UM CONJUNTO DE FERRAMENTAS PARA IMPLEMENTAÇÃO DE PROCESSOS COOPERATIVOS

Mariana Yumrell Miranda

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 A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Valmir Carneiro Barbosa, Ph.D.
(Presidente)

Prof. Cláudio Luís de Amorim, Ph.D.

Prof. Alberto Franço de Sá Santoro, Ph.D.

Rio de Janeiro, RJ - Brasil

ABRIL DE 1991

MIRANDA, MARIANO SUMRELL

Um Conjunto de Ferramentas Para implementação de Processos Cooperativos [Rio de Janeiro] 1991

X, 111 p. 29,7 cm (COPPE/UFRJ, Ms.C., Engenharia de Sistemas e Computação, 1991

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Processamento Paralelo I. COPPE/UFRJ II. Título (série)

À Elza

AGRADECIMENTOS

Agradeço à todas as pessoas que participaram do desenvolvimento do CPS e tornaram possível esse trabalho.

Agradeço a Valmir Carneiro Barbosa pela orientação e atenção.

Agradeço a Alberto Santoro pela oportunidade de realizar esse trabalho, pelo seu apoio e pela sua orientação.

Agradeço a todo o pessoal do LAFEX/CBPF pelo apoio.

Agradeço às pessoas do Computer Division R&D Department do Fermilab, pela sua cooperação. Em especial gostaria de agradecer a Joe Biel e Tom Nash pela oportunidade oferecida e a confiança em mim depositada.

Agradeço a Bruno Schulze, Raquel Schulze, Eliane Rodrigues de Ávila e Carla Osthoff de Barros pelo grande incentivo bem como pelas sugestões e pela revisão na expressão escrita.

Agradeço à Elza de Mattos Paiva pelo trabalho de revisão e pela compreensão dos meus momentos de ausência.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (Ms.C.).

UM CONJUNTO DE FERRAMENTAS PARA IMPLEMENTAÇÃO DE PROCESSOS COOPERATIVOS

Mariano Pumrell Miranda

Abril de 1991

Orientador: Prof. Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

Neste trabalho é apresentado um conjunto de ferramentas, chamado CPS, que permite o desenvolvimento de aplicações que explorem o paralelismo que pode ser obtido com o uso de diversos processadores na resolução de um problema computacional.

É feita uma breve discussão de conceitos e idéias encontradas na literatura e que serviram de base às escolhas efetuadas na definição do CPS.

É também apresentada a arquitetura do sistema para o qual o CPS foi originariamente concebido e por fim é discutido com detalhes os mecanismos de comunicação, sincronismo, transferência de dados e outros recursos do CPS.

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

A SET OF TOOLS FOR THE IMPLEMENTATION OF COOPERATIVE PROCESSES

Mariana Sumrell Miranda

April, 1991

Thesis Supervisor: Prof. Valmir Carneiro Barbosa

Department: Engenharia de Sistemas e Computação

The present work presents a set of tools, called CPS, that allows the development of aplications that take advantage of the parallelism that can be achieved with the use of multiple processors in the resolution of a computational task.

It shows a short discussion of the concepts and ideas found in the literature and were the basis of the choices made in CPS's specifications.

It is also presented the architecture of the system for which CPS was originally conceived. Finally it is presented in detail the mechanisms for processes synchronization, communication and other features of CPS.

ÍNDICE

Pág.

CAPÍTULO I - INTRODUÇÃO..... 1

CAPÍTULO II - PROCESSAMENTO PARALELO 4

II.1 - Por que Processamento Paralelo ?.... 5

II.2 - Definições 8

II.3 - Sistemas Fortemente Acoplados e Sistemas Fracamente Acoplados 10

II.3.1 - Sistemas Fortemente Acoplados 11

II.3.1.1 - Uso de Memória Local e Memória "Cache" 12

II.3.1.2 - Uso de Múltiplos Barramentos e Memórias 13

II.3.2 - Sistemas Fracamente Acoplados 16

II.3.3 - Comparação entre Sistemas Fortemente Acoplados e Sistemas Fracamente Acoplados 20

II.4 - Variáveis Compartilhadas e Troca de Mensagens 21

II.5 - Criação Estática de Processos X Criação Dinâmica de Processos 23

II.6 - Exclusão Mútua no Acesso a Variáveis Compartilhadas 24

II.6.1 - Semáforo 26

II.6.2 - Monitor 27

II.7 - Primitivas de Comunicação e Sincro-

nismo em Sistemas Baseados em Troca de Mensagens 27
II.7.1 - Endereçamento 28
II.7.2 - Primitivas Síncronas e Assíncronas 29
II.7.3 - Chamada Remota de Procedimentos .. 30
II.8 - Uso de Primitivas de Processamento Paralelo em Linguagens de Programação e Bibliotecas de Rotinas 31
II.8.1 - Linguagem de Programação 31
II.8.2 - Biblioteca de Rotinas 32

CAPÍTULO III - DEFINIÇÕES DAS ESTRATÉGIAS ADOTADAS NO PROJETO 34

CAPÍTULO IV - ARQUITETURA DO SISTEMA 37
IV.1 - Introdução 37
IV.2 - ACP/R3000 37
IV.3 - O Barramento VME 39
IV.4 - Branch Bus 40
IV.5 - VBBC e BVI 40
IV.6 - Software Básico do Sistema ACP 41

CAPÍTULO V - DESCRIÇÃO DO CPS 43
V.1 - Metodologia para Desenvolvimento de Aplicações com o CPS 45
V.2 - Primitivas de Troca de Mensagens ... 47
V.3 - Sincronismo de Processos 50
V.3.1 - Filas 50

V.3.2 - Chamadas Remotas de Procedimentos 53

V.3.3 - Pontos de Sincronismo 54

V.4 - Transferência de Dados 55

V.5 - Conversão de Dados 56

V.6 - O "Job Manager" 57

V.6.1 - Inicialização e Finalização de Processos 58

V.6.2 - Registro de Atividades 59

V.6.3 - Serviços de Sincronismo 60

V.6.4 - Monitoração de Processos 62

V.6.5 - Redirecionamento de Entrada e Saída para Terminal 62

V.7 - Depuração de Aplicações 63

V.8 - Interface com o Sistema Operacional 64

CAPÍTULO VI - DESCRIÇÃO DA BIBLIOTECA DE ROTINAS . 67

VI.1 - Rotinas de Inicialização e Finalização 68

VI.2 - Chamada Remota de Procedimentos 68

VI.3 - Transferência de Dados 71

VI.4 - Filas 72

VI.5 - Sincronismo 74

VI.6 - Troca Explícita de Mensagens 75

VI.7 - Conversão de Dados 76

VI.8 - Tratamento de Erros 77

VI.9 - Rotinas Auxiliares 80

CAPÍTULO VII - O PROGRAMA MONITOR 83

CAPÍTULO VIII - CONCLUSÃO 86

REFERÊNCIAS BIBLIOGRÁFICAS 90

APÊNDICE A 96

APÊNDICE B 100

CAPÍTULO I

INTRODUÇÃO

A crescente demanda por maior capacidade de processamento em diferentes áreas de aplicação e em quase todas as áreas do conhecimento humano tem exigido computadores cada vez mais poderosos. Essas máquinas, em geral, são conseguidas através de processadores cada vez mais potentes. No entanto, o aumento da capacidade desses processadores apresenta um limite imposto pelas condições tecnológicas e pelos custos muito altos.

Uma solução que tem se mostrado eficaz para o atendimento dessa necessidade de maior capacidade computacional é a utilização de processamento paralelo.

Uma definição bem abrangente de processamento paralelo pode ser encontrada em [1]: "Processamento paralelo é uma forma eficiente de processamento de informação que enfatiza a exploração de eventos concorrentes no processo computacional." Para esse trabalho, no entanto, interessa apenas o processamento paralelo baseado na utilização concorrente de mais de um processador para a execução de uma determinada tarefa.

Para que as aplicações aproveitem o paralelismo que pode ser obtido com o uso de diversos processadores, são necessárias ferramentas adequadas de software que permitam o desenvolvimento e execução de programas paralelizados.

Neste trabalho será apresentado um conjunto de tais ferramentas, chamado de "Cooperative Processes Software"

que, daqui por diante, será referenciado como CPS. Essas ferramentas foram desenvolvidas num projeto de colaboração do Laboratório de Física Experimental de Altas Energias (LAFEX) do Centro Brasileiro de Pesquisas Físicas com o Computer Research and Development Department, antigo Advanced Computer Program, do Fermi National Accelerator Laboratory. O CPS foi desenvolvido juntamente com o hardware da segunda geração do computador paralelo de alto desempenho, chamado ACP.

O CPS foi originalmente concebido para ser executado no hardware da segunda geração do ACP. No entanto é um software bastante portátil e atualmente está implementado não somente no ACP, mas também em várias máquinas executando UNIX (Sun, Apollo, Silicon Graphics, VAX sob o ULTRIX) e VAX executando VMS.

A importância do CPS pode ser compreendida se considerarmos a atual carência de software para aproveitamento do paralelismo que o hardware já é capaz de nos oferecer [2, 3]. A sua portabilidade, que permite a sua utilização em diferentes sistemas, permite-nos vê-lo como uma ferramenta que pode preencher a lacuna de software no campo de processamento paralelo.

O capítulo II deste trabalho trata da questão de processamento paralelo levando em consideração as diferentes possibilidades existentes para a obtenção do desejado paralelismo. Esse capítulo serve de subsídio para o capítulo III, no qual são apresentadas as estratégias escolhidas nesse projeto e as suas justificativas.

No capítulo IV é feita uma breve descrição da arquitetura do ACP de segunda geração e de sua interconexão com outros sistemas que também podem executar o CPS.

No capítulo V é apresentado o CPS. Esse capítulo é a própria essência desse trabalho.

No capítulo VI são descritas as rotinas desenvolvidas para a implementação de processos cooperativos.

No capítulo VII é descrito o programa monitor que é uma ferramenta do CPS que permite o acompanhamento da execução dos programas. Essa ferramenta é especialmente útil para a depuração dos programas paralelos.

No capítulo VIII são apresentadas as conclusões finais.

No apêndice A é feita uma descrição do arquivo de comandos usado para a execução do CPS e no apêndice B são listados alguns programas que exemplificam a utilização do CPS.

CAPÍTULO II
PROCESSAMENTO PARALELO

O emprego de paralelismo em sistemas computacionais não é uma prática recente. Há muito tempo os sistemas com apenas um processador têm se beneficiado de técnicas que permitem a execução paralela de certas funções. Dentre estas técnicas podemos citar:

1) Sobreposição de processamento com operações de entrada e saída;
2) "Pipelining";
3) Uso de múltiplas unidades funcionais;
4) Processamento Vetorial.

Neste capítulo será discutido o paralelismo visto como o emprego de mais de um processador para executar um determinado trabalho. Obviamente, cada processador que forma o sistema paralelo em questão pode se beneficiar das técnicas citadas acima, cuja análise no entanto foge ao escopo deste trabalho e pode ser encontrada em [1] e [4].

Devido à riqueza do assunto e à enormidade de questões que podem ser discutidas quando se trata de processamento paralelo, não se pretende esgotar aqui o estudo desse assunto. Serão abordados os aspectos e conceitos mais relevantes e principalmente aqueles que, durante o processo de definição do sistema proposto nesse trabalho, foram mais importantes na determinação de um caminho a ser seguido.

Inicialmente será discutido o porquê do uso de

processamento paralelo. Depois serão apresentadas algumas definições de termos usados no estudo de processamento paralelo. A seguir serão feitas comparações entre sistemas fortemente acoplados e fracamente acoplados; entre o uso de variáveis compartilhadas e troca de mensagens e; entre criação estática e criação dinâmica de processos.

Em seguida será feita uma análise da questão de exclusão mútua em sistemas com memória compartilhada e das primitivas de comunicação e sincronismo em sistemas baseados em troca de mensagens. Finalmente discutir-se-á a questão da implementação de primitivas para o uso de processamento paralelo a nível de linguagem de programação ou a nível de biblioteca de rotinas.

II.1 - POR QUE PROCESSAMENTO PARALELO ?

Ao aumentar a capacidade de processamento, busca-se basicamente uma diminuição no tempo de solução de uma tarefa computacional. Dado um computador com uma determinada funcionalidade, capacidade de memória e capacidade de acesso a periféricos, aumentando-se a sua "velocidade", estaremos basicamente diminuindo o tempo de espera pelo resultado da tarefa que se deseja executar.

O tempo de resposta pode determinar se uma dada tarefa é executável ou não. Suponha um sistema de previsão meteorológica, onde com os dados recolhidos em um dia se possa fazer uma previsão do tempo do dia seguinte. Se o tempo de processamento desses dados for de três dias, podemos considerar esse sistema totalmente inútil. Ele só

passará a ter validade se o tempo de solução for drasticamente diminuído, de forma que a previsão se faça em tempo hábil.

Tendo entendido a importância do tempo de solução, vamos analisar aqui como pode ser obtida a diminuição desse tempo, ao qual o conceito de capacidade computacional está intimamente associado.

O tempo de solução de um problema computacional pode ser sintetizado com a seguinte expressão [5]:

$$\text{tempo de solução}\left[\frac{\text{segundos}}{\text{problema}}\right] = \frac{\text{complexidade}\left[\frac{\text{operações}}{\text{problema}}\right]}{\text{taxa de execução}\left[\frac{\text{operações}}{\text{segundo}}\right]}$$

Na expressão acima, o tempo de solução é medido em número de segundos que um problema leva para ser resolvido. A complexidade é expressa em número de operações necessárias para a resolução do problema e a taxa de execução é medida em número de operações executadas pelo computador por segundo.

Pela expressão acima vê-se que para diminuir o tempo de execução é necessário ou diminuir a complexidade ou aumentar a taxa de execução. Como a complexidade é inerente ao problema a ser resolvido se for assumido que o algoritmo ideal esteja sendo utilizado, conclui-se que é necessário aumentar a taxa de execução para alcançar-se o tempo de solução desejado.

A taxa de execução (TE) pode ser expressa da seguinte forma:

$$TE = \frac{\text{Multiplicidade}\left[\frac{\text{operações / EP}}{\text{ciclos}}\right]}{\text{Ciclo}\left[\frac{\text{segundos}}{\text{ciclo}}\right]} \cdot \text{Concorrência [EP]}$$

onde EP = Entidade de Processamento.

Multiplicidade é o número de operações executadas por ciclo por cada EP. Indica o grau de paralelismo de cada EP, obtido através de técnicas como "pipelining", processamento vetorial e outras. Ciclo é o tempo em segundos que leva cada ciclo de relógio.

A concorrência indica o número de entidades de processamento.

Nos supercomputadores é dada ênfase à melhoria do primeiro termo da expressão de TE: aumento da multiplicidade e, principalmente, na diminuição do ciclo. No processamento paralelo, como abordado nesse trabalho, a ênfase é dada no aumento da concorrência com o emprego de muitas EPs ou processadores.

Cumpre lembrar que os supercomputadores comerciais também costumam usar um certo grau de concorrência, empregando, tipicamente, de dois a oito processadores. As máquinas paralelas por sua vez, também incorporam o aumento da multiplicidade e diminuição dos ciclos. A diferença entre as duas abordagens é a ênfase dada a cada técnica.

Um fator que torna o processamento paralelo atraente é a melhor relação custo/benefício [6]. É extremamente mais barato, dada uma tecnologia, duplicar o número de

processadores do que dobrar a taxa de execução de um único processador.

Outro fator que tem levado ao estudo e utilização de processamento paralelo é a percepção de que, mesmo com toda a evolução tecnológica, é cada vez menor a taxa com que se tem reduzido o ciclo de máquina nas máquinas de altíssimo desempenho. Projeções otimistas sugerem que ciclos de máquina da ordem de 1 nano segundo deve ser o limite que se poderá chegar [7]. Com isso, torna-se imperativo o uso de processamento paralelo para obter capacidade computacional acima do que pode ser oferecido pelo processador mais veloz que a tecnologia existente pode produzir. Isso se tornará cada vez mais comum, já que a demanda de capacidade computacional tem crescido mais rapidamente do que a capacidade de processamento de um único processador.

II.2 - DEFINIÇÕES

Para se fazer um estudo sobre processamento paralelo, é necessário antes definir alguns termos e conceitos que na literatura são apresentados de forma variada e às vezes conflitante.

Em [1] encontramos que *multiprocessadores* podem ser caracterizados por dois atributos: primeiro, "é um único computador com múltiplos processadores" e segundo, "os processadores devem se comunicar e cooperar em diferentes níveis na solução de um dado problema". Os sistemas multiprocessadores se dividem em *fortemente acoplados* e

fracamente acoplados. Os sistemas fortemente acoplados utilizam memória compartilhada para trocar informação enquanto os sistemas fracamente acoplados trocam informação através de um sistema de comunicação que interliga os nós de processamento.

Ainda em [1] encontramos que *sistema de múltiplos computadores* consiste em vários computadores autônomos que podem ou não se comunicar.

AMORIM et alli em [4] consideram *sistemas distribuídos* como sistemas em que os diversos processadores não compartilham memória e, conseqüentemente, toda a comunicação entre processadores deve ser realizada através de troca de mensagens.

KIRNER[8], baseando-se em [9] e [10], usa as seguintes definições:

- um *sistema multiprocessador* é formado por múltiplas CPUs interligadas através de uma memória compartilhada, e apresenta um sistema operacional integrado que gerencia os recursos do sistema como um todo;

- uma *rede de computadores*, por sua vez, é formada por múltiplos computadores autônomos interligados através de um subsistema de comunicação. Quanto ao software, a rede apresenta um sistema operacional que implementa a infraestrutura necessária para a comunicação entre vários computadores, respeitando e mantendo o sistema operacional de cada um;

-um *sistema distribuído* possui um "hardware" formado por uma rede de computadores, e apresenta um sistema

operacional integrado que gerencia os recursos do sistema como um todo.

Em [11] aparece que *sistemas multiprocessadores* são sistemas constituídos de duas ou mais máquinas ligadas entre si através de memória compartilhada (*fortemente conectados*) ou via conexão de dados de alta ou baixa velocidade (*fracamente conectados*).

Nesse trabalho será adotada essa última definição de *sistemas multiprocessadores*. Baseado nessa definição, pode se dizer que os sistemas focalizados nesse capítulo são *sistemas multiprocessadores*.

Um conceito importante para o estudo de multiprocessamento é o conceito de *processo*. *Processos* podem ser entendidos como programas em execução [12, 13]. São entidades seqüenciais que podem executar em paralelo, mas que eventualmente vão precisar se sincronizar, atrasando o seu processamento para esperar algum evento, e receber informações de outros processos [11].

Processos que colaboram na execução de um determinado trabalho são chamados *processos cooperativos*.

II.3 - SISTEMAS FORTEMENTE ACOPLADOS E SISTEMAS FRACAMENTE ACOPLADOS

Para que dois ou mais processadores trabalhem de forma cooperativa é necessário que eles possam se comunicar de alguma forma. A maneira pela qual os processadores se interligam caracteriza um sistema fortemente acoplado ou fracamente acoplado.

Quando os processadores se comunicam através de uma memória comum, o sistema é chamado fortemente acoplado. Os sistemas fracamente acoplados são aqueles que não possuem memória compartilhada e devem possuir um meio de interligação que permita a comunicação através de troca de mensagens.

Nos itens a seguir, serão discutidos os sistemas fortemente acoplados, os fracamente acoplados e será feita uma comparação entre os dois.

II.3.1 - SISTEMAS FORTEMENTE ACOPLADOS

A figura II.1 mostra um sistema fortemente acoplado, onde vários processadores compartilham uma memória e um espaço de endereçamento comum a todos eles.

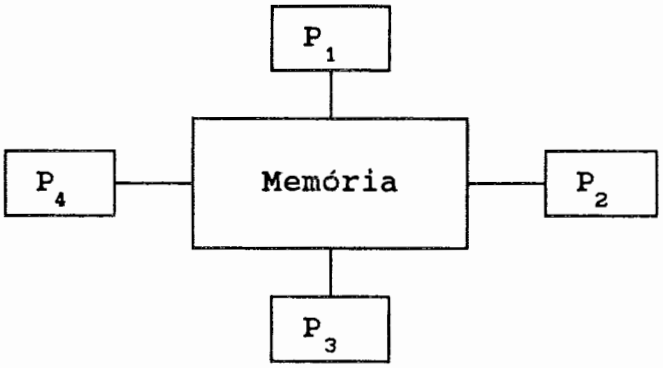

Figura II.1 - Sistema Fortemente Acoplado

Um problema encontrado nos sistemas com memória compartilhada é a degradação da performance causada pela contenção do barramento [14]. Essa contenção se deve ao fato de vários processadores competirem pelo uso do

barramento para acessar a memória. Essa situação é ilustrada na figura II.2, que mostra a implementação física de um sistema fortemente acoplado, onde os processadores P_i compartilham a memória M através de um único barramento.

Figura II.2 - Sist. Fortemente Acoplado com um único barramento

Para amenizar esse problema de contenção de barramento, dois métodos básicos podem ser utilizados, isolada ou conjuntamente: o uso de memória local e de memória "cache" para diminuir o acesso à memória global e o aumento do paralelismo no acesso à memória com a multiplicação de barramentos e de módulos de memória.

II.3.1.1 - Uso de Memória Local e Memória "Cache"

O uso de uma memória local para os programas e dados locais e a utilização da memória compartilhada apenas para os dados globais permitem que haja um menor acesso à memória global. Com isto, diminui-se a disputa pelo barramento e a contenção no acesso à memória global.

Uma variação dessa implementação permite que a memória local de um processador possa ser acessada pelos outros processadores. Nesse caso, a memória global pode

ser eliminada, ficando os dados globais distribuídos entre as memórias locais dos processadores.

O uso de memória "cache" local a cada nó também é uma forma de diminuir o acesso à memória. No entanto, se cada processador de um sistema fortemente conectado tiver uma memória "cache" individual, são necessários mecanismos, às vezes complexos,para resolver o problema de coerência de "cache". Esse problema ocorre quando um processador atualiza uma posição de memória e o dado antigo desse endereço já se encontra no "cache" de outro processador. Nesse caso, é necessário que o "cache" seja atualizado ou invalidado.

Essa questão da coerência de "cache" pode também ser resolvida por software, fazendo com que apenas as instruções e os dados locais passem pelo "cache". Os dados globais não são colocados na memória "cache" e não há necessidade dos mecanismos complexos de coerência de "cache". A desvantagem desse método é que o acesso aos dados globais fica mais lento. Esse método pode ser melhorado se for possível identificar os dados que são apenas lidos e os dados que são escritos. Nesse caso, os dados globais que não são alteráveis também podem passar pelo cache.

II.3.1.2 - Uso de Múltiplos Barramentos e Memórias

Já que o acesso à memória global é um gargalo dos sistemas fortemente acoplados, o uso de mais de um módulo de memória e mais de um barramento serve para aumentar o

desempenho do sistema.

No entanto essa técnica implica um custo adicional na implementação de múltiplos barramentos e de um sistema de arbitramento e chaveamento mais complexo. Além disso, como o circuito é mais complexo, o atraso no acesso de cada nó individualmente à memória é maior, devendo ser avaliado se os ganhos obtidos com o paralelismo nesse acesso são maiores do que as perdas provocadas pelo atraso.

Um dos sistemas que permite a obtenção dessa multiplicidade no acesso à memória é a chave "crossbar" [1, 2] que pode ser vista na figura II.3. Com essa chave, podem existir até m processadores, onde m é o número de módulos de memória, fazendo acesso a algum módulo de memória simultaneamente. Se m for maior que o número p de processadores, só haverá contenção se mais de um processador quiser ter acesso um determinado módulo de memória num mesmo instante. Nesse caso o sistema de arbitramento determinará qual processo terá acesso à memória, baseado numa política de prioridades estabelecida na implementação do sistema.

O número de linhas e circuitos de chaveamentos de um sistema "crossbar" é um dos fatores que restringe o número de módulos de memória e processadores que podem ser interconectados. Uma alternartiva para esse sistema é o uso de Redes de Chaveamento ou de Interconexão.

A figura II.4 mostra uma rede de interconexão com 4 processadores e 4 módulos de memória, usando 2 estágios de chaves de 2 entradas e 2 saídas.

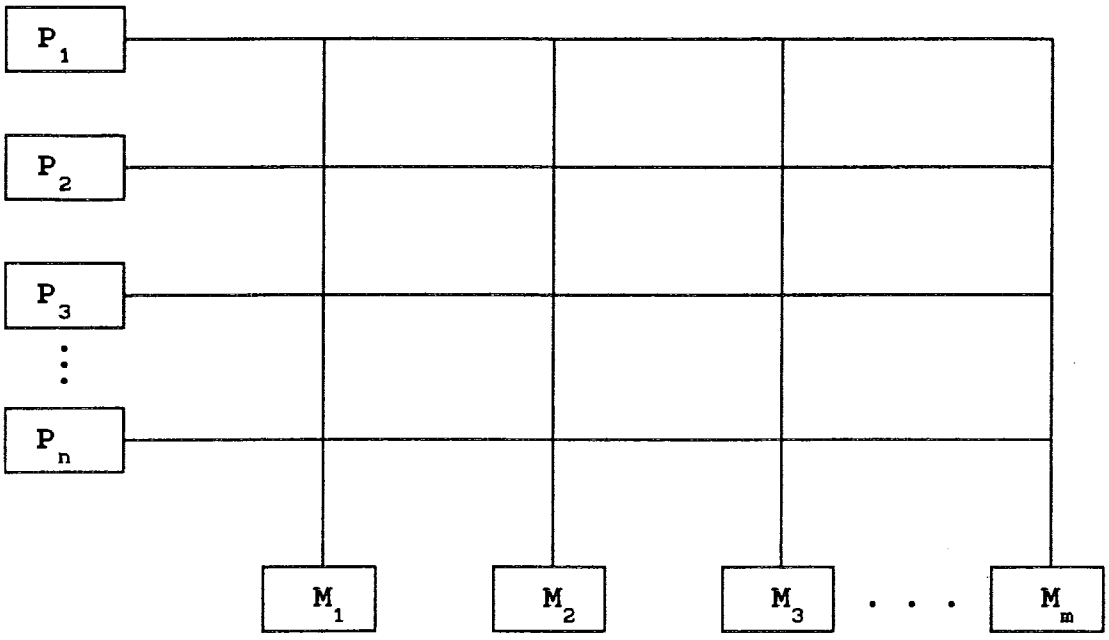

Figura II.3 - Chave "Crossbar"

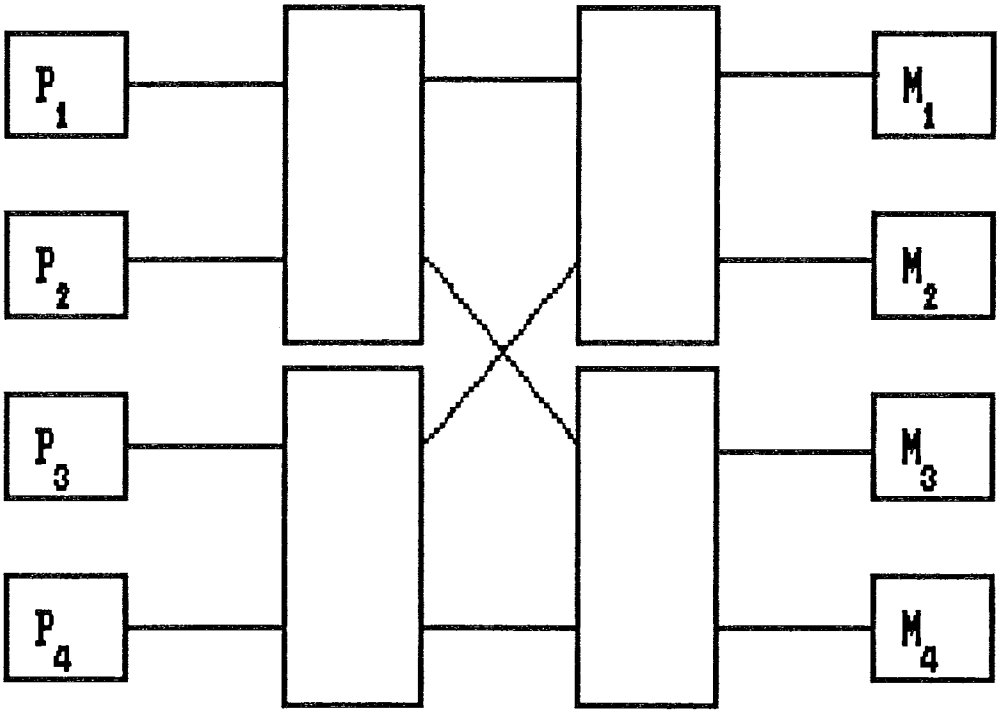

Figura II.4 - Rede de Interconexão

O uso de redes de interconexão permite uma redução no número de chaves e uma simplificação do arbitramento e chaveamento dos circuitos. A desvantagem em relação ao "crossbar" é a redução do número de acessos concorrentes pois cada chave suporta apenas uma ligação num determinado instante. Outra desvantagem é o aumento do retardo para o estabelecimento da interconexão à medida em que novos estágios vão sendo acrescentados ao circuito. É uma solução de compromisso entre o barramento único e o "crossbar".

II.3.2. - SISTEMAS FRACAMENTE ACOPLADOS

Sistemas que não possuem memória compartilhada são chamados sistemas fracamente acoplados. Nesse caso os processadores se comunicam através de troca de mensagens.

Inúmeras são as formas como os processadores de um sistema fracamente acoplado podem ser interligados. Essa interligação deve atender a vários critérios: Comunicação eficiente entre os processadores, simplicidade de interconexão dos nós e topologia adequada à aplicação [15]. Serão apresentados a seguir algumas das formas possíveis de interligação de processadores.

Uma maneira bastante comum de se conectar processadores é através de barramentos seriais ou paralelos. Esses barramentos podem ser barramentos padronizados, como o VME ou Ethernet, ou barramentos especificamente desenvolvidos para um determinado multiprocessador. O problema com esse tipo de interconexão

é o barramento poder vir a ser um gargalo do sistema com o aumento do número de processadores, enquanto a vantagem é a comunicação diretamente de um nó com qualquer outro nó do sistema.

Outra forma de conexão é a rede em anel, ilustrada na figura II.5. Nessa conexão, cada processador está ligado a outros dois processadores, formando um anel fechado.

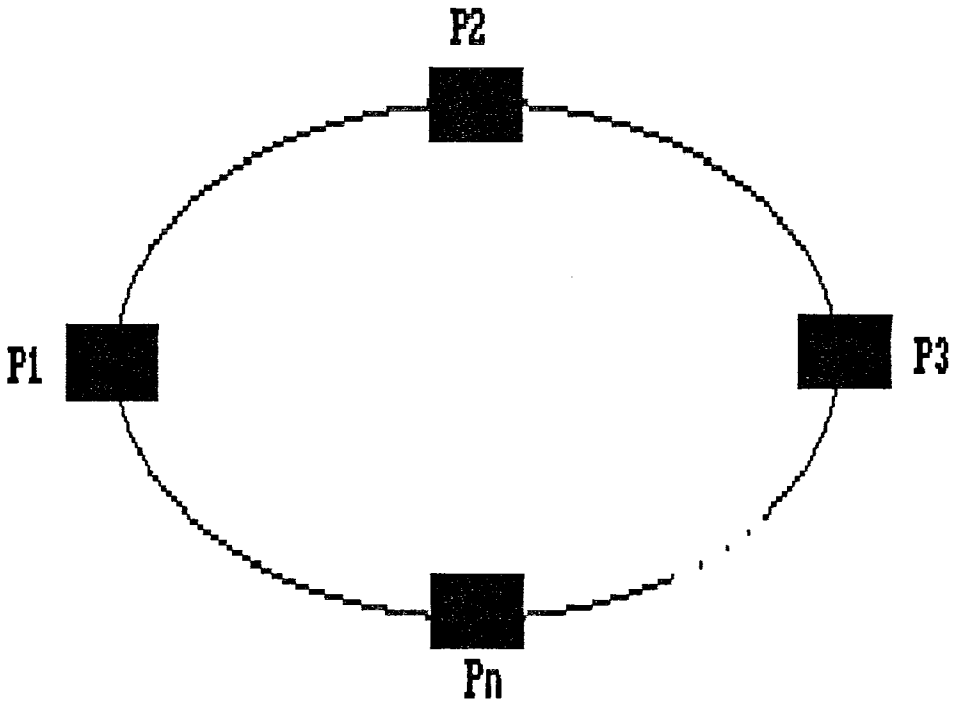

Figura II.5 - Rede em Anel

A solução ideal para a comunicação entre os processadores é aquela onde cada processador pode se comunicar diretamente com outro processador sem interferir na comunicação dos demais processadores. Para isso, cada nó deve ter um canal de comunicação com cada um dos outros

nós. No entanto, um grande número de ligações torna essa solução impraticável.

Para contornar esse problema do número de canais de comunicação, são adotadas algumas topologias que diminuem o número de canais, ao custo de uma mensagem ter de transitar através de vários nós antes de atingir o seu destino.

Uma dessas topologias é a rede em malha regular, onde um processador se comunica com os quatro processadores vizinhos. Essa topologia pode ser vista na figura II.6. Apesar de o número de processadores envolvidos numa comunicação poder ser muito grande, algumas aplicações, onde os nós só necessitam se comunicar com os seus vizinhos, podem ser perfeitamente mapeadas nessa topologia.

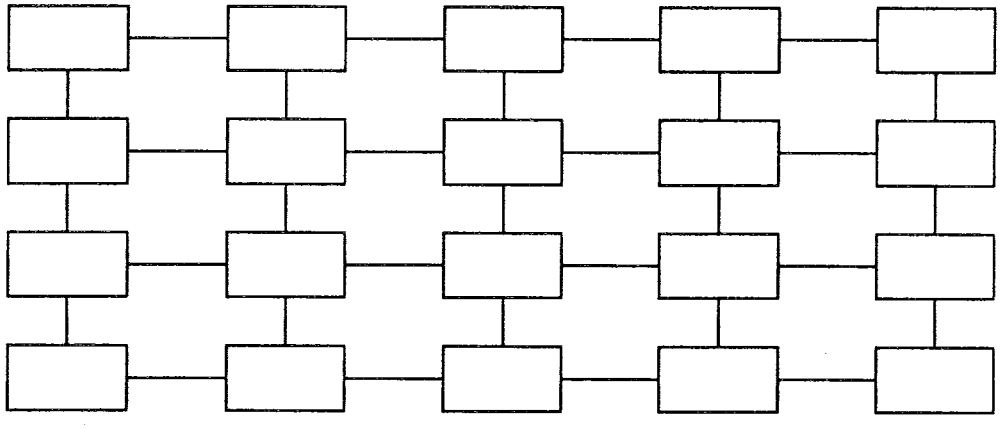

Figura II.6 - Rede em Malha Regular

Uma outra topologia adotada é a conhecida como rede hipercúbica [16]. É uma solução de compromisso entre a interconexão direta entre todos os processadores e o uso

de rede em malha ou anel. No primeiro caso, há um grande número de ligações físicas entre os processadores, mas uma mensagem não precisa passar por nenhum nó intermediário para atingir o seu destino. No segundo caso, o número de ligações fica reduzido, mas uma mensagem pode ter de transitar por um grande número de nós até chegar ao seu destino. O uso de hipercubos é muito difundido nos computadores paralelos comerciais.

Numa rede hipercúbica de 2^p processadores, cada processador está ligado a p outros processadores. Cada processador é o vértice de um "cubo" de p dimensões. Na figura II.7 podemos ver uma rede hipercúbica de 3 dimensões, com oito (2^3) nós.

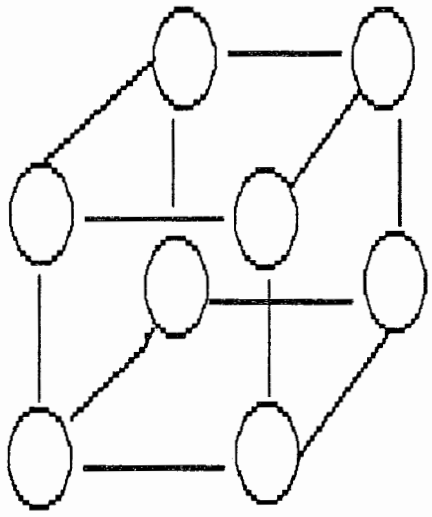

Figura II.7 - Rede Hipercúbica

II.3.3 - COMPARAÇÃO ENTRE SISTEMAS FORTEMENTE ACOPLADOS E SISTEMAS FRACAMENTE ACOPLADOS

Os sistemas fracamente acoplados são em geral mais simples de serem expandidos, podendo, em geral, apresentar um número mais elevado de processadores do que os sistemas fortemente acoplados.

A existência de meios de interligação bastante padronizados, como por exemplo o Ethernet ou o barramento VME, permite que máquinas de diferentes origens possam ser ligadas num mesmo sistema. Isso não ocorre nos sistemas fortemente acoplados.

Já a troca de dados em sistemas fortemente acoplados costuma ser mais rápida do que em sistemas fracamente acoplados. Isso ocorre porque o acesso à memória compartilhada em geral é mais rápido do que a troca de mensagens através de uma sistema de interconexão.

Este fato sugere que programas com grande quantidade de troca de informação, relativamente ao tempo de processamento, podem apresentar um desempenho melhor nas máquinas fortemente acopladas. Dependendo da taxa de comunicação apresentada pelo sistema de mensagens de uma máquina fracamente acoplada, o uso desse tipo de programa pode se tornar muito ineficiente.

Quanto ao escalonamento de processos, num multiprocessador fortemente acoplado, é possível a transferência de um processo de um processador para outro. Já nos sistemas fracamente acoplados isso não costuma ocorrer devido ao longo tempo que essa transferência

levaria. Além disso, a duplicação de um processo, numa operação como o *fork* do UNIX, pode ser feita muito rapidamente nos sistemas fortemente acoplados. Isso é possível porque o segmento de código e de dados que são apenas lidos não precisam ser duplicados, pois dois ou mais processos podem compartilhar a memória que contém esses segmentos.

Devido ao fato de os sistemas fortemente acoplados muitas vezes apresentarem memórias locais, conforme já visto, essas vantagens em relação à alocação e criação podem deixar de existir.

LUSK et alli em [6] fazem uma proposta de um modelo híbrido, onde grupos de processadores fortemente acoplados são interconectados por um sistema de troca de mensagens. Com isso ter-se-ia as vantagens dos sistemas fortemente acoplados preservando-se a possibilidade de expansão dos sistemas fracamente acoplados.

II.4 - VARIÁVEIS COMPARTILHADAS E TROCA DE MENSAGENS

Do ponto de vista lógico, existem duas maneiras pelas quais processos podem se comunicar e sincronizar: usando variáveis compartilhadas ou através de trocas de mensagens [1, 6, 11, 17].

O primeiro método é o mais natural quando se usa sistemas fortemente acoplados e consiste, como o nome já diz, no compartilhamento de variáveis ou posições de memória por mais de um processo. É através dessas variáveis que os processos se comunicam e sincronizam.

O uso de troca de mensagens é mais natural nos sistemas fracamente acoplados. Nos sistemas baseados em troca de mensagens, processos se comunicam através de mensagens que fluem através de vias de comunicação. Além de servir para a troca de dados entre processos, o uso de troca de mensagens permite o sincronismo de processos: Um processo pode esperar outro atingir determinado ponto de sua execução aguardando uma mesagem do mesmo.

Apesar do uso de variáveis compartilhadas ser mais comum em sistemas fortemente acoplados e de troca de mensagem em sistemas fracamente acoplados, esses modelos são intercambiáveis, sendo possível implementar um modelo de programação baseado em troca de mensagens usando variáveis compartilhadas e vice-versa [6, 18].

Programas baseados em troca de mensagens são considerados mais portáteis e mais fáceis de verificar do que aqueles baseados em variáveis compartilhadas. No entanto, enquanto alguns algoritmos são eficientemente formulados em termos de troca de mensagens, outros são mais adequados ao uso de variáveis compartilhadas. A questão de desempenho é um fator a ser levado em conta na escolha entre o uso de variáveis compartilhadas ou de troca de mensagens, mas que depende do tipo de algoritmo a ser utilizado [6].

Quando se usa variáveis compartilhadas, cuidado deve ser tomado no acesso a essas variáveis, sendo necessária a implementação de exclusão mútua no acesso a elas. Essa questão será discutida mais adiante nesse capítulo.

Mais adiante serão discutidas também as diferentes primitivas que podem ser usadas para troca de mensagens.

LISKOV em [19] apresenta um modelo de programação híbrida onde grupos de processos, que fazem parte de uma mesma estrutura chamada "guardian", se comunicam com variáveis compartilhadas. Processos pertencentes a "guardians" diferentes se comunicam através de troca de mensagens.

II.5 - CRIAÇÃO ESTÁTICA DE PROCESSOS X CRIAÇÃO DINÂMICA DE PROCESSOS

Uma questão que se apresenta quando se trabalha com processos cooperativos é a questão do momento em que os processos devem ser criados.

Uma abordagem é a criação do processo no momento em que ele é necessário. A isso chamamos criação dinâmica de processos.

Uma outra forma de se tratar o problema é simplesmente alocar todos os processos no início da execução da tarefa. Esse método é bem mais simples de ser implementado e chamamos de alocação estática de processos.

Uma vantagem da alocação dinâmica é o melhor aproveitamento dos recursos computacionais, já que o processo só existe enquanto executa trabalho útil. Além disso, não é necessário saber a priori quantos e quais processos serão criados. Outra vantagem é a possibilidade de se fazer um balanceamento mais eficiente de carga entre os processadores.

Em geral a criação de processos é uma operação que leva um determinado tempo e implica um certo custo em termos de tempo de processamento. Para a criação de um processo, é necessário designar um processador para executá-lo, alocar memória e muitas vezes carregar o programa de disco, além da criação, pelo sistema operacional, de tabelas com as informações necessárias para o gerenciamento e escalonamento de processos. Todo esse trabalho pode tornar a criação dinâmica de processos uma solução inviável em alguns sistemas, apesar das vantagens que poderia apresentar.

II.6 - EXCLUSÃO MÚTUA NO ACESSO A VARIÁVEIS COMPARTILHADAS

Quando mais de um processo tem acesso a uma mesma variável, às vezes é necessário que haja exclusão mútua no acesso a essa variável para garantir a consistência da informação.

Para exemplificar esse problema, consideremos um contador que pode ser incrementado por dois processos. A operação de incrementar um contador pode ser dividida em três etapas: Ler o valor corrente do contador, somar um a esse valor, escrever de volta a variável atualizada.

Suponha agora que um processo A deseja incrementar um contador que também pode ser incrementado por um processo B. Como esses processos são independentes, nada podemos assumir sobre a velocidade relativa deles. Pode acontecer a seguinte seqüência de eventos:

Processo A lê o valor do contador

Processo A incrementa o valor do contador

Processo B lê o valor do contador

Processo A escreve novo valor do contador

Processo B incrementa o valor do contador

Processo B escreve novo valor do contador

Podemos observar que a atualização feita pelo processo A foi perdida, acarretando um erro no valor final do contador que foi incrementado de um ao invés de dois.

Para que esse tipo de erro não ocorra, é necessário que um processo tenha acesso exclusivo à variável desde o momento imediatamente anterior à leitura da variável até momento imediatamente posterior à gravação do valor atualizado da variável. A essa parte do programa que tem acesso a uma variável compartilhada, com necessidade de exclusão mútua, chamamos *região crítica* [20].

Várias são as soluções existentes para a implementação de exclusão mútua. Algumas soluções são a nível de "hardware" e outras a nível de "software".

As soluções de "hardware" são muito comuns em sistemas monoprocessadores multiprogramáveis mas, a princípio, podem ser também implementadas em sistemas multiprocessadores fortemente acoplados. No caso de sistemas monoprocessadores, são utilizadas as instruções do tipo "test and set" que consiste em ler uma variável booleana, guardar o seu valor e tornar essa variável verdadeira, tudo isso numa única operação indivisível que não pode ser interrrompida. Nos sistemas multiprocessadores, essa operação costuma ser chamada

"read-modify-write". A idéia é um processador ler uma variável, modificá-la e escrevê-la de volta sem que outro procesador possa ter acesso a ela. Isso pode ser obtido com o bloqueio do barramento, por exemplo.

O matemático holandês Dekker apresentou uma elegante solução a nível de "software" para a exclusão mútua entre dois processos [20]. DIJKSTRA em [21] estendeu essa solução para *n* processos.

Variáveis compartilhadas são usadas também para sincronizar processos, como nos casos de processsos com relação do tipo *produtor-consumidor*.

Posteriormente foram propostas construções mais elegantes para a implementação de exclusão mútua e sincronização, como *semáforo* e *monitor*. Essas construções serão apresentadas a seguir.

II.6.1 SEMÁFORO

DIJKSTRA em [20] introduziu o conceito de *semáforo* para a implementação de exclusão mútua. *Semáforos* são variáveis protegidas as quais só podem ter acesso através das operações P e V.

A operação P num semáforo S, cuja notação é P(S), pode ser descrita da seguinte forma:

```
se S > 0
     então S := S - 1
     senão (espere por S)
```

A operação V num semáforo S, denotado V(S), pode ser

descrita da seguinte maneira:

```
se (existe processo esperando por S)
     então (acorde um desses processos)
     senão S:= S + 1
```

As operações P e V são usadas para encapsular regiões críticas.

II.6.2 MONITOR

Monitor [22, 23] é uma estrutura passiva que define um conjunto de dados compartilhados e operações ou procedimentos que podem ser executados nesses dados. Assim o monitor agrupa todas as regiões críticas na sua estrutura monolítica, retirando-as do corpo dos processos. Num determinado instante, apenas um processo pode estar executando os procedimentos ou funções de um monitor.

Com o uso de monitores, regiões críticas passam a ser executadas no monitor. A exclusão mútua é assegurada pelo fato de apenas um processo poder executar o monitor num determinado instante.

A implementação de monitores deve prover maneiras de suspender um processo quando desejar executar um procedimento de um monitor que esteja em uso e de acordar esse processo quando o monitor estiver livre.

II.7 - PRIMITIVAS DE COMUNICAÇÃO E SINCRONISMO EM SISTEMAS BASEADOS EM TROCA DE MENSAGENS

Trocas de mensagens são feitas através do uso de

primitivas de envio e recebimento de mensagens. Elas podem ser vistas como operações de entrada e saída, onde a saída de um processo é a entrada de outro [24].

Várias são as questões relacionadas com essas primitivas [25]. Dentre elas destacam-se: Como é feito o endereçamento e se as primitivas são síncronas ou assíncronas. Essas questões serão tratadas nos itens a seguir.

O uso de troca de mensagens para a obtenção de sincronismo e troca de dados pode ser escondido da programação com o uso de construções de mais alto nível. Essas cosntruções são implementadas usando as primitivas de envio e recepção já mencionadas. Dentre essas construções destacamos a *Chamada Remota de Procedimentos* que será apresentada mais adiante.

II.7.1 - ENDEREÇAMENTO

Um processo pode ser endereçado diretamente com um nome ou um número ou pode existir um canal de comunicação criado dinamicamente para associar a entrada de um processo à saída de outro processo.

As mensagens podem ser enviadas de um processo para outro ou de um processo para um conjunto de outros processos. Esse último caso é chamado "broadcast".

Ainda associada a essa questão de endereçamento, a primitiva de recepção pode especificar o endereço do processo emissor e só aceitar uma mensagem do processo designado ou pode dispensar a identificação do processo

emissor, aceitando mensagens de qualquer processo. Esse último caso é mais adequado quando se usa o modelo cliente-servidor, onde o servidor pode atender a diversos clientes.

II.7.2 - PRIMITIVAS SÍNCRONAS E ASSÍNCRONAS

Podemos dividir as primitivas de comunicação em bloqueantes ou síncronas e não-bloqueantes ou assíncronas. Assim, podemos ter envio bloqueante e não-bloqueante e recepção bloqueante e não-bloqueante.

No caso do envio bloqueante, o processo que envia é suspenso ou bloqueado até que o processo destinatário tenha chamado a primitiva de recepção e receba a mensagem. No caso não-bloqueante, o processo não precisa esperar que o outro esteja pronto para receber. O mesmo raciocínio pode ser feito para a primitiva de recepção.

Qualquer combinação de envio e recepção bloqueante ou não-bloqueante pode ser usada. No caso totalmente síncrono, tanto a primitiva de envio como a de recepção são bloqueantes. No caso totalmente assíncrono, ambas as primitivas são não-bloqueantes.

A vantagem do método síncrono é que a implementação é mais simples e combina, de forma elegante as primitivas de comunicação e sincronismo [11]. A vantagem do método totalmente assíncrono é que permite um maior grau de paralelismo. O sincronismo obtido com o uso de primitivas bloqueantes de envio e recepção é conhecido como *Rendez-vous*.

No caso do envio não-bloqueante, é necessário que existam "buffers" onde as mensagens são armazenadas até que a mensagem seja recebida pelo processo destinatário. Como o número de "buffers" numa implementação real é necessariamente finito, surge uma questão: o que fazer quando um processo envia uma mensagem e não há "buffers" disponíveis para recebê-la? Duas soluções se apresentam. Numa solução a primitiva de envio retorna um código que indica que o envio não pode ser completado por falta de "buffers". Na outra solução o processo é bloqueado até que haja um "buffer" disponível. Nesse caso, o envio é não-bloqueante se houver "buffer" disponível e se torna bloqueante se não houver "buffers".

II.7.3 -CHAMADA REMOTA DE PROCEDIMENTOS

A idéia da *Chamada Remota de Procedimentos* [26, 27] é que um processo chame outro para executar um procedimento ou sub-rotina. O processo que executa a sub-rotina é chamado processo *servidor* enquanto o processo que chamou é chamado *cliente*. Com o uso da *Chamada Remota de Procedimentos*, o processo cliente controla a execução de uma sub-rotina em outro processo, que em geral se localiza em outro processador. Do ponto de vista do programa cliente, a *Chamada Remota de Procedimentos* é similar a uma chamada a uma sub-rotima local, com passagem de parâmetros e resultados. A diferença, que dependendo da implementação pode ser transparente ao programador, é onde a sub-rotina é executada.

A *Chamada Remota de Procedimentos* é uma operação síncrona. O processo *cliente*, após chamar a rotina remota, é bloqueado até o término da execução por parte do processo *servidor*.

II.8 - USO DE PRIMITIVAS DE PROCESSAMENTO PARALELO EM LINGUAGENS DE PROGRAMAÇÃO E BIBLIOTECAS DE ROTINAS

Processos cooperativos necessitam de alguma forma se comunicar e sincronizar. Para isso é necessário que existam primitivas que permitam que os programas possam ser implementados. Inúmeras são as possibilidades para a definição e implementação dessas primitivas.

Essas primitivas podem estar definidas na linguagem de programação ou como um conjunto de rotinas. A seguir será feita uma análise dessas duas opções.

II.8.1 - LINGUAGEM DE PROGRAMAÇÃO

Linguagens de programação especificadas para processamento paralelo devem prover mecanismos que permitam a troca de informação e sincronismo entre processos.

Além disso, devem permitir a indicação de trechos que podem ser executados em paralelo e os trechos que devem ser executados seqüencialmente. Isso pode ser feito de duas maneiras. Uma maneira é prover a linguagem de diretivas que indiquem as operações que podem ser executadas em paralelo, como o *parbegin/parend* proposto por DIJKSTRA em [20]. A outra forma é usar o conceito de

processos seqüenciais e escrever cada um dos processos que compõe o programa separadamente.

As linguagens de programação para processamento paralelo podem adotar o modelo de variáveis compartilhadas, com o uso de monitores, ou o modelo de troca de mensagens. No primeiro grupo podemos citar Pascal Concorrente [28] e Modula [29]. No segundo grupo podemos citar CSP [24], Distributed Processes [26], ADA [17], OCCAM [30] e C Concorrente [31].

A implementação de processos concorrentes a nível de linguagem de programação tem a vantagem de tornar o programa mais compreensível. Outra vantagem é a possibilidade de verificação em tempo de compilação e otimização por parte dos compiladores.

Uma outra vantagem que a princípio pode ser atribuída ao uso de linguagens de alto nível, é o fato de elas esconderem os detalhes da arquitetura. No entanto, para o melhor aproveitamento das máquinas paralelas, devem ser exploradas as suas características particulares, que diferem de máquina para máquina. Assim, torna-se difícil mapear com eficiência uma linguagem que pretende ser genérica em uma máquina particular. Por outro lado, uma linguagem muito bem adaptada a um tipo de multiprocessador pode apresentar um desempenho fraco em outras arquiteturas [2].

II.8.2 - BIBLIOTECA DE ROTINAS

Ao invés de fazerem parte da linguagem de

programação, as primitivas para o processamento paralelo podem ser implementadas como funções de uma biblioteca de rotinas.

Duas são as vantagens dessa opção. A primeira é poder ser utilizada com as linguagens de programação já existentes, não implicando o aprendizado, por parte do usuário, de uma nova linguagem de programação. A segunda vantagem é a facilidade de portar a biblioteca de rotinas a um novo sistema. É mais fácil portar uma biblioteca de rotinas a um novo sistema do que implementar um compilador para este sistema.

C A P Í T U L O III

DEFINIÇÕES DAS ESTRATÉGIAS ADOTADAS NO PROJETO

Para que se compreenda as estratégias adotadas no desenvolvimento tanto do "hardware" quanto do "software" da segunda geração do sistema ACP é preciso levar em consideração que, além das questões abordadas no capítulo II, dois fatores de ordem prática nortearam o desenvolvimento do sistema.

O primeiro fator era a necessidade da implementação do sistema ser exeqüível num prazo relativamente curto. A razão para isso era que o sistema de processamento paralelo proposto seria utilizado ainda em 1990, nas experiências realizadas no Fermilab.

Outro fator que nos orientou na elaboração desse projeto foi a consciência de que os principais usuários do sistema seriam físicos que já possuíam diversos programas e bibliotecas implementadas em FORTRAN 77. Esses usuários, e provavelmente a maioria dos futuros usuários, teriam uma certa resistência em adotar uma nova linguagem para o processamento paralelo, a não ser que houvesse uma clara indicação de que essa nova linguagem seria uma espécie de padrão a ser amplamente adotado num futuro próximo. Ainda não parece claro que exista uma linguagem ou mesmo um modelo de processamento paralelo que possa ser considerado um padrão de aceitação geral. Dessa forma, o nosso sistema deveria ser capaz de suportar programas em linguagens tradicionais, principalmente FORTRAN 77 e C.

Tendo em mente esses fatores, foi escolhido a adoção de um sistema fracamente acoplado e um modelo de programação baseado em troca de mensagens. Como já discutimos no capítulo anterior, existem vantagens e desvantagens nessa opção. Do ponto de vista do hardware, dois foram os fatores principais que levaram à essa opção:

1) maior capacidade de expansão do sistema;
2) maior simplicidade do sistema.

Do ponto de vista de "software", esse modelo apresenta as seguintes vantagens:

1) maior portabilidade;
2) possibilidade de interconexão com outros computadores.

Optou-se também pela implementação das ferramentas de paralelismo através de biblioteca de rotinas. Essa opção foi mais ou menos óbvia dado o segundo fator condicionante já mencionado, ou seja, a necessidade do uso de FORTRAN ou outra linguagem tradicional. A implementação sob forma de bibliotecas torna trivial a integração do CPS com os compiladores já existentes.

O uso de biblioteca de rotinas também mostrou-se muito apropriado quando decidimos transportar o CPS para outros sistemas. Pelo fato de ser um conjunto de rotinas que funcionam em cima de um sistema operacional, transportá-lo para outros ambientes foi razoavelmente simples. Para isso foi necessário apenas modificar as rotinas de gerenciamento de mensagens ("message handler") e os procedimentos para iniciar os processos em outras CPUs. Essas duas partes do CPS são as únicas cujas

implementações dependem do sistema operacional.

Tendo em vista o primeiro fator anteriormente citado, decidiu-se que o sistema deveria ser implementado com o uso de um sistema operacional comercial já existente, descartando a possibilidade de desenvolvermos um sistema operacional específico para o sistema de processamento paralelo. Com isso pode-se contar com todo um conjunto de facilidades e utilitários já disponíveis em um sistema operacional. Dentre essas ferramentas destacam-se os compiladores, depuradores e editores, sem falar no acesso a todo tipo de periféricos e sistemas de arquivos.

Outra questão que teve de ser analisada foi a da criação dos processos, se dinâmica ou estática. As vantagens e desvantagens dessas duas opções já foram discutidas no capítulo II.

No CPS foi adotada a criação estática de processos. Isso foi uma consequência natural da utilização de sistemas operacionais convencionais. Nesse caso, o "overhead" da criação de um processo torna proibitivo o uso de criação dinâmica.

Estas foram as diretivas básicas para a definição do CPS e do sistema ACP como um todo. No capítulo V o CPS será tratado com detalhes.

C A P Í T U L O IV
ARQUITETURA DO SISTEMA

IV.1 - INTRODUÇÃO

Descreveremos nesse capítulo a arquitetura da segunda geração do sistema ACP, para o qual o CPS foi concebido.

O sistema é constituído de nós de processamento, denominados ACP/R3000, interligados através de um barramento VME (Versa Modular Eurocard) [32]. Até 16 bastidores VME podem ser conectados através de um barramento "Branch Bus" [33].

Além dos nós de processamento, são usados no barramento VME controladores comerciais SCSI para conectar discos e fitas aos nós. Uma interface Ethernet é usada para interconectar o sistema ACP a outros equipamentos, como por exemplo, estações de trabalho gráficas. Os outros dois módulos utilizados no barramento VME são o VBBC [34]e a BVI [35] que são utilizados para a interligação do VME ao Branch Bus.

A figura IV.1 [36] mostra a arquitetura básica do sistema.

Nos parágrafos seguintes serão apresentados o ACP/R3000, VME, Branch Bus, VBBC e BVI e o software básico do ACP.

IV.2 - ACP/R3000

O elemento básico do sistema ACP é o módulo processador ACP/R3000. O ACP/R3000, cujo diagrama de

blocos pode ser visto na figura IV.2 [36], é um módulo VME que utiliza o microprocessador R3000 [37] de tecnologia RISC ("Reduced Instruction Set Computer") desenvolvido pela MIPS Computer Systems Inc. É composto de dois módulos: a "mother board" e a "daugther board".

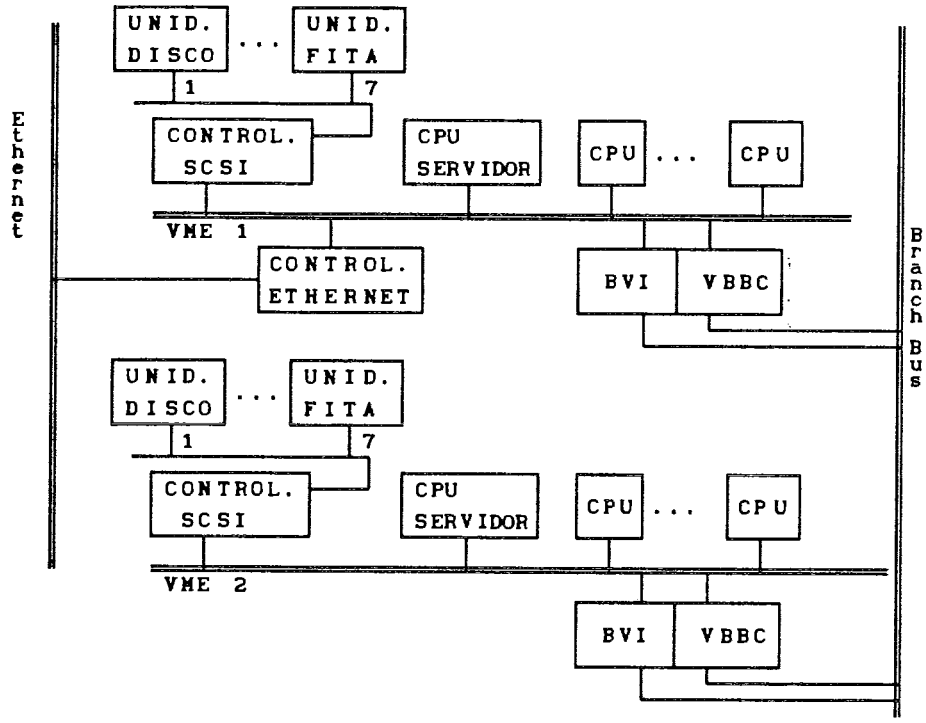

Figura IV.1 - Arquitetura Básica

A "mother board" é um módulo com uma interface VME, uma interface XBUS [38] e 8 a 32 Mbytes de memória. O XBUS é basicamente um barramento de memória que usa pinos não utilizados do VME e serve para: Acoplamento de até quatro processadores com compartilhamento de memória, possibilitando a implementação de um sistema fortemente acoplado; expansão de memória; e conexão de módulos de aquisição de dados.

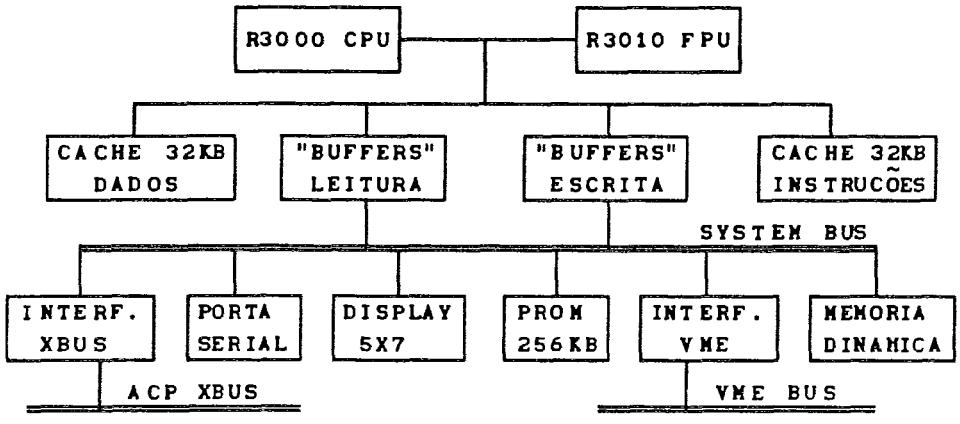

Figura IV.2 - Diagrama de Blocos do ACP/R3000

A "daugther board" é a placa que contém a CPU R3000 e ainda processador de ponto flutuante R3010, também da MIPS; "cache" de dados e instrução separados, podendo cada uma ter de 32 a 128 Kbytes; 256 Kbytes de EPROM, onde é armazenado o programa monitor que possibilita a carga do sistema operacional e testes; um buffer de escrita; e FIFO de interrupções, usado para a implementação de comunicação entre processadores; interface serial padrão RS-232; display alfanumérico e um relógio programável.

IV.3 - O BARRAMENTO VME

O barramento VME é um barramento multi-mestre usado para a interconexão de módulos de processamento, módulos de memória e módulos de entrada/saída.

A sua natureza assíncrona permite a interligação de módulos de características e velocidades diferentes.

O VME define quatro níveis de prioridade para o acesso ao barramento. O arbitramento é feito por um dos

módulos. Também estão definidos sete níveis de interrupção. O barramento também possui recursos para implementação de um ciclo de "read-modify-write", descrito no capítulo II.

IV.4 - BRANCH BUS

O Branch Bus é um barramento de 32 bits de dados que apresenta uma taxa de transmissão de até 20 Mbytes por segundo. Ele é usado para a interconexão de barramentos VME.

Esse barramento foi desenvolvido no Fermilab.

IV.5 - VBBC E BVI

Os módulos VBBC (VME Branch Bus Controller) e BVI (Branch Bus to VME Interface) são os módulos que possibilitam o acesso ao barramento Branch para a comunicação entre bastidores VME. A VBBC é escrava no barramento VME e mestre no Branch, enquanto a BVI é escrava no barramento Branch e mestra no VME. Assim se um nó quer se comunicar com um nó de outro bastidor, ele faz o acesso à VBBC que transfere, via Branch Bus, os dados para a BVI do outro bastidor, que por sua vez passa a informação para o nó destinatário.

A figura IV.3, mostra o fluxo de dados numa comunicação entre nós de bastidores diferentes.

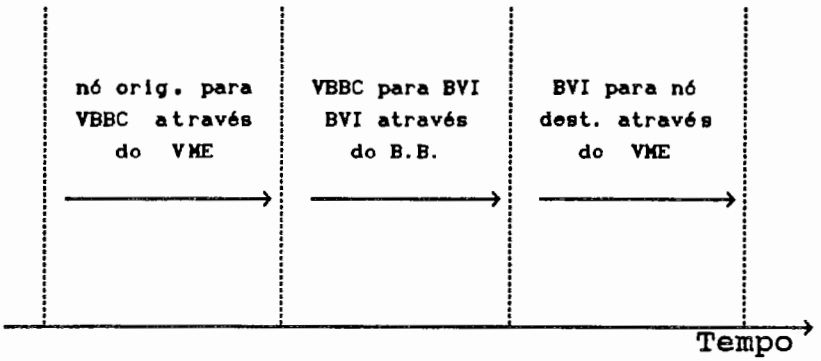

Figura IV.3 - Comunicação entre bastidores diferentes

IV.6 - SOFTWARE BÁSICO DO SISTEMA ACP

O "software" básico do sistema ACP consiste de dois elementos:

1. Software residente em PROM que inicializa o processador, executa rotinas de diagnóstico e permite a carga do sistema operacional.
2. Sistema Operacional.

No sistema ACP, usamos o sistema operacional UNIX gerado a partir de uma versão do UNIX da MIPS Computer Systems, Inc. [39]. Essa versão funciona em máquinas da MIPS que usam o mesmo processador R3000 utilizado no sistema ACP.

Além das modificações relativas ao uso em um "hardware" diferente, foram feitas alterações para que o UNIX rodasse em CPUs sem disco. Essas últimas modificações foram feitas para que não fosse necessário um disco para cada CPU do sistema, que, numa configuração típica, é formado por dezenas de CPUs. Dessa forma, foi possível simplificar e reduzir o custo de um sistema.

Com as modificações acima, um ou mais nós controlam os discos e funcionam como servidores para os outros nós que não possuem discos. A versão original do UNIX assume que uma CPU tem o seu próprio disco de onde ele carrega o sistema operacional e onde faz a paginação e o "swapping".

CAPÍTULO V
DESCRIÇÃO DO CPS

O CPS é um pacote de "software" que oferece meios para que processos possam trabalhar de maneira cooperativa e assim explorar o potencial oferecido pelo processamento paralelo.

A idéia básica é permitir que vários processos seqüenciais trabalhem conjuntamente na resolução de uma tarefa. Cada um desses processos utiliza os recursos normais de uma linguagem de alto nível e os serviços do sistema operacional da máquina onde está sendo executado, bem como os mecanismos oferecidos pelo CPS para comunicação, sincronismo, transferência e conversão de dados.

Esses mecanismos são implementados usando primitivas básicas de troca de mensagens em cima das quais são implementadas as funções de mais alto nível do CPS que permitem ao programador desenvolver as suas aplicações sem ter de usar essas primitivas. Elas, no entanto, estão disponíveis para os usuários que desejarem desenvolver soluções mais particulares.

O CPS é composto de duas partes: Um programa, chamado "Job Manager" ou JM, que gerencia a execução do trabalho computacional realizado pelos processos cooperativos e uma biblioteca de rotinas onde estão implementadas as funções de comunicação, sincronismo e transferência de dados.

Como o objetivo de CPS é permitir a paralelização de

tarefas computacionais, os processos que compõem uma aplicação, em geral, estarão distribuídos entre vários processadores. No entanto, mais de um processo pode rodar em uma mesma CPU, sendo possível que todos eles executem em um único processador, simulando assim um ambiente de processamento paralelo em um único computador.

Tipicamente, na fase de desenvolvimento e depuração, todos os processos podem rodar no mesmo processador e na fase de produção eles são distribuídos pelos processadores disponíveis, conforme mostra a figura V.1. O número de processos e os processadores utilizados são definidos na fase de execução, não sendo necessário modificar ou recompilar os programas quando aqueles forem alterados.

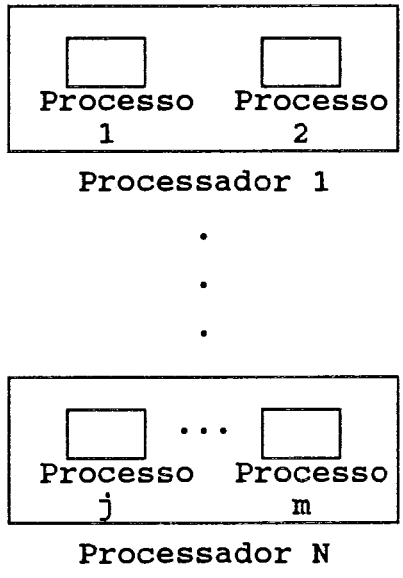

Figura V.1 - Distribuição de Processos entre Processadores

Nos ítens a seguir, serão apresentados a metodologia para desenvolvimento de aplicações com o CPS, as primitivas de troca de mensagens e os mecanismos de

sincronismo, tranferência de dados e conversão de dados. Também serão apresentados o funcionamento e funções do "Job Manager" e as formas de depuração dos programas que usam o CPS.

V.1 - METODOLOGIA PARA DESENVOLVIMENTO DE APLICAÇÕES COM O CPS

Para se desenvolver processos paralelos com o CPS, a primeira tarefa a ser feita é dividir o trabalho a ser executado em partes, conforme as funções a serem executadas. Cada função será executada por um ou mais processos. Os processos que executam a mesma função formam uma *classe de processos* e todos os processos de uma mesma *classe* executam o mesmo programa.

Para tornar mais clara a idéia de *classes de processos* e futuros conceitos apresentados, será usado um exemplo que apresenta uma estrutura muito simples: a reconstrução dos dados de uma experiência de física experimental de altas energias. Nessa recontrução, podemos dividir o trabalho em três funções diferentes:

1. ler dados de fitas de entrada;
2. analisar esses dados e produzir eventos reconstruídos;
3. gravar dados reconstruídos em fitas de saída.

Cada uma dessas funções é executada pelos processos de uma classe e portanto o trabalho é dividido em três *classes*. A classe 1 lê fita, a classe 2 faz a análise e a classe 3 grava os dados reconstruídos.

Nesse exemplo, ilustrado na figura V.2, a tarefa que

exige maior utilização de CPU, é a tarefa da classe 2, ou seja, a de análise. Nesse caso, podemos usar um processo na classe 1, um processo na classe 3 e diversos processos na classe 2, paralelizando assim as atividades dessa classe que são as mais demoradas.

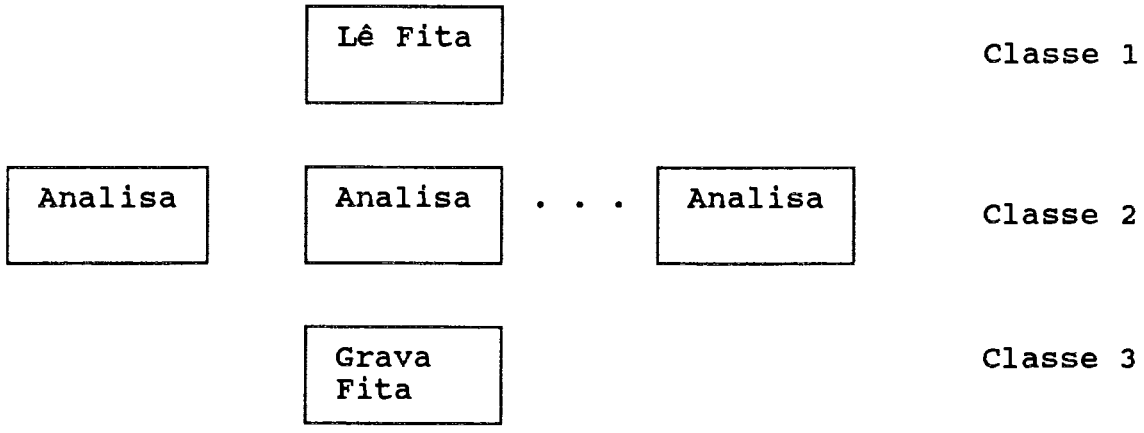

Figura V.2 - Processos da Reconstrução

O fluxo de dados, nesse exemplo, ocorre da seguinte forma: o processo da classe 1 lê dados de fita e os passa para os processos da classe 2. Esses, por sua vez, analisam os dados, que após analisados são recolhidos pelo processo da classe 3 que grava os dados analisados na fita de saída.

A implementação realizada restringe os processos de uma classe a serem executados num mesmo tipo de CPU, onde o tipo de CPU pode ser ACP, VAX rodando ULTRIX, VAX rodando VMS, SUN ou outro computador que suporte o CPS.

Essa restrição, no entanto, foi superada pela criação do conceito de *conjunto de classes*. Havendo necessidade de executar processos que desempenham a mesma função em

computadores diferentes, estes processos devem ser colocados em classes diferentes e agrupados em um *conjunto de classes*. Um *conjunto de classes* pode conter tantas *classes* quantas forem necessárias. A figura V.3 mostra a distribuição das classes no problema de reconstrução, usando-se duas classes para fazer a análise.

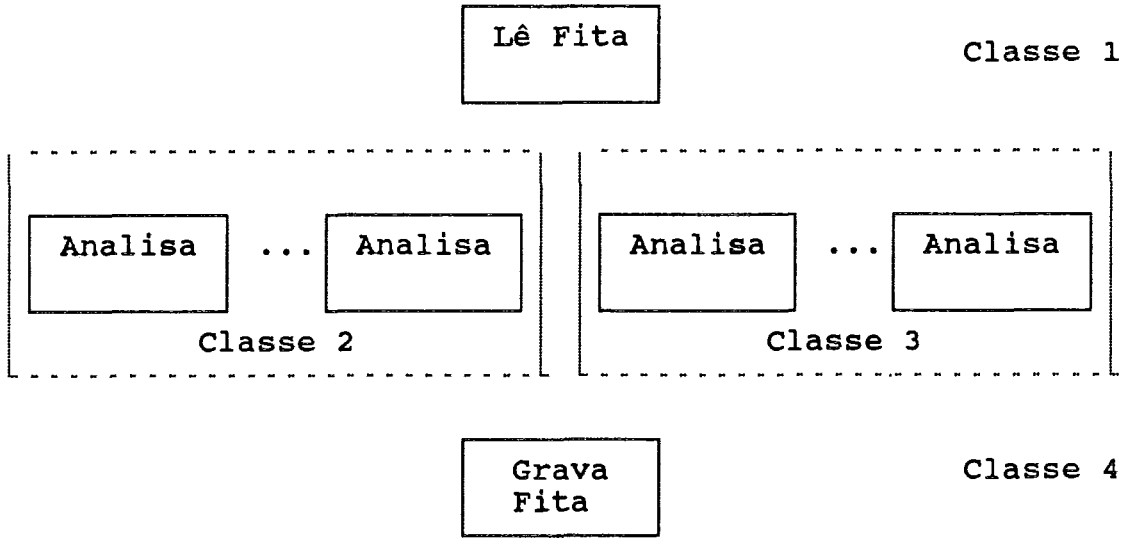

Figura V.3 - Processos da Reconstrução

Para cada classe ou conjunto de classes deve ser escrito um programa que execute as tarefas designadas. Esses programas são compilados e concatenados com a biblioteca de rotinas do CPS.

O CPS fornece também algumas ferramentas úteis para a depuração dos programas. Falaremos sobre esta questão mais adiante.

V.2 - PRIMITIVAS DE TROCA DE MENSAGENS

Todas as funções do CPS que requerem interação entre

processos são implementadas através de troca de mensagens. Essas trocas de mensagens são, no entanto, encapsuladas por rotinas de mais alto nível como chamada remota de procedimentos, rotinas de manipulação de filas e outras. Assim, essas trocas de mensagens são tranparentes ao usuário, que pode desenvolver toda a sua aplicação sem tomar conhecimento delas. No entanto, aplicações mais específicas e especializadas podem utilizar trocas de mensagens de forma explícita para implementar funções não previstas no CPS.

As duas primitivas básicas de troca de mensagem são *envia* e *recebe*.

Uma mensagem pode ser enviada para um processo específico ou para um grupo de processos, composto por processos de uma *classe* ou *conjunto de classes*. O tamanho da mensagem é limitado a 2048 bytes.

O endereçamento é feito indicando o número do processo ou um valor que indica um *conjunto de classes*. O número do processo é um valor inteiro que indentifica um processo da aplicação. Para indicar um *conjunto de classes* é usado o valor retornado por uma função especial, *acp_set_of_processes*, que será descrito no capítulo VI.

As mensagens possuem tipos que são identificados por números inteiros entre 1 e 31. Os tipos de 21 a 31 são reservados para o sistema e são usados para implementar chamadas remotas de procedimentos, funções de filas e outras. Os tipos de 1 a 20 são utilizados pelo usuário que desejar usar diretamente as primitivas de troca de

mensagens e o significado de cada tipo é determinado pelo usuário. As mensagens do sistema (21 a 31) são tratadas pelo CPS de forma invisível para a aplicação. A única mensagem de sistema que o usuário pode receber é a mensagem de chamada remota de procedimentos, conforme será visto mais adiante.

Quando é feita uma chamada à função de recepção de mensagens, é especificado o conjunto de tipos de mensagens que deve ser recebidas. Esse conjunto pode ser composto de apenas um tipo de mensagem, de todos os tipos, ou de um subconjunto qualquer dos tipos válidos.

O envio de mensagens é não-bloqueante, ou seja, o processo que enviou não fica esperando que o processo destinatário chame a função de *recebe*. No entanto, se o "buffer" de recepção do processo destinatário estiver cheio, o processo que enviou é bloqueado até que o processo destinatário receba pelo menos uma mensagem e libere uma entrada no "buffer".

A função *recebe* é bloqueante. Esse sincronismo torna mais simples a implementação da maioria dos programas. No entanto, para algumas aplicações é conveniente que, caso não exista mensagem, o processo não seja bloqueado, mas sim receba um aviso de que não há mensagens. Para estes casos, o CPS fornece uma função que permite verificar a existência ou não de mensagens. Assim pode ser conseguido o efeito de uma função não-bloqueante. Para isso, basta verificar a existência de mensagens e só chamar a função *recebe* se houver mensagem.

V.3 - SINCRONISMO DE PROCESSOS

O CPS oferece várias funções para o sincronismo entre os processos. Essas funções são desenvolvidas com o uso de troca de mensagens e as funções que envolvem o sincronismo de mais de dois processos utilizam serviços fornecidos pelo "Job Manager".

As ferramentas de sincronismo do CPS são *filas*, *chamadas remotas de procedimentos* e *pontos de sincronismo*.

V.3.1 - FILAS

O uso de *filas* no CPS está associado ao conceito de estados de processos. Com o uso deste mecanismo, é possível que um processo seja declarado em um determinado estado ou que selecione outro que esteja num estado específico. Junto com um processo, dados também podem ser colocados na *fila* sendo ambos retirados simultaneamente. O significado dos estados, bem como dos dados existentes nas *filas*, não é interpretado pelo CPS e sim pela aplicação, que desta forma não precisa obedecer a um modelo de estados pré-estabelecidos e pode definí-los da maneira mais natural ao programa.

É bom ressaltar que esses estados não estão associados aos estados que um sistema operacional costuma atribuir a um processo e sim às etapas pela qual uma aplicação deve passar na execução de um algoritmo.

Voltando ao exemplo da reconstrução, os processos da classe 2, que fazem a análise dos dados, podem estar em três estados: *executando, esperando dados* ou *com dados*

disponíveis. Quando esses processos estão no estado *esperando dados*, eles se colocam numa *fila* correspondente a esse estado. O processo da classe 1, que lê dados da fita, retira um processo da *fila* de *esperando dados* e lhe envia os dados lidos. O processo da classe 2 passa então para o estado *executando*. A esse estado não é necessário associar nenhuma *fila*, pois finda a análise, o processo já se coloca na *fila* associada ao estado *com dados disponíveis*. O processo da classe 3, por sua vez, retira os processos dessa *fila* para lhe pedir os dados analisados e gravá-los na fita. Uma vez que tenha recebido os dados analisados, o processo da classe 2 é colocado de volta na *fila* dos processos *esperando dados* e o ciclo se repete até o fim da análise de todos os dados.

Existem dois tipos de função para retirar um processo de uma *fila*: uma função bloqueante e uma não-bloqueante. Quando é usada a função bloqueante, o processo que chamou fica bloqueado até que haja um processo na *fila* desejada. No caso da chamada não-bloqueante, se não houver nenhum processo na *fila*, a função retorna com um código de retorno indicando essa condição.

Além das funções de colocar e retirar processos das *filas*, existe uma função que espera que a *fila* esteja cheia e outra função que espera que a *fila* esteja vazia. Essas funções são usadas, em conjunto com uma entrada especial que indica *fim-de-fila*, para fazer a passagem para uma nova fase do trabalho ou para terminar o trabalho.

A entrada especial *fim-de-fila* é colocada em uma *fila* por qualquer processo, e após essa entrada, nenhum processo pode mais ser colocado nessa *fila*. A função de retirar processo da *fila* retorna um valor especial quando ao invés de um processo encontra a marca de *fim-de-fila*.

Vamos retornar ao nosso exemplo para ilustrar o uso da função de esperar uma *fila* ficar cheia e da marca de *fim-de-fila*.

Quando o processo que faz a leitura da fita chega ao final desta, ainda podem existir processos na classe 2 que estão analisando dados e assim, o processo da classe 3 ainda terá dados analisados para gravar. Esse processo da classe 3 tem de saber quando ele gravou o último evento analisado para poder executar os procedimentos de finalização e terminar o processamento.

O processo da classe 1, ao terminar a fita, espera que a *fila* de processos esperando dados esteja cheia. Quando isso ocorrer é sinal que todos os dados já foram analisados e que o processo da classe 3 já recolheu todos os eventos analisados. O processo da classe 3 tem de ser avisado então que não haverá mais dados para serem gravados. Para isso, o processo da classe 1 coloca a marca de *fim-de-fila* na *fila* de processos com dados disponíveis. Ao retirar o próximo elemento dessa *fila*, o processo da classe 3 receberá a marca de *fim-de-fila* e fará os procedimentos de finalização e terminará o trabalho.

V.3.2 - CHAMADAS REMOTAS DE PROCEDIMENTOS

Outra ferramenta disponível no CPS é a *Chamada Remota de Procedimentos*, também chamado de RPC, do inglês "Remote Procedure Call".

A idéia da *Chamada Remota de Procedimentos* já foi discutida no capítulo II. Aqui será apresentado o uso de *Chamadas remotas de procedimentos* no CPS. A *Chamada Remota de Procedimentos*, como é classicamente concebida, é uma operação síncrona. O processo *cliente*, após chamar a rotina remota, é bloqueado até o término da execução, por parte do processo *servidor*, da rotina desejada.

No CPS, para permitir um maior grau de paralelismo, foram criados, além da *chamada remota* tradicional, dois tipos de chamadas assíncronas: a chamada não-bloqueante e a chamada não-bloqueante com enfileiramento.

Na chamada não-bloqueante com enfileiramento, o processo *cliente* especifica uma *fila* onde o processo *servidor* se coloca, junto com os resultados do procedimento executado, ao término da execução do mesmo. O processo *cliente* não é bloqueado e os resultados do procedimento remoto podem ser retirados da *fila* mais tarde pelo processo *cliente* ou outro processo qualquer.

Na chamada não-bloqueante simples, o processo *cliente* também não é bloqueado mas cabe ao processo *servidor* definir se entrará em uma *fila* e em qual *fila*, se for o caso.

No CPS, um processo se torna um *servidor* de chamadas remotas ao evocar uma função específica para esse fim. A

partir desse instante, ele passa a aceitar pedidos de outros processos. Os pedidos são atendidos na ordem em que são recebidos.

Uma outra maneira de um processo funcionar como servidor é usando a primitiva *recebe* que já foi discutida. Um processo que não está no modo servidor pode receber uma mensagem que seja uma requisição para executar um procedimento. Nesse caso ele se torna *servidor* para executar aquele procedimento e depois sai do modo servidor.

Antes de um processo se tornar *servidor* ele precisa declarar quais são os procedimentos que ele pode executar como tal.

V.3.3 - PONTOS DE SINCRONISMO

As funções de chamada remota de procedimentos, de filas e mesmo a troca explícita de mensagens são mecanismos de sincronização de processos. No entanto, às vezes, é útil ter formas mais diretas de sincronizar processos. Para isso existe no CPS o *ponto de sincronismo*.

O ponto do programa onde um processo deve esperar pelo outro ou pelos outros é chamado *ponto de sincronismo*. Esse ponto é estabelecido com uma chamada à função de sincronismo. O ponto de sincronismo no CPS tem a funcionalidade do *rendez-vous*, discutido no capítulo II, mas tem duas diferenças básicas:

1. permite que mais de um processo se sincronize ao invés de apenas um par como ocorre no *rendez-vous*;

2. existe um processo, o "Job Manager", gerenciando esse ponto de sincronismo.

Uma aplicação pode ter vários *pontos de sincronismo*. Na chamada à função de sincronismo, é fornecido como parâmetro um identificador do *ponto de sincronismo* desejado e o conjunto de classes que fazem parte desse ponto de sincronismo. Maiores detalhes sobre esses parâmetros podem ser vistos no capítulo VI.

Podemos novamente recorrer ao problema da reconstrução para exemplificar o uso de pontos de sincronismo. O processo que lê os dados, inicialmente lê parâmetros que devem ser passados para todos os processos que vão fazer a análise. Para poder fazer isso, o processo da classe 1 precisa saber que os processos da classe 2 já foram inicializados e estão prontos para receber os parâmetros de inicialização. Uma maneira de se conseguir isso é definir, no processo da classe 1, um ponto de sincronismo imediatamente antes de enviar os dados e nos processos da classe 2 logo após eles estarem prontos para receber os dados. Assim, se o processo da classe 1 atingir o *ponto de sincronismo* antes dos processos da classe 2, ele ficará bloqueado até que todos os processos da classe 2 tenham atingido o referido *ponto de sincronismo*.

V.4 - TRANSFERÊNCIA DE DADOS

Filas, *Chamada Remota de Procedimentos* e as primitivas de troca de mensagem, além de servirem para

sincronizar processos, são ferramentas que permitem a troca de dados entre processos. Além dessas funções, o CPS fornece rotinas que permitem a transferência de grandes seqüências de dados, sem haver mecanismos de sincronismo envolvidos. Essas rotinas permitem que blocos de dados, tais como vetores ou estruturas, possam ser transferidos de um processo para outro.

Na tranferência de dados, há sempre um processo ativo e um processo passivo. O processo ativo toma a iniciativa de transferir os dados de sua memória para a memória do outro ou vice-versa. O processo passivo precisa declarar os blocos de memória que devem ser usados, para a recepção ou transmissão de forma a tornar esses blocos disponíveis para os outros processos.

As rotinas de transferência de dados são implementadas usando, de forma transparente para o usuário, troca de mensagens. No entanto, se os processos envolvidos na transferência de dados estiverem executando em nós do ACP, essas rotinas podem ser otimizadas para utilizar as facilidades do "hardware" e implementar a transferência de dados da memória de um processo para a memória de outro processo. Com isso pode ser eliminado o "overhead" do protocolo das primitivas de troca de mensagens.

V.5 - CONVERSÃO DE DADOS

Quando uma aplicação possui processos sendo executados em diferentes CPUs e a representação de dados

numéricos é diferente em cada processador, é necessário que os dados sejam convertidos ao passarem de um computador para outro.

Se o número for um inteiro de 32 bits, a conversão é feita automaticamente pelo sistema de troca de mensagens. Se, no entanto, forem utilizados inteiros de 16 bits, números reais ou caracteres, deve ser feita uma conversão ou pelo processo de origem ou pelo processo de destino dos dados. O CPS fornece uma rotina que faz a conversão necessária.

V.6 - O "JOB MANAGER"

As aplicações do CPS têm um processo especial que é chamado de "Job Manager" ou JM. Esse processo inicializa e termina a execução da aplicação e de todos os processos que a compõem. O JM também é responsável por serviços centralizados de controle e sincronismo.

As principais funções do JM podem ser resumidas nos seguintes itens:

- inicialização dos processos do usuário que compõem a aplicação em um ou mais processadores;
- geração de um arquivo com registros da execução da aplicação ("log file");
- fornecimento de serviços de sincronismo, gerenciando *filas* e *pontos de sincronismo*;
- monitoração dos processos para verificar o mal funcionamento ou morte de um processo;
- tratamento de erros;

- redirecionamento de entrada e saída de terminal;
- finalização dos processos.

Os parâmetros para o JM são fornecidos através de um arquivo chamado "Job Descriptor File" ou JDF. Os parâmetros no JDF indicam, entre outras coisas, os programas de cada classe, onde os processos serão executados, quantos processos haverá em cada classe. No apêndice A o JDF é descrito com mais detalhes.

V.6.1 - INICIALIZAÇÃO E FINALIZAÇÃO DOS PROCESSOS

A inicialização dos processos é executada em quatro passos. Primeiro o JM lê o arquivo JDF e determina quais processos devem ser executados e onde. Depois aloca os processadores necessários. Após isso, os processos são criados. A criação dos processos é feita pelo sistema operacional, atendendo à requisição do JM. Finalmente o JM verifica se todos os processos começaram corretamente.

Quando um processo é inicializado, o JM lhe passa as informações necessárias para endereçar o próprio JM e todos os outros processos daquela aplicação.

Um ou mais processos podem ser iniciados manualmente pelo usuário. Isso é útil para a depuração dos programas, que nesses casos podem ser executados debaixo de um depurador. O usuário indica no arquivo JDF os processos que devem ser inicializados manualmente e o JM imprime uma mensagem na tela indicando que o processo deve ser incializado e aguarda que isso seja feito.

O JM determina que a aplicação chegou ao fim quando algum processo chama a função de fim de processamento. Outra condição em que o JM termina o processamento ocorre quando ele detecta que o número de processos vivos não é suficiente para a continuação do processo. No arquivo JDF é indicado o número mínimo de processos que devem estar executando em cada classe. Finalmente, a terceira forma de se terminar a execução da aplicação é teclar CONTROL-C no terminal do JM.

Para terminar o processamento, o JM envia uma mensagem a todos os processos indicando o fim do processamento. Após fazer isso, o JM ainda verifica, usando recursos do Sistema Operacional, se todos os processos realmente terminaram a sua execução. Caso algum processo ainda permaneça vivo,o usuário é notificado para que possa tomar as providências necessárias. Isso é importante para evitar que processos "fantasmas" fiquem rodando indefinidamente no caso de alguma falha que cause o não recebimento da mensagem de fim de processo.

V.6.2 - REGISTRO DE ATIVIDADES

O JM fornece informações sobre a execução da aplicação tanto no terminal como em um "log file". A quantidade de informação gerada pode ser controlada, com indicação do nível de informação que se deseja. A quantidade de informação é selecionada separadamente para o terminal e para o arquivo. É possível selecionar quatro níveis (de 0 a 3) de informação.

No nível 0, nenhuma informação é registrada. No nível 1, é fornecido um nível mínimo de informação, que inclui:

- indicação dos processos inicializados;
- indicação de qualquer erro fatal e a providência tomada;
- indicação de quando e porque a aplicação terminou.

O nível 2, que é o nível "default", além das informações do nível 1, mostra as mensagens para o terminal geradas pelos processos.

E finalmente, o nível 3 indica, além das informações do nível 2, todas as mensagens geradas pelo JM para os processos do usuário e as mensagens dos processos para o JM. Nesse nível também são indicadas estatísticas das rotinas de gerenciamento de mensagens ("message handler"). As informações do nível 3 em geral são usadas para a depuração dos programas.

V.6.3 - SERVIÇOS DE SINCRONISMO

Os serviços de sincronismo são executados de forma centralizada pelo JM. Esses serviços são o gerenciamento de pontos de sincronismo e o gerenciamento de filas.

Os pontos de sincronismo são de gerenciamento muito simples. Os processos, ao chamar a função de sincronismo, enviam uma mensagem ao JM, indicando o conjunto de processos envolvidos e a identificação do ponto de sincronismo. Após o envio da mensagem o processo vai dormir. O JM, guarda uma lista dos processos que já

chegaram ao ponto de sincronismo. Quando todos os processos tiverem chegado a esse ponto, o JM manda uma mensagem a cada um dos processos para acordá-los. As filas também são gerenciadas pelo JM. Um processo, antes de ser colocado numa fila, deve declarar essa fila. A declaração, além de indicar que o processo pode ser colocado numa fila, informa ao JM o número e tamanho dos argumentos.

Quando ocorre uma requisição para colocar um processo numa fila, o JM verifica se aquele processo a tinha declarado. Em caso negativo, o JM considera este erro fatal e mata o processo que fez a requisição. Em caso positivo, o processo especificado é colocado na fila desejada. Ao colocar um processo na fila, o JM verifica se ela encheu e se há algum processo esperando por essa condição. Os processos que estiverem esperando por esta condição recebem uma mensagem avisando-os que a fila está cheia e dessa forma eles podem prosseguir a sua execução. Se a fila estiver vazia, verifica-se se há algum processo esperando para retirar algum processo dela. Nesse caso a requisição é atendida e o processo que estava esperando é acordado com o envio de uma mensagem.

Quando um processo pede para retirar algum processo da fila, duas condições podem ocorrer. A primeira condição é que exista algum processo na fila. Nesse caso, a requisição é prontamente atendida e verifica-se se a fila ficou vazia e ocorre o mesmo procedimento da fila cheia. A segunda condição é que não exista nenhum processo na fila. Nesse caso, o processo que fez a requisição vai dormir até

que possa ser atendido.

V.6.4 - MONITORAÇÃO DOS PROCESSOS

O "Job Manager" periodicamente manda uma mensagem para cada processo, pedindo informações de "status" desses processos. Quando um processo recebe essa mensagem, ele envia uma resposta contendo essas informações. Caso o JM não receba uma resposta, ele assume que o processo está morto. Com isso, a morte prematura ou o mal funcionamento de um processo pode ser rapidamente detectado. Além disso, as informações obtidas com a mensagem de "status" podem ser usadas para a depuração dos programas. As informações fornecidas na mensagem de status são as seguintes:

- tempo de CPU de cada processo;
- número de mensagens transmitidas e recebidas;
- número de chamadas remotas a procedimentos;
- número de bytes transferidos;
- estatísticas do sistema de mensagens (de interesse apenas na implementação e calibração do mesmo).

Além dessas informações, é enviado o conteúdo de um vetor de variáveis cujos valores são preenchidos pelo usuário.

Todas as informações acima descritas podem ser continuamente observadas pelo usuário com o uso do programa Monitor que será descrito mais tarde.

V.6.5 - REDIRECIONAMENTO DE ENTRADA E SAÍDA PARA TERMINAL

Os processos da aplicação inicializados pelo "Job

Manager" não estão associados a um terminal. Uma exceção é o caso do processo ter sido inicializado manualmente, como descrito no item V.6.1.

Quando um processo inicializado automaticamente pelo JM faz uma operação de leitura ou escrita para o terminal, essa operação é interceptada pelas rotinas do CPS e transferida para o JM. Se a operação for de leitura, o JM pede ao usuário para digitar o dado no terminal do JM e envia o resultado para o processo requisitante. Dessa forma, os processos da aplicação podem fazer uso de um terminal virtual tratado pelo JM. Essa capacidade só está implementada nos nós do sistema ACP. Uma outra maneira de se imprimir textos no terminal do JM é usando a rotina *acp_log* que será descrita no capítulo VII.

V.7 - DEPURAÇÃO DE APLICAÇÕES

A depuração de programas envolvidos em processos cooperativos é muito mais difícil do que em programas sequenciais. A principal razão para isso é o não determinismo que pode existir na execução dos processos, devido ao fato de nada poder ser assumido sobre as velocidades relativas dos processos.

Por isso, é necessário prover mecanismos adequados para depuração de programas paralelos.

O CPS, além de possibilitar o uso dos depuradores convencionais de processos seqüenciais, conforme explicado no item V.6.1, possibilita outras formas de depuração que levam em conta a natureza concorrente dos processos. Um

desses mecanismos é o registro de atividades, descrito no item V.6.2. O outro mecanismo é o programa monitor que será descrito no capítulo VII.

V.8 - INTERFACE DO CPS COM O SISTEMA OPERACIONAL

O software do CPS é estruturado em camadas, conforme ilustrado na figura V.4. No topo da camada estão as rotinas de alto nível do CPS. Essas rotinas são as rotinas de ponto de sincronismo, chamada remota de procedimentos, filas, transferência de dados e outras. Elas utilizam as rotinas de envio e recepção de mensagens. Estas últimas. por sua vez, utilizam os serviços do "message handler" que interage com o sistema operacional para implementar a comunicação entre os processos.

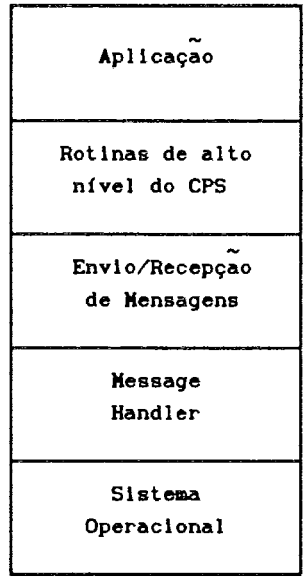

Figura V.4 - Estruturação do CPS

O "message handler" apresenta para as rotinas de

envio e recepção de mensagens, uma interface independente do sistema operacional. Com isso, para adaptar o CPS a um novo sistema, basta alterar as rotinas do message handler.

No caso do sistema operacional UNIX, usado na segunda geração do sistema ACP, a comunicação entre os processos é feita utilizando o UDP ("User Datagram Protocol") do protocolo TCP/IP [40]. O UNIX possui chamadas de sistema [41] que permitem a utilização do TCP/IP. Esse protocolo, além de ser um padrão nos sistemas operacionais compatíveis com o UNIX, possue implementação em quase todos os sistemas operacionais comerciais e, graças a isso, o CPS permite a comunicação entre processos executando em diferentes máquinas.

Existe uma diferença entre o endereçamento de processos usados pelo UDP e usados pelas primitivas de troca de mensagem do CPS. NO UDP, é necessário identificar a máquina e o processo desejado. No CPS, o endereço de um processo consiste num número inteiro, sendo transparente a CPU onde o processo é executado. O "message handler" faz a conversão desses endereços, utilizando tabelas que são preenchidas pelo Job Manager quando os processos são criados. O "message handler" prevê também a utilização de outros protocolos de comunicação. No caso da segunda geração do sistema ACP, está prevista, mas não implementada ainda, um protocolo, denominado ADP ("ACP Datagram Protocol"), que aproveita melhor as características do hardware e possue um protocolo mais simples do que o UDP. Com isso pretende-se diminuir o

"overhead" da comunicação. Cumpre ressaltar que os diferentes protocolos não são mutuamente exclusivos. O ADP será utilizado para a comunicação entre dois nós ACP e para um nó ACP se comunicar com outro tipo de máquina será utilizado o UDP.

Além da implementação da comunicação entre processos, o sistema operacional é utilizado pelo "Job Manager" para inicializar os processos de uma aplicação.

No caso do sistema operacional UNIX, existe um processo do sistema, chamado "rexecd" que tem como função inicializar processos locais a pedido de processos também locais ou de procesos remotos. A implementação em sistemas diferentes do UNIX implica o desenvolvimento de um programa compatível, a nível de protocolo, com o "rexecd". Isso foi feito para o VMS.

C A P Í T U L O VI

DESCRIÇÃO DA BIBLIOTECA DE ROTINAS

Nesse capítulo descreveremos a biblioteca de rotinas do CPS. Para efeito de apresentação, classificaremos as rotinas nos seguintes grupos:

- Inicialização e finalização
- Chamada Remota de Procedimentos
- Transferência de dados
- Filas
- Sincronismo
- Troca Explícita de Mensagens
- Conversão de Dados
- Tratamento de Erros
- Rotinas Auxiliares

Devemos resaltar que todos os parâmetros das rotinas do CPS são passados por endereço a fim de permitir que linguagens como o FORTRAN possam usar o CPS. Os parâmetros às vezes podem assumir um valor inteiro constante que são representados por nomes mnemônicos começados por ACP$. Esses valores são definidos num arquivo de inclusão chamado acp_user.h para programas em C e acp_user.inc para programas em FORTRAN.

Na apresentação das funções do CPS, os parâmetros, quando existirem, aparecerão, junto com a sua explicação, logo após o nome da rotina.

Uma descrição mais detalhada das rotinas pode ser vista em [42].

VI.1 - ROTINAS DE INICIALIZAÇÃO E FINALIZAÇÃO

a) *acp_init*

Esta rotina deve obrigatoriamente ser chamada no início de todo programa com CPS. Nela, o processo comunica ao JM que foi inicializado e o JM fornece ao processo uma tabela com o endereçamento de todos os outros processos da aplicação.

b) *acp_stop_process*

Essa rotina deve ser chamada quando um processo termina a sua execução. O processo então vai dormir a espera que toda a aplicação termine.

c) *acp_stop_job*

Essa rotina indica o fim de toda a aplicação. Pode ser chamada por qualquer processo. Pelo menos um processo deve chamar essa aplicação para que o JM dê a aplicação por terminada e comunique a todos os processos o fim da aplicação, inclusive aos que chamaram a rotina *acp_stop_process*.

VI.2 - CHAMADA REMOTA DE PROCEDIMENTOS

a) *acp_declare_subroutine*

rotina - endereço da rotina que está sendo declarada.

número_da_rotina - é o número de identificação da rotina.

número_de_argumentos - indica o número de argumentos da rotina.

tamanho_arg_1 - ... - *tamanho_arg_n* - lista com o tamanho

em bytes de cada um dos argumentos da rotina. O tamamho dessa lista é igual a *número_de _argumentos*.

Um processo servidor deve declarar as rotinas que ele pode executar e atribuir uma identificação para estas rotina. A identificação é um número inteiro definido pelo usuário. Nessa rotina também é definido o número de argumentos e o tamanho dos mesmos.

b) *acp_service_calls*

Coloca o processo num modo de servidor de rotinas. Essa rotina nunca retorna, uma vez que o processo fica dedicado a servir chamadas remotas de procedimentos.

c) *acp_call*

processo - é o número do processo chamado. Pode ser um conjunto de processos obtido com a função *acp_set_of_processes* (descrita em VI.9).

tipo - define o tipo da chamada remota, se com espera, sem espera ou sem espera com enfileiramento. Os valores são, respectivamente, ACP$WAIT, ACP$NOWAIT ou o número da fila desejada.

numero_rotina - identifica qual rotina deve ser executada pelo(s) processo(s) servidor(es).

arg_1 ... arg_n - são os argumentos passados para a sub-rotina remota. O número e o tamanho desses argumentos devem coincidir com a declaração da rotina feita pelo

servidor.

Faz uma requisição para um outro processo executar uma rotina. Na realidade, a requisição pode ser feita para um conjunto de processos.

d) *acp_change_return_queue*

nova_fila - é um número inteiro que especifica a nova fila.

Essa rotina é chamada por um processo servidor para alterar a fila onde o processo se colocará ao fim da execução do procedimento invocado. Só se aplica quando é usada a chamada sem espera com enfileiramento.

e) *acp_service_one_call*

mensagem - é um ponteiro para a mensagem com a chamada remota de procedimentos.

tamanho_mensagem - como o nome já diz, é o tamanho da mensagem recebida.

processo - é o número do processo que fez a chamada remota de procedimentos.

Essa função é usada por aplicações mais sofisticadas, nas quais o servidor não funciona de maneira dedicada. Nesse caso, o servidor pode receber mensagens diversas e se a mensagem for uma chamada remota de procedimento, chama *acp_service_one_call* para atender essa requisição e retornar. Uma chamada remota de procedimentos é identificada pelo tipo da mensagem, conforme será visto em **VI.6.**

VI.3 TRANSFERÊNCIA DE DADOS

a) *acp_declare_block*

endereço - posição de memória onde começa o bloco.

tamanho - tamanho em bytes do bloco de memória.

identificação - número inteiro que define o bloco de memória. É usado pelos processos que vão fazer acesso a esse bloco para identificá-lo.

Essa rotina define um bloco de memória onde outros processos podem transferir dados para essa posição ou transferir dados dessa posição.

b) *acp_send*

processo - número inteiro que identifica o processo destino ou valor retornado por *acp_set_of_processes* para identificar um conjunto de processos.

endereço - ponteiro para a posição da memória local de onde os dados são transferidos.

tamanho - número de bytes a serem transferidos.

bloco - identificação do bloco no processo destinatário.

offset - índice que indica a posição inicial da transferência no bloco de destino.

Essa função envia dados para um bloco de memória de outro processo ou um conjunto de processos.

c) *acp_get*

processo - número inteiro que identifica o processo remoto

ou valor retornado por *acp_set_of_processes* para identificar um conjunto de processos.

endereço - ponteiro para a posição da memória local para onde os dados são transferidos.

tamanho - número de bytes a serem transferidos.

bloco - identificação do bloco no processo remoto.

offset - índice que indica a posição inicial da transferência no bloco de origem.

Similar a *acp_send* só que a transferência nesse caso se faz do processo remoto para o processo local. Outra diferença é que se for especificado um conjunto de processos ao invés de um processo particular, os dados dos processos remotos são somados, como inteiros de 32 bits, para formar os dados locais.

VI.4 - FILAS

a) *acp_declare_queue*

número_fila - Número inteiro que identifica uma fila.

número_de_argumentos - indica o número de argumentos da fila.

tamanho_arg_1 - ... - *tamanho_arg_n* - lista com o tamanho em bytes de cada um dos arguementos da fila. O tamamho dessa lista é igual a *número_de _argumentos*.

Para que um processo possa ser colocado numa fila ele deve declarar essa fila com a função *acp_declare_queue*.

b) *acp_queue_process*

processo - é o número do processo a ser colocado na fila. Pode ser um conjunto de processos, obtido com a função *acp_set_of_processes* (descrita em VI.9).

fila - é o número inteiro que identifica a fila.

arg_1 ... arg_n - são os argumentos colocados na fila.

Essa função coloca um processo com os seus parâmetros na fila.

c) *acp_dequeue_process*

processo - a função *acp_dequeue_process* retorna em *processo* o número do processo retirado da fila. Pode ser retornado o valor especial ACP$END_OF_QUEUE que identifica o fim-de-fila conforme descrito no capítulo V.

fila - número inteiro que identifica a fila desejada.

arg_1 ... arg_n - recebem os argumentos retirados da fila.

Retira um processo da fila especificada. É uma função bloqueante, ou seja, se a fila estiver vazia o processo é suspenso até que algum outro processo seja colocado na fila.

d) *acp_dequeue_if_possible*

processo - a função *acp_dequeue_process* retorna em *processo* o número do processo retirado da fila. Pode ser retornado o valor especial ACP$END_OF_QUEUE que identifica o fim-de-fila conforme descrito no capítulo V.

fila - número inteiro que identifica a fila desejada.

arg_1 ... arg_n - recebem os argumentos retirados da fila.

É a versão não-bloqueante da *acp_dequeue_process*. Se não houver nenhum processo na fila, retorna em *processo* o valor especial ACP$EMPTY.

e) *acp_wait_queue*

fila - número inteiro que identifica a fila desejada.

estado - pode ser ACP$FULL ou ACP$EMPTY para indicar que se deseja esperar que a fila esteja cheia ou vazia, respectivamente.

Espera até que a fila se encontre no estado especificado que pode ser cheia ou vazia.

VI.5 - SINCRONISMO

a) *acp_sync*

processos - indica o conjunto de processos que devem participar do *rendez-vous*. Pode ser um conjunto de classes obtido com a função *acp_set_of_processes* ou o valor especial ACP$ALL_PROCESSES que indica todos os processos.

sinc_numero - é um número inteiro que indentifica o ponto de sincronismo.

Espera até que todos os processos especificados chamem *acp_sync* para o mesmo ponto de sincronismo.

VI.6 - TROCA EXPLÍCITA DE MENSAGENS

a) *acp_transmit_message*

destino - número que identifica o processo destino ou um conjunto de processos obtido com a função *acp_set_of_processes*.

tipo - número inteiro entre 1 e 20 que identifica o tipo da mensagem.

mensagem - ponteiro para a mensagem a ser transmitida.

tamanho - tamanho em bytes da mensagem a ser transmitida.

Essa função permite que uma mensagem de até 2048 bytes seja enviada a um outro processo ou conjunto de processos.

A toda mensagem é associada um *tipo* que é representado por um número inteiro entre 1 e 20. O significado do tipo é definido pela aplicação.

b) *acp_check_message*

tipos - Especifica os tipos de mensagens que se deseja verificar. Deve ser utilizado o valor retornado pela rotina *acp_set_of_messages_types* que será descrita mais adiante.

Essa função verifica se existe alguma mensagem dos tipos especificados para ser recebida. Retorna o valor ACP$OK se houver mensagem ou ACP$NO_MESSAGES_AVAILABLE se não houver nenhuma mensagem.

c) *acp_receive_mnessage*

tipos_desejados - define o conjunto de tipos de mensagens

que se deseja receber. Deve ser usado um valor retornado por *acp_set_of_messages_types*.

processo - essa função retorna em *processo* o número do processo que enviou a mensagem.

tipo_recebido - recebe o tipo da mensagem recebida.

mensagem - ponteiro para o "buffer" onde a mensagen recebida deve ser colocada.

tamanho - é preenchido com o tamanho, em bytes, da mensagem recebida.

Essa rotina é usada para receber uma mensagem enviada por outro processo.

d) *acp_set_of_messages_types*

num_tipos - indica o número de tipos que serão especificados.

tipo1 ... tipon - especifica cada um dos tipos que formarão o conjunto.

Essa função retorna um valor inteiro que especifica um conjunto de tipos de mensagens e é usado pelas rotinas *acp_receive_message* e *acp_check_message*.

VI.7 - CONVERSÃO DE DADOS

a) *acp_convert*

dado_original - é um ponteiro para os dados ainda não convertidos.

dado_convertido- é um ponteiro para a localização do vetor onde devem ser colocados os dados

convertidos. *dado_original* e *dado_convertido* podem apontar para a mesma posição de memória.

tamanho - indica o tamanho em bytes dos dados a serem convertidos.

tipo - indica o tipo de dado a ser convertido. Pode ter os valores ACP$INTEGER_2, ACP$REAL_4, ACP$REAL_8 ou ACP$CHARACTER.

origem - indica a classe do processo que originou os dados.

destino - indica a classe do processo de destino dos dados.

Essa rotina é usada para converter dados números quando se usa computadores com diferentes representações numéricas.

VI.8 - TRATAMENTO DE ERROS

a) *acp_fatal_job_error*

texto - "string" de caracteres que indica o erro ocorrido. Pode ter no máximo 2000 caracteres.

Essa rotina causa o término da aplicação após um texto, definido pelo usuário, ter sido impresso na tela e registrado no "log file". A rotina deve ser chamada quando for detectado um erro grave o suficiente para causar o fim do processamento.

b) *acp_fatal_process_error*

processos - identifica o processo ou um conjunto de

processos a serem declarados mortos. Pode ser usado o valor especial ACP$THIS_PROCESS para identificar o próprio processo. Para identificar um conjunto de processos usa-se o valor retornado por *acp_set_of_processes*.

texto - é o texto que é o JM deve escrever na tela e salvar no "log file".

Essa rotina é chamada para declarar que um processo, ou um conjunto de processos, apresenta um erro fatal. A chamada a essa função faz com que o "Job Manager" retire os processos de qualquer fila em que por ventura estejam e considere o processo morto. O texto definido pelo usuário é escrito na tela do JM e gravado no "log file".

c) *acp_handle_error*

ação - especifica que ação tomar quando o erro não é necessariamente local.

texto - texto que deve ser impresso pelo JM e registrado no "log file".

Essa rotina pode ser usada quando um código de erro é retornado por uma função do CPS e o usuário não deseja lidar com este erro, deixando isso a cargo do CPS.

Alguns erros são claramente erros no processo local. Outros no entanto podem ser devido a uma falha num processo remoto. Nesse caso, em geral, não é possível determinar se o erro ocorreu no processo local ou no processo remoto. Quando isso ocorre, o usuário deve especificar se o erro deve ser considerado erro no

processo local, erro no processo remoto, ou deixar o "Job Manager" decidir. Para isso, atribui a *ação* os valores especiais ACP$KILL_LOCAL_PROCESS, ACP$KILL_REMOTE_PROCESS ou ACP$REPORT_TO_JOB_MANAGER, respectivamente.

O usuário também deve, ao chamar essa rotina, fornecer um texto que será impresso na tela do JM e registrado no "log file".

d) *acp_change_action*

ação - define a função que será tomada no caso de um erro.

Essa rotina permite que o usuário controle o que acontece quando ocorre um erro que envolve mais de um processo remoto. Isto é, quando ocorre um erro que pode ser resultante de uma falha em um processo remoto e que se dá durante a chamada a uma rotina que envolve um conjunto de processos ao invés de um único processo.

O parâmetro *ação* pode assumir um dos seguintes valores especiais:

ACP$RETURN_ON_FIRST_ERROR - Faz com que a rotina termine após detectar um erro, sem tentar completar a operação.

ACP$KILL_REMOTE_PROCESS - O processo remoto é morto e a operação continua.

ACP$REPORT_TO_JOB_MANAGER - É a ação "default". O erro é reportado ao "Job Manager" que decide o que deve ser feito.

A chamada a *acp_change_action* altera o procedimento apenas da próxima rotina, voltando ao procedimento "default" após a execução da mesma.

VI.9 - ROTINAS AUXILIARES

a) *acp_log*

texto - texto a ser impresso pelo JM e gravado no "log file".

Essa rotina faz com que seja impresso no terminal do JM e gravado no "log file" um texto de até 2000 caracteres.

b) *acp_set_of_processes*

nclasses - indica o número de classes que comporão o conjunto de processos.

classes1 ... classen - identifica cada classe que fará parte do conjunto de processos.

Essa função retorna um número inteiro que é utilizado como parâmetro para outras funções do CPS que aceitam como valor um conjunto de processos. O conjunto de processos é composto pelo conjunto de processo de uma ou mais classes.

b) *acp_bytes*

primeira_palavra - é a primeira palavra do bloco de memória.

última_palavra - é a última palavra do bloco de memória.

Essa função retorna o tamanho, em bytes, de um bloco de memória que começa em *primeira_palavra* e termina em *última_palavra*.

c) *acp_process_number*

Essa função retorna o número do processo.

d) *acp_class_info*

classe - recebe o número da classe a qual o processo pertence.

primeiro - recebe o número do primeiro processo da classe.

número - recebe o número de processos que fazem parte da classe.

Essa rotina permite que se obtenha informações sobre a classe a qual o processo pertence. Essas informações são o número da classe, o número do primeiro processo de uma classe e o número de processos da classe.

A todos os processos de uma classe são atribuídos valores consecutivos e portanto, sabendo-se o número do primeiro processo e o número de processos de uma classe, sabe-se o número de todos os processos de uma classe. Por exemplo se uma classe tem 4 processos e o primeiro processo da classe é o processo 3, os processos que compõem essa classe são os processos 3, 4, 5 e 6.

e) *acp_job_info*

num_classes - indica o número de classes das quais se deseja informação.

lista_classes - é uma lista com o número das classes desejadas.

lista_de_primeiros - é um "array" de tamanho *num_classes* que recebe o primeiro processo de cada classe especificada em *lista_classes.*

lista_de_numero - é um "array" de tamanho *num_classes* que

recebe o número de processos em cada classe especificada em *lista_classes*.

Essa rotina permite que se obtenha informações sobre qualquer classe.

f) *acp_dead_process_info*

classe - indica o número da classe da qual se obterá informação.

máximo - indica o tamanho de *lista*.

lista - "array" preenchido com o número dos processos mortos.

número - recebe o número de processos mortos.

Essa rotina permite que se obtenha uma lista dos processos mortos de uma determinada classe.

g) *acp_update_user_status*

tamanho - indica o tamanho de *array*.

array - lista com os valores das variáveis de status.

Essa função permite que se atualize o valor de variáveis de status definidas pelo usuário, cuja utilização será descrita no capítulo VII.

h) *acp_user_status_length*

Retorna o número de variáveis de status disponíveis para o usuário. Na versão atual esse valor é 4, mas pode crescer em futuras versões.

CAPÍTULO VII

O PROGRAMA MONITOR

Uma aplicação rodando com o CPS pode ser monitorada pelo usuário usando o programa monitor. Esse programa permite que seja monitorado um processo particular, todos os processos de uma classe ou todos os processos da aplicação.

O monitor pode apresentar dois tipos de informações sobre os processos monitorados: Dados sobre a utilização de alto nível, de interesse geral dos usuários e dados sobre as rotinas de gerenciamento de mensagens.

Os dados sobre a utilização de alto nível mostra o número de mensagens transmitidas e recebidas, o número de chamadas remotas a procedimentos, o número de bytes transferidos com as funções de transferência de dados e os valores de variáveis de status que são definidas pela aplicação. Essas variáveis de status são atualizadas com a função do CPS *acp_update_user_status*. O significado dessa variável é definida pela aplicação. No exemplo da reconstrução, o processo da *classe* 1 poderia usar a variável de status para indicar quantos eventos foram lidos e na *classe* 3 poderia indicar quantos eventos foram escritos.

Quando mais de um processo é monitorado, os valores apresentados correspondem à soma dos valores de todos os processos.

Os dados sobre as rotinas de tratamento de mensagem

têm utilidade para depuração e ajuste do sistema de mensagens do CPS e em geral não é de interesse do usuário. Os dados apresentados são: Número de mensagens (baixo-nível) transmitidos e recebidos, número de retransmissões de mensagens devido a "time-outs", número de mensagens retransmitidas devido a falta de "buffer" de recepção no destinatário, e número de "acknowledges" recebidos após "time-out".

Além dessas duas opções descritas, pode também ser feito o monitoramento das mensagens enviadas pelos processos ao JM e do JM aos processos. A aplicação pode ser ativada com uma opção de "cópia carbono". Essa opção faz com que uma cópia de todas mensagens transmitidas entre processos seja enviada ao JM. Dessa maneira pode ser feito um acopanhamento em tempo real do que está se passando com a aplicação. A opção de "cópia carbono" é muito útil para a depuração mas apresenta um "overhead" considerável devido a duplicação de mensagens. Por causa disso, a opção "cópia carbono" deve ser utilizada basicamente para depuração.

A seguir apresentaremos as telas de diferentes opções do programa monitor.

```
Monitoring Job inta
Job Uptime: 00 00:00:18     Class 02 (processes 002 to 011)          16:07:31


   processes alive                10   processes dead                  0
   send calls                      0   get calls                       0
   bytes sent                      0   bytes gotten                    0
   remote calls made               0   remote calls serviced           0
   pages locked                    0

   cpu time                        0   page faults                     0
   swaps                           0   file I/O                        0

   messages transmitted            0   messages received               0

   user_status[0]                  0   user_status[1]                  0
   user_status[2]                  0   user_status[3]                  0
```

Figura VII.1 - Dados Gerais dos Processos da Classe 2

```
Monitoring Job inta
Job Uptime: 00 00:01:35        Processes 000 to 011                    16:39:22


   processes alive                12   processes dead                  0

   acpmh_bgsend                    0   acpmh_bhandler                 11

   messages transmitted           55   messages received              44

   msg interrupts                133
   msg alloc                     122   msg dalloc                     110

   rudp msgs sent                 55   rudp msgs rec                   66
   rudp acks sent                 66   rudp acks rec                   55
   rudp dgms sent                121   rudp dgms rec                  121
   rudp dgms late                  0   rudp dgms early                 0
   rudp retries                    0   rudp max retries                0
   rudp timeouts                   0   rudp badacks                    0
   rudp quenches                   0   rudp synchs                     0
```

Figura VII.2 - Dados do sistema de mensagem

```
Monitoring Job inta
Job Uptime: 00 00:00:42    Circular Buffer - Continuous                20:00:16


00024:wait completed from JM to P002
00025:wait completed from JM to P003
00026:wait completed from JM to P004
00027:wait completed from JM to P005
00028:wait completed from JM to P006
00029:request wait in queue from P001 to JM - queue 0 state ACP$FULL
00030:request queue proc from P002 to JM - queue 0 procs 0x00000002
00031:request queue proc from P003 to JM - queue 0 procs 0x00000003
00032:request queue proc from P004 to JM - queue 0 procs 0x00000004
00033:request queue proc from P005 to JM - queue 0 procs 0x00000005
00034:request queue proc from P006 to JM - queue 0 procs 0x00000006
00035:wait in queue from JM to P001 - completed
00036:request queue info from P001 to JM - queue 0
00037:queue info from JM to P001 - n args 1
00038:request d
```

Figura VII.3 - Comunicação do JM com os processos

CAPÍTULO VIII
CONCLUSÃO

Ao longo desse trabalho, apresentamos um conjunto de ferramentas para serem utilizadas no desenvolvimento de programas paralelos. Essas ferramentas, chamadas de CPS, foram concebidas de forma a atender a necessidade real de pesquisadores que dependem de uma grande capacidade computacional para resolver problemas científicos.

Para isso, o CPS fornece mecanismos de comunicação, sincronismo, transferência e conversão de dados e gerenciamento de processos. Para oferecer maior flexibilidade, diferentes opções de sincronismo e comunicação são oferecidas, tornando mais fácil a adaptação aos diferentes tipos de problemas.

Cuidado especial foi dedicado às ferramentas de depuração de programas. Essas ferramentas incluem registro dos eventos ocorridos na execução da aplicação, um programa monitor que permite o acompanhamento dos processos e a possibilidade de uso em conjunto com os depuradores convencionais de programas seqüenciais. Essas ferramentas são necessárias porque programas paralelos são bem mais difíceis de serem depurados do que programas seqüenciais.

Na definição da funcionalidade do CPS, foi levado em conta a viabilidade de implementação e possibilidade de uso por parte de usuários não especialistas em processamento paralelo. O resultado final incorpora

conceitos já discutidos na literatura e no capítulo II deste trabalho e apresenta também vários aspectos originais. Dentre esses aspectos originais podemos destacar:

1. os modos assíncronos de chamada remota de procedimentos;
2. o uso de *pontos de sincronismo*, que estende a funcionalidade do *rendez-vous* de forma que possa sincronizar um conjunto de processos ao invés de apenas um par de processos;
3. as *filas* do CPS, que associam estados genéricos de um processo e os dados relativos a esse estado a uma lista de procesos que estão no referido estado ou fila.

Ao incluir conceitos já existentes e conceitos novos, o CPS se apresenta como um conjunto completo e flexível para o desenvolvimento de diferentes aplicações paralelizadas.

Apresenta também um espectro bem amplo de funções, desde funções de alto nível como, por exemplo, as chamadas remotas de procedimentos, até as primitivas básicas de troca de mensagens. Dessa forma, o CPS pode ser usado por usuários comuns, que em geral usam as funções de alto nível e por usuários mais especializados que eventualmente utilizam as primitivas mais básicas de troca de mensagens para implementar soluções particulares. A possibilidade de utilização com diversas linguagens de programação também é um fator importante no seu uso por

diferentes tipos de usuários.

Cumpre lembrar que o CPS resulta do esforço de várias pessoas do LAFEX/CBPF e do R&D Department do Computer Division do Fermilab. Nesse esforço, foi importante a experiência adquirida no desenvolvimento e uso da primeira geração do sistema ACP. Comparando o CPS com o software dessa primeira geração, destacamos as seguintes vantagens:

1. conjunto mais completo de funções;
2. menor dependência em relação à arquitetura;
3. maior portabilidade;
4. mecansimos de depuração mais elaborados.

A importância do CPS deve ser compreendida numa conjuntura onde o uso de processamento paralelo se apresenta como uma importante opção na computação de alto desempenho mas que ainda não atingiu a maturidade plena, principalmente do ponto de vista de "software". Acreditamos que um longo caminho ainda deverá ser percorrido antes de ser atingida essa maturidade.

Dentre os principais problemas a serem resolvidos destacamos a falta de padronização e o fato das soluções existentes apresentarem uma grande dependência da arquitetura do sistema utilizado.

Tendo em vista esse cenário, acreditamos que o CPS é uma valiosa contribuição no campo de processamento paralelo. A sua disponibilidade em diferentes computadores além de servir de ferramenta para que se adquira maior familiaridade e experiência neste tipo de processamento atende às necessidades imediatas de usuários desejosos de

se beneficiarem já das vantagens do processamento paralelo.

É bom lembrar que, apesar de o CPS ter como alvo principal os problemas de Física Experimental de Altas Energias, a experiência com a primeira geração do sistema ACP mostra que o campo de aplicações onde pode ser utilizado é bem mais vasto do que se poderia supor inicialmente. A primeira geração do ACP, apesar de apresentar uma estrutura de "software" bem menos flexível que a da segunda geração, foi utilizada com sucesso em diversas áreas, tais como cálculos de estruturas, resolução de sistemas lineares, engenharia química e outras.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] HWANG, K. e BRIGGS,F. A., *Computer Architecture and Parallel Processing*, Singapura,McGraw-Hill, Inc., 1987.

[2] ANDERSON, A. J., *Multiple Processing: A Systems Overview*, Hertfordshire, Prentice Hall International (UK) Ltd, 1989.

[3] WOLFE, A., "Is Parallel Software Catching Up with the Hardware At Last?", *Supercomputing Review*, pp. 29-33, março de 1989.

[4] AMORIM, C. L., BARBOSA, V. C. e FERNANDES, E. S. T., *Uma Introdução àComputação Paralela e Distribuída*, Campinas, VI Escola de Computação,1988.

[5] ALMEIDA, V., "Computação de Alto Desempenho: Arquitetura e Tendêmncias", Palestra apresentada no Seminário A Modernização da Computação na PUC-RIO: Redes, Conectividade e Supercomputação, 18 de outubro de 1990.

[6] LUSK, E. et alli, *Portable Programs for Parallel Processors*,New York, Holt, Rinehart and Winston, Inc, 1987.

[7] HACK, J. J., "On the promisse of general-purpose parallel computing", *Parallel Conputing*, n°. 10, pp. 261-275, 1989.

[8] KIRNER, C., *Desenvolvimento de Suporte Básico para Sistemas Operacionais Distribuídos*, Tese de Doutorado (D.Sc.) em Engenharia de Sistemas e Computação, COPPE/UFRJ, agosto de 1986.

[9] TANENBAUM,A.S., *Computer Networks*, Englewood Cliffs, N.J., Prentice-Hall, Inc., 1981.

[10] ENSLOW JR., P. H., "Multiprocessor Organization - a Survey", *ACM Computing Surveys*, vol. 9, pp. 103-129, março de 1977.

[11] SANTOS, S. M., "Programação Concorrente: Mecanismos de Comunicação e Sincronização de Processos", Quarta Escola de Computação, São Paulo, Instituto de Matemática e Estatística - Universidade de São Paulo, jullho de 1984.

[12] TANENBAUM, A. S., *Structured Computer Organization*, Englewood Cliffs, N.J., Prentice-Hall International, Inc., 1976.

[13] DEITEL, H. M., *An Introduction to Operating Systems*, Reading, Mass., Addison-Wesley Publishing Company,

1984.

[14] GEHRINGER, E. F. et alli, A Survey of Commercial Parallel Processors, Commercial Parallel Processors, pp. 75-107, 1988.

[15] FOX, G. C. et alli, "Hands-On Parallel Processing", *Byte*, vol. 14, n°. 10, pp. 287-293, oct. 1989.

[16] RANKA, S. et alli, "Programming a Hypercube Multicomputer", *IEEE Software*, pp. 69-77, September 1988.

[17] PERROTT, R. H., *Parallel Programming,* Wokingham, England, Addison-Wesley Publishing Company, Inc., 1987.

[18] ATHAS, W. C. e SEITZ, C. L., "Multicomputers: Message-Passing Concurrent Computers", *IEEE Computer*, August 1988.

[19] LISKOV, B., "Primitives for Distributed Computing", *Proceedings of 7th Symposium on Operating Systems Principles*, Pacific Grove, N. Y., ACM, pp 33-42, dec. 1979.

[20] DIJKSTRA, E. W., "Cooperating Sequential Processes", *Programming Languages,* F. Genius, ed., Academic

Press, New York, pp. 43-122, 1968.

[21] DIJKSTRA, E. W., "Solutions of a Problem in Concurrent Programming Control", *Comunications of the ACM*, vol. 8, n°. 5, pp. 569-570, September 1965.

[22] HANSEN, P. B., *Operating Systems Principles*, Englewood Cliffs, N. J., Prentice-Hall, 1973.

[23] HOARE, C. A. R., "Monitors: an Operating System Structuring Conmcept", *Communications of the ACM*, vol. 17, n°. 10, Oct. 1974.

[24] HOARE, C. A. R., "Communicating Sequential Processes", *Communications of the ACM*, vol. 21, n°. 8, August 1978.

[25] GENTLEMAN, W. M., "Message Passing Between Sequential Processes: The Reply Primitive and the Administrator Concept", *Software - Practice and Experience*, vol. 11, n°. 5, pp. 435-466, may 1981.

[26] HANSEN, P. B., "Distributed Processes: A Concurrent Programming Concept", *Communications of the ACM*, vol. 21, n°. 11, pp. 934-941, november 1978.

[27] NELSON, B. J., *Remote Procedure Call*, Ph.D. Thesis,

Dept. of Computer Science, Carnegie-Mellon University, may 1981. 201 p. (Report CMU-CS-81-119).

[28] HANSEN, P. B., *The Architecture of Concurrent Programs*, Englewood Cliffs, N. J., Prenetice-Hall, Inc, 1977.

[29] WIRTH, N., "Modula: a Language for Modular Multiprograming", *Sofware Practice and Experience*, vol. 7, pp. 3-35, 1977.

[30] MAY, D., "OCCAM", *SIGPLAN Notices*, vol. 18, n°. 4, April 1983.

[31] GEHANI, N., "Working in Concurrent C", *UNIX Review*, vol. 7, n°. 5, pp. 60-70, may 1989.

[32] The VMEbus Specification, *Motorola Semiconductors Products Inc.*, Revision C-1, October 1985.

[33] Branch Bus Specification, *Advanced Computer Program - Fermilab*, March 6, 1986.

[34] VME/Branch Bus Controller - VBBC, *Advanced Computer Program - Fermilab*, April 1988.

[35] Branch Bus to VME Interface (BVI), *Advanced Computer*

Program - Fermilab, March 11, 1987.

[36] SCHULZE, B. et alli, "Arquitetura de Multiprocessamento Paralelo com Processadores ACP/R3000", XX Encontro de Física de Campos e Partículas, Caxambu, Setembro de 1990.

[37] KANE, G., *MIPS R3000 RISC Architecture*, MIPS Computer Systems Inc., Prentice Hall, 1988.

[38] ACP XBUS Specification, *Advanced Computer Programam - Fermilab*, August 1989.

[39] The UNIX User's Reference Manual, *MIPS Computer Systems, Inc.*, 1988.

[40] COMER, D. E., *Internetworking with TCP/IP - Principles, Protocols and Architecture*, Englewood Cliffs, N. J., Prentice-Hall, Inc., 1988.

[41] The UNIX Programmers's Reference Manual, *MIPS Computer Systems, Inc.*, 1988.

[42] ACP Cooperative User's Manual, *Fermilab Computer Research & Development Department*, Batavia, IL, july 1990.

APÊNDICE A
O ARQUIVO JDF

O arquivo JDF ("Job Descriptor File") é usado para passar informações para o JM a fim de que esse possa controlar a execução da aplicação.

Faremos aqui uma breve apresentação da sintaxe do JDF. Maiores detalhes podem ser encontrados em [42].

Cada linha do JDF deve apresentar uma palavra chave que identificará a informação contida naquela linha. Linhas em branco são desprezadas e linhas que começam com um ponto de exclamação são consideradas linhas de comentário. Também é comentário tudo que vier após um ponto de exclamação até o final de uma linha. As palavras chaves são separadas do valor associado a elas por um sinal de igual. Espaços em branco são ignorados.

Todas as linhas de informação são referentes a uma classe e por isso, a primeira informação deve identificar a classe. Para tal, usa-se a palavra chave CLASS para identificar a classe a qual as informações se aplicam. Todas as informações a seguir são associadas a essa classe, até que apareça novamente a palavra CLASS, que marca o início de informações de outra classe. Uma exceção é a linha SHARE CLASSES, que especifica quais classes podem compartilhar o mesmo processador e, se for utilizada, deve ser a primeira linha com informação do JDF.

O JDF pode ser automaticamente analisado por um

pré-processador compatível com o pré-processador da linguagem C, o que permite que se use parâmetros variáveis e que se use comandos condicionais. Na chamada do JM podem ser definidas variáveis simbólicas. O uso do pré-processador não está implementado em todas as máquinas que rodam o CPS.

Todos os valores do JDF tem um valor "default". A seguir faremos uma breve apresentação das linhas do JDF.

SHARE CLASSES:

Especifica quais classes podem compartilhar o mesmo processador. As classes listadas são separadas por vírgulas. Além dos números das classes, pode ser usado o valor NONE para dizer que nenhuma classe pode compartilhar o mesmo processador, ou o valor ALL, para dizer que todas as classes podem compartilhar a mesma CPU.

CLASS:

Essa linha identifica a classe à qual as linhas subseqüentes se referem.

PROGRAM:

Especifica o programa que será executado pelos processos da classe. Pode conter também parâmetros para o programa.

CPU TYPE:

Especifica tipo de CPU onde serão executados os

processos dessa classe.

SPECIFIC CPU:

Ao invés de especificar um tipo de CPU, pode ser especificado uma CPU específica. Essa possibilidade é útil quando se deseja utilizar um determinado computador que tenha um recurso desejado, como por exemplo unidades de E/S, vídeo gráfico ou outro recurso qualquer.

"Specific CPU" e "CPU Type" são mutuamente exclusivos.

NUMBER OF PROCESSES:

Essa linha permite que se determine o número de processos de cada classe.

PROCESSES PER CPU:

Essa opção serve para determinar quantos processos de uma classe serão executados num mesmo processador. Em simulações ou testes em geral todos os processos podem ser executados na mesma CPU.

MINIMUM PROCESSES:

Especifica o número mínimo de processos que devem estar vivos para que a aplicação possa proseguir.

DEBUG:

Especifica quantos processos da classe devem ser inicializados manualmente, conforme explicado em V.9.1.

TIMEOUT:

Permite especificar o tempo, em segundos, que um processo pode executar sem chamar uma das rotinas de alto nível do CPS acp_call, acp_send ou acp_get. Isso permite detectar processos que estejam presos em "loop" eternos ou outro erro do gênero.

APÊNDICE B

PROGRAMAS EXEMPLOS

Apresentaremos nesse apêndice três pequenos programas para exemplificar o uso do CPS.

O primeiro programa é uma versão muito simplificada do problema da reconstrução usado como exemplo no capítulo V. Esse programa é baseado no exemplo apresentado em [42].

O segundo programa mostra o uso do CPS para calcular uma integral usando o método do trapézio. Esse exemplo não pretende ser um estudo de métodos numéricos nem sugerir que o método apresentado seja o mais adequado para ser usado no CPS. Ele pretende apenas ser um exemplo de uso do CPS. Esse programa assume que será utilizado um número fixo de processos.

O terceiro exemplo é uma variação do cálculo da integral que permite que o número de processos utilizados seja variável.

EXEMPLO I

Como já foi visto no capítulo V, o problema da reconstrução pode ser dividido em 3 classes. A classe 1 lê os dados da fita de entrada e os passa para os processos da classe 2 que fazem a análise. O processo da classe 3 recebe os dados analisados e os grava na fita de saída.

A seguir apresentaremos o programa FORTRAN para as três classes e o arquivo JDF. O programa da classe 1 usa

uma rotina EVENT_READ que lê os dados da fita e retorna o número de bytes lidos e atualiza uma variável lógica para indicar o fim dos dados. O programa da classe 2 chama uma rotina EVENT_ANALYZE que faz a análise dos dados. O programa da classe 3 usa uma rotina EVENT_WRITE que grava os dados na fita de saída. Não mostraremos o código dessas rotinas.

Os programas desse exemplo incluem um arquivo chamado RECONSTRUCAO.INC. Esse arquivo define símbolos mnemônicos para valores inteiros que são usados nas rotinas do CPS.

RECONSTRUÇÃO.INC:

```
C       DEFINICAO DAS FILAS
        PARAMETER READY_QUEUE                   = 1
        PARAMETER ANALYZED_QUEUE                = 2

C       DEFINICAO DO PONTO DE SINCRONISMO
        PARAMETER BEGIN_SYNC                    = 0

C       DEFINICAO DOS BLOCOS DE MEMORIA
        PARAMETER IN_BLOCK                      = 1
        PARAMETER OUT_BLOCK                     = 2

C       DEFINICAO DA ROTINA REMOTA
        PARAMETER ANALYZE_REMOTE                = 1
```

PROGRAMA DA CLASSE 1:

```
        PROGRAM CLASS1

        INCLUDE '/usr/include/acp_user.inc'
        INCLUDE 'recosntrucao.inc'

        INTEGER P
        INTEGER IN_BYTES, IN (1000)
        LOGICAL END_OF_RUN
```

```
C       EXECUTA ROTINA DE INICIALIZACAO E ESPERA    TODOS   OS
C       PROCESSOS ESTAREM PRONTOS PARA COMECAR.

        CALL ACP_INIT
        CALL ACP_SYNC (ACP$ALL_PROCESSES, BEGIN_SYNC)

C
C       FICA EM LOOP LENDO DADOS E ENVIANDO PARA A CLASSE 2.
C       TERMINA LOOP QUANDO NAO HOUVER MAIS DADOS.
C
100     CALL EVENT_READ (IN, IN_BYTES, END_OF_RUN)
        IF (END_OF_RUM) GO TO 200
C       PEGA PROCESSO DA FILA DE PROCESSOS PRONTOS
        CALL ACP_DEQUEUE_PROCESS (P, READY_QUEUE)
C       ENVIA DADOS PARA O PROCESSO
        CALL ACP_SEND (P, IN, IN_BYTES, IN_BLOCK, 0)
C       INVOCA      PROCEDIMENTO      REMOTO      NO      MODO      NAO      BLOQUEANTE,
C       COLOCANDO    O    PROCESSO    REMOTO    NA    FILA    DE    PROCESSOS    COM
C       DADOS ANALISADOS
        CALL        ACP_CALL        (P,        ANALYZED_QUEUE,        ANALYZE_REMOTE,
                                IN_BYTES, 0)
        GO TO 100

C       ESPERA QUE O PROCESSO DA CLASSE 3 TENHA LIDO OS
C       DADOS DE TODOS OS PROCESSOS DA CLASSE 2. QUANDO
C       ISSO OCORRE A FILA DE READY_QUEUE FICA CHEIA. O
C       PROCESSO DA CLASSE 3 SABE QUE   O   PROCESSAMENTO
C'      TERMINOU AO ENCONTRAR ACP$END_OF_QUEUE.
        CALL ACP_WAIT_QUEUE (READY_QUEUE, ACP$FULL)
        CALL ACP_QUEUE_PROCESS (ACP$END_OF_QUEUE,
                                        ANALYZED_QUEUE)

C       TERMINA ESSE PROCESSO
        CALL ACP_STOP_PROCESS
        END
```

PROGRAMA DA CLASSE 2:

```
        PROGRAM CLASS2

        INCLUDE '/usr/include/acp_user.inc'
        INCLUDE 'recosntrucao.inc'
        INTEGER IN(1000) OUT(2000)
        COMMON /IN_COMMON/IN
        COMMON /OUT_COMMON/OUT
        EXTERNAL ANALYZE_SUBROUTINE

C       EXECUTA      ROTINA      DE      INICIALIZACAO
        CALL ACP_INIT

C       DECLARA BLOCOS DE MEMORIA PARA TRANSFENCIA DE DADOS
        CALL ACP_DECLARE_BLOCK (IN, 4000, IN_BLOCK)
        CALL ACP_DECLARE_BLOCK (OUT, 8000, OUT_BLOCK)
```

```
C       DECLARA ROTINA QUE SERA' CHAMADA REMOTAMENTE
        CALL            ACP_DECLARE_SUBROUTINE          (ANALYZE_SUBROUTINE,
                                                ANALYZE_REMOTE,2,4,4)

C       DECLARA FILAS EM QUE O PROCESSO PODE SER COLOCADO
        CALL ACP_DECLARE_QUEUE (READY_QUEUE, 0)
        CALL ACP_DECLARE_QUEUE (ANALYZED_QUEUE, 2, 4, 4)

C       COLOCA  PROCESSO  NA  FILA  DE  PRONTOS  E  SINCRONIZA  COM
C       OUTROS PROCESSOS
        CALL ACP_QUEUE (ACP$THIS_PROCESS, READY_QUEUE)
        CALL ACP_SYNC (ACP$ALL_PROCESSES, BEGIN_SYNC)

C       ENTRA NO MODO SERVIDOR REMOTO
        CALL ACP_SERVICE_CALLS
        END


C       SUBROTINA  ANALYZE_SUBROUTINE  E'  A  ROTINA  QUE  ANALISA
C       OS  DADOS  E  E'  CHAMADA  REMOTAMENTE.

        SUBROUTINE ANALYZE_SUBROUTINE (IN_BYTES, OUT_BYTES)

        INTEGER IN_BYTES, OUT_BYTES
        INTEGER IN(1000), OUT(2000)
        COMMON /IN_COMMON/IN
        COMMON /OUT_COMMON/OUT

        CALL EVENT_ANALYZE (IN, IN_BYTES, OUT, OUT_BYTES)
        RETURN
        END
```

PROGRAMA DA CLASSE 3:

```
        PROGRAM CLASS3
        INCLUDE '/usr/include/acp_user.inc'
        INCLUDE 'recosntrucao.inc'

        INTEGER P, IN_BYTES, OUT_BYTES, OUT(2000)

C       EXECUTA  ROTINA  DE  INICIALIZACAO  E  ESPERA  TODOS
C       OS PROCESSOS ESTAREM PRONTOS PARA COMECAR.
        CALL ACP_INIT
        CALL ACP_SYNC (ACP$ALL_PROCESSES, BEGIN_SYNC)

C       LOOP  LENDO  DADOS  DE  PROCESSOS  COM  DADOS  ANALISADOS  E
C       GRAVANDO  EM  FITA.  APOS  LER  PEGAR  OS  DADOS  COLOCA  O
C       PORCESSO  DA  CLASSE  2  DA  FILA  DE  PRONTOS.  TERMINA
C       QUANDO ENCONTRAR ACP$END_OF_QUEUE.

100     CALL ACP_DEQUEUE_PROCESS (P,ANALYZED_QUEUE,IN_BYTES,
                                          OUT_BYTRES)
```

```
        IF (P .EQ. ACP$END_OF_QUEUE) GO TO 200
        CALL ACP_GET (P, OUT, OUT_BYTES, OUT_BLOCK, 0)
        CALL ACP_QUEUE_PROCESS (P, READY_QUEUE)
        CALL EVENT_WRITE(OUT, OUT_BYTES)
        GO TO 100

C       TERMINA O PROGRAMA
200     CALL ACP_STOP_JOB
        END
```

ARQUIVO JDF:

```
!Job de reconstrucao

SHARE CLASSES = 1,3                    !  CLASSE 1 E 3 USAM POUCA
                                       !   CPU   E   PODEM   COMPARTILHAR
                                       ! O MESMO PROCESSADOR.

CLASS = 1
    PROGRAM = class1

CLASS = 2
    PROGRAM = class2
    PROCESSES PER CPU = 1
    NUMBER OF PROCESSES = AS MANY AS POSSIBLE
    CPU TYPE = ACP/R3000

CLASS = 3
    PROGRAM = class3
```

EXEMPLO II:

Neste exemplo apresentamos um programa que calcula a integral da função $f(x) = 6 - 6x^5$, no intervalo de 0. a 1.

Nesse exemplo, onde não houve a pretensão de se aprofundar na questão de métodos numéricos, foi usado o método do trapézio para calcular a integral. Esse método consiste em quebrar o intervalo de integração em vários intervalos pequenos e somar cada uma das áreas sob a função, calculadas nos referidos intervalos. O cálculo dessas áreas é feito aproximando-se a área desejada à área de um trapézio de altura igual ao tamanho do intervalo e bases iguais ao valor da função no início e no fim do mesmo intervalo.

No programa paralelizado, cada processador calcula um sub-intervalo do intervalo total de integração. Para a implementação do programa com o CPS, o problema foi dividido em duas classes. A classe 2 é composta por 10 processos que fazem o cálculo da área dos 10 sub-intervalos em que o intervalo de integração foi dividido. A classe 1 é composta por apenas um processo que faz a soma de todas as áreas calculadas pelos processos da classe 2.

Os processos da classe 2 usam o número de identificação do processo para determinar que intervalo da integração devem calcular.

Os programas da classe 1 e da classe 2 incluem umn arquivo chamado "integral.inc" onde são definidos símbolos mnemônicos para os valores inteiros usados nas rotinas do

CPS. Será apresentado também, para efeito de comparação, uma versão seqüencial desse programa.

PROGRAMA SEQÜENCIAL:

```
        PROGRAM SEQUENCIAL

        PARAMETER NUMPONTOS = 100000

C Declaracao de variaveis:
C ========================
        INTEGER I
        REAL*8 XL, XR, A, AREA, LARGURA

C Inicializacao das variavesi:
C ============================
        AREA = 0.
        LARGURA = 1./NUMPONTOS

C Calculo da integral:
C ====================
        DO 10 I=1,NUMPONTOS
            XL = (I-1) * LARGURA
            XR = I * LARGURA
            A= 3. * LARGURA * ((1. - XL**5.) + (1. - XR**5.))
            AREA = AREA + A
10      CONTINUE

C Impressao do resultado final:
C =============================
        WRITE (6,100)AREA
100     FORMAT (2X,'AREA = ',G12.4)

        STOP
        END
```

INTEGRAL.INC:

```
C Definicao do ponto de sincronismo:
C ==================================
        PARAMETER INICIO_SINC = 0

C Definicao da fila de processos prontos:
C =======================================
        PARAMETER PRONTO = 0
```

PROGRAMA DA CLASSE 1:

```
        PROGRAM CLASS1
        INCLUDE '/usr/include/acp_user.inc'
        INCLUDE 'integral.inc'
        IMPLICIT NONE

C Declaracao de variaveis:
C ========================
        REAL*8 INTEGRAL,PARTE
        INTEGER PROCESSO

C Inicializacao do CPS:
C =====================
        CALL ACP_INIT

C Espera que processos declarem as filas:
C ========================================
         CALL ACP_SYNC (ACP$ALL_PROCESSES, INICIO_SINC)


C INICIALIZA VARIAVEIS:
C =====================
        INTEGRAL = 0.

C Espera que todos os processos tenham terminado:
C ================================================
        CALL ACP_WAIT_QUEUE (PRONTO, ACP$FULL)

C Enquanto houver processo na fila, soma as areas:
C =================================================
10      CONTINUE
            CALL ACP_DEQUEUE_IF_POSSIBLE (PROCESSO, PRONTO, PARTE)
            IF (PROCESSO .EQ. ACP$EMPTY) GO TO 1000
            INTEGRAL = INTEGRAL + PARTE
        GO TO 10

C Imprime resultado:
C ==================
1000   WRITE (6,1010) INTEGRAL
1010   FORMAT (2X, 'INTEGRAL = ', G12.4)

C Termina a tarefa:
C =================
        CALL ACP_STOP_JOB
        END
```

PROGRAMA DA CLASSE 2:

```
        PROGRAM CLASS2
```

```
      INCLUDE '/usr/include/acp_user.inc'
      INCLUDE 'integral.inc'

      IMPLICIT NONE

      PARAMETER NUMPONTOS = 100000
      PARAMETER NUMPROC = 10

C Declaracao de variaveis:
C ========================
      REAL*8 XL, XR, A, AREA, LARGURA
      INTEGER PROCESSO, CLASSE, PRIMEIRO, NUM_DE_PROC
      INTEGER INICIO, I

C Inicializacao do CPS:
C =====================
      CALL ACP_INIT


C Declara fila de processos prontos:
C ==================================
      CALL ACP_DECLARE_QUEUE (PRONTO, 1, 8)

C Espera que todos os processos declarem a fila:
C ==============================================
      CALL ACP_SYNC (ACP$ALL_PROCESSES, INICIO_SINC)

C Pega informacoes da classe e do processo:
C =========================================
      CALL ACP_CLASS_INFO (CLASSE, PRIMEIRO, NUM_DE_PROC)
      CALL ACP_PROCESS_NUMBER (PROCESSO)

C Inicializa variaveis:
C =====================
      AREA = 0.
      LARGURA = 1. / NUMPONTOS
      INICIO=(NUMPONTOS / NUMPROC) * (PROCESSO - PRIMEIRO)

C Calcula integral:
C =================
      DO 10 I = 1, (NUMPONTOS/NUMPROC)
          XL = (INICIO + I - 1) * LARGURA
          XR = (INICIO + I) * LARGURA
          A= 3. * LARGURA * ((1. - XL**5.) + (1. - XR**5.))
          AREA = AREA + A
10    CONTINUE

C Apos o calculo, coloca valor calculado na fila:
C ================================================
      CALL ACP_QUEUE_PROCESS(ACP$THIS_PROCESS,PRONTO,AREA)

C Espera fim da tarefa:
C =====================
      CALL ACP_STOP_PROCESS

      END
```

ARQUIVO JDF:

```
! JOB PARA CALCULAR A INTEGRAL, PELO METODO DO TRAPEZIO, USANDO
! 10 PROCESSADORES

SHARE CLASSES = NONE

CLASS = 1
     PROGRAM = integral/class1
     CPU TYPE = ACP/R3000 8 MB

CLASS = 2
     PROGRAM = integral/class2a
     NUMBER OF PROCESSES = 10
     PROCESSES PER CPU = 1
     CPU TYPE = ACP/R3000 8 MB
     MINIMUM PROCESSES = 10
```

EXEMPLO III:

Esse exemplo é uma variante do exemplo II, onde ao invés de um número fixo de 10 processos para a classe 2, é usado um múmero variável, dependendo da disponibilidade dos mesmos.

Só apresentaremos o programa da classe 2 e o arquivo JDF. O programa da classe 1 e o arquivo "integral.inc" ficam inalterados em relação ao exemplo II.

PROGRAMA DA CLASSE 2:

```
        PROGRAM CLASS2

        INCLUDE '/usr/include/acp_user.inc'
        INCLUDE 'integral.inc'

        IMPLICIT NONE

C Declaracao de variaveis:
C ========================
        REAL*8 XL, XR, A, AREA, LARGURA
        INTEGER PROCESSO, CLASSE, PRIMEIRO, NUM_DE_PROC
```

```
        INTEGER INICIO, I
        INTEGER NUM_DE_PONTOS

C Inicializacao do CPS:
C ====================
        CALL ACP_INIT

C Declara fila de processos prontos:
C ==================================
        CALL ACP_DECLARE_QUEUE (PRONTO, 1, 8)

C Espera que todos os processos declarem a fila:
C ==============================================
        CALL ACP_SYNC (ACP$ALL_PROCESSES, INICIO_SINC)

C Pega informacoes da classe e do processo:
C =========================================
        CALL ACP_CLASS_INFO (CLASSE, PRIMEIRO, NUM_DE_PROC)
        CALL ACP_PROCESS_NUMBER (PROCESSO)

C Inicializa variaveis:
C =====================
        AREA = 0.
        NUM_DE_PONTOS = 100000 / NUM_DE_PROC
        LARGURA = 1. / (NUM_DE_PROC * NUM_DE_PONTOS)
        INICIO = NUM_DE_PONTOS * (PROCESSO - PRIMEIRO)

C Calcula integral:
C =================
        DO 10 I = 1, NUM_DE_PONTOS
            XL = (INICIO + I - 1) * LARGURA
            XR = (INICIO + I) * LARGURA
            A=3. * LARGURA * ((1. - XL**5.) + (1. - XR**5.))
            AREA = AREA + A
10      CONTINUE

C Apos o calculo, coloca valor calculado na fila:
C ================================================
        CALL ACP_QUEUE_PROCESS (ACP$THIS_PROCESS, PRONTO,
       C                                          AREA)

C Espera fim da tarefa:
C =====================
        CALL ACP_STOP_PROCESS
        END
```

ARQUIVO JDF:

```
! JOB PARA CALCULAR A INTEGRAL, PELO METODO DO TRAPEZIO,
! USANDO TODOS OS PROCESSADORES DISPONIVEIS

SHARE CLASSES = NONE
```

```
CLASS = 1
     PROGRAM = teste/cps/integral/class1

CLASS = 2
     PROGRAM = integral/class2b
     NUMBER OF PROCESSES = AS MANY AS POSSIBLE
     PROCESSES PER CPU = 1
     CPU TYPE = ACP/R3000 8 MB
```