SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 02.12.2003

Assinatura:

**Avaliação e aprimoramento de uma
implementação para simulação
distribuída conservativa visando
utilização em um ambiente automático**

Eliane Sayuri Tatsumi

Orientador: *Prof. Dr. Regina Helena Carlucci Santana*

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP – São Carlos
Dezembro/2003

A Comissão Julgadora:

Profa. Dra. Regina Helena Carlucci Santana Regina HC Santana

Prof. Dr. Celso Maciel da Costa Celso Maciel da Costa

Profa. Dra. Maria da Graça Brasil Rocha M.G.B. Rocha

*Tudo tem a sua ocasião própria, e há tempo para todo propósito debaixo do céu.
Há tempo de nascer, e tempo de morrer; tempo de chorar, e tempo de rir;
tempo de abraçar, e tempo de abster-se de abraçar;
tempo de buscar, e tempo de perder;
tempo de guardar, e tempo de deitar fora;
tempo de estar calado, e tempo de falar;
tempo de amar, e tempo de odiar;
tempo de guerra, e tempo de paz.
(Eclesiastes, 3)*

*Aos meus pais, pela dedicação,
compreensão e apoio incondicional.
Aos meus irmãos, pelo carinho e pela força.*

Agradecimentos

Aos meus pais, meus irmãos, minha madrinha e à *batian*, que me deram todo o suporte emocional e, apesar da distância e da saudade, compreenderam que era o melhor para mim. Obrigada por sempre acreditar!

À Dra Regina Santana, pela orientação nota 10!

À Sarita (Dra. Sarita Bruschi), por tudo: por me explicar como funcionava o SMPL, por explicar sobre o ASiA, pela ajuda inicial para entender o ParSMPL, por tirar minhas dúvidas quanto ao ASDA, por estar sempre disponível quando precisei de qualquer ajuda. Sobretudo, pela amizade e pelos incontáveis momentos de risos e descontração.

Ao Euclides, pelo amor, carinho, apoio e confiança. Obrigada por compreender minhas ausências.

À Simone, pela amizade, dedicação e lealdade inigualáveis. Si, São Carlos ficou muito melhor depois que você passou a fazer parte da minha vida!!!

À Célia, por me “apresentar” São Carlos. Pela companhia nas caminhadas pela manhã e pela ajuda no processo de validação do ParSMPL. Antes de você, fazer compras nunca foi tão prazeroso! ☺

À Kalinka, pelo apoio durante todos esses anos e pela força nas minhas crises existenciais.

Ao Dr Aleardo Manacero Jr., meu orientador na graduação, pelo incentivo inicial.

Ao Douglas, ao Renato (Japa) e à Thais, obrigada por disparar e cuidar dos meus testes nas (incontáveis) noites em que eu estava em Campinas.

À Juliana e à Fabíola, minhas companheiras de república.

Aos amigos do LABES, em especial Erika, Adenilso e André: a alegria de vocês é contagiante!

À CAPES, pelo apoio financeiro.

Ao Douglas, agradecimento especial por me fazer compreender o quanto vale uma noite de sono. Nunca vou esquecer as noites em claro no laboratório, terminando os trabalhos das disciplinas no último segundo!

Ao Dr. Marcos Santana e os aos companheiros do LASDPC, pelas sugestões durante o trabalho.

Sumário

<i>Agradecimentos</i>	v
<i>Sumário</i>	vii
<i>Lista de Figuras</i>	xi
<i>Lista de Tabelas</i>	xiii
<i>Lista de Abreviaturas</i>	xv
<i>Resumo</i>	xvii
<i>Abstract</i>	xix
<i>Capítulo 1 – Introdução</i>	1
<i>Capítulo 2 – Computação Paralela</i>	5
2.1. Considerações Iniciais	5
2.2. Conceitos Básicos	6
2.2.1. Paralelismo e Concorrência	6
2.2.2. Granulosidade	6
2.2.3. Speedup e Eficiência	7
2.2.4. Classificação das Arquiteturas Paralelas	8
2.3. Programação Concorrente	13
2.3.1. Mecanismos para ativação e finalização de tarefas concorrentes	13
2.3.2. Coordenação da Execução Paralela	16
2.4. Suporte à Programação Concorrente	19
2.5. Ambientes de Passagem de Mensagem	20
2.6. Considerações Finais	21
<i>Capítulo 3 – Simulação</i>	23
3.1. Considerações Iniciais	23
3.2. Técnicas para Avaliação de Desempenho	24
3.3. Ciclo de Vida da Simulação	25
3.4. Conceitos Básicos de Simulação Discreta	29
3.5. Técnicas para Decomposição da Simulação	30
3.6. Simulação Distribuída	33
3.7. Protocolos Conservativos	34
3.7.1. Prevenção de Deadlock Utilizando Mensagens Nulas	36
3.7.2. Detecção e Recuperação de Deadlock	37
3.7.3. Janelas de Tempo Conservativas	38
3.7.4. Eventos Condicionais	38
3.8. Protocolos Otimistas	39
3.8.1. Time Warp	39
3.8.2. Variações no Time Warp	40
3.9. Protocolo Assíncrono	41

3.10.	Considerações finais.....	42
<i>Capítulo 4 – Linguagens e Ambientes</i>		45
4.1.	Considerações Iniciais.....	45
4.2.	Linguagens e Bibliotecas para Simulação Distribuída.....	46
4.3.	Ambientes de Simulação Automáticos	49
4.4.	SMPL.....	50
4.4.1.	Principais Rotinas do SMPL.....	51
4.5.	A Extensão ParSMPL.....	52
4.5.1.	Estrutura do ParSMPL.....	53
4.5.2.	Principais rotinas do ParSMPL.....	53
4.6.	ASDA	55
4.6.1	Módulo Avaliador	56
4.6.2.	Módulo Gerador.....	57
4.7.	Considerações Finais.....	58
<i>Capítulo 5 – Adaptação, Aprimoramento e Validação do ParSMPL.....</i>		59
5.1.	Considerações Iniciais.....	59
5.2.	Adaptação do ParSMPL	60
5.3.	Aprimoramento do ParSMPL.....	61
5.4.	Validação	75
5.5.	Considerações Finais.....	84
<i>Capítulo 6 – Análise do Desempenho do Protocolo Conservativo.....</i>		87
6.1.	Considerações Iniciais.....	87
6.2.	Descrição dos Testes Realizados	87
6.2.1.	Ambiente.....	87
6.2.2.	Modelos	88
6.3.	Avaliação do Lookahead Dinâmico.....	97
6.4.	Avaliação da Influência do Aumento da Carga	103
6.5.	Influência da Granulosidade.....	109
6.6.	Avaliação da Influência do Tamanho do Modelo e Balanceamento entre Processos Lógicos.....	115
6.7.	Diretrizes para o Módulo Avaliador	119
6.8.	Considerações Finais.....	121
<i>Capítulo 7 – Inclusão do ParSMPLX ao ASDA.....</i>		123
7.1.	Considerações Iniciais.....	123
7.2.	Estrutura do Módulo Gerador.....	124
7.3.	Gabarito para Geração de Código ParSMPLX.....	125
7.4.	Alterações no Gerador de Aplicação do ASiA para Adaptação ao ASDA	127
7.5.	Considerações Finais.....	137
<i>Capítulo 8 – Conclusões</i>		139
8.1.	Comentários sobre a Revisão Bibliográfica	139
8.2.	Conclusões.....	140
8.3.	Contribuições	143
8.4.	Sugestões de Trabalhos Futuros.....	144
<i>Referências Bibliográficas.....</i>		147
<i>Apêndice A – Guia de utilização do ParSMPLX.....</i>		155
A.1.	Considerações iniciais	155
A.2.	Requisitos do sistema	156
A.3.	API do ParSMPLX.....	156
A.3.1	Funções para geração de variáveis aleatórias	156
A.3.2	Funções da biblioteca ParSMPLX	158
A.3.3	Funções para depuração.....	163
A.3.4	Variáveis do ParSMPLX	165

A.4.	Estrutura de um programa de simulação utilizando o ParSMPLX	166
A.5.	Compilação da biblioteca e dos programas de simulação	175
A.6.	Execução dos programas de simulação	177
	Referências:	181
	<i>Apêndice B – Alterações Gerais no Gerador de Aplicações do ASiA</i>	183
B.1.	Relação de Alterações	183
	Referência:	188

Lista de Figuras

Figura 2.1: Classificação de Duncan	11
Figura 2.2: Exemplo do controle de paralelismo com Fork/Join	14
Figura 2.3: Exemplo do controle de paralelismo com Cobegin/Coend	15
Figura 2.4: Exemplo do controle de paralelismo com Doall	16
Figura 3.1: Ciclo de vida da simulação	27
Figura 3.2: Estratégias de decomposição de domínio	32
Figura 3.3: Ocorrência de <i>deadlock</i> e estouro de memória	36
Figura 4.1: Hierarquia de Software no ParSMPL	52
Figura 4.2: Hierarquia de rotinas no ParSMPL	54
Figura 4.3: Diagrama Modular do ASDA	55
Figura 5. 1: (a) Declaração original da rotina <code>schedule()</code> (b) Declaração atual da mesma função	61
Figura 5. 2: Exemplo de rotina para alocação de memória	62
Figura 5. 3: Código original da rotina <code>parsmpl_pvm_exit()</code>	63
Figura 5. 4: Novo código da rotina <code>parsmpl_pvm_exit()</code>	64
Figura 5. 5: (a) Trecho original da rotina <code>insere_lista_eventos_futuros()</code> (b) Trecho corrigido da rotina	65
Figura 5. 6: Trecho do código original da rotina <code>cause()</code>	67
Figura 5. 7: Trecho do código modificado da rotina <code>cause()</code>	68
Figura 5. 8: Código original da rotina <code>schedule()</code>	70
Figura 5. 9: Novo código da rotina <code>schedule()</code>	70
Figura 5. 10: Código original da rotina <code>erro()</code>	71
Figura 5. 11: Novo código da rotina <code>erro()</code>	71
Figura 5. 12: Processo lógico com apenas um recurso	73
Figura 5. 13: Processo lógico com apenas um recurso	73
Figura 5. 14: Processo lógico com recursos em paralelo	73
Figura 5. 15: Processo lógico com mais de um ponto de alimentação	74
Figura 5. 16: Processo lógico com mais de uma saída	74
Figura 5. 17: Processo lógico combinado	75
Figura 5. 18: Servidor de Arquivos	76
Figura 5. 19: Servidor de arquivos com 4 discos	79
Figura 6. 1: Modelo 1 - Servidor Central	89
Figura 6. 2: Modelo 2 - configuração 1	90
Figura 6. 3: Modelo 2 - configuração 2	90
Figura 6. 4: Modelo 3 - configuração 1	92
Figura 6. 5: Modelo 3 - configuração 2	92
Figura 6. 6: Modelo 3 - configuração 3	93

Figura 6. 7: Modelo 4	94
Figura 6. 8: Modelo 5	96
Figura 6. 9: Modelo 2 – Tempo de Execução	104
Figura 6. 10: Modelo 2 – Variação do speedup com o aumento da carga.....	106
Figura 6. 11: Modelo 4 – Variação do speedup com o aumento da carga.....	108
Figura 6. 12: Modelo 4 - Tempo de Execução.....	108
Figura 6. 13: Modelo 3 - Tempo de execução	110
Figura 6. 14: Modelo 3 – Speedup	111
Figura 6. 15: Modelo 5 - Tempo de execução	113
Figura 6. 16: Modelo 5 - <i>Speedup</i>	113
Figura 7. 1: Estrutura de arquivos no módulo Gerador do ASDA	124
Figura 7. 2: Gabarito para geração de código utilizando ParSMPLX.....	126
Figura A. 1: Descrição de processos no ParSMPL	167
Figura A. 2: Sistema Computacional composto por CPU e Disco.....	167
Figura A. 3: Codificação de PL0 (pl0.c - Parte 1 de 4)	168
Figura A. 4: Codificação de PL0 (pl0.c - Parte 2 de 4)	169
Figura A. 5: Codificação de PL0 (pl0.c - Parte 3 de 4)	170
Figura A. 6: Codificação de PL0 (pl0.c - Parte 4 de 4)	172
Figura A. 7: Codificação de PL1 (pl1.c - Parte 1 de 2)	173
Figura A. 8: Codificação de PL1 (pl1.c - Parte 2 de 2)	174
Figura A. 9: Makefile para ParSMPLX.....	175
Figura A. 10: Makefile.aimk, utilizado para compilar a simulação composta pelos programas pl0.c e pl1.c	176
Figura A. 11: Ativação do console do PVM.....	177
Figura A. 12: Inclusão de máquinas à máquina virtual do PVM.....	178
Figura A. 13: Verificando a configuração da máquina virtual do PVM	179
Figura A. 14: Ambiente preparado para a execução da simulação	180

Lista de Tabelas

Tabela 5. 1: Resultados da simulação com o SMPL	76
Tabela 5. 2: Resultados da simulação com o ParSMPLX, utilizando <i>lookahead</i> estático	77
Tabela 5. 3: Resultados da simulação com o ParSMPLX, utilizando <i>lookahead</i> dinâmico.....	77
Tabela 5. 4: Resultados das simulações do Servidor de Arquivo Simples com o SMPL	77
Tabela 5. 5: Resultados das simulações do Servidor de Arquivos Simples com o ParSMPLX, utilizando <i>lookahead</i> estático	78
Tabela 5. 6: Resultados das simulações do Servidor de Arquivos Simples com o ParSMPLX, utilizando <i>lookahead</i> dinâmico.....	78
Tabela 5. 7: Resultados da simulação com o SMPL	80
Tabela 5. 8: Resultados da simulação com o ParSMPLX, utilizando <i>lookahead</i> estático	80
Tabela 5. 9: Resultados da simulação com o ParSMPLX, utilizando <i>lookahead</i> dinâmico.....	81
Tabela 5. 10: Resultados das simulações com o SMPL: utilização dos recursos	81
Tabela 5. 11: Resultados das simulações com o SMPL: tamanho médio da fila dos recursos	82
Tabela 5. 12: Resultados das simulações com o ParSMPLX utilizando <i>lookahead</i> estático: utilização dos recursos	82
Tabela 5. 13: Resultados das simulações com o ParSMPLX utilizando <i>lookahead</i> estático: tamanho médio da fila dos recursos.....	83
Tabela 5. 14: Resultados das simulações com o ParSMPLX utilizando <i>lookahead</i> dinâmico: utilização dos recursos	83
Tabela 5. 15: Resultados das simulações com o ParSMPLX utilizando <i>lookahead</i> estático: tamanho médio da fila dos recursos.....	84
Tabela 6. 1: Configuração das máquinas	88
Tabela 6. 2: Parametrização do Modelo 1	89
Tabela 6. 3: Parametrização do Modelo 2	91
Tabela 6. 4: Parametrização do Modelo 3	93
Tabela 6. 5: Parametrização do Modelo 4	94
Tabela 6. 6: Parametrização do Modelo 5	96
Tabela 6. 7: Tempo de execução dos modelos, em segundos.....	98
Tabela 6. 8: <i>Speedup</i> em relação à média do tempo sequencial	98
Tabela 6. 9: Diferença percentual do tempo de execução da simulação distribuída com <i>lookahead</i> dinâmico em relação à execução da simulação distribuída com <i>lookahead</i> estático.....	99
Tabela 6. 10: Porcentagem do tempo gasto com processamento, em relação ao tempo total de execução.....	100
Tabela 6. 11: Número de mensagens completas geradas	100
Tabela 6. 12: Porcentagem do tempo gasto com transmissão de mensagens completas, em relação ao tempo total de execução	101

Tabela 6. 13: Número de mensagens nulas geradas	101
Tabela 6. 14: Porcentagem do tempo gasto com transmissão de mensagens nulas, em relação ao tempo total de execução	101
Tabela 6. 15: Modelo 2 – Tempo de execução, em segundos	103
Tabela 6. 16: Modelo 2 – Utilização dos recursos	104
Tabela 6. 17: Modelo 2, configuração 1 – Resultados dos testes.....	105
Tabela 6. 18: Modelo 2, configuração 2 – Resultados dos testes.....	105
Tabela 6. 19: Modelo 2 – Número de mensagens completas e nulas geradas	105
Tabela 6. 20: Modelo 4 – Tempo de execução, em segundos	106
Tabela 6. 21: Modelo 4 – Utilização dos recursos	107
Tabela 6. 22: Modelo 4 – Resultado dos testes com <i>lookahead</i> estático.....	107
Tabela 6. 23: 4 – Resultado dos testes com <i>lookahead</i> dinâmico	107
Tabela 6. 24: Modelo 3 – <i>Speedup</i>	111
Tabela 6. 25: Modelo 3 – Porcentagem do tempo gasto com processamento, em relação ao tempo total de execução	112
Tabela 6. 26: Modelo 3 – Porcentagem do tempo gasto com transmissão de mensagens completas, em relação ao tempo total de execução	112
Tabela 6. 27: Modelo 3 – Porcentagem do tempo gasto com transmissão de mensagens nulas, em relação ao tempo total de execução.....	112
Tabela 6. 28: Modelo 5 – <i>Speedup</i>	114
Tabela 6. 29: Modelo 5 - Porcentagem de tempo gasto com processamento, em relação ao tempo total de execução.....	114
Tabela 6. 30: Modelo 5 - Porcentagem de tempo gasto com transmissão de mensagens completas, em relação ao tempo total de execução	114
Tabela 6. 31: Modelo 5 - Porcentagem de tempo gasto com transmissão de mensagens nulas, em relação ao tempo total de execução.....	115
Tabela 6. 32: Média dos tempos de execução, em segundos.....	116
Tabela 6. 33: Utilização dos recursos	116
Tabela 6. 34: <i>Speedup</i> em relação à média do tempo sequencial	117
Tabela 6. 35: Porcentagem de tempo gasto com processamento em relação ao tempo de execução total	118
Tabela 6. 36: Porcentagem de tempo gasto com transmissão de mensagens completas em relação ao tempo de execução total	118
Tabela 6. 37: Porcentagem de tempo gasto com transmissão de mensagens nulas em relação ao tempo de execução total	119

Lista de Abreviaturas

API	<i>Application Program Interface</i>
ASDA	<i>Ambiente de Simulação Distribuída Automático</i>
ASiA	<i>Ambiente de Simulação Automático</i>
CMB	<i>Chandy-Misra-Bryant</i>
CPU	<i>Central Processing Unit</i>
GVT	<i>Global Virtual Time</i>
LP	<i>Logical Process</i>
LVT	<i>Local Virtual Time</i>
MIMD	<i>Multiple Instruction, Multiple Data</i>
MISD	<i>Multiple Instruction, Single Data</i>
MPI	<i>Message Passing Interface</i>
MPP	<i>Massively Parallel Processor</i>
MRIP	<i>Multiple Replications In Parallel</i>
PL	<i>Processo Lógico</i>
PVM	<i>Parallel Virtual Machine</i>
RPC	<i>Remote Procedure Call</i>
SIMD	<i>Single Instruction, Multiple Data</i>
SISD	<i>Single Instruction, Single Data</i>
SRIP	<i>Single Replications In Parallel</i>

Resumo

Esta dissertação de mestrado apresenta o trabalho desenvolvido para avaliação e aprimoramento da biblioteca para simulação distribuída, ParSMPL, visando sua inclusão em um ambiente de simulação distribuída automático, ASDA. Durante o trabalho de mestrado desenvolvido foi feita uma análise do uso da ferramenta ParSMPL em plataforma Linux, coletando dados relevantes para o complemento de diretrizes apresentadas em trabalhos anteriores. Tais diretrizes visam auxiliar o desenvolvimento de aplicações utilizando simulação distribuída conservativa, e serão incluídas no módulo avaliador do ambiente ASDA. Novas características foram adicionadas ao ParSMPL, buscando melhorar seu desempenho e também facilitar sua depuração. Esse trabalho de adaptação e aprimoramento do ParSMPL encontra-se descrito nesta dissertação. Foi feito também um estudo do trabalho necessário à inclusão da biblioteca ao ASDA, apresentando uma visão geral das alterações necessárias ao gerador de aplicações existente para o ambiente ASiA, voltado à simulação seqüencial, de modo a torná-lo apto à produção de programas de simulação distribuída utilizando protocolo conservativo.

Abstract

This dissertation presents the work developed to evaluate and improve the performance of a distributed simulation library, ParSMPL, aiming its inclusion to the automatic environment for distributed simulation, ASDA. It is presented an analysis about ParSMPL tool utilization in Linux system. This analysis provides relevant data to complement the directrix presented in earlier works. Such directrix aims helping the development of conservative distributed simulation applications, and it will be included to the ASDA evaluator module. New features was included to ParSMPL, aiming to improve its performance and to facilitate its debugging process. The adapting and improvement work made to ParSMPL is related in this dissertation. Besides, it was made a study about the necessary effort to include ParSMPL library to ASDA environment. It results in an overview of necessary changes to the existent application generator that belongs to ASiA environment, in order to make it able to produce distributed simulation programs in conservative protocol.

Capítulo 1 – Introdução

Nas últimas décadas tem-se presenciado uma grande evolução na arquitetura de computadores. Isto pode ser constatado com o surgimento de máquinas cada vez mais rápidas e mais potentes, redes de comunicação mais rápidas e confiáveis, acompanhado de uma sensível redução no custo de máquinas paralelas (apesar deste custo ainda ser relativamente alto).

Este avanço tecnológico tem impulsionado o estudo de simulação distribuída, ou simulação paralela: utilizar máquinas paralelas ou computadores em rede para executar simulações tem se mostrado uma solução atrativa para reduzir o tempo necessário para a obtenção de resultados. Esta solução é particularmente interessante em simulações muito complexas, ou com grande nível de detalhes. Por exemplo, ao simular novas arquiteturas de computadores, o nível de detalhamento exigido é tão grande que seria inviável utilizar simulação seqüencial.

Contudo, apesar da grande vantagem oferecida pela computação paralela, a migração para o paradigma de computação paralela não é simples. Exigem-se conceitos de programação paralela aliados ao de simulação. Além disso, um problema não encontrado em simulação seqüencial surge na simulação distribuída: a necessidade de sincronização. Isto é necessário para garantir que os eventos sejam simulados obedecendo as relações de precedência encontradas no sistema real. Se estas relações não forem obedecidas, ocorre um *erro de causa e efeito*.

Existem três abordagens para se lidar com erros de causa e efeito: impedir que eles aconteçam; esperar que eles não aconteçam e, caso aconteçam, tratar eventuais ocorrências; ou então não se importar com tais ocorrências. Estas são as características dos três protocolos de sincronização existentes. O protocolo *conservativo* impede a ocorrência de erros de causa e efeito, processando apenas eventos seguros. O protocolo *otimista* assume

que a simulação seguirá corretamente e que tais erros não acontecerão. Caso seja detectado um erro de causa e efeito, um mecanismo para retornar a simulação a um estado seguro é disponibilizado. O protocolo *assíncrono* permite a ocorrência de erros e prossegue a simulação sem tratá-los.

Com os protocolos de sincronização inclui-se então mais um obstáculo na simulação distribuída: para se escrever programas de simulação distribuída, além de conhecimentos sobre computação paralela e simulação, é necessário também ter conhecimento sobre o funcionamento do protocolo escolhido.

Visando facilitar o trabalho do usuário e assim ampliar o uso da simulação distribuída, um número de ferramentas para simulação distribuída tem sido alvo de pesquisa e desenvolvimento. Estas ferramentas consistem principalmente de linguagens e extensões para simulação distribuída, que escondem detalhes do protocolo de sincronização e de comunicação envolvidos. Os ambientes de simulação automáticos são outro tipo de ferramenta para auxiliar nesta tarefa. Eles efetuam automaticamente a transcrição do modelo de simulação elaborado pelo usuário em um programa para simulação e o executam. Isto facilita o trabalho de usuários inexperientes e o torna mais rápido aos mais experientes, sem limitar a flexibilidade do usuário em incluir modificações próprias.

O objetivo deste trabalho de mestrado desenvolvido é a adaptação e avaliação do uso da ferramenta ParSMPL (Ulson, 1999) na plataforma Linux, e sua inclusão no contexto do ASDA (Bruschi et al, 2000; Bruschi, 2002). O ParSMPL é uma extensão da linguagem C para simulação distribuída, baseada em uma extensão da linguagem C para simulação sequencial, o SMPL (MacDougall, 1987). Implementa o protocolo conservativo CMB (Chandy & Misra, 1979; Bryant, 1977) e utiliza para comunicação o ambiente de passagem de mensagens PVM. O ASDA é um ambiente de simulação distribuída automático, capaz de gerar e executar automaticamente simulações distribuídas a partir de uma especificação gráfica do modelo desejado.

O objetivo do trabalho realizado inclui a avaliação do comportamento do ParSMPL em plataforma Linux, coletando dados relevantes para o complemento das diretrizes apresentadas em (Ulson, 1999). Essas diretrizes visam auxiliar o desenvolvimento de aplicações utilizando simulação distribuída conservativa, e serão incluídas no módulo avaliador do ambiente ASDA.

Para avaliar o comportamento do ParSMPL em plataformas Linux, foi feito um trabalho no sentido de adaptá-lo ao Linux, eliminando problemas de compilação e execução pré-existentes. Também foram aplicados aprimoramentos ao ParSMPL, buscando com isso

melhorar seu desempenho e também facilitar sua depuração. Devido às modificações inseridas ao ParSMPL original, sugere-se a denominação de ParSMPLX à esta nova versão.

Durante o programa de mestrado foi realizado também um estudo do trabalho que deve ser feito para inclusão do ParSMPLX ao ASDA. O resultado disso é uma visão geral das alterações necessárias ao gerador de aplicações existente para o ambiente ASiA (Spolon, 1994) para torná-lo apto à produção de programas de simulação utilizando o ParSMPLX.

Esta dissertação apresenta a revisão bibliográfica utilizada e descreve o trabalho desenvolvido durante o programa de mestrado, estando organizada da seguinte forma: o Capítulo 2 apresenta os conceitos básicos de Computação Paralela, como definições de *speedup* e eficiência, granulosidade, classificação de arquiteturas paralelas, além de descrever os principais mecanismos de programação concorrente. O Capítulo 3 traz as principais técnicas de avaliação de desempenho e descreve os principais aspectos da simulação discreta, com ênfase na simulação distribuída. As principais características dos protocolos conservativo e otimista também estão descritos no capítulo 3.

O Capítulo 4 apresenta as principais linguagens e extensões funcionais utilizadas para simulação distribuída. Descreve com maior detalhe a extensão SMPL, a extensão ParSMPL e descreve o ambiente ASDA, focos do trabalho. Também apresenta uma descrição de ambientes de simulação automáticos e uma visão geral de alguns dos ambientes existentes mais conhecidos.

No Capítulo 5 encontra-se uma descrição do trabalho feito para adaptação do ParSMPL ao Linux e os aprimoramentos aplicados, resultando na versão ParSMPLX. Neste capítulo também é apresentada a validação dos resultados apresentados pelo ParSMPLX.

Os testes para avaliação do desempenho do ParSMPLX em plataformas Linux estão descritos no Capítulo 6, assim como os modelos, suas parametrizações e os recursos de *hardware* utilizados. O Capítulo 7 descreve o trabalho necessário para a inclusão do ParSMPLX ao ASDA. O Capítulo 8 traz as conclusões e contribuições deste trabalho de mestrado e sugestões para trabalhos futuros.

Capítulo 2 – Computação Paralela

2.1. Considerações Iniciais

A computação paralela pode ser definida como uma coleção de elementos de processamento que, comunicando e cooperando entre si, possibilita a resolução de grandes problemas, de uma maneira mais rápida do que através da execução em um único elemento de processamento (Almasi & Gottlieb, 1994).

O uso da computação paralela tem permitido que problemas complexos sejam resolvidos e que aplicações de alto desempenho possam ser implementadas tanto em áreas tradicionais, como ciência e engenharia, quanto em novas áreas de aplicações, como em inteligência artificial e finanças. Entre as principais razões que justificam o uso do paralelismo, pode-se citar (Skillicorn & Talia, 1998):

- A natureza do mundo real é inerentemente paralela e geralmente sugere o nível certo de abstração que deve ser utilizada no projeto.
- O paralelismo disponibiliza maior desempenho computacional do que qualquer processador isolado, embora conseguir este maior desempenho em computadores paralelos não ocorra de forma direta.

Apesar das vantagens citadas, existem problemas que dificultam uma maior utilização do paralelismo. Um problema é a falta de ferramentas adequadas para computação paralela, que ainda é imatura e foi desenvolvida depois da tecnologia. Além disso, a portabilidade é um problema muito mais sério em programação paralela do que em programação sequencial. A transferência de um sistema de software de uma arquitetura paralela para outra pode exigir grande trabalho, incluindo a necessidade de ter que se reconstruir completamente o software (Skillicorn & Talia, 1998).

2.2. Conceitos Básicos

Serão apresentados nas próximas subseções conceitos básicos de paralelismo e concorrência, granulosidade, speedup e eficiência, e classificação de arquiteturas paralelas.

2.2.1. Paralelismo e Concorrência

Entende-se por concorrência quando dois ou mais processos já iniciaram sua execução, mas ainda não terminaram. Assim sendo, a concorrência pode ocorrer tanto em sistemas de um único processador quanto em sistemas com vários processadores.

A concorrência em sistemas com apenas um processador recebe o nome de *pseudo-paralelismo* ou *paralelismo lógico*. Neste caso o usuário tem a falsa impressão de que todos os processos estão sendo executados ao mesmo tempo, quando na verdade os processos compartilham o mesmo processador e existe apenas um processo sendo executado a cada instante.

Em sistemas com vários processadores pode-se explorar o *paralelismo real*, também chamado de *paralelismo físico*, que ocorre quando há mais de um processo sendo executado no mesmo instante. O paralelismo real pode ser classificado em espacial, temporal (*pipeline*) ou combinado (Ulson, 1999). O *paralelismo espacial* está ligado à sobreposição de execução de tarefas, podendo ser síncrono ou assíncrono. No paralelismo síncrono vários elementos de processamento executam funções idênticas e são sincronizados por uma única unidade de controle. No paralelismo assíncrono, os elementos de processamento executam funções diferentes e interagem entre si. O *paralelismo temporal* está relacionado com a execução de tarefas como uma cascata ou estágios de subtarefas, de forma que execuções diferentes ocorram com sobreposição e defasagem no tempo. O *paralelismo combinado* é obtido combinando-se o paralelismo espacial com o temporal e permite a execução sobreposta de vários conjuntos de tarefas, que por sua vez são executadas com defasagem no tempo.

Pode-se optar ainda por uma mistura de pseudo-paralelismo e paralelismo real, quando existem mais eventos em execução do que o número de processadores (Bruschi, 1997).

2.2.2. Granulosidade

A granulosidade, ou nível de paralelismo, é o termo utilizado para especificar o tamanho das tarefas que compõem um programa paralelo. A granulosidade dos processos é uma característica que está intimamente relacionada com a arquitetura utilizada. Existem

diversas classificações na literatura, com um maior ou menor número de níveis, dependendo do autor (Navaux, 1989; Almasi & Gottlieb, 1994; Hwang, 1993). Em geral, a granulosidade pode ser classificada como *fin*a, *méd*ia ou *gross*a.

O paralelismo de grão fino geralmente é explorado em nível de instruções ou laços, preferivelmente assistida por um compilador vetorial. O paralelismo de grão médio, em nível de tarefas ou procedimentos; já o paralelismo de grão grosso aplica-se em nível de programas e baseia-se fortemente em um sistema operacional eficaz e também na eficiência do algoritmo utilizado.

A comunicação através de variáveis compartilhadas geralmente é usada para dar suporte à computações de grão fino a grão médio. A troca de mensagens tem sido utilizada em computações de grão médio a grão grosso. Em geral, quanto mais fino o tamanho do grão, maior o potencial para paralelismo e maior também a sobrecarga de comunicação e escalonamento. O paralelismo em massa geralmente é explorado em nível de grão fino, como o paralelismo de dados em computadores SIMD ou MIMD (Hwang, 1993). A granulosidade fina pode limitar o *speedup* devido à grande quantidade de sincronismo necessária, pois é muito difícil obter sincronismo neste caso sem gerar uma sobrecarga excessiva (Bruschi, 1997).

2.2.3. *Speedup e Eficiência*

Os dois critérios mais importantes para a avaliação de algoritmos paralelos são o *speedup* e a *eficiência*.

Originalmente, *speedup* foi definido como a razão entre o melhor tempo de execução conhecido do algoritmo serial pelo tempo de execução da implementação correspondente em um computador paralelo de n processadores (Lewis, 1993; Zomaya, 1996). Esta definição é muito difícil de ser utilizada porque o melhor tempo de execução serial do programa não é conhecido, ou então o algoritmo paralelo difere tanto do algoritmo serial que uma comparação não teria muito sentido.

Como uma consequência, a definição de *speedup* evoluiu e pode ser medida executando-se o programa paralelo utilizando de um a vários processadores. O *speedup* então, é definido como a razão do tempo de execução do programa paralelo executando em um processador pelo tempo de execução do mesmo programa executando em n processadores (Lewis, 1993; Almasi & Gottlieb, 1994):

$$Speedup = \frac{T_1}{T_n}$$

onde T_1 é o tempo de execução do programa executando em 1 processador, e T_n é o tempo de execução do mesmo programa executando em n processadores. Os algoritmos que conseguem *speedup* linear à medida que são adicionados novos processadores são chamados de *completamente paralelizáveis*, o que seria o caso ideal (Almasi & Gottlieb, 1994).

A *eficiência* mede qual a redução de trabalho conseguida utilizando-se n processadores ao invés da execução serial em um único processador (Kronsjö, 1996). Em outras palavras, a *eficiência* é a medida de quanto se utiliza os processadores paralelos:

$$Eficiência = \frac{Speedup}{n}$$

A eficiência assume valores entre 0 (0%) e 1 (100%), este último quando se consegue *speedup* ideal. Dificilmente se obtém eficiência igual a 1, devido a três limitantes do *speedup*: sobrecarga da comunicação entre os processadores, porções do código estritamente sequenciais e a granulosidade escolhida (Almasi & Gottlieb, 1994).

2.2.4. Classificação das Arquiteturas Paralelas

Existem vários tipos de arquiteturas paralelas. Utilizando-se uma classificação para as arquiteturas, torna-se mais fácil discutir os diferentes tipos de computadores paralelos. Para atender esses requisitos, uma classificação ideal deve ser:

- **Hierárquica:** iniciando em um nível mais abstrato, a classificação deve ser refinada em subníveis à medida que se diferencie de maneira mais detalhada cada arquitetura;
- **Universal:** um computador único deve ter uma classificação única;
- **Extensível:** futuras máquinas que surjam devem ser incluídas sem que sejam necessárias modificações na classificação;
- **Concisa:** os nomes que representam cada uma das classes devem ser pequenos para que a classificação seja de uso prático;
- **Abrangente:** a classificação deve incluir todos os tipos de arquiteturas existentes.

Diversas classificações podem ser encontradas na literatura, porém todas apresentam falhas. Apesar de antiga e não abranger todos os tipos de arquiteturas atualmente existentes, a classificação mais amplamente utilizada é a de Flynn (Flynn, 1972; Flynn, 1996), baseada na noção de fluxos de instruções e dados. Uma classificação mais recente e mais abrangente é a classificação de Duncan (Duncan, 1990), embora menos conhecida.

Classificação de Flynn

A classificação proposta por Flynn é composta de quatro categorias de máquinas, conforme a multiplicidade do fluxo de dados e de instruções. Como é bastante simples, esta classificação é a melhor aceita pela comunidade científica. Neste contexto, “fluxo” descreve uma sequência de instruções ou de dados, executados em um processador (Flynn, 1972; Flynn, 1996; Navaux, 1989; Duncan, 1990; Hwang, 1993; Almasi & Gottlieb, 1994).

As quatro categorias são:

1. **SISD** (*Single Instruction Stream, Single Data Stream* – único fluxo de instruções, único fluxo de dados). Representa a maioria dos computadores convencionais disponíveis atualmente: os computadores seriais baseados no princípio de Von Neumann. As instruções são executadas sequencialmente, embora possam ser sobrepostas em seus estágios de execução (*pipeline*). Computadores SISD podem ter mais de uma unidade funcional, porém todos sob a supervisão de apenas uma unidade de controle.
2. **SIMD** (*Single Instruction Stream, Multiple Data Stream* – único fluxo de instrução, múltiplo fluxo de dados). Corresponde aos processadores matriciais paralelos e associativos. São vários elementos de processamento supervisionados por uma mesma unidade de controle, isto é, todos os elementos de processamento recebem a mesma instrução para operarem sobre diferentes dados. A memória do sistema pode estar dividida em módulos, vinculados a cada elemento de processamento, ou ser compartilhada.
3. **MISD** (*Multiple Instruction Stream, Single Data Stream* – múltiplo fluxo de instruções, único fluxo de dados). Corresponderia a uma máquina que tivesse vários elementos de processamento recebendo instruções distintas das unidades de controle, porém processando o mesmo fluxo de dados. Esta estrutura não existe na prática, sendo considerada sem validade por alguns autores (Navaux, 1989). Alguns autores consideram os arranjos sistólicos para execução em *pipeline* de

algoritmos específicos como representantes deste tipo de arquitetura (Hwang, 1993).

4. **MIMD** (*Multiple Instruction Stream, Multiple Data Stream* – múltiplo fluxo de instruções, múltiplo fluxo de dados). Inclui-se nesta categoria os multiprocessadores. Cada unidade de processamento possui sua unidade de controle, executando de maneira autônoma diversas instruções sobre um conjunto de dados diversos, com a memória mantendo a interação entre os processadores. Caso isto não ocorresse, o sistema seria de *múltiplos SISD (MSISD)*. Uma máquina MIMD pode ainda ser classificada como fracamente acoplada ou fortemente acoplada, dependendo do menor ou maior grau de interação existente entre os vários processadores através da memória (Navaux, 1989).

Classificação de Duncan

A classificação de Duncan (Duncan, 1990) procura especificar uma definição e conseqüente taxonomia para arquiteturas modernas paralelas satisfazendo as seguintes propostas:

- Excluir arquiteturas que incorporam apenas mecanismos de paralelismo de baixo nível, que já se tornaram lugar comum em computadores modernos. Exemplos desses mecanismos são o uso de *pipeline* e de unidades funcionais múltiplas dentro de uma única CPU.
- Manter elementos da taxonomia de Flynn, baseados em fluxos de dados e de instruções, devido à sua ampla utilização.
- Incluir processadores vetoriais (*pipeline*) e outras arquiteturas que intuitivamente deveriam receber o mérito de pertencer às classes de computadores paralelos, mas não se encaixam na classificação de Flynn.

A idéia básica nesta classificação é dividir as arquiteturas em duas frentes, as arquiteturas *síncronas* e as arquiteturas *assíncronas*, como pode ser visto na figura 2.1.

Figura 2.1: Classificação de Duncan

As **arquiteturas síncronas** são as arquiteturas que coordenam as operações através de relógios globais, unidades de controle centrais ou controladores de unidades vetoriais. Divide-se em três classes:

- **Processadores Vetoriais:** possuem *hardware* específico para realizar operações sobre vetores.
- **Arquiteturas SIMD:** utilizam apenas uma unidade de controle e vários processadores; podem ser organizadas em:
 - *Processadores matriciais:* projetados para cálculos científicos em larga escala sobre matrizes de dados. O acesso à memória é via endereço.
 - *Memória associativa:* utiliza comparações lógicas para recuperar, em paralelo, dados armazenados na memória, de acordo com seu conteúdo.
- **Arquiteturas Sistólicas:** são projetadas para solucionar problemas de sistemas de propósito específico que exigem computação intensa, com grande quantidade de operações de entrada e saída. Os processadores são organizados em *pipeline* e somente os processadores das extremidades possuem comunicação com a memória.

As **arquiteturas assíncronas** são caracterizadas pelo controle descentralizado de *hardware*, são compostas por vários processadores independentes executando instruções sobre diferentes conjuntos de dados. Estão divididas em:

- **Arquiteturas MIMD:** correspondem às arquiteturas MIMD da classificação de Flynn. Estas arquiteturas podem ser divididas em duas categorias, dependendo de como a memória está organizada:
 - *Arquiteturas de memória distribuída:* neste tipo de arquitetura cada processador possui sua memória local e está ligado a outros processadores por meio de uma rede. O compartilhamento de dados é feito através de troca de mensagens pela rede.
 - *Arquiteturas de memória compartilhada:* são aquelas nas quais a coordenação entre os processadores é feita através de uma memória compartilhada global a qual todos têm acesso. Computadores de memória compartilhada não apresentam alguns dos problemas encontrados pelas arquiteturas que utilizam troca de mensagens, como a latência ao enviar a mensagem. No entanto, outros problemas, como a sincronização do acesso ao dado e coerência de *cache*, devem ser resolvidos (Duncan, 1990).
- **Paradigma MIMD:** pertencem a esta classe as arquiteturas assíncronas que, além de possuírem características de arquiteturas MIMD, possuem características próprias que dificultam sua classificação como puramente MIMD. São elas:
 - *MIMD/SIMD (híbridas):* são arquiteturas MIMD que possuem partes da arquitetura controlada de modo SIMD.
 - *Fluxo de dados:* a característica fundamental desta arquitetura é que as instruções são habilitadas para a execução assim que seus operandos estiverem disponíveis. A execução das instruções é baseada na dependência de dados, permitindo explorar a concorrência em níveis de tarefas, rotinas e instruções.
 - *Redução:* são arquiteturas dirigidas à demanda – uma instrução é habilitada para execução quando seus resultados são exigidos como operando para outra instrução já habilitada para execução.
 - *Frente de onda:* combina estruturas *pipeline* sistólicas com o paradigma de execução assíncrona de fluxo de dados. Visa conservar as vantagens das arquiteturas sistólicas (balancear computação intensiva com altas taxas de entrada/saída) enquanto elimina algumas das suas desvantagens.

2.3. Programação Concorrente

A idéia básica por trás do processamento paralelo é dividir algum tipo de aplicação, formando um esforço cooperativo. De um modo mais formal, para executar um programa paralelo deve haver uma maneira para definir um conjunto de subtarefas a ser executadas em paralelo, ativar e finalizar sua execução, e também coordenar e especificar o modo como irão interagir durante a execução (Almasi & Gottlieb, 1994).

Estas ações requerem alguns novos construtores, que devem ou aparecer na linguagem de programação ou serem inseridos no programa automaticamente por um compilador ou outro processador de linguagem. Esta forma como os construtores devem aparecer irá depender da natureza das subtarefas ou do modelo computacional sendo utilizado.

2.3.1. Mecanismos para ativação e finalização de tarefas concorrentes

Dentre os mecanismos para ativação e finalização de tarefas concorrentes destacam-se o uso dos pares **Fork/Join** e **Cobegin/Coend**, que iniciam e encerram a execução de processos em paralelo, e do comando **Doall**, que permite que várias iterações de um laço sejam executadas em paralelo.

Fork/Join

O **Fork** é o mecanismo mais primitivo para ativação de processos concorrentes. Conceitualmente, é muito simples e permite que uma sequência de execução adicional inicie em qualquer lugar no programa. O **Join**, que recombina dois processos em um, também é conceitualmente simples. Os comandos apresentam as seguintes sintaxes:

FORK e_i – ativa um processo no endereço e_i , concorrentemente com os demais processos em execução.

JOIN $n, e_1; e_2$ – em uma operação indivisível, o Join subtrai 1 de n , que normalmente é uma variável compartilhada, e vai para o endereço e_1 se n for igual a zero, ou para o endereço indicado por e_2 caso contrário.

A execução de um processo iniciado pelo comando Fork é feita concorrentemente à execução de seu pai, até que o processo pai execute o comando Join ou até que o processo filho termine. O que terminar primeiro deve esperar o outro terminar a execução antes de continuar a execução dos processos (Hwang, 1993).

Figura 2.2: Exemplo do controle de paralelismo com Fork/Join

Como o Fork pode aparecer dentro de laços e desvios condicionais, pode-se criar um número arbitrário de atividades paralelas, e assim o Join pode unir quaisquer desses processos um ao outro, tendo-se então um modo muito flexível e poderoso de iniciar e finalizar atividades paralelas. Contudo, ao mesmo tempo é um procedimento muito perigoso, com grande potencial de escrever programas desestruturados e difíceis de se entender. Além disso, fazer com que o Join funcione apropriadamente em conjunto com o Fork é uma tarefa inteiramente da responsabilidade do programador (Almasi & Gottlieb, 1994). A figura 2.2 mostra um exemplo de trecho de código utilizando a declaração Fork/Join.

Em um ambiente UNIX, os comandos Fork/Join provêem um mecanismo direto para a criação dinâmica de processos, incluindo ativações múltiplas de um mesmo processo (Hwang, 1993).

Cobegin/Coend

Os comandos **Cobegin/Coend**, também conhecidas por **Parbegin/Parend**, provêem um comando de controle estruturado de única-entrada, única-saída, não sendo tão dinâmicos quanto o Fork/Join. Os programas escritos com Cobegin/Coend apresentam-se mais estruturados e com maior clareza do que aqueles escritos com Fork/Join, como pode ser visto na figura 2.3.

Figura 2.3: Exemplo do controle de paralelismo com Cobegin/Coend

Pelo fato de não conseguir representar adequadamente todas as aplicações concorrentes, apenas gráficos com fluxo aninhados, as instruções Cobegin/Coend são apresentadas como versões menos poderosas e menos versáteis do Fork/Join (Hwang, 1993; Almasi & Gottlieb, 1994).

Ao contrário do Fork/Join, o Cobegin/Coend permite que a identificação e execução do paralelismo sejam separadas. O significado das declarações dentro do bloco Cobegin/Coend pode ser interpretado e implementado de várias maneiras. A interpretação mais simples é que a execução de toda declaração entre o Cobegin e o Coend deve começar ao mesmo tempo, mas também são possíveis implementações que permitam que o compilador escolha quantas das declarações elegíveis devem ser executadas concorrentemente, baseado na configuração e situação do hardware no momento da execução (Almasi & Gottlieb, 1994).

Doall

O comando **Doall** pode ser pensado como um Cobegin no qual as instruções elegíveis para execução paralela aparecem dentro de um laço, como pode ser observada na figura 2.4. Todas as iterações no laço Doall são totalmente independentes umas das outras e elas podem ser executadas em paralelo, desde que haja um número suficiente de processadores para lidar com as diferentes iterações. As instruções *dentro* de cada iteração, porém, são executadas serialmente. É uma instrução muito útil para operações em vetores e matrizes e aparecem muito freqüentemente em FORTRAN e outras linguagens de programação para computação científica (Almasi & Gottlieb, 1994).

Figura 2.4: Exemplo do controle de paralelismo com Doall

A restrição ao uso do Doall se dá quando uma iteração pode precisar de algum resultado produzido por iterações prévias. Neste caso, a execução sequencial se torna essencial.

2.3.2. Coordenação da Execução Paralela

Para que a execução possa progredir, os processos paralelos que apresentam alguma cooperação entre si devem, em geral, ser capazes de trocar resultados. Para que esta comunicação seja eficiente, entretanto, é preciso assegurar uma correta sincronização, já que as atividades de cooperação também podem competir por algum recurso compartilhado.

Os métodos de comunicação entre os processos podem ser divididos em duas frentes: pelo uso de variáveis compartilhadas, que se dá em arquiteturas de memória compartilhada, ou através de troca de mensagens, que ocorre em arquiteturas de memória distribuída (Hwang & Briggs, 1984; Hwang, 1993; Almasi & Gottlieb, 1994).

Comunicação e sincronismo em arquiteturas de memória compartilhada

A comunicação em arquiteturas de memória compartilhada é feita usando-se variáveis compartilhadas. O uso de variáveis compartilhadas gera problemas de sincronismo, que devem ser resolvidos por mecanismos de controle de acesso, garantindo exclusão mútua, evitando que um processo escreva numa área de memória enquanto algum outro estiver lendo.

ou vice-versa. Alguns dos métodos mais conhecidos para se assegurar a exclusão mútua são: *test and set*, *sleep/wakeup*, *semáforos* e *monitores* (Almasi & Gottlieb, 1994; Tanenbaum, 2001).

O *test and set* é uma primitiva de sincronização de baixo nível que tem como característica chave o fato de ser uma instrução indivisível. Com ele, assegura-se a entrada na região crítica a apenas um processo, enquanto os demais ficam bloqueados em espera ocupada. Isto é uma das principais desvantagens, pois a espera ocupada acarreta em um gasto de tempo desnecessário, quando ao invés disso o processador poderia estar sendo utilizado em algum trabalho útil. Outro problema ao utilizar apenas primitivas de baixo nível é que seu uso acarreta em sobrecarga no programador, pois é ele quem tem que criar os protocolos para se entrar e sair da região crítica. Tais protocolos podem ser difíceis de se projetar, entender e provar sua exatidão (Almasi & Gottlieb, 1994). A vantagem de se utilizar espera ocupada é que ela oferece uma resposta mais rápida quando o objeto compartilhado se torna disponível (Hwang, 1993). Uma alternativa para contornar o problema da espera ocupada é usar as primitivas **sleep e wakeup**. Neste caso, se o processo for bloqueado, ele é colocado na fila de espera e sua execução fica suspensa (primitiva *sleep*) até que seja acordado pelo processo que está deixando a região crítica (primitiva *wakeup*).

O conceito de **semáforos** foi desenvolvido por Dijkstra e consiste de um par de primitivas de alto nível que garante exclusão mútua livre de *deadlock*. Essas primitivas, chamadas de **P** (ou DOWN) e **V** (ou UP), são indivisíveis, podendo ser utilizadas por vários processos e operam sobre variáveis compartilhadas inteiras não-negativas, os semáforos. Assim, para um semáforo *S*, a operação *P(S)* marca a entrada na região crítica sinalizada por *S*. Se algum processo executar a primitiva *P(S)* e descobrir que o valor de *S* é zero, este processo fica bloqueado até que o valor de *S* se torne positivo. Se o valor de *S* já for positivo ao executar *P(S)*, *S* é decrementado e este processo pode continuar sua execução. Ao deixar a região crítica, o processo deve executar a primitiva *V(S)*, que incrementa o valor do semáforo *S*. Quando a variável de semáforo *S* recebe valores de 0 a 1, é chamada de *semáforo binário* e permite apenas um acesso à região crítica por vez. Se a variável de semáforo puder receber qualquer outro valor inteiro trata-se de um *semáforo contador*: o valor máximo *n* atua como uma coleção de *n* permissões, correspondendo ao uso de *n* cópias de um recurso compartilhado. Pode-se utilizar níveis aninhados de pares (*P*, *V*), quando múltiplos tipos de recursos são utilizados (Hwang, 1993). Como desvantagens, os semáforos apresentam-se particularmente desestruturados, sendo responsabilidade do programador iniciar o acesso a cada região crítica com uma primitiva *P* e terminar esse acesso com uma *V*, além de

assegurar-se de que o semáforo correto está sendo lido em cada caso. É fácil cometer erros, especialmente quando são necessários semáforos múltiplos (Almasi & Gottlieb, 1994).

O **monitor** é uma construção de linguagem de alto nível para construir programas que enfatizam modularidade e encapsulamento. É um conjunto de procedimentos, variáveis e estruturas de dados agrupados em um módulo especial que permite o acesso de apenas um processo por vez. Em um monitor, as regiões críticas são removidas dos corpos dos processos e se tornam procedimentos ou funções definidas sobre variáveis confinadas dentro dos limites do monitor. Os processos podem chamar os procedimentos do monitor sempre que desejarem entrar na região crítica, mas nunca poderão acessar diretamente as estruturas de dados e variáveis internas do monitor a partir de procedimentos declarados fora do monitor.

Comunicação e sincronismo em arquiteturas de memória distribuída

A comunicação e o sincronismo em arquiteturas de memória distribuída são feitos utilizando-se troca de mensagens. Este mecanismo é mais auto-sincronizante do que a comunicação por variáveis compartilhadas, já que uma mensagem não pode ser recebida antes de ser enviada. No entanto, as mensagens podem ser perdidas e é preciso que protocolos de troca de mensagens tratem desse problema (Almasi & Gottlieb, 1994).

A comunicação pode ser classificada como síncrona ou assíncrona. Em comunicação **síncrona**, cada canal de comunicação provê uma conexão direta entre dois processos, e o transmissor fica bloqueado até que o outro processo tenha recebido a mensagem. Assim, cada troca de mensagem representa um ponto de sincronização entre os dois processos. Exemplos são: CSP, Occam, rendezvous e RPC. Se o canal de comunicação tiver capacidade de armazenamento, ocorre a comunicação **assíncrona**, e o processo transmissor não precisa parar ou ficar bloqueado até que a mensagem seja recebida pelo outro processo. Exemplos: Express, PVM e Linda (Almasi & Gottlieb, 1994).

Na maioria das vezes, os serviços de comunicação providos podem ser reduzidos a duas chamadas a biblioteca: **send** e **receive**, que são suportadas por muitas linguagens distribuídas e sistemas operacionais. O envio de uma mensagem é feito através da primitiva *send* e o recebimento pelo seu par *receive*. A sintaxe dessas operações, em geral, é descrita como (Tanenb - n, 2001):

send (destino, &mensagem)
receive (origem, &mensagem)

Uma outra notação mais completa para as primitivas *send/receive* seria (Hwang, 1993):

send (tipo, buffer, tamanho, nó, processo)
receive (tipo, buffer, tamanho)

na qual *tipo* identifica o tipo da mensagem, *buffer* indica a localização da mensagem, *tamanho* especifica o número de bytes da mensagem, *nó* designa o nó de processamento de destino e *processo* é o identificador de processo no nó de destino.

As primitivas *send* e *receive* podem ser tanto bloqueantes como não bloqueantes. Em alguns sistemas ambas estão disponíveis e os usuários podem escolher a mais adequada à aplicação que estão desenvolvendo (Tanenbaum, 2001).

O conceito de **RPC** (*Remote Procedure Call* – chamada a procedimento remoto) é uma combinação estruturada destas primitivas. O RPC provê um modo conveniente e claro para se programar interações cliente/servidor. A idéia do RPC é fazer com que uma chamada a procedimento remoto seja equivalente a uma chamada local. O processo que fez a chamada fica bloqueado até que o procedimento remoto seja executado, estabelecendo uma comunicação síncrona (Almasi & Gottlieb, 1994; Tanenbaum, 2001).

O *rendezvous* da linguagem Ada faz a mesma coisa, mas de um modo um pouco diferente: ambas as tarefas envolvidas em um *rendezvous* assemelham-se em sua capacidade de chamar a outra e de prosseguir sua execução, uma vez que o *rendezvous* tenha terminado. Já em RPC, o programa que fez a chamada e o procedimento remoto apresentam um relacionamento unilateral: o programa pode chamar o procedimento e prosseguir quando o procedimento retorna, mas não o contrário. Conceitualmente, um RPC pode ser visto como um subconjunto ou caso especial do *rendezvous* (Almasi & Gottlieb, 1994).

2.4. Suporte à Programação Concorrente

Três são os tipos de ferramentas mais usuais para o suporte ao desenvolvimento de aplicações paralelas (Almasi & Gottlieb, 1994):

- *Paralelização automática*: é feita com o uso de compiladores paralelizadores, que aceitam o programa sequencial original e produzem um programa paralelo, sem que seja necessário modificar o programa nem aprender uma nova linguagem. Apesar de apresentar como vantagens a rapidez e o baixo custo com que os programas paralelos são produzidos (Perrot, 1996), o *speedup* alcançado com este

método é modesto, e nem sempre tais compiladores estão disponíveis, especialmente para sistemas distribuídos. Exemplo: Parafrase (Almasi & Gottlieb, 1994).

- *Extensões paralelas para linguagens existentes:* requer mais trabalho do usuário do que o método anterior, mas ainda evita a necessidade de se aprender uma linguagem totalmente nova, já que se trata de uma extensão para uma linguagem já conhecida. Este método permite que programas seqüenciais existentes sejam adaptados e transferidos para máquinas paralelas, sem que o programa seqüencial seja completamente reescrito. Ambientes de passagem de mensagem como PVM e Linda são exemplos de extensões. Também existem variantes explicitamente paralelas de linguagens seqüenciais, como o Fortran 90. Além da disponibilidade destas ferramentas ser muito maior do que de compiladores paralelizadores, alcança-se maior grau de paralelismo.
- *Utilização de uma linguagem completamente nova:* todas as linguagens e aplicações já existentes são ignoradas na formulação da nova linguagem. Argumenta-se que isto facilitaria o tratamento coerente de todos os aspectos da programação paralela, permitindo que o usuário expresse e desenvolva uma solução explicitamente paralela. As desvantagens são a necessidade de se aprender a nova linguagem e principalmente o fato de que aplicações existentes devem ser completamente reescritas na nova linguagem (Perrot, 1996). Um exemplo deste método é a linguagem Ada.

Nos últimos anos, as extensões paralelas vêm apresentando um progresso notável, em comparação às outras ferramentas. Estas extensões buscam oferecer um ambiente de programação para criar programas portáteis, buscando obter bom *speedup* em uma variedade considerável de sistemas paralelos e distribuídos (Almasi & Gottlieb, 1994; Ensink et al., 2003).

2.5. Ambientes de Passagem de Mensagem

Ambientes de passagem de mensagem são bibliotecas que, atuando em conjunto com uma linguagem hospedeira, como C ou Fortran, facilitam o desenvolvimento de aplicações paralelas.

Inicialmente, os ambientes de passagem de mensagem foram desenvolvidos pelos próprios fabricantes de máquinas de processamento maciçamente paralelo (MPP – *Massively*

Parallel Processing). Apesar desses ambientes serem logicamente similares, a sincronização e a troca de dados entre os processos utilizavam abordagens diferentes e, às vezes, incompatíveis. Esta situação resultou em um grande número de sistemas incompatíveis. Exemplos desses sistemas são o CROS da Caltech, o EUI da IBM e o NX da Intel (McBryan, 1994).

Para contornar esta especificidade existente entre fornecedor e máquina, as chamadas plataformas de portabilidade têm sido desenvolvidas por vários grupos de pesquisa. As plataformas de portabilidade são ambientes de passagem de mensagem desenvolvidas por não-fornecedores de MPP, com a idéia de definir um ambiente que seja implementado em várias plataformas de *hardware*, seja como uma biblioteca nativa de passagem de mensagem, seja através de chamadas apropriadas para o ambiente de passagem de mensagem residente. Assim, um programa escrito utilizando um ambiente de passagem de mensagem em plataforma de portabilidade torna-se portátil a todos os sistemas.

Como ambientes em plataformas de portabilidade são executados em várias arquiteturas, eles tendem a ser mais apropriados à computação heterogênea, onde dois ou mais computadores diferentes cooperam em uma tarefa. Este tipo de computação tem se tornado cada vez mais importante, devido à necessidade de unir recursos distintos. Exemplos de plataformas de portabilidade para computação heterogênea são P4, Parmacs, PVM, MPI, Linda (McBryan, 1994; Sunderam et al., 1994).

2.6. Considerações Finais

A computação paralela vem sendo utilizada buscando-se melhorar a velocidade com que os problemas podem ser resolvidos e também possibilitar a resolução de problemas complexos que antes eram inviáveis. Além da melhora no tempo de execução, também apresenta outras vantagens como a tolerância a falhas, já que uma falha em um dos elementos de processamento pode ser contornada com a utilização de outro processador disponível, implicando apenas numa possível queda do desempenho.

Quando se fala em melhora no desempenho, tornam-se necessárias algumas métricas para quantificar esta melhora alcançada. As principais medidas de desempenho para aplicações paralelas são o *speedup* e a eficiência.

Com o surgimento de novas arquiteturas, a classificação de Flynn tornou-se insuficiente, embora seja a classificação mais conhecida e utilizada. Muitos dos novos

computadores desenvolvidos não puderam ser facilmente encaixados nesta classificação. Várias outras classificações foram propostas, na tentativa de sanar a deficiência encontrada na classificação de Flynn. Dentre estas novas classificações, destaca-se a classificação de Duncan, com termos mais genéricos e apresentando alguns níveis de hierarquia.

O paralelismo inerente de certas classes de problemas e a limitação da velocidade das máquinas com um único processador são alguns dos fatores que justificam o uso da computação paralela. Apesar disso, problemas como a portabilidade de código e a dificuldade de se aprender uma linguagem paralela limitam a utilização mais ampla da computação paralela.

Para solucionar esses problemas, surgiram os ambientes de passagem de mensagem em plataformas de portabilidade. Estes ambientes são extensões paralelas de linguagens já conhecidas – o que evita a necessidade do aprendizado de novas linguagens – que buscam obedecer a um padrão que possibilitaria tornar as aplicações independentes de máquinas.

Os ambientes de passagem de mensagem ainda permitem que máquinas paralelas e sistemas distribuídos, compostos por rede de computadores interconectados, possam ser utilizados com a mesma facilidade.

Capítulo 3 – Simulação

3.1. Considerações Iniciais

A simulação tem sido amplamente empregada em medicina e outras áreas do conhecimento devido ao grau de realismo que pode ser incluído em seus modelos, e também à facilidade com o qual tais modelos podem ser explicados para pessoas que não possuem domínio técnico (Hoover & Perry, 1990).

Na análise de desempenho de sistemas computacionais, a simulação é uma técnica bastante útil, especialmente se o sistema a ser caracterizado não estiver disponível, como durante o estágio de projeto ou de aquisição. Um modelo de simulação provê uma maneira fácil de prever o desempenho ou comparar várias alternativas. Além disso, mesmo se um sistema estiver disponível para medição, um modelo de simulação pode ser preferido a medições uma vez que ele permite comparar as alternativas sob uma variedade de cargas e ambientes, sem a necessidade de provocar alterações no sistema real.

Apesar das vantagens no uso da simulação, ela apresenta características que restringem sua aplicação, como o tempo de execução da simulação sequencial. Como a simulação baseia-se em variáveis aleatórias, é necessário executá-la por um longo período, garantindo assim que esta aleatoriedade não influa nos resultados finais. Além disso, quando se deseja simular um sistema mais complexo ou com alto nível de detalhes, o tempo de simulação pode se tornar inviável.

Uma solução para reduzir esse tempo de execução é aliar os conceitos de simulação aos de computação paralela, resultando numa simulação distribuída ou simulação paralela. Nesse caso, a simulação é dividida em vários processos lógicos, que são executados em paralelo e se comunicam utilizando ambientes de passagem de mensagens.

A simulação distribuída tem como problema crítico a necessidade de sincronização entre os processos lógicos que compõem o sistema sendo simulado. Isto é necessário porque é preciso manter a ordem de execução de eventos apresentada no sistema real. Vários protocolos de sincronização foram desenvolvidos com esse intuito e os principais serão vistos neste capítulo.

3.2. Técnicas para Avaliação de Desempenho

Existem diversas técnicas para a avaliação de desempenho, assim como diversas são as classificações encontradas na literatura (Kobayashi, 1978; Jain, 1991; Santanta et al., 1994; DeRose et al., 2000). Uma das classificações é a divisão das técnicas em duas categorias: técnicas de aferição e técnicas de modelagem (Santanta et al., 1994). As técnicas de aferição obtêm as informações de interesse a partir de medições realizadas no sistema real. Nas técnicas de modelagem, desenvolve-se um modelo representativo do sistema, que é então resolvido ou analiticamente ou por simulação.

Fazem parte das **técnicas de aferição**: coleta de dados, *benchmarks* e construções de protótipos. A **coleta de dados** envolve uma monitoração do sistema enquanto ele está sendo submetido a uma carga em particular. Esta monitoração pode ser feita ou através de *software* ou por meio de *hardware* específico e, para que as medições efetuadas sejam significativas, a carga ao qual o sistema é submetido deve ser cuidadosamente selecionada. Tal carga pode ser *real* ou *sintética*. Com carga real, a observação é feita utilizando um sistema em sua operação normal e não pode ser repetida nas mesmas condições. A carga sintética, por sua vez, é gerada artificialmente, apresentando características similares às da carga real, com a vantagem de poder ser aplicada repetidamente e de maneira controlada (Jain, 1991; DeRose et al., 2000). Uma outra técnica de aferição é o **benchmark**, que utiliza carga sintética tendo como principal objetivo comparar o comportamento de vários sistemas sob a mesma carga (Santana et al, 1994).

A **construção de protótipos** baseia-se na construção de uma aproximação simplificada do sistema real, utilizada quando se deseja obter dados de um sistema ainda não disponível. A desvantagem de técnicas de aferição reside no fato de ser aplicável apenas a sistemas existentes, além de apresentar o incômodo de ter que se alterar o sistema real. Por exigir sistemas reais, instrumentos e tempo, as técnicas de aferição são as que apresentam maior custo, embora possam apresentar resultados bastante precisos.

As **técnicas de modelagem** compreendem a modelagem analítica e a simulação. A **modelagem analítica** utiliza métodos matemáticos para obter a informação sobre os tópicos de interesse. O problema é formulado através de uma série de equações e esta técnica pode ser utilizada quando as relações que compõem o modelo são suficientemente simples. Como a maioria dos problemas de desempenho é relacionada a atrasos de fila causados pela retenção de recursos, geralmente utiliza-se a teoria de filas para a formulação desses problemas (Law & Kelton, 1991; Kobayashi, 1978; Soares, 1990). A desvantagem da modelagem analítica é que a quantidade de simplificações e suposições, muitas vezes necessárias, podem comprometer a precisão. Em contrapartida, apresenta baixo custo e, das três técnicas apresentadas, é a que consome menos tempo.

A simulação é um paradigma computacional usado para permitir que um programa de computador emule o comportamento de certos tipos de modelos físicos. Pode ser aplicada tanto a sistemas existentes quanto inexistentes. Sua flexibilidade permite que sejam construídos modelos detalhados e com alta precisão. Como o modelo analítico muitas vezes não é matematicamente tratável para sistemas de maior complexidade, a simulação torna-se a única alternativa quando se deseja avaliar o desempenho de um sistema não existente. A principal desvantagem deste método é a necessidade de uma validação cuidadosa de seus resultados (Nicol, 2000; Kobayashi, 1978; Soares, 1990; MacDougall, 1978; Jain, 1991; DeRose et al., 2000). O custo desta solução, juntamente com a facilidade de modificar as configurações do sistema, costuma ser a razão para se desenvolver simulações para sistemas de alto custo (Jain, 1991).

A escolha de qual das técnicas utilizar depende de vários fatores. Por exemplo, o *estágio do ciclo de vida* no qual se encontra o sistema, o *tempo disponível*, o *custo da avaliação* e o *nível de precisão desejado*. Muitas vezes duas técnicas são utilizadas simultaneamente, de modo que uma seja utilizada para verificar e validar o resultado da outra (Jain, 1991; Hoover & Perry, 1990).

3.3. Ciclo de Vida da Simulação

A figura 3.1, extraída de Osman (Osman, 2000), representa o ciclo de vida de um estudo de simulação. Diferentes autores apresentam diferentes modelos de ciclo de vida de simulações. No entanto, a maioria das fases apresentadas a seguir são comuns às de outros modelos encontrados em (Banks, 1998; Hoover & Perry, 1990; MacDougall, 1989; Soares, 1990).

O ciclo de vida para o desenvolvimento de simulações é naturalmente interativo e não deve ser interpretado como estritamente sequencial. Associadas a cada fase do ciclo de vida estão atividades de verificação, validação e teste. A atividade de verificação visa comprovar se o modelo foi transformado de uma forma à outra como pretendido, com precisão suficiente. A validação determina se o modelo comporta-se com precisão satisfatória dentro de seu domínio de aplicabilidade, e os testes buscam averiguar se há erros ou inexatidão no modelo. Estão disponíveis na literatura diversas técnicas para verificação, validação e testes de modelos e de programas de simulação (Osman, 2000; Jain, 1991; Law & Kelton, 1982; Kobayashi, 1978).

Essas três atividades, verificação, validação e testes, não se tratam de fases ou passos no ciclo de vida da simulação, mas sim uma atividade contínua a ser feita durante todo o ciclo da figura 3.1. Pretende-se com estas atividades revelar as deficiências que podem surgir à medida que o estudo de simulação progride. Assim, identificando e corrigindo tais deficiências durante a fase do ciclo de vida na qual eles aconteceram, evita-se que as consequências desses erros se propaguem ao longo do desenvolvimento.

Os 10 processos do ciclo de vida estão representados na figura 3.1 pelas setas tracejadas e encontram-se descritos a seguir. As fases, pelos símbolos de formato oval. As setas contínuas referem-se aos estágios de credibilidade de avaliação.

Formulação do problema: A formulação (ou definição) do problema é o processo pelo qual o problema comunicado é convertido para um problema formulado e bem definido, de modo a possibilitar uma ação de pesquisa específica. É um dos passos mais importantes numa análise de simulação, já que uma formulação deficiente irá comprometer seriamente os resultados alcançados (Hoover & Perry, 1990).

Investigação de técnicas de solução: Nesta fase deve ser feito um levantamento das técnicas alternativas que podem ser utilizadas para resolver o problema formulado, descartando-se as técnicas consideradas muito caras ou inadequadas. Entre as técnicas qualificadas, deve ser escolhida aquela que ofereça a melhor razão custo/benefício, que seja suficientemente confiável e se apresente mais apropriada. Por exemplo, uma outra técnica pode apresentar uma solução de custo menor do que a simulação, mas pode não ser tão útil.

Figura 3.1: Ciclo de vida da simulação

Investigação do sistema: devem ser investigadas as características do sistema onde reside o problema formulado, tendo em vista os objetivos do estudo identificados no processo de formulação do problema. Entre as principais características do sistema, pode-se citar a frequência das mudanças no sistema ao longo do tempo (que irá determinar a necessidade de atualizações no modelo) e o ambiente onde se insere o sistema, representado pelas variáveis de entrada que podem afetar o estado do sistema.

Formulação do modelo: durante este processo geram-se os requisitos dos dados de entrada. Os valores de entrada podem ser ou hipotéticos ou baseados em alguma análise preliminar. Os modelos que usam dados hipotéticos, obtidos de amostragem a partir de distribuições de probabilidade utilizando números aleatórios, são chamados de *self-driven*. Já os que utilizam seqüências de entradas derivadas de dados obtidos através de medições do sistema real são classificados de *trace-driven*.

Representação do modelo: é feita uma especificação do modelo da simulação, seja com o uso de grafos, fluxogramas, pseudocódigo ou outras técnicas de diagramação. Tem como resultado um *modelo comunicativo*, que possibilita a outras pessoas além do modelador

entender o modelo representado. A forma de representação deve ser escolhida baseada em sua aplicabilidade para descrever o sistema em estudo, no conhecimento técnico das pessoas que irão utilizar o modelo, no quanto eles se prestam para análise formal e verificação, em seu suporte para a documentação e manutenção do modelo e em sua capacidade de tradução para um modelo programado.

Programação: é a transferência do modelo comunicativo representado no processo anterior em um modelo identificável pelo computador (modelo executável). Neste processo não se incorpora ainda um projeto de experimentos. Este processo pode ser executado utilizando-se tanto um *software* de simulação, como uma linguagem de programação para simulação ou uma linguagem de alto nível.

Projeto de experimentos: é a formulação de um plano para se coletar a informação desejada ao custo mínimo e possibilitar que o analista extraia inferências válidas. Um modelo experimental é o modelo de programação incorporando uma descrição de operação executável, apresentada nesse plano. Existe uma variedade de técnicas para o projeto de experiências.

Experimentos: este é o processo onde se busca fazer experiências com o modelo de simulação visando um propósito específico. Esses experimentos podem ser feitos para fins de treinamento, comparação entre políticas operacionais diferentes, avaliação do comportamento do sistema, análise de sensibilidade, previsões, otimização e determinação de relações funcionais. O processo de experimentos produz os resultados da simulação.

Redefinição: procura-se neste processo atualizar o modelo experimental de modo que ele represente a forma atual do sistema; alterá-lo para obter um outro conjunto de resultados; mudá-lo para manutenção; modificá-lo para outro uso; ou redefinir o modelo em um novo sistema para estudar uma solução alternativa para o problema.

Apresentação dos resultados: neste processo, os resultados da simulação são interpretados e apresentados. A apresentação deve ser feita dependendo do uso pretendido do modelo. Se o modelo é utilizado num ambiente "e se...", os resultados devem ser integrados para auxiliar no processo de tomada de decisão. O relatório documentando o estudo e seus resultados, junto com sua apresentação, também constitui uma forma de auxiliar a tomada de decisão.

3.4. Conceitos Básicos de Simulação Discreta

Basicamente, todo modelo de simulação é uma especificação de um sistema físico em termos de um conjunto de eventos e estados. Fazer uma simulação é imitar a ocorrência desses eventos à medida que eles evoluem no tempo e reconhecer os seus efeitos, representados pelos estados. Se as mudanças no estado do sistema ocorrem suave e continuamente ao longo do tempo, tem-se a *simulação contínua*. Como exemplos de utilização de simulação contínua podem-se citar as equações diferenciais, que dependem do tempo, ou fenômenos físicos, como modelos de astrofísica, ou interações químicas. Na *simulação discreta* as mudanças no sistema ocorrem somente em pontos específicos no tempo, de modo instantâneo. A simulação de modelos digitais é naturalmente suscetível a esta abordagem, assim como a simulação de sistemas de tráfego aéreo e de fenômenos biológicos – como a disseminação de uma população ou de um vírus (MacDougall, 1989; Fujimoto, 1990; Jain, 1991; Nicol, 2000; Tropper, 2002).

O comportamento dinâmico de sistemas discretos é descrito através de três entidades: atividades, processos e eventos. A **atividade** é a menor unidade de trabalho e a ela está associado seu tempo de execução. Um **processo** corresponde a um conjunto de atividades logicamente relacionadas e seu tempo de execução é obtido através da soma do tempo de execução de suas atividades mais o atraso. Um **evento** corresponde a uma mudança de estado de qualquer entidade do sistema. Um sistema é visto dinamicamente como uma coleção de processos que se interagem, com interações controladas e coordenadas pela ocorrência de eventos.

Tipicamente, são utilizadas três estruturas de dados em simulação discreta: as *variáveis de estado*, que descrevem o estado do sistema, uma *lista de eventos* contendo todos os eventos pendentes que foram escalonados e um *relógio global* que denota até onde a simulação progrediu. Cada evento contém uma marca de tempo (um *timestamp*) e geralmente indica alguma mudança no estado do sistema sendo simulado. O *timestamp* indica quando essa mudança irá ocorrer no sistema real.

A simulação executa um laço cujo corpo remove o evento com o menor *timestamp* da lista de eventos e efetua os cálculos descritos pelo evento para modificar os estados do modelo. Como a execução do evento pode causar a geração de outros eventos no futuro, parte desta tarefa é inserir novos eventos na lista de eventos. Depois de executar o último evento com *timestamp* t (pode ser que haja múltiplos deles), o valor das estruturas de dados que codificam o modelo refletem o estado no tempo t .

A simulação discreta tradicional, como descrita anteriormente, é sequencial. No entanto, muitas simulações práticas, como em aplicações de engenharia, consomem várias horas ou até mesmo dias utilizando-se máquinas sequenciais. A computação paralela oferece ferramentas atrativas a serem usadas para reduzir o tempo de execução de tais programas de simulação (Ayani, 1993).

3.5. Técnicas para Decomposição da Simulação

Para que uma simulação sequencial seja executada em múltiplos processadores, é preciso primeiramente efetuar sua decomposição, o que pode ser feito de diversas maneiras (Vee & Hsu, 1999):

1. *Compiladores Paralelizadores*: este método é facilmente aplicável a programas de simulação existentes, pois não requer que sejam feitas alterações no código. Por outro lado, implica em um grau de paralelismo limitado, já que o compilador ignora totalmente a estrutura do problema.
2. *Testes Replicados – Replicated Trials* (ou *Parallel Independent Runs, Distributed Experiments*): executam-se independentemente n simulações sequenciais em n processadores, calculando-se a média de seus resultados ao final. Por não haver coordenação entre os testes, espera-se alta eficiência. Entretanto, os parâmetros de todas as execuções da simulação devem ser decididos antes do início da simulação, o que torna este método impróprio a ambientes de projeto onde os resultados de um experimento são usados para determinar o próximo (Fujimoto, 1990).
3. *Funções Distribuídas*: associam-se as subtarefas essenciais de uma simulação (como a geração de números aleatórios, manipulação de arquivos, coleta de estatísticas) a um certo número de processadores. A vantagem deste método é que ele exige mudanças mínimas no código sequencial existente. Apesar disso, é difícil balancear a carga entre os processadores e não se pode esperar muito paralelismo, pois a quantidade dessas subtarefas é limitada.
4. *Eventos Distribuídos (com Lista de Eventos Centralizada)*: neste método, o processador que se torna disponível passa a processar o próximo evento com o menor *timestamp* da lista de eventos global. Para evitar simulação incorreta, cada processador deve assegurar que o evento com o menor *timestamp* da lista

não será cancelado por eventos já computados em outros processadores. Deve assegurar também que os eventos processados concorrentemente estão em consistência com a semântica do sistema sendo simulado. Para isto é necessário o conhecimento do modelo de simulação, o que não é fácil de ser extraído. Além disso, se houver muitos processadores envolvidos na simulação, a lista de eventos global pode se tornar um gargalo.

5. *Decomposição de domínio*: baseia-se no ponto de vista de que executar uma simulação equivale a preencher uma região bidimensional. Uma dimensão representa o tempo simulado e outra, as variáveis de estado, como pode ser visto na figura 3.2. Assim, o domínio espaço-tempo pode ser decomposto em relação ao tempo, como na figura 3.2 (a), ou ao espaço, como na figura 3.2 (b).
 - a. *Decomposição de espaço paralelo (ou Simulação Distribuída)*: decompõe-se o modelo em submodelos ou componentes no domínio espaço. Cada componente é associado a um processo, e vários processos podem ser executados no mesmo processador. É uma decomposição atrativa, pois pode ser aplicável a qualquer modelo e mostra o maior potencial por oferecer desempenho escalável a modelos grandes.
 - b. *Decomposição de tempo paralelo (ou Paralelismo Temporal)*: o domínio é particionado em intervalos $[t_{i-1}, t_i]$ para $0 < i \leq p$, sendo p o número de processadores. Um intervalo é associado a cada processador, e este fica responsável por computar os valores das variáveis de estado dentro daquele intervalo, assegurando que o estado do sistema ao final de $[t_{i-1}, t_i]$ deve igualar-se ao estado inicial de $[t_i, t_{i+1}]$. Caso alguma inconsistência entre estados seja detectada, é necessário recalcular alguns intervalos. Assim, a eficiência deste método depende da possibilidade de se prever precisamente o estado inicial de cada intervalo e é aplicado a problemas de simulação específicos (Fujimoto, 1998).

Figura 3.2: Estratégias de decomposição de domínio

Entre os métodos de decomposição apresentados, o método de decomposição de espaço paralelo mostra o maior potencial, sendo considerado método mais promissor para executar simulação discreta em paralelo (Vee & Hsu, 1999). Como a lista de eventos também é decomposta, a lista de eventos não se tornaria o gargalo como no método de eventos distribuídos.

Existe ainda uma classificação da decomposição da simulação na qual os métodos existentes são divididos em duas abordagens: **SRIP** (*Single Replication In Parallel*) e **MRIP** (*Multiple Replication In Parallel*) (Ewing et al., 1999; Mota et. al., 1999). A abordagem *SRIP* compreende as técnicas que realizam a divisão do modelo de simulação entre vários processadores. A abordagem *MRIP* foi proposta e implementada independentemente no início dos anos 90 por (Sunderam & Rego, 1991; Paulikowski & Yau, 1992). Nesta abordagem, vários computadores de uma rede local executam simulações sequenciais independentes do mesmo modelo e enviam seus resultados a analisadores de dados globais responsáveis pelas medidas de desempenho. Estes analisadores globais são responsáveis por controlar a precisão dos resultados finais e determinar o término da simulação (Biles & Kleijnen, 2003).

Os primeiros trabalhos em decomposição do modelo de simulação ao longo da dimensão espaço referiram ao método como *simulação distribuída* (Chandy & Misra, 1979; Bryant, 1977), sendo a simulação que pretendia ser executada em uma rede de processadores. Trabalhos posteriores vezes usam também o termo *simulação paralela*, ou *parallel discrete event simulation (PDES)* ou *simulação paralela e distribuída* (Ferscha, 1996; Fujimoto, 1990). Esses termos serão tratados como sinônimos neste trabalho.

3.6. Simulação Distribuída

A simulação distribuída tem atraído um grande número de pesquisas nos últimos anos devido a vários fatores, entre eles, o potencial para reduzir o tempo de simulação de um programa de simulação discreta e também pelo fato de muitos sistemas da vida real conter quantias substanciais de paralelismo, de modo que seja natural simular tais fenômenos em paralelo (Ayani, 1993). Outro benefício do uso de técnicas de simulação distribuída é na criação de ambientes virtuais geograficamente distribuídos. Além disso, o uso de múltiplos processadores pode prover tolerância a falhas. Se um processador falhar, os demais podem prosseguir com a simulação, desde que os elementos críticos à simulação não estejam no processador onde houve a falha (Fujimoto, 2001).

Em simulação distribuída, o sistema (geralmente referido como *sistema físico*) é modelado como um conjunto de n processos lógicos $LP_1, LP_2, \dots, LP_n$, cada um representando um processo do sistema real. Esses processos lógicos interagem entre si enviando mensagens de eventos com *timestamp*. Existirá uma ligação (ou um *canal de comunicação*) de LP_i a LP_j se LP_i puder enviar mensagens a LP_j . Cada mensagem é uma tupla (E_k, T_k) composta por um evento E_k e seu *timestamp* T_k . Cada processo lógico é associado a um relógio local $clock_i$, ou um tempo virtual local (*local virtual time – LVT*), que indica até onde o tempo simulado avançou. Quando E_k é processado, $clock_i$ é avançado automaticamente a T_k .

Uma das principais dificuldades em simulação distribuída encontra-se na sincronização. Esta dificuldade vem das relações de precedência – ou relações de causa e efeito – que ditam quais eventos devem ser executados antes de outros. Em geral, essas relações são muito complexas e dependentes de dados.

É muito importante que se observe a relação de causa e efeito para se garantir a precisão da simulação. Para entender a relação de causa e efeito, considere dois eventos E_1 e E_2 com *timestamps* T_1 e T_2 , respectivamente, e $T_1 < T_2$. Se E_2 depende de E_1 , então existe uma restrição sobre a ordem de execução de E_1 e E_2 , que será chamada de *restrição de causa e efeito*. Por exemplo, se E_1 causa uma mudança de uma variável de estado que será referenciada por E_2 , então E_1 deve ser executado antes de E_2 , caso contrário irá ocorrer um erro de causa e efeito.

A condição formulada como *restrição de causa e efeito local* (*local causality constraint*) estabelece que uma simulação paralela obedece à restrição de causa e efeito se e somente se todo processo lógico executar eventos em ordem não-decrescente de *timestamp* (Misra, 1986). Esta condição é suficiente, porém não necessária: se dois eventos no mesmo

processo lógico possuem *timestamps* diferentes mas não há dependência direta ou indireta entre eles, então eles podem ser executados em paralelo sem produzir erros de causa e efeito.

Violar a restrição de causa e efeito significa que o futuro pode afetar o passado, o que pode resultar em comportamento anômalo e conseqüentemente, em simulação incorreta. É responsabilidade do *mecanismo de sincronização* assegurar interações exatas e corretas entre os *LPs*.

Os métodos de sincronização em simulação distribuída geralmente dividem-se em duas categorias – *conservativos* e *otimistas* – de acordo como eles abordam a restrição de causa e efeito local. Os métodos *conservativos* evitam qualquer erro de causa e efeito possível, obedecendo rigorosamente à restrição de causa e efeito local. Os métodos *otimistas*, por outro lado, tentam explorar a probabilidade de não produzir erros de causa e efeito e não aderem estritamente à restrição de causa e efeito. O método otimista garante detectar quaisquer ocorrências de erros de causa e efeito e provê um mecanismo de *rollback* para restaurar a simulação a um estado correto.

Existe ainda uma abordagem que visa minimizar as sobrecargas impostas pela sincronização, adotando um comportamento denominado *anarquista* ou *assíncrono* (Rao et al., 1998; Thondugulam et al., 1999; Dota, 2001): os erros de causa e efeito simplesmente são ignorados. A execução da simulação é feita mais rapidamente, porém seus resultados são aproximados. Estes três métodos serão discutidos nas seções seguintes.

3.7. Protocolos Conservativos

Historicamente, o primeiro mecanismo de simulação paralela foi baseado em métodos conservativos. No final dos anos 70, Chandy, Misra e Bryant (Bryant, 1977; Chandy & Misra, 1979; Misra, 1986) desenvolveram independentemente os algoritmos conhecidos como os protocolos ***Chandy-Misra-Bryant (CMB)***.

Nesse método, um sistema físico é modelado como um grafo direcionado onde os arcos representam os canais de comunicação entre os processos lógicos. Cada processo lógico mantém um conjunto de filas, cada fila é associada a um arco no grafo e em cada processo os eventos são simulados na ordem estrita de seu tempo simulado. A comunicação entre processos ocorre sempre que um evento associado a um processo lógico deseja escalonar um evento em outro processo lógico. Assume-se que o meio de comunicação

preserva a ordem de mensagens e que o *timestamp* das mensagens enviadas através de todo canal seja não-decrescente.

Um processo lógico não pode executar um evento com *timestamp* t até que se tenha certeza de que não irá receber mensagens de evento com *timestamp* menor que t , o que é garantido selecionando-se sempre o evento com o menor *timestamp*. Se cada arco de entrada de um nó possuir no mínimo uma mensagem não processada, então a próxima mensagem a ser processada é aquela com o menor *timestamp* entre todos dos arcos de entrada do nó. Os eventos que podem ser processados sem perigo são chamados de *eventos seguros*. No entanto, se algum dos arcos de entrada estiver vazio, não há evento seguro e portanto o nó ficará *bloqueado*, esperando alguma mensagem chegar. O mecanismo de bloqueio é necessário porque se um nó processar uma mensagem de uma de suas filas de entrada não-vazias, não há garantia de que uma mensagem que chegar depois a um arco de entrada vazio terá um *timestamp* maior ou igual do que o *timestamp* da mensagem processada (Ayani, 1993; Vee & Hsu, 1999; Fujimoto, 1998; Boukerche, 2002).

Existem dois problemas relacionados ao bloqueio de um nó: *estouro de memória* e *deadlock* (Ayani, 1993; Ferscha, 1996).

- (a) **Deadlock:** se o grafo direcionado representando o sistema contém um ciclo, como mostrado pelas setas contínuas da figura 3.3, então o paradigma Chandy-Misra é vulnerável a *deadlock*. Vários métodos foram propostos na literatura para resolver o problema de *deadlock*, como a utilização de mensagens nulas para preveni-lo (Chandy & Misra, 1979; Bryant, 1977), o envio de mensagens nulas sob demanda (Misra, 1986) ou a detecção de *deadlock* e posterior recuperação (Misra, 1986; Chandy & Misra, 1981). Estes métodos são ou baseados em prevenção ou detecção e recuperação de *deadlock*.
- (b) **Estouro de memória:** enquanto um nó estiver bloqueado porque há fila de entrada vazia, as outras filas podem crescer, levando a uma exigência de armazenamento imprevisível. Por exemplo, considere o sistema mostrado na figura 3.3. Os processos lógicos encontram-se bloqueados, mas as mensagens continuam a chegar pelos canais indicados pelas setas tracejadas.

Figura 3.3: Ocorrência de *deadlock* e estouro de memória

3.7.1. Prevenção de Deadlock Utilizando Mensagens Nulas

O método CMB utiliza *mensagens nulas* para evitar *deadlock* e requer que os canais de comunicação entre os processos lógicos sejam especificados estaticamente. Neste algoritmo de prevenção de *deadlock*, um *LP* envia uma mensagem nula ($null, T_{null}$) como uma garantia de que ele não enviará uma mensagem com *timestamp* menor do que T_{null} . Uma mensagem nula possui propósito apenas de sincronização e não corresponde a nenhum evento no sistema físico. Pode-se mostrar que um algoritmo de mensagem nula evita situações de *deadlock* contanto que não haja ciclo com incremento zero de *timestamp* (Chandy & Misra, 1979), isto é, um ciclo no qual uma mensagem possa percorrer em tempo zero.

Um *LP* pode determinar o *timestamp* de uma mensagem nula verificando o valor do *clock* de cada canal de entrada e o tempo que irá decorrer ao processar um evento. Sempre que um *LP* terminar de processar um evento, ele pode enviar mensagens nulas a todos os *LP* que se ligam a ele. Um problema bem conhecido com este método é a possibilidade de gerar uma quantia considerável de mensagens nulas, o que poderia transbordar a largura de banda de comunicação e provocar um atraso significativo na simulação (Vee & Hsu, 1999; Fujimoto, 2001).

Foram propostas variações do algoritmo para diminuir o número de mensagens nulas. Em algumas variações (Misra, 1986), as mensagens nulas são enviadas com base na demanda. O tráfego de mensagens nulas também pode ser reduzido pela informação de *lookahead*.

O *lookahead* refere-se à capacidade de prever o que irá acontecer na simulação, ou melhor, o que *não* irá acontecer no futuro simulado (Fujimoto, 1998; Vee & Hsu, 1999). Uma forma óbvia de *lookahead* é o incremento mínimo de *timestamp* para um *LP* processar um evento. Se um LP_i com relógio local $clock_i$ requer no mínimo l unidades de tempo

simulado para processar qualquer evento, então diz-se que o LP_i possui um *lookahead* l , porque ele pode garantir que não irá gerar mensagens com *timestamps* menores que $clock_i + l$. Além de reduzir o tráfego de mensagens nulas, a informação de *lookahead* também ajuda a reduzir a possibilidade de ocorrência de *deadlocks* e assim acelerar a simulação.

Quando o valor do *lookahead* é conhecido antes do início da simulação, este é conhecido como *lookahead* **estático**, ou *lookahead* explícito. Como valor de *lookahead* estático pode ser utilizado, por exemplo, o tempo mínimo de serviço de um processo lógico. Se o *lookahead* pode assumir diferentes valores ao longo da simulação, tem-se então *lookahead* **dinâmico** (Zarei & Pidd, 2001).

Vários métodos para o cálculo do *lookahead* dinâmico têm sido propostos com o objetivo de melhorar o desempenho da simulação conservativa. Em (Preiss & Loucks, 1989; Nicol, 1996) encontra-se uma sumarização de diversas técnicas, sendo que a maioria envolve a pré computação de dados como tempo de serviço e variáveis aleatórias. Outras técnicas mais atuais envolvem a análise do fluxo do grafo formado pelos recursos do modelo sendo simulado, considerando o algoritmo do menor caminho combinado à pré amostragem das variáveis envolvidas (Boukerche & Tropper, 2001; Nketsa & Khalifa, 2001; Zarei & Pidd, 2001).

3.7.2. Detecção e Recuperação de Deadlock

Uma alternativa para tratar *deadlock* foi proposta por Chandy e Misra (Misra, 1986; Chandy & Misra, 1981). Este método elimina a geração de mensagens nulas e, ao invés de evitar o *deadlock*, permite-se que a simulação entre em *deadlock*. Para este método, são necessários dois mecanismos: um para detectar a ocorrência do *deadlock* e outro para resolvê-lo.

Um *deadlock* pode ser detectado utilizando-se um método similar ao usado em computação distribuída, baseando-se no fato de que a mensagem com o menor *timestamp*, em qualquer ponto da simulação, é sempre segura para ser processada. No trabalho de Misra (Misra, 1986) encontram-se descritos mecanismos para detecção e recuperação do *deadlock*.

As vantagens do método de detecção e recuperação encontram-se em evitar completamente o tráfego de mensagens nulas, assim como em permitir a existência de ciclos com incremento zero de *timestamp* (embora isto geralmente implique em baixo desempenho). O método, no entanto, geralmente resulta em execuções sequenciais, o que pode afetar desfavoravelmente o desempenho global se o modelo de simulação for propenso a *deadlocks*.

Os *deadlocks* podem ocorrer com certa frequência se houver relativamente poucas mensagens, comparadas ao número de canais na rede (Vee & Hsu, 1999).

3.7.3. Janelas de Tempo Conservativas

O método de janelas de tempo é utilizado para reduzir a sobrecarga associada ao se determinar quando é seguro processar um evento. O limite inferior da janela é o menor *timestamp* entre todos os eventos não processados; já o limite superior depende do tamanho da janela. Somente os eventos não processados cujos *timestamps* residem entre os limites da janela são elegíveis para o processamento.

Uma questão importante é o método utilizado para se determinar o tamanho da janela de tempo. Se a janela for muito pequena, existirão poucos eventos disponíveis para execução concorrente. Por outro lado, se a janela for muito grande, o mecanismo de simulação comporta-se de modo semelhante à simulação sem o uso de janelas, implicando numa sobrecarga para gerenciar o mecanismo de janelas que não é justificada. Ajustar a janela a um tamanho apropriado requer informação específica da aplicação (Fujimoto, 1990).

3.7.4. Eventos Condicionais

O método de *eventos condicionais* (Chandy & Sherman, 1989) é um método onde os eventos são classificados em *eventos definidos* e *eventos condicionais*. Os eventos definidos, que geralmente possuem *timestamps* menores, serão processados definitivamente e não poderão ser cancelados nem desativados por outros eventos. Eventos condicionais, por outro lado, serão processados somente quando certos critérios forem satisfeitos. Claramente, o evento com o menor *timestamp* no sistema será sempre um evento definido (Vee & Hsu, 1999).

Processar eventos definidos é sempre seguro. O método de eventos condicionais explora esta propriedade e é capaz de distinguir os eventos definidos dos condicionais utilizando *conhecimento condicional* (Chandy & Misra, 1987). Uma característica deste método que o distingue de outros é que a comunicação não está restrita somente àqueles processos que trocam mensagens de eventos durante a simulação. Pares arbitrários de processos podem enviar mensagens entre si para determinar quais eventos são seguros e assim converter eventos condicionais em definitivos (Fujimoto, 1990).

Além dos métodos apresentados, existem propostas de alternativas que utilizam informações específicas à aplicação, a topologia de rede de processos lógicos do modelo, a arquitetura de hardware, etc., sempre visando explorar maior paralelismo. Alguns dos métodos e melhorias são descritos em (Ferscha, 1996; Fujimoto, 1998).

3.8. Protocolos Otimistas

Os métodos otimistas não aderem estritamente à restrição de causa e efeito. Ao contrário dos mecanismos conservativos, os otimistas não precisam determinar quando é seguro processar um evento. Desde que existam eventos a serem processados, o protocolo permite que a simulação prossiga sem garantir que o conjunto de eventos gerados estará consistente com a restrição de causa e efeito. Este método é otimista porque ele assume que as mensagens entre os processos lógicos chegam no tempo correto, e assim os processos lógicos podem ser processados independentemente. Caso ocorram erros de causa e efeito, eles são detectados e então um procedimento de recuperação é executado (Ayani, 1993; Ferscha, 1996; Fujimoto, 1990; Vee & Hsu, 1999; Overeinder, 1991; Boukerche, 2002).

Este método tem por vantagens apresentar *speedup* potencialmente maior do que os métodos conservativos, pois permite a exploração do paralelismo em situações onde erros de causa e efeito *podem* ocorrer, mas de fato não ocorrem. Além disso a topologia das possíveis interações entre os processos não precisa ser previamente conhecida, já que permite a criação dinâmica de processos lógicos (Fujimoto, 1990; Overeinder, 1991). O mecanismo otimista mais conhecido é o *Time Warp* (Jefferson, 1985), descrito na seção 3.8.1.

3.8.1. Time Warp

O mecanismo *Time Warp* é baseado no paradigma *Virtual Time* (Jefferson, 1985) e apresenta dois componentes principais: o *mecanismo de controle local*, responsável por garantir que os eventos são simulados na ordem correta, e o *mecanismo de controle global*, responsável pelo gerenciamento da memória e detecção do fim da simulação.

Um erro de causa e efeito é detectado sempre que uma mensagem com *timestamp* menor do que o relógio do processo for recebida (esta mensagem é chamada de *straggler*). Para restaurar a simulação a um estado correto, o processo lógico executa um mecanismo de *rollback* e retorna a um estado salvo com tempo simulado anterior ao tempo marcado pelo *straggler*. A simulação é então reiniciada a partir deste estado. Além de efetuar o *rollback*

neste processo lógico, o mecanismo *Time Warp* deve cancelar todos os efeitos colaterais causados pelas mensagens enviadas por este processo lógico com *timestamp* maior que o tempo ao qual ele retornou. Isto é feito enviando-se *antimensagens*, ou mensagens *negativas*, para cancelar as mensagens originais correspondentes (também referidas como mensagens *positivas*). Se um processo receber uma antimensagem correspondente a uma mensagem positiva já processada, então este processo também deve efetuar um *rollback* para desfazer os efeitos de ter processado a mensagem positiva que deveria ser destruída. Repetir este procedimento recursivamente permite que todos os efeitos da computação errônea sejam cancelados (Ayani, 1993; Fujimoto, 1990; Boukerche, 2002).

Um problema sério com este protocolo é a necessidade de se salvar periodicamente o estado de cada processo lógico, o que demanda grande quantidade de memória. No *Time Warp*, o menor *timestamp* entre todos os eventos não processados (tanto positivos quanto negativos) é chamado de *tempo global virtual* (*Global Virtual Time – GVT*). Como nenhum processo lógico produzirá eventos com *timestamp* menor do que o GVT, os estados salvos, que possuam *timestamp* menor do que o GVT podem ser descartados. Além disso, operações irrevogáveis (como entrada/saída) não podem ser validadas até que o GVT avance além do tempo de ocorrência simulado dessas operações. Estes processos de recuperação de memória e validação de operações irrevogáveis são chamados de *coleta de fóssil* (Ayani, 1993; Fujimoto, 1990; Overeinder, 1996). Entretanto, a sobrecarga gerada pelas operações de armazenamento dos estados e coleta de fóssil pode degradar o desempenho do protocolo.

3.8.2. Variações no Time Warp

O algoritmo *Time Warp* original utiliza **cancelamento agressivo**: sempre que um processo lógico executa um *rollback*, as antimensagens são enviadas imediatamente a todos os processos que receberam erroneamente. Para reduzir o tráfego de antimensagens e também a frequência de *rollbacks*, foi proposta uma estratégia de **cancelamento preguiçoso**.

No cancelamento preguiçoso, ao invés de enviar as antimensagens imediatamente, espera-se para ver se a reexecução da simulação gera a mesma mensagem. Se isto ocorrer, não há necessidade de cancelá-la. Embora o cancelamento preguiçoso reduza o tráfego de mensagens, ele pode permitir que uma computação errônea se propague além do que aconteceria se utilizasse o cancelamento agressivo, o que pode degradar o desempenho se houver muita computação incorreta. O cancelamento preguiçoso também requer memória adicional para gravar as mensagens positivas enviadas e há a sobrecarga de se comparar os

eventos gerados com as eventuais antimensagens. Apesar disso, (Fujimoto, 1990; Vee & Hsu, 1999) relatam que estudos teóricos e empíricos sugerem que o cancelamento preguiçoso tende a executar no mínimo tão bem quanto o cancelamento agressivo, se não melhor.

A estratégia de **reavaliação preguiçosa** é, de certa forma, similar ao cancelamento preguiçoso, com a diferença de lidar com vetores de estados ao invés de mensagens. Se um LP com relógio local $clock_i$ receber um *straggler* com timestamp t_{str} e se não houver mudanças de estado no LP no período de t_{str} até $clock_i$, o LP não precisa voltar. A reavaliação preguiçosa evita que os LPs executem uma recomputação desnecessária de estados. No entanto, ele requer memória adicional e sobrecarga que pode complicar significativamente a codificação de protocolos otimistas.

Uma estratégia simples, porém pouco utilizada, é o cancelamento utilizando o envio de **antimensagens por rajadas** (*bulk anti-messages*). Neste método, quando um *rollback* se faz necessário, apenas uma antimensagem é enviada para cada processo de destino. O tempo considerado para essa antimensagem é o tempo de criação do evento mais antigo que deve ser desconsiderado no processo de destino. Uma otimização desta estratégia é sua combinação com a estratégia de cancelamento preguiçoso. Assim, ao detectar a necessidade do envio de antimensagem na reexecução da simulação, apenas uma antimensagem por rajada é utilizada (Iskra et al, 2003).

Assim como nos métodos conservativos, mecanismos de **janelas de tempo otimistas** também foram propostos para métodos otimistas. Nos métodos otimistas, as janelas de tempo são utilizadas para evitar que a computação incorreta se propague muito além no tempo simulado. Críticos deste método apontam desvantagens como a falta de clareza em se determinar o tamanho das janelas; a dificuldade de se distinguir, dentro da janela, uma computação válida de uma incorreta; e também o obstáculo imposto pela janela de se impedir que eventos corretos além do limite superior da janela sejam processados (Ayani, 1993; Fujimoto, 1998).

Um levantamento de outras variantes do *Time Warp* e de métodos híbridos podem ser encontrados em (Fujimoto, 1990; Ferscha, 1996; Spolon et al., 2000).

3.9. Protocolo Assíncrono

O protocolo assíncrono ignora as violações de causa e efeito na tentativa de se reduzir o tempo de execução da simulação, eliminando a sobrecarga de sincronização de eventos. Ele

considera que a maior sobrecarga na simulação distribuída está relacionada à necessidade de sincronização dos eventos distribuídos: a sobrecarga no mecanismo de detecção e recuperação de *deadlocks* e cálculo do *lookahead* em simulação distribuída e a ocorrência de *rollbacks*, gravação de variáveis de estado e cálculo do GVT nos protocolos otimistas (Rao et al., 1998; Thondugulam et al. 1999).

Para uma certa classe de aplicações (por exemplo, sistemas que apresentam comportamento estocástico) houve ganhos significativos no tempo de execução da simulação e os valores resultantes apresentaram-se com erros pequenos e frequentemente dentro de um intervalo de limite de confiança, quando comparados com a execução de uma simulação sincronizada (Martini et. al., 1997; Thondugulam et al. 1999).

A estratégia para diminuir a sobrecarga devido à sincronização segue duas linhas: a redução da frequência com que a simulação necessita ser sincronizada; ou a eliminação da necessidade de sincronização ou manutenção da relação de causa e efeito.

Nem todas as aplicações porém, podem ser consideradas para a simulação com protocolo assíncrono. As aplicações que necessitam manutenção da relação de causa e efeito, como a simulação de lógica digital, não são candidatas a essa abordagem. Porém, alguns sistemas ou grupo de sistemas poderiam ser considerados e se beneficiariam das vantagens desta abordagem (Dota, 2001).

3.10. Considerações finais

A simulação destaca-se entre as técnicas para avaliação de desempenho principalmente devido ao baixo custo apresentado e à sua flexibilidade. Ao contrário do que acontece com as medições, em simulação não há necessidade de se alterar o sistema real, nem sequer exige a existência do mesmo. Pode ser aplicado tanto a sistemas simples como àqueles mais complexos, possibilitando variar os níveis de detalhes sem perder a precisão e também sem exigir as suposições e simplificações necessárias na modelagem analítica.

O ciclo de vida da simulação é dinâmico e permite que seu modelo seja adaptado a novos parâmetros, refletindo as modificações desejadas no sistema real. Uma desvantagem da simulação é que ela exige muito tempo para a obtenção de resultados confiáveis, especialmente ao simular sistemas muito complexos ou com um nível de detalhes muito alto, o que pode inviabilizá-la. A simulação distribuída mostra-se uma solução para diminuir esse tempo de execução e viabilizar tais simulações.

Um problema a ser atacado na simulação distribuída é a sincronização no processamento de eventos. Esta sincronização é necessária para que as relações de causa e efeito encontradas no sistema real sejam preservadas. Os protocolos para sincronização de simulação distribuída dividem-se em duas abordagens principais: *conservativos* e *otimistas*.

O problema dos protocolos conservativos consiste no bloqueio de processos. Isto pode acarretar em baixo desempenho, pois limita a execução de eventos visando garantir que erros de causa e efeito não sejam gerados, e também na ocorrência de *deadlocks*. O desempenho dos protocolos conservativos fica muito dependente do *lookahead*.

Os protocolos otimistas são atrativos pela possibilidade de se alcançar maior *speedup*, embora aplicações em que ocorram muitos *rollbacks* possam degradar o desempenho. A quantidade de memória necessária devido ao armazenamento freqüente de estados, assim como a necessidade de liberá-la também degradam seu desempenho.

Como ambos os protocolos possuem vantagens e desvantagens, a escolha do melhor protocolo depende muito do sistema que se deseja simular.

Visto que a principal sobrecarga nos protocolos conservativos e otimistas é a necessidade de sincronização, os protocolos assíncronos podem ser uma escolha a ser considerada. Nem todas as aplicações podem ser modeladas por esta abordagem.

Capítulo 4 – Linguagens e Ambientes

4.1. Considerações Iniciais

A utilização da simulação distribuída tem como principal objetivo o aumento na velocidade de execução que o uso de processadores em paralelo pode proporcionar. Outro atrativo da simulação distribuída é a possibilidade de se executar modelos de simulação maiores e mais complexos, já que a quantidade de memória disponível é potencialmente maior, quando comparada à versão com único processador.

Apesar do grande interesse da comunidade científica pela simulação distribuída, o emprego desta ainda não está difundido em aplicações do mundo real. Uma das razões para este fenômeno é o custo de máquinas paralelas, que ainda são usadas principalmente pela comunidade científica. A utilização de sistemas distribuídos tem tornado a computação paralela mais acessível. Além disso, outro problema é o esforço necessário para migrar simulações sequenciais a distribuídas, o que requer um grande esforço de programação e experiência, aliada à escassez de ferramentas que permitam seu uso por analistas pouco familiarizados com algoritmos de simulação paralela (Bagrodia, 1994; Low et al., 1999).

Para encorajar a adoção de simulação distribuída, o processo de migração de simulação sequencial a distribuída deve ser o mais fácil possível. O usuário deve ser capaz de concentrar seus esforços no processo de modelagem, sem se preocupar com assuntos relacionados a protocolos de sincronização da simulação paralela. Isto leva ao desenvolvimento de linguagens e bibliotecas (ou extensões) para implementação de simulação distribuída.

Os ambientes de simulação automáticos constituem uma possibilidade viável para ajudar a elaboração de simulações distribuídas. Compostos de editores gráficos,

interpretadores, compiladores e otimizadores, possibilitam que um programa de simulação seja gerado e executado automaticamente a partir de uma representação gráfica do modelo do sistema a ser avaliado (Bruschi, 2002).

Algumas das principais linguagens e bibliotecas para simulação distribuída serão descritas na seção 4.2, ambientes de simulação serão descritos na seção 4.3. A extensão ParSMPL e o ambiente ASDA, envolvidos diretamente no trabalho, serão descritos com mais detalhes nas seções 4.5 e 4.6, respectivamente. A extensão para simulação sequencial SMPL (MacDougall, 1987) será apresentada na seção 4.4, visando melhor compreensão do ParSMPL.

4.2. Linguagens e Bibliotecas para Simulação Distribuída

As principais abordagens para se projetar aplicações de simulação paralela são (Bagrodia, 1994):

- Abordagens baseadas em **bibliotecas**, como o YADDES (Preiss, 1989) e SPEEDES (Steinman, 1991). As operações de simulação e sincronismo são disponibilizadas através de chamadas a bibliotecas implementadas em linguagens sequenciais padrão como C ou C++. A maior vantagem dessa abordagem é que o usuário não precisa aprender uma nova linguagem. Entretanto, como não é utilizado nenhum tradutor, as rotinas de biblioteca em geral provêem menos funcionalidade do que seria possível numa linguagem.
- Aperfeiçoar **linguagens de simulação** sequenciais com primitivas para simulação paralela. Utiliza-se neste caso um tradutor, que permite à linguagem oferecer uma interface mais sucinta e “natural” para o programador, se comparado às bibliotecas de simulação. Como exemplo, pode-se citar o Maisie (Bagrodia & Liao, 1994), influenciado pela linguagem de simulação May (Bagrodia et al., 1987).
- Adicionar capacidades de simulação a **linguagens paralelas**, como o ModSim II (Bryan, 1989), derivada do Modula-2, e o SCE (Gill et al., 1989), baseada no Ada.
- Notações de alto nível baseadas em ambientes de programação lógicas, equacionais ou declarativas. Um exemplo disto é o ambiente baseado em UNITY e uma notação baseada em lógica temporal. Enquanto tais ambientes

oferecem o potencial para uma especificação de alto nível que pode subsequente ser executado com pouca ou nenhuma modificação utilizando uma variedade de protocolos de sincronização, existe pouca evidência experimental para confirmar a viabilidade desta abordagem para programação de simulações paralelas.

Uma importante característica de uma linguagem de simulação distribuída é a transparência, devendo ocultar aspectos do protocolo de sincronização utilizado. Ela geralmente provê ao usuário um conjunto completo de construtores de linguagem bem definidos, enquanto uma biblioteca oferece apenas um grupo de rotinas que devem ser utilizadas com uma linguagem de programação. Por outro lado, a biblioteca tem como vantagem maior flexibilidade no controle do comportamento do protocolo de simulação. Por exemplo, o usuário pode especificar o *lookahead* do protocolo conservativo, ou escolher um entre diferentes tipos de controle de *rollbacks* no protocolo otimista. Como as linguagens são projetadas para esconder tais questões, freqüentemente adotam um conjunto predefinido de opções, não permitindo ao usuário modificá-las (Low et al., 1999).

A seguir estão resumidas algumas linguagens e bibliotecas para simulação distribuída (Low et al., 1999; Spolon et al., 2000):

APOSTLE (Wonnacott & Bruce, 1995): linguagem orientada a objetos, baseada em protocolo otimista.

Maisie/Moose/Parsec: o Maisie (Bagrodia & Liao, 1994) é uma linguagem desenvolvida tanto para simulação sequencial quanto distribuída, e suporta vários protocolos, tanto otimistas quanto conservativos. Sua versão orientada a objetos é o Moose (Waldorf & Bagrodia, 1994). O Parsec (Bagrodia et al., 1998) é derivado do Maisie, com melhorias e construtores de linguagem reescritos. Estes três pacotes são desenvolvidos no UCLA (Universidade da Califórnia) e têm sido utilizados para simulação distribuída de propósito geral.

ModSim II (Bryan, 1989): linguagem orientada a objetos, derivada do Modula-2 e baseada em protocolo otimista. Utiliza estrutura de blocos, encapsulamento e herança múltipla.

NOTIME (Rao et. al., 1998; Rao & Wisley, 2002): sistema para simulação baseada em C++, utiliza protocolo assíncrono, possui núcleo e API (*Application Program Interface*) semelhantes ao do WARPED (Martin et. al., 1996).

YADES (Preiss, 1989): linguagem para especificação de simulação, provê suporte a vários protocolos de sincronização.

WARPED (Martin et. al., 1996, apud Rao & Wisley, 2002): sistema para simulação distribuída baseada em C++, utiliza o protocolo otimista *Time Warp*. É composto por um núcleo que provê a API para programação de processos lógicos.

Compose (Martin & Bagrodia, 1995): biblioteca baseada em C++; suporta a execução de modelos utilizando protocolo adaptativo, onde os objetos da simulação podem trocar seu modo de execução, de conservativo para otimista e vice-versa.

CPSim (Groselj, 1995): biblioteca comercial baseada em C, para protocolo conservativo. Tem sido distribuída comercialmente desde 1995 pela Boyan Tech Inc.

JUSTDistributed (Cassel & Pidd, 2001): pacote em Java, utiliza o protocolo conservativo CMB. Permite criar um modelo de simulação distribuída e executá-lo na Internet.

ParaSol (Edward et al., 1995): biblioteca baseada em C++, implementa protocolo otimista. Foi projetada para suportar transações: utiliza transações ao invés de mensagens com marcas de tempo para comunicação e sincronização entre processos lógicos.

ParSMPL (Ulson, 1999): biblioteca baseada em C, utiliza o mecanismo conservativo CMB para sincronização. É uma extensão da extensão funcional SMPL, e será visto com maiores detalhes na seção 4.5.

SPEEDES (Steinman, 1991): biblioteca baseada em C++, com suporte a protocolos otimistas e também a um protocolo híbrido denominado *Bounded Time Warp*, que adiciona características conservativas ao protocolo otimista *Time Warp*.

Além das linguagens e bibliotecas citadas, as linguagens para programação paralela CSP e Occam também têm sido utilizadas para simulação distribuída. Em seu trabalho, Theodoropoulos propõe um novo protocolo para sincronização (PDSP – *Program Driven Synchronization Protocol*), voltado à simulação de *hardware* assíncrono, implementado utilizando CSP/Occam e conceitos de *lookahead* e mensagens nulas para evitar *deadlock*, semelhante ao protocolo conservativo (Theodoropoulos, 2002).

4.3. Ambientes de Simulação Automáticos

O conceito de ambiente de simulação foi desenvolvido com o intuito de auxiliar o usuário no processo de transcrição do modelo especificado em um programa de simulação, tarefa quase sempre difícil e que pode se tornar um obstáculo na utilização da simulação. Para isso, tais ambientes oferecem uma série de recursos, como editores gráficos, interpretadores, compiladores e otimizadores. Se o ambiente também oferecer a possibilidade de gerar e executar o programa de simulação automaticamente, a partir de uma representação gráfica do usuário, esse ambiente é definido como um ambiente de simulação automático (Bruschi, 2002).

Atualmente existem vários ambientes de simulação, sendo que vários deles são comerciais. Segue-se uma breve descrição de alguns dos ambientes de simulação automáticos mais conhecidos (Bruschi, 2002):

ARENA (Swain, 1995; Banks, 1996): ambiente de simulação comercial de propósito geral, baseado na linguagem SIMAN V (SIMulation ANalysis), uma linguagem para modelar sistemas discretos, contínuos e híbridos. Este ambiente originou o uso de módulos pré-definidos de simulação, os *templates*, que possibilita adaptar a aplicação a projetos específicos. Outra característica do ARENA é a possibilidade de construir modelos animados.

ASiA (Santana, 1996; Santana, 1996a): ambiente acadêmico para simulação sequencial. Este ambiente visa afastar do usuário a tarefa de transcrição do modelo em programa de simulação: o programa de simulação é gerado automaticamente (Spolon, 1994), a partir da especificação gráfica do modelo do usuário (Spolon, 1994a). O ASiA foi desenvolvido para simular sistemas computacionais, arquiteturas e redes de computadores (Bruschi, 1997; Kawabata, 1999). O ambiente também fornece um modo de visualização gráfica dos resultados da simulação (Pedroso, 1996), bem como uma análise estatística desses resultados (Oliveira, 1998).

OMNeT++ (Varga, 2001): ambiente acadêmico para simulação discreta sequencial, desenvolvido para simular, principalmente, redes de computadores, multiprocessadores e outros sistemas distribuídos, podendo ser utilizado para simular outros tipos de sistemas. Foi criado com o objetivo de oferecer um sistema gratuito de simulação discreta, como uma alternativa aos caros produtos comerciais da área. O OMNeT também suporta animação e execução interativa (Varga, 2003).

UCLA Simulation Environment (Bagrodia et al., 1998): ambiente dirigido a simulação distribuída. Oferece fácil migração de modelos de simulação entre sistemas

operacionais, e possibilita sua utilização tanto em plataformas de memória compartilhada quanto distribuída. Suporta vários protocolos de sincronização e também modelagem visual ou hierárquica. Utiliza em seus modelos a linguagem de simulação Parsec (Bagrodia et al., 1998) ou a biblioteca Compose (Martin & Bagrodia, 1995)

ASDA (Bruschi et al., 2000; Bruschi, 2002): ambiente de simulação voltado à avaliação de desempenho de sistemas computacionais. Este ambiente é visto com detalhes na seção 4.6.

4.4. SMPL

O SMPL é uma extensão funcional da linguagem C para simulação orientada a eventos, para plataformas compatíveis com o IBM-PC (MacDougall, 1987). Um modelo de simulação em SMPL é implementado como um programa na linguagem hospedeira e as operações de simulação são realizadas através de chamadas às funções do subsistema de simulação. A representação de modelos em SMPL é feita utilizando-se três entidades básicas:

- **Facilidades:** cada facilidade representa um recurso do sistema sendo modelado (como uma CPU em um modelo de um sistema computacional). Um sistema é composto de uma coleção de facilidades interconectadas. As primitivas do SMPL permitem definir, reservar e liberar facilidades, além de verificar seu estado. A interconexão das facilidades é determinada através do roteamento dos *tokens* entre as mesmas.
- **Tokens:** representam as entidades ativas do sistema (como uma tarefa em um modelo de um sistema computacional). Existem dois tipos básicos de operações para o controle do fluxo dos *tokens*: a *reserva* de facilidades e o *escalonamento* de eventos. Se a facilidade reservada pelo *token* não tiver servidores disponíveis, este *token* é posto numa fila até que algum servidor fique disponível e possa atendê-lo. O único atributo que o SMPL disponibiliza aos *tokens* é a prioridade, o que permite a preempção das facilidades.
- **Eventos:** correspondem às mudanças de estado de qualquer entidade do sistema, ativa ou passiva. Por exemplo, quando uma tarefa encerra sua execução e libera a facilidade CPU, ocorrem dois eventos: o final de sua execução e a liberação da CPU. Um evento é diferenciado pelo seu número, tempo de ocorrência e identificação do *token* a ele relacionado.

Os programas desenvolvidos em SMPL são caracterizados pela existência de um laço principal que comanda as iterações da simulação. Dentro deste laço os eventos são

processados e os *tokens* são escalonados para novos eventos, também são realizadas as operações para a coleta de informações estatísticas.

O SMPL fornece rotinas para selecionar e submeter eventos que irão ocorrer, além de ferramentas para depuração, coleta de dados, análises estatísticas e uma interface interativa para a execução de programas de simulação, denominada *mtr* (MacDougall, 1987).

4.4.1. Principais Rotinas do SMPL

O SMPL fornece ao usuário um conjunto de rotinas para auxiliar o desenvolvimento de programas de simulação orientada a eventos. Além das rotinas principais, que estão descritas a seguir, esta extensão também possui diversas outras, para manipulação de filas de eventos, dos recursos e da lista de eventos futuros, além de primitivas para análises estatísticas.

As principais rotinas do SMPL são:

simpl(): inicia o sistema, atribuindo os valores iniciais às estruturas de dados e ao relógio da simulação, para que a simulação seja então iniciada;

facility(): cria, nomeia e retorna um descritor para cada facilidade do modelo. Este descritor é utilizado para qualquer operação a ser realizada na facilidade, e cada facilidade possui um número *n* de servidores e uma fila de acesso;

request(): requisita o atendimento de um dos servidores de uma facilidade. Se um servidor estiver livre, este é reservado ao *token* solicitante. Caso contrário, a função retorna um valor indicando que a facilidade está ocupada e a requisição é colocada na fila desta facilidade;

release(): libera um servidor de uma determinada facilidade, atualiza as estatísticas e decrementa o número de servidores ocupados da facilidade. Se houver evento pendente na fila da facilidade, insere este evento na lista de eventos futuros para execução imediata;

schedule(): escalona os eventos da simulação. Todo evento a ser escalonado é inserido na lista de eventos futuros, em ordem crescente do tempo de ocorrência dos eventos;

cause(): remove o primeiro elemento da lista de eventos futuros e avança o tempo da simulação (relógio) para o tempo de ocorrência do evento a ser executado.

4.5. A Extensão ParSMPL

O ParSMPL (Ulson, 1999) é uma ferramenta de simulação distribuída que utiliza o protocolo conservativo CMB para a sincronização entre os processos lógicos. Esta ferramenta baseia-se no SMPL (MacDougall, 1987), o que faz do ParSMPL uma extensão funcional da linguagem de programação C para simulação distribuída. O ParSMPL foi desenvolvido para plataformas IBM/RS6000 e IBM SP2 utilizando sistema operacional AIX, realizando comunicação e sincronização através dos ambientes de passagem de mensagem PVM e PVMe.

Esta extensão oferece os recursos necessários para a elaboração de programas de simulação distribuída, facilitando o trabalho tanto de usuários com pouca experiência em programação paralela, quanto daqueles mais experientes. Utilizando esta extensão, o usuário não precisa se preocupar com os aspectos de sincronização e comunicação entre processos, nem com características do protocolo de sincronização utilizado para simulação distribuída. Esses mecanismos estão inseridos no ParSMPL e são transparentes ao usuário.

A hierarquia de *software* que compõe o ambiente ParSMPL pode ser vista na figura 4.1. A aplicação do usuário é desenvolvida em linguagem de alto nível e pode interagir com as primitivas da extensão para simulação, com o ambiente de passagem de mensagem e também com o sistema operacional, o que possibilita maior flexibilidade aos usuários mais experientes (Ulson, 1999).

Figura 4.1: Hierarquia de *software* no ParSMPL

4.5.1. Estrutura do ParSMPL

O SMPL, como foi visto na seção 4.4, foi implementado em plataformas compatíveis com o IBM-PC (MacDougall, 1987). A partir do SMPL foi desenvolvida uma nova versão para plataformas Unix, o SMPLX (Ulson, 1999), com o intuito de adequá-lo às bibliotecas de passagem de mensagem PVM e PVMe, utilizadas para a comunicação entre os processos. A principal diferença entre o SMPL e o SMPLX é que o SMPL utiliza estruturas de dados estáticas, enquanto o SMPLX, assim como o ParSMPL, utilizam estruturas dinâmicas. A razão desta alteração foi a limitação imposta à complexidade dos modelos implementados, já que as dimensões das estruturas deveriam ser definidas em tempo de compilação. A partir de modificações no SMPLX foi então desenvolvido o ParSMPL.

No ParSMPL, os processos da simulação devem ser criados no início da execução da simulação e destruídos ao final, de modo que não haja criação dinâmica de processos, o que é uma característica do protocolo CMB. Contudo, deve-se ressaltar que fica a cargo do usuário garantir que novos processos não sejam criados ao longo da execução da simulação.

A extensão ParSMPL é constituída por dois módulos básicos (Ulson, 1999):

- *núcleo*: contém as primitivas para comunicação, sincronização e prevenção de *deadlock*;
- *biblioteca*: contém as primitivas necessárias para o desenvolvimento de um programa de simulação, além de primitivas para coleta e análise estatística dos dados obtidos.

Uma simulação distribuída é modelada como um conjunto de processos lógicos, cada um representando um processo físico do sistema real. As interações entre os processos físicos são representadas pela troca de mensagens entre os processos lógicos correspondentes. Cada processo possui um certo número de canais lógicos bidirecionais, através dos quais as mensagens são recebidas e enviadas, um relógio local, que controla o seu tempo de simulação, e uma lista de eventos, na qual ficam armazenadas em ordem cronológica as mensagens recebidas de outros processos.

4.5.2. Principais rotinas do ParSMPL

As principais rotinas que compõe o ParSMPL incluem as rotinas do SMPL, vistas na seção 4.4.1, e as novas rotinas criadas e incluídas no ParSMPL, descritas a seguir. As rotinas

do SMPL foram mantidas no ParSMPL, sendo que algumas delas foram alteradas para se adaptarem à simulação distribuída. Um grupo de novas rotinas foi incorporado para viabilizar a execução da simulação distribuída. Estas novas rotinas são (Ulson, 1999):

ReceiveMessage(): é chamada pela rotina *cause()* para verificar se o processo tem alguma mensagem (tanto completa quanto nula) para ser recebida:

SendNull(): envia mensagens nulas para os demais processos com os quais se comunica sempre que o processo lógico em questão for bloqueado, seja porque o tempo de ocorrência do elemento do início da lista de eventos futuros é maior que o relógio do processo, seja porque a lista de eventos se encontra vazia;

SendComplete(): envia uma mensagem completa para o processo destino contendo informações sobre o evento a ser escalonado, o tempo de ocorrência desse evento, a identificação do *token* e as identificações dos processos de origem e de destino;

InsertFEL(): insere um evento na lista de eventos futuros do processo;

InsertChannelBuf(): insere na estrutura de *buffer* de canais o tempo de ocorrência da mensagem recebida.

A hierarquia das principais rotinas no ParSMPL é apresentada na figura 4.2. No ParSMPL, o usuário interage apenas com as primitivas do primeiro nível da hierarquia.

Figura 4.2: Hierarquia de rotinas no ParSMPL

4.6. ASDA

O ASDA (Ambiente de Simulação Distribuída Automático) (Bruschi et al., 2000; Bruschi, 2002) é um ambiente de simulação voltado à avaliação de desempenho de sistemas computacionais. Seu objetivo é oferecer um ambiente automático para facilitar o desenvolvimento de simulações, a usuários com vários níveis de conhecimento, tanto de simulação quanto de computação paralela.

O ASDA é composto por sete módulos, como pode ser visto na figura 4.3. Cada módulo possui funções bem definidas, descritas a seguir (Bruschi, 2002).

Módulo de Interface com o usuário: oferece ao usuário um editor gráfico, no qual o usuário define o modelo a ser simulado. Esta interface também possibilita fácil acesso às opções disponibilizadas pelo ambiente, ao mesmo tempo em que proporciona aos usuários mais experientes flexibilidade para tomar suas próprias decisões.

Figura 4.3: Diagrama Modular do ASDA

Módulo Avaliador: tendo como base as variáveis do ambiente e o modelo a ser simulado, orienta o usuário na escolha da melhor abordagem para sua simulação.

Módulo Gerador: gera o código do programa de simulação, baseando-se na especificação gráfica e na decisão do módulo avaliador.

Módulo Executor: controla a execução dos programas de simulação.

Módulo de Interface de Software: fornece uma interface entre os programas de simulação já desenvolvidos e o módulo replicador.

Módulo Replicador: gerencia as replicações da simulação e garante a confiabilidade dos resultados da abordagem MRIP.

Módulo Escalonamento: responsável pelo escalonamento dos processos gerados pelo módulo executor.

Como os módulos envolvidos neste projeto são o Gerador e o Avaliador, estes são vistos com mais detalhes nas próximas seções.

4.6.1 Módulo Avaliador

A finalidade do módulo Avaliador é auxiliar o usuário, verificando as informações fornecidas e auxiliando nas decisões a serem tomadas no sistema. Este módulo divide-se em três níveis (Bruschi, 2002):

Nível 1: atua em conjunto com o módulo de Especificação Gráfica, e tem como objetivo verificar se o modelo do usuário está sendo especificado corretamente. Por exemplo, ele deve verificar a quantidade de nós de entrada e saída do modelo, se todos os centros de serviço estão conectados, etc., para que o modelo possa ser codificado sem erros.

Nível 2: auxilia o usuário na escolha do tipo de simulação: sequencial, distribuída SRIP ou distribuída MRIP. Caso o usuário solicite, este nível efetua uma avaliação do modelo e da arquitetura onde será executada a simulação e sugere o tipo de abordagem, informação que é passada ao módulo Gerador.

Nível 3: assim como o nível 2, este nível também auxilia o módulo Gerador. Este nível está dividido em dois sub-níveis. Se a decisão do nível 2 for pela simulação distribuída SRIP, o primeiro sub-nível auxilia na escolha do protocolo de sincronização. O segundo auxilia na divisão do modelo em processos lógicos. Este último sub-nível divide-se em três

partes, cada uma encarregada de tratar a divisão dos processos lógicos nos protocolos CMB, *Time Warp* e CMB $\rightarrow$ *Time Warp*.

4.6.2. Módulo Gerador

O módulo Gerador é responsável por gerar o programa de simulação, baseando-se na especificação do modelo e na parametrização do sistema fornecidas pelo usuário.

Este módulo divide-se em dois sub-módulos:

- Geração de código de programas MRIP;
- Geração de código de programas SRIP;

No ASDA, a decisão de qual abordagem utilizar pode vir tanto do usuário como também ser uma tarefa realizada pelo próprio ambiente, através do módulo Gerador, com o auxílio do nível 2 do módulo avaliador, descrito na subseção anterior.

O ambiente permitirá a geração de código utilizando linguagens ou bibliotecas pré-determinadas. Para cada linguagem/biblioteca deverá haver um arquivo de gabarito e um programa gerador. O gabarito corresponde a um arquivo com trechos de código na linguagem ou biblioteca alvo, combinados com comandos que indicam onde as informações necessárias são encontradas. O gerador unirá os dados da especificação, fornecidos pelo módulo de Interface Gráfica, com as informações do arquivo de gabarito para gerar o programa de simulação.

A geração de código para abordagem MRIP é a geração de um programa de simulação sequencial, acrescentando-se uma biblioteca com funções de comunicação para o envio de dados ao mestre, e outra biblioteca com funções para tratamento local das observações coletadas antes de enviar para o mestre (Bruschi, 2002).

A abordagem SRIP poderá ser feita utilizando protocolos conservativo, otimista ou troca de protocolo em tempo de execução. O usuário poderá escolher livremente um entre os três protocolos ou então contar com o auxílio do módulo Avaliador (neste caso, do nível 3). Além de auxiliar na escolha de protocolos, esse nível do modo avaliador também é responsável em auxiliar na divisão do modelo em processos lógicos.

Independente do protocolo escolhido, a geração do código SRIP é idêntica à geração de código MRIP. A diferença é que na abordagem MRIP apenas um programa é gerado, enquanto na SRIP é gerado um programa diferente para cada processo lógico.

Depois de gerado o programa, o módulo Gerador atualiza o arquivo do usuário, que contém informação referente aos seus modelos, e o(s) código(s)-fonte da simulação são compilados e enviados ao Módulo de Execução (Bruschi, 2002).

4.7. Considerações Finais

A tarefa de escrever simulações distribuídas, ou até mesmo migrar simulações sequenciais existentes ao paradigma de simulação distribuída é uma tarefa árdua, pois exige do modelador conhecimentos tanto sobre simulação quanto de programação paralela e protocolos de sincronismo para simulação distribuída. Esses obstáculos têm impedido a aceitação da simulação distribuída em aplicações do mundo real, fora do âmbito acadêmico.

O desenvolvimento de linguagens e bibliotecas (ou extensões funcionais) para simulação distribuída vem tentando reduzir esses obstáculos, tirando do usuário a necessidade de se preocupar com aspectos de sincronização e até mesmo de computação paralela, escondendo do usuário características próprias do protocolo de sincronização e da comunicação entre processos. A utilização de bibliotecas ou extensões para simulação distribuída apresenta a grande vantagem de não ser preciso aprender uma nova linguagem.

A tarefa de elaborar simulações distribuídas pode ser facilitada ainda mais com a utilização de ambientes de simulação automático.

Foi visto neste capítulo um resumo de linguagens e bibliotecas para simulação distribuída. A maioria dessas linguagens ou bibliotecas existentes é baseada em protocolos otimistas, poucas apresentam soluções com abordagem conservativa. Também foram descritos alguns dos principais ambientes de simulação, tanto para simulação sequencial como distribuída.

A extensão ParSMPL e o ambiente automático ASDA foram apresentados com mais detalhes, visto que são as ferramentas a serem utilizadas neste projeto.

Capítulo 5 – Adaptação, Aprimoramento e Validação do ParSMPL

5.1. Considerações Iniciais

O ambiente para simulação distribuída ASDA (Bruschi, 2001) teve todo seu desenvolvimento voltado para plataformas Linux. A biblioteca ParSMPL (Ulson, 1999; Ulson, 1999a), como foi visto no capítulo anterior, foi desenvolvida e utilizada em plataformas IBM/RS6000 e IBM SP2 com sistema operacional AIX. Tentativas anteriores de se utilizar o ParSMPL em plataformas Linux mostraram a existência de certas incompatibilidades que impossibilitavam sua utilização imediata. Apesar do sistema Linux possuir semelhanças com o sistema AIX, já que ambos são baseados em arquitetura Unix, as versões do PVM em cada sistema apresentam diferenças, o que foi um primeiro obstáculo na utilização do ParSMPL. Além disso, a versão dos compiladores em cada sistema também apresenta diferenças, já que o ParSMPL passou a apresentar problemas de compilação que não ocorriam no AIX.

Assim, um primeiro passo para possibilitar a utilização do ParSMPL no ambiente de simulação ASDA foi a adaptação do ParSMPL ao Linux.

Durante a fase de adaptação, com um melhor entendimento do código do ParSMPL e de suas características, percebeu-se que poderiam ser feitas melhorias na biblioteca, visando melhorar seu desempenho e também facilitar sua depuração. Uma dessas melhorias foi a introdução de *lookahead* dinâmico.

Como as modificações aplicadas à biblioteca ParSMPL incluíram aspectos da implementação do protocolo CMB (Chandy & Misra, 1979; Bryant, 1977), foi necessário realizar uma validação dessa nova versão do ParSMPL para Linux. Para diferenciar o ParSMPL original, projetado e desenvolvido no AIX, propõe-se chamar esta nova versão para Linux de *ParSMPLX*.

Os problemas iniciais encontrados na adaptação do ParSMPL ao Linux estão descritos na seção 5.2. As modificações para aprimoramento do ParSMPL serão discutidos na seção 5.3. O processo de validação do ParSMPLX está descrito na seção 5.4. A seção 5.5 traz as considerações finais desta fase do projeto.

5.2. Adaptação do ParSMPL

Os primeiros obstáculos encontrados na utilização da versão original do ParSMPL em sistemas Linux foram de compilação. No AIX, plataforma para o qual a biblioteca foi desenvolvida, a compilação de programas de simulação utilizando o ParSMPL era feita de modo direto em linha de comando, sem a necessidade de se gerar primeiro o código-objeto do ParSMPL para depois efetuar a link-edição com o programa de simulação. No Linux, entretanto, é necessário compilar utilizando *makefiles*. Isto é uma exigência da implementação em Linux do ambiente de passagem de mensagens PVM.

Assim, foi preciso criar um *makefile* para gerar o código-objeto da biblioteca ParSMPL, e um *makefile.aimk*, que efetua a compilação dos programas de simulação utilizando o código-objeto da biblioteca ParSMPLX e da biblioteca do PVM. O primeiro *makefile* é necessário apenas quando há modificações no código da biblioteca ParSMPLX. Portanto, uma vez que o ParSMPLX está estabilizado e não se deseja efetuar modificações na biblioteca, mas apenas criar programas de simulação, somente *makefile.aimk* é necessário.

Outro problema encontrado ao compilar os programas de simulação foi a notação. O compilador do Linux não aceitava a notação utilizada no AIX. Todas as funções, declaradas originalmente como no exemplo da figura 5.1 (a), tiveram que ser modificadas, como pode ser visto na figura 5.1 (b). Além disso, foi preciso incluir no arquivo *parsmpl.h* o protótipo das funções presentes no arquivo *parsmpl.c*. Diferente da utilização no AIX, ao utilizar o ParSMPLX deve-se incluir no código-fonte do programa de simulação o arquivo de cabeçalho *parsmpl.h*, ao invés do código fonte *parsmpl.c*, como descrito em (Ulson, 1999).

Algumas das variáveis tinham seu valor inicial definido no arquivo de cabeçalho *declaracoes.h*, o que não é permitido pelo compilador gcc (versão egcs-2.91.66). Ao invés de terem seus valores associados no momento da declaração, essas variáveis passaram a ser iniciadas na função *simpl()*, já que é a primeira função da biblioteca a ser chamada em um programa de simulação.

```

schedule (num_evento, te, token, ident_proc)
int num_evento;
double te;
int token,
    ident_proc;
{
    (a)

void schedule (int num_evento, double te, int token, int dent_proc)
{
    (b)

```

Figura 5. 1: (a) Declaração original da rotina schedule (), considerada obsoleta.

(b) Declaração atual da mesma função.

Como alguns arquivos de cabeçalho eram chamados em mais de um programa fonte, também havia problemas de variáveis duplicadas. Isso foi corrigido com a inclusão de diretivas *#define* nos arquivos de cabeçalho, evitando assim que o mesmo arquivo de cabeçalho fosse incluído mais de uma vez.

5.3. Aprimoramento do ParSMPL

Nesta seção serão discutidas as alterações efetuadas visando aprimoramentos do ParSMPL, que resultaram na versão ParSMPLX. Essas alterações serão subdivididas em duas partes:

1. **Correções:** Após aplicar as adaptações necessárias para que o ParSMPLX pudesse ser utilizado no Linux, foi necessário corrigir algumas características de implementação da biblioteca, que são apresentadas na seção 5.3.1.
2. **Introdução de novas características:** Novas características foram incluídas ao ParSMPLX, visando facilitar sua utilização e melhorar o desempenho obtido ao utilizar o ParSMPL em programas de simulação distribuída. Estas alterações são apresentadas na seção 5.3.2.

5.3.1. Correções

Apenas adaptar o ParSMPLX para ser executado no Linux não foi suficiente para que ele pudesse ser utilizado para construir programas de simulação. Foi necessário também efetuar correções na implementação de algumas rotinas, até mesmo modificações na implementação do protocolo CMB. As correções aplicadas em cada rotina estão descritas a seguir.

I. Rotinas para alocação de memória

As rotinas para alocação de memória são utilizadas para alocar memória dinamicamente para as estruturas de dados definidas na biblioteca ParSMPL. Retornam ou um ponteiro para a memória alocada, ou um ponteiro nulo caso não seja possível alocar memória. Um exemplo de rotina para alocação de memória pode ser visto na figura 5.2.

No ParSMPL original havia seis rotinas para alocação dinâmica de memória: uma para cada tipo de dado estruturado criado para a biblioteca. Foram detectados problemas de memória com a biblioteca ParSMPLX, os quais puderam ser visivelmente percebidos pela mudança repentina no valor de parâmetros entre chamadas a funções. Assim, foi preciso cercar todas as possibilidades de erros de memória, entre eles a alocação dinâmica de memória feita por meio de rotinas específicas.

Como pode se observar no exemplo da figura 5.2, uma variável local é criada e utilizada para apontar para a memória alocada, e então o endereço desta memória é retornado. O problema na utilização de variável local para alocação de memória dinamicamente é que esta variável é destruída ao final do bloco ao qual pertence. Assim, a função retorna um endereço de uma memória que foi liberada e pode, posteriormente, vir a ser utilizada.

Como o uso dessas rotinas poderia não ser eficiente, optou-se por fazer a alocação direta no código. Assim, nos pontos onde havia chamada a alguma rotina para alocação de memória, essa chamada foi eliminada e em seu lugar passou a ser feita uma chamada diretamente a função *malloc*, da biblioteca *stdlib.h*.

```
/* ----- aloca_mem_cabecalho -----  
Entrada:  
Funcao: Aloca memoria para a estrutura do cabecalho.  
Saida: Ponteiro para a memoria alocada ou null em caso de erro  
----- */  
tipo_cabecalho *aloca_mem_cabecalho()  
{  
    tipo_cabecalho *ponteiro;  
    ponteiro = malloc (sizeof(tipo_cabecalho));  
    return (ponteiro);  
}
```

Figura 5. 2: Exemplo de rotina para alocação de memória

II. Rotina `parsmpl_pvm_exit()`

Esta rotina tem como função liberar a memória alocada para a lista de eventos futuros e para a lista de cabeçalhos descritores dos recursos. A primeira lista corresponde a uma lista encadeada simples, enquanto a segunda é uma lista de listas.

Pelo código original, visto na figura 5.3, apenas a cabeça de cada lista era liberada, o restante da lista permanecia na memória, porém inacessível, já que o ponteiro de acesso a ela foi liberado. Esta falha na liberação de memória poderia influir nos erros de memória encontrados.

```
/* ----- parsmpl_pvm_exit ----- */  
  
void parsmpl_pvm_exit()  
{  
    /* libera ponteiros */  
    free(lista_eventos_futuros);  
    free(cabecalho_descritor);  
  
    pvm_exit(); /* toda task que nao foi criada por pvm_spawn deve  
    executar */  
    exit(0);  
}
```

Figura 5. 3: Código original da rotina `parsmpl_pvm_exit()`

Para corrigir, foi feito um laço de destruição para cada lista, e o novo código da função `pvm_parsmpl_exit()` pode ser visto na figura 5.4.

A linha `exit(0)` foi retirada da função `parsmpl_pvm_exit()`. O parâmetro zero passado à função `exit()` indica ao sistema operacional que o programa terminou normalmente. Em caso de término anormal do programa (devido a algum erro durante a execução), um valor diferente de zero deve ser utilizado. Como a função `parsmpl_pvm_exit()` é chamada para terminar o programa tanto em caso de erro quanto término normal da execução, optou-se por incluir a linha `exit(1)` ao final da rotina `erro()` e `exit(0)` ao fim do programa de simulação. Isto é útil, por exemplo, quando se utiliza um *script* para executar várias instâncias da simulação e é preciso verificar a ocorrência de erros em alguma delas.

III. Rotina `insere_lista_eventos_futuros()`

A rotina `insere_lista_eventos_futuros()` é chamada sempre que um novo evento é escalonado e deve ser inserido na lista de eventos futuros do processo. Este novo evento pode ter sido gerado localmente, como também pode ser um evento criado remotamente e escalonado para execução no processo lógico local, recebido de um dos canais de entrada.

Ao ser chamada, a rotina *insere_lista_eventos_futuros()* verifica se o novo evento a ser inserido na lista de eventos futuros foi gerado localmente ou se sua origem é outro processo lógico. Se esse novo evento foi gerado localmente, a variável *te* indica o *tempo entre eventos*, e deve ser somado ao valor marcado pelo relógio da simulação para se obter seu *tempo de ocorrência*. Caso esse novo evento tenha sido gerado remotamente, a variável *te* já indica o *tempo de ocorrência* do evento.

```

/* -----
 * parsmpl_pvm_exit
 * Funcao: Liberar a lista de eventos futuros e a lista de
 *         cabecalhos descritores
 * ----- */
void parsmpl_pvm_exit (void)
{
    tipo_lista_eventos *aux_lista_eventos;
    tipo_cabecalho *aux_cabecalho;
    tipo_servidor_recursos *serv_rec,
                          *aux_serv_rec;
    tipo_fila_eventos *fila_evt,
                      *aux_fila_evt;

    /* percorre apagando os nos da lista de eventos futuros */
    while (lista_eventos_futuros)
    {
        aux_lista_eventos = lista_eventos_futuros;
        lista_eventos_futuros = lista_eventos_futuros-
>prox_lista_eventos;
        free (aux_lista_eventos);
    }
    /* percorre apagando cabecalho_descritor */
    while (cabecalho_descritor)
    {
        aux_cabecalho = cabecalho_descritor;
        cabecalho_descritor = cabecalho_descritor->prox_cabecalho;
        serv_rec = aux_cabecalho->pont_descritor->fila_servidor_recursos;
        fila_evt = aux_cabecalho->pont_descritor->lista_eventos_recurso;
        /* libera a lista de servidores do recurso */
        while (serv_rec)
        {
            aux_serv_rec = serv_rec;
            serv_rec = serv_rec->prox_servidor_recursos;
            free (aux_serv_rec);
        }
        /* libera a fila de eventos associada ao recurso */
        while (fila_evt)
        {
            aux_fila_evt = fila_evt;
            fila_evt = fila_evt->prox_fila_eventos;
            free (aux_fila_evt);
        }
        free (aux_cabecalho->pont_descritor);
        free (aux_cabecalho);
    }
    pvm_exit(); /*toda task que nao foi criada por pvm_spawn deve executar
 */
}

```

Figura 5. 4: Novo código da rotina *parsmpl_pvm_exit()*

No código original do ParSMPL, que pode ser observado na figura 5.5(a), ao receber um evento vindo de processo remoto, o tempo de ocorrência do evento era associado ao valor do relógio da simulação do processo lógico receptor no momento em que a mensagem fora recebida. A informação contida na variável *te* (o tempo de ocorrência do evento, que foi definido no processo lógico remoto que o gerou) não era utilizada. Desse modo, todo evento que chegava de um processo remoto tinha seu tempo de ocorrência fixado para execução imediata. Isto foi corrigido atribuindo-se o valor de *te* ao tempo de ocorrência do novo evento (figura 5.5(a)), ao invés do valor de *clock_simulacao*, como consta originalmente.

```
void insere_lista_eventos_futuros (int num_evento, double te, int
                                token, int ident_proc)
{
    tipo_lista_eventos *novo_evento,
        *ponteiro; /* percorre a lista de eventos futuros */

    novo_evento = (tipo_lista_eventos *) malloc (sizeof
(tipo_lista_eventos));
    if (novo_evento == NULL)
        erro ("Rotina: insere_lista_eventos_futuros() - Erro:
nao ha memoria para a lista de eventos futuros");

    /* insere os dados no noh alocado */
    if (ident_proc == proc_local) /* evento gerado localmente */
        novo_evento->tempo_ocorrencia = te + clock_simulacao;
    else /* evento veio de processo remoto */
        novo_evento->tempo_ocorrencia = clock_simulacao;

    (a)

    /* insere os dados no noh alocado */
    if (ident_proc == proc_local) /* evento gerado localmente */
        novo_evento->tempo_ocorrencia = te + clock_simulacao;
    else /* evento veio de processo remoto */
        novo_evento->tempo_ocorrencia = te;

    (b)
}
```

Figura 5. 5: (a) Trecho original da rotina *insere_lista_eventos_futuros()*.

(b) Trecho corrigido da rotina

IV. Rotina *atualiza_buffer_marcas_tempo()*

Para cada canal de comunicação é mantida uma lista contendo as marcas de tempo das mensagens recebidas. Sempre que um evento proveniente de outro processo lógico é processado, a função *atualiza_buffer_marcas_tempo()* é chamada para remover a marca correspondente à mensagem (a cabeça da lista) do canal de comunicação.

No código original, ocorria acesso a ponteiro nulo sem verificação, o que resultava em falha de segmentação (*segmentation fault*). Isso foi corrigido com a inserção de um teste antes de tentar utilizar o ponteiro.

V. Rotina *envia_mensagem_nula()*

A rotina *envia_mensagem_nula()* é utilizada para enviar mensagens nulas aos outros processos com o qual se comunica, sempre que o processo lógico fica bloqueado.

Nesta função foi feita uma correção no envio de mensagem nula quando o tempo da mensagem nula era igual a zero. Ocorria erro porque as mensagens nulas são diferenciadas das mensagens completas por apresentar tempo negativo (antes que a mensagem seja enviada, o tempo é multiplicado por -1). Ao receber uma mensagem nula com tempo igual a zero, a mensagem era processada no destino como mensagem completa, resultando em erro ao desempacotar a mensagem. Para corrigir isto, foi incluído um teste no início da função, fazendo com que não sejam enviadas mensagens nulas com tempo igual a zero.

Outro problema detectado nesta função foi o envio de mensagens nulas para processos lógicos com o qual não deveria haver comunicação. Este é um problema que poderia levar à ocorrência de *deadlock* na simulação. Foi corrigido inserindo-se uma comparação com o relógio do canal do processo de destino antes de enviar a mensagem. Se houver comunicação entre o processo local e o processo de destino, o relógio desse canal de comunicação não deverá ser menor do que zero, pois no início da simulação todos os canais de comunicação não utilizados são marcados com (-1) . Se o relógio do canal sendo comparado não é menor do que zero, a mensagem é enviada.

VI. Rotina *cause()*

A rotina *cause()* retira eventos da lista de eventos futuros para serem executados. Se for preciso bloquear o processo, permanece em um laço para recepção de mensagens e envio de mensagens nulas.

É nesta rotina que se verifica se o processo pode prosseguir sua execução ou se deve ficar bloqueado, esperando por mensagens de outro(s) processo(s) lógico(s). Caso tenha que ficar bloqueado, calcula-se o tempo da mensagem nula considerando o *lookahead* e então uma mensagem nula é enviada para cada processo com o qual se comunica.

O problema nesta rotina estava justamente no cálculo do *lookahead* para o envio das mensagens nulas. O trecho de código que mostra como é feito o cálculo do *lookahead* e envio da mensagem nula no ParSMPL original é mostrado na figura 5.6, linhas 12 a 15. As palavras em destaque correspondem ao cálculo do *lookahead*. A variável *tempo_processo*, utilizada para o cálculo do *lookahead*, deve ter seu valor definido pelo usuário no código do programa de simulação.

Conceitualmente, a maneira mais simples de se calcular o *lookahead* é considerar o tempo mínimo necessário para se processar um evento (Fujimoto, 1998; Vee & Hsu, 1999). Na implementação original do ParSMPL foi feita uma tentativa de se melhorar o cálculo do *lookahead*, baseada no tempo de utilização do recurso modelado no processo lógico, em função da distribuição exponencial. O problema em se utilizar função de distribuição exponencial para calcular o *lookahead* é que, como valores arbitrários eram gerados, uma mensagem nula poderia ser gerada com tempo menor do que a última mensagem nula enviada, violando a restrição de causa e efeito de que apenas mensagens em ordem estritamente não decrescente poderiam ser enviadas através de cada canal. Esse problema provocava erros de causa e efeito e *deadlock* em alguns casos. Para solucioná-lo, a função exponencial foi retirada, deixando apenas o valor da variável *tempo_processo*. Esta variável deve ser definida cuidadosamente no código do usuário como o menor tempo necessário para processar um evento. A redefinição do *lookahead* no ParSMPLX possibilitou experimentos utilizando *lookahead* estático e *lookahead* dinâmico, visto no capítulo 6.

```

1  while ((tempo_cabeca_lista_eventos =
2      tempo_cabeca_lista_eventos_futuros()) >
3      (menor_clock = menor_clock_canais()))
4      ;
5      recebe_mensagem();
6      /* chegou mensagem, eh necessario atualizar o
7      * clock do processo */
8      if (menor_clock != clock_proc)
9      {
10         clock_proc = menor_clock;
11         if (lista_eventos_futuros != NULL)
12             clock_atual = lista_eventos_futuros->tempo_ocorrendia
13 + expntl(tempo_processo);
14         else
15             clock_atual = clock_proc + expntl(tempo_processo);
16
17         /* processo bloqueado, envia mensagem nula */
18         envia_mensagem_nula(clock_atual);
19     }
20 ;

```

Figura 5. 6: Trecho do código original da rotina *cause()*

Após esta correção, ainda se percebiam ocorrências de erros de causa e efeito. Isso ocorria porque, caso houvesse eventos na lista de eventos futuros esperando para serem processados, o tempo do primeiro evento dessa lista era utilizado como base no cálculo da mensagem nula, como pode ser visto na figura 5.6, linha 11. Esse cálculo não pode ser feito desse modo, já que o processo está bloqueado por haver um canal vazio cujo tempo é menor do que os tempos dos eventos na lista de eventos futuros, e portanto pode receber uma mensagem com marca de tempo menor do que esses eventos. Assim, para se calcular corretamente o tempo da mensagem nula, deve ser considerado somente o menor tempo entre

os canais de entrada do processo lógico, como mostra a figura 5.7, linha 10. Além disso, deve ser feita uma verificação se o tempo da mensagem nula não excede o tempo do primeiro evento na lista de eventos futuros. Isto é necessário porque esse evento na cabeça da lista de eventos futuros *pode* corresponder à liberação de um recurso e conseqüente escalonamento imediato de um evento em um processo remoto. Deste modo, o tempo da próxima mensagem completa a ser enviada pode ser igual ao tempo do primeiro evento da lista de eventos futuros. Esta correção pode ser vista na figura 5.7, linhas 15 e 16.

```

1 while ( (tempo_cabeca_lista_eventos =
2   tempo_cabeca_lista_eventos_futuros() ) >
3   (menor_clock = menor_clock_canais() ) )
4 {
5   recebe_mensagem();
6   /* chegou mensagem, eh necessario atualizar o clock do processo */
7   if (menor_clock != clock_proc)
8   {
9     clock_proc = menor_clock;
10    clock_atual = clock_proc + tempo_processo;
11
12    /* correcao para o caso de envio de mensagem nula
13     * com tempo maior do que o tempo
14     * do evento da cabeca da lista de eventos futuros */
15    if (clock_atual > tempo_cabeca_lista_eventos)
16      clock_atual = tempo_cabeca_lista_eventos;
17
18    /* processo bloqueado: envia mensagem nula */
19    envia_mensagem_nula (clock_atual);
20  }
21 }

```

Figura 5. 7: Trecho do código modificado da rotina cause()

5.3.2. Introdução de novas características

Além das correções aplicadas ao ParSMPLX, novas características foram incluídas, tanto por modificações em rotinas existentes, quanto pela inclusão de novas rotinas à biblioteca ParSMPLX e pela definição de uma metodologia para o cálculo do *lookahead* em programas de simulação.

Serão apresentadas, inicialmente, as rotinas alteradas para a inclusão de alguma nova característica. Em seguida, as novas rotinas para depuração que foram incorporadas ao ParSMPLX e, finalmente, será explicada a metodologia utilizada para o cálculo do *lookahead* dinâmico nos testes realizados.

I. Iniciação de variáveis

A rotina *smplt* () é a primeira rotina da biblioteca ParSMPL/ParSMPLX chamada num programa de simulação. Tem como finalidade iniciar algumas das variáveis da simulação.

Conforme foi explicado na seção 5.2, foram incluídas nesta função as iniciações de algumas variáveis do ParSMPLX que, originalmente, eram iniciadas nos arquivos de cabeçalho, o que não é permitido pelo compilador.

Também foi incluído nesta função um código para se definir um valor inicial para a semente utilizada para geração de números aleatórios, incrementando o valor dessa semente a cada chamada para *simpl()*. Isso foi feito para manter a coerência com o SMPL (MacDougall, 1987), extensão funcional na qual foi baseada o ParSMPL.

II. Adequação do cálculo do tempo entre eventos

A rotina *schedule()* é utilizada para escalonar o final do atendimento do *token* cliente no recurso e também o início do atendimento do *token* no próximo recurso. Esta rotina foi modificada para manter a coerência ao chamar a rotina *schedule()* em programas de simulação distribuída e facilitar sua utilização.

Um dos parâmetros a ser passado à rotina *schedule()* é o *tempo entre eventos*. Para escalonar eventos no próprio processo lógico local, o tempo entre eventos desejado deve ser informado, assim como é feito utilizando-se o SMPL (MacDougall, 1987). Para escalonar um evento em um processo remoto, segundo o código original do ParSMPL, o tempo entre eventos desejado deveria ser explicitamente somado ao valor do relógio da simulação do processo local antes de ser utilizado, no próprio código do usuário. Esta diferença entre a chamada da rotina para processo lógico local ou remoto não foi percebida no início da utilização do ParSMPLX. Assim, a chamada para ambos os casos era feita do mesmo modo, o que levou a erros de causa e efeito na simulação.

A rotina *schedule()* foi modificada para calcular internamente a soma do tempo entre eventos com o relógio da simulação, caso se tratasse de escalonamento de evento em processo remoto. O código original pode ser visto na figura 5.8, e sua versão modificada encontra-se na figura 5.9.

```

schedule ( int num_evento, double te, int token, int ident_proc)
{
    int bufid; /* para retorno de erro da envia_mensagem */
    if (te < 0.0) /* nao existe tempo negativo */
        erro("Rotina: schedule() - Erro: te menor que zero");

    /* verifica se evento e local ou remoto */
    if (tabela_tids[ident_proc] == tabela_tids[proc_local])

        /* evento vai ser escalonado localmente */
        insere_lista_eventos_futuros(num_evento, te, token, ident_proc);
    else
    {
        /* evento vai ser escalonado remotamente */
        bufid = envia_mensagem(num_evento, te, token, ident_proc);
        if (bufid < 0)
            erro("Rotina: schedule() - Erro: nao conseguiu enviar a
mensagem ");
    }
}

```

Figura 5. 8: Código original da rotina schedule()

```

void schedule (int num_evento, double te, int token, int ident_proc)
{
    int bufid; /* para retorno de erro do envia_mensagem */

    if (te < 0.0) /* nao existe tempo negativo */
        erro ("Rotina: schedule() - Erro: te menor que
zero");

    /* verifica se evento eh local ou remoto */
    if (tabela_tids[ident_proc] == tabela_tids[proc_local])
        /* evento local */
        insere_lista_eventos_futuros (num_evento, te, token,
ident_proc);
    else /* evento serah escalonado remotamente */
    {
        te = te + clock_simulacao;
        bufid = envia_mensagem (num_evento, te, token,
ident_proc);
        if (bufid < 0)
            erro ("Rotina: schedule() - Erro: nao
conseguiu enviar a mensagem");
    }
}

```

Figura 5. 9: Novo código da rotina schedule()

III. Tratamento de erros ocorridos em programas escravos

A rotina *erro()* é chamada quando algum erro é detectado durante a execução da simulação. Uma mensagem de erro é impressa, avisando ao usuário sobre o erro ocorrido, encerrando então o programa de simulação.

Esta rotina foi adaptada para imprimir em arquivo a mensagem de erro, quando este erro for originário de programas escravos. Inicialmente (figura 5.10), a mensagem de erro era

apenas impressa na tela, e só era visível se esse erro fosse causado pelo processo pai. Caso a execução de um escravo resultasse em erro, a mensagem de erro não era visível, pois os processos escravos executam em *background*. Com a modificação, caso algum erro ocorra em processo escravo, é criado um arquivo chamado *erro_parsmpl*, indicando o número do processo lógico que resultou em erro e a mensagem de erro apropriada.

```
void erro (mensagem)
char *mensagem;
{
    printf ("\n %s\n", mensagem);
    parsmpl_pvm_exit ();
}
```

Figura 5. 10: Código original da rotina erro()

Uma outra modificação nesta rotina foi a inclusão de código para terminar também a execução dos demais processos lógicos. Sem esse código adicional, era preciso terminar manualmente os processos restantes nas diferentes máquinas. Ao final da rotina, a linha `exit(1)` foi incluída, indicando término anormal do programa. O novo código da rotina *erro()* pode ser vista na figura 5.11.

```
void erro (char *mensagem)
{
    if (proc_local != 0) /* processo escravo */
    {
        saida_escravos = fopen (SAIDA_ERRO, "a");
        fprintf (saida_escravos, "\n Processo %d: \t %s\n",
                proc_local, mensagem);
        fclose (saida_escravos);
    }
    else
    {
        printf ("\n %s\n", mensagem);
        for (i=0; i < total_canaais; i++)
            if ((i != proc_local) && (tabela_tids[i] != 0))
                pvm_kill (tabela_tids[i]);
        parsmpl_pvm_exit ();
        exit (1);
    }
}
```

Figura 5. 11: Novo código da rotina erro()

IV. Rotinas para depuração

Durante a fase de correção do ParSMPLX, foram criadas três rotinas que compõem uma interface para depuração de simulações distribuídas desenvolvidas com o ParSMPLX: *imprime()*, *imprime_lista_eventos_futuros()* e *imprime_buffer_canaais()*.

- *imprime()*: aceita como parâmetros um nome de arquivo, que pode ser um nome padrão ou um nome definido pelo usuário, mais a mensagem para

depuração que deverá ser escrita nesse arquivo. Como foi visto anteriormente, os processos escravos não possuem saída de tela, daí a necessidade da utilização de arquivos. São criados arquivos separados para cada processo lógico, sendo diferenciados pelo número do processo lógico.

- *imprime_lista_eventos_futuros()*: percorre a lista de eventos futuros do processo lógico e imprime em arquivo o tempo de ocorrência de cada evento presente nessa lista.
- *imprime_buffer_canais()*: aceita como parâmetro o número do canal desejado e imprime em arquivo o tempo de ocorrência das mensagens contidas nesse canal.

Essas rotinas são chamadas em alguns pontos do código fonte *parsmpl.c*, onde é relevante saber valores de variáveis, como ao inserir eventos na lista de eventos futuros, enviar e receber mensagens, etc. Para ativar a impressão dessas mensagens de depuração, basta compilar incluindo a diretiva “-DDEBUG” no makefile.

Alternativamente, essas rotinas de depuração podem ser utilizadas diretamente no programa de simulação do usuário, quando não se deseja depurar a biblioteca, mas apenas o programa de simulação. Neste caso, não é necessário utilizar a diretiva “-DDEBUG” ao compilar, reduzindo assim o número de mensagens impressas nos arquivos.

V. Metodologia para cálculo do *lookahead* dinâmico

Como tentativa de melhorar o *speedup*, foram feitos testes utilizando-se *lookahead* dinâmico. A cada ciclo da simulação, um novo valor para o *lookahead* é calculado, considerando-se uma pré-amostragem dos tempos de serviço de cada recurso do processo lógico em questão.

Os exemplos apresentados a seguir mostram como obter uma fórmula para o cálculo do *lookahead* dinâmico, a partir da análise da disposição dos recursos no processo lógico. Suponha que cada exemplo seja um processo lógico, ou seja, cada exemplo representa apenas a parte do modelo sendo simulada em um determinado processador. Considere:

tempo_processo = variável do ParSMPLX que contém o valor do *lookahead*.

TS_X = tempo de serviço do recurso X.

Exemplo 1: No primeiro exemplo, mostrado na figura 5.12, tem-se um processo lógico com apenas um recurso. Neste caso, o tempo de serviço deste recurso será utilizado como *lookahead*.

$$\therefore \text{tempo_processo} = TS_A$$

Figura 5. 12: Processo lógico com apenas um recurso

Exemplo 2: Neste exemplo, é apresentado um processo lógico com recursos em série (figura 5.13). Neste caso, o *lookahead* é obtido somando-se os valores dos tempos de serviço desses recursos.

$$\therefore \text{tempo_processo} = TS_A + TS_B$$

Figura 5. 13: Processo lógico com apenas um recurso

Exemplo 3: No exemplo mostrado na figura 5.14, os recursos estão todos em paralelo. Quando isso ocorre, o *lookahead* considerado será igual ao menor tempo de serviço entre eles.

$$\therefore \text{tempo_processo} = \text{menor_tempo} (TS_A, TS_B, TS_C)$$

Figura 5. 14: Processo lógico com recursos em paralelo

Exemplo 4: A figura 5.15 mostra o caso de um sistema que pode ser alimentado a partir do recurso A e do recurso C. Assim, os tempos de serviço dos recursos A e B não podem ser considerados, pois todo *token* passará obrigatoriamente por C, mas não há garantias de que A e B serão sempre utilizados.

$$\therefore \text{tempo_processo} = TS_C$$

Figura 5. 15: Processo lógico com mais de um ponto de alimentação

Exemplo 5: Um pouco semelhante ao caso anterior, o sistema da figura 5.16 possui uma saída antes do recurso C. Isso faz com que haja a possibilidade do recurso C não ser utilizado em alguns casos. Esse recurso deve ser, então, descartado do cálculo do *lookahead*.

$$\therefore \text{tempo_processo} = TS_A + TS_B$$

Figura 5. 16: Processo lógico com mais de uma saída

Exemplo 6: O caso mostrado na figura 5.17 é um exemplo mais elaborado envolvendo a combinação das regras vistas anteriormente. Os recursos que estão em série devem ter seus tempos de serviço somados, para os recursos em paralelo deve ser considerado apenas o menor tempo de serviço e os recursos anteriores a entradas ou posteriores a saídas do sistema não devem ser considerados.

$$\therefore \text{tempo_processo} = TS_B + \text{menor_tempo}(TS_C + TS_E, TS_D + TS_G, TS_E)$$

Figura 5. 17 Processo lógico combinado

Como a informação necessária para a elaboração da fórmula para o cálculo do *lookahead* dinâmico depende de conhecimento sobre o modelo, essa fórmula deve ser obtida pelo usuário e inserida no código da simulação – ela não é obtida automaticamente pelo ParSMPLX.

5.4. Validação

Depois de aplicar as alterações para correção da biblioteca ParSMPLX, foi necessário seguir com um processo de validação, visando garantir que essas alterações não passassem a interferir nos resultados esperados da simulação.

Para efetuar o processo de validação foram feitos testes com modelos de redes de filas. Os resultados obtidos pela simulação com o ParSMPLX foram então confrontados com os resultados obtidos por simulação sequencial. No primeiro modelo, o resultado foi comparado também ao resultado por solução analítica. Os dados coletados foram a utilização de cada servidor e o tamanho médio da fila de espera. Os programas para simulação sequencial utilizam a biblioteca SMPL (MacDougall, 1987) e foram gerados pelo ASiA (Santana, 1996). Os resultados obtidos por solução analítica foram extraídos do trabalho de Silva (Silva, 2000).

5.4.1. Modelo 1 – Servidor de arquivos simples

O modelo 1 (Silva, 2000) representa um servidor de arquivos, na qual as solicitações são feitas ao processador. Essas solicitações podem ser enfileiradas ou atendidas imediatamente, conforme a situação do processador. Após utilizar o processador, existe a possibilidade de se acessar o disco e retornar ao processador, ou então deixar o sistema.

Foram considerados os seguintes parâmetros de entrada:

- tempo entre chegada de clientes: 0.0833;
- tempo de serviço do processador: 0.04;
- tempo de serviço do disco: 0.05;
- tempo simulado: 1000000;
- distribuições exponenciais para tempo de serviço e para tempo entre chegadas.

Figura 5. 18: Servidor de Arquivos

Os testes de validação foram feitos utilizando *lookahead* estático, considerando-se um valor de incremento mínimo igual a 0.1, e *lookahead* dinâmico, considerando-se o valor pré-calculado do tempo de serviço do recurso (processador para o processo lógico 1 e disco para o processo lógico 2). Foi feita uma execução para cada uma das 15 sementes para geração de números aleatórios definidas no SMPL e ParSMPLX.

As tabelas 5.1 a 5.3 mostram as médias dos resultados obtidos. Pode-se perceber que as médias dos resultados obtidos pela simulação distribuída são bem próximas, tanto comparadas à simulação seqüencial quanto à solução analítica. Os resultados individuais da execução de cada uma das 15 sementes encontram-se nas tabelas 5.4 (seqüencial utilizando SMPL), 5.5 (paralelo, utilizando ParSMPLX e *lookahead* estático) e 5.6 (paralelo, utilizando ParSMPLX e *lookahead* dinâmico).

Tabela 5. 1: Resultados da simulação com o SMPL

	Utilização do processador	Utilização do disco	Tamanho Médio da Fila do processador	Tamanho Médio da Fila do disco
média	0.80	0.40	3.20	0.27
desvio padrão	0.00027568	0.000159	0.00201659	3.9825E-09
intervalo de confiança	0.00013951	8.07E-05	0.00102052	2.0154E-09
Solução Analítica	0.8	0.4	3.2	0.2667

Tabela 5. 2: Resultados da simulação com o ParSMPLX, utilizando *lookahead* estático

	Utilização do processador	Utilização do disco	Tamanho Médio da Fila do processador	Tamanho Médio da Fila do disco
média	0.80	0.40	3.27	0.27
desvio padrão	0.000398720	0.000168167	0.011481876	0.000490928
intervalo de confiança	0.000201776	0.000085103	0.005810521	0.000248439
Solução Analítica	0.8	0.4	3.2	0.2667

Tabela 5. 3: Resultados da simulação com o ParSMPLX, utilizando *lookahead* dinâmico

	Utilização do processador	Utilização do disco	Tamanho Médio da Fila do processador	Tamanho Médio da Fila do disco
média	0.80	0.40	3.36	0.27
desvio padrão	0.00737	0.00554	0.37216	0.01621
intervalo de confiança	0.00373	0.00280	0.18834	0.00820
Solução Analítica	0.8	0.4	3.2	0.2667

Tabela 5. 4: Resultados das simulações do Servidor de Arquivo Simples com o SMPL

Semente	Utilização do processador	Utilização do disco	Tamanho Médio da Fila do processador	Tamanho Médio da Fila do disco
1	0.800	0.400	3.199	0.266
2	0.801	0.400	3.201	0.266
3	0.800	0.400	3.202	0.266
4	0.800	0.400	3.199	0.266
5	0.801	0.400	3.201	0.266
6	0.800	0.400	3.203	0.266
7	0.800	0.400	3.200	0.266
8	0.801	0.400	3.200	0.266
9	0.800	0.400	3.203	0.266
10	0.800	0.400	3.200	0.266
11	0.801	0.400	3.199	0.266
12	0.800	0.400	3.204	0.266
13	0.800	0.400	3.199	0.266
14	0.801	0.400	3.199	0.266
15	0.800	0.400	3.205	0.266

Tabela 5. 5: Resultados das simulações do Servidor de Arquivos Simples com o ParSMPLX, utilizando *lookahead* estático

Semente	Utilização do processador	Utilização do disco	Tamanho Médio da Fila do processador	Tamanho Médio da Fila do disco
1	0.800	0.400	3.289	0.266
2	0.801	0.401	3.284	0.267
3	0.799	0.400	3.273	0.267
4	0.800	0.400	3.268	0.267
5	0.800	0.400	3.270	0.267
6	0.800	0.400	3.262	0.266
7	0.801	0.400	3.283	0.267
8	0.800	0.400	3.242	0.267
9	0.800	0.400	3.273	0.267
10	0.800	0.400	3.262	0.267
11	0.800	0.400	3.262	0.267
12	0.800	0.400	3.271	0.268
13	0.800	0.400	3.269	0.268
14	0.801	0.400	3.282	0.268
15	0.800	0.400	3.269	0.268

Tabela 5. 6: Resultados das simulações do Servidor de Arquivos Simples com o ParSMPLX, utilizando *lookahead* dinâmico

Semente	Utilização do processador	Utilização do disco	Tamanho Médio da Fila do processador	Tamanho Médio da Fila do disco
1	0.804	0.403	3.413	0.302
2	0.808	0.404	3.497	0.290
3	0.809	0.395	3.430	0.256
4	0.803	0.399	3.725	0.292
5	0.798	0.394	2.969	0.254
6	0.822	0.411	3.901	0.279
7	0.802	0.402	3.000	0.270
8	0.800	0.401	2.975	0.258
9	0.794	0.396	3.148	0.253
10	0.804	0.403	3.512	0.262
11	0.802	0.408	2.950	0.265
12	0.794	0.402	2.966	0.253
13	0.800	0.405	3.119	0.276
14	0.814	0.411	4.040	0.290
15	0.808	0.411	3.739	0.277

Como pode ser visto nas tabelas 5.4 a 5.6, em todos os casos os valores estão muito próximos, praticamente constantes para qualquer semente. O valor do desvio padrão, próximo de zero, reflete o fato observado.

5.4.2. Modelo 2 – Servidor de arquivos com 4 discos

O modelo 2 (Ulson, 1999) representa um servidor de arquivos com quatro discos. As solicitações são feitas ao processador, que irá atendê-las caso esteja livre, ou então enfileirá-las se já estiver ocupado processando outra requisição. Após utilizar o processador, é escolhido um entre os quatro discos. Depois de utilizar o disco escolhido, a requisição retorna ao processador, e então o ciclo reinicia.

Foram considerados os seguintes parâmetros para a simulação:

- número de tarefas no sistema: 50;
- tempo de serviço do processador: 10;
- tempo de serviço de cada disco: média 30, com desvio padrão 7.5;
- tempo simulado: 1000000;
- distribuição exponencial para o tempo entre chegadas e distribuição erlang para os tempos de serviço de cada disco;
- probabilidade de escolha de cada disco: 25%.

Figura 5. 19: Servidor de arquivos com 4 discos

Assim como no teste de validação 5.4.1, foi escolhido como *lookahead* estático o valor de incremento 0.1. Para o *lookahead* dinâmico foi considerado o valor pré-calculado do tempo de serviço do processador, para o processo lógico que compreende o processador, e

para o processo lógico que compreende os quatro discos, foi considerado o menor valor entre os tempos de serviço de cada um dos discos, previamente calculados.

As tabelas 5.7 a 5.9 mostram as médias dos resultados obtidos pela simulação sequencial (SMPL) e paralela (ParSMPLX com *lookahead* estático e dinâmico), respectivamente. Neste exemplo também é possível notar que as médias dos resultados obtidos pela simulação paralela são semelhantes à média dos resultados obtidos pela simulação sequencial.

Tabela 5. 7: Resultados da simulação com o SMPL

	Média dos resultados com SMPL	Desvio Padrão	Intervalo de Confiança
Utilização do processador	1	0	#NÚM!
Utilização do disco 1	0.75	0.003899	0.0019731
Utilização do disco 2	0.75	0.0048323	0.0024454
Utilização do disco 3	0.75	0.0041899	0.0021204
Utilização do disco 4	0.75	0.0041038	0.0020768
Tam. Méd. Fila (disco 1)	1.20	0.0404924	0.0204916
Tam. Méd. Fila (disco 2)	1.20	0.0482523	0.0244186
Tam. Méd. Fila (disco 3)	1.18	0.0467956	0.0236814
Tam. Méd. Fila (disco 4)	1.19	0.0713554	0.0361102
Tam. Méd. Fila (processador)	41.22	0.1190063	0.0602243

Tabela 5. 8: Resultados da simulação com o ParSMPLX, utilizando *lookahead* estático

	Média dos resultados com ParSMPLX	Desvio Padrão	Intervalo de Confiança
Utilização do processador	1	0	#NÚM!
Utilização do disco 1	0.75	0.003716	0.0018805
Utilização do disco 2	0.75	0.0040777	0.0020635
Utilização do disco 3	0.75	0.0044065	0.0022299
Utilização do disco 4	0.75	0.0049296	0.0024947
Tam. Méd. Fila (disco 1)	1.21	0.0491329	0.0248642
Tam. Méd. Fila (disco 2)	1.21	0.0465876	0.0235762
Tam. Méd. Fila (disco 3)	1.17	0.0503124	0.0254611
Tam. Méd. Fila (disco 4)	1.18	0.0414065	0.0209542
Tam. Méd. Fila (processador)	41.23	0.1085339	0.0549247

Tabela 5. 9: Resultados da simulação com o ParSMPLX, utilizando *lookahead* dinâmico

	Média dos resultados com ParSMPLX	Desvio Padrão	Intervalo de Confiança
Utilização do processador	1.00	3.15016E-06	1.59417E-06
Utilização do disco 1	0.75	0.003257434	0.001648458
Utilização do disco 2	0.75	0.008701942	0.004403707
Utilização do disco 3	0.75	0.005343887	0.002704329
Utilização do disco 4	0.76	0.007257809	0.003672889
Tam. Méd. Fila (disco 1)	40.94	0.154252987	0.0780613
Tam. Méd. Fila (disco 2)	1.20	0.029432957	0.014894848
Tam. Méd. Fila (disco 3)	1.25	0.042446131	0.021480298
Tam. Méd. Fila (disco 4)	1.32	0.034959541	0.017691633
Tam. Méd. Fila (processador)	1.27	0.061523662	0.031134678

Os resultados da execução individual de cada uma das 15 sementes encontram-se nas tabelas 5.10 e 5.11 (seqüencial utilizando SMPL), 5.12 e 5.13 (paralelo, utilizando ParSMPLX com *lookahead* estático), 5.14 e 5.15 (paralelo, utilizando ParSMPLX com *lookahead* dinâmico). Pode-se perceber pelos valores dessas tabelas que as simulações seqüencial e paralela (tanto com *lookahead* estático quanto dinâmico) apresentaram comportamento semelhante na execução das 15 diferentes sementes.

Tabela 5. 10: Resultados das simulações com o SMPL: utilização dos recursos

Semente	Utilização do processador	Utilização do disco 1	Utilização do disco 2	Utilização do disco 3	Utilização do disco 4
1	1.000	0.748	0.748	0.752	0.759
2	1.000	0.750	0.749	0.743	0.756
3	1.000	0.746	0.752	0.744	0.748
4	1.000	0.752	0.755	0.749	0.751
5	1.000	0.748	0.741	0.753	0.751
6	1.000	0.750	0.755	0.744	0.753
7	1.000	0.751	0.757	0.744	0.753
8	1.000	0.755	0.749	0.751	0.753
9	1.000	0.747	0.751	0.746	0.746
10	1.000	0.748	0.753	0.756	0.754
11	1.000	0.762	0.742	0.747	0.752
12	1.000	0.753	0.750	0.745	0.749
13	1.000	0.748	0.752	0.750	0.742
14	1.000	0.750	0.748	0.754	0.750
15	1.000	0.752	0.757	0.744	0.754

Tabela 5. 11: Resultados das simulações com o SMPL: tamanho médio da fila dos recursos

Semente	Tamanho médio da Fila do processador	Tamanho médio da Fila do disco 1	Tamanho médio da Fila do disco 2	Tamanho médio da Fila do disco 3	Tamanho médio da Fila do disco 4
1	41.176	1.206	1.146	1.281	1.182
2	41.147	1.193	1.144	1.279	1.239
3	41.435	1.184	1.108	1.127	1.155
4	41.131	1.247	1.201	1.213	1.202
5	41.350	1.129	1.220	1.168	1.140
6	41.209	1.212	1.165	1.253	1.158
7	41.067	1.267	1.107	1.281	1.272
8	41.189	1.247	1.209	1.158	1.190
9	41.283	1.226	1.166	1.143	1.193
10	41.066	1.282	1.196	1.248	1.197
11	41.226	1.111	1.169	1.229	1.264
12	41.394	1.187	1.084	1.102	1.236
13	41.390	1.172	1.219	1.063	1.164
14	41.164	1.220	1.241	1.143	1.200
15	41.144	1.252	1.136	1.129	1.242

Tabela 5. 12: Resultados das simulações com o ParSMPLX utilizando *lookahead* estático: utilização dos recursos

Semente	Utilização do processador	Utilização do disco 1	Utilização do disco 2	Utilização do disco 3	Utilização do disco 4
1	1.000	0.750	0.745	0.760	0.757
2	1.000	0.747	0.749	0.755	0.750
3	1.000	0.742	0.748	0.748	0.754
4	1.000	0.748	0.753	0.753	0.754
5	1.000	0.749	0.754	0.756	0.747
6	1.000	0.748	0.743	0.754	0.758
7	1.000	0.751	0.747	0.749	0.756
8	1.000	0.759	0.754	0.754	0.749
9	1.000	0.752	0.750	0.754	0.751
10	1.000	0.749	0.742	0.745	0.755
11	1.000	0.746	0.748	0.746	0.753
12	1.000	0.751	0.744	0.750	0.752
13	1.000	0.752	0.752	0.754	0.742
14	1.000	0.750	0.743	0.746	0.757
15	1.000	0.751	0.749	0.746	0.743

Tabela 5. 13: Resultados das simulações com o ParSMPLX utilizando *lookahead* estático: tamanho médio da fila dos recursos

Semente	Tamanho médio da Fila do processador	Tamanho médio da Fila do disco 1	Tamanho médio da Fila do disco 2	Tamanho médio da Fila do disco 3	Tamanho médio da Fila do disco 4
1	41.106	1.193	1.208	1.268	1.214
2	41.234	1.175	1.251	1.181	1.157
3	41.172	1.179	1.224	1.245	1.187
4	41.206	1.186	1.179	1.268	1.152
5	41.123	1.259	1.222	1.203	1.188
6	41.387	1.117	1.151	1.196	1.146
7	41.218	1.183	1.159	1.223	1.213
8	41.016	1.254	1.261	1.140	1.313
9	41.229	1.148	1.145	1.200	1.272
10	41.367	1.151	1.078	1.210	1.204
11	41.341	1.119	1.116	1.256	1.175
12	41.269	1.139	1.170	1.170	1.254
13	41.111	1.261	1.168	1.200	1.260
14	41.269	1.161	1.150	1.207	1.256
15	41.353	1.196	1.138	1.132	1.193

Tabela 5. 14: Resultados das simulações com o ParSMPLX utilizando *lookahead* dinâmico: utilização dos recursos

Semente	Utilização do processador	Utilização do disco 1	Utilização do disco 2	Utilização do disco 3	Utilização do disco 4
1	1.000	0.751	0.742	0.749	0.748
2	1.000	0.756	0.741	0.747	0.746
3	1.000	0.749	0.760	0.754	0.761
4	1.000	0.748	0.760	0.752	0.762
5	1.000	0.754	0.761	0.756	0.762
6	1.000	0.749	0.761	0.759	0.765
7	1.000	0.758	0.760	0.758	0.760
8	1.000	0.747	0.751	0.762	0.762
9	1.000	0.751	0.759	0.754	0.761
10	1.000	0.755	0.739	0.744	0.743
11	1.000	0.749	0.740	0.748	0.749
12	1.000	0.752	0.750	0.759	0.764
13	1.000	0.750	0.751	0.754	0.757
14	1.000	0.754	0.760	0.760	0.764
15	1.000	0.755	0.758	0.761	0.762

Tabela 5. 15: Resultados das simulações com o ParSMPLX utilizando *lookahead* estático: tamanho médio da fila dos recursos

Semente	Tamanho médio da Fila do processador	Tamanho médio da Fila do disco 1	Tamanho médio da Fila do disco 2	Tamanho médio da Fila do disco 3	Tamanho médio da Fila do disco 4
1	41.249	1.152	1.172	1.273	1.163
2	40.952	1.190	1.250	1.350	1.210
3	40.864	1.229	1.304	1.311	1.268
4	40.926	1.179	1.272	1.289	1.311
5	40.837	1.209	1.283	1.297	1.340
6	40.829	1.192	1.290	1.336	1.319
7	40.835	1.269	1.263	1.300	1.298
8	40.992	1.201	1.213	1.281	1.290
9	40.968	1.190	1.250	1.290	1.276
10	41.010	1.210	1.260	1.320	1.210
11	41.257	1.165	1.161	1.284	1.147
12	40.752	1.227	1.272	1.390	1.335
13	41.066	1.188	1.213	1.301	1.221
14	40.820	1.219	1.284	1.341	1.298
15	40.789	1.234	1.269	1.372	1.302

Além dos dois testes descritos, outros foram realizados e, a partir dos resultados obtidos, a biblioteca ParSMPLX foi considerada validada e sua utilização adequada para a simulação distribuída.

5.5. Considerações Finais

Vários fatores dificultaram o processo de adaptação, aprimoramento e validação do ParSMPLX.

A depuração de programas em sistemas Linux não é tarefa fácil. A falta de ambientes de depuração adequados, aliada à dificuldade inerente à depuração quando se trata de programas paralelos, serviram de agravantes ao adaptar e aprimorar o ParSMPLX.

As informações presentes em (Ulson, 1999) sobre como escrever programas de simulação utilizando ParSMPL não foram suficientes para ajudar na elaboração de um programa de simulação. Em (Ulson, 1999), parte-se do pressuposto de que o usuário já está familiarizado com o desenvolvimento de simulações utilizando a extensão SMPL. Apesar da existência de descrições da funcionalidade das principais primitivas do SMPL (que continuam utilizadas no ParSMPL), não é feita nenhuma descrição ou explicação sobre os parâmetros esperados por cada primitiva, nem é feito nenhum aviso sobre alterações que ocorreram nos

parâmetros das primitivas SMPL em relação às do ParSMPL (discutido na seção 5.3.2). Isso levou a implementações erradas de programas de simulação com o ParSMPL no início do trabalho, o que acarretou em perda de tempo na busca de erros que na verdade não deveriam ocorrer.

Os problemas encontrados no início do trabalho tornaram clara a necessidade de um guia de utilização do ParSMPLX. Esse guia de utilização foi elaborado (e encontra-se no apêndice A) visando facilitar a utilização do ParSMPLX tanto por usuários já familiarizados com o SMPL quanto por usuários inexperientes.

Nesse guia é feita uma descrição da API do ParSMPLX, de modo a facilitar o trabalho de desenvolvimento de programas de simulação. Para facilitar o entendimento de cada primitiva, é mostrado um exemplo da utilização de cada primitiva explicada.

No ParSMPLX, há algumas variáveis que obrigatoriamente devem ter seu valor iniciado no programa de simulação do usuário. Essas variáveis estão descritas no guia, assim como os possíveis valores que elas podem assumir e seus significados dentro da simulação.

A ordem em que algumas das primitivas do ParSMPLX devem ser chamadas é relevante, tanto em relação a outras primitivas quanto em relação à definição do valor inicial de variáveis. Isso também é explicado no guia de utilização, assim como os efeitos causados pela não-obediência dessa ordem.

No guia também é mostrado em quais pontos no código de simulação do usuário devem ser feitas as definições de *lookahead* dinâmico (ou do *lookahead* estático). Finalmente, tem-se uma explicação sobre a criação dos códigos-objeto do ParSMPLX e compilação dos programas de simulação.

Capítulo 6 – Análise do Desempenho do Protocolo Conservativo

6.1. Considerações Iniciais

A análise de desempenho desenvolvida neste trabalho visa, principalmente, obter as informações necessárias para o desenvolvimento do nível 3 do módulo avaliador do ASDA, que é responsável pela decisão de quando utilizar o protocolo conservativo. Esta decisão se baseará em resultados apresentados em (Ulson, 1999) e nos resultados discutidos neste capítulo.

A seção 6.2 descreve o ambiente de testes utilizado, os modelos avaliados nas simulações e a parametrização. As seções seguintes apresentam os testes realizados. Na seção 6.3 é avaliada a utilização de *lookahead* dinâmico com o ParSMPLX. A avaliação da influência do aumento da carga de clientes no sistema é feita na seção 6.4, da inserção de laços de granulosidade é vista na seção 6.5. Na seção 6.6, são analisadas a influência do tamanho do modelo sendo simulado e o balanceamento entre os processos lógicos da simulação. Na seção 6.7 são apresentadas diretivas para inclusão no módulo avaliador e a seção 6.8 apresenta as considerações finais deste capítulo.

6.2. Descrição dos Testes Realizados

6.2.1. Ambiente

Os testes foram realizados no Laboratório de Sistemas Distribuídos e Programação Concorrente (LaSDPC), utilizando computadores pessoais interligados via rede Ethernet de 100 Mbits/s. As máquinas possuem sistema operacional Linux Red Hat versão 7.2, kernel 2.4.7-10 e a comunicação entre os processos lógicos foi feita utilizando-se o ambiente de passagem de mensagens PVM versão 3.4.2.

A tabela 6.1 mostra a configuração das máquinas (mestre e escravo) disponíveis para os testes. Todos testes foram feitos utilizando sempre duas das três máquinas relacionadas na tabela 6.1. A máquina *lasdpc04* foi utilizada apenas em alguns testes, sendo substituída posteriormente pela *lasdpc12*. Nos casos em que a máquina *lasdpc04* foi utilizada como escravo ao invés da *lasdpc12*, isto estará explicitamente especificado no início da descrição do teste.

Tabela 6. 1: Configuração das máquinas

Máquina	Processador	Memória RAM	Disco
Mestre (<i>lasdpc06</i>)	Pentium III 450 MHz	256MB	6 GB
Escravo (<i>lasdpc12</i>)	Pentium II MMX 350 MHz	128 MB	8GB
Escravo (<i>lasdpc04</i>)	Pentium II 400 MHz	128MB	13GB

Para minimizar os efeitos de ruídos no meio de comunicação, os testes foram realizados utilizando a rede local do laboratório em horários reservados e com a rede externa desconectada.

A estimativa dos tempos gastos com transmissão de mensagens nulas e completas foi feita utilizando-se o *benchmark* ping-pong (Dilon et al. 1995, apud Ulson, 1999). Uma descrição desse algoritmo pode ser encontrado em (Ulson, 1999).

6.2.2. Modelos

Os modelos utilizados para os testes de desempenho com o ParSMPLX foram alguns dos propostos em (Ulson, 1999). As diferentes configurações de um mesmo modelo dizem respeito à forma como estes foram particionados em processos lógicos.

Os seguintes modelos foram utilizados nos testes:

I. Modelo 1 – Servidor Central (MacDougall, 1987)

Representa um sistema computacional constituído por um processador e quatro unidades de disco. Nesse sistema existe um número fixo de tarefas, que requisitam a utilização da CPU e, após o término do processamento, escolhem uma entre as quatro

Das opções de particionamento propostas em (Ulson, 1999), os testes foram efetuados considerando-se a configuração 1 (figura 6.1). Este modelo foi utilizado para avaliar o impacto da utilização de *lookahead* dinâmico na simulação.

Parametrização do modelo:

Parâmetro	Valor
Recurso: CPU	tempo de serviço = expntl (10.0)
Recurso: DISCO 1	tempo de serviço = erlang (30, 7.5)
Recurso: DISCO 2	tempo de serviço = erlang (30, 7.5)
Recurso: DISCO 3	tempo de serviço = erlang (30, 7.5)
Recurso: DISCO 4	tempo de serviço = erlang (30, 7.5)
número de tarefas no sistema	50
tempo simulado	1 000 000

II. Modelo 2 – Sistema Computacional

Corresponde ao Modelo 4 de (Ulson, 1999). Representa um sistema computacional constituído por uma unidade de processamento e duas unidades de disco. As tarefas chegam ao sistema obedecendo a uma certa taxa de chegada e requisitam a utilização da CPU. Essas requisições são imediatamente atendidas, caso a CPU esteja livre. Caso contrário, elas são colocadas em fila. Após o processamento, a tarefa pode deixar o sistema ou escolher uma das duas unidades de disco. Sempre que uma tarefa escolher utilizar um dos discos, a tarefa deve voltar para a CPU depois da utilização do disco escolhido.

Em (Ulson, 1999) foram propostas três opções de particionamento. Dessas, foram utilizadas nos testes as configurações 1 (figura 6.2) e 2 (figura 6.3). Os testes feitos com este modelo foram utilizados na avaliação do uso de *lookahead* dinâmico e da influência do aumento da carga no desempenho.

Figura 6. 2: Modelo 2 - configuração 1

Figura 6. 3: Modelo 2 - configuração 2

Parametrização do modelo:

Tabela 6. 3: Parametrização do Modelo 2

Parâmetro	Valor
Recurso: CPU	tempo de serviço = expntl (10.0)
Recurso: DISCO 1	tempo de serviço = erlang (30, 7.5)
Recurso: DISCO 2	tempo de serviço = erlang (30, 7.5)
Número de tarefas iniciais no sistema	1
Chegada de novas tarefas (parâmetro testado com diferentes valores)	<ul style="list-style-type: none"> tempo entre chegadas = expntl (50) tempo entre chegadas = expntl (20) tempo entre chegadas = expntl (10) tempo entre chegadas = expntl (8)
tempo simulado	1 000 000

III. Modelo 3

Corresponde ao Modelo Hipotético 1 de (Ulson, 1999). Esse modelo descreve um sistema hipotético composto por sete recursos, sendo que três deles estão organizados em série e os demais estão relacionados a pontos de decisão.

O número de tarefas circulando no sistema é fixo e sempre que uma tarefa chega ao sistema, deve utilizar os serviços dos recursos identificados por REC1, REC2 e REC3. Ao término do processamento no recurso REC3 é feita uma escolha probabilística, e assim o próximo recurso a ser utilizado é selecionado. Em função desse recurso, a tarefa pode ou retornar para o recurso REC3 ou retornar ao início do sistema para o recurso REC1.

As duas configurações (figuras 6.4 e 6.5) propostas em (Ulson, 1999) foram consideradas nos testes. Uma terceira configuração foi utilizada nos testes (figura 6.6). Este modelo foi utilizado para avaliar a influência do *lookahead* dinâmico (as duas primeiras configurações) e da granulosidade (todas as configurações).

Figura 6. 4: Modelo 3 - configuração 1

Figura 6. 5: Modelo 3 – configuração 2

Figura 6. 6: Modelo 3 – configuração 3

Parametrização do modelo:

Tabela 6. 4: Parametrização do Modelo 3

Parâmetro	Valor
Recurso: REC1	tempo de serviço = expntl (10.0)
Recurso: REC2	tempo de serviço = expntl (10.0)
Recurso: REC3	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC4	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC5	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC6	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC7	tempo de serviço = erlang (30.0, 7.5)
Número de tarefas iniciais no sistema	50
Laço de granulosidade (parâmetro testado com diferentes valores)	<ul style="list-style-type: none">sem laço de granulosidadecom laço de 25000 voltascom laço de 50000 voltas
Tempo simulado	1 000 000

IV. Modelo 4

Corresponde ao Modelo Hipotético 2 de (Ulson, 1999). O Modelo 4 é semelhante ao Modelo 3, sendo diferenciado deste pelo número de tarefas que circulam pelo sistema. Enquanto no Modelo 3 esse número era fixo, o Modelo 4 é um sistema aberto, no qual novas tarefas chegam ao sistema obedecendo uma certa taxa de chegada e existe uma probabilidade para que as tarefas deixem o sistema.

Das duas opções de particionamento definidas em (Ulson, 1999), a configuração 1 foi utilizada nos testes (figura 6.6). Este modelo foi utilizado para avaliar o impacto do *lookahead* dinâmico, a influência do aumento da carga e do tamanho do modelo no desempenho.

Figura 6. 7: Modelo 4

Parametrização do modelo:

Tabela 6. 5: Parametrização do Modelo 4

Parâmetro	Valor
Recurso: REC1	tempo de serviço = expntl (10.0)
Recurso: REC2	tempo de serviço = expntl (15.0)
Recurso: REC3	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC4	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC5	tempo de serviço = erlang (30.0, 7.5)

Recurso: REC6	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC7	tempo de serviço = erlang (30.0, 7.5)
Número de tarefas iniciais no sistema	1
Chegada de novas tarefas (parâmetro testado com diferentes valores)	<ul style="list-style-type: none"> • tempo entre chegadas = expntl (15) • tempo entre chegadas = expntl (12.5) • tempo entre chegadas = expntl (10) • tempo entre chegadas = expntl (8.5)
tempo simulado	1 000 000
probabilidade de saída (parâmetro testado com diferentes valores)	<ul style="list-style-type: none"> • 40% • 80%

V. Modelo 5

O modelo 5 corresponde a uma duplicação do Modelo 4. Para compor um só modelo, mas de tamanho maior, cada cópia do Modelo 4 foi colocada em um processo lógico, a saída do primeiro foi utilizada como alimentação para o segundo.

Este modelo foi utilizado para avaliar a influência da granulosidade e do tamanho do modelo no desempenho da simulação.

Figura 6. 8: Modelo 5

Parametrização do modelo:

Tabela 6. 6: Parametrização do Modelo 5

Parâmetro	Valor
Recurso: REC1	tempo de serviço = expntl (10.0)
Recurso: REC2	tempo de serviço = expntl (15.0)
Recurso: REC3	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC4	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC5	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC6	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC7	tempo de serviço = erlang (30.0, 7.5)

Recurso: REC8	tempo de serviço = expntl (10.0)
Recurso: REC9	tempo de serviço = expntl (15.0)
Recurso: REC10	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC11	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC12	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC13	tempo de serviço = erlang (30.0, 7.5)
Recurso: REC17	tempo de serviço = erlang (30.0, 7.5)
Número de tarefas iniciais no sistema	1
Chegada de novas tarefas (parâmetro testado com diferentes valores)	<ul style="list-style-type: none"> tempo entre chegadas = expntl (15) tempo entre chegadas = expntl (12.5) tempo entre chegadas = expntl (10) tempo entre chegadas = expntl (8.5)
tempo simulado	1 000 000
probabilidade de saída de PL0 para PL1 (parâmetro testado com diferentes valores)	<ul style="list-style-type: none"> 40% 80%

6.3. Avaliação do Lookahead Dinâmico

Para a avaliação do *lookahead* dinâmico foram comparados os resultados de testes dos Modelos 1, 2, 3 e 4 utilizando *lookahead* estático e dinâmico.

O Modelo 1 foi executado com as configurações descritas na seção 6.2.2. O parâmetro utilizado como tempo entre chegadas de novos clientes para o Modelo 2 foi a função exponencial negativa com média 50 – função *expntl* (50). O Modelo 3 foi executado com os parâmetros descritos na seção 6.2.2, sem a utilização de laço de granulosidade. Para esses três modelos a máquina utilizada como escrava foi a *laspc04*. No Modelo 4, a máquina utilizada como escrava foi a *laspc12*, e foram testadas as seguintes configurações com diferentes valores para o tempo entre chegadas:

- a. configuração 1A: foi utilizada a função *expntl* (12.5).
- b. configuração 1B: foi utilizada a função *expntl* (10).
- c. configuração 1C: foi utilizada a função *expntl* (8.5).

Na tabela 6.7, encontram-se descritos os tempos de execução dos modelos, em segundos. Para o tempo seqüencial, foi considerada a média do tempo da simulação seqüencial executada na máquina mestre e na máquina escrava. Na tabela 6.8 encontram-se listados os valores de *speedup* alcançados pela execução de cada modelo.

Tabela 6. 7: Tempo de execução dos modelos, em segundos

Modelo	Tempo da Simulação Distribuída		Tempo da Simulação Seqüencial
	<i>Lookahead</i> estático	<i>Lookahead</i> dinâmico	
Modelo 1	229.7	143.4	1.7
Modelo 2 (configuração 1)	149.0	60.3	0.7
Modelo 2 (configuração 2)	124.1	43.8	0.7
Modelo 3 (configuração 1)	94.5	41.6	0.9
Modelo 3 (configuração 2)	87.8	54.0	0.9
Modelo 4 (configuração 1A)	571.6	447.9	529.2
Modelo 4 (configuração 1B)	885.7	750.7	856.1
Modelo 4 (configuração 1C)	1175.9	1111.0	1242.4

Tabela 6. 8: *Speedup* em relação à média do tempo seqüencial

Modelo	<i>Lookahead</i> Estático	<i>Lookahead</i> Dinâmico	Diferença Percentual
Modelo 1	0.007	0.012	60.2
Modelo 2 (configuração 1)	0.005	0.012	147.1
Modelo 2 (configuração 2)	0.006	0.016	183.3
Modelo 3 (configuração 1)	0.010	0.022	127.2
Modelo 3 (configuração 2)	0.010	0.017	62.6
Modelo 4 (configuração 1A)	0.926	1.182	27.6
Modelo 4 (configuração 1B)	0.967	1.140	18.0
Modelo 4 (configuração 1C)	1.057	1.118	5.8

Antes de prosseguir com a comparação entre os tipos de *lookahead* estático e dinâmico, algumas observações devem ser feitas sobre os dados apresentados:

1. Observa-se que os valores de *speedup* são muito pequenos e desfavoráveis, principalmente para os modelos 1, 2 e 3. Esse fato acontece devido à simplicidade dos modelos e à baixa carga a eles imposta. Como o objetivo deste item é analisar o efeito do *lookahead*, optou-se por cargas mais leves para facilitar a execução dos exemplos. Nos próximos itens, o problema do *speedup* será analisado.

2. O aumento no tempo da simulação sequencial no modelo 4, em relação aos outros modelos, deve-se à carga imposta nesse caso. Observa-se que os modelos 1 e 3 são fechados e possuem apenas 50 tarefas no sistema, o que não impõe a eles uma carga elevada. O modelo 2 é aberto, mas o tempo entre chegadas de novos clientes é elevada – 1 chegada a cada 50 unidades de tempo. No modelo 4 este tempo é menor – varia de 8.5 a 12 unidades de tempo.

Pode-se notar que em todos os casos houve aumento de *speedup*. Essa melhora variou de 5% até 183%, dependendo das características do modelo simulado. Para compreender melhor essa variação no *speedup*, é preciso antes analisar como a utilização do *lookahead* dinâmico influenciou no tempo de processamento, tempo de mensagens nulas e completas.

A tabela 6.9 mostra a diferença, em termos de porcentagem, entre os tempos de execução com *lookahead* estático e dinâmico em relação às mesmas simulações utilizando *lookahead* estático. Como pode ser observado, essa porcentagem variou de 5.5% (no modelo 4 – configuração 1C) a 64.7% (no modelo 2 – configuração 2).

Tabela 6. 9: Diferença percentual do tempo de execução da simulação distribuída com *lookahead* dinâmico em relação à execução da simulação distribuída com *lookahead* estático

Modelo	Diferença Percentual
Modelo 1	-37.6
Modelo 2 (configuração 1)	-59.5
Modelo 2 (configuração 2)	-64.7
Modelo 3 (configuração 1)	-56.0
Modelo 3 (configuração 2)	-38.5
Modelo 4 (configuração 1A)	-21.6
Modelo 4 (configuração 1B)	-15.2
Modelo 4 (configuração 1C)	-5.5

Observando-se essa variação, decidiu-se investigar qual parâmetro influenciava mais nesses resultados. Foram analisados o tempo de processamento, o tempo de transmissão de mensagens nulas e o tempo de processamento de mensagens completas. As tabelas 6.10 a 6.14 apresentam os resultados parciais.

A tabela 6.10 considera o tempo de processamento. É importante diferenciar tempo de execução e tempo de processamento. Neste trabalho, considera-se o “tempo de execução” o tempo total da simulação, enquanto o “tempo de processamento” foi obtido excluindo-se o tempo consumido com a transmissão de mensagens nulas e completas do tempo de execução do modelo.

Na tabela 6.10, nota-se que o percentual do tempo de processamento em relação ao tempo total também é ligeiramente menor utilizando-se o lookahead dinâmico, com exceção do Modelo 4 e suas variações. A diminuição de mensagens nulas pode ter contribuído para essa redução, pois parte do processamento foi dedicada a bloqueios e processamento dessas mensagens.

Tabela 6. 10: Porcentagem do tempo gasto com processamento, em relação ao tempo total de execução

Modelo	Lookahead Estático	Lookahead Dinâmico	Diferença
Modelo 1	5.4	4.4	-1.0
Modelo 2 (configuração 1)	7.5	6.0	-1.4
Modelo 2 (configuração 2)	9.9	8.3	-1.6
Modelo 3 (configuração 1)	12.1	11.4	-0.7
Modelo 3 (configuração 2)	8.8	5.9	-2.8
Modelo 4 (configuração 1A)	70.5	82.5	11.9
Modelo 4 (configuração 1B)	81.0	89.4	8.4
Modelo 4 (configuração 1C)	90.1	92.7	2.6

Nos casos com o modelo 4, a maior parte do tempo de execução é dedicada ao processamento, diferente dos demais casos, onde a transmissão de mensagens respondia pela maior parte do processamento. Neste caso, onde o gargalo encontra-se no processamento, a sobrecarga ocasionada com a introdução do cálculo do *lookahead* dinâmico ocasionou um aumento no tempo de processamento.

Tabela 6. 11: Número de mensagens completas geradas

Modelo	Lookahead Estático	Lookahead Dinâmico	Diferença Percentual
Modelo 1	198874.67	200314.00	0.72
Modelo 2 (configuração 1)	77158.67	77228.87	0.09
Modelo 2 (configuração 2)	38375.73	38537.40	0.42
Modelo 3 (configuração 1)	32986.40	32932.93	-0.16
Modelo 3 (configuração 2)	66674.13	66650.00	-0.04
Modelo 4 (configuração 1A)	58915.07	58887.93	-0.05
Modelo 4 (configuração 1B)	58924.47	58913.27	-0.02
Modelo 4 (configuração 1C)	58871.47	58591.80	-0.48

A diferença entre o número de mensagens completas geradas utilizando-se *lookahead* estático e dinâmico foi pouca, especialmente se comparar a diferença percentual (tabela 6.11). Esse comportamento é esperado, já que as mensagens completas são necessárias à simulação do modelo, independente do método de *lookahead* utilizado.

Tabela 6. 12: Porcentagem do tempo gasto com transmissão de mensagens completas, em relação ao tempo total de execução

Modelo	Lookahead Estático	Lookahead Dinâmico	Diferença
Modelo 1	31.9	51.4	19.6
Modelo 2 (configuração 1)	19.1	47.2	28.1
Modelo 2 (configuração 2)	11.4	32.4	21.0
Modelo 3 (configuração 1)	12.8	29.1	16.3
Modelo 3 (configuração 2)	28.0	45.4	17.5
Modelo 4 (configuração 1A)	4.7	6.0	1.3
Modelo 4 (configuração 1B)	3.0	3.6	0.5
Modelo 4 (configuração 1C)	2.3	2.4	0.1

Já em termos de porcentagem do tempo gasto com transmissão de mensagens completas (tabela 6.12), percebe-se que esse percentual aumentou com o lookahead dinâmico, já que o tempo de transmissão dessas mensagens permaneceu praticamente constante, enquanto o tempo de processamento e transmissão de mensagens nulas diminuiu. Considerando-se apenas o modelo 4 e suas variações, a diferença na porcentagem de tempo gasto com mensagens completas é muito pequena, devido ao impacto menor que as mensagens nulas representam para o sistema.

Tabela 6. 13: Número de mensagens nulas geradas

Modelo	Lookahead Estático	Lookahead Dinâmico	Diferença Percentual
Modelo 1	404711.9	177966.9	-56.0
Modelo 2 (configuração 1)	307523.7	79227.8	-74.2
Modelo 2 (configuração 2)	274467.5	72972.2	-73.4
Modelo 3 (configuração 1)	199383.1	69459.9	-65.2
Modelo 3 (configuração 2)	155988.1	73830.7	-52.7
Modelo 4 (configuração 1A)	317690.5	116048.9	-63.5
Modelo 4 (configuração 1B)	317439.2	118759.4	-62.6
Modelo 4 (configuração 1C)	201214.7	121399.3	-39.7

Tabela 6. 14: Porcentagem do tempo gasto com transmissão de mensagens nulas, em relação ao tempo total de execução

Modelo	Lookahead Estático	Lookahead Dinâmico	Diferença
Modelo 1	62.7	44.2	-18.5
Modelo 2 (configuração 1)	73.5	46.8	-26.7
Modelo 2 (configuração 2)	78.7	59.3	-19.4
Modelo 3 (configuração 1)	75.1	59.5	-15.6
Modelo 3 (configuração 2)	63.3	48.7	-14.6
Modelo 4 (configuração 1A)	24.7	11.5	-13.2
Modelo 4 (configuração 1B)	15.9	7.0	-8.9
Modelo 4 (configuração 1C)	7.6	4.9	-2.8

Ao contrário do que aconteceu com o número de mensagens completas geradas, o número de mensagens nulas diminuiu significativamente com o uso do *lookahead* dinâmico em todos os casos, como pode ser observado na tabela 6.13.

A porcentagem que a transmissão das mensagens nulas representava no tempo total da simulação também diminuiu em todos os casos (tabela 6.14). Essa diminuição foi menor nas configurações 1B e 1C do modelo 4, nas quais o sistema se encontra mais carregado (devido ao menor tempo entre chegadas).

Retomando os dados da tabela 6.8, os modelos que apresentaram melhoras mais significativas no *speedup* foram as duas configurações do modelo 2 e a configuração 1 do modelo 3. Nesses modelos, além da própria melhora causada pela utilização do *lookahead* dinâmico, o particionamento do modelo também influenciou nos resultados, pois favoreceu o cálculo do *lookahead* dinâmico.

Nas duas configurações do modelo 2 (figuras 6.2 e 6.3), sempre havia um processo lógico com um recurso e outro com dois recursos, sendo que para o processo lógico com dois recursos foi considerado sempre o menor tempo de serviço entre os dois recursos. Portanto, o aproveitamento dos tempos de serviço dos recursos para o cálculo do *lookahead* permaneceu o mesmo. Apesar disso, tanto o *speedup* alcançado na configuração 2 do modelo 2 quanto a melhora do *speedup* alcançado nessa configuração foram maiores. Isso se deve também ao fato de a configuração 2 do modelo 2 minimizar a comunicação (o que pode ser constatado nas tabelas 6.11 e 6.13) – reforçando o resultado já observado em (Ulson, 1999).

No caso do modelo 3, houve uma inversão: o *speedup* era ligeiramente melhor na configuração 2 utilizando-se o *lookahead* estático, e passou a ser melhor na configuração 1 quando o *lookahead* dinâmico foi utilizado. Isso explica-se pela melhor exploração do *lookahead* dinâmico na configuração 1, tanto que a redução do número de mensagens nulas foi maior na configuração 1 do que na configuração 2, resultando em melhor *speedup*. Na configuração 1 (figura 6.4), os tempos de serviço dos recursos REC1 e REC2 são considerados no cálculo do *lookahead* dinâmico do processo lógico PL1, enquanto na configuração 2 (figura 6.5) o tempo de serviço desses recursos são descartados no cálculo do *lookahead* do processo lógico PL0, segundo a metodologia para cálculo do *lookahead* dinâmico descrita na seção 5.3.2.

Os testes com o modelo 4 foram os que apresentaram melhoras menos significativas. Apesar de percentualmente apresentarem melhora menor do que os outros modelos, ainda assim a introdução do *lookahead* dinâmico provocou melhora do *speedup*. Considerando-se apenas o valor do *speedup*, e não a melhora percentual, foi o modelo que conseguiu maior

speedup. São os únicos que apresentavam *speedup* razoavelmente bons (ou deixou próximo de 1) mesmo com *lookahead* estático. Também são os únicos que passaram a apresentar *speedup* positivo com *lookahead* dinâmico. Observando-se a variação do tempo entre chegadas (que provocou variações na carga do sistema), as melhoras foram mais significativas quando o modelo se apresentava menos carregado (configuração 1A). Aumentando-se a carga no sistema, a diferença percentual do *speedup* alcançado com *lookahead* dinâmico passou a não ser tão significativa, embora ainda represente aumento no *speedup* alcançado. O impacto do aumento da carga no sistema é estudado na seção 6.4.

Apesar da utilização do *lookahead* dinâmico envolver mais cálculos durante a simulação do que a utilização do *lookahead* estático (em que um número fixo é utilizado), o custo extra no tempo de execução da é sempre menor do que a vantagem obtida pela sua utilização, visto que os tempos de todas as simulações diminuíram.

Os resultados apresentados mostram a viabilidade da utilização do *lookahead* dinâmico em conjunto com o ParSMPLX. Por apresentar resultados indiscutivelmente superiores, os demais testes foram executados utilizando-se o *lookahead* dinâmico.

6.4. Avaliação da Influência do Aumento da Carga

Nestes testes foi avaliada a influência do aumento da carga, provocado pela diminuição do tempo entre chegadas de novos clientes. Foram considerados os Modelos 2 e 4 para os testes, tendo como máquina mestre a *lasdpc06* e escrava a máquina *lasdpc12*. O Modelo 4 foi testado com a configuração 1, utilizando tanto *lookahead* estático quanto dinâmico. A probabilidade de saída utilizada para o modelo 4 foi de 40%.

A tabela 6.15 e a figura 6.9 mostram o tempo de execução dos testes realizados com as duas configurações do modelo 2, utilizando *lookahead* dinâmico. A utilização dos recursos do Modelo 2 é mostrada na tabela 6. 16.

Tabela 6. 15: Modelo 2 – Tempo de execução, em segundos

Tempo entre chegadas	Seqüencial	Configuração 1	Configuração 2
50	4.2	138.9	85.7
20	220.7	268.2	239.8
10	1516.8	1378.4	1456.8
8	2329.9	2144.8	2279.8

Figura 6. 9: Modelo 2 – Tempo de Execução

Percebe-se pela tabela 6.16 que mesmo com a média do tempo entre chegadas igual a 50, a utilização dos recursos já se encontrava alta, em especial os recursos correspondentes aos discos (disco 1 e disco 2), que já se encontravam com utilização próxima de 100%. Apesar disso, o speedup conseguido por essa configuração (tabelas 6.17 e 6.18) ainda é muito baixo. Com a diminuição dessa média de 50 para 20, a utilização do recurso “cpu” aproximou-se do valor máximo e provocou um aumento no tempo de execução da simulação sequencial de 4.1s para 220.7s., o que representou um aumento de 5200%.

Tabela 6. 16: Modelo 2 – Utilização dos recursos

Recurso	Tempo entre chegadas de novos clientes no sistema			
	50	20	10	8
cpu	0.8653	0.9999	1.0000	1.0000
disco 1	0.9990	0.9998	0.9999	0.9999
disco 2	0.9990	0.9998	0.9998	0.9998

Na simulação distribuída, para essa mesma variação de carga, houve um aumento de 95% no tempo de execução da configuração 1 e de 180% na configuração 2. O *speedup* conseguido pela aplicação apresentou melhoras visíveis.

O aumento no tempo da simulação sequencial muito maior do que o aumento provocado nas simulações distribuídas explica-se pelo aumento do processamento exigido pelo aumento da carga no sistema e assim, melhor aproveitamento do paralelismo. Antes desse aumento, as simulações distribuídas dedicavam a maior parte do seu tempo de execução com transmissão de mensagens, o que degradava o desempenho da simulação.

Tabela 6. 17: Modelo 2, configuração 1 – Resultados dos testes

Tempo entre chegadas	<i>Speedup</i>	% tempo com processamento	% tempo com mensagens completas	% tempo com mensagens nulas
50	0.03	17.45	44.48	38.07
20	0.82	54.92	24.91	20.16
10	1.10	91.01	4.85	4.14
8	1.09	94.16	3.13	2.72

Tabela 6. 18: Modelo 2, configuração 2 – Resultados dos testes

Tempo entre chegadas	<i>Speedup</i>	% tempo com processamento	% tempo com mensagens completas	% tempo com mensagens nulas
50	0.05	18.40	36.35	45.25
20	0.92	69.28	13.94	16.78
10	1.04	94.87	2.30	2.83
8	1.02	96.70	1.47	1.83

Tabela 6. 19: Modelo 2 – Número de mensagens completas e nulas geradas

Tempo entre chegadas	Configuração 1		Configuração 2	
	Mensagens completas	Mensagens nulas	Mensagens completas	Mensagens nulas
50	134906	118831	68299	87514
20	145882	121524	72994	90428
10	146081	128102	73104	92692
8	146541	130868	73222	93641

Observando o número de mensagens (tanto nulas como completas) geradas por cada configuração do modelo 2 (tabela 6.19), percebe-se que, para qualquer valor de tempo entre chegadas, a comunicação é sempre maior na configuração 1. Apesar da configuração 2 minimizar a comunicação, essa configuração não foi a melhor em todos os casos. A partir de certo ponto houve uma inversão no speedup conseguido por cada configuração, como pode ser visto na figura 6.10.

Figura 6. 10: Modelo 2 – Variação do *speedup* com o aumento da carga

O *speedup* máximo registrado pelos testes ocorreu com a média do tempo entre chegadas de valor 10. A partir disso, não foi registrado aumento do *speedup* apesar do aumento da carga, mas sim uma ligeira queda.

Resultados similares ao do modelo 2 foram obtidos para o modelo 4, como apresentado nas próximas tabelas. A tabela 6.20 apresenta o tempo de execução para cargas variando de 1 chegada a cada 20 unidades de tempo até 1 chegada a cada 7 unidades de tempo.

Tabela 6. 20: Modelo 4 – Tempo de execução, em segundos

Tempo entre chegadas	Sequencial	Lookahead Estático	Lookahead Dinâmico
20	123.7	259.2	212.1
15	312.7	340.7	317.3
12.5	529.2	571.6	447.9
10.0	856.1	885.7	750.7
8.5	1242.4	1175.9	1111.0
7	1934.4	1787.6	1741.6

Observando a tabela 6.21, percebe-se que a utilização dos recursos do Modelo 4 atinge um valor próximo da utilização máxima a partir do tempo entre chegadas 12.5. Com o tempo entre chegadas 15 o recurso “REC1” ainda não atingiu a utilização máxima, da mesma forma como aconteceu com o teste com o modelo 2, com tempo entre chegadas 50. Apesar disso o *speedup* conseguido com esta configuração no modelo 4 já passa de 0.9 nos testes com *lookahead* estático (tabela 6.22) e é próximo a 1 nos testes com *lookahead* dinâmico (tabela 6.23).

Tabela 6. 21: Modelo 4 – Utilização dos recursos

Recurso	Tempo entre chegadas de novos clientes no sistema					
	20	15	12.5	10	8	7
REC1	0.6661	0.8329	0.9666	0.9999	1.0000	1.0000
REC2	0.9932	0.9999	0.9999	0.9999	1.0000	0.9999
REC3	0.9999	0.9999	0.9999	0.9999	1.0000	0.9999
REC4	0.0986	0.0995	0.0983	0.0985	0.0989	0.0993
REC5	0.1981	0.1993	0.1985	0.1978	0.1978	0.1973
REC6	0.2088	0.2088	0.2071	0.2085	0.2080	0.2079
REC7	0.4947	0.4931	0.4960	0.4940	0.4948	0.4953

Analisando os resultados das tabelas 6.22 e 6.23, observa-se que à medida que o sistema se torna mais saturado, percebe-se a melhora do *speedup* tanto na simulação distribuída com *lookahead* estático quanto com *lookahead* dinâmico. Comparada à simulação com *lookahead* estático, a simulação distribuída com *lookahead* dinâmico apresenta um *speedup* melhor. À medida que o sistema foi se saturando, o *speedup* conseguido com as simulações utilizando *lookahead* dinâmico melhorou até certo ponto e depois apresentou uma ligeira queda, como pode-se ver pelos dados da tabela 6.23 e figura 6.9. Esse mesmo comportamento ocorreu com os testes com o modelo 2, visível na figura 6.8. Isso sugere que o aumento da carga provoca melhoras no *speedup* até um certo ápice e, após atingido esse ápice, a taxa de *speedup* alcançado pelo aumento de clientes no sistema tende a se estabilizar.

Tabela 6. 22: Modelo 4 – Resultado dos testes com *lookahead* estático

Tempo entre chegadas	<i>Speedup</i>	% tempo com processamento	% tempo com mensagens completas	% tempo com mensagens nulas
20.0	0.48	55.60	10.36	34.04
15.0	0.92	66.15	7.92	25.93
12.5	0.93	70.55	4.72	24.73
10.0	0.97	81.00	3.05	15.95
8.5	1.06	90.09	2.29	7.61
7.0	1.08	93.48	1.51	5.01

Tabela 6. 23: 4 – Resultado dos testes com *lookahead* dinâmico

Tempo entre chegadas	<i>Speedup</i>	% tempo com processamento	% tempo com mensagens completas	% tempo com mensagens nulas
20.0	0.58	64.83	12.65	22.52
15.0	0.99	75.83	8.52	15.65
12.5	1.18	82.45	6.02	11.53
10.0	1.14	89.37	3.59	10.13
8.5	1.12	92.72	2.42	4.86
7.0	1.11	95.31	1.55	3.14

O valor máximo para o *speedup* é conseguido mais rapidamente utilizando o *lookahead* dinâmico. Nos testes com *lookahead* estático, o *speedup* máximo ocorreu com o menor tempo entre chegadas. O comportamento observado nos testes com *lookahead* dinâmico sugere que os resultados utilizando *lookahead* estático também devem apresentar um ápice de *speedup* seguido por uma estabilização.

Figura 6. 11: Modelo 4 – Variação do *speedup* com o aumento da carga

Apesar do *speedup* alcançado apresentar ligeira queda e estabilização depois de atingido esse ápice, os gráficos das figuras 6.11 e 6.10 mostram que o *speedup* continua acima de 1 e a diferença entre o tempo da simulação sequencial e da simulação distribuída continua aumentando (gráficos das figuras 6.9 e 6.12).

Figura 6. 12: Modelo 4 - Tempo de Execução

Observando os valores de *speedup* alcançados pelos testes com *lookahead* dinâmico (com o modelo 2 e com o modelo 4), nota-se que os valores máximos de *speedup* ocorreram com o tempo entre chegadas 10 (modelo 2) e 12.5 (modelo 4). Considerando-se o tempo de serviço médio dos recursos pertencentes ao processo lógico onde está a entrada de novos clientes ao modelo, tem-se que:

- a. Modelo 2 – configuração 1: processo lógico PL0 composto por um recurso (CPU), cujo tempo de serviço obedece a distribuição $\text{expntl}(10.0)$. Portanto, o tempo de serviço tem média 10.
- b. Modelo 2 – configuração 2: processo lógico PL0 composto por dois recursos: CPU, com tempo de serviço obedecendo à distribuição $\text{expntl}(10.0)$ e Disco 1, com tempo de serviço obedecendo à distribuição $\text{erlang}(30.0, 7.5)$ e probabilidade de escolha 40%. Portanto, o tempo de serviço neste caso tem em média um valor 11:

$$\frac{\text{expntl}(10.0) + \text{erlang}(30.0, 7.5) * 0.4}{2} = 11$$

- c. Modelo 4 – configuração 1: processo lógico PL0 composto por dois recursos em série: REC1, com tempo de serviço obedecendo à distribuição $\text{expntl}(10.0)$ e REC2, com tempo de serviço $\text{expntl}(15)$. A média do tempo de serviço desse processo lógico fica em 12.5:

$$\frac{\text{expntl}(10.0) + \text{expntl}(15.0)}{2} = 12.5$$

Esses resultados podem indicar uma relação entre o tempo de serviço do processo lógico que compreende a entrada do modelo e o tempo entre chegadas com o qual pode se conseguir o *speedup* máximo para o modelo ao utilizar *lookahead* dinâmico. Outros testes realizados também levaram a esta conclusão.

6.5. Influência da Granulosidade

Além da influência do aumento da carga devido ao aumento de clientes no sistema, foram feitos testes para avaliar o impacto da inserção de laços de granulosidade na simulação do modelo. O laço de granulosidade é um laço do tipo *for* sem corpo de execução, com um número fixo de voltas, chamado sempre que uma tarefa consegue reservar determinado recurso. Esse laço representa uma simulação que envolveria cálculos mais complexos durante

a utilização do recurso, necessitando de mais processamento, sem que isso interfira no fluxo de clientes no sistema.

Foram avaliados os comportamentos de dois modelos: o Modelo 3, com suas três configurações, utilizando a parametrização descrita na seção 6.2.2; e o Modelo 5, com 40% de probabilidade de saída de PL0.

Figura 6. 13: Modelo 3 - Tempo de execução

A figura 6.13 mostra a variação das médias dos tempos de execução das simulações com o Modelo 3. Esse é um modelo que inicialmente apresentava *speedup* muito baixo. Para o teste, a população de clientes no sistema foi sempre a mesma, e o único parâmetro variável foi o valor da granulosidade inserida. A inserção do laço de granulosidade e seu aumento gradual fez com que a média de todos os tempos de execução aumentassem praticamente de forma linear, porém mais acentuada na simulação sequencial.

A tabela 6.24 e a figura 6.14 mostram o *speedup* alcançado por cada configuração. À medida que os laços de granulosidade vão aumentando de valor, a curva apresentada pelo aumento do *speedup* acompanha de forma mais ou menos semelhante à do aumento do tempo de execução sequencial.

Tabela 6. 24: Modelo 3 – *Speedup*

Configuração	sem laço de granulosidade	laço de 25000	laço de 50000	laço de 75000	laço de 100000
1	0.02	0.62	1.05	1.24	1.53
2	0.01	0.51	0.94	1.27	1.57
3	0.02	0.63	1.07	1.34	1.54

Figura 6. 14: Modelo 3 – *Speedup*

Os valores de *speedup* apresentados pelas configurações 1 e 3, na maioria dos casos, são muito próximos entre si e superiores em relação à configuração 2. O particionamento das configurações 1 e 3 foi feito de forma a tentar minimizar a comunicação, e a probabilidade de troca de mensagens entre os processos lógicos que compõem as configurações 1 e 3 são semelhantes. O balanceamento quanto ao número de recursos, porém é diferente: a configuração 3 possui três recursos em PL0 e quatro em PL1, enquanto a configuração 1 possui apenas dois recursos em PL0 e cinco em PL1. Essa diferença no balanceamento não provocou grandes alterações no *speedup*, embora o desempenho da configuração 3 tenha sido ligeiramente melhor.

A configuração 2 apresenta maior grau de comunicação entre os processos lógicos, e isso é evidenciado pelos dados nas tabelas 6.25, 6.26 e 6.27. Apenas o percentual do tempo gasto na transmissão de mensagens nulas é menor na configuração 2 em relação às demais.

Esse fato ocorre justamente devido ao maior grau de comunicação: por trocarem mais mensagens completas, a necessidade de mensagens nulas devido a bloqueios é menor.

À medida que a granulosidade aumenta, o desempenho das três configurações tende a ficar semelhante, pois a quantidade da comunicação, que não muda, passa a representar pouco diante da quantidade de tempo gasta com processamento. O aumento da granulosidade, porém, não causou o mesmo comportamento que aumento da carga no sistema causou. O *speedup* é sempre crescente, não ocorre um pico de *speedup* seguido de estabilização.

As tabelas 6.25 a 6.27 apresentam a contribuição do tempo de processamento e de comunicação (de mensagens completas e mensagens nulas) no tempo total de execução.

Tabela 6. 25: Modelo 3 – Porcentagem do tempo gasto com processamento, em relação ao tempo total de execução

Configuração	sem laço de granulosidade	25000	50000	75000	100000
1	20.08	29.96	39.74	51.45	54.38
2	18.35	20.36	24.85	31.53	35.70
3	20.77	28.16	37.23	46.21	52.67

Tabela 6. 26: Modelo 3 – Porcentagem do tempo gasto com transmissão de mensagens completas, em relação ao tempo total de execução

Configuração	sem laço de granulosidade	25000	50000	75000	100000
1	26.53	23.24	20.14	16.24	15.25
2	39.30	38.69	36.55	28.56	31.63
3	25.82	23.48	20.51	17.76	15.61

Tabela 6. 27: Modelo 3 – Porcentagem do tempo gasto com transmissão de mensagens nulas, em relação ao tempo total de execução

Configuração	sem laço de granulosidade	25000	50000	75000	100000
1	53.39	46.79	40.12	32.31	30.37
2	42.35	40.96	38.61	29.49	32.67
3	53.41	48.36	42.26	36.04	31.72

A figura 6.15 mostra o tempo de execução das simulações com o Modelo 5. Esse modelo já apresentava um *speedup* razoável, mesmo sem laços de granulosidade. Sem o laço de granulosidade, o tempo da simulação distribuída era bem próximo da simulação sequencial. A inserção de laços de granulosidade em ambos evidenciou a diferença entre sequencial e distribuída.

Figura 6. 15: Modelo 5 - Tempo de execução

Figura 6. 16: Modelo 5 - Speedup

Aumentando-se a carga de clientes no sistema, a vantagem no *speedup* causada pela inserção da granulosidade passa a ser menor, como pode ser visto na figura 6.16 e na tabela

6.28. A porcentagem de tempo gasta com processamento era alta mesmo sem o laço de granulosidade, e a inserção de mais processamento não provocou tanta diferença no aproveitamento do paralelismo, embora ainda haja bastante diferença no tempo de processamento sequencial e distribuído. Talvez isso possa ser um indicativo de que o modelo deva ser particionado em mais processos lógicos, aproveitando o paralelismo para distribuir melhor o alto grau de processamento.

Tabela 6. 28: Modelo 5 – *Speedup*

Tempo entre chegadas	sem laço de granulosidade	com granulosidade 50000
12.5	0.98	1.31
10.0	1.01	1.27
8.5	1.03	1.16

As tabelas 6.29 a 6.31 apresentam a contribuição do tempo de processamento e de comunicação (de mensagens completas e mensagens nulas) no tempo total de execução.

Tabela 6. 29: Modelo 5 - Porcentagem de tempo gasto com processamento, em relação ao tempo total de execução

Tempo entre chegadas	sem laço de granulosidade	com granulosidade 50000
12.5	83.16	85.31
10.0	88.13	89.18
8.5	91.23	91.82

Tabela 6. 30: Modelo 5 - Porcentagem de tempo gasto com transmissão de mensagens completas, em relação ao tempo total de execução

Tempo entre chegadas	sem laço de granulosidade	com granulosidade 50000
12.5	1.69	1.48
10	1.17	1.06
8.5	0.85	0.79

Tabela 6. 31: Modelo 5 - Porcentagem de tempo gasto com transmissão de mensagens nulas, em relação ao tempo total de execução

Tempo entre chegadas	sem laço de granulosidade	com granulosidade 50000
12.5	15.15	13.21
10	10.70	9.76
8.5	7.92	7.38

6.6. Avaliação da Influência do Tamanho do Modelo e Balanceamento entre Processos Lógicos

Os testes apresentados nesta seção visam avaliar a diferença do *speedup* conseguido dependendo do tamanho do modelo sendo simulado. Também foi avaliada a influência da diferença de balanceamento entre processos lógicos. Foram utilizados o Modelo 4 e o Modelo 5, ambos parametrizados conforme a descrição da seção 6.2.2 e executadas tendo como máquina mestre a *lasdpc06* e escrava a *lasdpc12*.

O Modelo 4 foi dividido em dois processos lógicos, como mostrado na figura 6.6. Já o Modelo 5 possui uma cópia do Modelo 4 inteiro em cada processo lógico (figura 6.7). A ligação entre essas duas cópias é feita pela saída do modelo na primeira cópia (PL0), que serve como entrada para a segunda cópia (PL1). Assim, o processo lógico PL1 do Modelo 5 não possui geração de novos clientes, apenas os recebe de PL0.

Além de variar o tempo com que novos clientes chegam ao sistema, foram testados dois valores para a probabilidade de saída do Modelo 4 (que, no Modelo 5 corresponde à probabilidade de escolha de PL1): 40% e 80%. Os tempos de execução dos testes estão descritos na tabela 6.32 e a utilização dos recursos, na tabela 6.33. Para o Modelo 4, devem ser considerados apenas as utilizações dos recursos REC1 a REC7.

Tabela 6. 32: Média dos tempos de execução, em segundos

Tempo entre chegadas	Modelo 4		Modelo 5		probabilidade de saída em PL0
	Distribuída	Seqüencial	Distribuída	Seqüencial	
15	283.7	312.7	451.7	400.4	40%
	153.4	87.4	364.7	322.6	80%
12.5	447.9	529.2	656.3	645.6	40%
	327.4	272.4	570.0	543.8	80%
10	750.7	856.1	951.5	958.6	40%
	633.7	619.3	887.5	1303.2	80%
8.5	1111.0	1242.4	1306.1	1350.4	40%
	995.8	1031.6	1264.7	1284.4	80%

Tabela 6. 33: Utilização dos recursos

Recurso	Tempo entre chegadas				probabilidade de saída de PL0
	15.0	12.5	10.0	8.5	
REC1	0.8332	0.9666	0.9999	1.0000	40%
	0.8315	0.9631	0.9999	1.0000	80%
REC2	0.9998	0.9999	0.9999	1.0000	40%
	0.9999	0.9999	0.9999	1.0000	80%
REC3	1.0000	0.9999	0.9999	1.0000	40%
	0.9995	0.9995	0.9996	0.9994	80%
REC4	0.0991	0.0983	0.0985	0.0990	40%
	0.0991	0.0989	0.0980	0.0978	80%
REC5	0.1979	0.1985	0.1978	0.1978	40%
	0.1986	0.1980	0.1982	0.1984	80%
REC6	0.2078	0.2071	0.2085	0.2080	40%
	0.2072	0.2079	0.2085	0.2079	80%
REC7	0.4949	0.4960	0.4940	0.4948	40%
	0.4939	0.4942	0.4946	0.4951	80%
REC8	0.2354	0.4080	0.4081	0.4081	40%
	0.6490	0.6508	0.6493	0.6498	80%
REC9	0.6120	0.6121	0.6114	0.6107	40%
	0.9748	0.9713	0.9739	0.9727	80%
REC10	0.9997	0.9997	0.9997	0.9998	40%
	0.9998	0.9999	0.9999	0.9998	80%
REC11	0.0985	0.0988	0.0981	0.0992	40%
	0.0986	0.0983	0.0988	0.0986	80%
REC12	0.1990	0.1971	0.1986	0.1978	40%
	0.1985	0.2000	0.1982	0.1991	80%

REC13	0.2085	0.2079	0.2083	0.2075	40%
	0.2063	0.2086	0.2069	0.2080	80%
REC14	0.4940	0.4954	0.4950	0.4957	40%
	0.4962	0.4934	0.4951	0.4941	80%

A tabela 6.34 mostra o *speedup* conseguido por cada modelo nos cenários testados. Nos testes com probabilidade de saída 40%, nota-se que o *speedup* alcançado pelo Modelo 5 foi menor em todos os casos, contrariando as expectativas de *speedup* melhor com modelos maiores e com maior nível de granulosidade. A granulosidade, neste caso, refere-se à quantidade de recursos compreendida em cada processo lógico, diferente dos testes da seção 6.5, na qual a granulosidade foi inserida por meio de laço de processamento.

Tabela 6. 34: *Speedup* em relação à média do tempo sequencial

Tempo entre chegadas	Probabilidade de saída 40%		Probabilidade de saída 80%	
	Modelo 4	Modelo 5	Modelo 4	Modelo 5
15.0	1.10	0.89	0.57	0.88
12.5	1.18	0.98	0.83	0.95
10.0	1.14	1.01	0.98	1.47
8.5	1.12	1.03	1.04	1.02

Analisando a utilização dos recursos do Modelo 5 na tabela 6.33, percebe-se que a utilização dos recursos do processo lógico REC8 e REC9 estão bem abaixo da utilização dos recursos REC1 e REC2 (seus correspondentes no processo lógico PL1).

Considerando apenas a utilização dos recursos que fazem parte do Modelo 4, os dois recursos que servem de entrada para cada processo lógico do Modelo 4, REC1 (em PL0) e REC3 (em PL1), estão com utilização máxima em todos os casos. Já no Modelo 5, apenas o recurso que serve de entrada para o processo lógico PL0, REC1, encontra-se com a utilização máxima. O recurso REC8, que serve de entrada para o processo lógico PL1 encontra-se com a utilização baixa. No Modelo 5, essa diferença acontece devido à probabilidade de se escolher o processo lógico PL1, que é baixa.

Alterando a probabilidade de saída em PL0, de 40% para 80%, os recursos REC8 e REC9 passam a ser melhor utilizados e o desempenho da simulação com o Modelo 5 melhora. Da mesma forma que os testes observados na seção 6.4, o *speedup* do Modelo 5 (com probabilidade de saída de PL0 80%) atinge um valor máximo e após isso segue-se um decréscimo do *speedup*, apesar do aumento da carga.

Tabela 6. 35: Porcentagem de tempo gasto com processamento em relação ao tempo de execução total

Tempo entre chegadas	Probabilidade de saída 40%		Probabilidade de saída 80%	
	Modelo 4	Modelo 5	Modelo 4	Modelo 5
15.0	72.9	75.6	57.1	63.2
12.5	82.5	83.2	79.4	75.7
10.0	89.4	88.1	89.3	84.0
8.5	92.7	91.2	93.1	88.6

Tabela 6. 36: Porcentagem de tempo gasto com transmissão de mensagens completas em relação ao tempo de execução total

Tempo entre chegadas	Probabilidade de saída 40%		Probabilidade de saída 80%	
	Modelo 4	Modelo 5	Modelo 4	Modelo 5
15.0	9.4	2.5	10.3	6.1
12.5	6.0	1.7	4.9	3.9
10.0	3.6	1.2	2.5	2.5
8.5	2.4	0.8	1.6	1.8

Em termos de porcentagem, o tempo gasto com processamento de mensagens completas diminuiu, como pode ser visto na tabela 6.36, e isso é bem evidenciado nos testes com probabilidade de saída de PL0 40%. Este comportamento era esperado, visto que a comunicação entre os processos lógicos é bem menor no Modelo 5 do que no Modelo 4. No Modelo 5, apenas o processo PL0 envia mensagens para o processo PL1, enquanto no Modelo 4, a comunicação é bilateral. Essa diferença entre o percentual de tempo gasto com processamento de mensagens nulas vai diminuindo à medida que o sistema se torna mais carregado (mais visível quando a probabilidade de saída de PL0 é de 80%). Isso porque o processamento dos modelos torna-se muito maior com o sistema carregado, como se vê na tabela 6.32, e impacto do tempo gasto com transmissão de mensagens diminui.

A porcentagem de tempo gasto com transmissão de mensagens nulas foi maior para o Modelo 5 em todos os casos (tabela 6.37), exceto no caso onde o sistema está menos carregado, com probabilidade de saída de PL0 80%. Esse aumento na porcentagem de mensagens nulas é justificado pela diferença no cálculo do *lookahead* dinâmico nos modelos.

No Modelo 5, em cada processo lógico são considerados apenas o tempo de serviço de dois dos sete recursos: REC1 e REC2 em PL0, e REC8 e REC9 em PL1. Como as execuções dos recursos REC3 e REC10 envolvem uma escolha probabilística, esses recursos e os que se seguem não são considerados para o cálculo do *lookahead* dinâmico. Já no Modelo 4, todos os tempos de serviço dos recursos em cada processo lógico são considerados, pois a divisão do modelo favoreceu o cálculo do *lookahead* dinâmico.

Tabela 6. 37: Porcentagem de tempo gasto com transmissão de mensagens nulas em relação ao tempo de execução total

Tempo entre chegadas	Probabilidade de saída 40 %		Probabilidade de saída 80 %	
	Modelo 4	Modelo 5	Modelo 4	Modelo 5
15.0	17.6	22.0	32.6	30.7
12.5	11.5	15.2	15.7	20.4
10.0	7.0	10.7	8.2	13.5
8.5	4.9	7.9	5.3	9.6

Os resultados vistos nesta seção mostram que o balanceamento dos processos lógicos em termos de exigência de processamento tem influência fundamental sobre o desempenho da simulação distribuída utilizando ParSMPLX. Apesar do Modelo 5 estar balanceado em termos de número de recursos em cada processo lógico, não estava balanceado em termos de exigência de processamento quando a probabilidade de saída de *PL0* era de 40%. Isso levou a *speedup* baixo, quando comparado ao desempenho obtido pelo Modelo 4, menor. Equilibrando-se o balanceamento quanto à exigência de processamento, o *speedup* aumentou e passou a ser melhor com o modelo maior. Isso leva à conclusão de que o impacto causado pela escolha do balanceamento dos processos lógicos em relação à exigência de processamento é maior do que o causado pelo balanceamento do número de recursos em cada processo lógico.

6.7. Diretrizes para o Módulo Avaliador

O módulo avaliador do ASDA deverá auxiliar o usuário, validando os dados informados pelo usuário e ajudando em tomadas de decisões, quando solicitado. Está dividido em três níveis, sendo que o nível envolvido com o trabalho desenvolvido é o nível 3, no qual o ambiente auxiliará o usuário realizando o particionamento do modelo em processos lógicos.

Combinando-se os resultados observados nos testes feitos com o ParSMPLX com os resultados descritos em (Dotta, 2001; Spolon, 2001; Ulson, 1999), pode-se estabelecer as seguintes diretrizes para alimentar o módulo avaliador:

- ✓ Se optar pela utilização do *lookahead* dinâmico, o particionamento do modelo deve ser feito de modo a maximizar o valor obtido pelo cálculo do *lookahead* dinâmico. Para isso, deve-se procurar colocar recursos em série no mesmo

processo lógico e diminuir a ocorrência de ramificações (saída para outros processos lógicos ou entrada de novos clientes) no meio do processo lógico.

- ✓ Em modelos que apresentem realimentação, é importante que esse laço de realimentação permaneça no mesmo processo lógico, para minimizar a comunicação e a transmissão de mensagens nulas.
- ✓ Nas simulações contendo laços de granulosidade visando aumentar a granulosidade dos processos lógicos, deve-se procurar balancear o número de recursos em cada processo lógico.
- ✓ Se for possível detectar que um processo lógico possa estar mais carregado em termos de exigência de processamento do que os demais processos lógicos (ou seja, a utilização dos seus recursos pode apresentar valor bem acima do que pode ser apresentado pelos demais), deve-se procurar balancear a utilização dos recursos nos processos lógicos. Para isso, é preciso verificar a frequência com que novos clientes chegam ao processo lógico *versus* tempo de serviço dos recursos.
- ✓ A granulosidade escolhida para o modelo deve levar em consideração a plataforma de comunicação utilizada. Plataformas com velocidade de comunicação mais rápida (como arquiteturas paralelas) pode-se utilizar granulosidade mais fina. Sistemas distribuídos, que apresentam velocidade de comunicação mais baixa, devem ser particionados de forma a apresentar granulosidade mais grossa.

Ainda no nível 3 do módulo avaliador, para auxiliar na escolha entre os protocolos de simulação distribuída *Time Warp*, CMB (utilizando ParSMPLX) ou troca de protocolos, pode-se considerar como fatores a favor do CMB:

- ✓ Aproveitamento da estrutura dos recursos no cálculo do *lookahead* dinâmico: se houver muitas estruturas em série que possam ser agrupadas nos mesmos processos lógicos, o modelo apresentará bom *lookahead* e provavelmente bom desempenho.
- ✓ Carga do modelo: se a frequência de chegada de novos clientes no processo lógico for próxima da média da utilização dos recursos, este modelo possui grande chance de ter bom speedup utilizando ParSMPLX.
- ✓ Quanto maior a granulosidade dos processos, melhor será o desenvolvimento da simulação distribuída utilizando o protocolo CMB.

- ✓ Modelos grandes, mesmo com granulosidade pequena, podem apresentar bom desempenho com o ParSMPLX, desde que a utilização dos recursos nos processos lógicos estejam balanceados (ou seja, a exigência de processamento esteja balanceada).

6.8. Considerações Finais

Foram apresentados e analisados neste capítulo os resultados de diversos testes, que levaram em consideração alguns dos fatores que podem influir no desempenho da simulação distribuída utilizando o ParSMPLX.

Esses resultados complementam as diretrizes para auxiliar o processo de desenvolvimento de simulações distribuídas conservativas apresentadas em (Ulson, 1999), e serão utilizados no desenvolvimento do nível 3 do Módulo Avaliador do ASDA (Bruschi, 2002).

De modo geral, a análise dos resultados dos testes estudados mostraram que:

1. Uso do *lookahead* dinâmico:
 - ✓ Sempre provoca melhoras no desempenho da simulação, ainda que em diferentes graus.
 - ✓ Maior grau de aumento de *speedup* em modelos pouco carregados (nos quais a utilização dos recursos apresenta-se com valor baixo) e com muita comunicação, principalmente devido à transmissão de mensagens nulas.
 - ✓ Forma como o *lookahead* dinâmico é calculado influi no desempenho, portanto o particionamento deve ser feito de modo a maximizar o aproveitamento dos tempos de serviços dos recursos no cálculo do *lookahead* dinâmico.
2. Aumento na carga do sistema devido à diminuição do tempo entre chegadas de clientes:
 - ✓ Provoca melhoras no desempenho até atingir um pico de *speedup*. Após esse pico, segue-se um decréscimo ou estabilização do *speedup* atingido.

- ✓ Grande melhora no *speedup* para modelos pouco carregados e com muita comunicação.
- ✓ Indicativos de que o pico de *speedup* ocorre quando o tempo entre chegadas de novos clientes é igual à média do tempo de serviço dos recursos do processo lógico no qual se encontra a entrada do modelo.

3. Aumento na granulosidade dos processos:

- ✓ Sempre provoca melhoras no *speedup* da simulação.
- ✓ Melhora é maior em modelos com muita comunicação e pouco processamento.
- ✓ Modelos com muito processamento apresentam melhoras, porém em menor grau.

4. Balanceamento do modelo:

- ✓ Balanceamento do modelo em termos de exigência de processamento mais importante do que o balanceamento em relação ao número de recursos no processo lógico.
- ✓ Balanceamento do modelo em relação ao número de recursos, quando não há alterações na comunicação, provoca pouca melhora no desempenho em comparação ao mesmo modelo cujo particionamento não privilegia o balanceamento.

5. Tamanho do modelo sendo simulado:

- ✓ Influência secundária, quando comparada à influência do balanceamento do modelo em termos de exigência de processamento.
- ✓ Modelos balanceados de tamanho grande, com processos lógicos maiores, tendem a apresentar desempenho melhor, devido ao menor grau de comunicação entre os processos lógicos.

Capítulo 7 – Inclusão do ParSMPLX ao ASDA

7.1. Considerações Iniciais

O objetivo desta dissertação de mestrado é tornar possível a inclusão do ParSMPL ao ambiente de simulação distribuída automático – ASDA. Nesse sentido, o código do ParSMPL foi adaptado e aprimorado para viabilizar sua utilização no referido ambiente, conforme descrito no capítulo 5. Um próximo passo realizado foi a avaliação da utilização do ParSMPLX. O capítulo 6 mostrou a viabilidade e em quais situações a utilização do ParSMPLX é adequada.

Assim, tem-se toda a base necessária para a inclusão do ParSMPLX ao ASDA. Isto é, a partir do momento em que o programa de simulação estiver pronto, já pode ser executado no ASDA. No entanto, para se ter a possibilidade de utilizar todos os recursos previstos no ASDA, tem-se a necessidade de gerar o programa de simulação de forma automática, a partir de um modelo definido pelo usuário. Para uma abordagem completamente automatizada, deve-se ainda desenvolver o módulo avaliador, que ajudará na decisão da escolha do protocolo e na divisão dos processos.

O módulo avaliador é motivo de uma outra dissertação de mestrado que, utilizando os resultados apresentados no capítulo 6 e em outros trabalhos, auxiliará o usuário nas decisões.

A geração do programa de simulação também não faz parte do escopo deste trabalho. Entretanto, durante o processo de adaptação, aprimoramento e avaliação do ParSMPL – que foi feito visando sua inclusão no ASDA – foi necessário definir como seria esta inclusão. Assim, este capítulo apresenta o estudo realizado, onde são definidas as alterações necessárias.

A seção 7.2 apresenta a estrutura geral do módulo gerador. A seção 7.3 descreve o gabarito a ser utilizado, visando a geração automática do código. A seção 7.4 detalha as

alterações necessárias em um gerador para simulação seqüencial, visando adaptá-lo para gerar programas de simulação distribuída.

7.2. Estrutura do Módulo Gerador

A geração de programas de simulação distribuída pelo módulo gerador do ASDA envolve três artefatos: um gabarito, um gerador de aplicações e os dados da especificação do modelo. Desses, dois devem ser específicos ao ParSMPLX: o gabarito e o gerador de aplicações. A figura 7.1 (Bruschi, 2002) mostra a estrutura desses artefatos envolvidos na geração de programa de simulação.

Figura 7. 1: Estrutura de arquivos no módulo Gerador do ASDA

Os dados da *especificação do modelo* são informados pelo usuário, e essas informações serão passadas ao gerador por intermédio da interface gráfica. O desenvolvimento dessa interface é o objetivo de um outro trabalho de mestrado (Aiza, 2003) e oferecerá ao usuário a possibilidade de definir seu modelo utilizando *queuing statecharts* (Francês, 2001). Uma vez definido e parametrizado o modelo, o sistema irá gerar o arquivo de especificação do modelo.

O *gabarito.dat* é um arquivo texto que contém trechos de código na linguagem alvo, juntamente com comandos que indicam ao gerador qual porção de código deve ser gerada.

Cada um dos comandos definidos no gabarito tem a função de gerar um trecho de código, que depende dos dados obtidos da especificação do modelo. O gabarito para gerar um programa em ParSMPLX e linguagem C é apresentado na seção 7.3.

A função do *programa gerador* é preencher o programa de simulação de acordo com o gabarito considerado, combinando seus comandos com os dados existentes na especificação do modelo. Atuando desta forma, o gerador irá produzir os programas que constituem os processos lógicos da simulação distribuída. O programa gerador do ASDA terá como base a lógica do gerador de aplicações existente para o ambiente de simulação ASiA (Spolon, 1994). Como este produz programas de simulação utilizando SMPL, que é a extensão na qual foram baseados ParSMPL e ParSMPLX, a reutilização do código existente – ou pelo menos da lógica envolvida – é justificada pela redução do esforço inerente ao desenvolvimento do novo gerador. As alterações necessárias para torná-lo apto a gerar programas de simulação distribuída são vistas na seção 7.4.

7.3. Gabarito para Geração de Código ParSMPLX

Em (Bruschi, 2002) é apresentada uma versão inicial do gabarito para ParSMPL, elaborado a partir do gabarito existente para o ASiA. Como a utilização do ParSMPLX em testes possibilitou um melhor conhecimento de suas particularidades, o gabarito apresentado foi aperfeiçoado. O novo gabarito pode ser visto na figura 7.2.

Neste gabarito, as linhas iniciadas com o símbolo de porcentagem (%) são os comandos que indicam a porção de código que deve ser gerada. As demais linhas são trechos em linguagem C, e devem ser simplesmente copiadas para o programa de simulação sendo gerado. Mantendo a notação do gabarito elaborado para o ambiente ASiA (Spolon, 1994), os comandos iniciados com número denotam trechos obrigatórios ao programa de simulação. Exemplos desses comandos são os presentes nas linhas 12 e 21. Os comandos que iniciam com letras, como os representados nas linhas 11, 15 e 16, indicam que os trechos por eles gerados são opcionais, e podem não estar presentes no programa sendo gerado.

```

1  /* ----- */
2  #include "parsmpl.h"
3
4  int main(void)
5  {
6  /* Fase de Definição */
7  #define tempo num_max
8
9  int Event = 1, Aleatorio, Customer;
10 int tid_processo pai, valor_semente = 1;
11 #Agera_contadores_loop
12 #define variaveis
13
14 /* Fase de Preparação */
15 %Bpai registra_pvm
16 %Cpai prepara_e_envia_tabelas
17 %Descravo recebe_tabela_tid
18
19 #define smpl
20 /* Inicia as variaveis do ParSMPL */
21 %3inicia_variaveis_parsmpl
22 %2define_lookahead_estatico
23
24 /* cria e da nome as facilidades */
25 #4define_server
26
27 /* escalona a chegada do primeiro cliente */
28 %Fpai_escalona_primeiro_evento
29
30 %Gpre_computa_lookahead_dinamico
31
32 /* Fase de Implementação */
33 %5gera_loop
34 {
35     %Hwarm_up
36     %Icalcula_lookahead_dinamico
37     %6gera_cause
38     %7gera_switch
39     {
40         %8gera_eventos
41     }
42 }
43 %Jpai_envia_comando_parada
44 /* gera o relatorio da simulacao */
45 %9gera_relatorio_padrao
46 %Kgera_relatorio_estatisticas_max_min
47 %Lgera_relatorio_estatisticas_fila_vazia
48
49 parsmpl_pvm_exit();
50 exit(0);
51 }
52 /* ----- */

```

Figura 7. 2: Gabarito para geração de código utilizando ParSMPLX

As linhas que aparecem em negrito na figura 7.2 são comandos incluídos para gerar trechos de código específicos para simulação distribuída utilizando ParSMPLX. Os que representam comandos para geração de código específico para o programa mestre iniciam com a palavra *pai*, enquanto as linhas iniciadas com a palavra *escravo* indicam trechos específicos para programas escravos. Assim, as linhas 15 e 16, por exemplo, gerarão código apenas se o programa sendo gerado for mestre. De modo análogo, as linhas 17 e 18 só terão efeito ao gerar código para programas escravos. Os comandos sem indicação de *pai* ou *escravo*, como os comandos indicados nas linhas 30 e 36, são comuns tanto ao programa mestre quanto ao escravo.

Os trechos de código que devem ser gerados pelos novos comandos encontram-se descritos na seção 7.4.3, que trata das novas funcionalidades a serem incluídas no gerador de aplicações existente para o ASiA.

7.4. Alterações no Gerador de Aplicação do ASiA para Adaptação ao ASDA

A estratégia para criação de um gerador de código de simulação distribuída utilizando o ParSMPLX é utilizar como base o gerador de aplicações existente para o ambiente ASiA (Spolon, 1994) e adaptá-lo. Como este gerador foi projetado para gerar código utilizando SMPL e há semelhanças entre a estrutura de programas utilizando SMPL e ParSMPLX, a reutilização do código existente contribuirá para diminuir o esforço de implementação. Entretanto, o ParSMPLX apresenta características próprias, por se tratar de programação paralela (como a preparação dos dados e a criação de processos escravos). Assim sendo, novas funcionalidades são necessárias ao gerador de aplicações.

Um primeiro ponto a ser analisado é que no ASiA, o gerador é uma função chamada pela interface gráfica. Para o ASDA, o gerador poderia continuar como uma função (ou método) chamado pela interface gráfica. Neste caso, seria necessário reescrever o gerador na linguagem escolhida para a interface gráfica (Kylix, Java, C ou C++), incluindo as modificações necessárias para a geração de código para simulação distribuída. Apesar deste trabalho em reescrevê-lo, a interação com a interface gráfica (por exemplo, para transferência de dados da especificação do modelo e tratamento de erros) é facilitada.

Outra possibilidade seria fazer um programa executável em C, independente da interface gráfica, que possa ser chamado pela interface gráfica. Com esta abordagem, boa

parte do código poderia ser reaproveitada, mas o tratamento de erros, por exemplo, teria que ser feito via valor de retorno do programa.

A descrição das alterações necessárias aos algoritmos do gerador do ASiA para torná-lo apto a gerar código para simulação distribuída com ParSMPLX e assim ser incluído ao ASDA está dividida em três partes: as alterações gerais, as alterações que precisam ser feitas pontualmente em algumas funções já existentes e as novas funções. Os dois primeiros tipos de alterações referem-se aos comandos herdados do gabarito para simulação sequencial do ASiA. As novas funções necessárias ao gerador correspondem aos trechos que devem ser gerados pelos novos comandos do gabarito, mostrados em negrito na figura 7.2. Os algoritmos descritos nas próximas seções foram escritos em linguagem C, considerando as funções existentes do Gerador de Aplicações para o ASiA (Spolon, 1994).

7.4.1. Alterações gerais no gerador de aplicações do ASiA

As alterações que devem ser feitas em vários pontos do código do gerador consistem, na maior parte dos casos, de nomes de funções que foram modificados e alteração nos tipos de parâmetros passados às funções.

Como essas alterações são pequenas e não representam uma modificação conceitual originada pela utilização da simulação distribuída, elas não serão apresentadas no texto deste capítulo. No apêndice B encontra-se a relação dessas alterações.

7.4.2. Alterações de funções existentes

Nesta seção encontra-se uma relação das alterações em funções específicas do gerador. Em cada caso, é apresentado o nome da função, uma breve descrição de sua funcionalidade, e a alteração necessária. Nos casos em que há modificações no algoritmo da função, o novo algoritmo é apresentado.

Prepara_Estruturas

Prepara as estruturas de dados que serão utilizadas na geração de código. Este passo é executado pelo gerador antes de iniciar a leitura do gabarito.

Alterações

São utilizadas algumas estruturas de dados próprias do sistema operacional Windows e métodos proprietários da Borland. Devem ser trocados considerando-se o ambiente utilização e desenvolvimento utilizado para o ASDA (Linux com Kylix ou Java, por exemplo).

Define_Tempo_Max

Essa função tem como objetivo gerar o comando para limitar a execução da simulação, seja pelo tempo, pelo número de clientes que deixam o sistema, ou ainda pelo número de ciclos do cliente no sistema. É executado pelo comando da linha 7 do gabarito (figura 7.2).

Alterações

Deve ser informada à função, seja por meio de passagem de parâmetros ou via variável global, uma variável *processo_logico* indicando se o processo lógico é escravo ou pai.

No início da função deve ser incluído um teste para verificar se o processo é mestre ou escravo. Se for processo mestre, o código prossegue como o original. Se o processo for escravo, deve ser criada uma variável do tipo inteiro chamada "fim", com valor inicial zero.

Algoritmo

```
void Define_Tempo_Max (int processo_logico)
{
    if (processo_logico == ESCRAVO)
    {
        Copy (" int fim = 0;");      // função existente
        Grava_Buffer();             // função existente
        Carriage Return             // função existente
    }
    else
    {
        //manter o código original da função
    }
}
```

Escalona_Release

A função Escalona_Release implementa a primitiva que irá escalonar o momento da liberação do recurso (*facilidade*). É uma das funções chamadas pelo comando da linha 40 do gabarito (figura 7.2).

Alterações

Deve ser informada à função uma variável *processo_destino*, contendo o número do processo lógico onde fica o recurso a ser escalonado. Essa variável deverá ser utilizada na alteração que deve ser feita na geração do trecho com a função *schedule*. A prioridade para o recurso também deve ser informada.

O método de análise *Batch Means* não está implementado no ParSMPLX. Assim, a porção de código que gera códigos para esse método de análise deve ficar inativa até a sua implementação no ParSMPLX.

Define_Var_Server

Gera as variáveis para definição dos recursos que compõem o modelo a ser implementado. É uma das funções chamadas pelo comando da linha 12 do gabarito (figura 7.2).

Alterações

Esta função não é mais necessária no ParSMPL/ParSMPLX, pois os recursos são identificados pelo nome, não mais por descritores (como era feito inicialmente no SMPL).

Gera_Loop

Gera o comando de repetição do laço da simulação. A simulação pode ser limitada ou pelo tempo, ou pelo número de clientes que passam pelo sistema, ou pelo número de ciclos da simulação, ou pelo método de análise *Batch Means*. Esta função é ativada pelo comando da linha 33 do gabarito (figura 7.2).

Alterações

Deve ser feita uma verificação se o processo sendo codificado é mestre ou escravo. Para isso, uma variável *processo_logico* deve ser passada como parâmetro à função. Se o processo lógico for mestre, a codificação permanece como está. Se for processo escravo,

deve ser gerado um comando diferente como critério de parada da simulação.

Como o método de análise *Batch Means* ainda não está implementado no ParSMPLX, a opção de limitar a simulação via *Batch Means* deve ficar desabilitado até sua implementação.

Algoritmo

```
void Gera_Loop (int processo_logico)
{
    if (processo_logico == ESCRAVO)
    {
        Copy (" while( !fim );");
        Grava_Buffer();
        Carriage Return
    }
    else
    {
        //manter o código original da função
    }
}
```

Gera_Eventos

Esta função gera os eventos que constituem a simulação. Corresponde ao trecho de código gerado pelo comando da linha 40 do gabarito (figura 7.2).

Alterações

Deve-se incluir a geração de um evento extra em cada processo escravo. Para essa modificação, é necessário informar à função uma variável *processo_logico*, indicando se o processo lógico sendo codificado é mestre ou escravo, e definir uma constante de valor alto, que irá representar o evento fim da simulação.

Esse evento receberá o aviso de parada, indicará internamente que a simulação chegou ao fim e enviará a resposta ao mestre.

Além disso, depois de cada chamada à função *Escalona_Release* feita dentro da função *Gera_Eventos*, é necessário chamar uma nova função que deverá fazer a pré-computação do tempo de serviço do recurso, que será utilizado para o cálculo do *lookahead* dinâmico.

Algoritmo

```
// Geração do comando de parada para o escravo:
// Deve ser inserido depois do término do laço
```

```

// while(total_recursos) {...}
if (processo_logico == ESCRAVO)
{
    Gera_Case(EVENTO_FIM);
    // marca o fim da simulação
    Copy("    fim = 1;\n");
    // envia ao pai a confirmação da parada
    Copy("    { int confirma = 1;\n");
    Copy("    pvm_initsend(PvmDataRaw);\n");
    Copy("    pvm_pkint(&confirma,1,1);\n");
    Copy("    pvm_send(tid_processo_pai, 0);\n)\n");
    Grava_Buffer();
    Gera_Break();
}

```

Gera_Relatorio_Final

Gera_Relatorio_Final cria a chamada à função *report()*, do SMPL, que imprime na tela as informações geradas pela simulação. É ativada pelo comando da linha 45 do gabarito (figura 7.2).

Alterações

No ParSMPL/ParSMPLX não está implementada a função *report()*.

Uma vez implementada a função *report()*, a função Gera_Relatório_Final permanecerá inalterada.

Warmup

Esta função gera o comando para *warmup* do sistema. O *warmup* é um período inicial para que o sistema entre em equilíbrio, visando com isso representar o modelo a ser simulado de forma mais realística. Decorrido um tempo pré-estabelecido, todas as estatísticas coletadas até então são descartadas. É uma das funções chamadas pelo comando da linha 35 do gabarito (figura 7.2).

Alterações

Não está implementada a função *reset()* do SMPL. Assim, esta função deve permanecer inativa até a implementação de *reset()*.

Uma vez implementada a função *reset()*, a função Warmup permanece inalterada.

Le_Gabarito

Esta função percorre o arquivo de gabarito, determinando qual a próxima primitiva a ser executada.

Alterações

As alterações devem ser aplicadas a esta função tendo como base o novo gabarito, chamando as funções apropriadas segundo o número ou letra que indica cada primitiva.

7.4.3. Inclusão de novas funcionalidades

Como as funções existentes no gerador presta-se para a geração de código para simulação seqüencial, ainda que se apliquem as modificações discutidas anteriormente, não será suficiente para a geração de código para simulação distribuída, já que esta possui algumas características e funções próprias.

Serão apresentadas nesta seção as novas funcionalidades necessárias ao gerador. Para cada nova função, será apresentada sua finalidade, o comando do gerador ao qual ela corresponde e uma sugestão de algoritmo.

Finalidade

Gerar a parte de código no processo mestre necessária para identificar o processo mestre e criar os processos escravos. Corresponde ao comando descrito na linha 15 do gabarito (figura 7.1).

Será necessário informar à função o número de escravos da simulação (variável `total_escravos`), um vetor contendo os nomes dos processos escravos (com o caminho inteiro) e um vetor contendo os nomes das máquinas onde os processos escravos devem ser criados. Mesmo que o usuário não deseje especificar em qual máquina cada processo lógico deva ser criado, esse vetor deve ser informado com os nomes das máquinas vazios.

Algoritmo

```
Copy(" /* identifica o processo pai */ \n");
Copy(" tid_processo_pai = pvm_mytid(); \n");

for (int i = 0; i < total_escravos; i++)
{
    int maquina_definida = 0;
```

```

    if (length(nome_maquina[i]) > 0)
        maquina_definida = 1;

    Copy(" /* cria o processo escravo */ \n");
    sprintf(comando, "info = pvm_spawn(%s, (char**)0, ", escravo[i]);
    Copy(comando);
    sprintf(comando, "%d, %s, 1, &tabela_tids[%d]);", maquina_definida,
nome_maquina[i], i);
    Copy(comando);
    Copy(" if( info < 1 ) \n { \n");
    Copy("\tprintf(\"Erro ao criar escravo.Codigo de erro:\\n\\n\"); ");
    Copy("\tprintf(\"TID %d \\n\\n\", ");
    sprintf(comando, "tabela_tids[%d]);", i);
    Copy(comando);
    Copy("\t pvm_exit(); \n exit(1); \n");
    }
}

```

Finalidade

Gerar a parte de código necessária para que o processo mestre prepare a tabela de *task IDs* e a envie aos processos filhos. Corresponde ao comando descrito na linha 16 do gabarito (figura 7.1).

Será necessário informar à função o número de escravos da simulação (variável `total_escravos`).

Algoritmo

```

char *comando;
// zerar a tabela de tids (variável do ParSMPL)
Copy(" for (indice = 0; indice < 10; indice++) \n { ");
Copy(" tabela_tids[indice] = 0; \n }");
// enviar aos processos filhos
Copy(" /* Inicia comunicacao com o processo filho \n");
Copy(" * Parâmetro PvmDataRow - utilizando máquinas homogêneas */
\n");
Copy(" pvm_initsend(PvmDataRow); \n");
Copy(" /* Empacota e envia a tabela para o processo filho */ \n");
Copy(" pvm_pkint(tabela_tids,10,1); \n ");
sprintf(comando, " for (int i = 0; i < %d ; i++) \n",
total_escravos);
Copy(comando);
Copy(" pvm_send(tabela_tids[i],0);\n ");

```

Finalidade

Gerar a parte de código necessária nos processos escravos para que seja feita a associação do *task ID* do processo pai e receber dele a tabela de *task IDs*. Corresponde ao

comando descrito na linha 17 do gabarito (figura 7.1).

Algoritmo

```
Copy(" /* Associa o tid do processo pai */ ");
Copy(" tid_processo_pai = pvm_parent(); ");
Copy(" /* Recebe do pai a tabela de tids */ ");
Copy(" pvm_recv(tid_processo_pai,-1); ");
Copy(" pvm_upkint(tabela_tids,10,1); ");
```

Finalidade

Gerar a parte de código responsável por iniciar as variáveis da simulação `clock_canal` e `proc_local`. Corresponde ao comando descrito na linha 21 do gabarito (figura 7.1).

Deverá ser informada à função o número do processo lógico sendo codificado (variável `processo_logico`) e um vetor contendo os números dos processos lógicos com o qual este processo lógico se comunica (variável `processos`).

Algoritmo

```
char *comando;
Copy("\t/* define o numero do processo local */ \n");
sprintf(comando, "\t proc_local = %d; \n", processo_logico);
Copy(comando);
Copy("\t/* inicia os canais de comunicacao utilizados */ \n");
for (int i = 0; i < tamanho_vetor; i++)
{
    sprintf(comando, "clock_canais[%d] = 0.0;\n", processos[i]);
}
Copy(comando);
```

Finalidade

Uma função para inserir a linha que define o valor do *lookahead* estático, quando o usuário optar por utilizar *lookahead* estático na simulação. Corresponde ao comando da linha 22 do gabarito (figura 7.1).

Deve ser informado à função o valor do *lookahead* estático (variável `lookahead_estatico`).

Algoritmo

```
char *comando;
Copy("\t/* define o valor do lookahead (estatico) */ \n");
```

```
sprintf(comando, "\t tempo_processo = %f; \n", processo_logico);
Copy(comando);
```

Finalidade

Será necessário ao novo gerador uma função que gere o comando para calcular o lookahead dinâmico (corresponde ao comando da linha 36 do gabarito da figura 7.1).

Como no ASDA a estrutura dos processos lógicos serão armazenadas em grafos, será preciso fazer uma análise do grafo de cada processo lógico, considerando a metodologia descrita na seção 5.3.1 (ou outras metodologias para cálculo de *lookahead* dinâmico).

A elaboração de um algoritmo para a geração desse código, entretanto, exige um estudo mais aprofundado.

Finalidade

Gerar o código para que o programa de simulação pré compute o tempo de serviço do recurso. Deve ser chamado sempre depois da função *escalona_release*, e é necessário informar de qual é o recurso corrente (variável *recurso*).

Esta função também deve ser chamada ao pré-computar os valores dos recursos antes de entrar no laço de simulação (linha 30 do gabarito da figura 7.1).

Deve ser informado à função o número do recurso que está tendo seu tempo de serviço pré-calculado (variável *numero_recurso*).

Algoritmo

```
char *nome = "";
char *tempo_atend = "";
sprintf(nome, "Server%d = ", numero_recurso);
Copy(nome);
strcat(Buffer, Tabela_Distribuicao[Recurso_Corrente->
distribuicao_servico]);
sprintf(tempo_atend, "Ta%d", numero_recurso);
strcat(Buffer, tempo_atend);
// distribuicoes implementadas pelo SMPL e que necessitam
// de dois parametros: erlang, hyperx, uniform
if (Recurso_Corrente->distribuicao_servico)
{
    strcat(Buffer, ", ");
    strncat(Buffer, Recurso_Corrente->
desvio_padrao_servico, strlen(Recurso_Corrente->
desvio_padrao_servico));
}
```

```
// a distribuicao expntl necessita de apenas um parametro
if (Recurso_Corrente->distribuicao_servico == 4)
{
    strcat(Buffer, ",");
    strncat(Buffer, Recurso_Corrente-
>pto_triangu_servico, strlen(Recurso_Corrente->pto_triangu_servico));
}
strcat(Buffer, " ");
Grava_Buffer();
```

Finalidade

Gerar o trecho de código no processo lógico mestre para que este informe os processos lógicos escravos o fim da simulação. Corresponde ao comando da linha 43 do gabarito.

Deve ser informado à nova função o número de escravos (variável `total_escravos`) e o valor `EVENTO_FIM`, que pode ser uma constante definida no gerador do ASDA.

Algoritmo

```
char *comando;
int buffer = 0;
Copy(" int recebidos = 0; \n");
Copy(" // enviar para todos os escravos \n");
for (int i=1; i <= total_escravos; i++)
{
    sprintf(commando, " schedule (EVENTO_FIM, 0.0, Customer, %d); ",
i);
    Copy(commando);
}
Copy(" // Espera até receber confirmação de todos os escravos
Copy(" while ( recebidos < total_escravos ) \n { ");
Copy(" buffer = pvm_nrecv(-1, -1); \n");
Copy(" if (buffer > 0) // recebeu mensagem de um escravo \n");
Copy("     recebidos++; \n");
Copy(" } ");
```

7.5. Considerações Finais

Foram vistas neste capítulo as alterações necessárias para incluir o ParSMPL ao ambiente de simulação distribuída automático ASDA (Bruschi, 2002), re-aproveitando o trabalho desenvolvido para o ambiente ASiA (Spolon, 1994).

Apesar da reutilização de código ou da lógica envolvida no gerador reduzir os esforços para a criação do novo gerador, ainda existem pontos que devem ser estudados, especialmente

em relação à geração do trecho de código responsável por calcular o valor do *lookahead* dinâmico que será utilizado em cada iteração.

O melhor conhecimento do ParSMPLX decorrente de sua utilização em testes de desempenho possibilitou a elaboração de uma versão aperfeiçoada do gabarito que deve ser utilizado pelo módulo gerador. Com isso, a visão de novas funcionalidades para o gerador também ficaram mais claras e mais fáceis de ser definidas.

Capítulo 8 – Conclusões

8.1. Comentários sobre a Revisão Bibliográfica

A revisão bibliográfica apresentada nos capítulos 2 e 3 apresentaram os conceitos necessários sobre computação paralela e simulação, tanto sequencial como distribuída. O capítulo 4 expôs as ferramentas para auxiliar a elaboração de simulações distribuídas, um resumo de ambientes para simulação, linguagens e extensões para simulação distribuída mais conhecidos. No capítulo 4 também foram apresentados com mais detalhes a extensão ParSMPL (Uison, 1999) e o ambiente para simulação distribuída ASDA (Bruschi et al., 2000; Bruschi, 2002).

Dentre as vantagens oferecidas pelo processamento paralelo, destaca-se o aumento na velocidade de execução, possibilitando que aplicações complexas sejam executadas em tempo consideravelmente reduzido. Entretanto, apesar das vantagens oferecidas pelo paralelismo, o desenvolvimento ou migração de aplicações sequenciais ao paradigma de computação paralela não ocorre de forma direta.

Dos três tipos de ferramentas para suporte à programação concorrente apresentados na seção 2.4, o uso de extensões paralelas para linguagens existentes mostra-se particularmente atrativa: permite elaborar aplicações em diferentes níveis de granulosidade (de grão médio a grosso), e principalmente não exige a aprendizagem de nova linguagem.

Como foi visto no capítulo 3, diversas são as técnicas para avaliação de desempenho, cada uma apresentando um conjunto de peculiaridades, vantagens e desvantagens. A simulação destaca-se por seu baixo custo e flexibilidade. Não exige a existência do sistema a ser avaliado, nem de instrumentos específicos. Como consiste de um programa, pode ser facilmente alterado e adaptado a várias situações, permitindo avaliar e comparar várias alternativas de um projeto. Mostra-se altamente reutilizável, fato enfatizado em seu ciclo de

vida, apresentado na seção 3.3. Pode-se perceber neste ciclo de vida a importância da validação e verificação constante da simulação, para garantir que a representação da simulação condiz com o sistema real.

A simulação distribuída tem sido desenvolvida para atacar a principal desvantagem da simulação: seu tempo de execução, que pode inviabilizar simulações muito complexas ou com grande nível de detalhes. Entretanto, a simulação distribuída apresenta os mesmos problemas encontrados na programação paralela: a necessidade de sincronização, de ferramentas de suporte à programação, e a granulosidade, representada pela partição da simulação em processos lógicos.

Foram apresentados no capítulo 3 os protocolos de sincronização conservativos, otimistas e assíncronos. Para escolher o melhor entre eles é necessário levar em consideração as características da aplicação. Protocolos conservativos são propensos a *deadlock* e são muito dependentes do *lookahead*. O estudo de *lookahead* dinâmico é uma alternativa para buscar melhoras no desempenho dessas simulações. O problema dos protocolos otimistas está no número de *rollbacks* necessários quando um erro de causa e efeito é detectado, e os protocolos assíncronos geram resultados apenas estimados.

As ferramentas para auxiliar o desenvolvimento de simulações distribuídas foram apresentadas no capítulo 4. Assim como na programação paralela, o uso de bibliotecas (extensões de linguagens padrão) é atrativo, pelos mesmos motivos apresentados, além de esconder os detalhes de sincronismo e comunicação a usuários pouco familiarizados com programação paralela. A utilização de ambientes automáticos também é muito atrativa, pois amplia o uso da simulação, possibilitando o uso da simulação distribuída a usuários com poucos conhecimentos sobre simulação ou programação paralela, ao mesmo tempo que facilita e agiliza o processo a usuários mais experientes.

8.2. Conclusões

O trabalho realizado neste mestrado levaram às conclusões que podem ser divididas em três enfoques:

1. Adaptação do ParSMPL ao Linux.
2. Desempenho de uma simulação no ParSMPL.
3. Inclusão do ParSMPL ao ASDA.

Adaptação do ParSMPL ao Linux

A adaptação de uma ferramenta como o ParSMPL ao Linux não é uma tarefa trivial. As principais dificuldades encontradas durante esta fase de adaptação do ParSMPL ocorreram devido à complexidade inerente à depuração de programas paralelos e à falta de ambientes adequados para a depuração de programas paralelos. Aliadas a isso, a falta de conhecimento sobre o funcionamento da extensão SMPL (MacDougall, 1987) e a documentação escassa do ParSMPL contribuíram para o aumento no esforço necessário a essa adaptação.

Desempenho de uma simulação no ParSMPL

Alcançar um bom desempenho em uma simulação distribuída é um grande desafio. A avaliação do protocolo conservativo, implementado no ParSMPL, permitiu algumas conclusões a respeito do desempenho desse protocolo. A avaliação realizada visa, principalmente, determinar em quais situações pode se alcançar bom desempenho com protocolo conservativo.

Durante a realização dessas avaliações, observou-se que o desempenho da simulação distribuída é prejudicado pela imposição do protocolo utilizado em seqüencializar eventos. Para minimizar essa característica foi avaliada a utilização de *lookahead* dinâmico.

A seguir são apresentadas as principais conclusões em relação ao desempenho das simulações distribuídas.

I. Utilização de *lookahead* dinâmico

As simulações com o ParSMPLX em conjunto com *lookahead* dinâmico mostraram-se muito mais eficientes do que as mesmas utilizando *lookahead* estático. A introdução do *lookahead* dinâmico na simulações levou a uma redução considerável do número de mensagens nulas trocadas entre os processos lógicos. Esta redução foi de no mínimo 40%, chegando a 74%, dependendo do modelo. Nos testes feitos com modelos cujos recursos encontravam-se menos utilizados, o impacto foi mais visível, devido à maior influência da comunicação no desempenho.

O modo como é feito o cálculo o *lookahead* dinâmico também tem influência no desempenho, e deve ser considerado no momento da escolha da forma de particionamento do modelo. É importante que os recursos em série permaneçam no mesmo processo lógico e que

se diminua a ocorrência de ramificações no meio do processo lógico, ainda que isso provoque um desbalanceamento em termos de número de recursos em cada processo lógico.

II. Influência do número de clientes no sistema

Outro fator que influenciou bastante o desempenho dos testes realizados foi o aumento de carga no sistema provocado pelo aumento de clientes chegando ao sistema. Modelos pouco carregados, que alcançavam valor de *speedup* 0.03 ou 0.48, por exemplo, chegaram a atingir *speedup* acima de 1. O fato curioso observado nos testes foi a ligeira redução do *speedup* seguida de estabilização, depois de atingido um pico. Há indicativos de que exista uma relação entre o tempo entre chegadas de novos clientes e o tempo de serviço dos recursos que compõem processo lógico onde se encontra a entrada do modelo.

O resultado de um estudo mais aprofundado deste fenômeno poderá ser utilizado para prever se o modelo apresentará um bom *speedup* utilizando simulação distribuída com ParSMPLX, a partir dos dados da especificação do modelo. Uma vez que isto seja parametrizável, esta informação poderá ser utilizada no nível 3 do módulo avaliador do ASDA, auxiliando na escolha do protocolo.

III. Influência da granulosidade dos processos

A inserção de laços de granulosidade, de modo a simular um processamento mais complexo dentro de cada recurso do processo lógico, também representa grande influência no desempenho. Como esses laços agregam mais processamento sem que haja aumento de comunicação (tanto de mensagens completas quanto nulas), sempre há melhora de *speedup*, especialmente em modelos onde há muita comunicação e por isso o *speedup* original é baixo.

Em modelos mais carregados, com maior número de clientes e que apresentam mais processamento do que comunicação, a melhora do *speedup* provocada pelo acréscimo de um laço de granulosidade com 50 mil voltas foi de 13%, enquanto em modelos menos carregados, que apresentavam muita comunicação, a melhora provocada pelo acréscimo de laços de 50 mil voltas foi de cerca de 5150%.

IV. Influência do balanceamento

O impacto do balanceamento do modelo em termos de número de recursos em cada processo lógico não é tão relevante, quando comparado aos fatores descritos anteriormente (utilização de *lookahead* dinâmico, carga de clientes no sistema e granulosidade do modelo).

Em contrapartida, o balanceamento do modelo quanto à utilização dos recursos de cada processo lógico mostrou-se fundamental ao desempenho da simulação distribuída com ParSMPLX e deve ser levado em consideração no particionamento ou parametrização do modelo, ainda que seja utilizado *lookahead* dinâmico, ou que haja muita carga no sistema e que se esteja simulando modelos maiores.

Inserção do ParSMPL no ASDA

A partir da adaptação do ParSMPL ao Linux e da avaliação de seu desempenho foi possível determinar os passos necessários para utilizá-lo no ASDA. O gabarito existente para auxiliar a geração de código utilizando ParSMPL (Bruschi, 2002) foi adaptado para se adequar às características do ParSMPLX. Foi feito um estudo da possibilidade de reutilização da lógica envolvida no gerador de aplicações existente para o ambiente ASiA, visando reduzir o esforço de implementação de um novo gerador para programas de simulação utilizando ParSMPLX. Um trabalho nesse sentido mostra-se totalmente viável e justificado.

8.3. Contribuições

As seguintes contribuições deste trabalho devem ser destacadas:

- ✓ O levantamento dos pontos de extensão do gerador de aplicações do ASiA (Spolon, 1994), de modo a reutilizar a lógica existente no desenvolvimento do programa gerador para simulação distribuída CMB, a ser incluído ao ambiente ASDA (Bruschi, 2002). A previsão de novas funcionalidades e uma sugestão dos algoritmos também auxiliará o futuro desenvolvimento do gerador.
- ✓ A elaboração de um guia de utilização do ParSMPLX, visando facilitar sua utilização, mesmo por usuários que não conheçam a extensão funcional SMPL (MacDougall, 1987), na qual foi baseada.

- ✓ A adaptação do ParSMPLX que possibilitou a utilização de *lookahead* dinâmico, assim como a sugestão de uma metodologia relativamente simples para o cálculo de lookahead dinâmico.
- ✓ A elaboração de diretrizes para o particionamento do modelo em processos lógicos, de modo a complementar as diretrizes descritas em (Ulson, 1999), e que serão utilizadas no desenvolvimento do nível 3 do módulo avaliador do ASDA.
- ✓ A detecção da existência de um pico de *speedup* no desempenho das simulações com o ParSMPLX, que poderá vir a ser utilizada como informação para escolha de protocolo no nível 3 do módulo avaliador do ASDA.
- ✓ A solução de problemas de compilação e execução do ParSMPL (Ulson, 1999) no sistema operacional Linux.
- ✓ A inclusão de funções para depuração no ParSMPLX.
- ✓ O aprimoramento do gabarito para ParSMPLX, apresentado inicialmente em (Bruschi, 2002), que será incluído no módulo gerador do ASDA.

Ainda como uma contribuição, porém mais pontual ao trabalho no grupo de Sistemas Distribuídos e Programação Concorrente, foi a transferência de conhecimento no que diz respeito ao desenvolvimento de simulações utilizando o ParSMPL/ParSMPLX. Isso se fez também pelo desenvolvimento do guia de utilização (apêndice A), mas principalmente pelo auxílio na utilização do ParSMPLX em trabalho relacionado (Kawabata, 2002).

8.4. Sugestões de Trabalhos Futuros

Como sugestões para trabalhos futuros:

- ✓ A implementação do módulo gerador para simulação distribuída conservativa CMB. O capítulo 7 apresenta as diretivas para essa implementação, reutilizando a lógica existente para o gerador de aplicações do ambiente ASiA (Spolon, 1994). Como existem vários programas já codificados, que foram utilizados nos testes com o ParSMPLX, eles podem ser utilizados como parâmetros de comparações com as possíveis saídas do gerador de aplicações.
- ✓ Um estudo de uma maneira de se analisar o grafo do processo lógico e assim gerar a fórmula para se calcular automaticamente o *lookahead* dinâmico. Como os modelos no ASDA serão armazenados em grafos (Bruschi, 2002),

devem ser considerados algoritmos para análise de grafos e árvores. Uma sugestão de abordagem é que o grafo seja primeiro transformado em árvore. A partir disso, se só houver nós irmãos, deve-se considerar o menor valor de tempo de serviço entre eles. Se for nós filhos, os valores de tempo de serviço devem ser somados. A partir deste princípio, elaborar a fórmula para o cálculo do *lookahead* dinâmico.

- ✓ Um estudo de outras técnicas para *lookahead* dinâmico, para que possa ser incluído ao módulo gerador do ASDA, ou diretamente ao ParSMPLX.
- ✓ Um estudo mais aprofundado do comportamento da simulação distribuída com ParSMPLX, quando há aumento gradual de clientes no sistema. O resultado desse estudo pode ser incluído no nível 3 do módulo avaliador e servirá como parâmetro para auxiliar na escolha do protocolo de simulação distribuída a ser utilizado.

Referências Bibliográficas

- (Aiza, 2003) AIZA, T. *Avaliação do Uso de Statecharts Estocásticos e Queuing Statecharts em Um Ambiente de Simulação Automático*. 2003. 68p. Monografia de qualificação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Almasi & Gottlieb, 1994) ALMASI, G. S; GOTTLIEB, A. *Highly Parallel Computing*. 2.ed., The Benjamin Cummings Publishing Company, Inc. 1994. 690p.
- (Ayani, 1993) AYANI, R. Parallel Simulation. In *Performance Evaluation of Computer and Communication*. LNCS v. 729. p.1-20. Springer Verlag, 1993.
- (Bagrodia & Liao, 1994) BAGRODIA, R.; LIAO, W.T. Maisie User Manual Release 2.2, UCLA Parallel Computing Lab, 1995.
- (Bagrodia et al., 1987) BAGRODIA, R. et al. A Message-Based Approach to Discrete-Event Simulation. *IEEE Transaction on Software Engineering*, abr. 1987.
- (Bagrodia et al., 1998) BAGRODIA, R. et al. PARSEC: A Parallel Simulation Environment for Complex Systems. *IEEE Computer*, v.3, n.10, p.77-85, 1998.
- (Bagrodia, 1994) BAGRODIA, R. Language Support for Parallel Discrete-Event Simulations. In *Proceedings of 1994 Winter Simulation Conference*, p.1324-1331. 1994.
- (Banks, 1996) BANKS, J. Output Analysis Capabilities of Simulation Software. *Simulation*, v.66, n.1, p. 23-30, Janeiro. 1996.
- (Banks, 1998) BANKS, J. Principles of Simulation. In: BANKS, J. (ed.). *Handbook of Simulation: Principles, Metodology, Advances, Applications and Practice*. John Wiley and Sons, Inc., 1998. 848p.
- (Biles & Kleijnen, 2003) BILES, W.E. KLEIJNEN, J.P. Statistical Methodology for WEB-Based Simulation. In: *Proceedings of the 7th IEEE International Symposium on Distributed Simulation and Real-Time Applications*. p. 1530-1990. 2003.
- (Boukerche & Tropper, 2001) BOUKERCHE, A. TROPPER, C. Local versus Global Lookahead in Conservative Parallel Simulations. *Parallel Computing*, v. 27, p. 1033-1055. 2001.
- (Boukerche, 2002) BOUKERCHE, A. An Adaptative Partitioning Algorithm for Distributed Discrete Event Simulation Systems. *Journal of Parallel and Distributed Computing*, v. 62, p. 1454-1475. 2002.

- (Bruschi et al., 2000) BRUSCHI, S. M.; SANTANA, R. H. C.; SANTANA, M. J. ASDA - An Automatic Distributed Simulation Environment. In *Proceedings of the 2000 Summer Computer Simulation Conference (SCSC' 2000)*, Vancouver, Canada, July 16-20, 2000.
- (Bruschi, 1997) BRUSCHI, S. M. *Extensão do ASiA para Simulações de Arquiteturas de Computadores*. 1997. 88p. Dissertação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Bruschi, 2002) BRUSCHI, S. M. *ASDA - Um Ambiente de Simulação Distribuída Automático*. 2002. Tese (Doutorado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Bryan, 1989) BRYAN, O. MODSIM II – An Object Oriented Simulation Language for Sequential and Parallel Processors. In *Proceedings of the 1989 Winter Simulation Conference*, p. 172-177. Washington, DC: dez. 1989.
- (Bryant, 1977) BRYANT, R.E. *Simulation of Packet Communications Architecture Computer Systems*. Relatório Técnico, MIT-LCS-TR-188. Massachusetts Institute of Technology, 1977.
- (Cassel & Pidd, 2001) CASSEL, R. PIDD, M. Distributed Discrete Event Simulation Using the Three-Phase Approach and Java. *Simulation Practice and Theory*, v. 8, p. 491-507. 2001.
- (Chandy & Misra, 1979) CHANDY, K.M.; MISRA, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, v. SE-5, n.5, p.440-452, set. 1979.
- (Chandy & Misra, 1981) CHANDY, K. M.; MISRA, J. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of ACM*, v.24, n.11, p.198-205, 1981.
- (Chandy & Misra, 1987) CHANDY, K.M.; MISRA, J. *Conditional Knowledge as a Basis for Distributed Simulation*. Relatório Técnico, Computer Science Department, Califórnia Institute of Technology, 1987. Disponível em: <<http://resolver.library.caltech.edu/caltechCSTR:1987.5251-tr-87>>. Acesso em 15 novembro 2003.
- (Chandy & Sherman, 1989) CHANDY, K.M.; SHERMAN, R. The Conditional Event Approach to Distributed Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, v. 21, n.2, p.93-99, mar. 1989.
- (DeRose et al., 2000) DEROSE, L.A. et al. Performance Issues in Parallel Processing Systems. In HARING, G. et al. (eds). *Performance Evaluation: Origins and Directions*. LNCS v.1769. p.141-159. Springer Verlag, 2000.
- (Dota, 2001) DOTA, M.A. *Avaliação de Desempenho de uma Ferramenta para Simulação Distribuída Baseada no Protocolo Otimista Time Warp*. 2001. 100p. Dissertação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Duncan, 1990) DUNCAN, R. A Survey of Parallel Computer Architectures. *IEEE Computer*, p.5-15. 1990.
- (Edward et al., 1995) EDWARD, M. et al. ParaSol: A Multithreaded System for Parallel Simulation Based on Modifiable Threads. In *Proceedings of 1995 Winter Simulation Conference*, p.690-697. 1995.

- (Ensink et al., 1995) ENSINK, B. et al. Program Control Language: a programming language for adaptative distributed applications. *Journal of Parallel and Distributed Computing*, v. 63, p.1082-1104. 2003.
- (Ewing et al., 1999) EWING, G.C.; PAWLIKOWSKI, K.; MCNICKLE, D. Akaroa-2: Exploiting Network Computing by Distributing Stochastic Simulation. In *Proceedings of European Simulation Multiconference ESM-99*, Varsóvia, Polônia, p.175-181. 1999.
- (Fersha, 1996) FERSCHA, A. Parallel and Distributed Simulation of Discrete Event Simulation. In: ZOMAYA, A. Y. *Parallel and Distributed Computing Handbook*. McGraw-Hill, Inc. 1996. 1198p.
- (Flynn, 1972) FLYNN, M. J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, v. c-21, n. 9, p.948-960, set. 1972.
- (Flynn, 1996) FLYNN, M. J. Parallel Architectures. *ACM Computing Surveys*, v.28, n.1, p.67-70, mar. 1995.
- (Francês, 2001) FRANCÊS, C. R. L. Statecharts Estocásticos e Queuing Statecharts: Novas Abordagens para Avaliação de Desempenho Baseadas em Especificação Statecharts. 2001. Tese (Doutorado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Fujimoto, 1990) FUJIMOTO, R. Parallel Discrete Event Simulation. *Communications of the ACM*, v.33, n.9, p.30-53, Out. 1990.
- (Fujimoto, 1998) FUJIMOTO, R. Parallel Discrete Event Simulation. In: BANKS, J. (ed.). *Handbook of Simulation: Principles, Metodology, Advances, Applications and Practice*. John Wiley and Sons, Inc., 1998. 848p.
- (Fujimoto, 2001) FUJIMOTO, R. Parallel and Distributed Simulation Systems. In *Proceedings of the 2001 Winter Simulation Conference*, p.147-157. 2001.
- (Gill et al., 1989) GILL D. et al. An Interface for Programming Parallel Simulations. In *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, p.151-154. Tampa, Florida: mar. 1989.
- (Groselj, 1995) GROSELJ, B. CPSim: A Tool for Crating Scalable Discrete-Event Simulations. In *Proceedings of 1996 Winter Simulation Conference*, p.579-583, dez. 1995.
- (Heidelberger, 1988) HEIDELBERGER, P. Discrete Event Simulation and Parallel Processing: Statistical Properties. *SIAM J. Stat. Comput.*, v.9, n.6, p.1114-1132, 1988.
- (Hoover & Perry, 1990) HOOVER, S.V. PERRY, R.F. *Simulation: A problem-solving approach*. Reading, Mass.: Addison-Wesley, 1990. 296 p.
- (Hwang & Briggs, 1984) HWANG, K.; BRIGGS, F. A. *Computer Architecture and Parallel Processing*. McGraw-Hill, Inc. 1993. 772p.
- (Hwang, 1993) HWANG, K. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc. 1993. 772p.
- (Iskra et al, 2003) ISKRA, K.A. Time Warp Cancellation Optimisations on High Latency Networks. In: *Proceedings of the Seveth IEEE International Symposium on Distributed Simulation and Real-Time Applications*. 2003.
- (Jain, 1991) JAIN, R. *The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Inc. 1991. 685p.

- (Jefferson, 1985) JEFFERSON, D. Virtual Time. *ACM Transaction on Programming Languages and Systems*. v.7, n.3, p.405-425, jul. 1985.
- (Kawabata, 1999) KAWABATA, C. L. O. *Extensão do Ambiente de Simulação Automático (ASiA) para Simulação de Redes de Computadores*. 1999. 147p. Dissertação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Kawabata, 2002) KAWABATA, C. L. O. *Projeto e Implementação do Mecanismo de Troca de Protocolos de Sincronização em Tempo de Execução*. São Carlos, 2002. Monografia de Qualificação (Doutorado em Ciências da Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo.
- (Kobayashi, 1978) KOBAYASHI, H. *Modelling and Analysis: An Introduction to System Performance Evaluation Methodology*. Addison-Wesley Publishing Company. 1978. 446p.
- (Kronsjö, 1996) KRONSJÖ, L. I. PRAM Models. In: ZOMAYA, A. Y. *Parallel and Distributed Computing Handbook*. McGraw-Hill, Inc. 1996. 1198p.
- (Law & Kelton, 1991) LAW, A.M., KELTON, W.D. *Simulation Modeling & Analysis*. 2.ed. Singapore : Mc Graw-Hill International, 1991. 760p.
- (Lewis, 1993) LEWIS, T. G. *Foundations of Parallel Programming: A machine-independent approach*. Los Alamitos, California : IEEE Computer Society Press. 1993. 296p.
- (Low et al., 1999) LOW, Y.H. Survey of Languages and Runtime Libraries for Parallel Discrete Event Simulation. *Simulation*, n.72, v.3, p.170-186, mar. 1999.
- (MacDougall, 1987) MACDOUGALL, M.H. *Simulating Computer Systems: Techniques and Tools*. Cambridge : The MIT Press, 1987. 292p.
- (Martin & Bagrodia, 1995) MARTIN, J.M.; BAGRODIA, R.L. COMPOSE: An Object-Oriented Environment for Parallel Discrete-Event Simulations. In *Proceedings of the 1995 Winter Simulation Conference*, p.163-166, 1995.
- (Martin et. al, 1996) MARTIN, D.E. et al. WARPED: a time warp simulation kernel for analysis and application development. In: *29th Hawaii International Conference of Distributed System Sciences*, Computer Science Press, Big Island, HI, p. 383-386, jan. 1996.
- (Martini et. Al., 1997) MARTINI, P. et. al. Tolerant Synchronization for Distributed Simulations of Interconnected Computer Networks. *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*. Society for Computer Simulation, p.138-141, jun. 1997.
- (McBryan, 1994) MCBRYAN, O. A. An Overview of Message Passing Environments. *Parallel Computing*. v. 20, n. 4, p. 417-443, abr. 1994.
- (Misra, 1986) MISRA, J. Distributed Discrete-Event Simulation. *Computing Surveys*. v. 18, n.1, p.39-65, mar. 1986.
- (Mota et. al., 1999) MOTA, E. et. al. Methods of Batch Means in Simulation Output Analysis under Multiple Replications in Parallel. In: *Proceedings of the 4th United Kingdom Simulation Society Conference*, p. 62-66, Cambridge, UK, abr. 1999.
- (Navaux, 1989) NAVAUX, P. O. A. Introdução ao Processamento Paralelo. *Revista Brasileira de Computação*. Rio de Janeiro, v.5, n.2, p. 31-43, out./dez. 1989.
- (Nicol, 1996) NICOL, D. M. Principles in Conservative Parallel Simulation. In *Proceedings of 1996 Winter Simulation Conference*, p.128-136. 1996.

- (Nicol, 2000) NICOL, D. M. Discrete Event Simulation in Performance Evaluation. In HARING, G. et al. (eds). *Performance Evaluation: Origins and Directions*. LNCS v.1769, p.443-457. Springer Verlag, 2000.
- (Nketsa & Khalifa, 2001) NKETSA, A.; KHALIFA, N. B. Timed Petri Nets and Preciction to Improve the Chandy-Misra Conservative-Distributed Simulation. In *Applied Mathematics and Computation*, v.120, p.235-254. Elsevier, 2001.
- (Oliveira, 1998) OLIVEIRA, L.L. Um Módulo de Análise de Resultados para o ASiA. 1998. 153p. Dissertação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Orlandi, 1995) ORLANDI, R. C. G. S. *Ferramentas para Análise de Desempenho de Sistemas Computacionais Distribuídos*. 1995. 1997. 88p. Dissertação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Osman, 2000) OSMAN, B. Verification, Validation, and Testing. In: BANKS, J. (ed.). *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. John Wiley & Sons, Inc. 1998. 848p.
- (Overeinder et. al., 1991) OVEREINDER, B. et. al. Parallel Discrete Event Simulation. In *The Third Workshop Computersystems*, Faculty of Electrical Engineering, Eindhoven University. p.19-30. Eindhoven. The Netherlands: mai. 1991. Disponível em: <<http://www.science.uva.nl/research/scs/papers/index.html>>. Acesso em 15 de novembro de 2003.
- (Paulikowski & Yau, 1992) PAULIKOWSKI, K.; YAU, V. An Automatic Partitioning, Runtime Control and Output Analysis Methodology for Massively Parallel Simulations. In *Proceedings of the European Simulation Symp*, p.135-139. 1992.
- (Pedroso, 1996) PEDROSO, W.G. Módulo visual: uma interface gráfica para o ambiente de simulacao automatico. 1996. 87p. Dissertação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Perrot, 1996) PERROT, R. H. Parallel Languages. In: ZOMAYA, A. Y. *Parallel and Distributed Computing Handbook*. McGraw-Hill, Inc. 1996. 1198p.
- (Preiss & Loucks, 1989) PREISS, B.R; LOUCKS, W. Prediction and Lookahead in Distributed Simulation. Relatório Técnico. University of Waterloo, Canadá. 1989.
- (Preiss, 1989) PREISS, B.R. The YADDES Distributed Discrete Event Simulation Specification Language and Execution Environments. In *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, p.139-144. 1989.
- (Rao et al., 1998) RAO, D.M. et al. Unsynchronized Parallel Discrete Event Simulation. In *Proceedings of the 1998 Winter Simulation Conference*. p.1563-1570. 1998.
- (Rao & Wisley, 2002) RAO, D.M. WISLEY, P.A. An Ultra-Large Simulation Framework. *Journal of Parallel and Distributed Computing*, v. 62, p.1670-1693. 2002.
- (Santana et al., 1994) SANTANA, R.H.C. et al. *Técnicas para Avaliação de Desempenho de Sistemas Computacionais*. Notas Didáticas. n.14. ICMC – USP: 1994. 31p.
- (Santana, 1996) SANTANA, M.J. Graphical user interface for an automatic simulation system. In: *Proceedings of Summer Computer Simulation Conference*. Portland: Scs, p.139-143. 1996.

- (Santana, 1996a) SANTANA, M.J. Automatic generation of discrete system simulation program.. In: *Proceedings of Summer Computer Simulation Conference*. Portland: Scs, p.133-138, 1996.
- (Skillicorn & Talia, 1998) SKILLICORN, D.B.; TALIA, D. Models and Languages for Parallel Computation. *ACM Computing Surveys*, v.30, n.2, p.123-169, jun. 1998.
- (Soares, 1990) SOARES, L.F.G. *Modelagem e Simulação Discreta de Sistemas*. São Paulo, IME-USP: Escola de Computação, 7, 1990. 250p.
- (Spolon et al., 2000) SPOLON, R. et al. *Simulação Distribuída, Timw Warp e Variantes: Descrição e Análise Comparativa*. Relatório Técnico. ICMC – USP, São Carlos: abr. 2000.
- (Spolon, 1994) SPOLON, R. *Um Gerador de Aplicação para um Ambiente de Simulação Automático*. 1994. 1997.109p. Dissertação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Spolon, 1994a) SPOLON, R. *Editor gráfico para um Ambiente de Simulação Automático*. 1994. 1997. 139p. Dissertação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.
- (Steinman, 1991) STEINMAN, J. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. In: *Advances in Parallel and Distributed Simulation, SCS Multiconference*, p. 95-103. 1991.
- (Sunderam & Rego, 1991) SUNDERAM, V.S.; REGO, V.J. EcliPse: A System for High Performance Concurrent Simulation. *Software – Practise and Experience*, n.11, p.1189-1219, 1991.
- (Sunderam et al., 1994) SUNDERAM, V.S.; GEIST, A.; DONGARRA, J. The PVM concurrent computing system: evolution, experiences and trends. *Parallel Computing*, v.20, n.4, p.531-545, 1994.
- (Swain, 1995) SWAIN, J.J. Simulation on Survey: Tools for process understanding and improvement. *OR/MS Today*, Agosto, 1995.
- (Tanenbaum, 2001) TANENBAUM, A. S. *Modern Operating Systems*. 2.ed. Upper Saddle River, Nova Jérsei: Prentice Hall, Inc. 2001. 952p.
- (Theodoropoulos, 2002) THEODOROPOULOS, G.K. Distributed Simulation of Asynchronous Hardware: The Program Driven Synchronization Protocol. *Journal of Parallel and Distributed Computing*, v. 62, p. 622-655. 2002.
- (Thondugulam et. al., 1999) THONDUGULAM, N.V. et. al. Relaxing Casual Constraints in PDES. In: *Proceedings of the 13th International Parallel Processing Symposium (IPPS/SPDP '99)*, p. 696-700, abr. 1999.
- (Tropper, 2002) TROPPER, C. Parallel Discrete-Event Simulation Applications. *Journal of Parallel and Distributed Computing*, v.62, p. 327-335. 2002.
- (Ulson, 1999) ULSON, R. S. *Simulação Distribuída sobre Plataformas de Portabilidade: Viabilidade de Uso e Comportamento do Protocolo CMB*. 1999. 277p. Tese (Doutorado em Física Aplicada Computacional) - Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos.
- (Ulson, 1999a) ULSON, ROBERTA S.; MORSELLI JR., J. C. M.; SANTANA, R. H. C.; SANTANA, M. J. Conservative Distributed Simulation on Portability Plataforms: The CMB Protocol Behaviour. In: *Proceedings of IASTED International Conference on*

- Applied Modelling and Simulation (AMS'99)*. Cairns, Queensland, Austrália (September 1-3), 1999.
- (Varga, 2001) VARGA, A. The OMNeT++ Discrete Event Simulation System, In: *Proceedings of the European Simulation Multiconference – ESM'2001*, June 6-9, 2001, Prague, Czech Republic.
- (Varga, 2003) VARGA, A. OMNeT++ Discrete Event Simulation System. Disponível em <<http://www.omnetpp.org/>>. Acesso em 14 novembro 2003.
- (Vee & Hsu, 1999) VEE, V.; HSU, W. *Parallel Discrete Event Simulation: A Survey*. Relatório Técnico, Centre for Advanced Information Systems, Nanyang Technological University, Singapore: ago, 1999.
- (Waldorf & Bagrodia, 1994) WALDORF, J.; BAGRODIA, R. MOOSE: A Concurrent Object-Oriented Language for Simulation. *International Journal of Computer Simulation*, v.4, n.2, p.235-257, 1994.
- (Wonnacott & Bruce, 1995) WONNACOT, P.; BRUCE, D. The Design of APOSTLE – A High-Level, Object-Oriented Language for Parallel and Distributed Discrete Event Simulation. In *Proceedings of Object-Oriented Simulation, 1995 Western Multiconference on Computer Simulation*, p.101-106. Las Vegas: jan. 1995.
- (Zarei & Pidd, 2001) ZAREI, B.; PIDD, M. Performance analysis of automatic lookahead generation by control flow graph: some experiments. In *Simulation, Practice and Theory*, v.8, p. 511-527, 2001.
- (Zomaya, 1996) ZOMAYA, A. Y. Parallel and Distributed Computing: The Scene, the Props, the Players. In: ZOMAYA, A. Y. *Parallel and Distributed Computing Handbook*. McGraw-Hill, Inc. 1996. 1198p.

Apêndice A – Guia de utilização do ParSMPLX

A.1. Considerações iniciais

O ParSMPLX é uma biblioteca para simulação distribuída utilizando o protocolo de sincronização CMB (Bryant, 1977; Chandy & Misra, 1979; Misra, 1986), baseado na biblioteca SMPL para simulação seqüencial.

No ParSMPLX, assim como no SMPL, os sistemas são modelados em função de três tipos de entidades: *facilidades* (também chamadas de *recursos*), *tokens* e *eventos* (MacDougall, 1987).

As *facilidades* compreendem os recursos sendo modelados no sistema. Podem representar recursos de *hardware* (como uma CPU, uma unidade de disco, ou um computador em uma rede), como também recursos de software, como um *lock* do sistema operacional. De modo geral, elas representam as unidades *estáticas* do sistema. A interconexão entre elas é definida pelo fluxo de *tokens* no sistema.

Os *tokens* representam as entidades *ativas* do sistema, e o movimento dos *tokens* entre as facilidades modela o comportamento do sistema sendo simulado. Um token pode representar, por exemplo, uma tarefa em um sistema computacional, como também um acesso a disco.

Os eventos representam uma mudança de estado de qualquer entidade, seja ela ativa (*tokens*) ou passivas (*facilidades*). Se por exemplo uma tarefa completa seu intervalo de execução em uma CPU e a libera, então dois eventos ocorreram: o fim da atividade da tarefa e a mudança de estado da CPU.

MacDougall (MacDougall, 1987) apresenta uma descrição detalhada das funções do SMPL. Deve-se observar, porém, que as funções *report*, *preempt* e *cancel* do SMPL não foram implementadas em ParSMPLX.

A.2. Requisitos do sistema

- Sistema Operacional Linux. Os ajustes aplicados ao ParSMPLX e os testes foram realizados em distribuições RedHat 7.2 e Conectiva 7.0.
- PVM versão 3.4
- Compilador C *gcc*. Na fase de adaptação e testes do ParSMPLX foi utilizada a versão egcs-2.91.66.

A.3. API do ParSMPLX

A descrição das funções que compõem a API (*Application Program Interface*) do ParSMPLX segue a seguinte notação:

nome_função(tipo_parâmetro parâmetro) : tipo_retorno

A descrição está agrupada em três blocos: funções para geração de variáveis aleatórias, funções da biblioteca e funções para depuração.

A.3.1 Funções para geração de variáveis aleatórias

erlang (double x, double s) : double

Retorna uma variável pseudo-aleatória a partir de uma distribuição Erlang, com média *x* e desvio padrão *s*.

Parâmetros:

- *x*: média da distribuição
- *s*: desvio padrão

Exemplo:

```
double tempo_servico = erlang (30.5, 15);
```

expntl (double x) : double

Retorna uma variável pseudo-aleatória a partir da distribuição exponencial negativa com média *x*.

Parâmetros:

- x : média da distribuição

Exemplo:

```
double tempo_servico = expntl (4.5);
```

hiperx (double x , double s): double

Retorna uma variável pseudo-aleatória a partir da distribuição hiper exponencial de dois estágios de Morse, com média x e desvio padrão s .

Parâmetros:

- x : média da distribuição
- s : desvio padrão

Exemplo:

```
double tempo_servico = hiperx (5.5, 6.25);
```

normal (double x , double s): double

Retorna uma variável pseudo-aleatória a partir de uma distribuição normal com média x e desvio padrão s . Como pode gerar valores negativos, é necessário testar o valor retornado antes de utilizá-lo.

Parâmetros:

- x : média da distribuição
- s : desvio padrão

Exemplo:

```
double tempo_servico = normal (100.0, 7.5);
```

randompar (int i , int n): int

Retorna um número inteiro selecionado equiprobabilisticamente no intervalo $[i, n]$.

Parâmetros:

- i : menor valor do intervalo
- n : maior valor do intervalo

Exemplo:

```
int recurso_escolhido = (1, 5);
```

`stream ( int n ): int`

Seleciona uma entre as 15 sementes disponíveis para geração de números aleatórios.

O uso desta função no código do programa de simulação do usuário não é obrigatório, pois a semente de valor 1 é selecionada ao se iniciar o programa de simulação, dentro da função `smpl()`. Quando se deseja iniciar o programa de simulação com outro valor de semente para geração de números aleatórios, isto deve ser feito através desta função, que deve ser chamada **depois** da função `smpl()`. O valor retornado pela função é o valor da semente associada.

Parâmetros:

- *n*: valor da semente para geração de números aleatórios, deve estar no intervalo [1, 15].

Exemplo:

```
stream (2);
```

`uniform ( double a, double b ): double`

Retorna uma variável pseudo-aleatória a partir de uma distribuição uniforme dentro do intervalo [a, b].

Parâmetros:

- *a*: menor valor do intervalo
- *b*: maior valor do intervalo

Exemplo:

```
double tempo_servico = uniform (32.5, 12.0);
```

A.3.2 Funções da biblioteca ParSMPLX

`cause ( int *evento, int *token ): void`

Remove um evento da lista de eventos futuros e retorna o número do evento que deve ser simulado e o número do token associado a ele.

Devem ser passados os **ponteiros** para as variáveis *evento* e *token*, como mostra o exemplo, pois essas são as variáveis de retorno e terão seus valores alterados dentro da função `cause()`.

Parâmetros:

- *evento*: evento que será selecionado para simulação imediata
- *token*: token associado ao evento

Exemplo:

```
int  evento,  
    token;  
  
...  
cause (&evento, &token);  
  
...
```

clock_sim(): double

Retorna o valor do tempo simulado marcado pelo relógio da simulação. Substitui a função *time()* do SMPL.¹

Parâmetros:**Exemplo:**

```
if ( clock_sim() > 10000 )  
{  
    // executa alguma ação...  
}
```

comprimento_fila (char *nome_recurso): int

Indica o comprimento da fila no instante sendo simulado.

Parâmetros:

- *nome_recurso*: nome do recurso cuja fila se deseja saber o comprimento.

Exemplo:

```
if ( comprimento_fila("disco") == 0 )  
    // incrementa contador de fila vazia  
    fila_vazia_disco ++;
```

comprimento_medio_fila (char *nome_recurso): double

Dado de desempenho da facilidade indicada por *nome_recurso*. Indica o comprimento médio da fila até o instante sendo simulado.

¹ No ParSMPL/ParSMPLX a função *time()* teve seu nome modificado devido à confusão que poderia causar com a função *time()* de sistemas Unix/Linux.

Parâmetros:

- *nome_recurso*: nome do recurso cuja fila se deseja saber o comprimento.

Exemplo:

```
printf ("Número médio de tarefas na fila da CPU: %d",  
comprimento_medio_fila("cpu") );
```

facility (char *nome_recurso, int num_serv): void

Esta função define as facilidades (ou *recursos*) da simulação.

Parâmetros:

- *nome_recurso*: especifica o nome associado ao recurso sendo modelado pela facilidade.
- *num_serv*: indica quantos servidores o recurso sendo modelado possui.

Exemplo:

```
// define um recurso caixa que contém três  
// atendentes (i.e., servidores) para uma fila única  
facility ( "Caixa Rápido", 3);
```

parsmpl_pvm_exit (): void

Função que termina o subsistema de simulação, liberando as estruturas de dados alocadas. Deve ser chamada ao fim do programa de simulação.

Parâmetros:

Exemplo:

```
int main () {  
    ...  
    // corpo da simulação  
    ...  
    parsmpl_pvm_exit();  
    exit(0);  
}
```

`periodo_medio_ocupado ( char *nome_recurso ): double`

Dado de desempenho da facilidade indicada por *nome_recurso*. Indica o período médio em que a facilidade esteve ocupada até o instante sendo simulado.

Parâmetros:

- *nome_recurso*: nome do recurso que se deseja saber o período médio ocupado.

Exemplo:

```
printf ("Período médio ocupado da CPU: %d",
periodo_medio_ocupado("cpu") );
```

`release ( char *nome_recurso, int token ): void`

Libera o servidor da facilidade indicada por *nome_recurso* que foi reservada ao token.

Parâmetros:

- *nome_recurso*: nome do recurso que deve ser liberado
- *token*: token que está utilizando o recurso

Exemplo:

```
int cliente;
...
// um cliente (token) termina de ser atendido pelo caixa
release ("Caixa Rápido", cliente);
```

`request (char *nome_recurso, int evento, int token,
int prioridade, int proc): int`

Esta função serve para solicitar a reserva de um recurso. Caso o recuso solicitado possua algum servidor livre, esse servidor é reservado ao token e a função retorna valor zero. Caso contrário, o evento é colocado na fila de eventos do recurso e a função retorna valor 1.

Parâmetros:

- *nome_recurso*: nome do recurso sendo requisitado.
- *evento*: indica o número do evento ocorrido associado ao token
- *token*: token que está solicitando o recurso
- *prioridade*: prioridade do token
- *proc*: número do processo lógico onde se encontra o recurso solicitado.

Exemplo:

```
// solicita reserva da cpu
if (request("cpu", evento, token, 1, 0) == 0)
{
    // se conseguiu a reserva, chama a função schedule
    // para agendar a liberação da cpu
}
```

```
schedule ( int num_evento, double te, int token,
int ident_proc ): void
```

Agenda a próxima ocorrência de um evento. Esse evento pode ser, por exemplo, a solicitação ou liberação de um recurso.

Parâmetros:

- *num_evento*: número do evento sendo escalonado
- *te*: tempo decorrido entre a ocorrência deste evento e o próximo evento
- *token*: token associado ao evento
- *ident_proc*: número do processo lógico onde se encontra a definição do evento sendo escalonado.

Exemplo:

```
// calcula o tempo de serviço do recurso
double tempo_servico = erlang(30.0, 15.0);
// agenda a liberação da cpu (evento 3) para depois de
// decorrido o tempo simulado marcado por tempo_servico
schedule ( 3, tempo_servico, token, PL1);
```

```
smpl ( int m, char *modelo ): void
```

Esta função inicia o subsistema de simulação, associando os valores iniciais às variáveis da simulação.

Parâmetros:

- *m*: indica se a interface *mtr* deve ser ativada ($m = 1$) ou não ($m = 0$). O *mtr* é parte do ambiente de simulação SMPL e tem como objetivo prover uma interface interativa com a execução do modelo. Entretanto, o *mtr* não foi implementado no ParSMPL/ParSMPLX, de modo que o valor associado a *m* não fará diferença no comportamento da simulação.
- *modelo*: define o nome do modelo, que aparece no cabeçalho do relatório exibido ao fim da simulação. Como esse relatório é construído pela função *report()*, não implementada no ParSMPL, indicar um nome ao modelo não fará diferença no resultado.²

Exemplo:

```
// inicia simulação
smpl(0, "Sistema Computacional");
```

utilizacao_recurso (char *nome_recurso): double

Dados de desempenho da facilidade indicada por *nome_recurso*. Indica a utilização do recurso (número médio de servidores ocupados) até o instante sendo simulado.

Parâmetros:

- *nome_recurso*: nome do recurso que se deseja saber a utilização.

Exemplo:

```
printf ("Utilização da CPU: %f", utilizacao_recurso("CPU"));
```

A.3.3 Funções para depuração

imprime (char *nome_arq, char *mensagem): void

Imprime em arquivo mensagens para depuração. As mensagens geradas pelos processos lógicos são impressas em arquivos separados. As mensagens geradas pelo processo lógico pai serão impressas no arquivo *<nome_arquivo>_pl0*, as mensagens geradas pelo processo lógico 1 serão impressas no arquivo *<nome_arquivo>_pl1* e assim por diante.

² Os parâmetros *m* e *modelo*, embora não reflitam diferenças no comportamento da simulação, permaneceram na função *smpl()* visando manter a coerência com o SMPL e a compatibilidade com futuras implementações do ParSMPL/ParSMPLX.

Parâmetros:

- *nome_arq*: nome do arquivo onde será escrita a mensagem. Se não for informado um nome para o arquivo, será utilizado o nome padrão *debug_parsmpl_pl<número_processo_lógico>*.
- *mensagem*: mensagem utilizada para depuração

Exemplo:

```
// imprime a mensagem no arquivo de saída padrão
imprime("", "O valor do buffer está nulo");
// imprime a mensagem no arquivo saída_pl0
imprime("saida", "O valor do buffer está nulo");
```

imprime_buffer_canais (int canal): void

Imprime no arquivo de depuração padrão (*debug_parsmpl_pl<número_processo>*)

o tempo das mensagens recebidas pelo canal de comunicação indicado pelo índice e que ainda estão no *buffer* de canais.

Parâmetros:

- *canal*: número do canal de comunicação que se deseja saber o conteúdo do buffer

Exemplo:

```
// imprime o conteúdo do buffer do canal atual
imprime_buffer_canais (canal_atual);
```

imprime_lista_eventos_futuros ():void

Imprime no arquivo de depuração padrão (*debug_parsmpl_pl<número_processo>*)

o tempo de execução de todos os elementos contidos na lista de eventos futuros do processo lógico.

Parâmetros:

Exemplo:

```
imprime_lista_eventos_futuros();
```

A.3.4 Variáveis do ParSMPLX

As variáveis apresentadas a seguir fazem parte do ParSMPLX. Essas variáveis são declaradas no arquivo de cabeçalho *declaracoes.h* e devem ter seus valores definidos no programa de simulação.

tabela_tid[s]: int

Tabela contendo os *task IDs* dos processos lógicos envolvidos na simulação (processo pai + escravos). A simulação comporta até 10 processos lógicos e a posição zero é reservada ao processo pai. Caso a simulação possua menos de 10 processos lógicos, as entradas não utilizadas da tabela devem ser iniciadas com valor zero.

Esta tabela deve ter seu valor iniciado após a criação dos escravos (isto é feito via função *spawn* do PVM).

proc_local: int

Armazena o número do processo lógico local. Dentro da simulação, esse número é único e está na faixa de 0 a 9. Por convenção, é associado o valor zero ao processo lógico pai e valores de 1 a 9 aos processos escravos.

Esta variável deve ter seu valor iniciado após a criação dos escravos.

clock_canais[10]: double

Armazena o valor dos relógios de cada canal da simulação.

Deve ser iniciado com valor zero para cada canal utilizado. Por exemplo, se o processo lógico 1 comunica-se com o processo lógico pai (cujo número do processo é zero) e com o processo lógico escravo de número 2, então deve iniciar com valor zero apenas os relógios dos canais zero e 2.

É importante que os valores sejam iniciados **depois** da chamada à função *smpl*, pois nesta função é associado valor (-1) aos relógios de todos os canais, marcando que não estão utilizados.

tempo_processo: double

Armazena o valor utilizado como *lookahead*.

Utilizando *lookahead* **estático**, esse valor deve ser definido no **início** da simulação com o menor valor que o tempo de serviço do processo lógico pode assumir. É importante que esse valor seja definido **depois** da chamada à função *smpl*, pois esta função associa valor zero à variável.

Para utilização do *lookahead* **dinâmico**, esse valor deve ser definido logo **antes** da chamada à função *cause* e assim é re-calculado sempre que um dos tempos de serviço envolvidos no cálculo forem utilizados.

cont_mens_completas_enviadas: long

cont_mens_nulas_enviadas: long

cont_mens_completas_recebidas: long

cont_mens_nulas_recebidas: long

Contadores dos números de mensagens recebidas e enviadas, para mensagens nulas e completas.

Não precisam ser iniciadas no programa de simulação (pois elas têm seu valor iniciado dentro da função *smpl*), mas podem ser utilizadas ao fim da simulação como informativos de desempenho.

A.4. Estrutura de um programa de simulação utilizando o ParSMPLX

A figura A.1 mostra como deve ser a estrutura dos processos lógicos que implementam uma simulação distribuída utilizando o ParSMPL.

Para ilustrar melhor a implementação de um programa de simulação utilizando ParSMPLX, considere o sistema computacional da figura A.2. Esse sistema é composto por uma CPU e um disco. As tarefas chegam ao sistema com tempo entre chegadas obedecendo a uma distribuição exponencial com média igual a 50. Suponha que a CPU apresente seu tempo de serviço obedecendo a uma distribuição exponencial com média 50, e o tempo de serviço do disco obedece a uma distribuição exponencial com média igual a 100. Toda tarefa

apresenta uma probabilidade de 40% de sair do sistema e 60% de escolher utilizar o disco. Sempre que o disco for requisitado, a tarefa deve voltar à CPU.

PROCESSO MESTRE	PROCESSO FILHO
Inclui bibliotecas + núcleo ParSMPL Inicia Processos Prepara Tabela de Identificação Envia Tabela de Identificação Prepara Variáveis de Estado Inicia Variáveis <i>smpl()</i> Inicia clock_canais e tempo_processo <i>facility()</i> <i>schedule()</i> <i>cause(evento, token)</i> Evento 1: Evento 2: Evento n: Recebe Resultado Encerra Processo	Inclui bibliotecas + núcleo ParSMPL Recebe Tabela de Identificação Prepara Variáveis de Estado Inicia Variáveis <i>smpl()</i> Inicia clock_canais e tempo_processo <i>facility()</i> <i>cause(evento, token)</i> Evento 1: Evento 2: Evento n: Recebe Resultado Encerra Processo

Figura A. 1: Descrição de processos no ParSMPL

Esse sistema foi dividido em dois processos lógicos: PL0 (mestre), composto pela CPU, e PL1 (escravo), composto pelo disco. A codificação do processo mestre PL0 é apresentada nas figuras A.3 a A.6. Do processo escravo PL1, nas figuras A.7 e A.8.

Figura A. 2: Sistema Computacional composto por CPU e Disco

A inclusão de bibliotecas e núcleo do ParSMPL é feita nas linhas de 4 a 6 da figura A.3. As linhas 7 a 11 compreendem a definição de constantes utilizadas no programa de simulação. Os identificadores dos processos lógicos (no exemplo, as constantes *PL0* e *PL1*) devem seguir numeração de 0 a 9, pois esses identificadores referem-se às posições na tabela *tabela_tids[]*.

Note que o nome do programa escravo (linha 11) deve ser informado com seu caminho completo. Caso seja informado somente o nome, o PVM irá procurar o programa apenas na raiz do diretório *home* do usuário.

O ParSMPL/ParSMPLX não faz coleta do tempo de execução da simulação automaticamente. Caso essa informação seja desejável, isso deve ser feito manualmente, coletando-se os tempos inicial e final da simulação e calculando-se a diferença. As linhas 31 e 32 contêm a declaração das variáveis para coleta de tempo, e a coleta do tempo inicial é feita na linha 35. A coleta do tempo final e o cálculo do tempo de execução da simulação são apresentados nas linhas 135 e 136 da figura A.6.

```

1  /* -----
2  * pl0.c (Processo Lógico 0 - Mestre)
3  * ----- */
4  #include <stdio.h> /* biblioteca para I/O */
5  #include <sys/time.h> /* biblioteca para coleta de tempo */
6  #include "parsmpl.h" /* biblioteca para simulação distribuída CMR */
7  #define PL0 0 /* identificador do processo lógico pl0 */
8  #define PL1 1 /* identificador do processo lógico pl1 */
9  #define SIM_TIME 1000000 /* tempo simulado */
10 /* localização e nome do escravo */
11 #define SLAVENAME "/home/naninha/modelo3.pl1"
12
13 int main(void)
14 {
15     /* Declaração das variáveis */
16     int
17         tid_processo_pai, /* task id do processo pai */
18         tid_pl1, /* task id do processo filho */
19         token,
20         evento_execut, /* próximo evento a ser executado */
21         indice, /* índice para utilização no laço for */
22         info, /* retorno da fc. pvm_spawn (verif. erros) */
23         option,
24         num_tours = 0, /* número de rodadas da simulação */
25         continua = 1; /* variável de controle da simulação */
26     double
27         valor_exp,
28         tchegada = 50, /* tempo de chegada de novos eventos */
29         tsCPU = 50; /* tempo de serviço da CPU */
30     float dtl; /* tempo gasto na simulação */
31     struct timeval t1, /* tempo inicial da simulação */
32                 t2; /* tempo final da simulação */
33
34     /* Coleta o tempo inicial da simulação */
35     gettimeofday (&t1, (struct timezone*)0);
36
37     /* identifica o processo pai */
38     tid_processo_pai = pvm_mytid();
39

```

Figura A. 3: Codificação de PL0 (pl0.c - Parte 1 de 4)

A criação de processos escravos é mostrada nas linhas 41 a 48 da figura A.4. No exemplo, há apenas um processo escravo (cujo nome foi definido pela constante *SLAVENAME*) e é criado pela função *pvm_spawn* para ser executado na máquina "lasdpc12" (linha 41). O número da tarefa (*Task ID*) associado ao escravo criado é retornado na variável *tid_pl1* (último parâmetro da função *pvm_spawn*).

A preparação da tabela de identificação pode ser vista nas linhas 50 a 58 da figura A.4. Primeiro, todas as entradas devem ser iniciadas com zero (linhas 52 a 55). Depois, apenas as entradas da tabela referentes ao mestre e ao escravo devem ser preenchidas com os valores de *task ID* retornados pelo PVM (linhas 57 e 58).

```

40  /* cria o processo escravo */
41  info = pvm_spawn(SLAVENAME, (char**)0, 1, "lasdpc12", 1, &tid_pl1);
42  if( info < 1 )
43  {
44      /* detecta ocorrência de erros ao criar escravo */
45      printf("Erro ao criar escravo.Codigo de erro:\n");
46      printf("TID %d \n",tid_pl1);
47      pvm_exit();
48      exit(1);
49  }
50  /* Preparando tabela de identificacao */
51  /* inicia tabela de TIDs */
52  for (indice = 1; indice < 10; indice++)
53  {
54      tabela_tids[indice] = 0;
55  }
56  /* Preenche as entradas utilizadas */
57  tabela_tids[PL0] = tid_processo_pai;
58  tabela_tids[PL1] = tid_pl1;
59
60  /* Inicia comunicacao com o processo filho
61   * Parâmetro PvmDataRaw - utilizando máquinas homogêneas */
62  pvm_initsend(PvmDataRaw);
63
64  /* Empacota e envia a tabela para o processo filho */
65  pvm_pkint(tabela_tids,10,1);
66  pvm_send(tid_pl1,0);
67
68  /* Inicio da simulacao */
69  smpl(0,"Processo CPU");
70
71  /* define a semente para geracao de numeros aleatorios */
72  stream (10);
73
74  /* Inicia as variaveis do ParEMPL */
75  proc_local = PL0; /* define o numero do processo local */
76  clock_canais[PL1] = 0.0; /* inicia o canal de comunicacao utilizado */
77

```

Figura A. 4: Codificação de PL0 (pl0.c - Parte 2 de 4)

O envio da tabela de identificação aos processos escravos é feito nas linhas 60 a 66, utilizando funções do PVM.

As linhas 69 e 72 mostram, respectivamente, o início da simulação e a definição de um valor para a semente para geração de números aleatórios. Nas linhas 75 e 76 é feita a iniciação das variáveis. Note que a chamada à função *streamt*) e a iniciação das variáveis devem ser feitas depois da chamada à função *smplt*).

```

75      * Declara facilidade do processo */
76      facility("CPU",1);
77
78      /* Define apenas uma classe de tokens */
79      token = 1;
80
81      * gera as tarefas iniciais do sistema */
82      schedule(1, 0.0, token, PL0);
83
84      /* pré-computa o tempo de utilização da CPU,
85      * para ser utilizada no cálculo do lookahead dinâmico */
86      valor_exp = expntl (tsCPU);
87
88      /* Inicia o ciclo da simulação */
89      while (continua)
90      {
91          /* define o valor do lookahead do processo */
92          tempo_processo = valor_exp;
93          /* busca o próximo evento a ser executado */
94          cause(&evento_execut, &token);
95          /* Verifica o tipo do evento e toma a ação correspondente */
96          switch(evento_execut)
97          {
98              case 1: /* Gera novas chegadas ao sistema */
99                  schedule(1, expntl (tchegada), token, PL0);
100                  /* escalona a CPU */
101                  schedule(2,0.0,token,PL0);
102                  break;
103              case 2: /* reserva CPU */
104                  if (request("CPU",2,token,0,PL0) == 0)
105                  {
106                      /* escalona o fim da utilização da CPU */
107                      schedule(3,valor_exp,token,PL0);
108                      /* pré-computa o valor do tempo de serviço da CPU,
109                      * para ser utilizado no cálculo do lookahead */
110                      valor_exp = expntl (tsCPU);
111                  }
112                  break;
113          }
114      }
115

```

Figura A. 5: Codificação de PL0 (pl0.c - Parte 3 de 4)

A declaração de facilidades é feita na linha 79 da figura A.5. No exemplo, existe apenas uma facilidade ("CPU") no processo lógico sendo codificado, e esse recurso possui apenas um servidor.

Na linha 82, define-se que a simulação possui apenas uma classe de tokens (todos os tokens têm a mesma prioridade). Para um exemplo com mais de uma classe de tokens e prioridades, recomenda-se ver o exemplo existente em (MacDougall, 1987).

A geração das tarefas iniciais do sistema, representada pela linha *schedule()* na figura A.1, é feita na linha 85. No exemplo, a simulação possui uma tarefa inicial e novas tarefas são criadas no evento 1.

A linha 89 mostra a pré-computação do valor inicial que será utilizado como *lookahead* dinâmico. Esse valor será consumido na linha 110 e representa o tempo de serviço da CPU. Seu novo valor é calculado logo em seguida (linha 113) e será utilizado como *lookahead* na próxima iteração do ciclo da simulação (linha 95). Para utilizar *lookahead* dinâmico, o valor da variável *tempo_processo* sempre deverá ser definido dentro do ciclo de simulação, na linha imediatamente anterior à chamada à função *cause()*. Para o caso de *lookahead* estático, essa variável deve ser definida apenas uma vez, antes de se iniciar o ciclo de simulação.

Da linha 92 (figura A.5) até a linha 132 (figura A.6) está representado o ciclo da simulação. Esse ciclo irá se repetir até que uma condição de parada seja satisfeita (no exemplo, essa condição é o tempo simulado – linha 125). A estrutura do ciclo de simulação consiste, basicamente, na requisição (*request*) e liberação (*release*) de recursos. A ligação entre esses eventos é feita agendando-se os eventos seguintes (*schedule*).

A impressão dos resultados do mestre é feita nas linhas 139 a 148. Na figura A.1, considera-se que o processo escravo envia os resultados ao processo mestre. Isso pode ser feito utilizando funções do PVM, de modo similar ao que foi feito para enviar a tabela *tabela_tids[]* aos escravos, no início do programa de simulação.

```

116     case 3: /* libera CPU */
117         release("CPU", token);
118         /* decide se vai esvaziar o disco ou sair do sistema */
119         option = randompar (1,10000);
120         num_tours ++; /* incrementa a contagem do número de ciclos */
121         if (option > 6001)
122             schedule (b, 0.0, token, PL1);
123         /* verifica se chegou ao final da simulação
124         * Se chegou, avisa o processo filho para terminar */
125         if (clock_sim() > SIM_TIME)
126             schedule (1000, 0.0, token, PL1);
127         break;
128     case 1000: /* resposta ao comando de parada */
129         continua = 0;
130         break;
131     }
132 }
133
134 /* Coleta o tempo final da simulação e calcula o tempo gasto na simulação */
135 gettimeofday (&t2, (struct timezone*)0);
136 dt1 = (float) (t2.tv_sec - t1.tv_sec)
137       + (float) ((float) (t2.tv_usec - t1.tv_usec) / 1000000);
138
139 /* Imprime os resultados da simulação (somente os resultados da PL0) */
140 printf ("Num_tours \t Tempo de execução (Segundos) \t Tempo simulado\n");
141 printf (" \t mensagens completas enviadas \t mensagens completas recebidas\n");
142 printf (" \t mensagens nulas enviadas \t mensagens nulas recebidas \t Utilizacao\n");
143 printf (" \t Comprimento Medio Fila \t Período M. Ocupado\n");
144 printf ("%d \t %f \t %f \t %f \t %f \t %d \t %d \t %d \t %d \t %f \t %f \n",
145         num_tours, dt1, clock_sim(), cont_mens_completas_enviadas,
146         cont_mens_completas_recebidas, cont_mens_nulas_enviadas,
147         cont_mens_nulas_recebidas, utilizacao_recurso("CPU"),
148         comprimento_medio_fila("CPU"), periodo_medio_ocupado("CPU"));
149 /* Termina o parsmpl e sai do programa */
150 parsmpl_pvm_exit();
151 exit (0);
152 }

```

Figura A. 6: Codificação de PL0 (pl0.c - Parte 4 de 4)

A codificação do processo lógico PL1 (escravo) é semelhante à codificação do processo lógico mestre.

A recepção da tabela *tabela_tids[]*, enviada pelo processo mestre, é feita nas linhas 30 e 31.

```

1  * -----
2  * pl1.c "Processo Logico 1 - Escravo"
3  * ----- */
4  #include "parsmpl.h"
5
6  #define PL0 0      /* identificador do processo PL0 */
7  #define PL1 1      /* identificador do processo PL1 */
8  /* localização do arquivo com os resultados da simulação do escravo */
9  #define SAIDA "/home/naninha/modelo3/saida_PL1"
10
11 int main(void)
12 {
13
14     int
15         tid_processo_pai, /* task id do processo pai */
16         token,
17         fim = 0,          /* variavel de controle da simulação */
18         valor_semente,    /* semente para geracao de numeros aleatorios */
19         evento_execut;    /* próximo evento a ser executado */
20     double
21         tsDisco = 100,    /* tempo medio de utilizacao do disco */
22         valor_tsDisco;
23     /* arquivo para armazenar resultado da simulacao */
24     FILE *saida;
25
26     /* Associa os tids */
27     tid_processo_pai = pvm_parent();
28
29     /* Recebe do pai a tabela de tids */
30     pvm_recv(tid_processo_pai,-1);
31     pvm_upkint(tabela_tids,10,1);
32
33     /* inicio da simulacao */
34     smpl(0,"Processo Disco");
35
36     /* define a semente para geracao de numeros aleatorios */
37     stream (10);
38
39     /* inicia variaveis do ParSMPL */
40     proc_local = PL1;      /* define o numero do processo local */
41     clock_canais[PL0] = 0.0; /* inicia o canal de comunicacao utilizado */
42

```

Figura A. 7: Codificação de PL1 (pl1.c - Parte 1 de 2)

Note que na codificação do escravo não há chamada à função *schedule* antes do início do ciclo da simulação, como ocorre no processo mestre (figura A.5, linha 85). Isso se deve ao fato de o escravo não possuir tarefas iniciais no sistema: todas as tarefas virão do processo mestre.

```

43  * Declara a facilidade do processo *
44  facility("Disco",1);
45
46  valor_tsDisco = expntl (tsDisco);
47  while (!fim)
48  {
49      /* define o valor do lookahead do processo */
50      tempo_processo = valor_tsDisco;
51      /* busca o próximo evento a ser executado */
52      cause(&evento_execut, &token);
53      /* Verifica o tipo do evento e toma a ação correspondente */
54      switch(evento_execut)
55      {
56          case 5: /* reserva o disco */
57              if (request ("Disco", 5, token, 0, PL1) == 0)
58              {
59                  /* escala o fim da utilização do disco */
60                  schedule(6, valor_tsDisco, token, PL1);
61                  /* pre-computa o tempo de serviço do disco */
62                  valor_tsDisco = expntl (tsDisco);
63              }
64              break;
65          case 6: /* libera o disco */
66              release("Disco",token);
67              /* retorna para a CPU */
68              schedule(2, 0.0, token, PL0);
69              break;
70          case 1000: /* recebeu comando de parada da simulacao */
71              fim = 1;
72              schedule (1000, 0.0, token, PL0);
73              break;
74      }
75  }
76  saida = fopen (SAIDA, "a");
77  fprintf (saida, "\t Msg completas enviadas \t Msg completas recebidas
78  \t
79  Msg nulas enviadas \t Msg nulas recebidas \t Utilizacao (Disco) \t
80  Comprimento Medio da Fila (Disco) \t Período Medio Ocupado (Disco)\n");
81  fprintf (saida, "\t %ld \t %ld \t %ld \t %ld",
82  cont_mens_completas_enviadas, cont_mens_completas_recebidas,
83  cont_mens_nulas_enviadas, cont_mens_nulas_recebidas);
84  fprintf (saida, "\t %g \t %g \t %g \n", utilizacao_recurso("disco"),
85  comprimento_medio_fila("disco"), periodo_medio_ocupado("disco"));
86  fclose (saida);
87  parsmpl_pvm_exit();
88  exit (0);

```

Figura A. 8: Codificação de PL1 (pl1.c - Parte 2 de 2)

A impressão dos resultados (linhas 76 a 85) é feita em arquivo, diferente do processo mestre, que foi impressa na tela. Os resultados foram impressos em arquivo porque o processo escravo executa em *background* (segundo plano), de modo que a impressão de resultados na tela não é visível.

A.5. Compilação da biblioteca e dos programas de simulação

Para utilizar o ParSMPLX, é necessário primeiramente gerar os arquivos objetos *parsmpl.o*, que consiste na biblioteca ParSMPLX em si, e *randpar.o*, que contém as funções para geração de números aleatórios utilizados pelo ParSMPLX. Isso é feito utilizando um Makefile simples, como o exemplo mostrado na figura A.9, executado com o comando *make* do Linux.

Uma vez que o ParSMPLX foi compilado e os códigos objetos *parsmpl.o* e *randpar.o* foram gerados, esse procedimento não é mais necessário, a menos que haja modificações no código-fonte *parsmpl.c*.

```
PVMIDIR    =    $(PVM_ROOT)/include
PVMLDIR    =    $(PVM_ROOT)/lib/$(PVM_ARCH)
PVMLIB     =    -lgpvm3 -lpvm3 -lm
CFLAGS     =    -g -I$(PVMIDIR) -L$(PVMLDIR) $(ARCHCFLAGS) -c
BDIR       =    $(HOME)/parsmpl
XDIR       =    $(BDIR)/$(PVM_ARCH)
CC         =    gcc

all: parsmpl randpar

parsmpl: parsmpl.c
        $(CC) $(CFLAGS) parsmpl.c $(PVMLIB) -O3

randpar: randpar.c
        $(CC) $(CFLAGS) randpar.c $(PVMLIB) -O3
```

Figura A. 9: Makefile para ParSMPLX

Para compilar programas de simulação desenvolvidos com o ParSMPLX, é necessário criar um arquivo chamado *Makefile.aimk*. A figura A.10 mostra o exemplo de um *Makefile.aimk*, elaborado para compilar programas para simulação distribuída composto por dois processos lógicos: *pl0* e *pl1* (cujos arquivos fontes são *pl0.c* e *pl1.c*, respectivamente).

```

#
# Makefile.aimk for PVM example programs.
#
# Set PVM_ROOT to the path where PVM includes and libraries are installed.
# Set PVM_ARCH to your architecture type (SUN4, SPARC, SPARC64, etc.)
# Set ARCHLIB to any special libs needed on PVM_ARCH (-lipc, -lsocket, etc.)
# otherwise leave ARCHLIB blank
#
# PVM_ARCH and ARCHLIB are set for you if you use "PVM_ROOT/aimk"
# instead of "make".
#
# aimk also creates a $PVM_ARCH directory below this one and will cd to it
# before invoking make - this allows building in parallel on different arches.
#

SDIR    = .
BDIR    = $(SDIR)
XDIR    = $(BDIR)

CC       = gcc
OPTIONS  = -O
CFLAGS   = $(OPTIONS) -I/soft/pvm3/include $(ARCHCFLAGS)

LIBS     = /soft/pvm3/lib/LINUX/libpvm3.a
          /soft/pvm3/lib/LINUX/libpvm3.a /usr/lib/libm.so
GLIBS    =

#F77     = f77
FORT     = 'case "$FC" in f77) in *) echo $(FC) ;; *) echo $(F77) ;; *)
echo t77;; esac'
FFLAGS   = -g $(ARCHFFLAGS)
FLIBS    =

LFLAGS   = $(LOPT) -L/soft/pvm3/lib/$PVM_ARCH

CPROGS   = p10 p11
FPROGS   =

all: c-all f-all

c-all: $(CPROGS)
f-all: $(FPROGS)

clean:
rm -f *.o $(CPROGS) $(FPROGS)

$(XDIR):
mkdir $(BDIR)
mkdir $(XDIR)

p10: $(SDIR)/p10.c $(XDIR)
$(CC) $(CFLAGS) -o $@ $(SDIR)/p10.c ../randpar.o ../parsmpl.o $(LFLAGS)
$(LIBS)

p11: $(SDIR)/p11.c $(XDIR)
$(CC) $(CFLAGS) -o $@ $(SDIR)/p11.c ../randpar.o ../parsmpl.o $(LFLAGS)
$(LIBS)

```

Figura A. 10: Makefile.aimk, utilizado para compilar a simulação composta pelos programas p10.c e p11.c

O makefile.aimk é utilizado executando-se o comando *aimk*. Os códigos executáveis (no exemplo, os programas *p10* e *p11*) serão criados em um subdiretório chamado LINUX. Por exemplo, se os arquivos p10.c e p11.c foram compilados no diretório /home/parsmpl, os executáveis p10 e p11 serão criados no diretório /home/parsmpl/LINUX. Se o caminho (*path*) do programa escravo foi definido em um diretório diferente, é bom movê-lo para onde está

indicado no código-fonte. Caso contrário, ao tentar executar a simulação, o executável não será encontrado e o processo escravo não será criado, o que resultará em erro.

A.6. Execução dos programas de simulação

Para executar os programas de simulação, é necessário primeiro preparar o ambiente, adicionando as máquinas necessárias (mestre e escravos) à máquina virtual do PVM.

1. Ative o console PVM, digitando o comando `pvm`, conforme representado pela figura A.11.

Figura A. 11: Ativação do console do PVM

2. Adicione a(s) máquina(s) utilizando o comando `add <nome da máquina>` (figura A.12).

The screenshot shows a terminal window titled "1: lasdix.icmc.usp.br - default - SSH Secure Shell". The terminal content is as follows:

```
[naninha@lasdpc06 ~]$ pvm
pvm> add lasdpc12
add lasdpc12
1 successful
```

	HOST	DTID
	lasdpc12	80000

```
pvm> 
```

The status bar at the bottom of the window displays: "Connected to lasdix.icmc.usp.br", "SSH2 - 3des-cbc - hmac-md5 - none", "80x24", and "NUM".

Figura A. 12: Inclusão de máquinas à máquina virtual do PVM

3. Se desejar verificar quais máquinas compõem a máquina virtual do PVM, utilize o comando `conf`. Será feita uma listagem das máquinas incluídas à máquina virtual (figura A.13).

The screenshot shows an SSH terminal window titled "1: lasdix.icmc.usp.br - default - SSH Secure Shell". The user is logged in as "naninha@lasdpc06". The terminal shows the following commands and output:

```
[naninha@lasdpc06 ~]$ pvm
pvm> add lasdpc12
add lasdpc12
1 successful
```

HOST	DTID
lasdpc12	80000

```
pvm> conf
conf
2 hosts, 1 data format
```

HOST	DTID	ARCH	SPEED	DSIG
lasdpc06	40000	LINUX	1000	0x00408841
lasdpc12	80000	LINUX	1000	0x00408841

```
pvm>
```

The status bar at the bottom indicates "Connected to lasdix.icmc.usp.br", "SSH2 - 3des-cbc - hmac-md5 - none", "80x24", and "NUM".

Figura A. 13: Verificando a configuração da máquina virtual do PVM

4. Saia do console usando o comando `quit`. Irá aparecer uma mensagem indicando que o PVM continua executando (figura A.14).

```
[naninha@lasdpc06 ~]$ pvm
pvm> add lasdpc12
add lasdpc12
1 successful
      HOST    DTID
lasdpc12  80000

pvm> conf
conf
2 hosts, 1 data format
      HOST    DTID    ARCH    SPEED    DSIG
lasdpc06  40000    LINUX    1000  0x00408841
lasdpc12  80000    LINUX    1000  0x00408841

pvm> quit
quit

Console: exit handler called
pvmd still running.
[naninha@lasdpc06 ~]$
```

Figura A. 14: Ambiente preparado para a execução da simulação

5. Feito isso, basta disparar o programa mestre. O programa mestre executará na máquina local e irá criar os escravos nas máquinas especificadas no programa de simulação. Se as máquinas escravas não forem especificadas no programa de simulação, o PVM irá escalona-las nas máquinas presentes na máquina virtual utilizando o algoritmo *Round-Robin*.

Referências:

- (Bryant, 1977) BRYANT, R.E. *Simulation of Packet Communications Architecture Computer Systems*. Relatório Técnico. MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.
- (Chandy & Misra, 1979) CHANDY, K.M.; MISRA, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, v. SE-5, n.5, p.440-452, set. 1979.
- (Misra, 1986) MISRA, J. Distributed Discrete-Event Simulation. *Computing Surveys*, v. 18, n.1, p.39-65, mar. 1986.
- (MacDougall, 1987) MACDOUGALL, M.H. *Simulating Computer Systems: Techniques and Tools*. Cambridge : The MIT Press. 1987. 292p.

Apêndice B – Alterações Gerais no Gerador de Aplicações do ASiA

Encontram-se relacionadas neste apêndice as alterações que devem ser feitas em vários pontos do gerador de aplicações existente para o ASiA (Spolon, 1994) para possibilitar a geração de programas de simulação utilizando ParSMPLX. São pequenas modificações, necessárias devido a alterações em tipos de parâmetros e nomes de funções.

As alterações apresentadas neste apêndice complementam as que se encontram descritas no Capítulo 7, e devem ser utilizadas em conjunto ao aplicar as modificações no gerador de aplicações.

Para cada alteração, está indicada a ocorrência como aparece atualmente no gerador de aplicações do ASiA, o motivo pelo qual deve ser feita a alteração e a alteração em si.

B.1. Relação de Alterações

Ocorrência:	<i>inq</i>
Motivo	Função renomeada para <i>comprimento_fila</i> .
Alteração	Trocar ocorrências de “ <i>inq</i> ” no código do gerador do ASiA para “ <i>comprimento_fila</i> ”.

Ocorrência:

random

Motivo

Função renomeada para *randompar*.

Alteração

Trocar ocorrências de “*random*” no código do gerador do ASiA para “*randompar*”.

Ocorrência:

real

Motivo

Inexistência do tipo de dado *real* em C.

Alteração

Definir uma constante chamada *real* e associar ao tipo de dado *double* ou *float*, ou trocar as ocorrências de *real* por *double* ou *float*.

Ocorrência:

release

Motivo

Alteração no tipo do primeiro parâmetro da função.

Alteração

Nas ocorrências da função *release()*, trocar a variável “*Server*” pelo nome do recurso. Nas funções do gerador do ASiA onde ocorre criação de código do *release*, o nome do recurso deve ser passado como parâmetro.

Código original do gerador do ASiA:

```
Copy("      release(Server\0");  
strncat(Buffer,Aux1,strien(Aux1));
```

```
strcat(Buffer, ", Customer");};
Grava_Buffer();
```

Código alterado:

```
Copy("      release(\"");
strcat(Buffer, nome_recurso);
strcat(Buffer, "\", Customer");};
Grava_Buffer();
```

Ocorrência:

request

Motivo

Alterações no número e tipos dos parâmetros da função.

Alteração

Código original do gerador do ASiA:

```
Copy("      if (request(Server\0");
itoa(numero_recurso,Aux1,10);
strncat(Buffer,Aux1,strlen(Aux1));
strcat(Buffer, ",Customer,0) == 0");
Grava_Buffer();
```

Código alterado:

```
Copy("      if (request(\"");
strcat(Buffer, nome_recurso);
strcat(Buffer, "\", ");
itoa(Numero_Evento-1,Aux1,10);
strncat(Buffer,Aux1,strlen(Aux1));
strcat(Buffer, ",Customer, ");
itoa(prioridade,Aux2,10);
strncat(Buffer,Aux2,strlen(Aux2));
strcat(Buffer, ",");
itoa(processo_destino,Aux3,10);
strncat(Buffer,Aux3,strlen(Aux3));
strcat(Buffer, ") == 0");
Grava_Buffer();
```

Ocorrência:

schedule

Motivo

Inclusão de mais um parâmetro na função.

Alteração

Nas funções do gerador do ASiA onde ocorre criação de código do *schedule*, deve ser passado um parâmetro *processo_destino* (número inteiro) indicando em qual processo lógico o evento deve ser escalonado. Incluir como último parâmetro esse número do processo lógico em que o evento deve ser escalonado.

Código original do gerador do ASiA:

```
itoa(Numero_Evento,Aux1,10);
Copy("      schedule(\0");
strncat(Buffer,Aux1,strlen(Aux1));
strcat(Buffer," ");
// ...
strcat(Buffer,"Customer");
Grava_Buffer();
```

Código alterado:

```
itoa(Numero_Evento,Aux1,10);
Copy("      schedule(");
strncat(Buffer,Aux1,strlen(Aux1));
strcat(Buffer," ");
// ...
strcat(Buffer,"Customer,");
itoa(processo_destino,Aux2,10);
strncat(Buffer,Aux2,strlen(Aux2));
strcat(Buffer,");");
Grava_Buffer();
```

Ocorrência:

status

Motivo

Função não implementada no ParSMPL e ParSMPLX.

Alteração

Manter as partes do código do gerador onde aparecem chamadas à função “*status*” comentadas (ou seja, inativas) até a implementação dessa função.

Ocorrência:

facility

Motivo

Alteração no tipo retornado, de *int* para *void*.

Alteração

Deve-se eliminar a porção do código em que é feita a atribuição do resultado da função *facility()* a uma variável.

Código original do gerador do ASiA:

```
itoa(pclAux->ident_rec,Aux1,10);  
Copy(" Server\0");  
strncat(Buffer,Aux1,strlen(Aux1));  
strcat(Buffer," = facility(\"");
```

Código alterado:

```
// a preparação da variável que receberia o valor de facility e a  
// atribuição não são mais necessárias  
strcat(Buffer," facility(\"");
```

Ocorrência:

time

Motivo

Função renomeada por *clock_sim*.

Alteração

Trocar ocorrências de “*time*” no código do gerador do ASiA para “*clock_sim*”.

Referência:

(Spolon. 1994) SPOLON, R. *Um Gerador de Aplicação para um Ambiente de Simulação Automático*. 1994. 1997.109p. Dissertação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos.