d-smpl: Um Simulador Distribuído para Transputers*

Cláudia Maria Ribeiro Azevedo
Universidade Federal de Pernambuco
Departamento de Informática
claudia@di.ufpe.br

José Augusto Suruagy Monteiro
Universidade Federal de Pernambuco
Departamento de Informática
suruagy@di.ufpe.br

Sumário

Simulações de eventos raros, como é o caso da probabilidade de perda de células em Redes Digitais de Serviços Integrados de Faixa Larga (RDSI-FL) utilizando o Modo de Transferência Assíncrono (ATM), costumam levar muito tempo para serem executadas seqüencialmente num único processador. Nesses casos, a simulação distribuída se mostra como uma abordagem promissora para reduzir o tempo de execução dessas simulações.

Neste trabalho será apresentado um simulador distribuído baseado na linguagem de simulação smpl. Ele visa a obter um ganho em velocidade de execução em relação aos simuladores seqüenciais. Inicialmente será dada uma visão geral sobre simulação distribuída, enfocando suas duas principais classes de métodos. Em seguida será brevemente descrito o método de simulação utilizado na implementação do simulador. Serão também apresentados diversos aspectos do projeto e implementação de um simulador paralelo usando transputers e a filosofia C Paralelo/CSP (Communicating Sequential Processes). Por fim, serão ressaltadas as dificuldades práticas encontradas para se implementar um simulador numa rede de transputers comparadas com as encontradas num processador convencional.

Abstract

The simulation of rare events, as it is the case of cell losses in Broadband Integrated Service Digital Networks (B-ISDN) using Asynchronous Transfer Mode (ATM), usually takes a long time to be executed sequentially on a single processor. In these

*Este trabalho foi desenvolvido com o apoio da FACEPE e do CNPq.

cases, distributed simulation appears to be a promising approach in order to reduce execution times.

In this paper, a distributed simulator based on the simulation language smpl is presented. It is intended to execute faster than sequential simulators. Initially, we make an overview of distributed simulation, emphasizing its two main method classes. Afterwards, the method used to implement the simulator is described briefly. Many design and implementation aspects of a parallel simulator using transputers and the C Parallel/CSP (Communicating Sequential Processes) phylosophy are presented. Finally, the practical difficulties faced during the implementation of a transputer based simulator are compared to those faced in sequential ones.

1 Introdução

Graças à evolução tecnológica, os computadores SISD (*Single Instruction Single Data*) obtiveram um grande aumento em velocidade de execução. Entretanto, há barreiras físicas que limitam a velocidade dessas máquinas [Rat85]. Além disso, existem problemas importantes que podem não requerer essa quantidade enorme de velocidade computacional, mas que são essencialmente paralelos, sendo mais naturalmente modelados e implementados como tais. ("Concorrência é um aspecto fundamental da natureza" [Rat85]).

Segundo Kleinrock [Kle90], "O desafio de projetar o sistema operacional para computadores de tempo compartilhado foi o principal tema nos anos 60, o projeto e acesso de redes de longa distância foi nosso interesse na década de 70, o projeto de redes locais foi nosso foco nos anos 80 e nós prevemos que o problema genérico de processamento distribuído será certamente o alvo de nossas atenções na década de 90 que está chegando".

Por outro lado, há inúmeros tipos de situações em que um tratamento analítico torna-se quase que impraticável, tendo que se recorrer, então, a soluções por simulação. Ou, em muitos casos, usa-se simulação como forma de validar os modelos analíticos. Simulações de eventos raros, como é o caso da probabilidade de perda de células em redes ATM (RDSI-FL), costumam levar muito tempo para serem executadas seqüencialmente num único processador. Nesses casos, a simulação distribuída se mostra como uma abordagem promissora para reduzir o tempo de execução dessas simulações [SX92, BT91, Fuj88]. No entanto, muito trabalho precisa ser feito para determinar a validade dessa promessa.

Os benefícios de distribuir a simulação dependerão da aplicação, do sistema de multiprocessador disponível e da abordagem (método de simulação distribuída) usada. A melhor abordagem, por sua vez, depende da aplicação e da máquina disponível. Muitos algoritmos de simulação distribuída têm sido propostos, mas muito pouco tem sido feito em termos de avaliação de desempenho.

Estudos empíricos e analíticos precisam ser realizados para determinar os méritos relativos das diferentes abordagens, quais são as melhores e sob que circunstâncias. São necessárias mais ferramentas para medir empiricamente o desempenho de uma simulação. É como foi afirmado em [Fuj90] : "O grau em que as

técnicas aqui descritas podem ser aplicadas para paralelizar simulações arbitrárias está apenas começando a ser explorado".

Com o surgimento de diversas arquiteturas paralelas, os trabalhos na área de simulação distribuída puderam adquirir um teor mais pragmático. A arquitetura do sistema de processamento, em particular o número de processadores, a quantidade de memória, e se a memória é compartilhada ou não, é também um ponto determinante na abordagem usada para simulação distribuída.

Um outro fator importante para o desempenho de simulação distribuída, que também será considerado neste trabalho, é a alocação de tarefas ou processos em processadores. Uma alocação não-balanceada pode degradar seriamente o desempenho da simulação, pois ter-se-ia alguns processadores sub-utilizados, enquanto que outros estariam sobrecarregados.

O propósito deste projeto é exatamente este: através de implementações de simulações distribuídas, avaliar os benefícios que essas podem trazer, seja em velocidade de execução e/ou em facilidade de expressão do problema a ser simulado. Essa avaliação será baseada em comparações com resultados de simulações seqüenciais.

O restante do artigo é organizado como segue. Na seção 2 é dada uma visão geral sobre simulação distribuída, destacando-se as suas duas principais classes de métodos. Na seção 3 é apresentado brevemente o método CMB, utilizado na implementação do simulador. Na seção 4 é discutido o ambiente, *C Paralelo* e *transputers*, usado na implementação do simulador. Na seção 5 o simulador *d-smpl* é apresentado e, por fim, na seção 6 são tecidas considerações finais sobre o trabalho.

2 Simulação Distribuída

Historicamente, duas das principais técnicas para modelar sistemas têm sido teoria das filas e simulação de eventos discretos. Quando efetivas, as técnicas baseadas em teoria das filas podem prover rapidamente uma compreensão matemática do comportamento dos sistemas, permitindo a consideração de uma ampla gama de valores de parâmetros desses sistemas. A maior limitação dessas técnicas é o grande número de suposições restritivas que devem ser satisfeitas para garantir a exatidão.

Por outro lado, os modelos de simulação podem imitar um sistema do mundo real tão rigorosamente quanto a compreensão permita e as necessidades requeiram. Entretanto, os modelos de simulação altamente detalhados são normalmente muito carregados, computacionalmente falando, podendo-se recorrer, então, a simulações distribuídas como um possível caminho para tornar mais rápidas as simulações.

Sabe-se também que simulações são aplicações extremamente difíceis de se paralelizar, devido à natureza irregular da dependência de dados de seus programas[Fuj90]. São problemas onde técnicas de vetorização (como as usadas no cálculo do produto escalar) usando supercomputadores trazem pouco benefício.

Embora a simulação distribuída venha se mostrando como uma excelente al-

ternativa para minimizar o tempo de execução das simulações, devido a algumas limitações intrínsecas às aplicações, o seu uso pode não ser muito vantajoso em todas as áreas.

A simulação de arquitetura de processadores, por exemplo, reflete a execução de um ciclo de instrução: busca/decodifica/executa e é, por natureza, seqüencial; a paralelização desta aplicação requer pesquisa na área de arquitetura. Já a simulação de circuitos lógicos, embora claramente tratável por processamento paralelo [Pfi82], envolve usualmente ativação síncrona de muitas entidades, dificultando a paralelização do algoritmo. Ao contrário, simulação de redes é tipicamente assíncrona, sendo uma das áreas que mais pode se beneficiar dos mecanismos de simulação distribuída.

Nos últimos anos, grande parte dos trabalhos desenvolvidos em simulação tratam sobre simulação distribuída. Um histórico muito completo desses trabalhos pode ser encontrado em [Fuj90].

Simulação distribuída de um sistema é uma simulação em que o simulador é um sistema distribuído de processos; onde esses processos comunicam-se entre si apenas através de mensagens. Esta discussão ignora as anomalias que podem surgir na implementação dessas construções numa arquitetura específica. Por exemplo, foi assumido que os processos se comunicam apenas através de mensagens. Entretanto, numa implementação em máquina de memória compartilhada, os processos também podem compartilhar informações através de variáveis globais.

A maioria dos esquemas de simulação distribuída usa a seguinte estrutura básica: cada processo simula um sub-sistema, isto é, parte do sistema completo que está sendo simulado. Qualquer processo desses, quando simulando um evento ocorrendo no seu sub-sistema, envia mensagens aos outros processos (que estão simulando os outros sub-sistemas) que são afetados pelo processamento do primeiro evento. A qualquer instante durante a simulação, processos distintos podem ter simulado seus sub-sistemas até tempos de simulação diferentes.

Nenhum relógio de simulação global é mantido; a sincronização entre os processos é possível graças à inclusão, em cada mensagem de evento, do tempo de ocorrência deste, juntamente com outras descrições do evento. A motivação por trás da simulação distribuída é obter um ganho em velocidade em relação à simulação seqüencial tradicional os grupos formados pelos processos podem ser executados paralelamente em processadores distintos de um sistema distribuído, resultando potencialmente num ganho em relação à simulação seqüencial.

As diversas estratégias de simulação paralela são divididas, basicamente, em dois grandes grupos: métodos "conservativos" [Mis86] e métodos "otimistas" [JS85] (esses termos têm sido adotados na literatura de simulação [Fuj90]).

Nos métodos otimistas, os eventos são processados à medida que chegam a um processo, esperando-se, com otimismo, que eles estejam em ordem não decrescente de suas marcas de tempo. Se um evento chegar fora dessa ordem, o processo, então, volta ao estado anterior ao que essa mensagem deveria ser tratada, processa essa mensagem na sua ordem correta e reprocessa todas as outras mensagens que foram processadas fora de ordem. Esta abordagem assume que a informação dos estados

é salva periodicamente para que seja possível voltar atrás.

Nos métodos conservativos, um processo só trata um evento (uma mensagem) com uma certa marca de tempo se puder garantir que não chegará depois nenhum outro evento com uma marca de tempo menor que esta. Enquanto essa garantia não existe, o processo fica bloqueado. Se esses bloqueios ocorrerem em ciclos, pode-se chegar a situações de impasse (*deadlock*). Para evitar tais situações é utilizada a técnica de envio de mensagens nulas. Esta técnica ficou conhecida por CMB, graças aos seus criadores: Chandy, Misra e Bryant.

3 O Método CMB

Independentemente, Chandy e Misra [CM79], e Bryant [Bry77] desenvolveram alguns dos primeiros algoritmos de simulação paralela. Essa abordagem exige que se especifique estaticamente os *links* que indicam que processos podem se comunicar com que outros processos. Para se determinar quando é seguro processar uma mensagem, é necessário que a seqüência das marcas de tempo das mensagens enviadas por um *link* seja não decrescente. Isso garante que a marca de tempo da última mensagem recebida por um *link* de entrada seja o limite mínimo da marca de tempo de qualquer mensagem subseqüente que será recebida por este mesmo link.

As mensagens que chegam em cada *link* de entrada são armazenadas em ordem FIFO (*first-in, first-out*), que é também a ordem das marcas de tempo, pela restrição acima. Cada *link* tem um *clock* associado a ele, cujo valor depende do estado da fila do seguinte modo:

- Se a fila contém pelo menos uma mensagem:

  o relógio da porta de entrada será igual à marca de tempo da mensagem da cabeça da fila.
  Caso contrário,

- Se a fila está vazia:

  será igual à marca de tempo da última mensagem recebida

O processo repetidamente seleciona o *link* com o menor relógio e, se há uma mensagem naquela fila do *link*, a mensagem é processada. Caso contrário (fila vazia), o processo fica bloqueado. Este protocolo garante que cada processo tratará os eventos somente em ordem não decrescente das marcas de tempo.

Se surgir um ciclo de filas vazias que tenha marcas de tempo suficientemente pequenos, cada processo nesse ciclo ficaria bloqueado, e a simulação entraria num impasse. A figura 1 mostra uma dessas situações. Cada processo fica indefinidamente esperando uma mensagem no *link* de entrada que contém o menor valor de relógio, já que a fila correspondente está vazia. Todos os três processos estão bloqueados, embora existam mensagens em outras filas de entrada de cada processo que estão esperando para serem processadas.

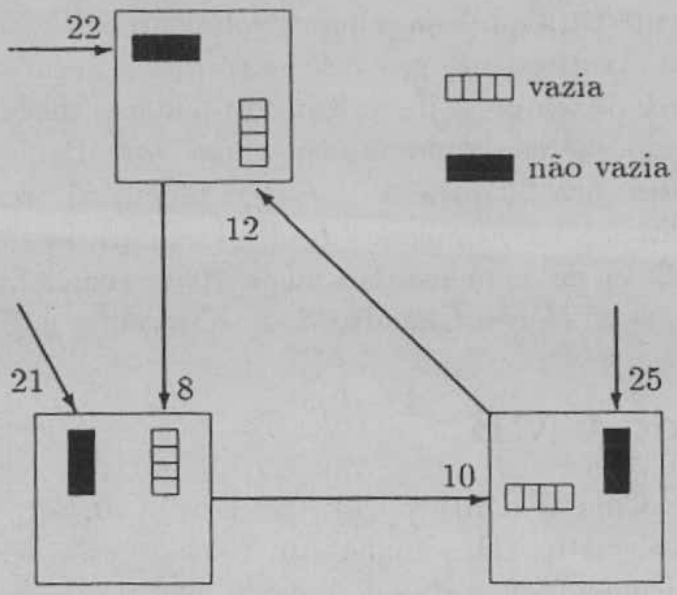


Figura 1: Situação de Impasse.

O impasse ocorre muito freqüentemente se:

- Existirem poucas mensagens não-processadas comparado com o número de *links* na rede.
- Os eventos não-processados ficarem agrupados numa porção da rede.

Para que a simulação não entre num impasse, mensagens nulas são utilizadas. Mensagens nulas são usadas apenas com o propósito de sincronização, e não correspondem a nenhuma atividade no sistema físico. Uma mensagem nula com marca de tempo T enviada do processo A para o processo B é uma promessa do processo A de que não mandará uma mensagem para o processo B com uma marca de tempo inferior a T.

Como um processo determina as marcas de tempo das mensagens nulas que ele envia?

- Pelas marcas de tempo dos seus *links* de entrada, que fornecem um limite mínimo à marca de tempo do próximo evento não-processado que será removido do *buffer* do *link* de entrada, e,
- Pelo tempo de processamento de uma mensagem (um incremento mínimo na marca de tempo para qualquer mensagem que passe através do processo lógico).

Quando um processo termina de executar um evento, ele manda uma mensagem nula por cada uma de suas portas de saída indicando o limite mínimo; o receptor da mensagem nula pode então computar novos limites para os seus *links* de saída, enviar esta informação para os seus vizinhos, e assim por diante. É dever do programador da aplicação determinar as marcas de tempo atribuídas às mensagens nulas.

3.1 Discussão sobre o Método de Prevenção do Deadlock

Pode ser mostrado que este mecanismo evita o impasse, desde que não se tenha ciclos em que o incremento coletivo da marca de tempo de uma mensagem que atravessa um ciclo possa ser zero. Isso implica que certos tipos de simulação não podem ser realizadas por este método. Por exemplo, simulação de redes de filas em que o tempo de serviço mínimo para tarefas que passam através de um servidor seja zero.

É interessante notar também que, neste método, a simulação não entra num impasse mesmo que o sistema físico entre num impasse. Se isso ocorrer, o simulador continua sua computação transmitindo mensagens nulas com valores incrementados de T. Isso simula corretamente a situação física correspondente, em que, enquanto o tempo progride, nenhuma mensagem real é transmitida no sistema físico.

A simplicidade deste esquema é um de seus pontos mais atraentes. Ele necessita de pequenas mudanças na codificação de uma implementação distribuída original, sujeita a impasses, para que passe a enviar mensagens nulas.

Estudos empíricos mostraram que este esquema é bastante eficiente para redes acíclicas [See79]. Segundo [Mis86], muitos fatores podem afetar a eficiência em redes mais gerais, entre eles os mais importantes são:

1. **O grau de ramificação da rede**

   Considere uma rede com uma fonte e um sorvedouro. O número de caminhos distintos entre a fonte e o sorvedouro é uma medida (grosseira) da quantidade de ramificações da rede. As mensagens nulas tendem a ser criadas nas ramificações e podem se proliferar por todos os ramos sucessivos. Um nó de bifurcação recebe um único fluxo de mensagens de entrada e distribui este fluxo para N saídas. Após receber uma mensagem de entrada real ou nula, um nó de bifurcação direciona a mensagem à saída selecionada e cria $N-1$ mensagens nulas, cada uma roteada a um dos destinos não selecionados. Então, pode-se esperar que, quanto menor o número de ramos, menor a quantidade de mensagens nulas e melhor o desempenho. Estudos empíricos confirmam isso [See79]. Teoricamente, a eficiência ótima é alcançada numa rede de filas em série (*tandem*), e resultados excelentes são obtidos em redes com poucas ramificações. Em geral, redes acíclicas exibem níveis de desempenho razoavelmente bons.

   Peacock *et al.* [PWM79b, PWM79a] realizaram experimentos levando em conta várias topologias de rede. Suas conclusões foram: "Para algumas topologias de modelos de redes de filas, esta abordagem resulta num aumento de velocidade (*speedup*) no tempo total para completar uma simulação. Entretanto, para outras topologias, especialmente as que contêm laços, o aumento de velocidade pode não ser significativo."

2. **Mecanismos de time-out para prevenir a transmissão de mensagens nulas**

   Claramente, uma das desvantagens do método de Prevenção do Deadlock usando mensagens nulas é a abundância de mensagens nulas. Uma possível variação deste método baseia-se na constatação de que uma mensagem nula não tem muito efeito se for imediatamente seguida de outra mensagem nula. Então, pode ser eficiente retardar as transmissões de mensagens nulas com a esperança de que futuras mensagens recebidas por um processo tornem desnecessárias as suas transmissões. "A quantidade de tempo '*q*' que um processo espera antes de transmitir uma mensagem nula é um fator muito importante, embora ainda não se tenha realizado estudos empíricos sobre isso" [Mis86].

3. **Quantidade de armazenamento nos links**

   O número de espaços de *buffer* nos *links* pode ter efeitos substanciais no desempenho. Quando o número de espaços de *buffer* é reduzido a zero, os remetentes têm que esperar até que os receptores estejam prontos para receber, e uma quantidade de tempo considerável é gasta esperando. O número de espaços de *buffer* foi então incrementado e a seguinte regra foi usada para aniquilar mensagens nulas: Quando uma mensagem qualquer chega no *buffer*, ela deve eliminar todas as mensagens nulas que se encontram na sua frente no *buffer*, já que elas estão obsoletas, e não há mais razão para que sejam processadas: seria apenas um gasto de tempo desnecessário do processo.

   Foi descoberto que, na simulação de uma certa classe de redes de filas, o desempenho aumentou rapidamente até que o número de espaços de *buffer* num *link* atingiu 10, incrementou menos rapidamente até 20, e o desempenho permaneceu essencialmente inalterado desde então. No entanto, esse números não podem ser aplicados diretamente a outros problemas; espera-se que esse números sejam dependentes do tipo de problema e da velocidade dos processadores e das linhas de transmissão.

4 Descrição do Ambiente

O objetivo desta seção é prover um melhor entendimento do ambiente utilizado. Primeiro serão vistos os aspectos de *hardware* e em seguida os de *software*. Por fim, será feita uma breve análise de como este ambiente se comporta para realizar implementações de simulação distribuída. Alguns dos pontos abordados nessa análise serão: o mecanismo de passagem de mensagens, a alocação estática de processos, e as limitações de entrada e saída dos *transputers*. Serão analisados também os problemas no funcionamento de algumas funções do *C Paralelo*, a escolha da linguagem *C Paralelo* ao invés de *Occam* para implementar o simulador, entre outros.

4.1 Os Transputers

Os *transputers* foram desenvolvidos pela INMOS Corporation com a finalidade de implementar em *hardware* o modelo de concorrência e comunicação de Occam. Sua arquitetura é do tipo RISC (Reduced Instruction Set Computer) e, devido às facilidades da tecnologia VLSI, este componente funciona como um microcomputador, já que possui memória RAM, quatro *links* seriais para comunicação, uma unidade processadora, além de, em alguns casos, dispor de interfaces especiais para periféricos, unidade aritmética de ponto flutuante, etc.

A arquitetura foi projetada para permitir que uma rede de componentes programáveis tenha qualquer topologia desejada (regular, cubo, rede de Peterson, etc.), limitada apenas pelo número de *links* em cada *transputer*.

A troca de mensagens entre processadores é feita através de seus *links* seriais, que executam comunicação bidirecional, *full-duplex*, ponto-a-ponto. A taxa padrão de transmissão é de 10 Mbit/s, embora os mais modernos possam ser configurados para taxas mais altas, como o T800 (20 Mbit/s).

O *transputer*, desde seu surgimento, está tendo seu principal uso em sistemas dedicados, inclusive como co-processador para sistemas de comunicação ou mesmo para execução de operações de ponto flutuante, sendo que esta última aplicação é devida principalmente ao surgimento do T800, que processa 1,5 MFLOPS com clock de 20 Mhz e 2,25 MFLOPS com clock de 30 Mhz. Já o T414 processa 10 MIPS a 20 Mhz.[1]

A rede de *transputers* do DI UFPE é composta de uma placa B008 contendo um *transputer* T800 e de uma placa B003 com quatro *transputers* T414 (ambas, placas da INMOS). A configuração da rede pode ser vista na figura 2.

Tanto o T800 como o T414 são processadores de 32 bits, sendo que o T800 tem uma unidade de ponto flutuante de 64 bits. Cada T414 tem uma memória RAM de 256 KB mais 2 KB de memória *on-chip*, resultando em aproximadamente 1 MB na placa B003. Já o T800 dispõe de 2 MB de RAM e mais 4 KB de memória *on-chip*.

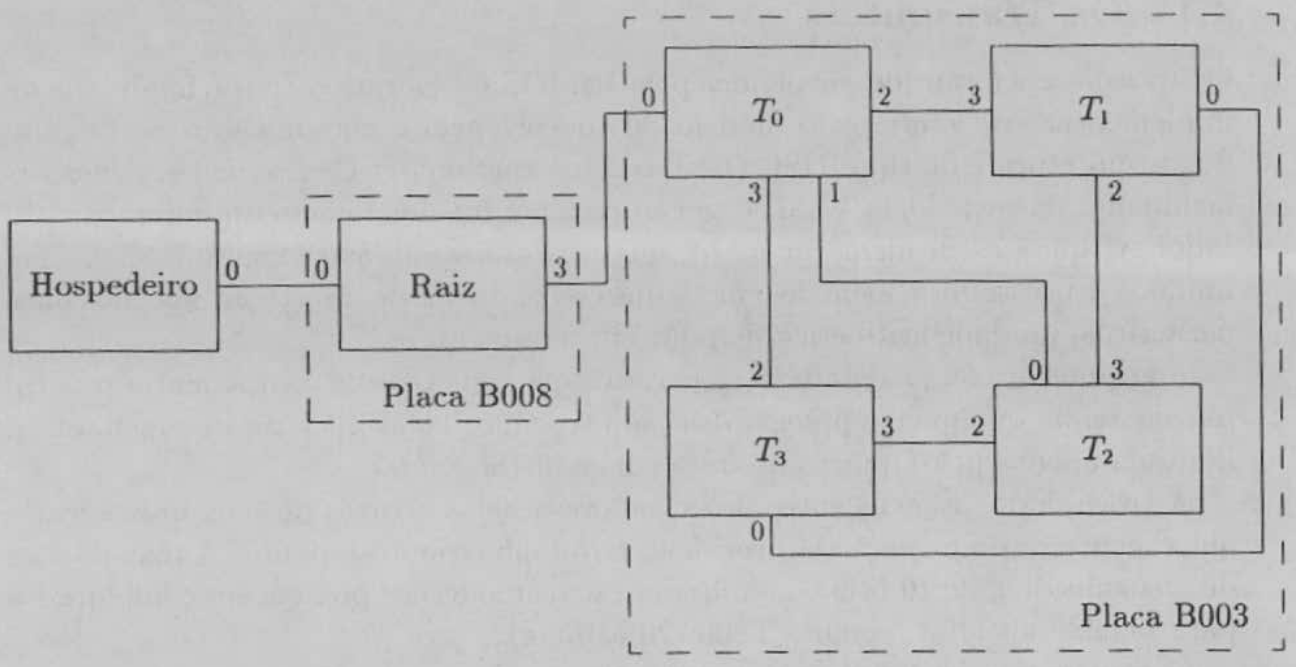
Na placa B003 todos os *tranputers* foram ligados entre si, evitando problemas de roteamento. Desse modo, não é necessário passar na mensagem um parâmetro de endereço, toda mensagem que chega a um *transputer* é para ele mesmo. Apenas para o *transputer* T0, que se comunica com o T800, pode haver diferença nesse aspecto do tratamento das mensagens.

4.2 C Paralelo

O *C Paralelo*, assim como o *hardware* do *transputer*, é baseado no modelo abstrato de concorrência de *Communicating Sequential Processes* (CSP)[Hoa78].

Uma aplicação completa é vista como uma coleção de uma ou mais tarefas que executam concorrentemente. Cada tarefa tem sua própria região de memória para

[1] Para efeito de comparação, a SPARCstation SLC a 20 Mhz processa 1,2 MFLOPS e 12,5 MIPS e a SPARCstation 2 a 40 Mhz processa 4,2 MFLOPS e 28,5 MIPS, ambas arquiteturas RISC.



Raiz:	Transputer T800
T_0 - T_3:	Transputers T414
Hospedeiro:	Intel 80286

Figura 2: Ligação Física dos Transputers.

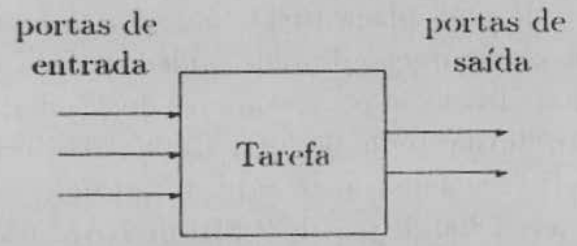


Figura 3: Uma tarefa vista como uma "caixa preta".

código e dados, um vetor de portas de entrada, e um vetor de portas de saída. Os vetores de portas são passados à tarefa como argumentos da função `main`.

As tarefas podem ser vistas como "caixas pretas" de *software*, conectadas entre si através de suas portas, como mostrado na fig. 3, e podem também ser tratadas como blocos atômicos para a construção de sistemas paralelos.

Cada elemento nos vetores de portas de entrada e de saída é do tipo "apontador para um canal de palavra" (`CHAN *`). As portas são ligadas aos endereços reais dos canais através de um *software* de configuração externo à própria tarefa; as ligações podem ser trocadas sem precisar recompilar ou religar (através do `ilink`) a tarefa.

O *software* de configuração também provê modos de se especificar que tarefas de *software* serão executadas em que processadores de *hardware*. Cada processador pode executar qualquer número de tarefas, limitado apenas pela disponibilidade de memória.

As tarefas alocadas no mesmo processador podem ter qualquer quantidade de canais de interconexão. As tarefas alocadas em processadores distintos só podem ser conectadas quando fios físicos conectarem os *links* dos processadores. A qualquer conexão lógica entre duas tarefas alocadas em processadores diferentes, é atribuído o uso exclusivo de um dos canais dos *links* físicos que conectam os processadores. O número de interconexões entre as tarefas em processadores diferentes é, então, limitada pelo número de *links* físicos que cada um tenha. Se forem necessárias mais de quatro conexões lógicas em cada sentido entre um *transputer* e os seus vizinhos, o projetista do sistema deve prover explicitamente tarefas multiplexadoras.

O *C Paralelo* também provê os mecanismos para criar dinamicamente novos fluxos concorrentes de execução numa mesma tarefa. Cada fluxo tem sua própria pilha, alocada pelo seu criador, mas compartilha seu código, dados estáticos e espaços de memória *heap* com outros fluxos na mesma tarefa. Existem, na biblioteca de *run-time*, funções que lidam com semáforos para controlar o acesso aos dados e canais compartilhados. Esses fluxos podem se comunicar tanto por canais como por dados compartilhados.

Um programa que utiliza vários *transputers* consiste de uma rede de tarefas que se comunicam e que estão distribuídas numa rede física de *transputers*. Nesse caso, um arquivo de configuração deve ser criado, que descreve:

- os *transputers* na rede física e como eles estão configurados.
- os nomes das tarefas e como elas estão conectadas.
- a alocação das tarefas nos *transputers*.

Os programas que utilizam vários *transputers* são construídos em dois estágios. Primeiro, cada tarefa individual é submetida ao compilador, ao `ilink` e depois é convertida numa tarefa executável. Logo após, o configurador é utilizado para criar o programa final. O configurador usa como entrada o arquivo de configuração e o código executável das tarefas e gera como saída um programa que pode ser carregado pelo servidor na rede de *transputers*.

4.3 Análise da adequabilidade do ambiente

Nesta seção será feita uma análise do ambiente utilizado (*C Paralelo* e *transputers*), tanto sob o aspecto do *hardware* como do *software*.

Este simulador foi implementado em *C Paralelo* e sua infra-estrutura desenvolvida de forma independente da aplicação em questão. Segundo [Mis86], a implementação de simulação distribuída é possível em qualquer linguagem que permita a criação de processos que se comuniquem por mensagens e o modelo de concorrência do *C Paralelo* apresenta exatamente esta característica.

O fato do mecanismo de comunicação dos *transputers* ser através de passagem de mensagens torna mais simples e natural a implementação dos métodos de

simulação distribuída, já que esses métodos foram desenvolvidos baseando-se na troca de mensagens entre os diferentes processos.

Quando esses métodos são implementados em sistemas de memória compartilhada, faz-se necessária uma atenção especial com a sincronização de filas de mensagens compartilhadas e com o gerenciamento de situações de impasse. Numa implementação de simulação distribuída em memória compartilhada, todas as informações de estado das tarefas, inclusive filas de mensagens de entrada, residem na memória compartilhada. As comunicações baseadas em mensagens entre as tarefas são implementadas através do acesso compartilhado às filas de mensagens de cada tarefa. Cada fila de mensagem é protegida por um semáforo, para garantir a exclusão mútua

Uma outra questão relevante é que numa rede de *tranputers* a alocação das tarefas nos processadores é estática, ou seja, permanece inalterada durante toda a simulação. Do contrário ter-se-ia uma alocação dinâmica, onde as tarefas são alocadas nos processadores durante a simulação, visando principalmente a minimizar a ociosidade dos processadores.

A alocação estática pode ser vista como uma limitação dos *tranputers*, como, por exemplo, se num determinado programa de simulação composto por várias tarefas algumas dessas terminarem a sua execução antes das demais. É possível que alguns dos processadores sejam liberados e, no entanto, não se possa alocar neles algumas tarefas que ainda estão precisando ser processadas.

Uma outra limitação do ambiente é que apenas uma tarefa pode realizar operações de entrada e saída. Uma `iotask` é uma tarefa que realiza operações de E/S com o sistema de arquivos do hospedeiro, por exemplo, usando as funções padrão de E/S `getchar`, `printf` etc. Apenas a tarefa `iotask` pode usar funcões como o `printf`, que requer acesso à máquina hospedeira. Todas as outras tarefas, chamadas de "tarefas de computação", podem realizar entrada e saída apenas através de suas portas.

A tarefa que realiza E/S é ligada (pelo `ilink`) com a biblioteca completa de *run-time* `crtl.lib`. Todas as outras tarefas de computação devem ser ligadas com a biblioteca reduzida de *run-time* `sacrtl.lib` (*standard*).

Essa restrição cria um obstáculo porque apenas uma tarefa pode, por exemplo, ler do teclado, escrever na tela e/ou em arquivos, dificultando bastante o processo de desenvolvimento do simulador.

Torna-se um trabalho extremamente árduo depurar algumas tarefas, já que essas não podem escrever diretamente na tela. A única maneira é enviar mensagens à tarefa de E/S para que esta escreva na tela. Mas, e se a falha ou o erro estiver justamente na comunicação entre as tarefas? E se, por algum motivo, as tarefas estiverem num impasse e incomunicáveis? É quase que impraticável acompanhar a execução dessas tarefas através da tela.

A idéia de que todas as tarefas para utilizarem qualquer função de E/S precisem enviar mensagens para uma só tarefa, para que esta desempenhe essas funções, já sugere a existência de uma gargalo, podendo depreciar sensivelmente o desempenho da rede.

Um outro problema encontrado foi que, na versão usada [2], algumas funções importantes do *C Paralelo* não executam segundo o manual. É o caso da função `thread-create`, que serve para criar vários fluxos distintos de execução numa mesma tarefa. A função se propõe a criar "n" fluxos, inclusive contém um exemplo ilustrativo no próprio manual, mas na realidade só funciona corretamente para se criar 1 fluxo adicional, além do principal.

Esse tipo de falha trouxe problemas ao desenvolvimento, já que é projetado um sistema partindo-se do pré-suposto de que as funções descritas no manual funcionam e, após longos e exaustivos testes, conclui-se que isso não é bem verdade. Além do enorme tempo gasto, necessita-se de reformular os programas.

O fato de estar se trabalhando com *transputers* normalmente levanta a questão se não seria melhor utilizar *Occam*, ao invés do *C Paralelo*. Apesar deste ponto ser bastante polêmico, far-se-á aqui uma abordagem resumida.

A princípio, vale mencionar que, afora qualquer discussão sobre os méritos relativos das duas linguagens, no caso do simulador em questão, a "opção" foi forçada por basicamente dois motivos: ir-se-ia fazer uma extensão distribuída à linguagem de simulação *smpl*, que é toda escrita em *C*. Então, nada mais natural do que aproveitar boa parte de seu código, utilizando-se para isso o *C Paralelo*. No caso de *Occam*, ter-se-ia que reescrever todas as funções do *smpl*. O outro motivo é que o *hardware* usado foi configurado para executar *C Paralelo* e não *Occam*. Somente hoje em dia é que se encontra disponível, no laboratório do DI-UFPE, uma chave que permite utilizar uma ou outra linguagem.

A portabilidade da linguagem *C* é um dos grandes méritos do *C Paralelo*. Se, para utilizar o simulador distribuído, fosse necessário aprender uma nova linguagem, no caso *Occam*, é possível que se desistisse *a priori*. Por outro lado, devido ao vasto uso da linguagem *C* em diferentes áreas, é muito mais provável que haja interesse em se utilizar um simulador onde se possa aproveitar grande parte do código de simulação em *C* que já estiver pronto.

Outro ponto positivo do *C Paralelo* em relação a *Occam* é o fato do primeiro ser uma linguagem dinâmica, enquanto que o segundo é estática. *Occam* não permite recursão nem estruturas dinâmicas de dados.

Por outro lado, *Occam* foi uma linguagem já concebida como paralela, enquanto que ao *C* foram incorporadas algumas primitivas básicas de paralelismo, gerando o *C Paralelo*. Com isso, existem em *Occam* algumas funções essenciais à programação paralela para as quais não há equivalente nesta versão do *C Paralelo*, como é o caso da função `ALT`. No *C Paralelo* o desenvolvimento dessas funções fica a cargo do programador.

Como os *transputers* foram projetados para executar *Occam* e vice-versa, espera-se que *Occam* seja mais eficiente que o *C Paralelo*.

[2] 3L Parallel C IMS D711

5 Implementação do Simulador

Nesta seção será explicado como o mecanismo conservativo de simulação distribuída CMB, também conhecido como Método de Prevenção do *Deadlock*, foi implementado no *C Paralelo*. Inicialmente será feita uma breve introdução à linguagem de simulação *smpl*. Logo após, serão discutidas as alterações realizadas no *smpl* para torná-lo um *smpl* distribuído, denominado *d-smpl* (*distributed smpl*), permitindo usá-lo para simular modelos paralelamente.

Não é do escopo deste trabalho abordar alguns temas tradicionais em simulação, como: geração de números aleatórios, análise estatística das saídas, etc. Os métodos desenvolvidos nessas áreas para simulação seqüencial ainda se aplicam. Assim, esta seção enfocará principalmente a implementação da parte distribuída do simulador, sendo a parte seqüencial adaptada do *smpl* (Simulation Programming Language) [Mac87], como é o caso das funções que lidam com os recursos (*facilities*). Desse modo, o simulador também pode ser visto como uma extensão distribuída à linguagem de simulação *smpl*.

5.1 O smpl

O *smpl* é uma extensão funcional de uma linguagem de programação de propósito geral, chamada de linguagem hospedeira. Essa extensão consiste de um conjunto de funções de biblioteca (o sub-sistema de simulação *smpl*) que, juntamente com a linguagem hospedeira, compõe uma linguagem de simulação orientada a eventos. Um modelo de simulação em *smpl* é implementado como um programa da linguagem hospedeira: as operações de simulação são realizadas através de chamadas às funções do sub-sistema de simulação.

Existem versões de *smpl* implementadas em diversas linguagens, dentre elas, *Basic*, *C*, *Fortran*, *Pascal* e *PL/I*. A versão usada neste trabalho foi implementada em *C*.

A única função do *smpl* dependente de máquina é a *ranf()* (geradora de números aleatórios), que já foi devidamente alterada para ser executada nos transputers.

O arquivo *smpl.c* contém 37 funções, de onde 26 podem ser chamadas por um programa de simulação. Essas funções estão divididas em basicamente 6 grupos: inicialização, alocação de elementos do conjunto, escalonamento de eventos, processamento de listas, definição, operação e consulta aos recursos e, finalmente, depuração e relatório.

O mecanismo de escalonamento na lista de eventos compreende um procedimento para o escalonamento de eventos e outro para a seleção do próximo evento, uma variável que representa o valor atual do tempo de simulação, e a própria estrutura de dados da lista de eventos. Quando um evento, como por exemplo a chegada do próximo usuário numa fila M/M/1, deve ser escalonado, o procedimento `schedule` é invocado para criar e adicionar uma entrada na lista de eventos. Essa entrada contém o tempo de ocorrência do evento, a sua identidade e, geralmente, a identidade do cliente associado ao evento.

A lista de eventos é ordenada ascendentemente de acordo com os tempos de ocorrência dos eventos; a nova entrada é colocada na lista na posição determinada pelo tempo de ocorrência do evento. Quando o programa de simulação completa todo o processamento referente a um dado instante do tempo de simulação, ele chama o procedimento `cause` para determinar qual o próximo evento a ocorrer. A função `cause` remove a entrada da cabeça da lista de eventos, avança o tempo de simulação para o tempo de ocorrência do evento contido na entrada e retorna a identidade do evento (e do cliente, se incluído). Após a chamada de `cause`, o programa de simulação inicia o tratamento do evento.

5.2 Alterações no smpl

Quando se está realizando uma simulação seqüencial com o *smpl*, tem-se um único fluxo de programa que, segundo as suas necessidades, realiza chamadas às funções do *smpl*. Tem-se uma única lista de eventos, onde se escalona e se seleciona (para tratamento) os eventos.

Quando essa estrutura passa a ser distribuída, tem-se vários processos executando concorrentemente. Cada processo tem sua própria lista de eventos e esses processos se comunicam através de mensagens. Quando um processo deseja escalonar um evento em outro processo, ele deve enviar uma mensagem a esse outro processo.

Cada processo tem o seu próprio relógio de simulação, que corresponde ao menor valor dos relógios de seus *links* de entrada. Os processos devem assumir que as mensagens que chegam por um determinado *link* vêm em ordem não decrescente de suas marcas de tempo. Conseqüentemente, todo processo deve garantir que mantém essa mesma ordem no envio de mensagens por um *link* específico.

Para se implementar o *d-smpl*, as funções que sofreram mais alterações foram as do grupo de escalonamento de eventos. Além disso, foi criado um novo arquivo, o *d-smpl.c*, contendo funções específicas para a simulação distribuída.

Na lista de eventos, na estrutura evento escalonado, foi criado mais um campo (de tipo inteiro). A função desse novo campo é guardar, juntamente com cada evento escalonado, a informação referente a de que porta, e conseqüentemente de que processo, esse evento é proveniente. Essa informação é armazenada quando o evento é escalonado e recuperada quando o evento é selecionado para ser simulado. Ao final da função `cause`, usa-se essa informação para atualizar o relógio da referida porta de entrada, indicando que aquela mensagem já foi processada.

Quando a função `cause` é invocada, o evento da cabeça da lista não é mais automaticamente selecionado para simulação. Antes disso, a função `cause` fica num laço testando se o tempo da mensagem da cabeça da lista é menor que o relógio do processo. Esse teste é realizado através de duas novas funções criadas, que são chamadas pela função `cause`. Para se retornar o tempo da mensagem da cabeça da lista foi desenvolvida a função `HeadTime`, e para estipular o relógio do processo foi implementada a função `LowerClk`. A função `LowerClk` lê os valores dos relógios das portas de entrada e retorna como o relógio do processo o menor valor entre os relógios das portas de entrada.

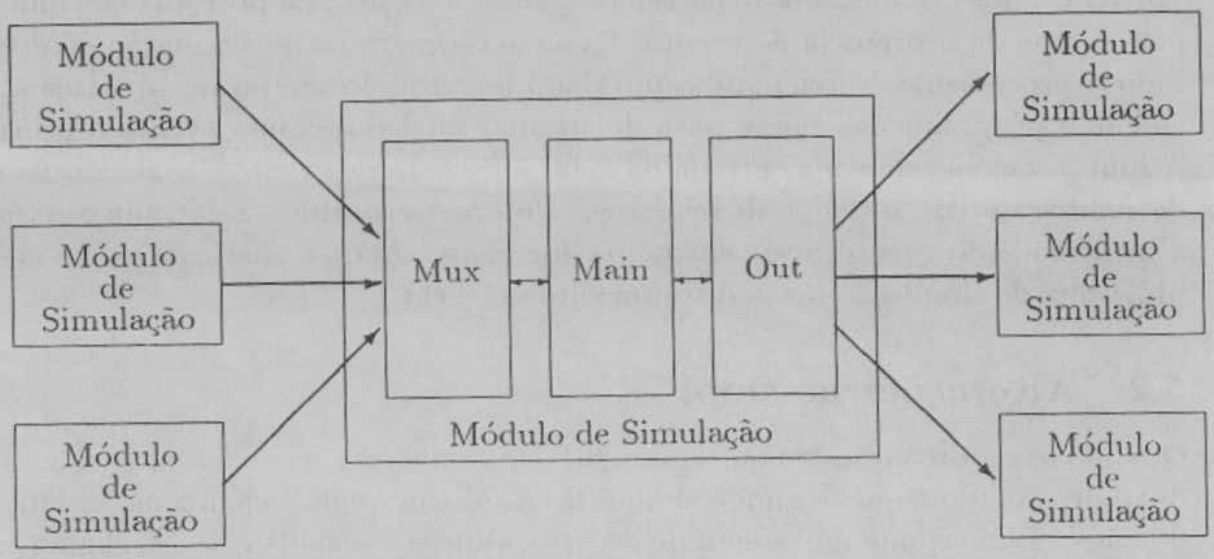


Figura 4 Estrutura de um Programa de Simulação em d-smpl.

Quando o tempo da mensagem da cabeça da lista for menor que o relógio do processo, o evento pode ser selecionado para simulação com segurança. Isso significa que não chegará posteriormente a esse processo nenhum outro evento com uma marca de tempo inferior à marca do evento que está sendo simulado. Em caso negativo, a função `cause` fica temporariamente bloqueada nesse laço até que a condição seja satisfeita. Esse é o princípio básico dos métodos conservativos: um evento só é processado quando ele for seguro.

No *smpl*, os dados relacionados com o tempo são todos do tipo `real`. No *d-smpl*, eles são todos inteiros. Essa modificação deve-se a basicamente dois motivos: a velocidade de comparação de inteiros é bem superior a de números reais. Mesmo em simulações seqüenciais com o *smpl* é comum fazer-se essa alteração. A outra razão é intrínseca ao ambiente utilizado. No *C Paralelo* só existem funções para transmitir e receber inteiros através dos canais dos processos. Para a comunicação de números reais, ter-se-ia que subdividi-los em inteiros na transmissão e rearrumá-los na recepção, causando um gasto de tempo ainda maior. Dado que não há grandes prejuízos nessa mudança, preferiu-se optar por essa simplificação.

5.3 Novas Funções do d-smpl

Quando um processo utiliza as funções distribuídas do *d-smpl*, ele necessita de pelo menos mais dois fluxos de execução concorrentes com o fluxo principal do programa. Um fluxo é destinado a fazer a recepção e o armazenamento das mensagens que chegam ao processo, e será aqui chamado de `mux`. Um outro fluxo cuida do envio de mensagens aos demais processos, denominado de `out`. O terceiro fluxo seria o que contém o programa de simulação em si, conhecido como `main`. Essa estrutura pode ser vista na figura 4.

A necessidade desses três fluxos concorrentes é que as funções por eles desempenhadas devem ser realizadas simultaneamente. Alguns dados, como a lista de

eventos, os relógios das portas de entrada, o relógio do processo, entre outros, são compartilhados por mais de um desses fluxos, tendo sido usados os recursos de semáforo que o *C Paralelo* oferece.

O fluxo `mux` trata da recepção das mensagens. Uma mensagem, ao ser recebida, é escalonada na lista de eventos e seu tempo de evento é armazenado num *buffer* de marcas de tempo de eventos específico para cada porta de entrada. Isso corresponde à mensagem estar virtualmente num *buffer* de entrada, quando na realidade ela está armazenada já na lista de eventos. Quando o evento ocorre, sendo retirado da lista de eventos, esse *buffer* de marcas de tempo, juntamente com o relógio dessa porta, é atualizado.

O fluxo `out`, por sua vez, tem como função cuidar de toda a parte de envio de mensagens aos outros processos. Este fluxo recebe os eventos do fluxo principal e os coloca em *buffers* específicos para cada porta de saída. Ele recebe as mensagens fora de ordem das marcas de tempo e precisa garantir que uma vez tendo emitido uma mensagem com uma certa marca de tempo por uma determinada porta, não enviará depois, por essa mesma porta, nenhuma mensagem com uma marca inferior a essa.

O fluxo `main` é o que contém o corpo do programa de simulação. Ele equivale a um programa de simulação seqüencial.

Quando uma mensagem nula chega a um processo, sua marca de tempo é usada para atualizar o vetor de marcas de tempo de sua porta de entrada. Com isso, o valor do relógio de sua porta de entrada e o valor do relógio do processo são incrementados, permitindo assim que mensagens que se encontram na lista de eventos possam ser consideradas seguras e serem processadas. Como as mensagens nulas têm efeito apenas de sincronização, elas não são escalonadas na lista de eventos.

6 Conclusões

Este trabalho apresentou o projeto e implementação de um simulador distribuído baseado no método conservativo de simulação distribuída CMB.

O simulador foi desenvolvido como um extensão distribuída à linguagem de simulação *smpl* e foi implementado utilizando o *C Paralelo* e *transputers*. Apesar da implementação de métodos de simulação distribuída não ser uma tarefa trivial, a estrutura simples do *smpl* tornou relativamente fácil modificá-lo. A idéia básica durante a implementação do simulador *d-smpl* foi, a exemplo do próprio *smpl*, tornar a simulação o mais transparente possível ao usuário. Essa característica é necessária para que um programador ao utilizar o simulador possa se abster ao máximo dos detalhes de implementação.

A simulação mostra-se como uma ferramenta adequada à validação de resultados analíticos ou ao tratamento de problemas em que uma abordagem analítica seria impraticável. Desse modo, a simulação distribuída representa um avanço, na medida em que aumenta a eficiência na obtenção dos resultados por simulação. Portanto, vale destacar a relevância deste trabalho em apresentar o desenvolvi-

mento de um simulador distribuído para uma arquitetura paralela.

A avaliação experimental de simulação distribuída requer não apenas a implementação de um simulador como também um conjunto de casos de teste. Isso é particularmente importante à luz de estudos de simulação já realizados [RMM88, See79], que mostraram que o desempenho da simulação distribuída é extremamente sensível à topologia da rede de filas simulada. Testes simples (ex: redes em tandem) têm resultados facilmente interpretáveis, mas não refletem simulações típicas. Por outro lado, simulações de redes de filas complexas, embora realistas, dificultam a tarefa de interpretar as fontes de degradação de desempenho em simulação distribuída.

O próximo passo é mostrar a utilidade da ferramenta criada (*d-smpl*) através do seu uso em diversos estudos de caso. Inicialmente, serão estudados casos simples, partindo-se depois para aplicações mais complexas, como por exemplo a de simulação de uma rede ATM com vários nós comutadores. Será feita também uma análise de desempenho buscando avaliar as seguintes medidas: desempenho em relação à simulação seqüencial, taxa de utilização das CPU's (eficiência), entre outras. Uma apresentação completa desses estudos de caso com os respectivos resultados das simulações estará disponível em [Aze93].

Referências

[Aze93] Cláudia M. R. Azevedo. d-*smpl*: Um simulador distribuído para transputers. Tese de mestrado, Universidade Federal de Pernambuco, 1993. A ser apresentada.

[Bry77] R.E. Bryant. *Simulation of Packet Communications Architecture Computer Systems*. Tese de doutorado, Massachusetts Institute of Technology, 1977.

[BT91] A. Boukerche and C. Tropper. A performance analysis of distributed simulation with clustered processes. *Distributed Simulation*, 23(1):112–121, 1991.

[CM79] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5), November 1979.

[Fuj88] R.M. Fujimoto. Performance measurements of distributed simulation strategies. *Distributed Simulation*, 19(3):14–20, 1988.

[Fuj90] R.M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):31–53, October 1990.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of ACM*, 21(8):666–677, 1978.

[JS85] D.R. Jefferson and H. Sowizral. Fast concurrent simulation using the time warp mechanism. In *Proceedings of the SCS Distributed Simulation Conference*, 1985.

[Kle90] L. Kleinrock. On distributed systems performance. *Computer Network and ISDN Systems*, 20:209–215, 1990.

[Mac87] M.H. MacDougall. *Simulating Computer Systems - Techniques and Tools*. The MIT Press, 1987.

[Mis86] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[Pfi82] G. F. Pfister. The Yorktown simulation engine: Introduction. *Proc. ACM IEEE 19th Design Automation Conference*, pages 51–54, June 1982.

[PWM79a] J.K. Peacock, J.W. Wong, and E.G. Manning. A distributed approach to queueing network simulation. In *Proceedings of the Winter Simulation Conference*. IEEE Press, 1979.

[PWM79b] J.K. Peacock, J.W. Wong, and E.G. Manning. Distributed simulation using a network of processors. *Computer Networks*, 3(1), 1979.

[Rat85] J. Rattner. Concurrent processing: A new direction in scientific computing. In *AFIPS National Computer Conference Proceedings Reprint*, Santa Clara, CA, 1985. Intel Corp.

[RMM88] D.A. Reed, A.D. Malony, and B.D. McCredie. Parallel discrete event simulation using shared memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, April 1988.

[See79] M. Seethalakshmi. A study and analysis of performance of distributed simulation. Master's thesis, Computer Science Dept., Univ. of Texas at Austin, 1979.

[SX92] J.T. Stephen and M.Q. Xu. Performance evaluation of the bounded time warp algorithm. *Distributed Simulation*, 24(3):117–126, 1992.