que se proceda à leitura, o que ocorre porque, na preparação dos novos eventos a serem agendados em outros LPs, a variável que controla a sua quantidade (“Novos”) somente é atualizada após o armazenamento do evento no “Vetor n”.

4.5.6 Finalização do processo de simulação

O processo de simulação utilizando o protocolo descrito neste capítulo fica alternando entre as partes de execução e de sincronização enquanto existirem eventos a serem simulados; no momento em que esses acabam, o processo poderia encerrar sua execução, pois não existiriam mais eventos. Porém, essa prática, aliada à opção de ler

[9] Neste ponto, “n” corresponde ao nodo atual,

remotamente os eventos ao invés de recebê-los, pode ocasionar situações em que eventos são agendados, porém não são efetivados nas EVLs e, conseqüentemente, não são executados.

Considerando uma situação em que um LP se encontre na sua última janela de tempo, ou seja, após o final dessa não existam mais eventos a serem executados e ele seja o primeiro a alcançar esta condição no ambiente paralelo, quando chegar à barreira, irá realizar a sincronização, obter os novos eventos, inseri-los na sua EVL e executá-los, podendo, assim, encerrar suas atividades. Porém, como foi o primeiro a realizar a sincronização, os demais LPs ainda não executaram todos os seus eventos. Dessa forma, um evento que será executado num momento posterior pode agendar um novo evento no LP que já teria terminado a simulação.

Esse problema de terminação da simulação é solucionado com a utilização de uma verificação posterior ao término da execução dos eventos da EVL de um LP. Após um LP encerrar o processamento de eventos de sua EVL, ele espera que os demais também acabem a execução dos contidos nas suas respectivas listas. Após todos passarem esse estágio, uma nova verificação na memória compartilhada é realizada a fim de verificar se novos eventos a serem executados foram criados, buscando garantir, assim, que nenhum evento deixe de ser executado.

Caso eventos sejam criados, esses devem ser lidos remotamente, inseridos na EVL e executados, processo que visa não perder os últimos eventos do modelo simulado. Essa abordagem foi necessária porque os eventos não são enviados aos LPs, mas os processos os lêem na memória compartilhada dos outros. Nos casos com a utilização de trocas de mensagens, antes de encerrar a execução, um LP deveria enviar seus novos eventos aos outros, realizando também o recebimento de eventos. Assim, os protocolos com uso de MP não necessitam fazer esta última verificação.

A verificação ao final do processamento da EVL somente é executada uma vez em todos os LPs. Obedecendo à regra de que todos os processos já terminaram a execução dos eventos contidos inicialmente na EVL, somente os demais deverão ser processados. Conforme modelagem da lista de eventos, os criados dinamicamente não agendam outros; dessa forma, não é necessária uma verificação recursiva em busca de novos eventos agendados após o término dos inicialmente contidos na EVL.

4.6 Resultados produzidos

O protocolo descrito não simula nenhuma situação do mundo real; portanto, não se produzem resultados práticos de aplicações. Aplicações de simulação geralmente produzem, ao final do processo, valores comprovando se a alternativa simulada é melhor ou pior. Resultados como a aprovação ou rejeição de uma modificação de uma via de trânsito, por exemplo, são comuns em processos normais de simulação.

Na simulação paralela, além de resultados da porção do mundo real simulada, o tempo consumido na simulação também pode ser considerado como resultado. Se o modelo for simulado em menos tempo, maiores alternativas poderão ser testadas. Como o presente modelo não tem nenhuma relação com aplicação específica, apenas são

extraídos resultados referentes aos tempos envolvidos na simulação. Nesse enfoque, duas medidas de tempo podem ser obtidas:

- tempo de simulação: é a quantia de tempo real que um processo consome para a execução de uma simulação;

- tempo ocioso: é a quantia de tempo em que um LP não está executando eventos da simulação. Nele são computados os tempos necessários à sincronização e, também, o tempo em que os LPs ficam esperando pelos demais.

O tempo de simulação é o tempo real da sua duração, não podendo ser confundido com o tempo simulado, comum em simulações e que representa, geralmente, uma escala com que os simuladores trabalham para acelerar o processamento. Ele não tem referência alguma com qualquer tempo simulado por dois motivos: não há simulação de nenhuma parte do mundo real e o tempo consumido pelos eventos é o tempo que, fisicamente, eles consumiriam na execução de um evento qualquer (é o tempo de máquina de um evento). No tempo de simulação, o tempo ocioso também é computado visto que essas tarefas são necessárias para a execução da simulação.

O cômputo do tempo ocioso é realizado sempre que um LP entra na fase de sincronização com os demais. Ações como leitura de memória remota consomem uma grande quantidade de tempo e, neste momento, o LP não pode estar executando eventos. Os casos em que o LP deve ficar esperando, impossibilitado de avançar seu LVT, por causa de uma diferença entre o valor da barreira e o *timestamp* do menor evento que está sendo executado, também são computados no tempo ocioso.

4.7 Considerações finais

No capítulo, descreveu-se um novo protocolo para simulação paralela, tendo como base de funcionamento os protocolos conservadores; utilizaram-se diversas características dos protocolos CMB, mas também se introduziram várias mudanças em seu funcionamento. A sua execução assemelha-se à dos protocolos conservadores de execução síncrona, utilizando conceitos como o *lookahead* e janelas de tempo para execução de eventos.

A principal diferença é em nível conceitual, proporcionada pelo acesso remoto à memória de um LP por outro. Essa mudança produziu diversas outras nas funções relativas à sincronização dos processos, como o avanço local da simulação e o agendamento de novos eventos oriundos de outros LPs. Um ganho adicional obtido foi a fácil resolução do *deadlock*, um dos pontos a serem resolvidos nos protocolos conservadores de simulação paralela.

A construção de uma interface de comunicação eficiente com uso de memória compartilhada é o principal enfoque do protocolo, motivo pelo qual nenhum resultado específico é obtido além da quantidade de tempo consumida na simulação. Além disso, o tempo ocioso (gasto com a comunicação entre os processos ou parado) também é

fornecido para que experiências com diversos parâmetros de entrada possam ser efetuadas.

Além de uma implementação facilitada, propiciada pelo uso de memória compartilhada ao invés de trocas de mensagens, o modelo oferece a possibilidade de melhor ocupar o tempo ocioso dos processadores. Resta salientar que as modificações efetuadas, em nenhum momento, infringiram o princípio operacional dos protocolos conservadores, que é não possibilitar a ocorrência de erros de causalidade local.

5 Implementação do protocolo

O protocolo de simulação descrito proporciona uma possibilidade de redução do tempo de processamento ocioso dos protocolos conservadores, além do que a programação com uso do paradigma de memória compartilhada é mais simples de ser compreendida. Para validar as idéias gerais do novo protocolo, realizou-se uma implementação seguida de testes.

Este capítulo apresenta a implementação do protocolo e de outros dois simuladores implementados para fins de comparação. Inicialmente, descreve-se a estrutura da implementação, juntamente com o ambiente paralelo utilizado e a linguagem de programação escolhida. Como o protocolo faz uso de memória compartilhada e o ambiente é baseado em memória distribuída, introduzem-se noções da biblioteca utilizada para emular um ambiente DSM. Ao final, destaca-se um algoritmo que descreve o funcionamento do simulador descrito.

5.1 Introdução

Na implementação de aplicações paralelas, podem ser utilizados diversos ambientes paralelos e linguagens de programação. As aplicações construídas são paralelizadas de forma automática através de compiladores paralelizantes, como *Suif - Stanford University Intermediate Format compiler*[10], ou com uso de bibliotecas de comunicação que exploram o paralelismo de maneira explícita, como LAM/MPI[11] e PVM[12], as quais são as mais utilizadas na construção de programas a serem executados em multicomputadores. Há, ainda, a possibilidade da escrita de programas paralelos utilizando linguagens paralelas, como SR[13], entre outras.

Um dos objetivos das aplicações paralelas é o aumento de seu desempenho, tentando obter resultados confiáveis com menor tempo de execução. Dessa forma, os mais rápidos ambientes paralelos disponíveis, as linguagens de programação mais flexíveis e bibliotecas de comunicação menos ineficientes se sobressaem no momento da escolha do ambiente e das ferramentas a serem utilizadas. Porém, o ambiente que apresenta o melhor desempenho em uma aplicação pode não alcançar a mesma posição em outra com diferentes características. A disponibilidade do equipamento ideal para uma implementação nem sempre é conseguida, em razão de fatores que extrapolam os objetivos deste texto.

A facilidade da programação assim como a adaptação ao tipo de programa implementado devem ser levadas em conta na escolha do *software* utilizado. Na implementação de programas seqüenciais, existem linguagens especializadas para praticamente todos os tipos de aplicações, contudo, por causa de suas particularidades, nem todas conseguem se adaptar a implementações com exploração do paralelismo.

[10] http://suif.stanford.edu/
[11] http://www.mpi.nd.edu/lam/
[12] http://www.epm.ornl.gov/pvm/
[13] http://www.cs.arizona.edu/sr/www/index.html

Outro ponto a ser considerado, não somente em implementações de novas soluções, é a questão da inovação. O trabalho de pesquisa de novas ferramentas computacionais, assim como de ambientes de execução, somente pode avançar com a sua utilização nas mais variadas áreas.

Implementações que descrevem novas opções para resolução de antigos problemas, como o novo protocolo descrito no capítulo 4, devem ser acompanhadas de outras tradicionais. A comparação entre elas é fator decisivo para a aprovação ou rejeição das novas opções, sempre com o cuidado de se resguardar os objetivos a que se destinam e as facilidades visadas.

5.2 Estrutura da implementação

Na implementação do protocolo de simulação paralela com uso de memória compartilhada em ambientes distribuídos, utilizaram-se três programas, dois deles seqüenciais e o simulador propriamente dito paralelo. A Figura 5.1 ilustra a estrutura da implementação realizada, assim como os arquivos de entrada e saída utilizados pelos programas.

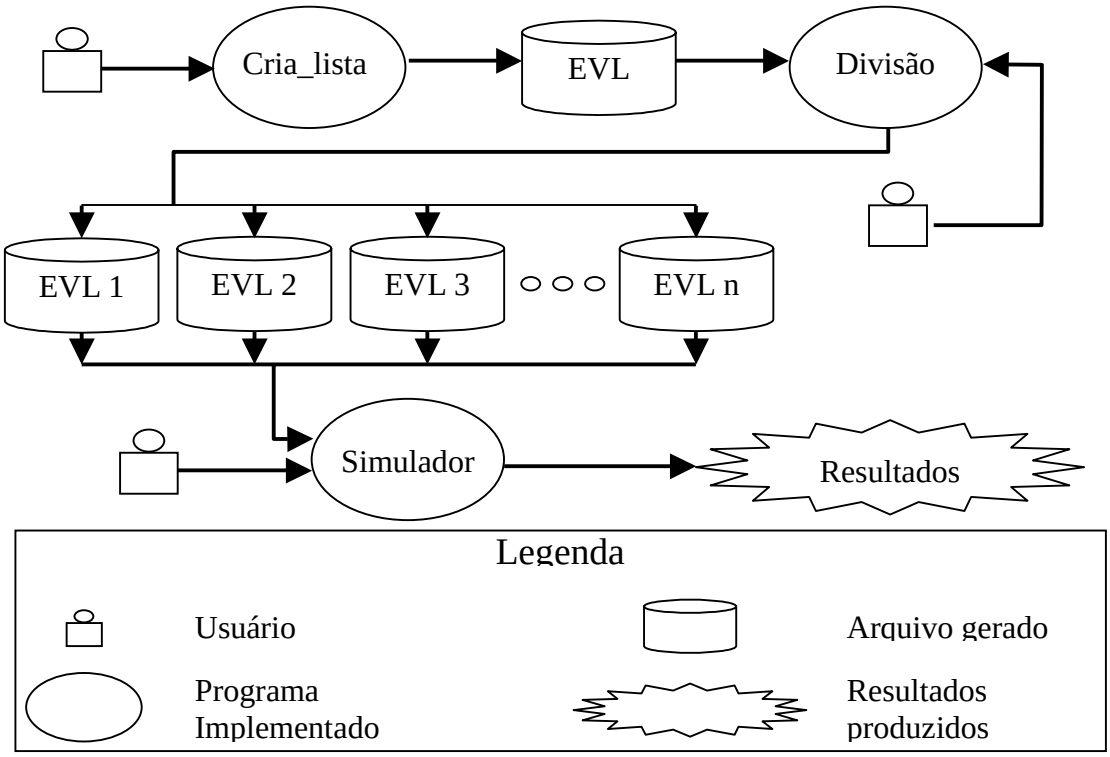


FIGURA 5.1 – Estrutura da implementação

Inicialmente, o usuário informa os parâmetros de entrada ao programa que gera as listas de eventos a serem executados (Cria_lista), o qual possui a execução seqüencial, criando um arquivo que os contém. Esse arquivo serve como entrada no programa de execução em apenas um processador que divide a EVL em diversas outras para execução paralela. A quantidade de listas resultantes é necessária para a divisão, sendo informada pelo usuário; as listas divididas alimentam o simulador paralelo, gerando os resultados comentados na descrição do protocolo. O usuário deve informar qual é o valor do *lookahead* a ser utilizado na simulação, bem como o nome da lista e a

quantidade de eventos que ela contém. A quantidade de eventos foi adicionada apenas por questões de desempenho, evitando uma passagem por toda a lista a fim de encontrar esta informação, a qual é necessária antes da carga dos eventos a fim de alocar dinamicamente a memória necessária a cada processo. Como alternativa, tal informação poderia estar no cabeçalho de cada lista.

A passagem da execução de um programa para outro não é automática, porém, caso o usuário queira, pode informar todos os parâmetros na linha de comando, agilizando o processo de testes. A interface de operação de todos os programas é baseada em texto, e os resultados, mostrados no dispositivo de saída padrão.

Os programas auxiliares ao simulador (cria e divide lista) foram implementados contendo o código fonte em um arquivo. Por outro lado, o simulador teve sua implementação particionada em vários módulos, dividindo o fonte de acordo com as características da utilização. A Tabela 5.1 descreve cada módulo do simulador assim como as funções que desempenha.

TABELA 5.1 – Módulos do simulador paralelo

Módulo	Funções
simupar.c	É o programa principal. É de sua responsabilidade receber os parâmetros da linha de comando, chamar tarefas para carregar a lista de eventos e chamar o procedimento de simulação.
simupar.h	Arquivo que contém os protótipos das funções utilizadas e as estruturas de dados comuns entre todos os módulos.
simula.c	É o principal módulo; executa a simulação, realizando a sincronização dos LVTs e o agendamento de novos eventos baseados em SM. O cômputo dos tempos de simulação e ocioso dos processadores também é realizado pelo mesmo.
aux_simula.c	Arquivo que contém funções para auxiliar o processo de simulação. Nele encontram-se funções necessárias à execução da simulação, porém triviais, como controle de tempo consumido nos eventos e cálculos de intervalos de tempo.
estrutura.c	Programa responsável pelas tarefas sobre a EVL. A carga a partir da memória estável e a inserção física dos eventos são exemplos de funções realizadas.
leituras.c	Arquivo que contém funções que servem para o usuário informar, ao início do processo, os parâmetros necessários para a simulação, no caso de não terem sido passados na linha de comando.

5.3 Ambiente e ferramentas utilizadas

O ambiente de execução planejado para a implementação do modelo descrito baseia-se em estações de trabalho ligadas através de uma rede de conexão e, além de ser escalável, apresenta-se como promissor na execução de simulações [GEO 99]; além de

ter estado disponível para a implementação e testes. O paradigma de implementação utilizado foi o de memória compartilhada, com exploração explícita do paralelismo.

O modelo foi implementado e testado em três máquinas SUN monoprocessadas, sendo duas de 270MHz e outra de 300MHz, todas contendo 192Mb de memória RAM. A comunicação foi proporcionada por um *Hub FastEthernet*, sendo que, durante os testes realizados, as estações de trabalho constituíram uma LAN isolada. Salienta-se que o uso de apenas três processadores na simulação ocorreu em virtude de esse equipamento estar disponível, porém o programa pode ser executado utilizando-se um número maior de unidades processadoras.

Simulação paralela de eventos são aplicações nas quais a qualidade das metodologias de paralelização pode render aumentos dramáticos de desempenho [MUS 99]. Dessa forma, o paralelismo foi explorado de maneira explícita no código fonte através da inserção de primitivas, possibilitando a exploração do paralelismo em pontos nos quais compiladores paralelizantes talvez não o possam.

A biblioteca *Athapascan0* ([BRI 98a], [BRI 98b]) foi escolhida para a exploração do paralelismo na implementação do modelo. Desenvolvida no LMC-IMAG (*Laboratoire de Modélisation et Calcul - Institut d'Informatique et de Mathematiques Appliquees de Grenoble*) pelo grupo Apache[14], permite a programação paralela em computadores de memória distribuída com as características da memória compartilhada. Além da programação em SM, essa biblioteca também possibilita a criação de aplicações paralelas baseadas em trocas de mensagens.

Para a execução dos programas paralelos, a biblioteca necessita da formação de uma máquina virtual, a qual é composta de um conjunto de nodos, cada um representado por um processo. Um único computador pode executar vários nodos, sendo que, na presente implementação, apenas um foi ativado em cada computador. Para a criação de uma máquina virtual, o número de nodos deve ser especificado. A alocação de processos e sua criação não são parte integrante do *Athapascan0* [BRI 98a], constituindo tarefas que devem ser fornecidas pela camada de suporte às comunicações da biblioteca, constituída, atualmente, pelo MPI, um padrão mundial que a torna, assim, portável para diferentes plataformas.

A versão do *Athapascan0* utilizada foi a 2.4.13, que trabalha junto com o LAM/MPI versão 6.1 no suporte às comunicações. O modelo de programação das aplicações imposto pelo MPI é o SPMD, no qual os processadores executam um mesmo programa. A distinção que ocorre na execução é realizada com as variáveis de ambiente que determinam o nodo atual e quantos estão ativos no processo paralelo, as quais podem ser obtidas usando-se as primitivas "`a0SelfNode`" e "`a0NodeCount`". A primeira fornece o nodo atual; a segunda quantos estão presentes. Eles são numerados de zero até a quantidade de processadores menos um, tendo o nodo zero a responsabilidade de inicializar os processos [BRI 98b]. As primitivas relacionadas fornecem, em tempo de execução, os valores para essas variáveis. A Figura 5.2 ilustra um código com uso das variáveis mencionadas.

[14] http://www-imag.fr/apache

```
me  = a0SelfNode;
nbp = a0NodeCount;

. . .

onde = me;                        /* sorteia um nodo que */
while (onde == me)                /* nao o proprio para  */
   onde = (random() % nbp);       /* agendar um evento   */
```

FIGURA 5.2 – Variáveis que diferenciam a execução nos nodos

Inicialmente, as primitivas citadas têm seu valor atribuído a variáveis locais nos nodos. Após, o programa sorteia aleatoriamente um valor, atribuído à variável "`onde`", representando o nodo em que será agendado um novo evento. Neste cálculo, é utilizado o resto da divisão do resultado da função "`random()`" pela quantidade de nodos que atuam no processo atual. O processo repete-se até que seja encontrado um valor diferente do nodo atual.

Athapascan0 é responsável pela comunicação e suas primitivas podem ser chamadas dentro de programas nos momentos de comunicações entre os processos. Esses programas podem ser escritos na linguagem C, caso da presente implementação, cuja escolha se deve ao fato de ser largamente utilizada em aplicações que buscam um bom desempenho. Ela é uma das linguagens imperativas mais populares, utilizando-se, na implementação realizada do modelo descrito, a programação estruturada. Em razão da sua proximidade da máquina, ela é uma linguagem poderosa e eficiente, utilizada para construção de sistemas operacionais e compiladores, entre outras aplicações.

O padrão de programação ANSI foi seguido em todos os fontes, controlado pelo compilador GCC versão 2.8.1, utilizando a diretiva de compilação "`-ansi`"; sua adoção possibilita grande compatibilidade, desta forma, testes do simulador paralelo implementado podem ser realizados em várias plataformas.

5.4 Particularidades da implementação

Todo o protocolo descrito no capítulo quatro foi implementado, quando foram adicionadas algumas modificações com o objetivo de conseguir uma maior precisão e controle mais apurado do processo. Como exemplo, variáveis complementares foram necessárias para o acesso remoto à memória; a execução dos eventos passou, explicitamente, a controlar a ocorrência de erros LCC, além de a quantidade de eventos simulados ter sido controlada para garantir a exatidão da simulação.

Em virtude da execução em um ambiente multicomputador sem que todas as permissões para os usuários nas diferentes máquinas tivessem sido concedidas, não foi possível, em cada máquina, abrir o arquivo referente a sua EVL. Apenas a que iniciou a máquina virtual e disparou a simulação conseguiu, na presente implementação, abrir e ler os arquivos da memória estável. Para resolver esse problema, uma modificação ao protocolo proposto foi realizada na implementação, a qual faz com que o nodo zero proceda à leitura de todos os arquivos, enviando aos demais os eventos a eles pertencentes. A Figura 5.3 descreve a forma de carga inicial dos eventos.

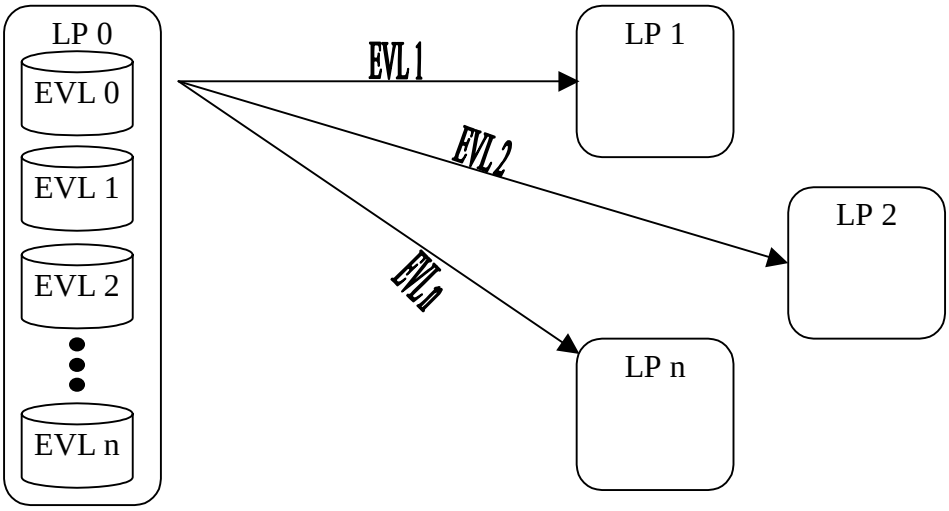


FIGURA 5.3 – Carga das listas de eventos a partir do nodo 0

Com a modificação realizada, todas as listas de eventos devem permanecer na memória estável do LP_0. A leitura é efetuada iniciando na última lista de eventos da simulação até encontrar a primeira, sendo todas as listas, exceto a lista referente ao nodo zero, enviadas aos seus respectivos LPs.

As estruturas compartilhadas descritas na seção 4.4.1 somente são acessíveis remotamente após uma preparação da região de memória onde se encontram, a qual envolve criação de um descritor de tipo exclusivo à biblioteca chamado "`a0tDMARegion`"[15]. Esse passará a responder pela variável ou conjunto de variáveis definidas em sua criação [PAS 98], processo que envolve o tipo de dado a ser acessado e a quantidade de variáveis distintas a serem armazenadas em uma região.

Os LPs que realizam os acessos remotos à memória devem receber os descritores das regiões através de mensagens enviadas pelos nodos que as contêm fisicamente, os quais são armazenados em variáveis de mesmo tipo; quanto aos acessos remotos à memória, devem ser realizados usando os dados recebidos. Nesta implementação, foram utilizados vetores de descritores de região de memória compartilhada tanto para o armazenamento de descritores de outros nodos quanto para a criação e envio de variáveis referentes à própria memória compartilhada. A Figura 5.4 mostra a forma de utilização dos descritores de regiões de memória.

[15] a0t representa um tipo Athapascan0 e DMA é abreviação de "Direct Memory Access" [PAS 98]

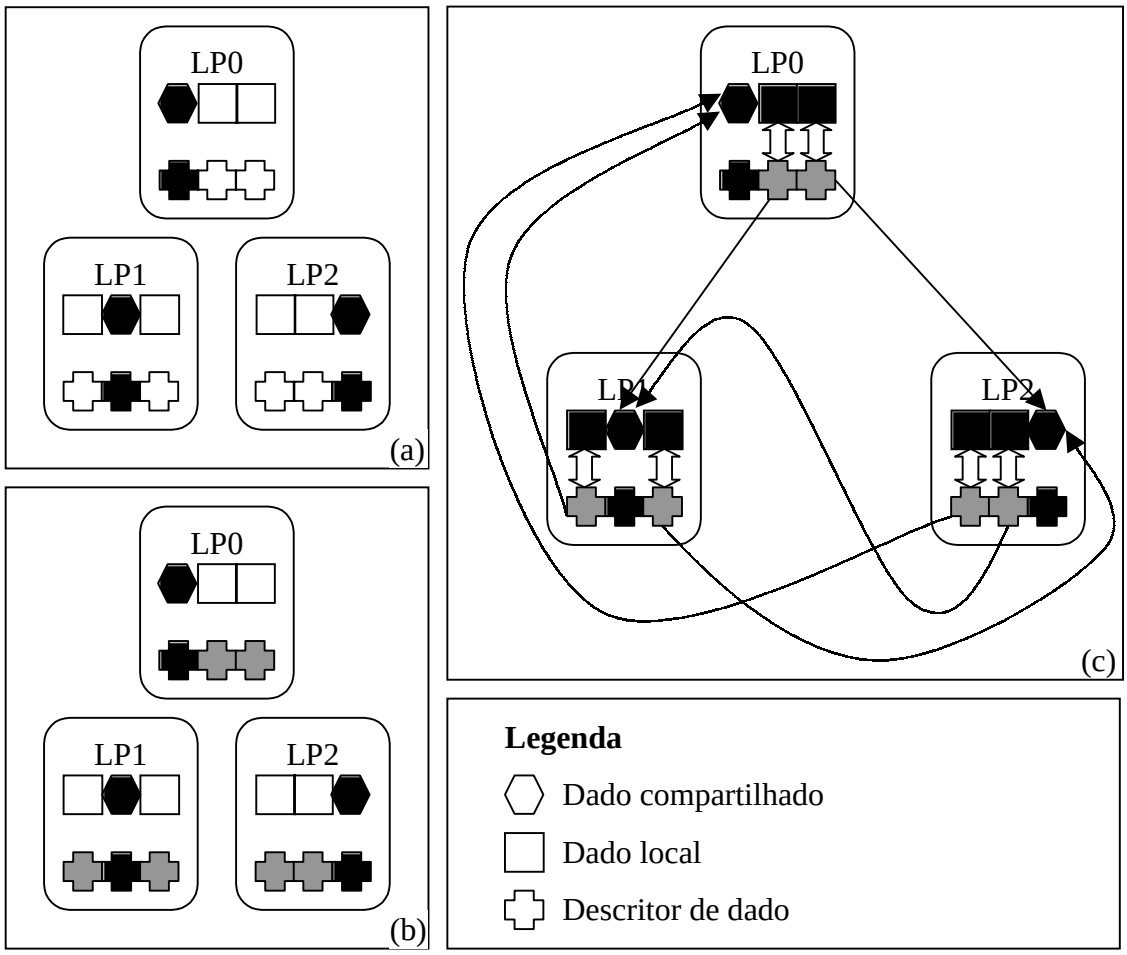


FIGURA 5.4 – Descritores para acesso remoto à memória

De forma semelhante ao que ocorreu no caso da Figura 4.6, a Figura 5.4 foi extraída de um processo de simulação envolvendo apenas três LPs e descreve as estruturas de dados necessárias para o acesso remoto às variáveis compartilhadas. Em cada LP, o primeiro vetor representa a estrutura LVT, onde uma posição é compartilhada e as demais são atualizadas a partir dos outros; o segundo representa os descritores para essas regiões de memória.

Na Figura 5.4a, a posição referente ao nodo atual de LVT é compartilhada (por exemplo, no nodo zero (0) a posição 0), tendo seu respectivo descritor criado. Logo após, os descritores criados são enviados aos demais nodos, sendo armazenados na mesma estrutura que guarda o descritor de região compartilhada criado, porém nas posições referentes aos LPs de origem (o descritor de região compartilhada de LP_0 em LP_1 é armazenado na posição zero). O estado dos LPs, após o recebimento das informações dos demais pode ser observado na Figura 5.4b, onde as estruturas sombreadas representam apontadores para regiões de memória remota recebidos de outros LPs. As posições do vetor de descritores de região de memória funcionam como apontadores para as regiões de memória compartilhada dos demais nodos, sendo que as atualizações do vetor de LVTs somente podem ser realizadas com a utilização dos descritores de região compartilhada. Esses acessos são resumidos na Figura 5.4c.

A fim de garantir a não-ocorrência de erros LCC (*Local Causality Constrains*), três modificações foram efetuadas no modelo, duas durante a execução da simulação e

outra na finalização do processo. Tais modificações foram inseridas com o objetivo de proporcionar um complemento na garantia da não-execução de um evento com *timestamp* menor que o LVT atual, ou a não-execução de um evento.

Na execução dos eventos, ocorre uma verificação do *timestamp* de cada evento em relação ao LVT; de forma semelhante, no momento da inserção dos eventos oriundos de outros LPs, o mesmo teste é realizado. Em qualquer caso, se o *timestamp* do evento for menor que o LVT, um erro LCC ocorreu; portanto, o simulador não é válido, pois viola o princípio operacional dos protocolos conservadores.

Outra providência para garantir a não-ocorrência de erros LCC foi a inserção de um contador de eventos executados. Ao final da simulação, o número de eventos presentes na EVL deve ser o mesmo do contador; um valor diferente aponta erros LCC não ocorridos, quando um evento foi inserido na EVL, porém não executado. Além desse controle, um benefício extra em relação aos resultados obtidos no modelo foi alcançado. Como, no agendamento de eventos em outros LPs, o resultante é gerado aleatoriamente, podendo ocorrer maior geração em um LP, pode-se saber quantos eventos cada LP executou.

Os principais resultados do modelo são baseados em tempo fornecidos em segundos. No cômputo do tempo total da simulação, é levada em consideração apenas a quantidade decorrida após o início da execução dos eventos até o seu final. O tempo necessário para a carga dos eventos a partir da memória estável foi desconsiderado, uma vez que todos os eventos são lidos seqüencialmente por um LP.

Os tempos são extraídos do sistema usando a primitiva "`gettimeofday`", fornecida pela linguagem de programação C, que possui um tempo de execução de um microssegundo (μs) nas estações SUN onde foi realizada a implementação, ou seja, a cada chamada, ela consome 1μs para fornecer o resultado, dando o valor da hora atual que pode ser armazenada em uma estrutura de memória.

Para o cômputo do tempo de simulação, ao ser essa iniciada, a função "`gettimeofday`" é disparada com seu conteúdo armazenado; ao final, é realizado mais um armazenamento da hora atual; assim, o tempo de simulação é computado pela subtração da hora final pela inicial. Toda a vez que o simulador deixar de executar eventos e realizar funções de comunicação, será armazenada a hora atual antes e depois dessa tarefa, contabilizando o tempo que o simulador ficou com processamento ocioso na simulação. Tais valores serão acumulados, resultando no tempo total de processamento ocioso.

Após a explanação das principais particularidades do modelo, segue a descrição de um algoritmo básico de funcionamento do simulador (Figura 5.5).

```
barreira := lookahead
enquanto (EVL não vazia)
   se existem eventos <= barreira
   inicio
      executar eventos
      se evento agendar novo
      inicio
        geração do tempo para a execução do novo evento
        se criação for na própria EVL
           inserção na EVL
        senão
           preparação para inserção em outra EVL
      fim
      avança LVT
   fim
   senão
      se existem eventos na EVL
      inicio
         aux1 := MENOR_TRABALHANDO(LVT) + lookahead
         aux2 := barreira_atual + lookahead
         barreira := (aux1 < aux2, aux1, aux2)
         buscar novos eventos em outros nodos
         se existirem novos eventos em outros nodos
            inserir na EVL
       fim
fim_enquanto
verificação após o final
se existirem novos eventos
inicio
   inserir na EVL
   executar novos eventos
fim
```

FIGURA 5.5 – Algoritmo completo de funcionamento da implementação

Na Figura 5.5, foram mostradas apenas as linhas gerais do protocolo, o que possibilita uma visão completa da implementação. Expuseram-se todas as funções utilizadas, faltando apenas a visão global da implementação. Maiores detalhes sobre as porções de código utilizadas podem ser encontrados no capítulo 4.

5.5 Outros simuladores implementados

Para que pudesse ser realizada uma comparação entre o protocolo proposto e outros simuladores, dois simuladores usando técnicas tradicionais foram desenvolvidos. Em sua construção, diversas porções de código do simulador descrito foram aproveitadas, como o gerador da lista de eventos e o programa de divisão da mesma.

Um dos simuladores foi desenvolvido sem a exploração do paralelismo, tendo sido escrito todo o seu código na linguagem C, sem utilização de nenhuma outra biblioteca. Para sua execução, ele toma por base a lista de eventos antes de sua divisão, carregando-a toda em sua memória e executando os eventos. Uma modificação em relação ao simulador paralelo ocorre no momento do agendamento de novos eventos. Independentemente de o novo ser criado em outra lista, o simulador seqüencial sempre insere os eventos na própria EVL, mantendo, assim, o mesmo número de eventos

executados que o simulador paralelo com variáveis compartilhadas. O tempo envolvido em comunicações não foi extraído por motivos óbvios. À semelhança da implementação descrita anteriormente, o tempo de carga dos eventos a partir da memória estável também não foi considerado.

Além da comparação com um simulador seqüencial, um paralelo utilizando trocas de mensagens foi implementado com o uso da mesma biblioteca de comunicação. Utilizou-se um protocolo clássico de simulação paralela com uso de janelas de tempo conservadoras, o qual manteve a metodologia de cálculo do tempo de execução para um novo evento igual ao protocolo descrito no capítulo anterior. A escolha do LP (sorteio), no caso de a inserção não ser realizada na EVL própria, também permaneceu inalterada. As particularidades em relação à carga dos eventos e à execução com testes de erros LCC foram mantidas de maneira idêntica. Os tempos de processamento e ocioso também foram medidos, seguindo as mesmas características utilizadas no simulador com variáveis compartilhadas. A verificação realizada pelos LPs ao final da execução em busca de novos eventos a serem neles agendados não foi necessária em virtude da utilização de trocas de mensagens.

Uma descrição completa dos outros dois simuladores poderia ser fornecida, porém extrapola os limites e ambições do presente trabalho. Suas máquina de simulação são idênticas à do novo protocolo conservador descrito anteriormente, e, também, não simula nenhuma situação do mundo real, produzindo apenas resultados de tempo de simulação e de processamento ocioso.

5.6 Considerações finais

A implementação do protocolo de simulação paralela utilizando variáveis compartilhadas não foi uma tarefa trivial. Além de a programação utilizando *Athapascan0* ser desconhecida até o início da implementação, diversos outros problemas surgiram com a complexidade do protocolo proposto. Ressalta-se também que o manual do *Athapascan0* não proporciona informações claras sobre o uso do acesso remoto à memória. Sua metodologia de uso somente foi possível através de vários testes, fazendo uso de tentativa-e-erro para encontrar sua correta utilização.

As particularidades da implementação do modelo foram oriundas da forma de utilização da memória compartilhada e do excesso de zelo com o protocolo que está sendo descrito. Porém, após dominada a tecnologia para acessos remotos, a implementação das demais foi simples, tornando, assim , mais robusta e fornecendo maiores detalhes sobre os processos de simulação.

Em relação aos demais simuladores, o seqüencial teve sua implementação de maneira simples e direta; por sua vez, o paralelo com trocas de mensagens resultou em um código de difícil compreensão pela dificuldade de uma boa estruturação. Para amenizar a implementação do simulador paralelo baseado em MP, aproveitou-se grande parte do código utilizado em funções comuns ao simulador com variáveis compartilhadas.

6 Resultados

No capítulo anterior, descreveram-se a forma e as ferramentas utilizadas para a implementação do novo protocolo de simulação paralela com uso de variáveis compartilhadas. Também se apresentaram dois outros simuladores construídos, a serem utilizados em comparações. Neste capítulo, relatam-se os resultados obtidos pelos simuladores, juntamente com a metodologia utilizada para sua extração e os parâmetros de geração das listas de eventos utilizados.

6.1 Listas de eventos utilizadas

A fim de tornar possível a comparação entre os simuladores desenvolvidos, quatro grupos de listas de eventos foram criados e armazenados em memória estável, possibilitando, dessa forma, que, na execução dos simuladores, fossem realizadas as mesmas seqüências de eventos. Cada grupo de listas é composto de quatro listas, variando entre elas apenas a quantidade de eventos inicialmente gerados. A Tabela 6.1 mostra as listas de eventos geradas com seus parâmetros de criação.

TABELA 6.1 – Listas de eventos

Grupo de listas	Nome dos arquivos	Quantidade de eventos	Tempo mínimo e máximo dos evento (μs)	% Agendamento Próprio – outra
Listas00	ev0,5mil00	500.000	[5..10]*	[20..20]
	ev1mil00	1.000.000	[5..10]*	[20..20]
	ev1,5mil00	1.500.000	[5..10]*	[20..20]
	ev2mil00	2.000.000	[5..10]*	[20..20]
Listas01	ev0,5mil01	500.000	[5..30]	[80..50]
	ev1mil01	1.000.000	[5..30]	[80..50]
	ev1,5mil01	1.500.000	[5..30]	[80..50]
	ev2mil01	2.000.000	[5..30]	[80..50]
Listas02	ev0,5mil02	500.000	[5..30]	[20..20]
	ev1mil02	1.000.000	[5..30]	[20..20]
	ev1,5mil02	1.500.000	[5..30]	[20..20]
	ev2mil02	2.000.000	[5..30]	[20..20]
Listas03	ev0,5mil03	500.000	[5..30]	[0..100]
	ev1mil03	1.000.000	[5..30]	[0..100]
	ev1,5mil03	1.500.000	[5..30]	[0..100]
	ev2mil03	2.000.000	[5..30]	[0..100]

* Listas geradas com eventos variando em ms.

Todos os grupos foram criados com o mesmo número de listas, porém possuem nomes distintos, os quais seguem um padrão para facilitar a identificação: “ev” + [0,5 / 1 / 1,5 / 2] + “mil” + [grupo]. A segunda parte do nome dos arquivos faz referência à quantidade de eventos (apenas um dos valores é utilizado em cada nome), enquanto que a última identifica a que grupo ela pertence. Cada grupo possui um conjunto de quatro listas, cada uma delas com a mesma quantidade de eventos, variando apenas os parâmetros de criação do grupo. As listas começam com 500 000 eventos e crescem

neste valor até alcançarem o número de 2 000 000 eventos a serem inicialmente executados.

Os parâmetros de tempos mínimo e máximo dos eventos e também percentual de agendamento próprio e em outra lista variam apenas de um grupo de listas para outro. Os valores de tempos mínimo e máximo dos eventos são utilizados em microssegundos (μs), com exceção das listas00, cujos valores foram informados em milissegundos (ms). Todas as listas foram criadas com o intervalo de incremento do tempo de execução variando ente [0..1]. Esses parâmetros foram obtidos através de contatos com pesquisadores da área de simulação paralela ([LIN 99a], [LIN 99b], [PHA 99b], [PHA 2000a]).

Os percentuais de agendamento em EVL própria e de outros LPs também foram obtidos através de contatos com os mesmos pesquisadores na área supracitados. Apenas os valores das listas03 foram informados pelo autor, objetivando testar os simuladores no pior caso possível para as funções de criação de eventos em um LP que não o próprio.

6.2 Metodologia de obtenção dos resultados

A implementação dos simuladores foi testada num ambiente baseado em estações de trabalho ligadas através de uma rede de comunicação (vide *seção* 5.3). De forma semelhante, os testes obtidos pelos simuladores sobre as listas de eventos foram realizados no mesmo ambiente.

Na realização dos testes, os simuladores foram executados tendo como parâmetros uma lista de eventos, sua quantidade de eventos e um valor para o *lookahead*[16]. Um valor médio a ser utilizado foi obtido em contatos com pesquisadores ([LIN 99a], [PHA 2000b]), sendo 30 nas aplicações representadas pelas listas citadas. A fim de explorar a variação do *lookahead*, os simuladores também foram executados com valores maiores e menores, perfazendo execuções com *lookahead* 20, 30 e 40.

Como o meio de comunicação entre os processadores do ambiente pode influenciar nos resultados, dependendo do tráfego no momento da execução, todas as execuções foram realizadas isolando-se as máquinas do restante da rede. Dessa forma, conseguiu-se que, nas execuções, apenas as máquinas engajadas nas simulações estivessem se comunicando, o que foi obtido pela ligação das estações SUN em um *hub* separado, isolado de qualquer outra rede. Além do isolamento das comunicações, as estações, no momento das execuções, foram bloqueadas para que nenhum outro usuário pudesse iniciar processos durante a execução de uma simulação, resguardando as execuções para que o ambiente fosse o mesmo em todas as simulações.

Aliados a esses cuidados, os resultados produzidos pelos simuladores foram repetidos em séries de cinco execuções, sendo, após, extraída a média entre elas, a qual é a medida que será utilizada e discutida no restante do capítulo. Além da média, outras

[16] No simulador seqüencial, o *lookahead* foi mantido apenas por motivos de padronização com os outros visto que, em tal execução, ele não teve influência.

medidas foram extraídas das repetições a fim de preservar a veracidade dos resultados a serem apresentados.

6.3 Resultados obtidos

Duas medidas freqüentemente utilizadas nas aplicações paralelas em relação ao desempenho são o *speedup* e a eficiência. A Tabela 6.2 apresenta esses valores para os simuladores implementados, calculados a partir de execuções em três máquinas, extraídos dos simuladores paralelos em relação ao seqüencial, todos nos conjuntos de listas02.

TABELA 6.2 – *Speedup* e eficiência do grupos de listas 02

Eventos	Processos	Tempo		*Speedup*	Eficiência
		Seqüencial	Paralelo		
500 000	sm0,5mil20	3.817,59	2.576,80	1,4815	0,4938
	sm0,5mil30		2.245,43	1,7002	0,5667
	sm0,5mil40		2.069,90	1,8443	0,6148
	mp0,5mil20		1.940,62	1,9672	0,6557
	mp0,5mil30		1.819,89	2,0977	0,6992
	mp0,5mil40		1.736,08	2,1990	0,7330
1 000 000	sm1mil20	15.233,23	8.351,84	1,8239	0,6080
	sm1mil30		7.482,54	2,0358	0,6786
	sm1mil40		7.091,13	2,1482	0,7161
	mp1mil20		7.156,45	2,1286	0,7095
	mp1mil30		6.806,41	2,2381	0,7460
	mp1mil40		6.623,43	2,2999	0,7666
1 500 000	sm1,5mil20	34.222,92	17.303,37	1,9778	0,6593
	sm1,5mil30		15.720,34	2,1770	0,7257
	sm1,5mil40		15.010,28	2,2800	0,7600
	mp1,5mil20		15.823,29	2,1628	0,7209
	mp1,5mil30		14.988,37	2,2833	0,7611
	mp1,5mil40		14.756,84	2,3191	0,7730
2 000 000	sm2mil20	60.802,06	29.264,61	2,0777	0,6926
	sm2mil30		27.002,87	2,2517	0,7506
	sm2mil40		25.849,92	2,3521	0,7840
	mp2mil20		27.892,02	2,1799	0,7266
	mp2mil30		26.633,71	2,2829	0,7610
	mp2mil40		25.980,93	2,3403	0,7801

Na Tabela 6.2, os valores estão divididos pela quantidade de eventos, sendo demonstrados os do simulador com uso de memória compartilhada, identificados pelos processos que começam por “sm” , e os do simulador com uso de trocas de mensagens, iniciados por “mp”. A segunda parte do nome dos processos indica a quantidade de eventos, e a última porção, o valor do *lookahead* utilizado. A coluna de tempo seqüencial varia apenas de lista para lista, fazendo referência ao simulador seqüencial.

Os valores de *speedup* apresentam um mínimo de 1,4815 e um máximo de 2,3521. Analisados dentro de uma mesma lista, eles crescem à medida que o valor do *lookahead* utilizado também cresce. À medida que a quantidade de eventos aumenta, o *speedup* de todos os processos também aumenta, levando os processos com 2 000 000 de eventos a obterem os maiores valores.

A eficiência segue a variação do *speedup*, iniciando com um mínimo de 49,38% e crescendo até 78,40%, demonstrando que os processos paralelos gastam quantia significativa de tempo para operações de comunicação e sincronização. Como poderia ser esperado, os processos menos eficientes são aqueles com um menor número inicial de eventos, e, quando o número é o mesmo, os processos com o menor *lookahead* são também os de menor eficiência.

Como as medidas apresentadas na Tabela 6.2 descrevem os resultados de cinco repetições em cada processo, a Tabela 6.3 apresenta medidas complementares a fim de demonstrar as variações ocorridas na execução das simulações.

TABELA 6.3 – Resultados complementares de repetições de processos

Eventos	Execução	Média	Mínimo	Máximo	Desvio padrão	Relação
500 000	sm0,5mil20	2.576,80	2.562,00	2.619,97	24,49	0,95%
	sm0,5mil30	2.245,43	2.230,98	2.260,64	13,35	0,59%
	sm0,5mil40	2.069,90	2.051,33	2.086,66	14,99	0,72%
	mp0,5mil20	1.940,62	1.899,82	2.000,90	37,96	1,96%
	mp0,5mil30	1.819,89	1.802,83	1.834,07	13,98	0,77%
	mp0,5mil40	1.736,08	1.709,75	1.752,34	19,34	1,11%
	seque0,5mil	3.817,59	3.817,00	3.818,23	0,50	0,01%
1 000 000	sm1mil20	8.351,84	8.298,27	8.393,09	44,96	0,54%
	sm1mil30	7.482,54	7.411,06	7.586,03	72,99	0,98%
	sm1mil40	7.091,13	7.024,36	7.210,43	73,71	1,04%
	mp1mil20	7.156,45	7.070,07	7.294,05	87,63	1,22%
	mp1mil30	6.806,41	6.752,57	6.892,41	54,68	0,80%
	mp1mil40	6.623,43	6.569,66	6.717,08	62,92	0,95%
	seque1mil	15.233,23	15.227,99	15.241,14	4,99	0,03%
1 500 000	sm1,5mil20	17.303,37	17.166,56	17.607,43	179,12	1,04%
	sm1,5mil30	15.720,34	15.602,12	15.866,60	112,57	0,72%
	sm1,5mil40	15.010,28	14.937,05	15.139,27	84,42	0,56%
	mp1,5mil20	15.823,29	15.484,08	16.355,27	339,03	2,14%
	mp1,5mil30	14.988,37	14.803,91	15.119,49	132,95	0,89%
	mp1,5mil40	14.756,84	14.586,80	15.052,33	180,77	1,23%
	seque1,5mil	34.222,92	34.191,75	34.262,00	32,55	0,10%*
2 000 000	sm2mil20	29.264,61	28.914,88	29.583,98	250,77	0,86%
	sm2mil30	27.002,87	26.835,03	27.074,46	98,71	0,37%
	sm2mil40	25.849,92	25.752,76	26.013,71	100,58	0,39%
	Mp2mil20	27.892,02	27.507,25	28.246,89	267,45	0,96%
	Mp2mil30	26.633,71	26.466,34	26.863,02	203,41	0,76%
	Mp2mil40	25.980,93	25.809,28	26.239,31	176,28	0,68%
	Seque2mil	60.802,06	60.761,73	60.845,54	36,11	0,06%

* valor com quatro casas decimais 0,0951

A coluna de execução representa os processos de forma semelhante à Tabela 6.2, porém acrescenta uma linha em cada conjunto: o processo seqüencial, que é identificado pela inicial “seque” em seu nome. Para todas as repetições além da média, mostram-se os valores mínimo e máximo. Também se efetua o cálculo do desvio-padrão (S(X)) das repetições, tendo, na coluna relação, a identificação do que S(X) representa em relação à média em termos percentuais. Salienta-se que a coluna relação manteve valores inferiores a 0,10% em todos os processos seqüenciais. Nos processos paralelos, houve maiores variações, porém, na maioria dos casos, não ultrapassaram 1,25%, demostrando que os processos possuem uma relativa constância na sua execução, principalmente no simulador com memória compartilhada. Exceções ocorreram nos processos “mp0,5mil20” e “mp1,5mil20”, onde se podem observar variações de 1,96% e 2,14%, respectivamente.

A fim de garantir uma maior variedade nos processos de simulação, os simuladores também foram testados nos demais conjuntos de listas, porém as repetições não foram realizadas. A Tabela 6.4 representa os resultados das demais listas de eventos, com resultados apenas do simulador com memória compartilhada distribuída, executado em três processadores.

TABELA 6.4 – *Speedup* e eficiência do grupos de listas 00, 01 e 03

Grupo de Listas	Processos Nome	Tempo Seqüencial	Tempo Paralelo	Speedup	Eficiência
Listas ev00	sm0,5ml20	8.262,77	6.657,77	1,2411	0,4137
	sm0,5ml30		6.164,93	1,3403	0,4468
	sm0,5ml40		5.916,87	1,3965	0,4655
	sm1ml20	24.186,56	16.179,80	1,4949	0,4983
	sm1ml30		15.323,49	1,5784	0,5261
	sm1ml40		14.453,05	1,6735	0,5578
	sm1,5ml20	47.784,87	28.803,11	1,6590	0,5530
	sm1,5ml30		26.952,57	1,7729	0,5910
	sm1,5ml40		26.067,69	1,8331	0,6110
	sm2ml20	78.614,77	44.144,43	1,7809	0,5936
	sm2ml30		41.773,89	1,8819	0,6273
	sm2ml40		40.728,55	1,9302	0,6434
Listas ev01	sm0,5ml20	11.880,70	5.337,68	2,2258	0,7419
	sm0,5ml30		5.009,51	2,3716	0,7905
	sm0,5ml40		4.902,80	2,4232	0,8077
	sm1ml20	47.614,99	19.370,83	2,4581	0,8194
	sm1ml30		18.588,42	2,5615	0,8538
	sm1ml40		18.066,47	2,6355	0,8785
	sm1,5ml20	107.274,02	41.379,45	2,5924	0,8641
	sm1,5ml30		40.571,78	2,6441	0,8814
	sm1,5ml40		39.791,30	2,6959	0,8986
	sm2ml20	190.674,71	73.188,58	2,6053	0,8684
	sm2ml30		70.453,20	2,7064	0,9021
	sm2ml40		68.745,06	2,7736	0,9245
Listas ev03	sm0,5ml30	14.158,10	5.780,22	2,4494	0,8165
	sm1ml30	56.637,05	21.466,44	2,6384	0,8795
	sm1,5ml30	127.441,71	46.083,48	2,7655	0,9218
	sm2ml30	195.090,00	80.279,79	2,4301	0,8100

Nota-se, analogamente aos resultados do grupo de listas 02, que o *speedup* aumenta com os acréscimos no *lookahead*. De forma semelhante, no aumento da quantidade de eventos, o *speedup* também aumenta, com exceção do grupo de listas 03, no qual o maior valor encontra-se na lista com 1 500 000 de eventos. A eficiência varia de acordo com o *speedup*. Os dados da Tabela 6.4 dizem respeito apenas ao simulador paralelo com uso de variáveis compartilhadas, sendo os dados do simulador com trocas de mensagens demonstrados no Anexo A.

Uma comparação entre os resultados dos simuladores variando o *lookahead* pode ser visualizada na Figura 6.1, que foi dividida em quatro partes, cada uma representando os resultados em um mesmo número inicial de eventos. O simulador com uso de memória compartilhada é identificado por "Par_SM", e o que usa troca de mensagens, por "Par_MP". Os valores resultantes do programa de simulação seqüencial são mostrados com a legenda "Seqüencial".

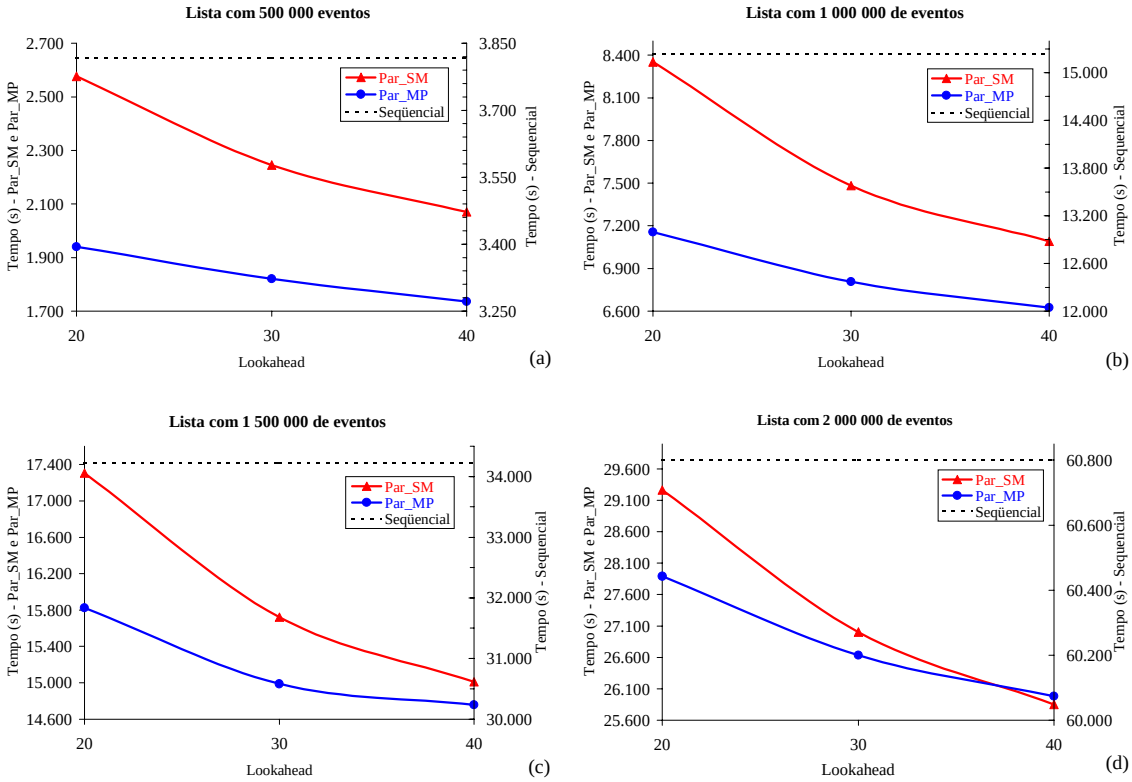


FIGURA 6.1 – Resultados entre simuladores com diferentes *lookahead*.

Para uma melhor visualização do tempo de simulação, exibido na Figura 6.1, esse foi representado por meio de dois eixos das ordenadas: um para os simuladores paralelos (eixo da esquerda) e outro para o simulador seqüencial (eixo da direta). Na representação gráfica dos resultados do simulador seqüencial, nota-se um único valor de tempo de simulação, não havendo variações com a alteração do *lookahead*.

A mudança do *lookahead* origina variações no tempo de execução dos simuladores paralelos. A Figura 6.1a ilustra as variações ocorridas em execuções de 500 000 eventos, tendo o simulador baseado em trocas de mensagens um menor tempo de execução para todos os valores de *lookahead* analisados. O mesmo ocorre com a lista de 1 000 000 de eventos, representada na Figura 6.1b, assim como na lista de 1 500 000, demonstrada na Figura 6.1c. Uma diferença entre as figuras citadas é que, à medida que

cresce o número de eventos e o *lookahead*, os tempos de execução dos simuladores paralelos gradualmente se aproximam.

A aproximação entre os valores dos simuladores paralelos, com o aumento do número de eventos e do *lookahead*, resulta na Figura 6.1d, que demonstra a evolução dos processos de simulação em listas com 2 000 000 de eventos. Mantendo a tendência da listas com menos eventos, as diferenças entre os valores de tempo de execução ficam cada vez menores, sendo que, na execução com *lookahead* 40, o simulador baseado em variáveis compartilhadas obtém um melhor desempenho[17] em relação ao simulador com trocas de mensagens. Essa situação foi alcançada em razão do beneficio proporcionado pelo novo protocolo de simulação, que é não bloquear um LP até que todos LPs alcancem a barreira de sincronização (*vide* seção 4.5.3); dessa forma, nos testes realizados com o maior número de eventos e executados com o mais elevado valor de *lookahead*, pôde tirar proveito o maior tempo possível desse benefício.

A quantidade de eventos executada por simulador em cada LP também foi verificada. Os processos de simulação oriundos de uma mesma lista de eventos devem executar a mesma quantidade de eventos. A Tabela 6.5 apresenta a quantidade de eventos executados pelos simuladores no grupo de listas 02.

TABELA 6.5 – Quantidade de eventos simulados no grupo de listas 02

Eventos	Programa	LP_0	LP_1	LP_2
500 000	Seqüencial	640 685		
	Paralelo SM	213 432	213 720	213 533
	Paralelo TM	213 432	213 720	213 533
1 000 000	Seqüencial	1 280 691		
	Paralelo SM	426 607	426 970	427 114
	Paralelo TM	426 607	426 970	427 114
1 500 000	Seqüencial	1 920 701		
	Paralelo SM	639 963	640 322	640 416
	Paralelo TM	639 963	640 322	640 416
2 000 000	Seqüencial	2 560 438		
	Paralelo SM	853 175	853 745	853 518
	Paralelo TM	853 175	853 745	853 518

A Tabela 6.5 mostra a quantidade dos eventos executados em cada LP nos processos de simulação paralelos e seqüencial. Uma ressalva é que, no processo seqüencial, não se faz necessária a divisão da execução em um ou outro LP, sendo atribuídos todos os eventos executados ao LP_0.

Na implementação dos simuladores, foi adicionado o total de eventos executados por LP, bem como se verificou que algum evento da EVL não tivesse sido executado. O resultado obtido é que todos os eventos contidos na EVL ao final da simulação foram executados em ambos os simuladores paralelos. A soma do total de eventos dos LPs nos simuladores paralelos confere com a quantidade de eventos manipulada pelo simulador seqüencial.

[17] Por melhor desempenho dos simuladores entende-se um menor tempo de execução do processo de simulação analisado.

Outro fato foi que, em cada execução da simulação de uma lista de eventos, independentemente de repetição ou simulador utilizado, a quantidade de eventos que cada um dos LPs executou permaneceu inalterada. Essa igualdade na distribuição deve-se ao método para escolha do LP em que vai ser agendado um novo evento, baseado na função "`random`", que não teve em cada execução uma inicialização da semente do gerador de números aleatórios (função *srand()*). Esse fato permitiu execuções idênticas em relação à quantidade de eventos de cada LP, possibilitando uma comparação sem que acúmulos de eventos em um LP pudessem acarretar divergências de tempo de execução de um ou outro simulador.

Um exemplo da distribuição dos eventos executados nos LPs pode ser visto em termos percentuais na Figura 6.2, que representa os dados referentes à lista com 1 500 000 eventos .

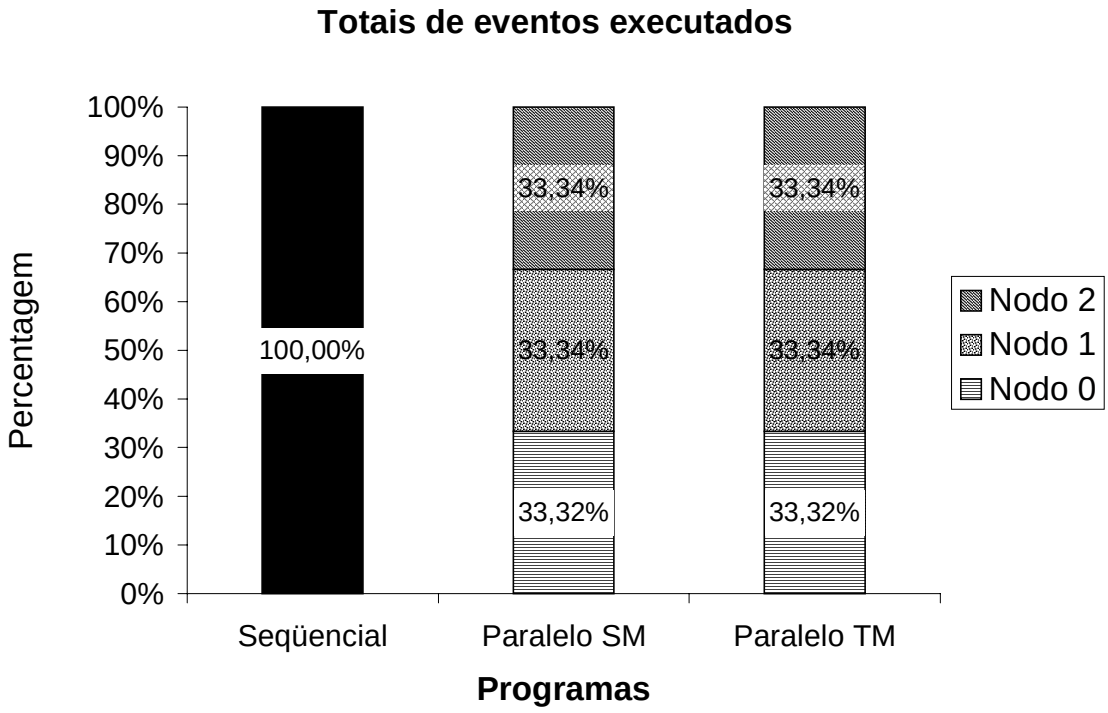


FIGURA 6.2 – Percentuais de eventos executados pelos nodos.

Foi constatado um equilíbrio em relação à quantidade de eventos executados por LP na simulação paralela. Todos os LPs da simulação paralela executaram praticamente a mesma fatia percentual de eventos, com alterações apenas na segunda casa decimal. Isso significa que, a partir de uma distribuição igual de eventos no processo de divisão da EVL, os LPs agendaram de maneira uniforme os eventos entre eles. Os dados referentes às demais listas de eventos estão no Anexo B.

Uma comparação direta entre os tempos de execução obtidos pelos simuladores pode ser visualizada na Figura 6.3, observando-se, para sua composição, um valor constante para o *lookahead*, variando apenas a quantidade inicial de eventos. A figura foi dividida em três partes, demonstrando em cada uma a variação dos tempos de simulação sobre as listas de eventos em um valor de *lookahead*; a identificação dos simuladores em toda a figura segue o padrão definido na Figura 6.1.

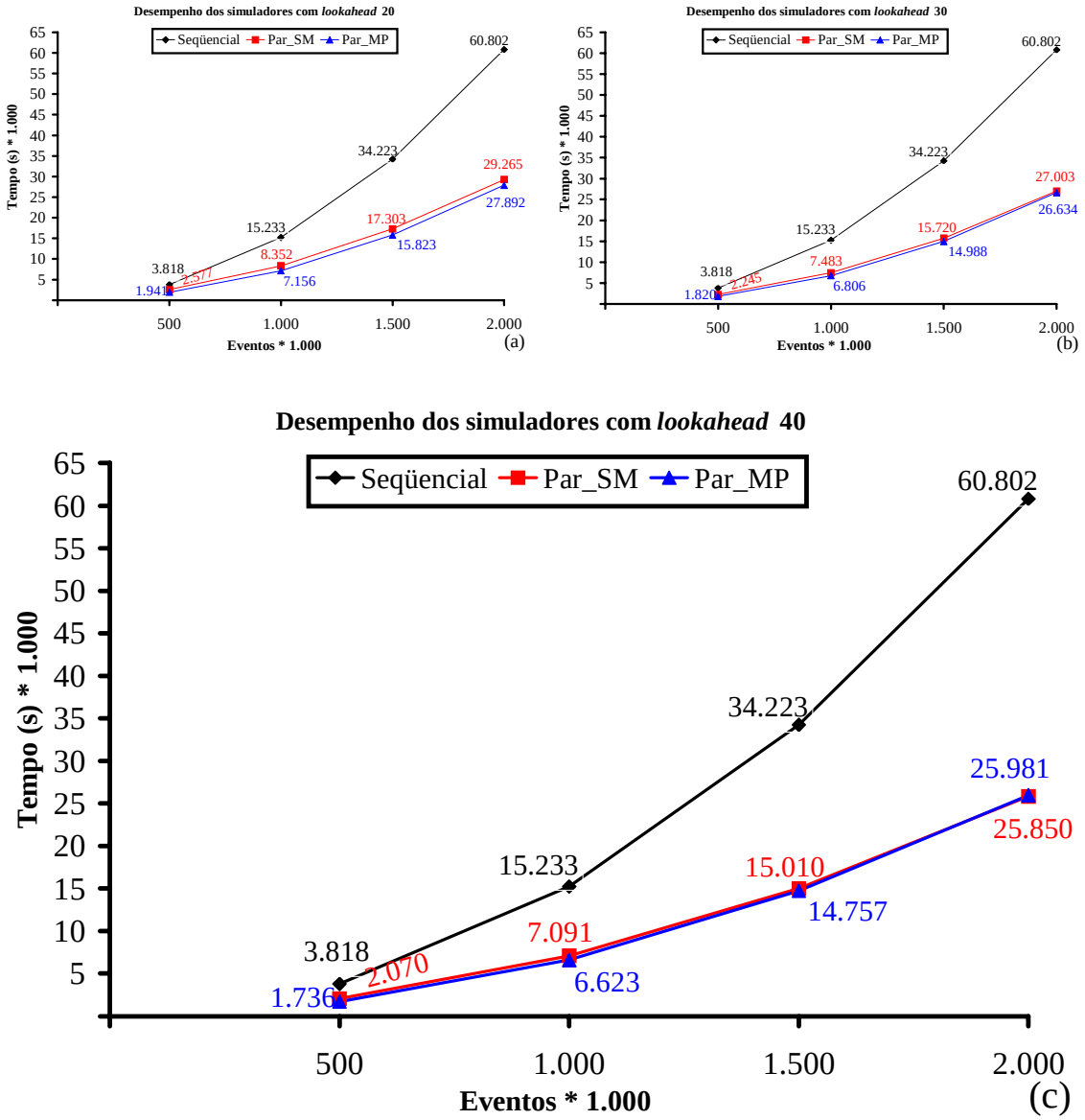


FIGURA 6.3 – Desempenho dos simuladores

Os resultados do simulador seqüencial aparecem inalterados em todas as partes da Figura 6.3, variando apenas o tempo gasto com cada uma das listas analisadas. Esse resultado deve-se ao fato de todos os eventos serem executados em um único processador, portanto o valor que um simulador pode "olhar à frente", em relação ao agendamento de novos eventos (*lookahead*), não proporciona variações no tempo de simulação.

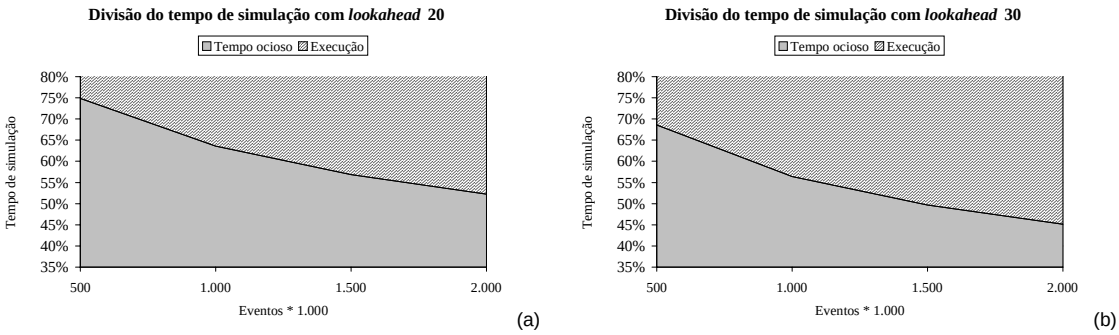
A Figura 6.3a ilustra o desempenho dos simuladores com o *lookahead* 20, constatando-se que o simulador com uso de trocas de mensagens tem um desempenho superior em todas as listas analisadas. As diferenças em relação ao simulador com variáveis compartilhadas mantiveram-se praticamente constantes em todas as listas de eventos.

Na Figura 6.3b, são mostrados os tempos de execução dos simuladores com *lookahead* 30. De maneira semelhante à Figura 6.3a, o simulador com uso de trocas de mensagens obteve menor tempo de execução em todas as listas analisadas. Uma

diferença em relação à Figura 6.3a é que, à medida que o número inicial de eventos aumenta, a diferença em relação ao simulador com variáveis compartilhadas diminui.

Os resultados das execuções com *lookahead* 40 são mostrados na Figura 6.3c. Nela, nota-se um desempenho semelhante de simuladores paralelos em todas as listas de eventos analisadas. Verifica-se uma pequena diferença nos tempos de simulação, tendo o simulador com uso de trocas de mensagens, nas listas de 500 000 até 1 500 000 de eventos, um desempenho levemente superior. Porém, diferentemente de todas as execuções com *lookahead* 20 e 30 e também das listas menores com *lookahead* 40, na lista com 2 000 000 de eventos, o simulador com variáveis compartilhadas obteve melhor desempenho do que o baseado em trocas de mensagens, fato que ocorreu devido ao novo protocolo de simulação conseguir tirar proveito da técnica de espiar o LVT dos LPs e não os bloquear ao chegar a uma barreira de sincronização (*vide* seção 4.5.3).

No tempo total da simulação, está computado o tempo ocioso dos simuladores, quando o LP pára de executar eventos e realiza tarefas de comunicação e/ou espera pelos demais LPs. Uma ilustração do percentual que representa o tempo ocioso nas simulações do simulador baseado em variáveis compartilhadas pode ser visualizada na Figura 6.4. O tempo de processamento ocioso foi definido na descrição do modelo (*vide* seção 4.6). Na implementação, cada vez que um LP alcança uma barreira de sincronização, ele pára de executar eventos e realiza funções de sincronização. Logo após encerrar a execução de eventos, a hora atual é armazenada, sendo, então, realizados acessos à memória compartilhada pertencente a outros LPs para conhecer o LVT e a quantidade de eventos a serem criados (*vide* seção 4.5.5). Logo que o LP encerrar suas atividades de sincronização e puder voltar à execução de eventos, novamente a hora atual é armazenada, produzindo o tempo ocioso da simulação através da subtração da primeira hora armazenada (antes de entrar na sincronização) pela última hora armazenada (antes de recomeçar a executar eventos). O processo, repetidamente, realiza sincronizações e todos os tempos de processamento ociosos são acumulados para, ao final, constituir o tempo total ocioso do processo de simulação (*vide* final da seção 5.4).



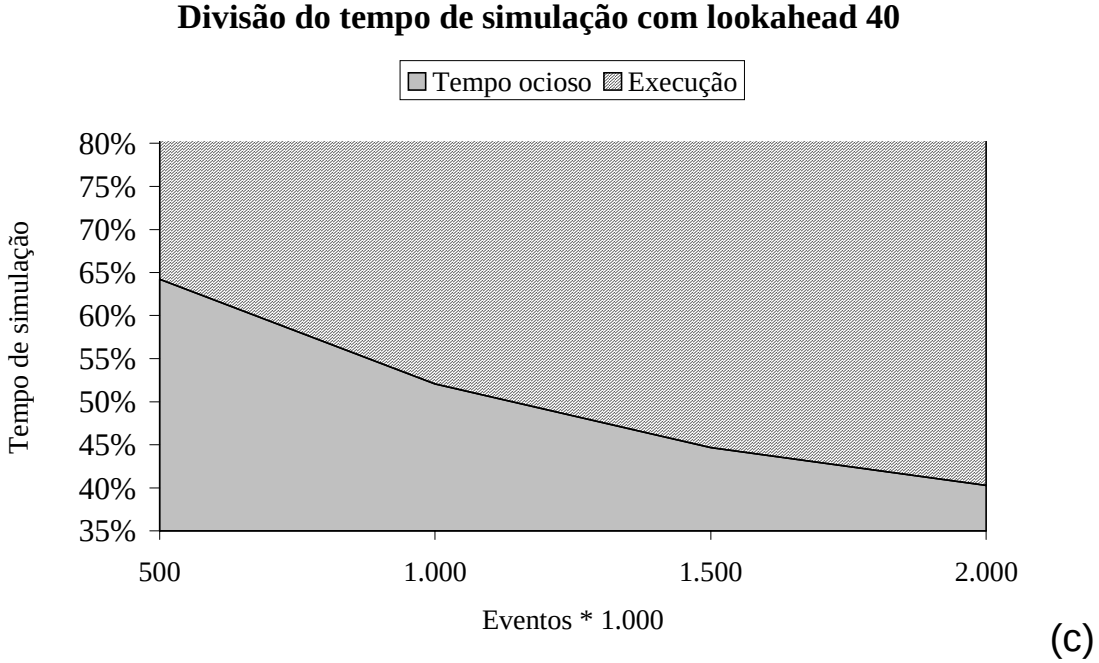


FIGURA 6.4 – Divisão de tempo dos processos de simulação

A Figura 6.4 foi dividida em três partes, cada uma delas representando os resultados de um único valor de *lookahead* aplicado nas listas que compõem o grupo de listas02. Todos os gráficos ilustram apenas a região percentual onde ocorrem as variações entre o tempo ocioso e o tempo de execução de eventos, observando-se em todos a mesma escala para proporcionar uma correta visualização.

Na Figura 6.4, os percentuais de tempo ocioso variaram entre 40,28% e 74,88%. Com o aumento do *lookahead*, o ociosidade dos processos diminuiu, aumentando, conseqüentemente, o percentual de execução de eventos. A Figura 6.4a, que representa os resultados com o menor *lookahead*, ilustra os maiores percentuais de ociosidade. Por outro lado, a Figura 6.4c, que representa o maior *lookahead* analisado, mostra os maiores percentuais de tempo de execução.

Ao analisar separadamente cada gráfico, onde variam apenas ao número de eventos, nota-se que, à medida que a quantidade de eventos a serem simulados aumenta, também cresce o percentual de tempo em que o simulador se dedica à execução. Em todas as partes da Figura 6.4, a lista com 500 000 eventos apresentou o maior percentual de tempo ocioso, diminuindo à medida que a quantidade de eventos aumentou, até alcançar o menor valor na lista, com 2 000 000 de eventos. A redução no tempo ocioso

não é gradual, notando-se uma abrupta queda inicial nas simulações com 1 000 000 de eventos em relação às com 500 000. Após essa redução inicial, duas outras de menor escala são percebidas, tendo-se, a cada aumento de eventos, uma menor redução de queda do percentual ocioso das simulações.

6.4 Considerações finais

Os processos de simulação executados pelo novo simulador com uso de variáveis compartilhadas, implementado em ambientes de memória distribuída, apresentaram uma execução completa dos modelos de simulação. Em todas as execuções, foram simulados todos os eventos contidos inicialmente nas listas de eventos, assim como todos os criados dinamicamente, tanto externos quanto internos (*vide* seção 4.5.1).

Em relação ao princípio operacional durante a execução, o novo simulador nunca, em nenhum caso simulado, permitiu a ocorrência de erros de causalidade local. Tal fato pode ser comprovado pela implementação, que a cada evento executado ou agendado na EVL, explicitamente, verificou sua ocorrência. Propostas de novos protocolos ou mesmo simulações específicas podem ser encontrados com tal restrição violada, mesmo quando utilizados mecanismos conservadores de simulação paralela.

O desempenho do simulador, implementado a partir da descrição do novo protocolo, foi inferior, na maioria dos casos, ao simulador tradicional implementado com trocas de mensagens. Tal fato pode ser justificado pela utilização de memória compartilhada distribuída, sabidamente de desempenho inferior a trocas de mensagens em ambientes distribuídos. Contudo, em alguns casos, o novo simulador apresentou desempenho semelhante ou, até mesmo, melhor, fornecendo uma nova possibilidade para tentativa de aceleração de processos de simulações paralelas.

7 Conclusões e próximos passos

Neste trabalho, apresentou-se um novo protocolo de simulação paralela com uso de variáveis compartilhadas, o qual foi implementado e testado em um ambiente de memória distribuída, fazendo uso de memória compartilhada distribuída (DSM). A simulação paralela com protocolos que utilizam memória compartilhada distribuída é uma área na qual se encontram poucos trabalhos, os quais, em sua quase totalidade, utilizam trocas de mensagens para a comunicação e sincronização dos LPs.

7.1 Conclusões

O princípio operacional do protocolo criado foi o da não-ocorrência de erros de causalidade local, característica dos protocolos conservadores de simulação paralela. Por outro lado, a forma de sincronização dos processos e o algoritmo de comunicação dos LPs utilizados não se basearam em nenhum outro. A especificação da interface de comunicação do protocolo foi desenvolvida com base em conceitos de simulação paralela, como *lookahead* e janelas de tempo, a partir do estudo do funcionamento de protocolos tradicionais, descritos com uso de trocas de mensagens.

Para a validação do protocolo criado, ele foi inteiramente implementado e testado com diversas listas de eventos, não tendo apresentado, em nenhuma das execuções, erros de causalidade local ou perda de eventos a serem simulados. A fim de analisar o seu desempenho, ele teve os seus resultados comparados com dois outros simuladores, implementados a partir de protocolos tradicionais descritos na literatura estudada. Um simulador seqüencial foi construído para verificação da quantidade de eventos executados e para que se pudesse medir a aceleração (*speedup*) e a eficiência do novo protocolo proposto. Também foi implementado um simulador paralelo com uso de trocas de mensagens a fim de propiciar uma comparação com o novo protocolo. Salienta-se que os três simuladores foram implementados com os mesmas ferramentas e testados num único ambiente computacional, fazendo-se simulações a partir de idênticas listas de eventos.

Ao simular corretamente os eventos contidos nas EVLs, pôde-se concluir que é possível a construção de protocolos de simulação paralela nos quais cada LP não possua acesso somente a sua memória local. O acesso a porções de memória compartilhada em outros LPs proporcionou uma implementação facilitada da interface de comunicação, obtida através do uso de memória compartilhada ao invés de trocas de mensagens.

O uso da mesma biblioteca para exploração do paralelismo nas implementações dos simuladores paralelos proporcionou uma comparação entre esses sem que questões relativas ao desempenho das bibliotecas pudessem influenciar nos resultados. Outro fator levado em conta é que ambos os simuladores paralelos utilizaram as mesmas técnicas para melhorar o paralelismo (*lookahead* e janela de tempo), implementadas de maneira idêntica. Tais fatos possibilitaram a comparação direta dos resultados entre os simuladores paralelos.

Aplicações que utilizam trocas de mensagens executadas em ambientes de memória distribuída obtêm um melhor desempenho em relação às que usam DSM ([LU 95], [LU 97]). A biblioteca *Athapascan0* foi testada durante os trabalhos em

programas nos quais cada nodo apenas realiza acessos nos demais, apresentando pior desempenho quando foram utilizados os recursos de variáveis compartilhadas (Anexo C). Ressalta-se que as diferenças entre as aplicações foram grandes, visto que em nenhum do casos testados, foram inferiores a 30% do tempo de execução.

Entretanto, quando os recursos de variáveis compartilhadas foram aplicados em conjunto com o novo protocolo de simulação descrito, os resultados de tempo de execução no novo protocolo proposto, em relação ao tradicionalmente implementado com trocas de mensagens, foram levemente diferentes. Como se poderia esperar, o simulador sem uso de memória compartilhada apresentou melhor desempenho na maior parte dos testes realizados. Os dois simuladores tiveram desempenhos semelhantes, com exceção das simulações com poucos eventos e um baixo valor de *lookahead*; nesses casos, o simulador com trocas de mensagens obteve significativa vantagem em relação ao baseado no protocolo proposto.

Um resultado que pode ser considerado como exceção entre os demais é o caso representado na lista com 2 000 000 de eventos do grupo 02, simulado com *lookahead* 40. Neste caso, o desempenho do novo simulador foi superior ao baseado em trocas de mensagens (Figura 6.1d e Figura 6.3c). A diferença entre os simuladores foi mínima, porém aponta uma nova opção para simulações paralelas de determinadas situações: utilizar o novo protocolo e produzir resultados mais rapidamente.

Uma diferença que pode ser constatada entre os simuladores é que o *lookahead* (e, em função dele, a barreira de sincronização) influenciou mais as simulações do novo protocolo do que as do tradicional. À medida que se aumenta o *lookahead*, os resultados de ambos os simuladores paralelos se aproximam. Tal fato originou-se de uma maior redução no tempo de simulação do novo simulador com o aumento do *lookahead* em relação ao simulador com trocas de mensagens, que, mesmo com uma janela de tempo maior, não obriga os LPs a ficarem esperando os demais ao chegarem a uma barreira de sincronização para prosseguir a simulação.

Um menor número de eventos a serem simulados também prejudica o desempenho do novo protocolo, que é mais bem adaptado às simulações com elevado número de eventos. Uma grande diferença na aceleração dos processos ocorre quando é aumentado pela primeira vez o número de eventos, de 500 000 para 1 000 000; contudo, a diferença não se mantém constante com os demais aumentos do número de eventos, encontrando diferenças muito pequenas entre processos com 1 500 000 e 2 000 000 de eventos.

A tendência de melhora do desempenho com aumento do *lookahead* e do número de eventos foi originada pela funcionalidade do novo protocolo, que fornece um novo intervalo de tempo para executar eventos seguros quando um LP chega a uma barreira de sincronização, sem que ele tenha de esperar pelos demais para seguir a simulação. Quanto maior for o valor do *lookahead*, maiores serão as janelas de tempo onde o LVT pode ser livremente incrementado e, conseqüentemente, maiores podem ser os tempos de ociosidade dos processadores com o simulador baseado no protocolo tradicional (Figura 2.3). Essa situação não acontece no protocolo proposto, no qual o LP fica esperando apenas a realização dos acessos remotos para prosseguir com a simulação. Quanto maior for a quantidade de eventos, mais o novo protocolo poderá se beneficiar dessa estratégia. Analisando essas situações, conclui-se que é viável a

utilização de DSM em simulações paralelas desde que o valor do *lookahead* possa ser alto e o simulador tenha de executar uma grande quantidade de eventos. Nesses casos, o protocolo consegue amenizar os lentos acessos à memória compartilhada e, ainda, produzir resultados rapidamente.

A principal contribuição deste trabalho consiste na especificação completa de um novo protocolo de simulação paralela, com seu mecanismo de sincronização baseado em variáveis compartilhas, implementado com uso de DSM, juntamente com os resultados produzidos e a comparação com outro protocolo de simulação paralela tradicionalmente implementado. Com o funcionamento do protocolo, é fornecida também a descrição das estruturas de memória utilizadas, tanto as compartilhadas quanto as locais, auxiliares no uso das estruturas compartilhadas. Tal especificação das estruturas de memória utilizadas não é geralmente encontrada em trabalhos da área, os quais apenas detalham o funcionamento dos protocolos, sem se preocupar como os algoritmos descritos devem ser modelados.

Outras importantes contribuições são: o algoritmo para acessos remotos somente através de leituras e a modelagem do gerador de listas de eventos. O algoritmo evita que aconteçam acessos remotos de escrita e libera o protocolo de controles que poderiam causar grande *overhead* de processamento. Tal algoritmo também pode ser utilizado em outras aplicações paralelas que façam uso de variáveis compartilhadas e necessitem de coleta periódica de informações em vários processadores. A modelagem do gerador de listas de eventos passou, primeiro, pela especificação dos dados necessários a uma implementação fiel de uma interface de comunicação. A estrutura de dados utilizada para a manipulação dos eventos e o gerador de listas podem ser utilizados para testes de um novo protocolo de simulação ou, mesmo, para comparações de protocolos tradicionais a fim de verificar o melhor/pior em simulações com características específicas.

7.2 Melhoramentos e trabalhos futuros

O armazenamento das listas de eventos pode sofrer melhoramentos a fim de diminuir a quantidade de memória estável necessária. A retirada de campos desnecessários, como os caracteres de *pipe* "|", pode reduzir o tamanho da lista de eventos; de forma semelhante, o número seqüencial do evento não é necessário e pode ser retirado. O formato de armazenamento dos arquivos, que é hoje baseado em texto, pode ser modificado, proporcionando também a redução no tamanho em *bytes* das listas.

A manipulação da EVL nos simuladores é outra parte sujeita a passar por estudos para possíveis melhoramentos. Nas implementações realizadas, a lista de eventos é mantida ordenada em uma variável dimensionada (um vetor da estrutura definida na Figura 4.5), sendo despendido muito tempo para a manipulação dos eventos. Assim, podem ser necessários novos estudos a fim de se encontrar melhores alternativas, ações essas que não foram realizadas neste trabalho por extrapolarem os objetivos para ele estipulados.

Como melhoramento ainda dos programas implementados, poderia ser construída uma interface gráfica que auxiliaria na execução das simulações, a qual seria

a mesma para todos os simuladores e poderia, em tempo de execução, mostrar aos usuários informações relativas ao avanço da simulação que está sendo executada.

Um dos pontos a serem trabalhados futuramente é a realização de mais testes do novo protocolo em ambientes com maior número de processadores e com uma melhor rede de conexão entre as estações de trabalho. As execuções ficaram limitadas a três LPs por ser o número de máquinas disponíveis, contudo seria interessante analisar sua execução num número maior de processadores. Testes com redes de comunicação de melhor desempenho também devem ser realizados para verificar o impacto da rede de comunicações do novo protocolo. A tendência é que, com uma rede comunicação mais eficiente, o novo protocolo consiga melhores resultados e possa aproximar-se ou, até mesmo suplantar, o desempenho do protocolo tradicional, baseado em trocas de mensagens, diminuindo, assim, o tempo de resposta das simulações.

A proposta inicial era a implementação de simulação paralela com uso de memória compartilhada distribuída, pelo fato de as redes de computadores formarem hoje a principal arquitetura paralela na maior parte dos laboratórios das instituições. Para a realização deste trabalho, um novo protocolo de simulação paralela foi desenvolvido, fazendo uso de variáveis compartilhadas. Um trabalho futuro poderia consistir na implementação do novo protocolo em um multiprocessador de memória centralizada e, se possível, na programação do protocolo com trocas de mensagens no mesmo ambiente, para que novas comparações possam ser efetuadas.

A implementação do protocolo utilizou apenas alguns recursos das variáveis compartilhadas, suportadas pela biblioteca *Athapascan0*. Uma alternativa que pode ser futuramente estudada é a implementação do protocolo descrito utilizando em cada LP duas *threads*: uma executaria a simulação enquanto a outra se comunicaria com os demais LPs através de trocas de mensagens, quando necessário. Outra possibilidade é implementação do protocolo utilizando ainda com DSM, porém com um ambiente de mais alto nível como a biblioteca de comunicações *Athapascan1*.

Anexo A – Resultados dos grupos de listas 00, 01 e 03

Este anexo é um subproduto do trabalho fornecido como um suplemento para busca de informações adicionais em relação aos resultados do simulador baseado em trocas de mensagens.

TABELA A.1 – *Speedup* e eficiência do grupo de listas 00, 01 e 03.

Grupo de Listas	Processos Nome	Tempo Seqüencial	Tempo Paralelo	Speedup	Eficiência
Listas ev00	mp0,5ml20	8.262,77	6.204,79	1,3317	0,4439
	mp0,5ml30		6.131,40	1,3476	0,4492
	mp0,5ml40		5.815,18	1,4209	0,4736
	mp1ml20	24.186,56	15.547,65	1,5556	0,5185
	mp1ml30		15.256,13	1,5854	0,5285
	mp1ml40		14.718,98	1,6432	0,5477
	mp1,5ml20	47.784,87	28.341,62	1,6860	0,5620
	mp1,5ml30		27.151,92	1,7599	0,5866
	mp1,5ml40		26.725,52	1,7880	0,5960
	mp2ml20	78.614,77	43.389,88	1,8118	0,6039
	mp2ml30		42.710,60	1,8406	0,6135
	mp2ml40		41.774,51	1,8819	0,6273
Listas ev01	mp0,5ml20	11.880,70	4.816,36	2,4667	0,8222
	mp0,5ml30		4.694,59	2,5307	0,8436
	mp0,5ml40		4.554,02	2,6088	0,8696
	mp1ml20	47.614,99	18.841,47	2,5271	0,8424
	mp1ml30		18.478,78	2,5767	0,8589
	mp1ml40		18.057,35	2,6369	0,8790
	mp1,5ml20	107.274,02	41.493,22	2,5853	0,8618
	mp1,5ml30		40.956,21	2,6192	0,8731
	mp1,5ml40		40.169,03	2,6706	0,8902
	mp2ml20	190.674,71	73.865,61	2,5814	0,8605
	mp2ml30		72.099,62	2,6446	0,8815
	mp2ml40		70.972,96	2,6866	0,8955
Listas ev03	mp0,5ml30	14.158,10	4.851,64	2,9182	0,9727
	mp1ml30	56.637,05	18.966,82	2,9861	0,9954
	mp1,5ml30	127.441,71	42.521,25	2,9971	0,9990
	mp2ml30	195.090,00	75.665,10	2,5783	0,8594

Anexo B – Totais de eventos executados

Em seqüência, apresentam-se as figuras que mostram os percentuais de eventos executados pelo simulador com uso de variáveis compartilhadas em listas com 500 000, 1 000 000 e 2 000 000 de eventos.

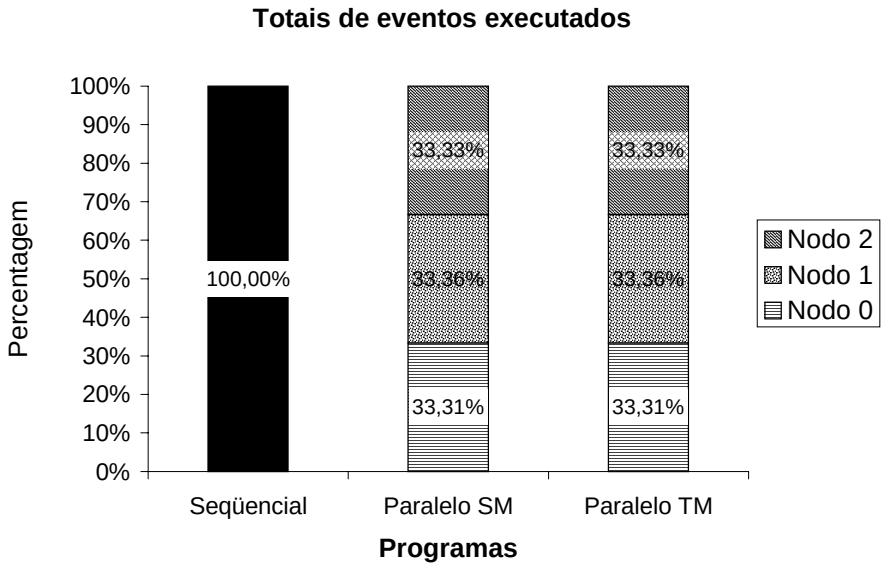


FIGURA B.1 – Percentuais de eventos executados pelos nodos em 500 000 eventos

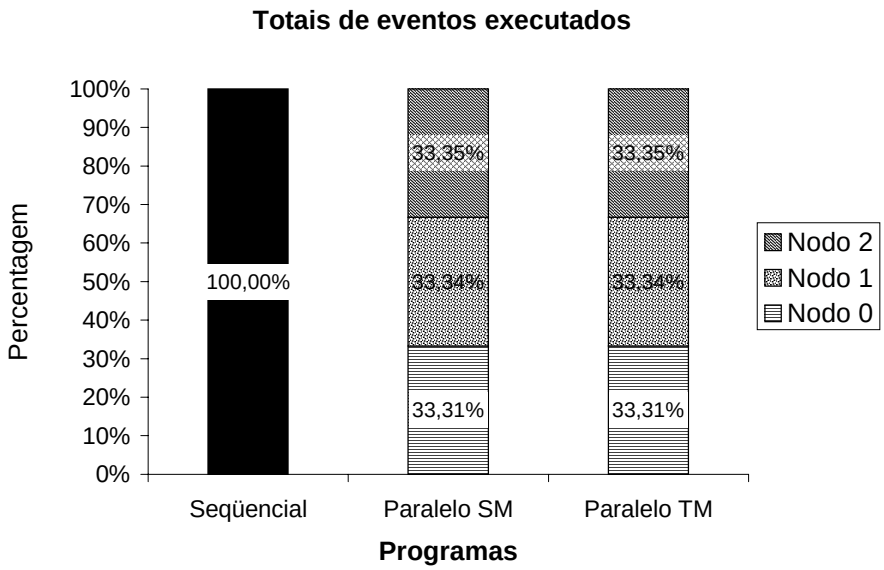


FIGURA B.2 – Percentuais de eventos executados pelos nodos em 1 000 000 eventos

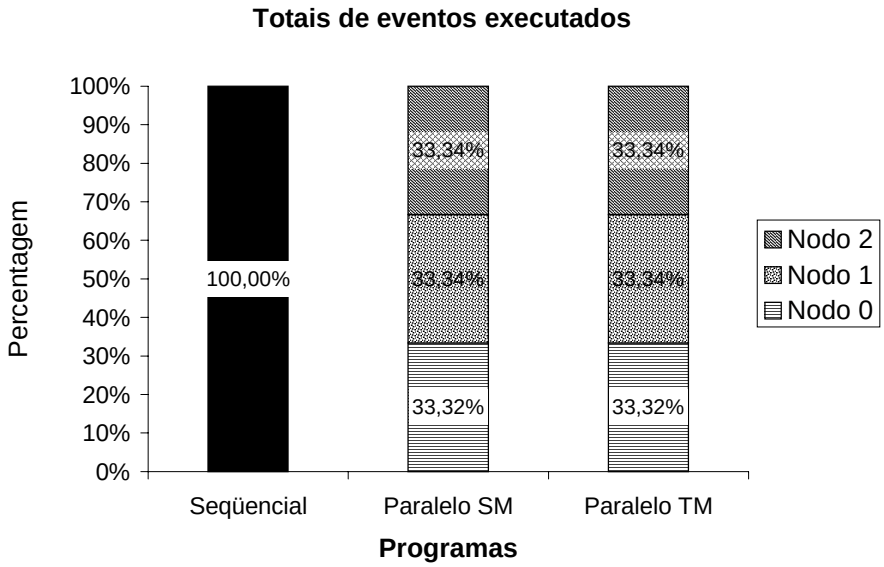


FIGURA B.3 – Percentuais de eventos executados pelos nodos em 2 000 000 eventos

Anexo C – Resultados em Athapascan0

TABELA C.1 – Medidas de desempenho entre MP e DSM em Athapascan0

Tipo	Acessos									
	5.000	10.000	15.000	20.000	25.000	30.000	35.000	40.000	45.000	50.000
2TM_r1	21,1178	40,2573	61,2646	92,4461	114,4210	124,9360	149,6850	174,7840	189,5980	206,4590
2TM_r2	22,6051	40,9340	60,6502	88,6485	107,1710	128,0940	147,8730	170,0660	195,4780	206,4940
2TM_r3	20,6056	40,2563	64,8424	90,6819	106,1800	129,3720	150,2030	177,8200	189,2930	211,0120
2TM_r4	21,0230	44,1800	63,2799	89,7073	109,0720	132,1120	147,7490	170,2640	191,5930	207,8860
2TM_r5	22,1480	38,4306	60,0274	85,4519	106,9250	128,2610	153,4230	177,9090	190,0140	206,9450
2TM	21,5799	40,8116	62,0129	89,3871	108,7540	128,5550	149,7860	174,1690	191,1950	207,7590
2DSM_r1	30,9897	62,5836	95,8749	129,7670	162,9190	184,9610	211,3580	249,9870	272,1910	304,0620
2DSM_r2	31,3414	63,2760	94,8371	129,0370	169,3590	177,5530	212,2830	247,5770	281,2930	313,1440
2DSM_r3	31,5578	61,2371	99,7202	127,8340	164,7630	180,9010	210,7790	247,0570	276,0820	304,8710
2DSM_r4	30,9075	60,9326	90,4946	129,3930	162,1100	178,4640	213,2840	248,2250	279,4180	315,7830
2DSM_r5	30,8237	61,6209	93,8579	125,2100	161,1660	178,7320	212,4090	245,8900	280,2670	305,9190
2DSM	31,1240	61,9299	94,9569	126,2480	164,0640	180,1220	212,0220	247,7470	277,8540	308,7560
3TM_r1	41,7576	85,6515	132,9690	175,1530	211,5750	253,4450	297,3090	365,2460	375,7820	427,0000
3TM_r2	45,8849	84,9991	128,5070	187,5840	209,9460	262,5710	295,2240	340,8490	382,5910	427,0420
3TM_r3	40,6470	85,2921	135,9070	179,6630	208,9170	267,0310	298,5210	344,1380	382,3930	422,4240
3TM_r4	44,0163	85,2696	132,7190	189,7150	212,7560	257,1180	296,6320	341,8540	395,3140	418,8290
3TM_r5	41,6636	86,4121	145,7940	176,3590	216,1120	265,8660	295,6010	345,6840	393,0240	416,9890
3TM	42,7939	85,5249	135,1790	181,6950	211,8610	261,2060	296,6570	347,5540	385,8210	422,4570
3DSM_r1	61,2482	120,3770	179,1050	234,6190	307,1480	370,0630	407,6400	452,3640	529,5410	607,9240
3DSM_r2	56,7983	116,1500	177,5860	235,8240	303,2790	364,9610	410,8640	462,0160	533,6510	605,3720
3DSM_r3	59,1622	117,1380	177,9860	239,1140	288,3940	367,6040	408,2480	452,2330	523,7960	596,7380
3DSM_r4	56,8668	118,6520	187,3190	240,4190	307,6940	368,0420	408,8710	450,9270	536,7270	584,4350
3DSM_r5	58,8490	116,5510	179,9060	237,8090	296,7540	372,0560	405,5300	479,5680	535,7570	593,2810
3DSM	58,5849	117,7740	180,3800	237,5570	300,6540	368,5450	408,2310	459,4210	531,8940	597,5500

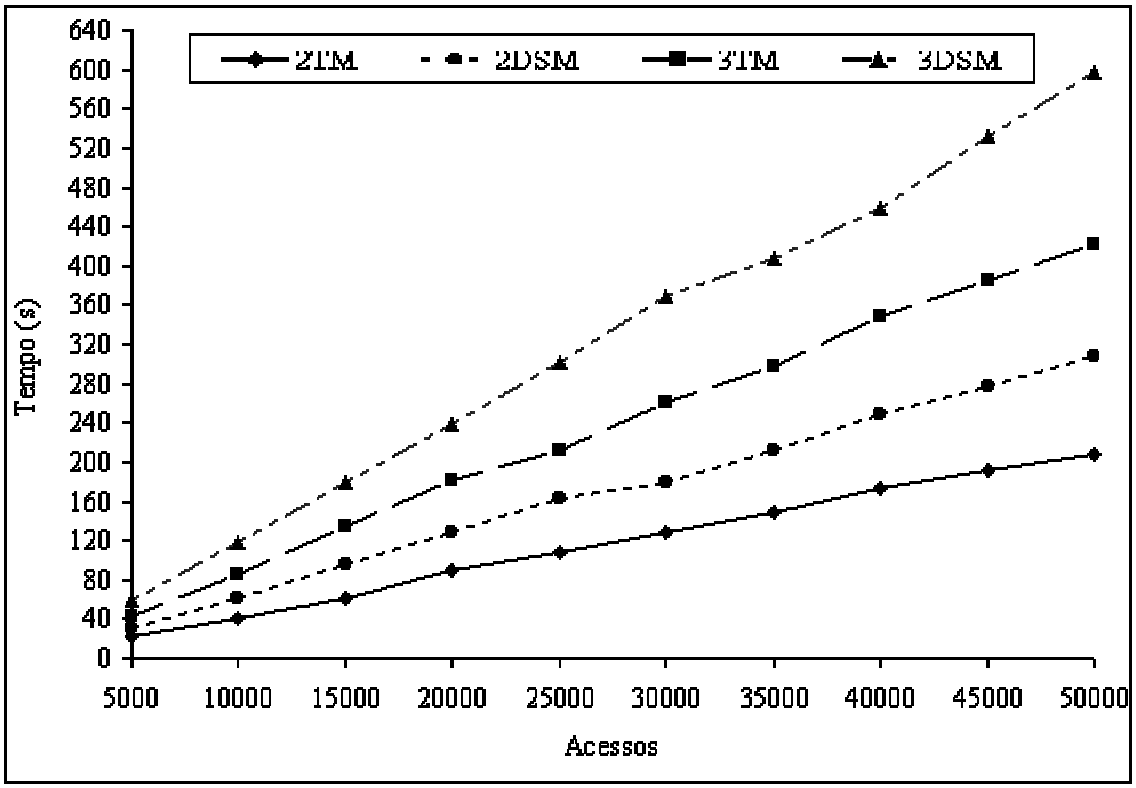


FIGURA C.2 – Desempenho entre trocas de mensagens e DSM em Athapascan0

Bibliografia

[ADV 99] ADVE, Sarita V. *et. al.* Recent Advances in Memory Consistency Models for Harware Shared Memory Systems. **Proceedings of the IEEE, Special Issue on Distributed Shared Memory**, New York, v. 87, n. 3, p. 445-455, Mar. 1999.

[ARA 99] ARAUJO, Edvar Bergmann. **Um estudo sobre Memória Compartilhada Distribuída**: trabalho individual. Porto Alegre: PPGC/UFRGS, 1999. 59p. (TI-868).

[AYA 92] AYANI, Rassul; RAJAEI, Hassan. Parallel Simulation using Conservative Time Windows. In: WINTER SIMULATION CONFERENCE, WSC, 1992, Arlington. **Proceedings...** San Diego: SCS, 1992. 1410p. p.709-717.

[BAL 95] BALDONI, Roberto *et al.* **Consistent Checkpointing in Message Passing Distributed Systems**. Rennes: Université de Rennes/Institut de Recherche en Informatique et Systèmes Aléatoires, 1995. (Publication Interne 925). 25p.

[BAR 96] BARBOSA, Valmir C. Simulation. In: **An introduction to distributed algorithms**. Cambridge: MIT,c1996. 365p. chap.10, p.291-321.

[BRI 98a] BRIAT, Jacques; GINZBURG, Ilan; PASIN, Marcelo. **Athapascan-0 User Manual**. Grenoble: LMC-IMAC, 1998. 17p. Disponível em: <http://www-apache.imag.fr/software/ath0/manuals.html>. Acesso em: 19 mar. 1999.

[BRI 98b] BRIAT, Jacques; GINZBURG, Ilan; PASIN, Marcelo. **Athapascan-0 Reference Manual**. Grenoble: LMC-IMAC, 1998. 82p. Disponível em: <http://www-apache.imag.fr/software/ath0/manuals.html>. Acesso em: 19 mar. 1999.

[CAL 99] CALEGARIO, Vanusa Menditi. **Análise de Desempenho de Programação em Lógica**. Rio de Janeiro: COPPE/UJRJ, 1999. 112p. Dissertação de Mestrado.

[CAM 96] CAMPOS, Alvaro E.; ASTABURUAGA, Miguel Angel Castillo. **Checkpointing Through Garbage Collection.** 1996. Disponível em: <http://alesna.ing.puc.cl/~mca/chicago/paper.html>. Acesso em: 05 out. 1998.

[CHA 83] CHANDY, K. M.; MISRA, J.; HASS, L.M. Distributed Deadlock Detection. **ACM Transactions on Computer Systems**, New York, v.1, n.2, p.144-156, May 1983.

[CON 97] CONCEIÇÃO, Sérgio Ricardo da. **Um Protocolo para Rastreamento de Mensagens em Sistemas com Checkpointing Assíncrono**. 1997. Disponível em: <http://www.ime.usp.br/dcc/posgrad/testes/ric>. Acesso em: 03 out. 1998.

[COP 96] COPSTEIN, Bernardo; PEREIRA, Carlos E; Wagner, Flavio R. The Object-Oriented Approach and the Event Discrete Simulation Paradigms. In: EUROPEAN SIMULATION MULTICONFERENCE, 1996, Budapest. **Proceedings...** San Diego: SCS, 1996. 1137p. p.57-61.

[CUL 99] CULLER, David E.; SINGH, Jaswinder P.; GUPTA, Anop. **Parallel Computer Architecture:** a hardware and software approach. San Francisco: Morgan Kaufmann, 1999. 1025p.

[DAS 94] DAS, Samir *et al*. GTW: A Time Warp System for Shared Memory Multiprocessors. In: WINTER SIMULATION CONFERENCE, WSC, 1994, Lake Buena Vista. **Proceedings...** San Diego: SCS, 1994. 1500p. p.1332-1339.

[FEL 94] FÉLIX, Cláudio Antônio. **Sistemas de Memória Compartilhada Distribuída:** Um Estudo Comparativo: trabalho individual. Porto Alegre: CPGCC/UFRGS, 1994. 53p. (TI-442).

[FER 96] FERSCHA, Alois. Parallel and Distributed Simulation of Discrete Event Systems. In: ZOMAYA, Albert Y. H. **Parallel and Distributed Computing Handbook.** New York: McGraw-Hill, 1996. 1199p. chap.35, p.1003-1041.

[FUJ 90] FUJIMOTO, Richard M. Parallel Discrete Event Simulation. **Communications of the ACM,** New York, v. 33, n. 10, p. 31-53, Oct. 1990.

[FUJ 92] FUJIMOTO, Richard; NICOL, David. State of Art in Parallel Simulation. In: WINTER SIMULATION CONFERENCE, WSC, 1992, Arlington. **Proceedings...** San Diego: SCS, 1992. 1410p. p.246-254.

[FUJ 2000] FUJIMOTO, Richard M. **Parallel and Distributed Simulation Systems**. New York: John Wiley & Sons, 2000. 300p.

[GEO 99] GEOFFRAY, Patrick *et al*. High-Speed LANs: New Environments for Parallel and Distributed Applications. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, PDPTA, 1999, Las Vegas. **Proceedings...** Las Vegas:[s.n.], 1999.

[GEY 98] GEYER, Cláudio Fernando Resin. **Programação paralela e distribuída**. Porto Alegre: CPGCC/UFRGS, 1998. Notas de Aula.

[HOA 78] HOARE, Charles Antony Richard Communicating Sequential Processes. **Communications of the ACM,** New York, v. 8, n. 21, p. 666-677, 1978.

[HIR 97] HIRATA C.; KRAMER J. On the Optimisation of Shared Variables in Time Warp. In: EUROPEAN SIMULATION SYMPOSIUM, 9., 1997, Passau. **Simulation in Industry.** Ghent: SCS, 1997. 755p. p.264-268.

[HWA 93] HWANG, Kai. **Advanced Computer Architecture:** Parallelism, Scalability, Programmability. New York: McGraw-Hill, Inc. 1993. 771p.

[HWA 98] HWANG, Kai.; XU, Zhiwei. **Scalable Parallel Computing**. Boston: McGraw-Hill, WCB, 1998. 802p.

[IKO 98] IKONEN, Jouni; PORRAS, Jari; HARJU, Jarmo. Analyzing distributed simulation. In: INTERNATIONAL CONGRESS OF THE FEDERATION OF EUROPEAN SIMULATION SOCIETIES - EUROSIM, 3., 1998, Helsinki. **Proceedings...** Helsinki: SIMS, 1998. p.33-38.

[IKO 99] IKONEN, Jouni; PORRAS, Jari. Load Balancing in conservative simulation. In: EUROPEAN SIMULATION SYMPOSIUM, 11., 1999, Erlangen. **Proceedings...** Ghent: SCS, ASIM, 1999. 744p. p.250-257.

[JAL 94] JALOTE, Pankaj. Recovering a Consistent State. In: **Fault Tolerance in distributed systems.** Englewood Cliffs: Prentice Hall, 1994. 432p. p.185-215.

[JOH 93] JOHNSON, David Bruce. Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 12., 1993. **Proceedings...** New York: IEEE Computer Society, 1993.

[KEL 96] KELLER, Jörg; RAUBER, Thomas; REDERLECHNER, Bernd. Conservative Circuit Simulation on Shared-Memory Multiprocessors. In: WORKSHOP ON PARALLEL AND DISTRIBUTED SIMULATION, 10., 1996, Philadelphia. **Proceedings...** Los Alamitos: IEEE Computer Society Press, ACM, SCS, 1996. 207p. p.126-134.

[KON 97] KONTOTHANASSIS, Leonidas *et al*. VM-based Shared Memory on Low-Latency Remote-Memory-Access Networks. **SIGARCH Computer Architecture News**, New York, v.25, n.2, p.157-169, May 1997. Trabalho apresentado no Annual International Symposium on Computer Architecture, 24., 1997.

[KUM 94] KUMAR, V. *et al*. Models of Parallel Computers. In: **Introdutction to Parallel Computing:** Design and Analysis of Algortithms. Redwood City: The Benjamin/Cummings Publishing Company, 1994. 597p. chap. 2, p.15-64.

[LEE 98] LEE, Jae Bum. Reducing Coherence Overhead of Barrier Synchronization in Software DSMs. In: THE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS, 1998, Orlando. **Proceedings...** New York: ACM Press and IEEE Computer Society Press, 1998.

[LIN 99a] LINCOLN, Patrick. **Parallel simulation:** real values for cpu time and others variables. Disponível em: <lincoln@csl.sri.com>. Acesso em: 27 dez. 1999.

[LIN 99b] LINCOLN, Patrick. **Parallel simulation:** real values for cpu time and others variables. Disponível em: <lincoln@csl.sri.com>. Acesso em: 28 dez. 1999.

[LU 95] LU, Honghui *et al*. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In: THE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS, 1995, San Diego. **Proceedings...** New York: ACM Press and IEEE Computer Society Press, 1995.

[LU 97] LU, Honghui *et al*. Quantifying the Performance Differences Between PVM and TreadMarks. **Journal of Parallel and Distributed Computation**, Orlando,v.43, n.2, p.65-78, June 1997.

[LUM 97] LUMPP, James E. **Checkpointing and Rollback for Distributed Applications**. 1997. Disponível em: <http://www.dcs.uky.edu/~jeldemo/chkpt.html> Acesso em: 30 set. 1998.

[LYN 96] LYNCH, Nacy A. Modelling III: Asynchronous Shared Memory Model. In: **Distributed Algorithms**. San Francisco: Morgan Kaufmann, 1996. 872p. chap. 9, p.239-253.

[MAC 96] MACKENZIE, Peter A.; TROPPER, Carl. Parallel Simulation of Billiard Balls using Shared Variables. In: WORKSHOP ON PARALLEL AND DISTRIBUTED SIMULATION, 10., 1996, Philadelphia. **Proceedings...** Los Alamitos: IEEE Computer Society Press, ACM, SCS, 1996. 207p. P.190-195.

[MEH 93] MEHL, Horst; HAMMES, Stefan. Shared Variables in Distributed Simulation. In: PARALLEL AND DISTRIBUTED SIMULATION, 7., 1993, San Diego. **Proceedings...**. San Diego: ACM/SCS/IEEE, 1993. 168p. p.68-75.

[MIL 96] MILUTINOVIC, Veljko. Some Solutions for Critical Problems in Distributed Shared Memory: New Ideas to Analyze. **IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter**, [S.l.], p.1-6, September 1996. Disponível em: <http://www.computer.org/tab/tcca/news/sept96/sept96.htm>.

[MUK 97] MUKHERJEE, Shubhendu S. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulato. In PAID, 1997. **Proceedings …** Disponível em: <http://cs.wisc.edu/~shubu/papers.html#survey>. Acesso em: 22 mar. 1999.

[MUS 99] MUSSI, Philippe. Parallel Discrete Event Simulation. In: INTERNATIONAL SCHOOL ON ADVANCED ALGORITHMIC TECHNIQUES FOR PARALLEL COMPUTATION WITH APPLICATIONS, 1999, Natal. **[Papers]**. [Natal: s.n.], 1999.

[NKE 98] NETSA, Alexandre; KHALIFA, Nabil Ben. Static and dynamic lookahead computation for conservative distributed simulation of Timed Petri nets. In: EUROPEAN SIMULATION SYMPOSIUM, 1998, Nottingham. **Simulation Technology:** Science and Art. Ghent: SCS, 1998. 766p. p.195-199.

[NOG 98] NOGUEIRA, Mauro Lúcio Baioneta. **Um ambiente para avaliação de políticas de balanceamento de carga**. Porto Alegre: CPGCC/UFRGS, 1998. Dissertação de Mestrado.

[OVE 91] OVEREINDER, Benno; HERTZBERGUER, Bob; SLOOT, Peter. Parallel Discrete Event Simulation. In: THE WORKSHOP COMPUTERSYSTEMS, 3., 1991, Eindhoven. **Proceedings...** [S.l.]: W. J. Withagen, 1991. p.19-30.

[PAS 98] PASIN, Marcelo. **Athapascan-0 Formats User and Reference Manual**. Grenoble: LMC-IMAC, 1998. 61p. Disponível em: <http://www-apache.imag.fr/software/ath0/manuals.html>. Acesso em: 19 mar. 1999.

[PEN 98] PENIX, John *et al*. Experiences in Verifying Parallel Simulation Algorithms. In: WORKSHOP ON FORMAL METHODS IN SOFTWARE PRACTICE, 2., 1998. **Proceedings...** New York: ACM Press,1998. p.16-23.

[PHA 99a] PHAM, CongDuc. High performance clusters: A promising environment for parallel discrete event simulation. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, PDPTA, 1999, Las Vegas. **Proceedings...** Las Vegas:[s.n], 1999. Disponível em: <http://resam.univ-lyon1.fr/PUBLICATIONS/>. Acesso em: 23 mar. 1999.

[PHA 99b] PHAM, CongDuc. **Parallel simulation:** real values for cpu time and others variables. Disponível em: <cpham@lhpca.univ-lyon1.fr>. Acesso em: 29 dez. 1999.

[PHA 2000a] PHAM, CongDuc. **Parallel simulation:** real values for cpu time and others variables. Disponível em: <cpham@lhpca.univ-lyon1.fr>. Acesso em: 06 jan. 2000.

[PHA 2000b] PHAM, CongDuc. **Parallel simulation:** real values for cpu time and others variables. Disponível em: <cpham@lhpca.univ-lyon1.fr>. Acesso em 07 jan. 2000.

[POR 97a] PORRAS, Jari *et al*. Improving the Performance of the Chandy-Misra Parallel Simulation Algorithm in a Distributed Workstation Environment. In: SUMMER COMPUTER SIMULATION CONFERENCE, SCSC, 1997, Arlington. **Proceedings...** Arlington:[s.n.], 1997. p.657-662. Disponível em: <http://www.lut.fi/~trappis/list.html>. Acesso em: 23 mar. 1999.

[POR 97b] PORRAS, Jari *et al*. Reducing the Number of Logical Channels in the Chandy-Misra Algorithm. In: EUROPEAN SIMULATION SYMPOSIUM, 9., 1997, Passau. **Simulation in Industry.** Ghent: SCS, 1997. 755p. p.252-256.

[POR 98] PORRAS, Jari; IKONEN, Jouni; HARJU; Jarmo. Applying a Modified Chandy-Misra Algorithm to the Distributed Simulation of a Cellular Network. In: WORKSHOP ON PARALLEL AND DISTRIBUTED SIMULATION, 12., 1998, Alberta. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998. 197p. p.188-195.

[POR 99] PORRAS, Jari; IKONEN, Jouni. Approaches to the analysis of ditributed simulation. In: EUROPEAN SIMULATION SYMPOSIUM, 11., 1999, Erlangen. **Proceedings...** Ghent: SCS, ASIM, 1999. 744p. p.551-555.

[REB 99a] REBONATTO, Marcelo Trindade. **Um estudo sobre simulação paralela de trânsito**: trabalho individual. Porto Alegre: PPGC/UFRGS, 1999. 72p. (TI-826).

[REB 99b] REBONATTO, Marcelo Trindade. Simulação paralela de eventos discretos em DSM. In: SEMANA ACADEMICA DO PPGC, 4., Porto Alegre, 1999. **Anais...** Porto Alegre: PPGC/UFRGS, 1999. 391p. p.87-90.

[REI 2000a] REINHARDT, Steven K. **Parallel simulation:** real values for cpu time and others variables. Disponível em: <stever@eecs.umich.edu>. Acesso em: 03 jan. 2000.

[REI 2000b] REINHARDT, Steven K. **Parallel simulation:** real values for cpu time and others variables. Disponível em: <stever@eecs.umich.edu>. Acesso em: 03 jan. 2000.

[RIC 95] RICCIULLI, Livio. **A Technique for the Distributed Simulation of Parallel Computers**. 1995. Disponível em: <http://www.csl.sri.com/~livio/papers.html>. Acesso em 05 jun. 1999.

[RIC 97] RICKERT, Marcus. **Traffic simulation on distributed memory computers**. Cologne: Faculty of Mathematics and Natural Science of University of Cologne, 1997. 189p. PhD Thesis.

[ROD 99] RODMAN, Andreas; BRORSSON, Mats. Programming Effort vs. Performance with a Hybrid Programming Model for Distributed Memory Parallel Architectures. In: INTERNATIONAL EURO-PAR CONFERENCE, 5., 1999, Toulouse. **Proceedings ...** Berlin: Springer, 1999. 1503p. p.888-898.

[SAT 96] SATO, Liria Matsumoto; MIDORIKAWA, Edson Toshimi; SENGER, Hermes. **Introdução a Programação Paralela e Distribuída**. 1996. Disponível em: <http://www.lsi.usp.br/~liria/jai96/apost.ps>. Acesso em: 07 maio 1999.

[SEI 98] SEIDEL, Cristiana Bendes. **A Técnica Lock Acquirer Prediction e sua aplicação em Sistemas de Memória Compartilhada Distribuída**. Rio de Janeiro: COOPE/UFRJ, 1998. 141p. Tese de Doutorado.

[SHA 92] SHANNON, Robert E. Introduction to Simulation. In: WINTER SIMULATION CONFERENCE, 1992, Arlington. **Proceedings...** San Diego: SCS, 1992. 1410p. p.65-73.

[SIL 98] SILVA, Ermes Medeiros da *et al*. Simulação. In: **Pesquisa Operacional.** São Paulo: Atlas, 1988. 184p. cap. 9, p.143-155.

[SIL 99] SILVA, Márcio Gonçalves da. **Influência de Parâmetros Arquiteturais em Sistemas Paralelos de Programação em Lógica**. Rio de Janeiro: COPPE/UJRJ, 1999. 101p. Dissertação de Mestrado.

[SIV 94] SIVILOTTI, Paul A. G.; CARLIN, Peter A. **A Tutorial for CC++**. Pasadena, California: Department of Computer Science, California Institute of Technology, 1994. 98p. (Caltech CS-TR-94-02).

[SOA 90] SOARES, Luiz Fernando Gomes. **Modelagem e simulação discreta de sistemas**. São Paulo: IME-USP, 1990. 250p.

[TAN 95] TANENBAUM, Andrew S. Distributed Shared Memory. In: **Distributed Operating Systems**. Upper Saddle River: Prentice-Hall, 1995. 614p. chap. 6, p.289-375

[TIB 96] TIBAUT, Andrej. **Parallel traffic simulation with an algorithm for dynamic load-balancing**. Nathan, Australia: School of Computing and Information Technology (CIT), Griffith University; Maribor, Slovenia: Faculty of Civil Engineering, 1996. 21p. Research Report number CIT-96-02. Disponível em: <http://www.cit.gu.edu.au/research/reports/postscript/CIT-96-02.ps>. Acesso em: 04 dez. 1988.

[VAC 99] VACCARO, Guilherme Luis Roehe. **Simulacao Paralela e Distribuida com vistas ao co-Design**: trabalho individual. Porto Alegre: CPGCC/UFRGS, 1999, 50p. (TI-811).

[WIL 96] WILSON, Gregory V.; LU, Paul. **Parallel Programming Using C++** . Cambridge: MIT, 1996. 758 p.

[XU 93] XU, Jian; NETZER, Robert H. B. Adaptative Independent Checkpointing for Reducing Rollback Propagation. In: SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, 5., 1993, Dallas. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1993. p. 754-761.

[ZOM 96] ZOMAYA, Albert Y. Parallel and Distributed Computing: The Scene, the Props, the Players. In: **Parallel and Distributed Computing Handbook**. New York: McGraw-Hill, 1996. 1199p. chap.1, p.5-23.