Um Mecanismo de Reexecução Determinística de Programas Paralelos

Anna Thereza Cortiñas Albuquerque

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

Aprovada por:

Prof. Claudio Luis Amorim, Ph. D.
(presidente)

Prof. Valmir Carneiro Barbosa, Ph. D.

Prof. Orlando Gomes Loques Filho, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
AGOSTO DE 1992

ALBUQUERQUE, ANNA THEREZA CORTIÑAS

Um Mecanismo de Reexecução Determinística de Programas Paralelos
[Rio de Janeiro] 1992
V, 88 p., 29.7 cm, (COPPE/UFRJ, M. Sc., ENGENHARIA DE SISTEMAS E COMPUTAÇÃO, 1992)
TESE – Universidade Federal do Rio de Janeiro, COPPE
1 – Depuração Distribuída 2 – Reexecução de Programas 3 – Sistemas Distribuídos
I. COPPE/UFRJ II. Título(Série).

A meus pais

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

Um Mecanismo de Reexecução Determinística de Programas Paralelos

Anna Thereza Cortiñas Albuquerque

Agosto de 1992

Orientador: Claudio Luis Amorim

Programa: Engenharia de Sistemas e Computação

A depuração de um programa paralelo é uma tarefa complexa. Os programas paralelos possuem uma natureza inerentemente não determinística. Muitas das metodologias de depuração aplicadas aos programas seqüenciais não se adequam aos programas paralelos. Esta característica tem estimulado o desenvolvimento de mecanismos que controlam a execução dos programas paralelos, eliminando o carácter não determinístico. Um dos mecanismos de reexecução mais destacados é denominado *Instant Replay.* Este trabalho se destina a programas paralelos usando memória compartilhada e apresenta um impacto bastante reduzido no desempenho dos sistemas. O objetivo deste trabalho de tese é apresentar o projeto e a avaliação de um mecanismo de reexecução determinística voltado para programas paralelos em memória distribuída. Os fundamentos que inspiraram o seu desenvolvimento foram extraídos do *Instant Replay.* A implementação dos algoritmos envolvidos foi feita na linguagem de programação *occam* e os testes conduzidos em um hipercubo formado por 8 processadores do tipo *transputer.*

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

A Mechanism for Deterministic Replaying of Parallel Programs

Anna Thereza Cortiñas Albuquerque

August, 1992

Thesis Supervisor: Claudio Luis Amorim

Department: Programa de Engenharia de Sistemas e Computação

The debugging of a parallel program is an extremely difficult task. Parallel programs are inherently nondeterrninistic. Many debugging methodologies for sequential programa are not effective for parallel programs. This feature has estimulated the development of mechanisms for controlling parallel programs' execution, avoiding the nondeterministic behavior. One of the most important replaying mechanisms is called Instant Replay. This work addresses parallel programs nsing shared memory and presents a minor impact on systems' performance. The aim of this work thesis is to describe the project and the evaluation of a mechanism for deterministic replaying of parallel programs running on distributed memory multi-processors. Several theorical aspects of the mechanismb design were proposed during Instant Replay's devolpment. The algorithms envolved were writen in occam programming language and the tests were performed in an 8 transputer hypercube.

Índice

I Introdução 1

II Características Básicas de Sistemas Distribuídos 5

II.1 Sistema Distribuído Ponto-a-Ponto 5

II.2 Algoritmo Distribuído 6

II.3 Comunicação entre Processos 6

II.4 Comandos Guardados 10

Problemas Associados Depuração de Algoritmos Distribuídos . . . . .

III Mecanismo de Reexecução 13

III.1 Metodologia de Reexecução 13

III.2 Descrição do Mecanismo 15

III.3 Estrutura do Mecanismo 16

III.4 Algoritmo Monitor 17

III.4.1 Unidade Registra Evento 19

III.4.2 Procedimento Buffer Local (BUFL) 23

III.4.3 Processo Buffer Remoto (BUFR) 24

III.4.4 Processo Disco Monitor (DISCM) 25

III.5 Algoritmo Repetidor 27

III.5.1 Unidade Habilita Evento 28

III.5.2 Processo Buffer Replay (BUFREP) 31

III.5.3 Processo Disco Replay (DISCR) 35

III.6 Mapeamento dos Processos 37

III.7 Prevenção de Deadlock 37

IV Análise dos Resultados **47**

IV.1 Computador Paralelo NCP1 48

IV.2 Aplicações Estudadas 48

IV.3 Avaliação 52

IV.3.1 Overhead x Custo da Comunicação 54

IV.3.2 Overhead x Carga Computacional 61

IV.3.3 Medidas de Tempo com I/O 65

V Conclusão **69**

A Interface de Comunicação **74**

B Algoritmo de Multiplicação **77**

Capítulo 1

Introdução

A depuração de um programa paralelo ou distribuído é uma tarefa complexa. As computações paralelas apresentam certas características inerentes a sua construção que impossibilitam a aplicação de diversas técnicas de depuração, corriqueiramente usadas pela maioria dos programadores. Dentre estas técnicas está a *depuração cíclica*. O programador analisa o comportamento do programa, insere sondas ou modificações e, em seguida, reexecuta o programa, observando os resultados obtidos. A eficácia da depuração cíclica se sustenta na reprodutibilidade dos programas a cada reexecução.

Os programas paralelos, no entanto, possuem uma natureza não determinística; ou seja, sucessivas execuções de um programa a partir dos mesmos dados de entrada podem apresentar resultados distintos. Portanto, um certo comportamento exibido durante uma execução não é necessariamente reproduzido nas execuções subseqüentes. E, assim sendo, a depuração cíclica, a príncipio, não pode ser aplicada a este tipo de computação.

Inicialmente, as alternativas disponíveis para a depuração de um programa paralelo estão, aparentemente, restritas a análises estáticas de *snapshots* extraídos durante a execução.

No sentido de viabilizar a aplicação da depuraçiio cíclica e de muitas outras técnicas, foram propostos diferentes mecanismos que *controlam* a execução dos programas paralelos, eliminando o carácter não determinístico. Estes mecanismos atuam em duas etapas de operação, denominadas *monitoração* e *reexecução*. Durante a primeira etapa, o programa é executado e, concorrentemente, o mecanismo observa e registra certos eventos. Na segunda etapa, os eventos monitorados são usados para coordenar a reexecução do programa, que apresenta, então, as mesmas características observadas quando da etapa de monitoração.

Os mecanismos propostos se diferenciam gela natureza dos eventos monitorados e pelo volume de informações usadas para controlar a reexecução. A granularidade destas variáveis determina uma provável interferência imposta aos programas submetidos a este tipo de mecanismo. Esta interferência, freqüentemente referenciada na literatura como *Probe Effect*, pode ocasionar uma inofensiva redução na *performance* ou uma danosa adulteração do comportamento do programa. A *reação* de uma computação a presença de um mecanismo é imprevissível. O único aspecto conclusivo indica que mecanismos pouco intrusivos determinam uma pequena interferência que, *possivelmente*, é insuficiente para modificar o comportamento dos programas.

As interferências efetivamente nocivas ocorrem durante a etapa de monitoração. A reexecução representa, simplesmente, a repetição determinística de um comportamento anteriormente monitorado.

Este documento descreve o projeto e a avaliação de um mecanismo de reexecução determinística, destinado a programas paralelos usando memória distribuída. Os príncipios que inspiraram o seu desenvolvimento foram originalmente propostos para o *Instant Replay*. Este mecanismo, voltado para programas paralelos em memória compartilhada, se destaca por provocar uma interferência bastante reduzida sobre o desempenho das computações.

O *Instant Replay* se fundamenta nas seguintes observações: *um mecanismo de re-*

execução deve garantir) simplesmente, que cada um dos processos da computação receba as mensagens segundo a mesma ordem parcial a cada execução. Não é necessário o armazenamento dos conteúdos das mensagens, pois estes são reconstruídos idênticos durante as reexecuções. Com isso, o volume de informações usadas na reprodução de uma certa execução é consideravelmente pequeno. E, conseqiientemente, a inserção de sondas para a monitoração destas informações ocasiona um impacto limitado sobre a performance da computação. As propostas discutidas neste trabalho de tese se baseiam nestas observações.

O projeto do mecanismo, aqui descrito, especifica um algoritmo distribuído que se superpõe as computações e implementa procedimentos de monitoração e reexecução. O algoritmo é composto por uma coleção de sondas e processos bastante simples, que distribuem o processamento, aumentando o grau de concorrência, evitanto a criação de gargalos e, assim, procurando minimizar as perturbações impostas as computações.

A avaliação do desempenho do mecanismo procura determinar o grau de interferência imposto as computações, relacionando medidas de tempo, envolvendo a execução de uma aplicação e os procedimentos de monitoração e reexecução.

O conteúdo deste documento pode ser resumido como:

O Capítulo II discute diversos conceitos relacionados a computação distribuída, visando apresentar a nomenclatura, notação e os aspectos teóricos que contribuem para o entendimento das definições, conceitos e algoritmos apresentados nos capítulos subsequentes.

O Capítulo III descreve o mecanismo de reexecução, discutindo, extensamente, os algoritmos de monitoração e reexecução implementados, além das dificulades associadas ao mapeamento em sistemas distribuídos ponto-a-ponto.

O Capítulo IV analisa medidas de tempo obtidas em uma bateria de testes que obje-

tivou quantificar o grau de interferência imposto pelo mecanismo a duas aplicações. Os testes procuraram exercitar o comportamento do mecanismo em relação ao custo da comunicação entre processadores e as **flutuações** na carga **computacional.** Os programas envolvidos foram **implementados** utilizando a linguagem de programação *occam* e os testes realizados em um **hipercúbo** formado por 8 processadores do tipo ***transputer.***

O Capítulo V resume os resultados mais significativos alcançados durante o desenvolvimento deste trabalho e apresenta algumas sugestões para futuras extensões.

Capítulo II

Características Básicas de Sistemas Distribuídos

Este capítulo discute diversos conceitos relacionados a computação distribuída, visando apresentar a nomenclatura, notação e os aspectos teóricos que contribuem para o entendimento das definições, teoremas e algoritmos apresentados nos demais capítulos. A descrição dos príncipios aqui discutidos é conduzida de forma sucinta. Uma apresentação mais formal e extensa pode ser encontrada nas referências fornecidas.

II.1 Sistema Distribuído Ponto-a-Ponto

Um sistema distribuído é composto por uma coleção de processadores e canais unidirecionais de comunicação FIFO (*First-In-First-Out*) [Figura II.3 (a)]. Cada processador possui uma memória e um relógio local independentes e, portanto, o sistema é assíncrono e toda a interação entre os processadores é realizada através da troca de mensagens. Os canais de comunicação, também chamados de *canais físicos*, são modelados livres de quaisquer condições de erro e sujeitos a atrasos finitos, porém indeterminados, durante o trânsito das mensagens. Cada canal conecta somente dois processadores de forma que um destes executa, exclusivamente, operações de escrita

ou envio de mensagem, e o outro operações de leitura ou recepção de mensagem. Para o processador remetente, o canal é dito *de saída* e para o receptor *de entrada* [9] [21].

II.2 Algoritmo Distribuído

A realização de uma certa tarefa distribuidamente envolve o projeto de um algoritmo distribuído que especifica a natureza da computação a ser executada em cada processador do sistema. A computação associada a um processador é composta por uma coleção de um ou mais processos concorrentes e por uma rede de canais lógicos de comunicação FIFO. A representação de um algoritmo distribuído é feita a partir de um grafo orientado, onde cada vértice simboliza um processo e cada arco entre dois vértices representa um canal unidirecional de comunicação de uso exclusivo dos dois processos interconectados pelo canal [Figura II.3 (b)]. De uma forma geral, o grafo que descrever um certo algoritmo é diferente da topologia do sistema distribuído disponível. Ainda assim, a partir de modificações na estrutura original do grafo, é possível adequar um algoritmo a uma topologia específica, como descrito na seção II.3. Semelhante aos canais físicos, os canais lógicos também podem ser denominados como *de entrada* ou *de saída* [5] [6].

II.3 Comunicação entre Processos

A interação entre os processos é realizada através da troca de mensagens sobre os canais de comunicação. As operações de envio e recepção são implementadas pelas primitivas:

- **EnviaMsg (C, M).** Envia a mensagem *M* através do canal *C*.
- **RecebeMsg (C, M).** Recebe a mensagem *M* pendente no canal C

A capacidade de armazenamento dos canais é *zero* e, portanto, para que uma comunicação se redize é necessário que os processos emissor e receptor estejam sincronizados. A capacidade de qualquer canal pode ser expandida para um valor finito `(capacidade limitada)` mediante a inserção de *buffers*, permitindo, com iseo, que o processo emissor transfira um certo número de mensagens até se bloquear.

Cada canal de comunicação está associado a um protocolo determinando o formato das mensagens que nele transitam. Este formato admite mensagens de tamanhos fixo ou variável, contendo diferentes tipos de dados que vão de estruturas complexas a um simples rótulo [4]. Exemplos:

- **C : canal of END;**
  Mensagens que transitam em *C* contém um rótulo do tipo *END*.
- **C : canal of integer [4];**
  Mensagens que transitam em *C* contém 4 inteiros.
- **C : canal of { integer, byte, real [ ] };**
  Mensagens que transitam em *C* contém 1 inteiro, 1 *byte* e um número variável de reais.

A troca de mensagens entre processos residentes em processadores distintos é realizada seguindo o mesmo procedimento adotado quando os processos pertencem a um mesmo processador. Ou seja, a partir de chamadas as primitivas *EnviaMsg* e *RecebeMsg*. O tratamento das mensagens destinadas a processos externos a um processador é implementado, de forma transparente, por um grupo de processos especiais denominados *Interface de Comunicação* (ICOM). Os processos da ICOM abstraem os algoritmos das características físicas de um certo sistema distribuído, tornando os programas mais portáveis. A ICOM pode ser vista como uma coleção de *buffers* e, assim sendo, os canais conectados a Interface possuem capacidade de armazenamento limitada.

Os processos da ICOM podem ser enumerados e descritos como [Figura II.3 (c)]:

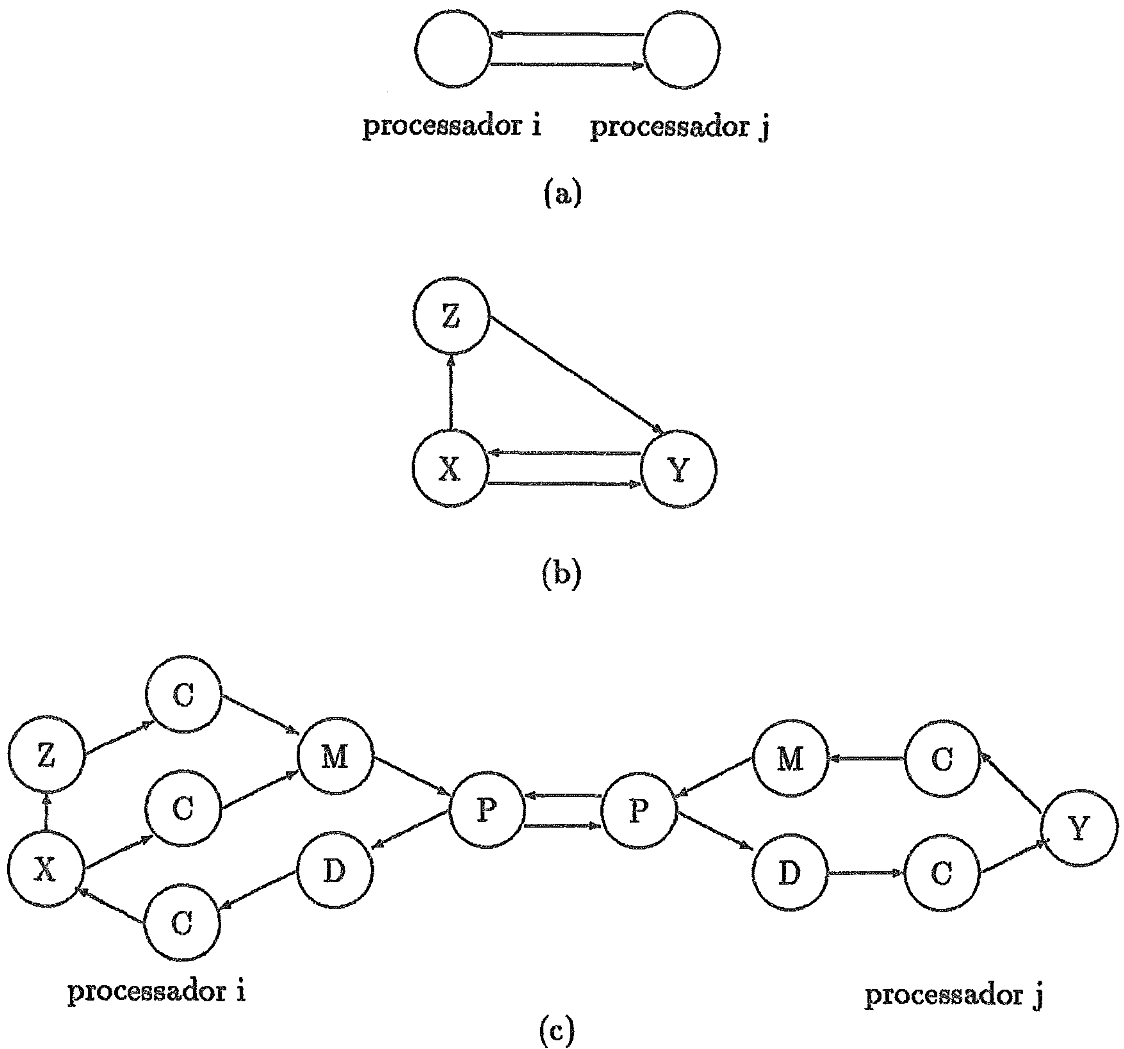


Figura II.1: Mapeamento de um Grafo em uma Topologia. (a) Sistema distribuído ponto-a-ponto. (b) Grafo de processos representando um algoritmo distribuído. (c) Grafo de processos *(b)* mapeado no sistema distribuído *(a)*. As legendas *C*, *M*, *D* e *P* representam, respectivamente, os processos CONV, MUX, DUX e PVC.

1. *Processador Virtual de Comunicação (PVC).* É responsável, dentre outras funções, pela implementação de um mecanismo de roteamento transparente e livre de interbloqueios de comunicação (*deadlock*). O acesso ao PVC é realizado através de operações de envio e recepção de mensagem usando, respectivamente, os canais *to.PVC e from.PVC* [3].

2. *Multiplexador (MUX).* Está associado a uma coleção de canais de entrada e um único canal de saída, *to.PVC.* Quando existem urna ou mais mensagens pendentes em algum dos seus canais de entrada, o MUX seleciona somente uma das mensagens e, em seguida, a transmite para o PVC. O MUX pode armazenar uma única mensagem e, portanto, nenhuma outra mensagem é recebida até que a primeira seja transferida para o PVC.

3. *DeMultiplexador (DUX).* Está conectado a um conjunto de canais de saída e um único canal de entrada, *from.PVC.* Quando uma mensagem é enviada pelo PVC, o DUX seleciona um dos seus canais de saída e envia a mesma mensagem recebida. Tal qual o MUX, o DUX pode guardar uma única mensagem, permanecendo bloqueado até que o processo Conversor de Protocolos receba a mensagem pendente. O MUX e o DUX centralizam o acesso aos canais *from.PVC* e *to.PVC* contornando a restrição imposta pelo modelo do sistema discutido, que impossibilita o acesso concorrente de dois ou mais processos a um mesmo canal de comunicação.

4. *Conversor de Protocoko (CONV).* É responsável por compatibilizar os protocolos usados pelos processos com aquele usado pelo PVC. O CONV possui um canal de entrada e um de saída que estão conectados, respectivamente, a um processo e ao MUX (protocolo PVC → protocolo processo), ou ao DUX e a um processo (protocolo processo → protocolo PVC). Também armazena um única mensagem somente e, assim sendo, não recebe outras mensagens até que a primeira seja despachada.

Os algoritmos dos processos Multiplex, DeMultiplex e Conversar de Protocolo são apresentados no Apêndice A.

II.4 Comandos Guardados

Um comando guardado é formado por um *guarda* e uma *lista de comandos.* Um guarda consiste de uma lista de declarações, expressões booleanas e um comando de recepção de mensagem, podendo cada um destes itens existir sozinho ou combinado com os demais. A lista de comandos é executada quando o guarda é *factível.* Isto 6, as expressões booleanas tem valor *true* e/ou o comando de recepção foi completado [2].

Diversos comandos guardados podem ser reunidos em um *comando alternativo*, responsável por selecionar, arbitrariamente, um único guarda dentre aqueles *factíveis.* Somente a lista dê comandos do guarda escolhido é executada. Exemplo:

```
flag and (i = j) →
        flag := false;
    or
flag RecebeMsg (C1, M1) →
        i := 0;
        j := 0;
    or
flag and RecebeMsg (C2, M2) →
        i := i + 1;
        flag := false;
```

Os algoritmos apresentados nos demais capítulos utilizam uma notação, denominada *cornando alternativo replicado*, que compacta a representação dos comandos guardados associados a estruturas de dados tais como vetores ou matrizes. Exemplo:

RecebeMsg (C [i], M), $\forall\, i\ (0 \leq\ i <\ n) \rightarrow$

(*lista de comandos*);

é equivalente a

```
RecebeMsg (C [0], M) →
        ( lista de comandos );
  or
RecebeMsg (C [1], M) →
        ( lista de comandos );
  or

  or
RecebeMsg (C [n - 1], M) →
        ( lista de comandos );
```

II.5 Problemas Associados Depuração de Algoritmos Distribuídos

Os algoritmos distribuídos possuem certas características particulares que tornam a atividade de depuração uma tarefa bastante custosa. Segue, então, a descrição de algumas destas características [15] [14].

- *Complexidade.* Um sistema distribuído é composto por múltiplos processos assíncronos espalhados entre diferentes processadores, o que dificultada a localização e correção de erros.

- *Depuração.* Muitas das técnicas de depuração desenvolvidas para os algoritmos puramente seqüências como, por exemplo, *tracing* ou *breaking point*, não podem ser diretamente aplicadas aos algoritmos distribuídos.

- *Não Determinísmo.* Sucessivas execuções de um algoritmo distribuído a partir dos mesmos dados de entrada podem apresentar resultados completamente distindos.

- *Probe Effect.* A inserção de sondas para a monitoração de um algoritmo distribuído pode adulterar o comportamento do sistema, mascarando a presença de condições de erro [13] [12].

Capítulo III

Mecanismo de Reexecução

Este capítulo estabelece as condições necessárias para que uma execução seja reproduzida. Os fundamentos que inspiram o desenvolvimento deste trabalho foram originalmente propostos em [8] e [22] para programas paralelos usando memória compartilhada. As propostas discutidas se destinam a programas paralelos ou distribuídos projetados segundo os conceitos expostos no Capítulo II.

Os algoritmos aqui apresentados possibilitam a reexecução controlada de um programa paralelo, livre do carácter não determinístico. Os algoritmos manipulam referências que descrevem uma ordem parcial [16] sobre os eventos associados a recepção de mensagens. O reduzido volume dos dados manipulados aliado ao espalhamento das atividades dos algoritmos, possibilitam a minimização do impacto sobre as aplicações. As próximas seções descrevem o mecanismo, apresentam os algoritmos e discutem as dificuldades associadas ao mapeamento em um sistema distribuído ponto-a-ponto.

III.1 Metodologia de Reexecução

Considere um programa seqüencial. Se a cada execução são fornecidos a este programa os mesmos dados de entrada, então os valores resultantes da computação

também são exatamente os mesmos. Além disso, o fluxo computacional descrito pelo programa é unico e a computação é determinística.

Observe agora a seguinte analogia. Em uma computação paralela ou distribuída, os dados de entrada de um processo podem ser vistos como os valores contidos nas mensagens que *chegam* através dos canais de entrada. Da mesma forma que, os valores enviadoe através dos canais de saída representam o resultado da computação, usado por algum outro processo como dado de entrada.

Considere também que os resultados calculados pelos processos são função exclusivamente dos seus dados de entrada. Ou seja, a computação não é ponderada por nenhum fator probabilístico; como acontece, por exemplo, em computações onde o fluxo computacional é alterado por parâmetros associados ao *hardware*, tais como, interrupções ou número de *ticks* do relógio físico.

Se o conteúdo e a ordenação das mensagens recebidas forem invariantes, então os resultados da computação do processo são os mesmos a cada nova execução e, portanto, o comportamento do processo é determinístico.

Generalizando esta observação para todos os processos da computação distribuída, é possível afirmar que se um mecanismo qualquer garante a invariância dos dados de entrada dos processos então uma certa execução pode ser reproduzida.

Intuitivamente, um mecanismo de reexecução determinística deve garantir, simplesmente, que as mensagens sejam recebidas pelos processos sempre na mesma ordem a cada execução. Não é necessário o armazenamento dos conteúdos das mensagens, pois estes são reconstruídos durante as reexecuções.

O volume de informações necessárias para a reprodução de uma certa execução é consideravelmente pequeno. E, conseqüentemente, a inserção de *sondas* para a monitoração destas informações, provavelmente, ocasiona uma interferência reduzida sobre o desempenho da computação.

III.2 Descrição do Mecanismo

Considere um algoritmo distribuído formado por uma coleção de processos onde cada processo possui n canais de entrada, com $n \geq 0$. Cada canal está associado a uma única *identidade,* representada por um número inteiro positivo. Em um processo não existem dois ou mais canais de entrada vinculados a uma mesma identidade, não existindo, no entanto, qualquer restrição no que concerne a canais pertencentes a processos distintos.

Um certo mecanismo atua concorrentemente aos processos, observando os eventos associados à recepção de mensagens. Estes eventos são rotulados de acordo com as identidades atribuídas aos canais de entrada. Exemplificando, seja c_{in} um dos canais de entrada do processo p_i e x o inteiro que representa a identidade de c_{in}. Os eventos ocorridos em p_i, envolvendo o canal c_{in}, recebem um *rótulo* x idêntico aquele que identifica c_{in}.

Durante uma execução α do algoritmo, o seguinte procedimento de *monitoração* é conduzido em todos os processos:

> sejam c_1 e c_2 canais de entrada do processo p_i cujas identidades são representadas pelos números *"1"* e *"2"*. Os eventos ocorridos em p_i tem seus respectivos rótulos anotados em uma seqüência s_i. A ordenação de s_i é consistente com a ordem em que os eventos ocorreram em p_i, ou seja, se *"1"* precede *"2"* em s_i então c_1 recebeu uma mensagem *ANTES* de c_2, segundo o relógio do processador onde p_i reside. Ao final da computação α, cada seqüência contém a *ordem parcial* dos eventos de recepção associados a um processo p_i.

Considere agora que uma outra execução do mesmo algoritmo, os rótulos reunidos nas seqüências são usados para produzir uma *execução* β *equivalente a* α. O procedimento de *reexecução* conduzido é descrito como:

observe novamente os canais de entrada c_1 e c_2 pertencentes ao processo p_i com identidades simbolizadas pelos inteiros *"1"* e *"2"*. As mensagens são recebidas por p_i respeitando a ordenação armazenada em s_i, isto é, se *"2"* precede *"1"* na seqüência então c_2 é habilitado *ANTES* de c_1 e, assim sendo, nenhuma mensagem é recebida por c_1 até que c_2 complete sua comunicação. Ao término de β a mesma ordem parcial estabelecida em α foi seguida.

As execuções α e β apresentam os mesmos resultados e fluxo computacional e, assim sendo, os processos submetidos ao mecanismo possuem uma natureza determinística.

As próximas seções utilizam a palavra *evento* para referenciar, exclusivamente, aquelas operações associadas à recepção de mensagens. Qualquer outra interpretação é explicitamente mencionada.

III.3 Estrutura do Mecanismo

O mecanismo descrito representa um algoritmo distribuído que se superpõe a uma computação e implementa os procedimentos de *monitoração* ou *reexecução*. O algoritmo é composto por uma coleção de sondas e processos bastante simples, que distribuem o processamento, aumentado o grau de concorrência, evitando a criação de gargalos e, assim, reduzindo a interferência (*probe effect*) [13] [12] sobre as computações.

O projeto do mecanismo considera as tarefas de monitoração e reexecução como atividades independentes, exercidas por dois algoritmos diferentes, denominados *Algoritmo Monitor* e *Algoritmo Repetidor*. Estes algoritmos são acionados, separadamente, conforme a natureza da operação que se deseja realizar, ou seja, monitorar ou reexecutar. Os algoritmos podem ser extratificados em [Figura III.1]:

- **Sondas.** atuam junto aos canais de entrada.
- **Buffers.** concentram temporariamente os dados associados a um processo.
- **Unidade de Disco.** centraliza os dados de associados a todos os processos.

III.4 Algoritmo Monitor

O Algoritmo Monitor coleta e armazena dados relativos a uma computação distribuída, preenchendo um arquivo que é, posteriormente, usado pelo Algoritmo Repetidor para controlar a reexecução da mesma computação, a partir dos mesmos parâmetros de entrada. Os dados coletados são, essencialmente, os rótulos dos eventos associados a recepção de mensagens. Ou seja, as identidades atribuídas aos canais de entrada dos processos.

O funcionamento do algoritmo pode ser descrito como: durante uma execução, as sondas monitoram os rótulos dos eventos ocorridos nos processos. Estes rótulos permanecem armazenados em variáveis, denominadas *buffers*, até que, ao final da execução, são reunidos em um único arquivo. Cada processo possui seu *buffer* exclusivo e, portanto, eventos associados a processos distintos não são embaralhados. A disposição dos rótulos no arquivo é consistente com a ordem parcial dos eventos de cada processo, individualmente. A ordenação descrita não representa, necessariamente, uma *ordem total* viável [10].

O projeto do algoritmo considera duas propostas de implementação:

1. *Orientada ao Procedimento.* Os *buffers* são declarados no contexto dos processos da aplicação. As sondas, aqui chamadas de *Unidades Registra Evento*, exercem o controle sobre o armazenamento dos rótulos, diretamente. Cada processo possui um canal de saída adicional, incidente ao *Processo Disco Monitor* que centraliza os acessos ao arquivo [Figura III.2 (c)].

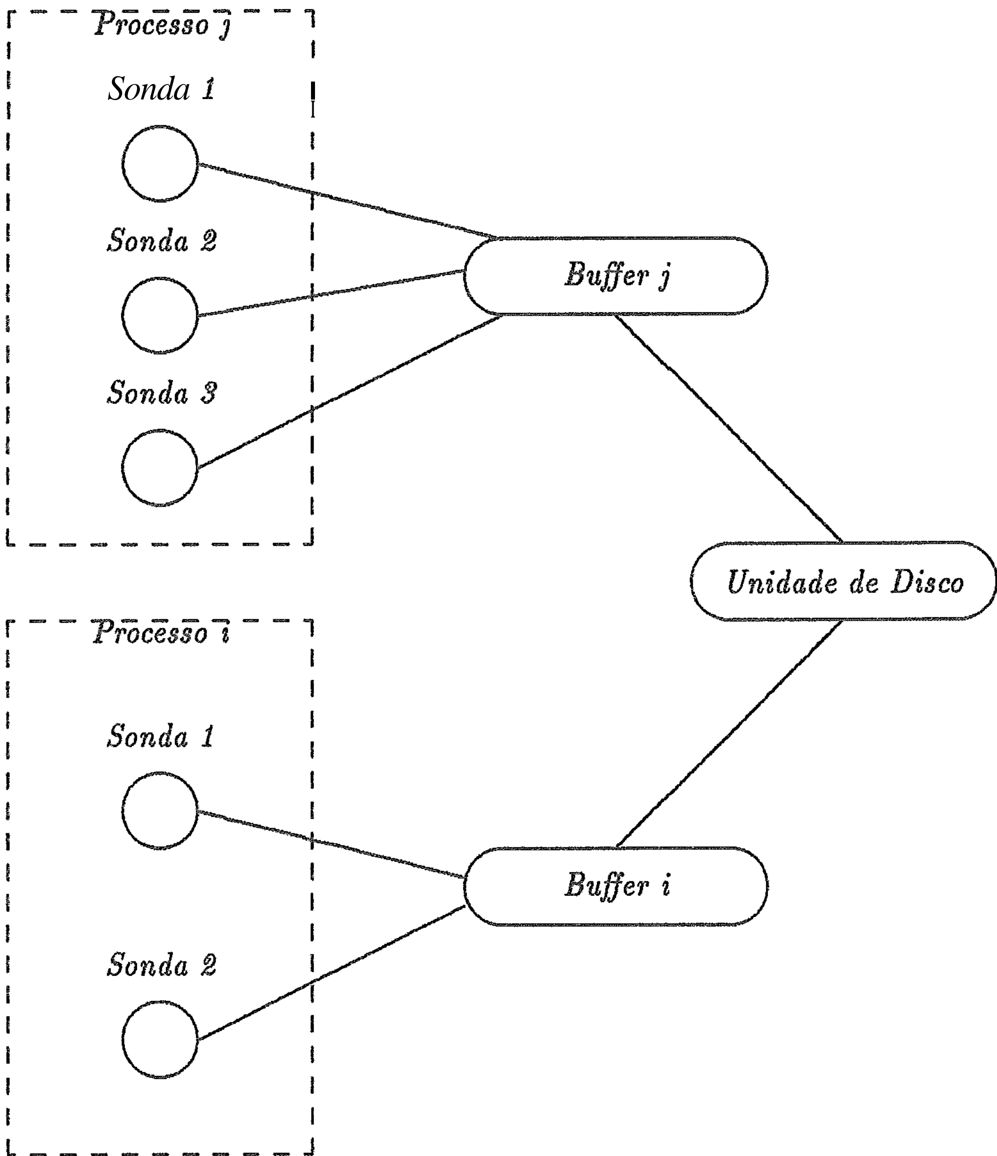


Figura III.1: Algoritmos Monitor e Repetidor - o exemplo ilustra a estrutura formada para uma aplicação composta pelos processos i e j associados a dois e três canais de entrada, respectivamente

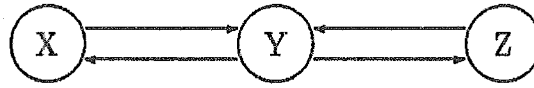
2. *Orientada à Mensagem.* Os *buffers* são declarados no contexto dos processos *Buffer Remoto*, que concentram a tarefa de armazenamento dos rótulos. As Unidades Registra Evento atuam, agora, de forma indireta, transmitindo os valores monitorados. Cada processo da aplicação possui uma coleção de canais de saída usados, estritamente, na comunicação com os processos Buffer Remoto. Este, por sua vez, possui um único canal de saída ligado ao processo *Disco Monitor* [Figura III.2 (b)].

A terminação do algoritmo coincide com o final da execução de cada um dos processos presentes no sistema. Os processos da aplicação terminam após transmitirem um sinal de terminação (*stop*) sobre somente um dos seus canais de saída dentre aqueles conectados a processos do Monitor. O processo Buffer Remoto cessa ao receber um único *stop*, que é propagado através do seu canal de saída. O processo Disco Monitor acaba ao receber um sinal de terminação em cada um dos seus canais de entrada. Ou seja, um *stop* por processo da aplicação. O fluxo da propagação das mensagens de terminação é mostrado na Figura III.2.

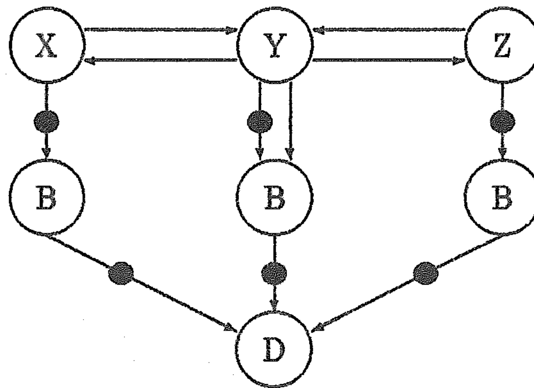
III.4.1 Unidade Registra Evento

A *Unidade Registra Evento* atua associada a cada um dos canais de entrada dos processos no sentido de reconhecer e registrar a ocorrência de eventos. As funções da Unidade são implementadas a partir da inserção de um *observador* imediatamente após as primitivas de recepção de mensagem, encontradas no código fonte dos processos. Quando uma mensagem é recebida, o observador correspondente é ativado e a identidade do canal de entrada associado ao evento é anotada.

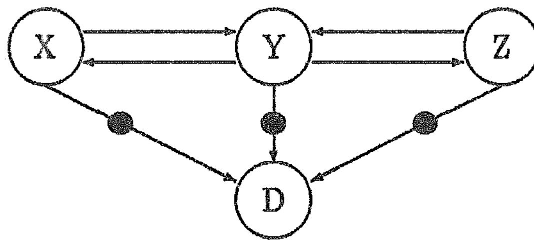
O projeto do observador considera duas abordagens relativas as implementações *orientada à mensagem* e *orientada ao procedimento.* A primeira, o conjunto dos rótulos dos eventos de um processo é armazenado, remotamente, em um *buffer* pertencente ao *Processo Buffer Remoto* (BUFR) (seção III.4.3). Cada processo é vinculado a um BUFR distinto, ou seja, existe um igual número de processos monito-



(a)



(b)



(c)

Figura III.2: Representação do Algoritmo Monitor. (a) Grafo de processos da aplicação. (b) Grafo formado a partir da inserção dos processos do Algoritmo Monitor na implementação orientada à mensagem. (c) Grafo formado a partir da inserção dos processos do Algoritmo Monitor na implementação orientada ao procedimento. As letras B e D simbolizam os processos Buffer Remoto e Disco Monitor. Os círculos em negrito indicam o caminho percorrido pelo sinal de terminação.

rados e *buffers*. As Unidades Registra Evento estão conectadas aos seus respectivos Processos Buffer Remoto através de uma coleção de canais de comunicação. Cada canal conecta uma única Unidade ao seu BUFR correspondente, de forma que duas ou mais Unidades não compartilhem um mesmo canal de comunicação.

Observe o exemplo mostrado na Figura III.2 itens (a) e (b). O processo da aplicação Y possui dois canais de entrada. Logo, são utilizadas duas Unidades Registra Evento na monitoração e, conseqüentemente, o processo Y modificado possui dois canais de saída adicionais conectados ao Processo Buffer Remoto.

O procedimento executado por uma Unidade, no sentido de reconhecer e registrar um evento, se restringe ao envio de uma mensagem, contendo a identidade do canal de entrada associado ao evento para o Processo Buffer Remoto, correspondente. A implementação do *observador* representa, simplesmente, o posicionamento de uma primitiva do tipo *EnviaMsg* após aquela presente no cógido fonte do processo.

A Figura III.3 apresenta o aspecto de um programa de fonte antes e depois da inserção do observador. As variáveis envolvidas são:

- p_i : processo
- c_{in} : canais de entrada de p_i
- n : dimensão do vetor c_{in}
- M : mensagens recebidas através de c_{in}
- $Id.c_{in}$: identidades dos canais c_{in}
- $to.monitor$: canais que conectam p_i ao Processo Buffer Remoto

A segunda proposta de implementação, cada processo está vinculado a um *buffer* distinto onde são armazenados os rótulos dos eventos monitorados pelas Unidades

Programa fonte antes da inserção do observador

```
RecebeMsg (c_in [i], M), ∀i (0 ≤ i < n) →
   ( lista de comandos );
```

Programa fonte após a inserção do observador

```
RecebeMsg (c_in [i], M), ∀i (0 ≤ i < n) →
   BEGIN
      EnviaMsg (to.monitor [i], Id.c_in [i]);
      ( lista de comandos );
   END;
```

Figura III.3: Unidade Registra Evento - Implementação Orientada à Mensagem

Registra Evento. Os acessos ao *buffer* são controlados, exclusivamente, pelo *Procedimento Buffer Local* (BUFL) (seção III.4.2). O reconhecimento e registro de um evento é realizado através de uma chamada ao Procedimento, usando como parâmetros o rótulo correspondente e o endereço do *buffer* associado ao processo onde o evento ocorreu. A implementação do observador representa a inserção de uma chamada a BUFL posicionada imediatamente após as primitivas de recepção de mensagem encontradas no código fonte dos processos [III.2(c)].

A Figura III.4 mostra o código fonte de um programa genérico antes e depois do posicionamento de um observador. As variáveis encontradas são descritas a seguir.

- p_i : processo
- c_{in} : canais de entrada de p_i
- n : dimensão do vetor c_{in}

Programa fonte antes da inserção do observador

```
RecebeMsg (c_in [i], M), ∀i (0 ≤ i < n) →
    ⟨ lista de comandos ⟩;
```

Programa fonte após a inserção do observador

```
RecebeMsg (c_in [i], M), ∀i(0 ≤ i < n) →
    BEGIN
        BufferLocal (Id.c_in [i], buff);
        ⟨ lista de comandos ⟩;
    END;
```

Figura III.4: Unidade Registra Evento - Implementação Orientada ao Procedimento

- M : mensagens recebidas através de c_{in}
- $buff$: buffer associado a p_i
- $Id.c_{in}$: identidades dos canais c_{in}

III.4.2 Procedimento Buffer Local (BUFL)

O procedimento centraliza o acesso aos *buffers* usados no armazenamento dos rótulos dos eventos monitorados pelas *Unidades Registra Evento* (implementação orientada ao procedimento). Os parâmetros recebidos por BUFL são um rótulo *rot* e *o* endereço de uma variável do tipo *buffer*. A função do procedimento é copiar o valor de *rot* para a próxima entrada livre de *buffer*, e caso não existam mais entradas disponíveis, enviar os rótulos armazenados para o *Processo Disco Monitor* (seção III.4.4).

```
BEGIN
    buff := rot;
    IF buff cheio THEN
        EnviaMsg (to.disco, buff);
END;
```

Figura III.5: Algoritmo do Procedimento BuffeE Local

As variáveis do algoritmo, mostrado na Figura III.5, são:

- p_i : processo
- *rot* : rótulo do evento registrado
- *buff* : variável para o armazenamento doã rótulos
- *to.disco* : canal que conecta p_i ao Processo Disco Monitor

III.4.3 Processo Buffer Remoto (BUFR)

O Processo Buffer Remoto recebe e trata as mensagens enviadas pelas *Unidades Registra Evento* inseridas em um processo (implementação orientada à mensagem). O processamento das mensagens é semelhante aquele executado pelo *Procedimento Buffer Local.* O conteúdo de cada mensagem que, na verdade, representa o rótulo de um evento, é copiado para um *buffer* pertencente a BUFR. Caso não existam mais posições livres no *buffer*, então, o seu conteúdo, é enviado ao *Processo Disco Monitor*, permitindo, assim, o arquivamento de novos rótulos.

A execução de BUFR cessa, após o recebimento de um sinal de terminação, *stop*. Esta mensagem, que indica o fim das atividades do processo monitorado, precipita

a transmissão dos valores presentes em *buffer* e a propagação de um *stop* para o processo Disco Monitor.

As variáveis do algoritmo [Figura III.6] são descritas como:

- p_i : processo
- n : número de canais de entrada de p_i
- *to.disco* : canal que conceta BUFR ao Processo Disco Monitor
- *to.monitor* : canais que interligam as Unidades Registra Evento a BUFR
- M : mensagem recebida através de *to.monitor*
- *rot* : mensagem contendo o rótulo de um evento
- *stop* : sinal de terminação
- *buff* : variável para o armazenamento dos rótulos dos eventos
- *fim* : booleano que controla a terminaçao de BUFR

III.4.4 Processo Disco Monitor (DISCM)

O Processo Disco Monitor reune em um único arquivo os *buffers*, contendo os rótulos dos eventos monitorados durante uma execução. Os *buffers* são remetidos pelos processos *Buffer Remoto* ou pelos próprios processos da aplicação de acordo com a implementação em questão, isto é, orientada à mensagem ou orientada ao procedimento. Os rótulos associados a um processo são escritos em uma ordenação consistente com aquela recebida no *buffer*, juntamente com algumas informações de controle, que identificam o processo onde os rótulos foram coletados.

A execução de DISCM acaba após a recepção de um sinal de terminação, *stop*, através de cada um dos seus canais de entrada. Estas mensagens indicam o final da computação dos processos da aplicação e/ou processos Disco Remoto.

```
REPEAT
    RecebeMsg (to.monitor [i], M), ∀i (0 ≤ i < n) →
        CASE M
            rot :
                BEGIN
                    buff := M;
                    IF buff cheio THEN
                        EnviaMsg (to.disco, buff);
                END;
            stop :
                BEGIN
                    EnviaMsg (to.disco, buff);
                    EnviaMsg (to.disco, stop);
                    fim := TRUE;
                END;
UNTIL fim;
```

Figura III.6: Algoritmo do Processo Buffer Remoto

O algoritmo do Processo Disco Monitor [Figura III.7] manipula as seguintes variáveis:

- n : número de processos da aplicação
- *to.disco* : canais que ligam os processos da aplicação ou Processos Buffer Remoto a DISCM
- *buff* : mensagem contendo os rótulos dos eventos monitorados
- *stop* : sinal de terminação
- *fim* : booleano que controla a terminação de DISCM
- *cont* : contador que indica o número de *stops* recebidos

```
BEGIN
    Abre o arquivo de eventos;
    REPEAT
        RecebeMsg (to.disco [i], M), ∀i (0 ≤ i < n) →
            CASE M
                buff :
                    armazena os eventos associados ao processo i ;
                stop :
                    BEGIN
                        cont := cont + 1;
                        IF cont = n THEN
                            fim := TRUE;
                    END;
    UNTIL fim;
    Fecha o arquivo de eventos;
END;
```

Figura III.7: Algoritmo do Processo Disco Monitor

III.5 Algoritmo Repetidor

O Algoritmo Repetidor coordena a reexecução determinística de uma computação, a partir dos dados coletados pelo Algoritmo Monitor. Os parâmetros de entrada fornecidos à aplicação são, obrigatoriamente, os mesmos utilizados quando da etapa de monitoração. Essencialmente, os dados manipulados pelo algoritmo são as identidades ou rótulos dos eventos associados a recepção de mensagens.

O funcionamento do algoritmo pode ser descrito como: durante uma execução, as sondas bloqueiam as operações de recepção de mensagens sobre os canais de entrada dos processos. Os *buffers* armazenam rótulos de eventos, descrevendo uma ordenação, usada para determinar a ordem em que as sondas devem ser ativadas e, conseqüentemente, quando as mensagens devem ser recebidas. Cada processo possui seu *buffer* particular e, assim sendo, os rótulos pertencentes a processos diferentes permanecem separados. Inicialmente, os rótulos estão concentrados em um único

arquivo, que é analisado e distribuído entre os *buffers*. A ativação das sondas é conduzida paralelamente em todos os processos e, portanto, somente os eventos associados a cada processo, individualmente, são serializados. Não é imposta, em momento algum, uma ordem total [10] para a recepção das mensagens.

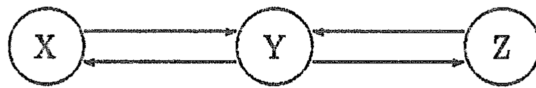
O algoritmo é composto pelas sondas, também chamadas de Unidades Habilita Evento que atuam acopladas aos canais de entrada dos processos, bloqueando a recepção de mensagens. Os processos Buffer Replay que manipulam os *buffers* e coordenam a reexecução de cada processo, separadamente. E, o processo Disco Replay que gerência o arquivo montado pelo Algoritmo Monitor, espalhando, entre os processo Buffer Replay, as informações usadas para orientar a reexecução dos processos.

A terminação do algoritmo é condicionada ao final da execução de cada um dos processos presentes no sistema. Primeiramente, os processos da aplicação acabam após transmitirem um sinal de terminação (*stop*) sobre somente um dos seus canais de saída, dentre aqueles ligados aos processos Buffer Replay. Este, por sua vez, acaba ao receber um único *stop*, que é propagado através do seu canal de saída. O processo Disco Replay cessa após receber um sinal de terminação em cada um dos seus canais de entrada. Ou seja, um *stop* por processo da aplicação. O fluxo das mensagens de terminação é apresentado na Figura III.8(b).

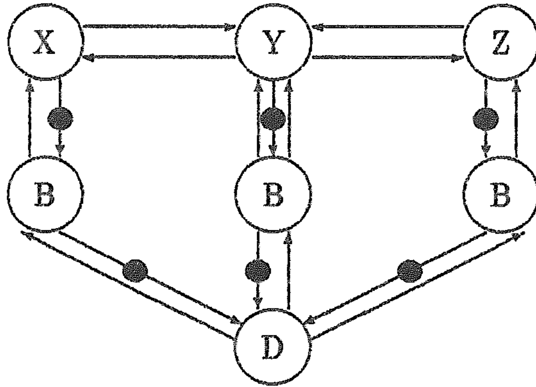
III.5.1 Unidade Habilita Evento

A *Unidade Habilita Evento* atua junto aos canais de entrada dos processos no sentido de habilitar ou bloquear a recepção de mensagens. Cada Unidade é composta por uma *barreira* e um *sinalizador*, que são inseridos imediatamente antes e depois as primitivas de recepção de mensagem, presentes no código fonte dos processos.

A ocorrência de eventos associados a canais ligados a uma Unidade envolve um procedimento formado por três etapas. A primeira, uma barreira é retirada, habi-



(a)

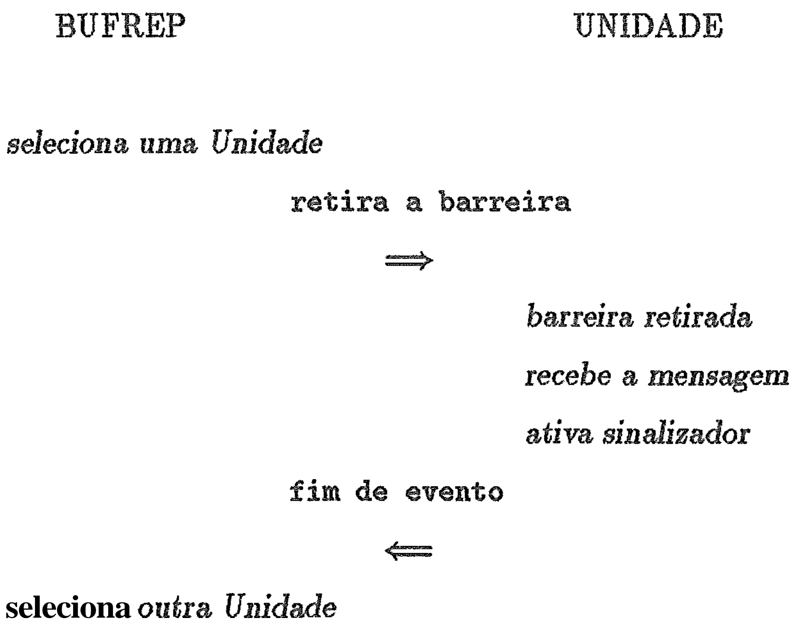


(b)

Figura III.8: Repesentação do Algoritmo Repetidor. (a) Grafo de processos da aplicação. (b) Grafo formado a partir da inserção dos processos do Algoritmo Repetidor. As letras B e D simbolizam os processos Buffer Replay e Disco Replay. Os círculos em negrito indicam o caminho percorrido pelo sinal de terminação.

litando um único canal de entrada. As demais barreiras permanecem erguidas até que o evento seja considerado finalizado. A segunda etapa, a mensagem é recebida, E, finalmente, a terceira, o sinalizador é ativado, caracterizando que a comunicação foi completada. No decorrer dos instantes entre a queda da barreira e o acionamento do sinalizador, nenhuma outra mensagem é recebida senão aquela pendente no canal habilitado.

Cada processo da aplicação é vinculado a uma imagem do *Processo Buffer Replay* (BUFREP) (seção III.5.2), que por sua vez, controla as Unidades Habilita Evento, posicionadas naquele processo, especificamente. O controle sobre as Unidades é exercido através da troca de comandos, estabelecendo o seguinte protocolo:



As Unidades associadas a um processo recebem comandos através de canais de comunicação distintos, existinto um igual número de canais e Unidades. As réplicas aos comandos são transmitidas usando um único canal, compartilhado por todas as Unidades presentes no processo.

Observe o exemplo mostrado na Figura III.8 itens (a) e (b). O processo da aplicação Y possui dois canais de entrada. Logo, são utilizadas duas Unidades Habilita

Evento e, conseqüentemente, o processo Y modificado possui três canais adicionais, dois de entrada e um de saída, conectados ao Processo Buffer Replay.

As implementações da barreira e do sinalizador representam, simplesmente, a colocação das primitivas *RecebeMsg* e *EnviaMsg*, inseridas, respectivamente, antes e depois, a cada uma das primitivas de recepção de mensagem, associadas aos canais de entrada do processo a ser controlado. A Figura III.9 exibe o aspecto de um programa genérico antes e depois da inserção da barreira e do sinalizador. As variáveis envolvidas são:

Variáveis:

- p_i : processo
- c_{in} : canais de entrada de p_i
- $to.replay$: canal que conecta p_i ao Processo Buffer Replay
- $from.replay$: canais que interligam o Processo Buffer Replay a p_i
- n : dimensão do vetor c_{in}
- M : mensagem recebida através de c_{in}
- $enable$: mensagem de habilitaçao
- $eoev$: mensagem de fim de evento

III.5.2 Processo Buffer Replay (BUFREP)

O Processo *Buffer Replay* controla o acionamento das Unidades Habilita Evento inseridas em um processo, garantinto que uma certa ordem parcial dos eventos associados a recepção de mensagens é estabelecida. Ao habilitar uma Unidade, o

Programa fonte antes da inserção da barreira e do sinalizador

RecebeMsg (c_{in} [i], M), $\forall i \ (0 \leq i < n) \rightarrow$
  ⟨ lista de comandos ⟩;

Programa fonte após a inserção da barreira e do sinalizador

RecebeMsg (from.replay [i], enable), $\forall i \ (0 \leq i < n) \rightarrow$
  RecebeMsg (c_{in}, M) $\rightarrow$
    BEGIN
      EnviaMsg (to.replay, eoev);
      ⟨ lista de comandos ⟩;
    END;

Figura III.9: Unidade Habilita Evento

Processo aguarda até que um sinal de fim de evento seja retornado, significando que uma nova Unidade pode ser ativada.

As Unidades são acionadas uma a uma, não existindo, portanto, duas ou mais Unidades habilitadas, simultaneamente, dentro do mesmo processo. Nenhuma restrição, no entanto, é imposta no que concerne a Unidades associadas a processos diferentes. A ordenação dos eventos é controlada para cada processo, individualmente e concorrentemente, e, assim sendo, não existe uma *ordem total* pré-determinada para a ocorrência dos eventos.

A seqüência de acionamento das Unidades é controlada a partir dos valores armazendos em uma variável do tipo *buffer*, declarada no contexto de BUFREP. Estes valores, que na verdade representam os rótulos dos eventos, determinam a escolha do canal de saída, através do qual, é enviado um sinal que habilita a Unidade selecionada. O carregamento do *buffer* é conduzido através da troca de mensagens com o *Processo Disco Replay* (seção III.5.3).

A execução do algoritmo do Processo Buffer Replay cessa após a recepção de um sinal de terminação, *stop*. Esta mensagem que indica o fim das atividades do processo controlado, dispara a propagação do mesmo sinal para o Processo Disco Replay. As variáveis encontrados no algoritmo [Figura III.10] são descritas como:

- p_i : processo
- n : número de canais de entrada de p_i
- *to.replay* : canal que interliga p_i a BUFREP
- *from.replay* : canais que ligam BUFREP a p_i
- *to.disco* : canal que conecta BUFREP ao Processo Disco Replay
- *from.disco* : canal que liga o Processo Disco Replay a BUFREP
- *req* : mensagem de pedido de eventos

```
REPEAT
    RecebeMsg (to.replay, M) →
        CASE M
            eoev :
                BEGIN
                    ptr := buff;
                    EnviaMsg (from.replay [ptr], enable);
                    IF buffer vazio THEN
                        BEGIN
                            EnviaMsg (to.disco, req);
                            RecebeMsg (from.disco, buff);
                        END;
                END;
            stop :
                BEGIN
                    EnviaMsg (to.disco, stop);
                    fim := TRUE;
                END;
UNTIL fim;
```

Figura III.10: Algoritmo do Processo Buffer Replay

- *eoev* : mensagem de fim de evento
- *stop* : mensagem de terminação
- *enable* : mensagem de habilitação
- *fim* : booleano que controla a terminação de BUFREP
- *buff* : variável para o armazenamento dos rótulos dos eventos
- *ptr* : apontador para o próximo evento em *buff*

III.5.3 Processo Disco Replay (DISCR)

O *Processo* Disco *Replay* analisa o arquivo montado pelo Algoritmo Monitor e distribui, entre os processos Buffer Replay, os rótulos dos eventos correspondentes a cada um dos processos da aplicação. A distribuição e transferência dos rótulos é conduzida mediante a recepção de um sinal de *pedido de eventos*. A resposta a este sinal é enviada sobre o canal de comunicação que conecta DISCR a processo Buffer Replay onde o pedido se originou.

A execução de DISCR cessa quando é recebido um sinal de terminação, *stop*, através de cada um dos seus canais de entrada. Estas mensagens caracterizam o término da computação dos processos da aplicação e Buffer Replay.

O algoritmo do Processo Disco Replay [Figura III.11] manipula as variáveis descritas a seguir.

- n : número de processos da aplicação
- $to.disco$: canais que conectam os Processos Buffer Replay a DISCR
- $from.disco$: canais que ligam DISCR aos Processos Buffer Replay
- M : mensagens recebidas através de $to.disco$
- req : mensagem de pedido de eventos
- $stop$: mensagem de terminação
- $cont$: contador que indica o número de $stops$ recebidos
- $buff$: variável para o armazenamento dos rótulos dos eventos

```
BEGIN
    Abre o arquivo de eventos;
    Leitura do arquivo;
    REPEAT
        RecebeMsg (to.disco [i], M), ∀i(0 ≤ i < n) →
            CASE M
                req :
                    BEGIN
                        buff := eventos do processo i;
                        EnviaMsg (from.disco [i], buff);
                    END;
                stop :
                    BEGIN
                        cont := cont + 1;
                        IF cont = n THEN
                            fim := TRUE;
                    END;
    UNTIL fim;
    Fecha o arquivo de eventos;
END;
```

Figura III.11: Algoritmo do Processo Disco Replay

III.6 Mapeamento dos Processos

O mapeamento dos processos especificados pelos algoritmos Monitor e Repetidor em um sistema distribuído ponto-a-ponto é bastante flexível. Os processos da aplicação continuam sendo configurados nos mesmos processadores, como previsto antes da inserção dos processos de monitoração e reexecução. Estes, por sua vez, podem ser posicionadas em qualquer um dos processadores do sistema. Os processos da Interface de Comunicação (ICOM), descritos no Capítulo II, resolvem os problemas relativos a adaptação do grafo de processos a configuração desejada. Cabe, no entanto, desenvolver algumas considerações no que concerne ao mapeamento dos processos dos algoritmo Monitor e Repetidor de forma a reduzir as perturbações impostas sobre a aplicação.

Os processos Disco Monitor e Disco Replay devem ser alocados, preferencialmente, em um processador conectado diretamente a unidade de disco, diminuindo, com isso, o tempo gasto nas operações de escrita e leitura do arquivo de eventos.

Os processos Buffer Remoto e Buffer Replay devem ser mapeados nos mesmos processadores onde residem os processos da aplicação, no sentido de minimizar o trânsito de mensagens entre processadores. O custo da comunicação envolvendo mensagens que transitam na rede de canais físicos é superior aquele onde as mensagens somente circulam em canais lógicos. Além disso, as mensagens entre processos residentes em processadores distintos passam, obrigatoriamente, pela ICOM. E, assim sendo, estas mensagens são manipuladas por vários processos até atingirem seus destinos, aumentando o tempo gasto na comunicação.

III.7 Prevenção de Deadlock

Como descrito anteriormente, o algoritmo Repetidor coordena a reexecução de uma computação a partir das informações coletadas pelo algoritmo Monitor. Essencial-

mente, a ocorrência dos eventos associadoR a recepção de mensagens é controlada de forma que a mesma ordem parcial estabelecida durante a monitoração seja repetida. Em certos casos, o controle exercido pelo algoritmo pode levar a aplicação a entrar em *deadlock*, como discutido a seguir.

Quando a aplicação e o algoritmo Repetidor são mapeados em um sistema distribuído, os canais de comunicação conectados a Interface de Comunicação (ICOM) assumem uma capacidade de armazenamento limitada. Com isso, um processo pode enviar um certo número de mensagens sobre um canal sem se bloquear, caracterizando uma comunicação assíncrona (Capítulo II). Considere, então, a seguinte situação:

Seja p_i um processo associado aos canais de entrada c_1 e c_2, conectados a ICOM. As mensagens m_1, m_2 e m_3, destinadas a p_i, estão vinculadas aos canais c_1, c_2 e c_1, respectivamente. Suponha que o algoritmo Repetidor determina que as mensagens devem ser recebidas por p_i na ordem:

$$\langle m_2 \rightarrow m_1 \rightarrow m_3 \rangle$$

Suponha também que a ordem de chegada das mensagens ao processador onde p_i reside é:

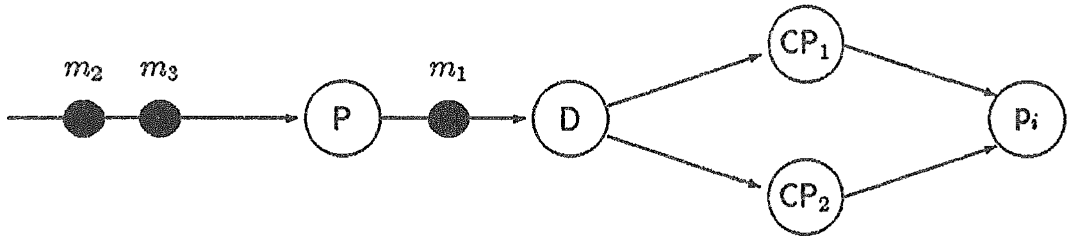
$$\langle m_1 \rightarrow m_3 \rightarrow m_2 \rangle$$

O caminho descrito pelas mensagens em direção ao processo p_i é formado pelas etapas [Figura III.12]:

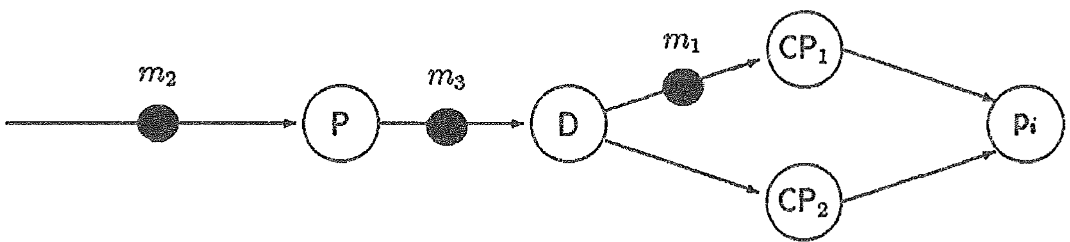
Etapa 1. m_1 é recebida pelo Processador Virtual de Comunicação (PVC) e é transferida para o DeMultiplex (DUX);

Etapa 2. m_1 é propagada para o Conversor de Protocolo (CP_1), conectado a c_1. m_3 é recebida pelo PVC e encaminhada para o DUX;

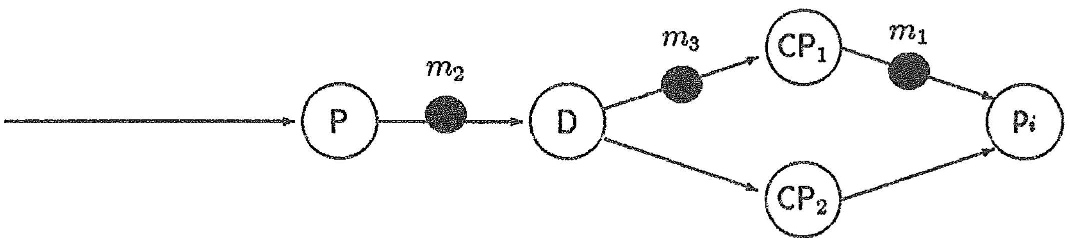
Etapa 3. m_1 e m_3 estão retidas em CP_1 e DUX. m_2 é recebida pelo PVC;



Etapa 1



Etapa 2



Etapa 3

Figura III.12: Exemplo de uma Situação onde Ocorre Deadlock. As letras P, D e CO simbolizam o Processador Virtual de Comunicação e os processos DeMultiplex e Conversor de Protocolo

A expansão da capacidade de armazenamento do canal c_1 possibilita que m_3 seja transmitida antes de m_1 ser recebida. O PVC despacha as mensagens para o DUX de acordo com a ordem de chegada ao processador e, assim sendo, m_3 é despachada antes de m_2, obrigatoriamente. Para que m_2 seja transferida para o DUX, primeiramente, m_1 deve ser propagada para p_i e m_3 para CP$_1$. O algoritmo Repetidor, no entanto, impede que m_1 seja recebida antes de m_2 e, portanto, os processos p_i, CP$_1$ e DUX ficam bloqueados, indefinidamente, paralizando a execução da aplicação [1].

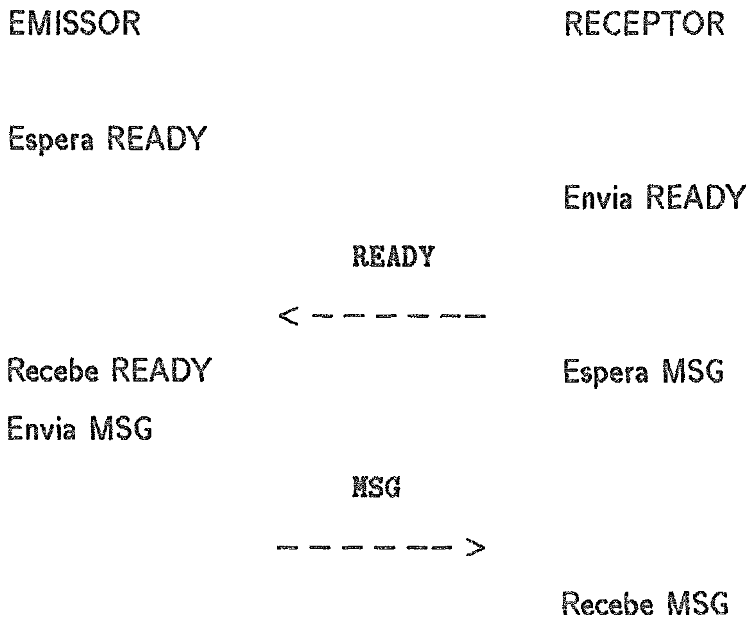
Ainda neste exemplo, suponha que c_1 tem capacidade de armazenamento *zero*. Assim, m_1 não é transmitida até que m_2 seja recebida, da mesma forma que, m_3 não é propagada antes de m_1. A ordem de envio das mensagens é consistente com aquela imposta pelo algoritmo Repetidor e as comunicações são processadas tal qual a ICOM não existisse. Suponha agora que c_1 tem capacidade *um*. A mensagem m_1 segue o caminho $PVC \rightarrow DUX \rightarrow CP_1$, permanecendo retira em CP$_1$ até que m_2 seja aceita por p_i. A mensagem m_3 não é transferida antes de m_1 ser recebida. Com isso, m_2 é propagada através do $PVC \rightarrow DUX \rightarrow CP_2$, chegando a p_i. Ainda que a ordem de processamento das mensagens seja diferente daquela seguida pelo algoritmo Repetidor, nenhuma situação de interbloqueio é possível.

Observando as considerações desenvolvidas no exemplo, um mecanismo de prevenção de *deadlock* deve limitar a capacidade de armazenamento dos canais, conectados a ICOM, aos valores *zero* ou *um*. Cabe destacar que, o mecanismo de prevenção de *deadlock* aqui proposto não é unico. Outras abordagens, baseadas inclusive na expansão da capacidade de armazenamento dos canais, são igualmente válidas e eficazes.

A fim de simular *capacidade zero*, o mecanismo deve garantir que o processo emissor permaneça bloqueado até o receptor estar pronto para receber a mensagem. Um

[1] Os processos Conversor de Protocolo e DeMultiplex tem capacidade para armazenar nina única mensagem, somente

protocolo distribuído, baseado na troca de um sinal de sincronização, *READY*, entre os processos emissor e receptor, simula uma comunicação síncrona, ainda que, o ambiente seja assíncrono [1].



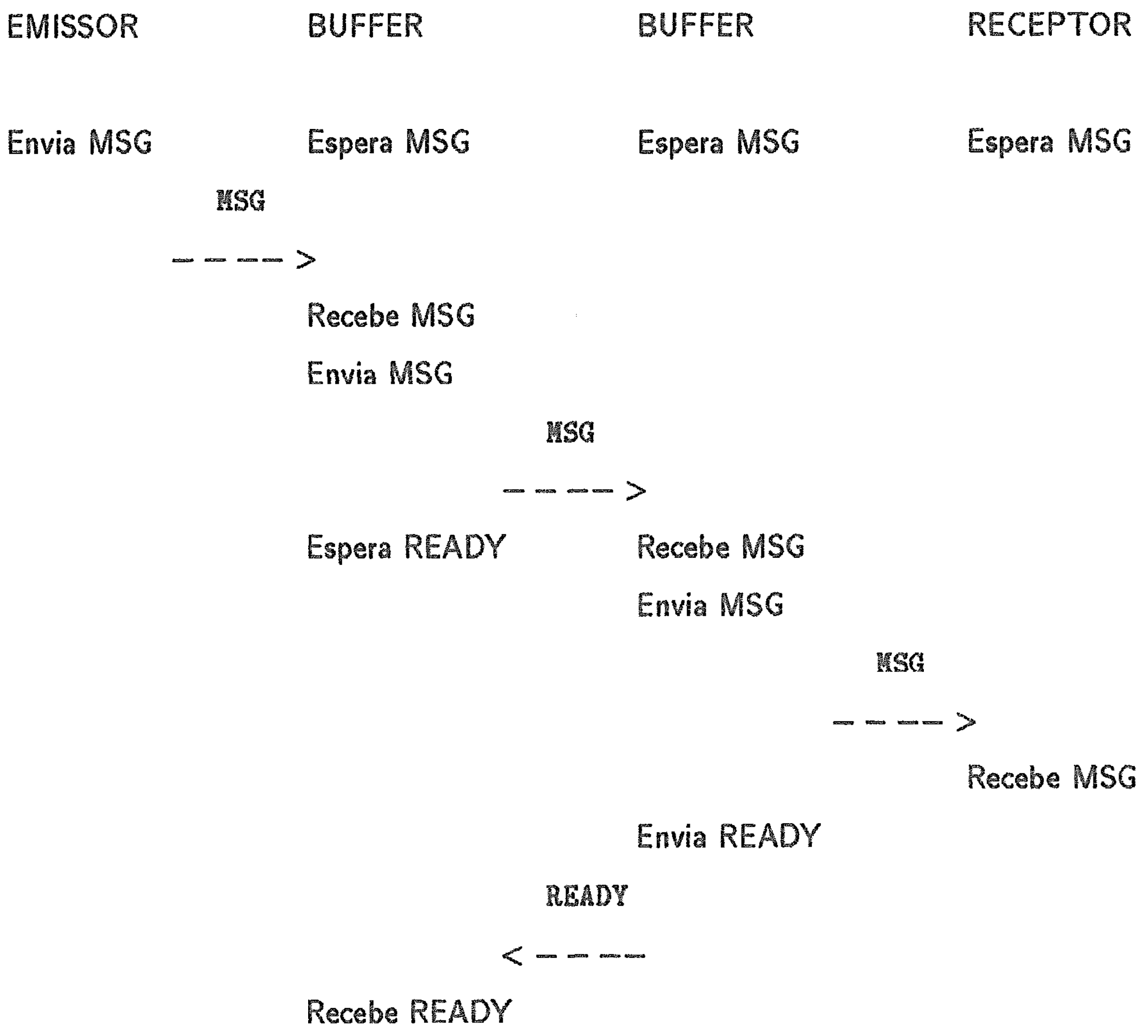
O projeto do protocolo utiliza um canal de comunicação adicional, ligando os processos emissor e receptor, dedicado, exclusivamente, a transportar o sinal *READY*. O código fonte do emissor e do receptor é modificado de forma a comportar um dispositivo de sincronização, acoplado a cada um dos canais conectados a ICOM. Com isso, a estrutura original dos processos é alterada, passando a ser dependente do tipo de mapeamento adotado. Qualquer mudança na alocação dos processos implica, provavelmente, na correção do programa. A Figura III.13 apresenta um programa genérico, exemplicando a implementação do protocolo.

A simulação da *capacidade um*, *o* mecanismo deve garantir que o processo emissor pode enviar uma única mensagem sem se bloquear. Assim sendo, uma segunda mensagem somente pode ser transmitida mediante ao término da recepção da primeira. Para expandir a capacidade de *zero* para *um*, dois processos do tipo *buffer* são inseridos entre o emissor e o receptor. Um protocolo distribuído, semelhante aquele apresentado anteriormente, é baseado na troca de sinais de sincronização, *READY*,

```
{* Processo Emissor *}

REPEAT
    ⋮
    RecebeMsg (to.emissor, READY) →
        EnviaMsg (c, MSG);
    ⋮
UNTIL fim;


{* Processo Receptor *}

REPEAT
    ⋮
    EnviaMsg (to.emissor, READY);
    RecebeMsg (c, MSG) →
        ⟨ lista de comandos ⟩
    ⋮
UNTIL fim;
```

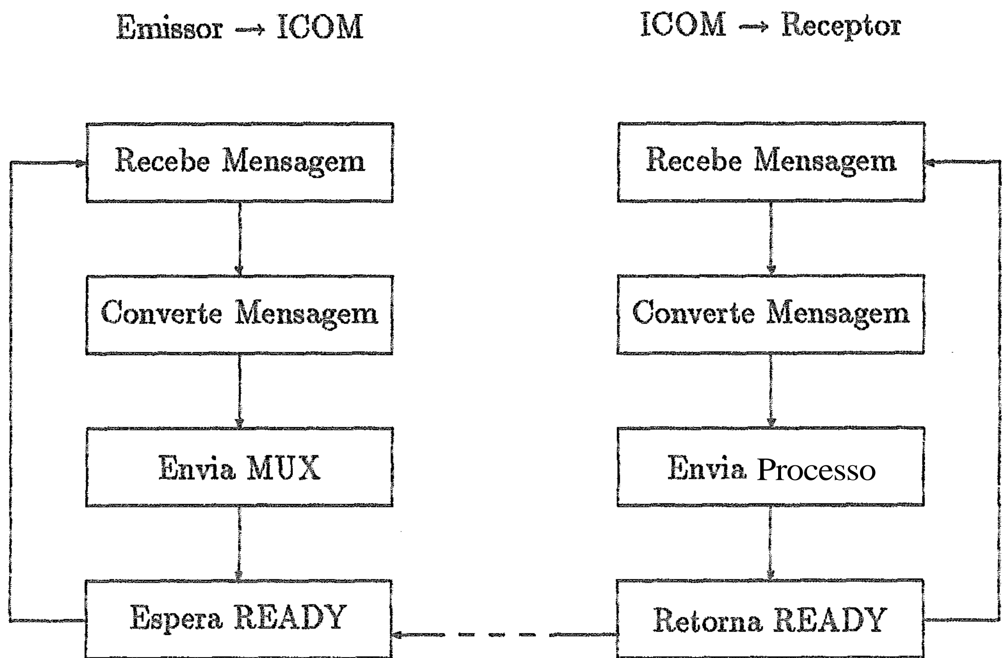
Figura III.13: Implementação do Protocolo Distribuído que Simula Capacidade Zero

entre os processos *buffer*.



O projeto do protocolo distribuído que simula *capacidade um* utiliza um versão modificada da Interface de Comunicação que implementa, de forma transparente, o mecanismo apresentado. Na verdade, somente o processo Conversor de Protocolo (CP) é ajustado no sentido de comportar as funções de sincronização e de compatibilização dos formatos da mensagens trocadas entre a aplicação e a ICOM. Aqui, o CP assume as características antes atribuídas aos processos do tipo *buffer*. Diferentemente da simulação da *capacidade zero*, os processos da aplicação conservam sua estrutura original, livre de quaisquer dispositivos adicionais de sincronização. Novamente, o mapeamento da aplicação em um sistema distribuído é independente da construção dos processos. Em virtude disto, o mecanismo de simulação da *ca-*

pacidade um foi escolhido em detrimento daquele de *capacidade zero*. O esquema, apresentado a seguir, ilustra o funcionamento do processo Conversor de Protocolo. A Figura III.14 mostra o algoritmo do CP, usando o mecanismo de sincronização discutido.



```
{* Processo Conversor do Protocolo do Processo *}

REPEAT
    RecebeMsg (from.processo, MSG) →
        BEGIN
            MSG' := ConverteProtocolo (MSG);
            EnviaMsg (to.MUX, MSG');
            RecebeMsg (from.MUX, READY);
        END;
UNTIL fim;


{* Processo Conversor do Protocolo do PVC *}

REPEAT
    RecebeMsg (from.DUX, MSG) →
        BEGIN
            MSG' := ConverteProtocolo (MSG);
            EnviaMsg (to.processo, MSG');
            EnviaMsg (to.MUX, READY);
        END;
UNTIL fim;
```

Figura III.14: Implementação do Protocolo Distribuído que Simula Capacidade Um

Capítulo IV

Análise dos Resultados

Os processos e procedimentos especificados pelos algoritmos Monitor e Repetidor, detalhados nos capítulos anteriores, foram implementados usando a linguagem de programação *occam* [7] [18] [19] e testados em um hipercubo formado por processadores do tipo *transputer* [17]. Os testes realizados objetivaram determinar o grau de interferência imposto às aplicações submetidas ao mecanismo. A quantificação da interferência foi feita a partir de comparações entre medidas de tempo de execução, envolvendo uma aplicação individualmente e quando esta é superposta aos procedimentos de monitoração e reexecução.

Em virtude da natureza do mecanismo, que inspeciona os eventos associados a recepção de mensagens, os testes consideram duas classes de aplicações. A primeira, denominada *fracamen.de acoplada*, o custo da comunicação 6 relativamente pequeno, se comparado a carga computacional. E a segunda, chamada *fortemente acoplada*, o custo da comunicação é relativamente grande, em face a carga computacional. Para ambas as classes estudadas, foram coletadas medidas de tempo para variações de parâmetros como: tamanho dos dados de entrada e número de processadores. Os resultados são exibidos em gráficos e tabelas.

As aplicações estudadas foram desenvolvidas na linguagem *occam* e representam duas implementações distribuídas de um mesmo algoritmo de multiplicação de matrizes.

As próximas seções descrevem e apresentam o ambiente cios testes, as aplicações estudadas, as medidas de tempo levantadas e as avaliações qualitativa e quantitativa das pertubações introduzidos pelo mecanismo [11].

IV.1 Computador Paralelo NCP1

O NCP1 é uma máquina paralela de memória distribuída baseada na arquitetura hipercúbica, que utiliza os microprocessadores Intel i860, unidade de processamento vetorial, e transputer T800, unidade de processamento e comunicação. O protótipo desenvolvido apresenta oito nós de processamento, podendo alcançar um desempenho de até 320 MIPS (*millons of intructions per second*) e 640 MFLOPS (*millons of floating-point operations per second*).

A configuração adotada nos testes é constituída por oito unidades de processamento e comunicação, conectadas ponto-a-ponto através de *links* bidirecionais de comunicação, e um microcomputador do tipo IBM-PC/XT, utilizado como dispositivo de entrada e saída, que oferece acesso as unidades de disco ou terminal gráfico. Cada unidade apresenta um transputer T800, operando a 20 Mhz, quatro *links* seriais de comunicação de 20 Mbits/s, 64 Kbytes de memória EPROM e 2 Mbytes de memória RAM [23] [24].

IV.2 Aplicações Estudadas

As aplicações desenvolvidas para a avaliação do mecanismo de reexecução representam duas implementações distribuídas de um algoritmo de multiplicação de matrizes. O fluxo computacional deste algoritmo é insensível a ordem de recepção das mensagens, o que permite a comparação entre as execuções *monitorada* e *não-monitorada*. Um outro aspecto diz respeito a carga computacional. As aplicações estudadas executam a mesma tarefa. A diferença reside, exclusivamente, na forma

de interação entre os processos, o que determina o número de mensagens trocadas e, conseqüentemente, exercita o grau de atuação do mecanismo.

O funcionamento do algoritmo pode ser descrito como: o processo *Centralizador* particiona as matrizes multiplicando A e multiplicador B em n sub-matrizes cada, formadas, respectivamente, pelas linhas de A e colunas de B. As $2n$ matrizes resultantes são distribuídas entre n processos *Multiplica*, encarregados de calcular uma região distinta da matriz solução C. O cálculo de uma sub-matriz solução C' envolve a multiplicação de uma sub-matriz de A, A', pela matriz B. Cada processo Multiplica manipula todos os elementos de B, ainda que, inicialmente, somente tenha acesso a uma região desta matriz. Os demais elementos de B, utilizados pelos processos Multiplica, são fornecidos pelos processos *Buffer*, dedicados a armazenar as cópias das sub-matrizes de B. Com isso, o processamento é dissociado da comunicação, permtindo uma maior eficiência do programa [20]. Ao final da computação os processos Multiplica retornam ao Centralizador as sub-matrizes calculadas.

O exemplo, apresentado a seguir, e a Figura IV.1 esclarecem o que foi descrito.

Exemplo:

As matrizes A, B e C são divididas em duas regiões cada.

$$\left(\begin{array}{c} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \\ \\ \begin{bmatrix} a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \end{array} \right) * \left(\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix} \begin{bmatrix} b_{13} & b_{14} \\ b_{23} & b_{24} \\ b_{33} & b_{34} \\ b_{43} & b_{44} \end{bmatrix} \right) = \left(\begin{array}{c} \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \end{bmatrix} \\ \\ \begin{bmatrix} c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix} \end{array} \right)$$

Um dos processos Multiplica recebe as sub-matrizes A' e B'.

$$A' = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix} \text{ e } B' = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ b_{41} & b_{42} \end{pmatrix}$$

E calcula o valor resultante da multiplicação de A' e B, correspondente a sub-matriz C'.

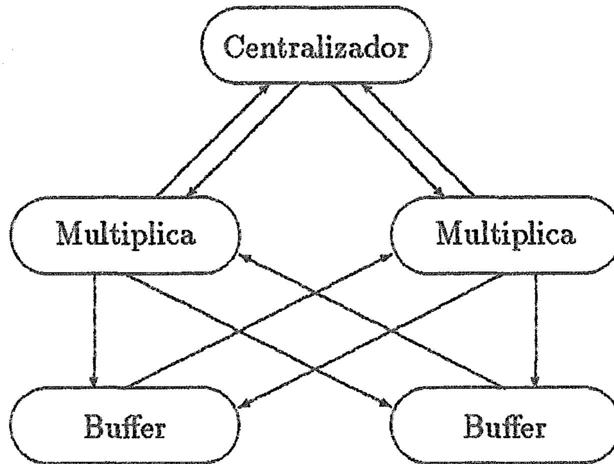


Figura IV.1: Exemplo do Grafo de Processos do Algoritmo de Multiplicação

$$C' = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \end{pmatrix}$$

Onde, para $n = 4$, vem

$$c_{ij} = \sum_{k=1}^{n} a_{ij} * b_{kj}, \quad com \quad \begin{cases} 1 \leq i \leq \frac{n}{2} \\ 1 \leq j \leq \frac{n}{2} \end{cases}$$

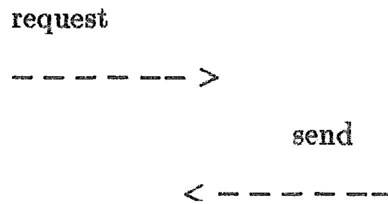
Para calcular o valor de C', o processo Multiplica pede os elementos da sub-matriz B", armazenados no processo Buffer, sendo

$$B" = \begin{pmatrix} b_{13} & b_{14} \\ b_{23} & b_{24} \\ b_{33} & b_{34} \\ b_{43} & b_{44} \end{pmatrix}$$

Analogamente, o segundo processo Multiplica manipula as demais sub-matrizes.

O algoritmo de Multiplicação considera duas propostas de implementação, referenciadas como *bloco* e *elemento*, que se diferenciam, unicamente, **pela** forma como **a**

sub-matriz B' é transferida dos processos BuffeE para os processos Multiplica. As mensagens trocadas entre estes processos estabelecem um protocolo síncrono do tipo,



onde cada unidade de informação trocada entre estes processos implica que duas mensagens sejam enviadas.

A proposta de implementação por *bloco*, a sub-matriz B' é transmitida em um único pacote, somando, portanto, duas mensagens permutadas, uma de pedido (*request*) e a outra de reposta (*send*), contendo uma cópia de B'. Supondo que as matrizes A e B, de dimensão $n \ x \ n$, são divididas em k sub-matrizes, então o número de mensagens recebidas pelos processos Multiplica e Buffer é dado pela expressão:

$$num = k((k-1) + (k-1)) = 2k^2 - 2k$$

onde a complexidade de mensagens vale $O(k^2)$, independente da dimensão das matrizes A e B. Da mesma forma, o número total de comunicações entre todos os processos é dado pela função:

$$C(n,k) = 2(k^2 + k) \qquad \text{(IV.1)}$$

A proposta de implementação por *elemento*, a sub-matriz B' é transferida elemento a elemento, significando que são permutadas pq mensagens de pedido e pq mensagens de respostas, totalizando $2pq$ mensagens, onde p e q representam os números de linhas e colunas de B', respectivamente. Considerando, portanto, que as matrizes A e B, de dimensões $n \ x \ n$, são particionadas em k sub-matrizes, então o número de mensagens recebidas pelos processos Multiplica e Buffer é dado pela equação:

$$num = k\left(\left(\frac{n^2}{k} \right)(k-1) + \left(\frac{n^2}{k} \right)(k-1) \right) = 2n^2(k-1)$$

onde a complexidade das mensagens vale $O(n^2k)$, crescendo com o quadrado da dimensão das matrizes A e B. O número de mensagens trocadas entre todos os processos é dado pela função:

$$C(n,k) = 2n^2(k-1) + 4k \quad \text{(IV.2)}$$

IV.3 Avaliação

A avaliação do grau de perturbação imposto pelo mecanismo de reexecução foi desenvolvida a partir de comparações entre medidas de tempo de execução, envolvendo uma aplicação, individualmente, e quando esta é superposta aos procedimentos de monitoração e reexecução. As medidas coletadas desconsideram o tempo gasto pelo mecanismo em operações de entrada e saída (I/O), por ser este parâmetro estritamente dependente da tecnologia da máquina hospedeira utilizada. Como ilustração, foram levantados alguns valores que consideram o tempo total (com I/O) para um hospedeiro do tipo IBM-PC/XT.

Os testes realizados objetivam quantificar o *overhead* introduzido pelo mecanismo para duas classes de aplicações. A primeira, denominada *fortemente acoplada*, a comunicação entre os processos da aplicação é intensiva e, conseqüentemente, o *overhead* tende a se aproximar do seu limite superior. Esta classe é representada pelo algoritmo de *Multiplicação por Elemento*. A segunda, chamada fracamente acoplada, a comunicação entre os processos da aplicação é escassa e, portanto, *overhead* tende a se aproximar do seu limite inferior. Esta classe é simbolizada pelo algoritmo de *Multiplicação por Bloco*.

As medidas de tempo levantadas são apresentadas em tabelas e gráficos que correlacionam parâmetros como número de processadores, dimensão das matrizes de entrada, número de eventos processados, entre outros. Primeiramente, são analisadas as variações do *overhead* em função do número de processadores presentes na rede. A medida que o número de processadores cresce, o custo da comunicação,

que envolve trânsito e roteamento das mensagens, aumenta. Assim sendo, a análise tenciona exercitar o comportamento do mecanismo em relação ao custo da comunicação, considerando a carga computacional constante [1]. Em seguida, são estudadas as flutuações no *overhead* em função da dimensão das mantrizes de entrada. Este item é desmenbrado em dois levantamentos. O primeiro, que utiliza o algoritmo de Multiplicação por Bloco, determina o comportamento do *overhead* para uma carga computacional crescente, porém, com um número de eventos constante [2]. O segundo, que corresponde ao algoritmo de Multiplicação por Elemento, caracteriza a evolução do *overhead*, para uma carga computacional e um número de eventos crescentes.

Os parâmetros envolvidos no estudo do comportamento do *overhead* são simbolizados pelas seguintes legendas:

- *EXEC:* tempo de execução da aplicação;
- *ORP:* tempo de execução da aplicação durante a monitoração para a implementação orientada ao procedimento (Seção III.4);
- *ORM:* tempo de execução da aplicação durante monitoração para a implementação orientada à mensagem (Seção III.4);
- *REXEC:* tempo de execução da aplicação durante a reexecução (Seção III.5);
- *Dim:* matriz quadrada de dimensão $n \ x \ n$;
- *Proc:* número de processadores (seção IV.1);
- *E*ventos: número de eventos processados (Equações IV.1 e IV.2);
- *Overhead:* percentual relacionando os tempos de execução da aplicação;

$$Overhead\,(\%) = \frac{x - EXEC}{EXEC} * \mathbf{100}, \ onde \ x \in ORP, \ ORM \ ou \ REXEC$$

[1] A carga computacional varia deacordo com a dimensão das matrizes fornecidas aos algoritmos de Multiplicação

[2] O número de eventos inspecionados gelo mecanismo é dado gela Equação IV.1 que é independente da dimensão das matrizes de entrada

Os testes realizados em um processador utilizaram duas implementações concorrentes correspondentes aos algoritmos de Multiplicação por Bloco e por Elemento, onde somente foram eliminados os processos associados a Interface de Comunicação (ICOM). Neste caso, o problema foi dividido em quatro sub-problemas e, portanto, as matrizes Multiplicando e Multiplicador também foram quebradas em quatro sub-matrizes. Os demais testes, envolvendo múltiplos processadores, adotaram um particionamento coincidente com o número de nós da máquina usada, ou seja, n processadores correspondem a n sub-problemas.

As unidades para o armazenamento dos eventos, *buffers*, declaradas no contexto dos processos *Buffer Monitor* (Seções III.4.3 e III.4.2) e *Buffer Replay* (Seção III.5.2), foram dimensionadas para comportar até 512 rótulos, onde cada rótulo ocupa o espaço equivalente a um *byte*. Os testes foram realizados para matrizes com dimensões variando entre 8 x 8 e 160 x 160, para 1, 2, 4 e 8 processadores, organizados em topologias hipercúbicas. As medidas foram coletadas em *ticks* do *transputer* [3], ainda que, aqui, os valores apareçam escritos em segundos.

IV.3.1 Overhead x Custo da Comunicação

Aplicações Fortemente Acopladas

As aplicações fortemente acopladas apresentam um custo de comunicação relativamente alto, quando comparado ao tempo de processamento, devido ao grande número de mensagens trocadas entre os processos. Quando estas aplicações são mapeadas em redes com múltiplos processadores, o custo da comunicação, que envolve o roteamento e trânsito das mensagens através dos canais físicos, tende a se elevar a medida que cresce o número de processadores mapeados. Considere, então, a seguinte situação: uma aplicação fortemente acoplada é configurada em uma máquina paralela formada por n processadores, onde residem n processos (1 processo por processador), que recebem m mensagens cada. Quando esta aplicação é submetida

[3] Um tick do transputer equivale a 64 μs

ao mecanismo de reexecução, o número de rótulos de eventos processados é igual nm. Admitindo que o *buffer* para armazenamento dos eventos comporta 512 *bytes*, vem:

• Para o procedimento de monitoração na implementação orientada ao procedimento (ORP), o número de mensagens trocadas entre os processos do mecanismo é dado por $q = \frac{m}{512} * n$. O que representa, apenas,

$$p\,(\%) = \frac{q}{mn + q} * 100 = \frac{\frac{mn}{512}}{mn + \frac{mn}{512}} * 100 = \qquad \text{(IV.3)}$$

0,19% do número total de mensagens, excluindo, deste total, as mensagens transmitidas pela Interface de Comunicação. Aqui, o custo adicional de comunicação, referente a interação dos processos de monitoração, 6 relativamente pequeno, se comparado aquele inerente à construção da aplicação. O *overhead* se deve, principalmente, aos atrasos introduzidos pelas Unidades Registra Eventos (Seção III.4.1).

O tempo consumido por uma Unidade é bastante reduzido, envolvendo

$$\begin{pmatrix} \text{chamada ao} \\ \textit{procedimento} \\ \textit{Buffer Local} \end{pmatrix} + \begin{pmatrix} \textit{cópia do} \\ \textit{rótulo para} \\ \textit{o buffer} \end{pmatrix} + \begin{pmatrix} \textit{retorno} \\ \textit{ao} \\ \text{processo} \end{pmatrix}$$

Portanto, o *overhead* durante a monitoração na implementação orientada ao procedimento 6, também, bastante pequeno para um número qualquer de processadores. AR Tabelas V.1 e '6.2 exemplificam o que foi discutido. Para matrizes de entrada de dimensões 80 x 80, o *overhead* para 1, 2, 4 e 8 processadores oscila entre 5% e 6%. E para matrizes 160 x 160 entre 3% e 6%.

• Para o procedimento de monitoração na implementação orientada à mensagem, o número de mensagens trocadas entre os processos do mecanismo é igual a

$$num = \frac{m}{512} * n + nm \qquad \text{(IV.4)}$$

desconsiderando as comunicações pertencentes a Interface de Comunicação. Diferentemente da implementação orientada ao procedimento, aqui, o número de mensagens associadas aos processos de monitoração é comparável aquele referente a aplicação, isto é, nm.

A parcela dominante da Equação IV.4 (nm) é relativa as mensagens enviadas pelas Unidades Registra Evento aos processos Buffer Remoto (Seção III.4.3), com a finalidade de informar a ocorrência dos eventos. Considerando que os processos Buffer são alocados nos mesmos processadores onde residem os processos monitorados, as nm mensagens transitam, exclusivamente, em canais lógicos e, portanto, o tempo gasto nestas comunicações não é sensível ao aumento do número de processadores. A parcela restante da Equação IV.4, $\frac{m}{512} * n$, representa uma fração diminuta do total de mensagens.

O *overhead* introduzido durante a monitoração, mais uma vez, se deve, grandemente, aos atrasos introduzidos pelas Unidades Registra Evento. Agora, no entanto, o tempo de processamento consumido por uma Unidade é relevante e engloba a transmissão da mensagem mais o tempo de sincronização dos processos. Antecipando o que será detalhado futuramente, a implementação orientada à mensagem demonstra uma grande sensibilidade as variações no número de eventos processados. As Tabelas V.1 e V.2 ilustram o comportamento discutido. Para 1 processador o *overhead* vale 50.28%, para matrizes 80 x 80, contra 9.73%, 17.99% e 32.52%, correspondentes a 2, 4 e 8 processadores, respectivamente. Como previsto, o atraso introduzido pelas Unidades Registra Evento na implementação orientada à mensagem é mais significativo do que na implementação orientada ao procedimento. Para matrizes 160 x 160 o *overhead* na ORM ($\approx$ 15%) supera em 9% o *overhead* na ORP, para um configuração formada por 4 processadores.

- Para o procedimento de reexecução a número de mensagens trocadas entre os processos do mecanismo é dado pela equação

$$num = \underbrace{2n * \frac{m}{512}}_{Q_1} + \underbrace{2nm}_{Q_2} + \underbrace{nm}_{Q_3} \tag{IV.5}$$

excluindo, mais uma vez, as comunicações associadas a Interface de Comunicação. Neste caso, o número de mensagens manipuladas pelos processos de reexecução supera, em muito (em tomo de três vezes), aquele referente a aplicação. A parcela Q_1, que corresponde as mensagens trocadas entre os processos Buffer Replay e Disco

Replay, discutidos nas Seções III.5.2 e III.5.3, representa, apenas,

$$p\,(\%) = \frac{Q_1}{Q_1 + Q_2 + Q_3 + nm} * 100 = \frac{\frac{nm}{256}}{4nm + \frac{nm}{256}} * 100 =$$

0.097% do número total de mensagens transmitidas. O custo adicional de comunicação, referente a Q_1, é relativamente pequeno, se comparado aquele inerente as parcelas Q_2 e Q_3. Assim sendo, a avaliação, aqui apresentada, despresa a contribuição de Q_1 para o valor do *overhead*.

A parcela dominante da Equação IV.5, Q_2, é relativa as mensagens permutadas entre as Unidades Habilita Evento (Seção III.5.1) e os processos Buffer Replay, no sentido de orientar a ordem de ocorrência dos eventos. Considerando que os processos Buffer são, preferencialmente, alocados nos mesmos processadores onde residem os processos da aplicação, as $2nm$ mensagens transitam, unicamente, em canais lógicos e, portanto, Q_2 é insensível ao número de processadores presentes no sistema. O tempo de processamento consumido por uma Unidade Habilita Evento, entretanto, é significativo e envolve o somatório dos tempos de

$$\begin{pmatrix} \textit{sincronização} \\ \text{dos} \\ \text{processos} \end{pmatrix} + \begin{pmatrix} \text{transmissão} \\ \text{da} \\ \text{mensagem} \end{pmatrix} + \begin{pmatrix} \textit{sincronização} \\ \text{dos} \\ \text{processos} \end{pmatrix} + \begin{pmatrix} \textit{recepção} \\ \textit{da} \\ \textit{mensagem} \end{pmatrix}$$

Observe os resultados dos testes realizados em 1 processador (Tabelas V.1 e V.2), que demonstram o impacto dos atrasos introduzidos pelas Unidades Habilita Evento sobre o *overhead* [4]. Para matrizes de entrada de dimensões 80 x 80, o *overhead* vale 67.39%. E para matrizes 160 x 160, 36.45%.

A última parcela da Equação IV.5, nm, representa o número de mensagens transmitidas pelo Protocolo Distribuído de Capacidade Um, com o propósito de prevenir a ocorrência de *deadlocks*. Relembrando a discussão da Seção III.7, o projeto do Protocolo é baseado na troca de sinais de sincronização, chamados *READY*, entre os processas Conversor de Protocolo, associados a Interface de Comunicação. Cada mensagem entre processadores, proveniente de algum dos processos da aplicação, é acompanhada por um *READY*. Com isso, durante a reexecução, o número total de comunicações entres processadores totaliza $2nm$ mensagens, onde nm pertecem

[4] O custo de deslocamento e roteamento das mensagens em 1 processador é zero

Proc	Eventos	Tempo de Execução (s)				Overhead (%)		
		EXEC	ORP	ORM	REXEC	ORP	ORM	REXEC
1	38416	6.90	7.32	10.37	11.55	6.08	50.28	67.39
2	12808	4.83	5.08	5.30	8.17	5.17	9.73	69.15
4	38416	25.39	26.99	29.96	69.43	6.30	17.99	173.45
8	89632	60.56	64.00	80.26	184.25	5.68	32.52	204.24

Tabela IV.1: Comportamento do Algoritmo de Multiplicação por Elemento em Relação ao Número de Processadores - Matrizes de Entrada 80 x 80

Proc	Eventos	Tempo de Execução (s)				Overhead (%)		
		EXEC	ORP	ORM	REXEC	ORP	ORM	REXEC
1	153616	49.43	51.09	63.63	67.45	3.35	28.72	36.45
2	51208	30.98	32.01	32.87	43.63	3.32	6.10	41.28
4	153616	113.32	120.55	130.33	273.50	6.38	15.11	141.35
8	358432	257.88	274.03	337.06	757.53	6.28	30.70	193.75

Tabela IV.2: Comportamento do Algoritmo de Multiplicação por Elemento em Relação ao Número de Processadores - Matrizes de Entrada 160 x 160

a aplicação, e as nm restantes ao Protocolo. A parcela Q_3, portanto, é sensível as variações na quantidade de processadores presentes no sistema e contribui, grandemente, para o aumento do custo da comunicação.

O *overhead* introduzido durante a reexecução é consideravelmente alto, e resulta da soma dos efeitos induzidos pelo Protocolo Distribuído de Capacidade Um e pelas Unidades Habilita Evento. O custo adicional de comunicação imposto pelo mecanismo provoca um impacto significativo sobre o desempenho da aplicação, que pode chegar, em alguns casos, a cerca de 200%. Para matrizes de entrada de dimensões 80 x 80 (Tabela V.1), o *overhead* para 2, 4 e 8 processadores vale 69.15%, 173.45% e 204.24%, respectivamente, revelando uma tendência para o crescimento acelerado do *overhead* em função do número de processadores. A Tabela V.2 apresenta outras medidas, que confirmam esta tendência.

Aplicações Fracamente Acopladas

As aplicações fracamente acopladas apresentam um custo de comunicação relativa-

mente pequeno, se comparado a carga computacional, pois o número de mensagens permutadas entre os processos é bastante reduzido. Quando estas aplicações são executadas em sistemas com múltiplos processadores, o custo da comunicação representa uma parcela menor do tempo total de processamento. Suponha, então, que uma aplicação fracamente acoplada é mapeada em uma máquina composta por n processadores, onde residem n processos, que trocam m mensagens cada, com $1 \leq m \leq 512$.

Considere que esta aplicação é submetida ao mecanismo de reexecução e que o *buffer* para o armazenamento dos rótulos dos eventos comporta até 512 *bytes*. Os m eventos associados a cada um dos processos da aplicação são, portanto, insuficientes para preencher o *buffer*. Assim sendo, a expressão $q = \frac{m}{512}$, que fornece o número de mensagens necessárias para transportar m rótulos, assume o valor 1. É importante destacar que a discussão apresentada a seguir utiliza equações e argumentos desenvolvidos para as aplicações fortemente acopladas, que também são válidos para as aplicações fracamente acopladas.

Para o procedimento de monitoração na implementação orientada ao procedimento, o número de comunicações entre processos do mecanismo é dado por $q = \frac{m}{512} * n = n$. Aplicando o valor de q a Equação IV.3, vem

$$p = \frac{n}{mn+n} = \frac{1}{m+1} \Rightarrow 50 \leq p \leq 0.19\,(\%) \qquad \text{(IV.6)}$$

onde p representa a contriuição percentual da parcela q para o número total de mensagens, desconsiderando as comunicações pertencentes a Interface de Comunicação.

Os atrasos introduzidos pelas Unidade Registra Evento, mais uma vez, são irrelevantes, em face ao tempo total de processamento. Da mesma forma, a interferência provocada pelas q mensagens associadas ao mecanismo é relativamente pequena, pois o número de eventos observados durante a monitoração é reduzido. O percentual p, no entando, demonstra que a parcela q pode atingir grandezas comparáveis aquelas inerentes a própria aplicação. Assim sendo, ainda que o *overhead* assuma

Proc	Eventos	Tempo de Execução (s)				Overhead (%)		
		EXEC	ORP	ORM	REXEC	ORP	ORM	REXEC
1	40	5.71	5.71	5.71	5.81	0.00	0.00	1.75
2	12	3.21	3.21	3.21	3.34	0.00	0.00	4.00
4	40	4.19	4.20	4.28	4.41	0.23	2.14	5.25
8	144	4.23	4.35	4.43	5.83	2.83	4.72	37.82

Tabela IV.3: Comportamento do Algoritmo de Multiplicação por Bloco em Relação ao Número de Processadores - Matrizes de Entrada 80 x 80

valores bastante restritos, o mecanismo apresenta uma sensibilidade acentuada as variações no custo da comunicação e, conseqüentemente, ao número de processadores. Esta característica é confirmada pelos resultados exibidos na Tabela V.3. Para matrizes de entrada de dimensões 80 x 80, o *overhead* para 2, 4 e 8 processadores vale 0.00%, 0.23% e 2.83%, respectivamente, relevando uma tendência para o crescimento acelerado em função do número de processadores.

As consideraç8ea desenvolvidas para a implementação orientada ao procedimento são, também, válidas para a implementação orientada á mensagem e para a reexecução. O volume relativamente pequeno dos eventos tratados contribui para a limitação do valor do *overhead.* As Equações IV.4 e IV.5 comprovam que, para $q = n$, o número de mensagens associadas aos processos do mecanismo se aproxima daquele associado a aplicação, o que torna os procedimentos de monitoração e reexecução sensíveis a topologia do sistema distribuiído e ao custo da comunicação entre processadores. As Unidades Registra Evento e Habilita Evento, agora, induzem atrasos significativos, e, portanto, o *overhead* atinge valores superiores aqueles observados para a implementação orientada ao procedimento. A Tabela V.3 exemplifica as caracteríticas descritas. Para teste em hipercubos com 1, 2, 4 e 8 processadores, o *overhead* oscila entre 0.00% e 4.75%, para a implementação oriendata a mensagem, e 1.75% e 37.82%, para a reexecução.

IV.3.2 Overhead x Carga Computacional

A superposição do mecanismo de reexecução determina um acréscimo na carga computacional associada a cada processador do sistema, que oscila de acordo com o grau de atuação do mecanismo. Como a variação no volume de processamento da aplicação afeta o comportamento do *overhead* é dependente, principalmente, da natureza da computação processada.

As características dos algoritmos aqui estudados são plenamente conhecidas, o que possibilita relacionar as *quantidades* de processamento da aplicação Q_{apl} e do mecanismo Q_{mec} no sentido de revelar o perfil do comportamento do *overhead* em função da carga computacional. Observe, portanto, a formulação apresentada a seguir.

Seja P o percetual referente ao processamento do mecanismo, expresso pela equação

$$P = \frac{Q_{mec}}{Q_{apl} + Q_{mec}} \qquad \text{(IV.7)}$$

A parcela Q_{apl} é dada pela expressão

$$Q_{apl} = C\, n^2$$

onde

- C : constante de proporcionalidade
- n : dimensão das matriz quadradas de entrada A e B

A parcela Q_{mec} é representada por

$$Q_{mec} = C\, nev$$

onde

- C : constante de proporcionalidade

- *nev* : número de eventos tratados pelo mecanismo

Reescrevendo a Equação IV.7 usando os valores de Q_{apl} e Q_{mec} vem

$$P = C \frac{nev}{n^2 + nev} \quad \text{(IV.8)}$$

Para o algoritmo de Multiplicação por Bloco, o valor da variável *nev* é fornecido pela Equação IV.1

$$nev = 2k^2 + 2k \rightarrow constante$$

O valor de k é igual ao número de sub-matrizes em que as matrizes de entrada A e B são divididas. Aqui, k coincide com o número de processadores utilizados, considerado constante. Aplicando o valor de *nev* na Equação IV.8 vem

$$P = C \frac{2(k^2 + k)}{n^2 + 2(k^2 + k)}$$

resultando em

$$P = C' \frac{1}{n^2 + C''}$$

A expressão obtida demonstra que quando a dimensão das matrizes de entrada A e B cresce a parcela P, referente a ação do mecanismo, diminui, tendendo a tangenciar o *zero* a medida que n se aproxima de *infinito*. A velocidade de decaimento de P depende dos valores das constantes C' e C'' que possuem valores diferentes para o algoritmo Monitor, nas implementações orientada ao procedimento e a mensagem, e o algoritmo Repetidor. Este comportamento é confirmado pelos resultados experimentais exibidos nos gráficos apresentados nas Figuras IV.2 e IV.3.

Para o algoritmo de Multiplicação por Elemento o valor do termo *nev* é obtido a partir da Equação IV.2

$$nev = 2n^2(k-1) + 4k, \; com \; k \; cte$$

Onde, mais uma vez, k representa o número de sub-matrizes em que são particionadas as matrizes de entrada A e B e, assim sendo, assumido constante. Reescrevendo a Equação IV.8 utilizando o valor de *nev* vem

$$P = C \frac{2n^2(k-1) + 4k}{n^2 + 2n^2(k-1) + 4k}$$

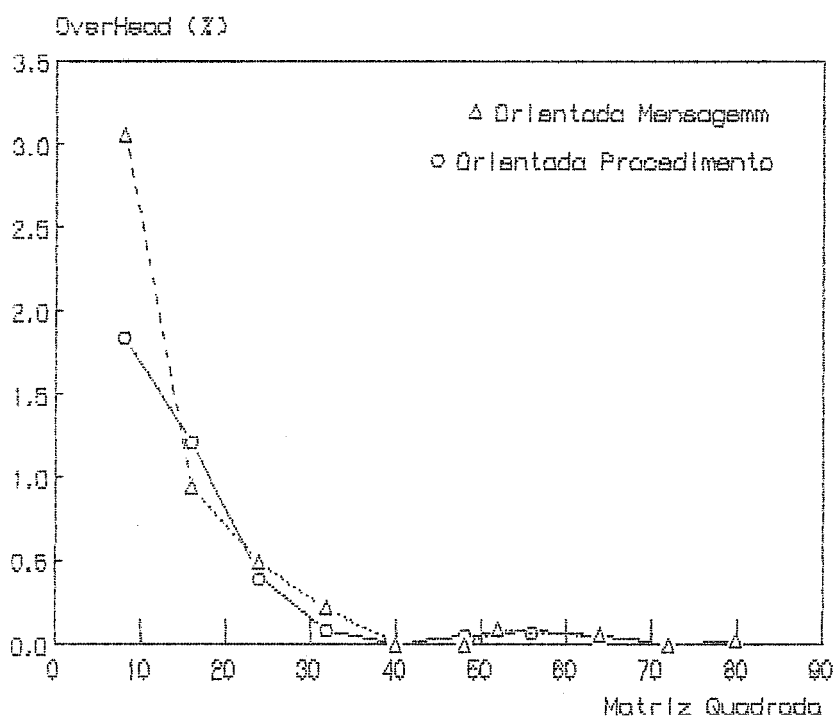


Figura IV.2: Comportamento do Algoritmo de Multiplicação por Bloco em Relação a Carga Computacional - Medidas realizadas em 2 processadores utilizando o algo. ritmo Monitor nas implementações orientada ao procedimento e orientada a mensagem

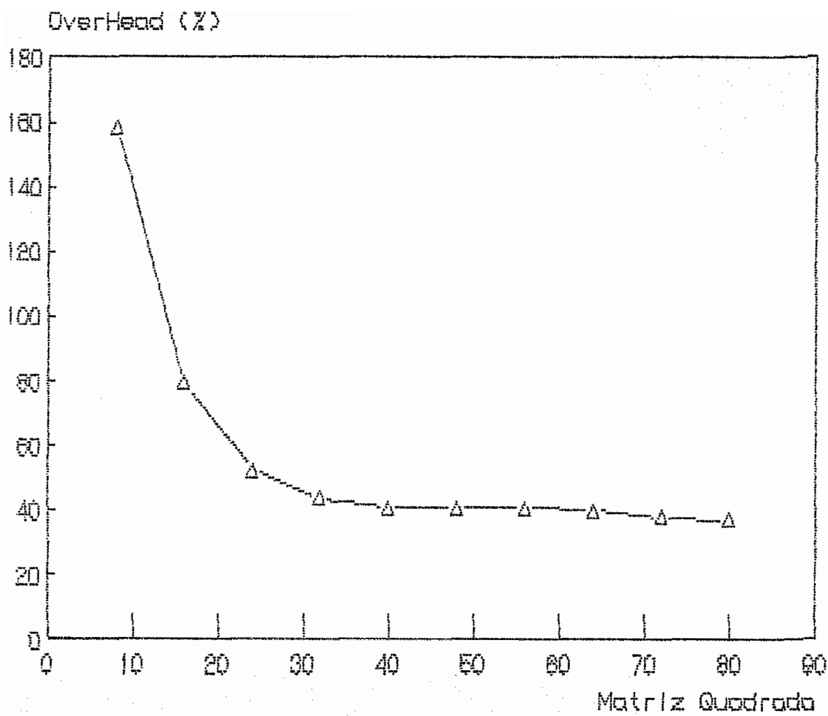


Figura IV.3: Comportamento do Algoritmo de Multiplicação por Bloco em Relação a Carga Computacional - Medidas realizadas em 8 processadores utilizando o algoritmo Repetidor

resultando em

$$P = C\,\frac{n^2 + C'}{C''n^2 + C'}$$

A expressão matemática obtida permite extrapolar **as** seguintes **observações**

1. Para $n = 0 \Rightarrow P = C$

2. **Para** $n \rightarrow \infty \Rightarrow P \rightarrow \frac{C}{C''}$

Concluindo, inicialmente, P atinge seu ponto de máximo, decaindo a medida em que a dimensão das matrizes de entrada A e B cresce, até atingir um valor praticamente constante. Os **valores** das constantes C, C' e C" variam de acordo com as características **dos** algoritmos **Monitor** e Repetidor, determinado a velocidade do decaimento e o ponto de máximo. As curvas apresentadas nas Figuras IV.4 e IV.5 confirmam as **considerações** desenvolvidas.

IV.3.3 Medidas de Tempo com I/O

A medidas tempo apresentadas anteriormente desconsideram o tempo gasto pelo mecanismo de reexecução em operações de entrada e salda (I/O), por ser este **parâmetro** estritamente dependente da tecnologia da máquina hospedeira utilizada. Como ilustração, foram levantados alguns valores que incluem o tempo total de execução (com I/O) para testes realizados em 1 processador conectado a um hospedeiro do tipo IBM/PC-XT. Os resultados apresentados nas Tabelas V.4 e V.5 comparam valores de *overhead*, que somam ou não a parcela referente as atividades de entrada e salda. Estas atividades envolvem, exclusivamente, operações de leitura e escrita em uma unidade de disco rígido.

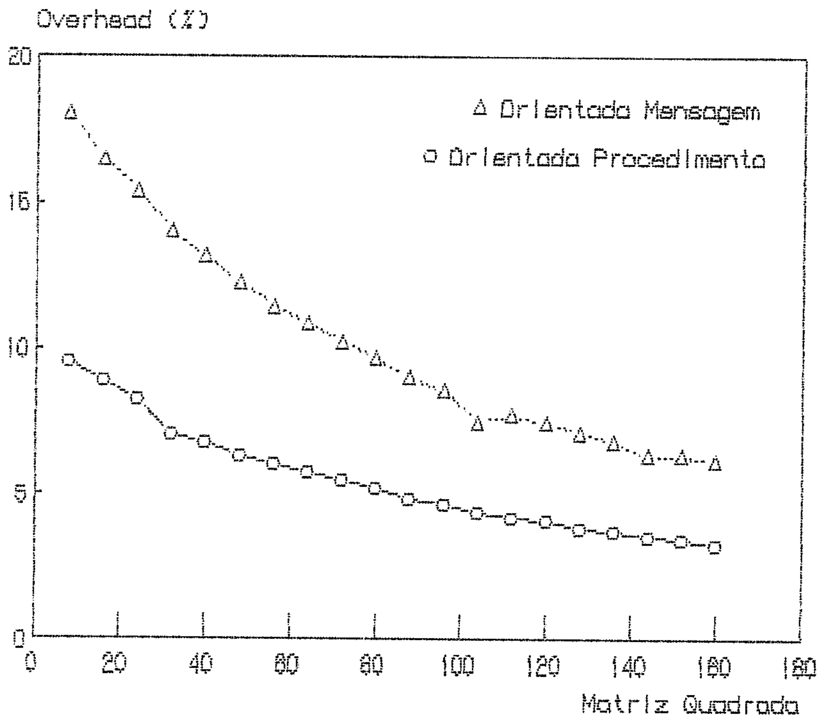


Figura IV.4: Comportamento do Algoritmo de Multiplicação por Elemento em Relação a Carga Computacional - Medidas realizadas em 2 processadores utilizando o algoritmo Monitor nas implementações orientada ao procedimento e orientada a mensagem

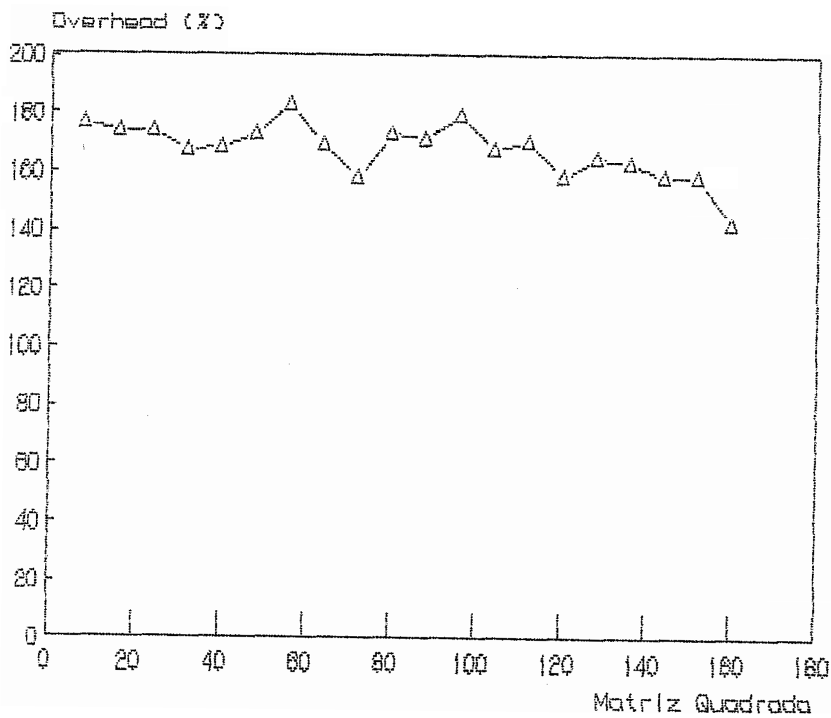


Figura IV.5: Comportamento do Algoritmo de Multiplicação por Elemento em Relação a Carga Computacional - Medidas realizadas em 2 processadores utilizando o algoritmo Repetidor

Dim	Overhead (%)					
	Sem I/O			Com I/O		
	ORP	ORM	REXEC	ORP	ORM	REXEC
8 x 8	5.33	34.00	63.33	9.33	48.00	60.66
16 x 16	0.00	5.01	11.61	0.00	7.50	11.50
24 x 24	0.00	0.95	4.61	0.00	1.80	4.46
32 x 32	0.00	0.00	2.80	0.00	0.22	2.76
40 x 40	0.00	0.00	2.18	0.00	0.00	2.13
48 x 48	0.00	0.00	1.88	0.00	0.00	1.86
56 x 56	0.00	0.00	1.75	0.00	0.00	1.74
64 x 64	0.00	0.00	1.65	0.00	0.00	1.67
72 x 72	0.00	0.00	1.68	0.00	0.00	1.64
80 x 80	0.00	0.00	1.63	0.00	0.00	1.63

Tabela IV.4: Comportamento do Algoritmo de Multiplicação por Bloco em Relação as Operações de I/O - Medidas realizadas em 1 processador

Dim	Overhead (%)					
	Sem I/O			Com I/O		
	ORP	ORM	REXEC	ORP	ORM	REXEC
8 x 8	19.81	168.73	245.51	24.14	176.47	267.70
16 x 16	14.72	133.31	188.27	16.65	135.43	180.78
24 x 24	13.56	110.71	154.01	13.92	111.28	147.84
32 x 32	11.58	94.13	130.32	28.95	95.31	280.36
40 x 40	9.88	82.06	113.01	16.74	82.98	280.10
48 x 48	8.76	72.77	99.64	22.19	73.94	252.48
56 x 56	7.89	65.37	89.01	19.39	66.08	230.16
64 x 64	7.14	59.39	80.39	17.07	61.30	244.58
72 x 72	6.51	54.44	73.27	15.97	56.05	214.15
80 x 80	6.01	50.24	67.28	16.81	52.39	216.56

Tabela IV.5: Comportamento do Algoritmo de Multiplicação por Elemento em Relação as Operações de I/O - Medidas realizadas em 1 processador

Capítulo V

Conclusão

O objetivo deste trabalho de tese foi o projeto e a avaliação de um mecanismo de reexecução determinística voltado para programas paralelos usando memória distribuída. O mecanismo proposto deveria provocar uma interferência reduzida, sobretudo na etapa de monitoração, quando a presença do monitor pode induzir a adulteração do comportamento das computações.

A implementação do mecanismo e das aplicações foi totalmente desenvolvidas utilizando a linguagem *occam*; ainda que, as idéias e algoritmos propostos sejam abrangentes a outras linguagens de programação. Os testes foram conduzidos em um hipercubo formado por processadores do tipo *transputer* em configurações com 1, 2, 4 e 8 nós.

As medidas de tempo levantadas exercitaram o comportamento do mecanismo em relação as flutações no custo da comunicação entre processadores e na carga computacional das aplicações.

Os testes permitiram o refinamento dos algoritmos e demonstraram que o procedimento de monitoração introduz uma perturbação aceitável. Para a implementação orientada ao procedimento, o *overhead*, determinado pela presença do mecanismo, oscilou entre 0.00% e 6.30%. Para a implementação orientada à mensagem, o

overhead ficou limitado entre 0.00% e 30.70%.

O procedimento de reexecução revelou percentuais de *overhead* consideralvelmente altos, provocados pela atividade do algoritmo de pervenção de *deadlocks*. As medidas mostraram variações entre 1.75% e 204.24%, apresentando uma tendência para o crescimento acelerado em função do número de processadores mapeados.

A viabilidade de reproduzir deterministicamente um certo comportamente motiva o desenvolvimento de diversas ferramentas para a depuração e análise de desempenho dos programas paralelos. Seguem, então, algumas sugestões para futuras extensões deste trabalho.

- As condições de erro associadas a estratégia de sincronização dos processos exigem do programador uma análise cansativa e demorada do andamento do programa. Uma ferramenta, executando juntamente com o procedimento de reexecução, poderia fornecer, dinamicamente, diagramas que exibiriam as relações de sincronismo existentes entre os processos.

- O refinamento da *performance* dos algoritmos e o balanceamento da carga entre os processadores podem ser implementados a partir de um mapa, revelevando as dependências temporais existentes entre os processos. Uma ferramente, semelhante ao procedimento de monitoração, poderia registrar medidas de tempo, referentes as operações de troca de mensagens, que seriam usadas na construção do citado mapa.

Referências Bibliográficas

[1] B. Awerbuch. Complexity of Network Synchronization. *Journal of the Association for Computing Machinery*, 32(4), Outubro 1985.

[2] E. W. Disjkstra. Guarded Command, Non-Determinancy and Formal Derivation of Programs. *Communications of the ACM*, 18(8), Agosto 1975.

[3] L. M. A. Drummond. *Projeto e Implementação de um Processador Virtual de Comunicação.* Master's thesis, COPPE - Programa de Engenharia de Sistemas e Computação, Agosto 1990.

[4] J. L. Peterson e A. Silberschatz. *Operating System Concepts.* Addison-Wesley Publishing Company, 1985.

[5] E. S. T. Fernandes e C. L. Amorim, V. C. Barbosa. Uma Introdução à Computação Paralela e Distribuída. In *VI Escola de Computação*, 1988.

[6] W. C. Athas e C. L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, Agosto 1988.

[7] D. Pountain e D. May. *A Tutorial Introduction to Occam Programming.* BSP Professional Books, 1988.

[8] T. J. LeBlanc e J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4), Abril 1987.

[9] K. M. Chandy e J. Misra. *Parallel Programs Design - A Foundation.* Addison-Wesley Publishing Company, 1988.

[10] K. M. Chandy e L. Lamport. Distributed Snapshots: Determining Globa States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1) Fevereiro 1985.

[11] G. C. Fox *et all. Solving Problems on Concurrent Processors - General Te niques and Regular Problems.* Volume Volume 1, Prentice Hal

[12] J. Gait. A Debugger for Concurrent Programas. *Software Practice and E rience*, 15(6), Junho 1985.

[13] J. Gait. A Probe Effect in Concurrent Programs. *Software Practice and Experience*, 16(3), Março 1986.

[14] F. Germano e W. H. Kohler H. Garcia-Molina. Debugging a Distributed Computing System. *IEEE Transactions on Software Engineering*, SE-10(2), Março 1984.

[15] K. Slind e B. Unger J. Joyce, G. Lomow. Monitoring Distributed *ACM Transactions on Computer Systems*, 5(2), Maio 198'

[16] L. Lamport. Times, Clocks and the Ordering of Events in a Distributed System *Communications of the ACM*, 21(7), Julho 19

[17] INMOS Limited. *IMS T800 Transputer.* INMOS Limited, 19

[18] INMOS Limited. *Occam 2 Reference Manual.* Prentice Hall, 1988

[19] INMOS Limited. *Transputer Development System.* Prentice Hall, 1988

[20] INMOS Limited. *Transputer Technical Notes.* Prentice Hall, 198

[21] A. E. Oldehoeft e R. R Oldehoeft M. Maekawa. *Operating Systems - Concepts.* The Benjamin/Cummings Plublishing Company, 1

[22] J. M. Mellor-Crummey. *Debuggging and Analysis of Large-Scale Parallel Programs.* Technical Report 312, University of Rochester, Department of Compu Science, Setembro 1989

[23] C. L. Amorim e E. M. Chaves Filho R. Citro, A. F. Souza. *Debuggging and Analysis of Large-Scale Parallel Programs.* Technical Report ES-241, Universidade Federal do Rio de Janeiro - COPPE Programa de Engenharia de Sistemas e Computação, 1991.

[24] H. S. Stone. *High-Performance Computer Architecture.* Addison-Wesley Publishing Company, 1987.

Apêndice A

Interface de Comunicação

Este apêndice apresenta o código fonte dos processos que formam a Interface de Comunicação, discutida no Capítulo II. A implementação dos processos foi desenvolvida na liguangem de programação *occam*, porém, aqui, os programas são exibidos em pseudo-código.

Algoritmo do Processo Multiplex

```
REPEAT
    EsperaMsg (from.CONV [i], msg) ∀ i (0 ≤ i < n) →
        EnviaMsg (to.PVC, msg);
    OR
    EsperaMsg (to.MUX, stop) →
        fim := TRUE;
UNTIL FALSE;
```

Algoritmo do Processo DeMultiplex

```
REPEAT
    EsperaMsg (from.PVC, msg) →
```

```
        BEGIN
            i := DestinoMsg (msg);
            EnviaMsg (to.CONV [i], msg);
        END;
    OR
    EsperaMsg (to.DUX, stop) →
        fim := TRUE;
UNTIL FALSE;
```

Algoritmo do Processo Conversor de Protocolo do PVC

```
REPEAT
    EsperaMsg (from.DUX, msg) →
        BEGIN
            msg' := ConverteProtocolo (msg);
            EnviaMsg (to.processo, msg');
        END;
    OR
    EsperaMsg (to.CONV, stop) →
        fim := TRUE;
UNTIL FALSE;
```

Algoritmo do Processo Conversor de Protocolo do Processo

```
REPEAT
    EsperaMsg (from.processo, msg) →
        BEGIN
            msg' := ConverteProtocolo (msg);
            EnviaMsg (to.MUX, msg');
        END;
```

```
    OR
    EsperaMsg (to.CONV, stop) →
        fim := TRUE;
UNTIL FALSE;
```

Apêndice B

Algoritmo de Multiplicação

Este apêndice apresenta o código dos processos especificados pelo Algoritmo de Multiplicação, descrito no Capítulo V. Ainda que a implementação dos processos tenha sido desenvolvida na linguagem de programação occam, *aqui, por simplicidade, os programas aparecem escritos em pseudo-código. O algoritmo de Multiplicação considera duas versões,* bloco e elemento, *e, assim sendo, os processos são identificados por um cabeçalho formado pelo nome do processo e/ou versão. A ausência da versão representa que o mesmo procedimento foi usado em ambas as implementações, ou seja,* bloco e elemento.

Algoritmo do Processo Centralizador

```
BEGIN
    Leitura das matrizes A e B;
    Particioana A e B em n sub-matrizes;
    FOR i = 1 TO n DO
        BEGIN
            EnviaMsg (to.Multiplica, A'i);
            EnviaMsg (to.Multiplica, B'i);
        END;
    END;
```

```
        WHILE i ≠ n DO
            RecebeMsg (from.Multiplica [i], C'i) ∀i(0 ≤ i < n) →
                i := i + 1;
        END;
    END;
```

Algoritmo do Processo Buffer - Versão Bloco

```
BEGIN
    RecebeMsg (to.Buffer [0], B');
    WHILE k ≠ n - 1 DO
        RecebeMsg (to.Buffer [i], REQUEST) ∀i(0 ≤ i < n) →
            BEGIN
                EnviaMsg (from.Buffer [i], B');
                k := k + 1;
            END;
    END;
END;
```

Algoritmo do Processo Buffer - Versão Elemento

```
BEGIN
    RecebeMsg (to.Buffer[0], B');
    WHILE k ≠ tam.sub.matriz * (n - 1) DO
        RecebeMsg (to.Buffer [i], {p,q}) ∀i(0 ≤ i < n) →
            BEGIN
                EnviaMsg (from.Buffer [i], B' [p] [q]);
                k := k + 1;
            END;
    END;
END;
```

Algoritmo do Processo Multiplica - Versão Bloco

```
BEGIN
    RecebeMsg (to.Multiplica, A');
    RecebeMsg (to.Multiplica, B');
    Multiplica (A', B', C');
    FOR i = 1 TO n - 1 DO
        BEGIN
            EnviaMsg (to.Buffer [i], REQUEST);
            RecebeMsg (from.Buffer, B');
            Multiplica (A', B', C');
        END;
    END;
    EnviaMsg (from.Multiplica, C');
END;
```

Algoritmo do Processo Multiplica - Versão Elemento

```
BEGIN
    RecebeMsg (to.Multiplica, A');
    RecebeMsg (to.Multiplica, B');
    Multiplica (A', B', C');
    FOR i = 1 TO n - 1 DO
        BEGIN
            FOR p = 1 TO tam.sub.matriz DO
                FOR q = 1 TO tam.sub.matriz DO
                    BEGIN
                        EnviaMsg (to.Buffer [i], {p,q});
                        RecebeMsg (from.Buffer, B' [p] [q]);
                    END;
                END;
```

```
            END;
            Multiplica (A', B', C');
        END;
    END;
    EnviaMsg (from.Multiplica, C');
END;
```