Universidade de São Paulo
Departamento de Ciências de Computação e Estatística

Computação Paralela

Regina Helena Carlucci Santana
Marcos José Santana
Márcio Augusto de Souza
Paulo Sérgio Lopes de Souza
Ana Elisa Tozetto Piekarski

São Carlos
Setembro de 1997

Índice

Índice de Figuras

1. Introdução

Desde o surgimento do primeiro computador digital eletrônico, ENIAC (Electronic Numerical Integrator and Computer), em 1946, a computação passou por um processo evolutivo intenso, a nível de hardware e software, a fim de proporcionar maior desempenho e ampliar o leque de aplicações que podem ser computacionalmente resolvidas de maneira eficiente.

A primeira geração de computadores estabeleceu os conceitos básicos de organização dos computadores eletrônicos, e na década de 50 apareceu o modelo computacional que se tornaria a base de todo o desenvolvimento subsequente, chamado “modelo de von Neumann”.

O modelo de von Neumann é formado basicamente pelo conjunto: processador, memória e dispositivos de E/S. O processador executa instruções seqüencialmente, de acordo com a ordem ditada por uma unidade de controle.

Durante as décadas seguintes, houve um aperfeiçoamento muito grande dos componentes do modelo von Neumann, e este ciclo evolutivo permanece até os dias atuais.

Aproximadamente a partir da década de 70, porém, novas tecnologias de organização computacional começaram a se desenvolver, em paralelo à evolução do modelo de von Neumann, buscando maior eficiência e facilidade no processo computacional. E, dentre estas, podem ser citadas a computação paralela, as redes de computadores, os mecanismos pipeline, entre outras.

O início da década de 80 foi marcado pelo surgimento da filosofia VLSI de projeto de microcomputadores, que significou um fator estimulante para o desenvolvimento da computação, tanto em nível de arquiteturas paralelas como arquiteturas de von Neumann, visto que ofereceu maior facilidade e menor custo no projeto de microprocessadores cada vez mais complexos e menores.

Na última década, as redes de computadores se tornaram mais rápidas e mais confiáveis, o que possibilitou a interligação dos computadores pessoais e *workstations* de maneira eficiente formando os sistemas distribuídos. Sistemas distribuídos são construídos com o intuito de oferecer uma imagem de sistema único (apesar da distribuição de seus componentes), de maneira que o usuário não perceba que está trabalhando em vários computadores ao mesmo tempo.

Sistemas distribuídos têm sido utilizados para a execução de programas paralelos, em substituição às arquiteturas paralelas, em virtude de seu menor custo e maior flexibilidade. Nesse sentido, apesar de terem surgido por motivações diferentes, a computação paralela e a computação distribuída têm demonstrado uma certa convergência, visto que ambas possuem características e problemas semelhantes (por exemplo: balanceamento de carga, modularidade, tolerância a falhas, etc.).

Dentro desse contexto, pode-se observar que a computação paralela e a computação seqüencial de von Neumann, apesar de desenvolverem-se de maneira relativamente independente, atualmente têm apresentado uma relação de interseção significativa.

Este trabalho apresenta uma visão geral da computação paralela através da apresentação de vários conceitos em nível de software e hardware. São revisados os fundamentos da computação paralela e discutido o uso de máquinas paralelas virtuais.

1.1. Definições e Motivações

Hwang [HWA84] define processamento paralelo como:

> "Forma eficiente do processamento de informações com ênfase na exploração de eventos concorrentes no processo computacional"

Processamento paralelo implica na divisão de uma determinada aplicação de maneira que esta possa ser executada por vários elementos de processamento, que por sua vez deverão cooperar entre si (comunicação e sincronismo), buscando eficiência através da quebra do paradigma de execução seqüencial do fluxo de instruções, ditado pela filosofia de von Neumann.

A idéia de se utilizar paralelismo no processo computacional é quase tão antiga quanto os computadores eletrônicos. O próprio von Neumann, em seus artigos, sugeria formas paralelas de se resolver equações diferenciais. O passo inicial do processamento paralelo, porém, é considerado o surgimento do computador ILLIAC IV, construído na Universidade de *Illinois* em 1971, composto por 64 processadores [AMO88] [NAV89].

Vários fatores explicam a necessidade do processamento paralelo. O principal deles trata-se da busca por maior desempenho [ALM94] [ZAL91]. As diversas áreas nas quais a computação se aplica, sejam científicas, industriais ou militares, requerem cada vez mais poder computacional, em virtude dos algoritmos complexos que são utilizados e do tamanho do conjunto de dados a ser processado [KIR91].

Além da busca de maior desempenho, outros fatores motivaram (e motivam) o desenvolvimento da computação paralela [ALM94] [AMO88]. Entre eles:

- O desenvolvimento tecnológico (principalmente o surgimento da tecnologia VLSI de projeto de microprocessadores), permitiu a construção de microprocessadores de alto desempenho que, agrupados, possibilitam um ganho significativo de poder computacional. Além disso, tais configurações possibilitam uma melhor relação custo/desempenho, quando comparadas aos caros supercomputadores [ZAL91];
- Restrições físicas, como a velocidade finita da luz, tornam difícil o aumento de velocidade em um único processador;

- Vários processadores fornecem uma configuração modular, o que permite o agrupamento desses processadores em módulos, de acordo com a natureza da aplicação. Além disso, tem-se um meio de extensão do sistema através da inclusão de novos módulos;
- Tolerância a falhas (por exemplo, através da redundância de hardware).

Além disso, várias aplicações são inerentemente paralelas, e perde-se desempenho pela necessidade de torná-las seqüenciais. O chamado "gargalo de von Neumann", segundo Almasi [ALM94], tem diminuído a produtividade do programador, daí a necessidade de novas maneiras de organização do processamento computacional.

Contudo, substituir uma filosofia computacional já firmemente estabelecida pelas várias décadas de existência da computação, como é a filosofia de von Neumann, é algo que representa um obstáculo de dimensões muito grandes, e que de certa maneira dificulta a difusão da computação paralela [ALM94].

1.2. Questões Relacionadas

Enquanto a definição da computação paralela possa parecer simples, existem muitos aspectos que devem ser levados em consideração. Almasi [ALM94] apresenta uma definição de um processador paralelo, e em seguida enumera um conjunto de questões relacionadas.

Segundo Almasi, um processador paralelo é:

1. "Coleção de elementos de processamento
2. que se comunicam e cooperam entre si e
3. resolvem um problema mais rapidamente"

Algumas das questões relacionadas a cada uma das partes da definição acima são:

1. "Coleção de elementos de processamento...
   - qual o número ideal de elementos?
   - qual a potência e função de cada um?
   - qual a tecnologia adotada?
   - qual a quantidade de memória disponível?
   - como é organizada a memória?
   - todos os elementos de processamento são iguais?

2.que se comunicam e cooperam entre si e...

- como os elementos de processamento irão se comunicar?
- como será implementado o sincronismo entre as tarefas?
- como separar o problema em diversas tarefas?
- como organizar a execução paralela, visando minimizar a comunicação e atingir um bom balanceamento?

3.resolvem um problema mais rapidamente”

- que tipos de problemas serão considerados?
- qual *speedup* a ser alcançado?

2. Conceitos Básicos

Este Capítulo apresenta alguns conceitos fundamentais que serão adotados nesta apostila.

2.1. Paralelismo e Concorrência

Concorrência existe quando, em um determinado instante, dois ou mais processos começaram a sua execução, mas não terminaram. Por essa definição, concorrência pode ocorrer tanto em sistemas com um único processador, quanto em sistemas com múltiplos processadores.

Afirmar que processos estão sendo executados em paralelo implica na existência de mais de um processador, ou seja, paralelismo (ou paralelismo físico) ocorre quando há mais de um processo sendo executado no mesmo intervalo de tempo. Esse tipo de concorrência é demonstrado na Figura 2.1 que mostra a execução de três processos (e1, e2 e e3) em função do tempo.

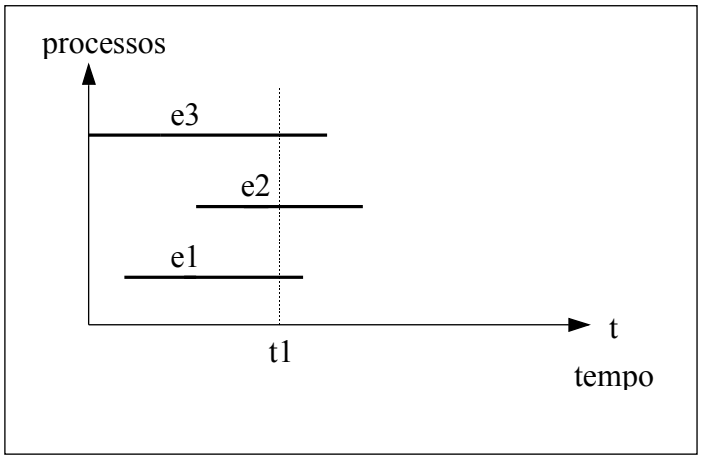

Figura 2.1 - Paralelismo Físico

Quando vários processos são executados em um único processador, sendo que somente um deles é executado a cada vez, tem-se um pseudo-paralelismo (paralelismo lógico). O usuário tem a falsa impressão de que suas tarefas são executadas em paralelo, mas, na realidade, o processador é compartilhado entre os processos. Isso significa que, em um determinado instante, somente um processo é

executado, enquanto que os outros que já foram iniciados aguardam a liberação do processador para continuarem (Figura 2.2).

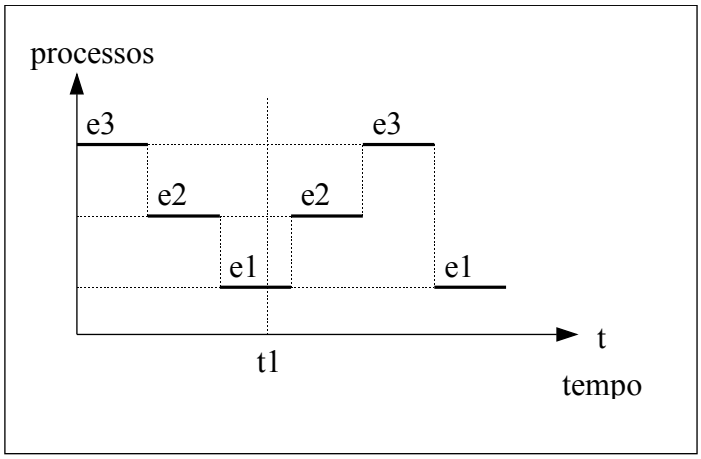

Figura 2.2 - Paralelismo Lógico

Baseado nas definições apresentadas acima, é possível definir três tipos de estilo de programação dentro da computação, que são:

- **Programação seqüencial**: caracteriza-se pela execução de várias tarefas uma após a outra;
- **Programação concorrente**: caracteriza-se pela iniciação de várias tarefas, sem que as anteriores tenham necessariamente terminado (sistemas multi ou uniprocessadores);
- **Programação paralela**: caracteriza-se pela iniciação e execução das tarefas em paralelo (sistemas multiprocessadores).

Para melhor entender as diferenças entre os três estilos acima, considere o exemplo abaixo, onde é apresentado um algoritmo para a preparação de um jantar. O cardápio será constituído de carne, salada e sobremesa. Uma tarefa eqüivale ao preparo de cada um dos pratos.

Um algoritmo seqüencial seria o seguinte:

```
Abrir refrigerador;
Se estiver vazio
        então   vá ao restaurante;
        senão   preparar salada
                preparar carne
                preparar sobremesa
                comer.
```

Percebe-se que no algoritmo acima cada um dos pratos é preparado separado e em seqüência. Esse algoritmo é ineficiente, pois gera desperdício de tempo (por exemplo, enquanto se espera a carne assar). Esse tempo poderia ser aproveitado no início de outra atividade. Esse problema é resolvido pela adoção de um algoritmo concorrente, onde todos os pratos são preparados juntos (concorrentemente). Um algoritmo concorrente é, por exemplo:

```
Abrir refrigerador
Se estiver vazio
        então  vá ao restaurante;
        senão  lavar a alface
                colocar de molho
                temperar a carne
                colocar a carne para cozinhar
                preparar a sobremesa
                escorrer a alface
                temperar a alface
                retirar a carne do forno
comer.
```

Neste caso, as tarefas são executadas concorrentemente, por uma única pessoa (processador). Evita-se assim o tempo ocioso, e os resultados são mais satisfatórios, no sentido que todos os pratos serão servidos logo após o seu preparo. Uma maneira mais eficiente de se executar esse algoritmo, é através do trabalho de duas pessoas (dois processadores). Nesse caso, as tarefas são distribuídas entre as duas pessoas, de maneira que consiga-se melhor eficiência (menor tempo) no preparo dos alimentos. Nesse caso, constrói-se um algoritmo paralelo real.

```
Fernando:                                   Marina:

Abrir refrigerador
Se estiver vazio                            Se estiver vazio
        então  vá ao restaurante;                   então  vá ao restaurante;
        senão                                       senão
                temperar a carne                            preparar a salada
                preparar a sobremesa                        colocar a carne para assar
                                                            retirar a carne do forno
comer.                                      comer.
```

2.2. Granulação

Granulação, ou nível de paralelismo, relaciona o tamanho das unidades de trabalho submetidas aos processadores. Esta é uma definição muito importante na computação paralela, visto que está intimamente ligada ao tipo de plataforma (o porte e a quantidade de processadores) à qual se aplica o paralelismo [KIR91].

Diversas definições de granulação podem ser encontradas na literatura [ALM94] [HWA84] [KIR91] [NAV89]. Mas, de maneira simples, a granulação pode ser dividida em três níveis: fina, média e grossa.

Granulação grossa relaciona o paralelismo ao nível de processos e programas, e geralmente se aplica a plataformas com poucos processadores grandes e complexos [KIR91] [NAV89].

Granulação fina, por outro lado, relaciona paralelismo ao nível de instruções ou operações e implica em grande número de processadores pequenos e simples. A granulação média situa-se em um patamar entre as duas anteriores, implicando em procedimentos sendo executados em paralelo [KIR91] [NAV89].

2.3. *Speedup* e Eficiência

Uma característica fundamental da computação paralela trata-se do aumento de velocidade de processamento através da utilização do paralelismo. Neste contexto, duas medidas muito importantes para a verificação da qualidade de algoritmos paralelos são speedup e eficiência. Novamente, várias definições existem na literatura [QUI87] [KIR91].

Uma definição largamente aceita para speedup é: aumento de velocidade observado quando se executa um determinado processo em p processadores em relação à execução deste processo em 1 processador. Então, tem-se:

$$speedup = \frac{T1}{Tp}$$

Onde: *T1* = tempo de execução em 1 processador

Tp = tempo de execução em p processadores

Idealmente, o ganho de *speedup* deveria tender a p, que seria o seu valor ideal[1]. Porém, três fatores podem ser citados que influenciam essa relação, gerando sobrecargas que diminuem o valor de *speedup* ideal: sobrecarga da comunicação

[1]Alguns autores citam casos de possíveis ganhos maiores a *p* (*SuperSpeedup)*

entre os processadores, partes do código executável estritamente seqüenciais (*Amdahl's Law*) e o nível de paralelismo utilizado (em virtude do uso de granulação inadequada à arquitetura) [ALM94] [QUI87].

Outra medida importante é a eficiência, que trata da relação entre o speedup e o número de processadores.

$$eficiência = \frac{speedup}{p}$$

No caso ideal (*speedup* = p), a eficiência seria máxima e teria valor 1 (100%).

2.4. Pipeline

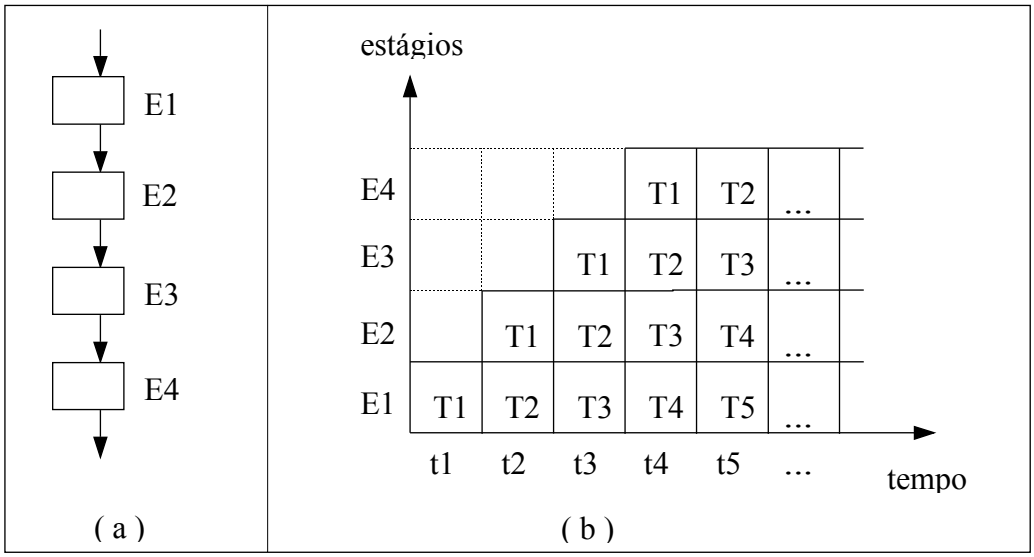

Figura 2.3 - *Pipeline*. (a) Organização dos Estágios (b) Gráfico de Execução.

O paralelismo físico pode ser subdividido em dois tipos: paralelismo espacial (paralelismo real) e temporal (pipeline) [KIR91].

O **paralelismo temporal**, ou ***pipeline***, implica na execução de eventos sobrepostos no tempo. Uma determinada tarefa é subdividida em uma seqüência de subtarefas, sendo cada uma delas executada por um estágio especializado de hardware e software que opera concorrentemente com os outros estágios do pipeline [HWA84].

Considere, por exemplo, uma determinada tarefa dividida em 4 estágios, chamados E1, E2, E3 e E4. Os estágios são organizados seqüencialmente, de maneira que a partir do estágio inicial E1, cada um deles é alimentado pelo estágio imediatamente anterior (Figura 2.3 (a)). A cada instante de tempo, uma nova tarefa é

iniciada, denominadas T1, T2, T3, ... , Tn. A Figura 2.3 (b) apresenta um gráfico relacionando os vários estágios de execução *pipeline* em relação ao tempo transcorrido. Após 4 unidades de tempo, o fluxo de tarefas completas (que completaram os 4 estágios) torna-se contínuo, sendo uma tarefa terminada a cada unidade de tempo. Daí resulta o paralelismo de eventos sobrepostos.

Período de *Clock*:

Considere *k* como o número de tarefas submetidas e *n* o número de estágios do *pipeline*. Cada um dos *n* estágios do *pipeline* possuem tempos de duração igual a t_1, t_2, ..., t_n. Suponha que exista um *buffer* intermediário entre cada um dos estágios *pipeline*, e que o tempo de acesso a esse *buffer* seja igual a t_B. Então, o período de *clock* de um *pipeline* é definido como [HWA84]:

$$\tau = \max \{ t_1, t_2, ..., t_n\} + t_B$$

A partir do momento que um *pipeline* esteja completo (todos os estágios ocupados), uma tarefa será completada a cada período de *clock*. Portanto, com uma freqüência $f = 1/\tau$.

Speedup:

Se as *k* tarefas fossem executadas seqüencialmente, seriam gastos aproximadamente $T_1 = n.k$ períodos de *clock*. Em uma estrutura *pipeline*, têm-se um tempo de $T_k = k + (n-1)$ períodos de *clock*, onde k ciclos são utilizados para completar o *pipeline* e terminar a primeira tarefa, e (n-1) ciclos são utilizados para executar as (n-1) tarefas restantes. Têm-se então:

$$speedup = \frac{T1}{Tp} = \frac{n.k}{k + (n + 1)}$$

Percebe-se que, se k >> n, o *speedup* tende a *k*, que é o seu valor ideal. Porém, esse valor nunca é alcançado em virtude de vários fatores de sobrecarga, como dependências entre instruções, interrupções, entre outros.

Classificações:

A idéia da utilização de *pipeline* surgiu na década de 70, e atualmente é empregada em vários níveis do processo computacional. Hwang [HWA84] apresenta uma classificação para os vários tipos possíveis de *pipeline*. São eles:

- **pipeline aritmético**: as unidades lógicas e aritméticas (ULA) de um computador podem ser organizadas de maneira pipeline. Por exemplo, os 14 ou mais estágios pipeline usados no Cray-1;

- **pipeline de instruções**: a execução de um conjunto de instruções pode ser estruturada de maneira pipeline pela sobreposição da execução da instrução corrente com as operações de busca, decodificação e busca de operandos das instruções subseqüentes. Por exemplo, o Pentium;
- **pipeline de processadores**: refere-se ao processamento pipeline de um conjunto de dados em uma cadeia de processadores, onde cada um processa uma tarefa específica. Por exemplo, processadores vetoriais pipeline.

Pode-se também classificar os tipos de *pipeline* de acordo com suas configurações e estratégias de controle [HWA84]:

- **Unifunção ou multifunção**: executa sempre a mesma função ou várias funções diferentes;
- **Estático ou dinâmico**: pode-se mudar a função durante a execução (dinâmico) ou não (estático);
- **Escalar ou vetorial**: dependendo do tipo de instruções ou dados, definem-se pipeline escalar (instruções escalares em tipos básicos de dados) ou vetorial (instruções vetoriais em vetores de dados).

3. Arquiteturas Paralelas

O conjunto de arquiteturas computacionais pode ser dividido em três grupos básicos:

1.Arquiteturas seqüenciais;

2.Arquiteturas com mecanismos de paralelismo de baixo nível [DUN90]:

- pipeline de instruções;
- múltiplas unidades funcionais na UCP, fornecendo unidades independentes para operações booleanas e aritméticas que operam concorrentemente;
- processadores de E/S separados da UCP;

3.Arquiteturas paralelas.

Este trabalho apresenta apenas arquiteturas paralelas explicítas (3), definidas da seguinte forma [DUN90]:

> "Uma arquitetura paralela fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções utilizando o processamento paralelo, através da existência de múltiplos processadores, estes simples ou complexos, que cooperam para resolver problemas através de execução concorrente"

A utilização de arquiteturas paralelas possui pontos positivos e negativos. Entre as vantagens, todos os aspectos positivos citados na Seção 1.2 (Definições e Motivações) podem ser aplicados às arquiteturas paralelas.

Entre os pontos negativos, citam-se:

- Programação mais difícil;
- Há necessidade de técnicas de balanceamento de carga para melhor distribuição dos processos nos vários processadores;
- Há necessidade de sincronismo e comunicação entre os processos, o que gera certa complexidade;
- A sobrecarga gerada no sistema (por exemplo, na comunicação entre processos), que impede que se tenha um *speedup* ideal.

Existem muitas maneiras de se organizar computadores paralelos. Para que se possa visualizar melhor todo o conjunto de possíveis opções de arquiteturas paralelas,

é interessante classificá-las. Segundo Ben-Dyke [BEN93], uma classificação ideal deve ser:

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

Muito já foi desenvolvido em termos de *hardware* paralelo, e várias classificações foram propostas [ALM94] [BEN93] [DUN90] [HWA84]. A mais conhecida pela comunidade computacional é a classificação de Flynn [FLY72].

3.1. Classificação de Flynn

Segundo Flynn, o processo computacional deve ser visto como uma relação entre fluxos de instruções e fluxos de dados. Um fluxo de instruções equivale a uma seqüência de instruções executadas (em um processador) sobre um fluxo de dados aos quais estas instruções estão relacionadas [ALM94] [DUN90] [FLY72] [HWA84] [NAV89].

Baseando-se nas possíveis unicidade e multiplicidade de fluxos de dados e instruções, dividem-se as arquiteturas de computadores em 4 classes. Para cada classe, é apresentada uma esquematização genérica.

SISD

Single Instruction Stream/Single Data Stream (Fluxo único de instruções/Fluxo único de dados): corresponde ao tradicional modelo von Neumann. Um processador executa seqüencialmente um conjunto de instruções sobre um conjunto de dados (Figura 3.1).

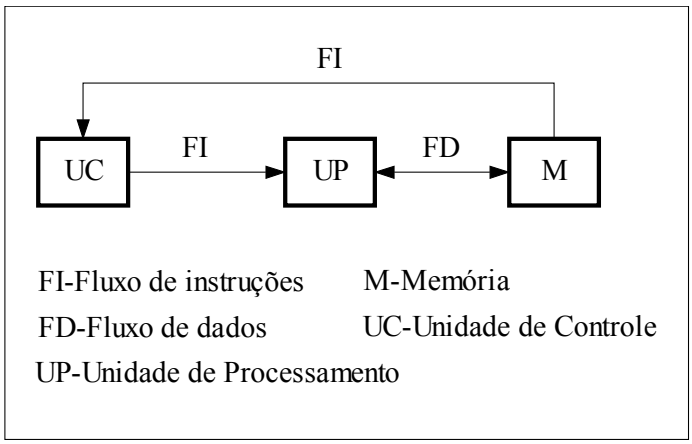

Figura 3.1 - Modelo Computacional SISD.

SIMD

Single Instruction Stream/Multiple Data Stream (Fluxo único de instruções/Fluxo múltiplo de dados). Envolve múltiplos processadores (escravos) sob o controle de uma única unidade de controle (mestre), executando simultaneamente a mesma instrução em diversos conjuntos de dados (Figura 3.2). Arquiteturas SIMD são utilizadas, por exemplo, para manipulação de matrizes e processamento de imagens.

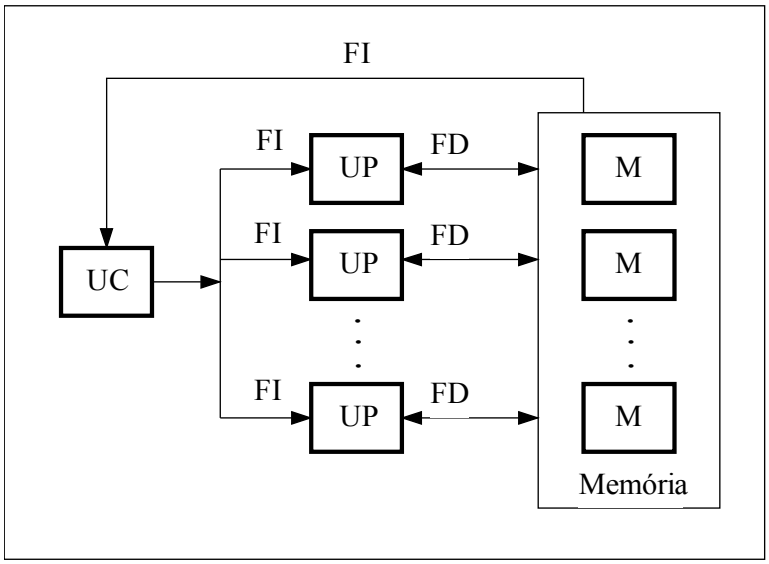

Figura 3.2 - Modelo Computacional SIMD.

MISD

Multiple Instruction Stream/Single Data Stream (Fluxo múltiplo de instruções/Fluxo único de dados). Envolve múltiplos processadores executando diferentes instruções em um único conjunto de dados. Geralmente, nenhuma arquitetura é classificada como MISD, isto é, não existem representantes desta

categoria. Alguns autores consideram arquiteturas *pipeline* como exemplo deste tipo de organização (Figura 3.3).

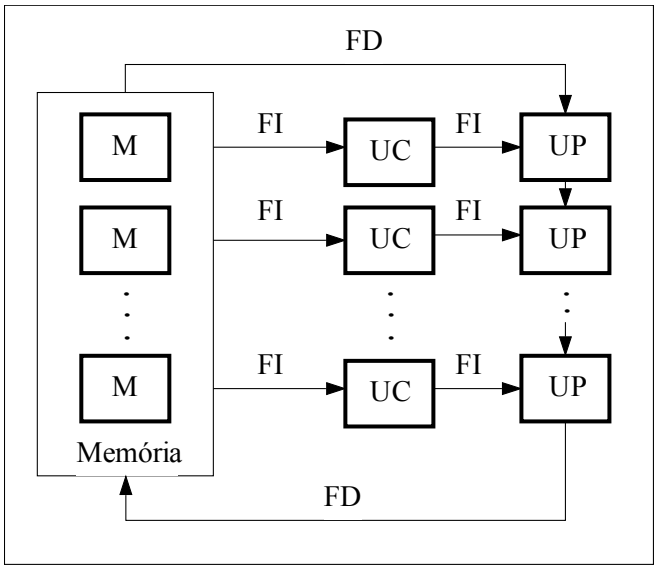

Figura 3.3 - Modelo Computacional MISD.

MIMD

Multiple Instruction Stream/Multiple Data Stream (Fluxo múltiplo de instruções/Fluxo múltiplo de dados). Envolve múltiplos processadores executando diferentes instruções em diferentes conjuntos de dados, de maneira independente (Figura 3.4). Esta classe engloba a maioria dos computadores paralelos.

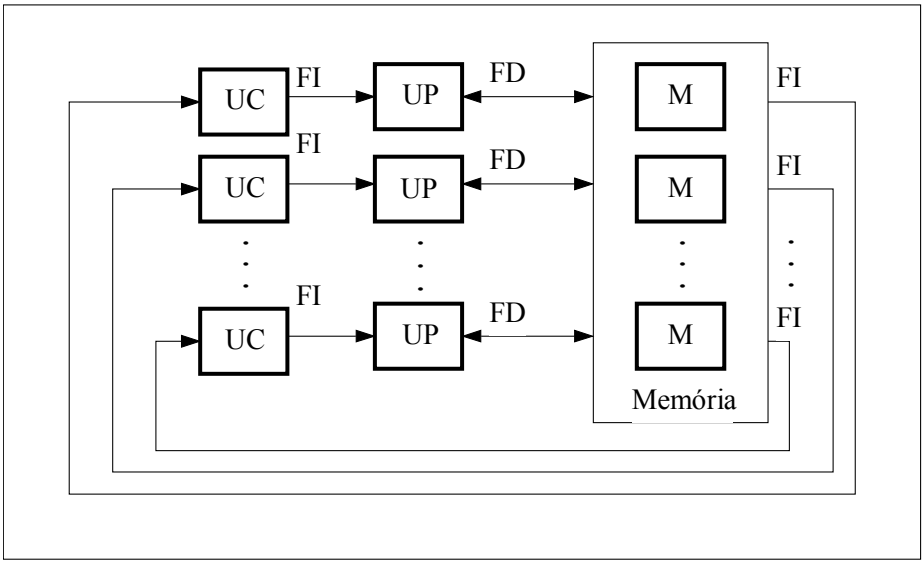

Figura 3.4 - Modelo Computacional MIMD.

MIMD versus SIMD

Ambos os tipos de organizações computacionais apresentam vantagens e desvantagens. Arquiteturas SIMD, por apresentarem fluxo único de instruções, oferecem facilidades para a programação e depuração de programas paralelos. Além disso, seus elementos de processamento são simples, pois são destinados à computação de pequena granulação. Por outro lado, arquiteturas MIMD apresentam grande flexibilidade para a execução de algoritmos paralelos (arquiteturas SIMD geralmente se destinam a processamento de propósito específico), e apresentam bom desempenho em virtude de seus elementos de processamento serem assíncronos [ALM94].

A classificação de Flynn apresenta alguns problemas. Ela não é abrangente o suficiente para incluir alguns computadores modernos (por exemplo, processadores vetoriais e máquinas de fluxo de dados), falhando também, no que concerne a extensibilidade da classificação. Outro inconveniente desta classificação é a falta de hierarquia. A classificação MIMD, por exemplo, engloba quase todas as arquiteturas paralelas sem apresentar subníveis. No entanto, apesar de antiga (proposta em 1972), a classificação de Flynn é bastante concisa e a mais utilizada.

A fim de acrescentar novas arquiteturas paralelas surgidas, sem descartar a classificação de Flynn (visto que esta é muito difundida), Duncan [DUN90] propôs uma classificação mais completa, e que permite apresentar uma visão geral dos estilos de organização para computadores paralelos da atualidade.

3.2. Classificação de Duncan

Duncan, em sua classificação, exclui arquiteturas que apresentem apenas mecanismos de paralelismo de baixo nível, que já se tornaram lugar comum nos computadores modernos. Exemplos desses mecanismos são: *pipeline* dos estágios de execução de uma instrução e unidades funcionais múltiplas em uma única UCP.

A classificação de Duncan é apresentada na Figura 3.5, e a seguir cada um dos seus componentes são brevemente discutidos.

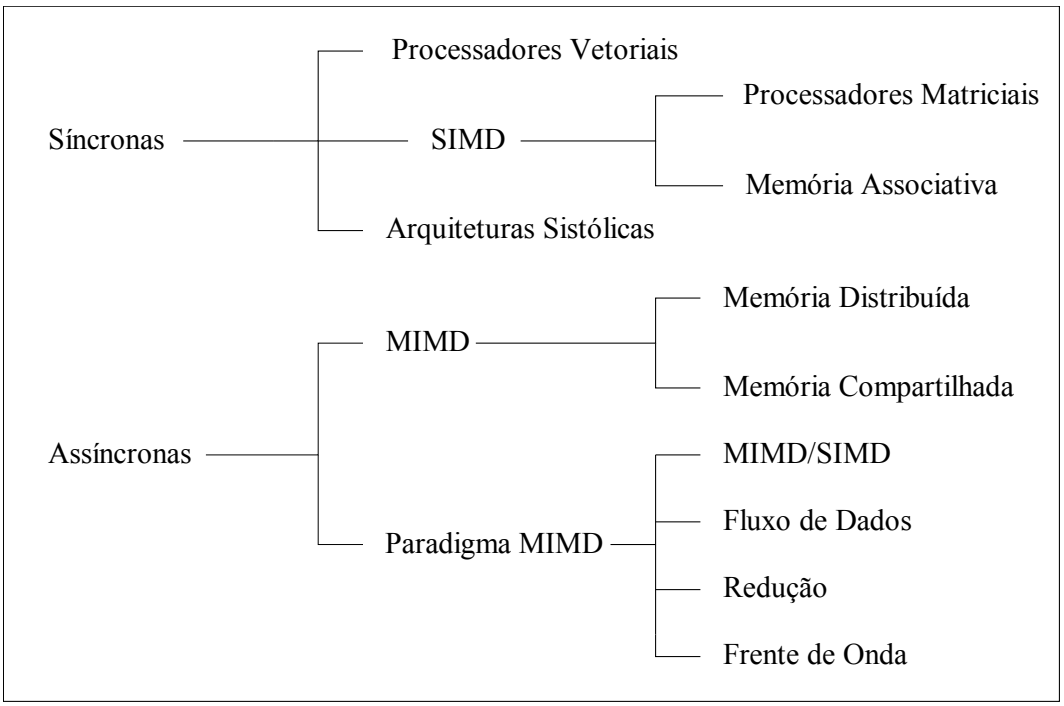

Figura 3.5 - Classificação de Duncan

Arquiteturas Síncronas

Arquiteturas paralelas síncronas coordenam suas operações concorrentes sincronamente em todos os processadores, através de relógios globais, unidades de controle únicas ou controladores de unidades vetoriais [DUN90]. Tais arquiteturas apresentam pouca flexibilidade para a expressão de algoritmos paralelos [BLE94].

Processadores Vetoriais: caracterizam-se pela existência de um *hardware* específico para a execução de operações em vetores. Essas operações são implementadas através de instruções vetoriais. Os processadores vetoriais *pipeline* são caracterizados por possuírem múltiplas unidades funcionais organizadas de maneira *pipeline*, onde são implementadas as instruções vetoriais. Essas arquiteturas foram criadas com o intuito de prover mecanismos eficientes para o suporte de cálculos pesados em matrizes ou vetores.

Esquematicamente, a organização básica de um processador vetorial apresenta: um processador escalar (para a execução do código não vetorizável do programa), uma unidade de processamento vetorial, e um processador de instruções (que define quais instruções serão executadas em qual dos processadores) [KIR91].

Exemplificando uma instrução vetorial, considere o código:

```
for  i = 1 to 100
     a_i = 0
```

Em uma máquina vetorial, apenas a estrutura $a_{i:100} = 0$ executa esta instrução, enquanto que, em uma máquina escalar, a busca e decodificação das instruções ocorrerá 100 vezes.

É importante ressaltar que o paralelismo dessas arquiteturas é explorado em nível de compilação e não de programação. Exemplos de processadores vetorias são: Cray 1 e CDC Cyber 205 [ALM94].

Arquiteturas SIMD: arquiteturas SIMD apresentam múltiplos processadores, sob a supervisão de uma unidade central de controle, que executam a mesma instrução sincronamente em conjuntos de dados distintos. Podem ser organizadas de duas maneiras [HWA84]:

- **Processadores Matriciais**: projetados especialmente para fornecer estruturas para computação sobre matrizes de dados, são empregados para fins específicos. Fornecem acesso à memória via endereço, o que os diferencia do modelo associativo, discutido a seguir. Exemplos: *Illiac IV* e *IBM GF11* [ALM94].
- **Memória Associativa**: relacionam arquiteturas SIMD cujo acesso à memória é feito de acordo com o seu conteúdo, em contraste ao método de acesso usual, via endereço. O esquema associativo visa permitir o acesso paralelo à memória, de acordo com certo padrão de dados. Exemplos: *Goodyear Aerospace STARAN* e *Parallel Element Processing Esemble (PEPE)* [HWA84].

Arquiteturas Sistólicas: propostas no início da década de 80, por H.T. Kung na Universidade de Carnegie Mellon, estas arquiteturas têm como principal objetivo fornecer uma estrutura eficiente para a solução de problemas que necessitem de computação intensiva junto a grande quantidade de operações de E/S. Essas arquiteturas se caracterizam pela presença de vários processadores, organizados de maneira *pipeline*, que formam uma cadeia na qual apenas os processadores localizados nos limites desta estrutura possuem comunicação com a memória.

Desta maneira, o conjunto de dados percorre toda a cadeia de processadores, de maneira rítmica e sincronizada por um relógio global, não havendo armazenamento temporário em memória na comunicação entre os processadores.

Arquiteturas Assíncronas

Estas arquiteturas caracterizam-se pelo controle descentralizado de *hardware*, de maneira que os processadores são independentes entre si. Esta classe é formada basicamente pelas arquiteturas MIMD, sejam convencionais ou não [DUN90].

Arquiteturas MIMD: relacionam arquiteturas compostas por vários processadores independentes, onde se executam diferentes fluxos de instruções em dados locais a esses processadores.

Este tipo de organização pressupõe algoritmos de granulação mais grossa (implicando em plataformas com poucos processadores complexos) com pouca comunicação entre processos (em virtude da sobrecarga de comunicação ser maior), enquanto arquiteturas síncronas utilizam algoritmos de granulação mais fina. Além disso, arquiteturas MIMD oferecem grande flexibilidade para a construção de algoritmos paralelos [BLE94]. Apesar de independentes, os processos executando nos diversos processadores devem cooperar entre si, tornando necessárias a comunicação e o sincronismo entre esses processos. A implementação dos métodos de comunicação e sincronismo depende da organização de memória, que pode ser centralizada ou distribuída (Figura 3.6).

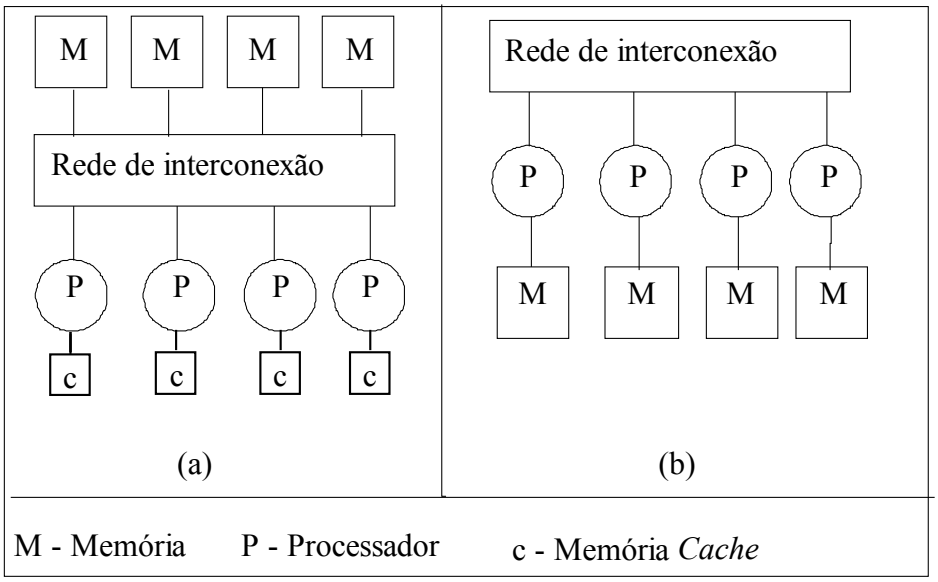

Figura 3.6 - Arquiteturas MIMD. (a) Mem. Centralizada (b) Mem. Distribuída

Arquiteturas de memória centralizada (ou multiprocessadores) caracterizam-se pela existência de uma memória global e única, a qual é utilizada por todos os processadores (fortemente acoplados), de maneira que através do compartilhamento de posições desta memória ocorre a comunicação entre processos. Para evitar que a memória se torne um gargalo, arquiteturas de memória centralizada devem implementar mecanismos de *cache*, além de garantir a coerência dos dados (através de mecanismos de *hardware* e *software*).

Apesar das complicações geradas pela necessidade de gerenciamento de memória *cache*, arquiteuras de memória centralizada apresenta grande flexibilidade de programação, como é descrito adiante, de maneira que estas arquiteturas tornam-se uma opção interessante para o programador paralelo.

Em **arquiteturas de memória distribuída,** cada processador possui sua própria memória local, sendo então fracamente acoplados. Em virtude de não haver compartilhamento de memória, os processos comunicam-se via **troca de mensagens**, que se trata da transferência explicita de dados entre os processadores. Este tipo de organização é também conhecida como multicomputador.

A distinção entre as duas organizações de memória citadas deve ser feita com dois referenciais em mente: *hardware* e *software* [ALM94] [TAN95] [BLE94].

Do ponto de vista de programação, a diferença básica reside nas primitivas de comunicação entre processos. A comunicação em memória centralizada é baseada no compartilhamento de posições de memória, e para que os dados se mantenham consistentes, é necessário um método de controle de acesso a essas variáveis compartilhadas, como por exemplo: semáforos, monitores, etc. Estes mecanismos de programação já são bem conhecidos pelo programador em geral.

No caso de memória distribuída, a comunicação é feita através de troca de mensagens, o que gera algumas complicações, entre elas, o controle de fluxo, controle sobre mensagens perdidas, *bufferização* e bloqueio de rotinas. Então, do ponto de vista do programador, a melhor escolha é a memória compartilhada. É interessante ressaltar que pode-se usar o paradigma de troca de mensagens em memória compartilhada, o que aumenta o leque de opções para o programador.

Em relação ao *hardware*, esta situação se inverte. Arquiteturas de memória distribuída são mais fáceis de se construir e são naturalmente ampliáveis [MCB94] [BLE94]. Portanto, do ponto de vista do projetista, a melhor escolha é uma arquitetura de memória distribuída.

O caso ideal é a existência de uma arquitetura que seja fácil de projetar e programar. Com esse objetivo em mente, foi criada a idéia das arquiteturas de memória centralizada distribuída (ou memória compartilhada virtual). Nesse caso, constrói-se uma plataforma de memória distribuída e, via mecanismos implementados em *hardware* e *software*, emula-se um ambiente de memória centralizada [MCB94] [TAN95] [BLE94].

Paradigma MIMD: essa classe engloba as arquiteturas assíncronas que, apesar de apresentarem a característica de multiplicidade de fluxo de dados e instruções das arquiteturas MIMD, são organizadas segundo conceitos tão fundamentais a seu projeto quanto suas características MIMD. Estas características próprias de cada arquitetura, dificultam a sua classificação como puramente MIMD. Por isso, tais arquiteturas se denominam paradigmas arquiteturais MIMD. Estas arquiteturas são divididas em: arquiteturas MIMD/SIMD, *Dataflow*, Redução e Frente de onda.

Arquiteturas MIMD/SIMD (híbridas), também conhecidas por MSIMD, caracterizam-se por apresentarem controle SIMD para determinadas partes de uma arquitetura MIMD. A flexibilidade que este modelo pode apresentar é atrativa (por exemplo, alguns nós podem ser processadores vetoriais), e aplicações nas quais podem ser utilizadas são processamento de imagens e sistemas especialistas.

Máquinas convencionais von Neumann são denominadas computadores baseados no fluxo de controle, uma vez que instruções são executadas de acordo com a seqüência ditada pelo contador de programas [HWA84]. Para explorar o paralelismo máximo em um programa, foi proposto mudar o controle da execução das instruções de acordo com a dependência dos dados aos quais se aplicam estas instruções, gerando **Arquiteturas a Fluxo de Dados (*dataflow)***. Basicamente, uma instrução é executada assim que todos os seus operandos estão disponíveis. Assim, controla-se de maneira não centralizada a execução de um programa, com os dados fluindo de instrução a instrução, de maneira que se consiga paralelismo em alta escala, de granulação fina, encontrado a tempo de execução.

Arquiteturas de Redução, também conhecidas como arquiteturas dirigidas a demanda, baseiam-se no conceito de redução, que implica que partes do código fonte original sejam reduzidas aos seus resultados em tempo de execução [KIR91]. As instruções são ativadas para serem executadas quando os seus resultados são necessários como operandos por outra instrução já ativada para execução.

Construídas com os mesmos objetivos de balanceamento entre computação e E/S das arquiteturas sistólicas, as **Arquiteturas de Frente de Onda** caracterizam-se por apresentarem uma estrutura sistólica de processadores, combinada ao paradigma assíncrono de execução baseada no fluxo de dados.

Do que foi apresentado acima, conclui-se que o processo computacional pode ser dividida em três tipos de organização, divididos de acordo com o tipo de mecanismo que dita a sequencia na qual as instruções devem ser executadas, que são:

- computação dirigida pelo controle: representa a maioria dos computadores convencionais, controla a execução das instruções através de contadores de instruções (um contador em arquiteturas SIMD e SISD e vários em arquiteturas MIMD);
- computação dirigida pelos dados: o fluxo de dados controla a seqüência de instruções. Exploram um paralelismo muito fino, que a tecnologia atual não permite uma exploração total, restrigindo tais arquiteturas ao meio acadêmico. Por exemplo: arquiteturas *Dataflow*;
- computação dirigida pela demanda: executa instruções à medida que seus resultados são requisitados. Estes tipos de arquiteturas geram grande sobrecarga no seu gerenciamento (maior que em computadores *dataflow*) e tem pouca aplicação prática. Por exemplo, arquiteturas de redução.

Conforme visto nesta Seção, a classificação de Duncan engloba a maioria dos tipos de arquiteturas existentes, tendo atingido os itens abrangência e extensibilidade das características ideais de uma classificação citadas por Ben-Dyke [BEN93], nos quais a classificação de Flynn falha. Além disso, uma característica importante da classificação proposta por Duncan é a presença dos termos MIMD e SIMD, propostos por Flynn e largamente aceitos.

3.3. Topologias de Comunicação

Uma topologia de comunicação trata-se da maneira como se estrutura a ligação entre os vários processadores e entre os processadores e a memória (rede de comunicação). Esse é um aspecto muito importante de uma arquitetura paralela, visto que a comunicação entre processos influencia diretamente o desempenho de um programa paralelo [ALM94].

Alguns aspectos que devem ser considerados quando analisa-se o desempenho de uma rede de comunicação são:

1.Latência (tempo de trânsito de uma mensagem pela rede de comunicação);

2.*Bandwidth* (quantidade de tráfego de mensagens que a rede de comunicação suporta);

3.Conectividade (quantidade de vizinhos que cada processador possui);

4.Confiabilidade (conseguida, por exemplo, através de caminhos redundantes).

Topologias de comunicação podem ser organizadas de duas maneiras: estaticamente ou dinamicamente.

Topologias estáticas se caracterizam por conectarem os elementos de processamento diretamente entre si, enquanto topologias dinâmicas utilizam-se de chaves na ligação entre os elementos de processamento. Enquanto que em topologias estáticas há necessidade de determinar explicitamente o caminho a ser seguido por uma mensagem trocada entre dois processadores (bibliotecas de comunicação, em nível de *software*, podem esconder este roteamento), em topologias dinâmicas, somente é necessário o envio de uma mensagem para uma chave, que esta responsabiliza-se por rotear a mensagem até o seu destino. Por exemplo, na Figura 3.7, uma mensagem trocada entre os processadores P1 e P4 necessitaria ser explicitamente transmitida para P2, para então alcançar P4. Este procedimento caracteriza uma topologia estática. Já, no caso dinâmico, apenas se envia a mensagem para a rede (que no caso trata-se de uma estruturas de chaves), que esta será roteada automaticamente [ALM94].

Topologias estáticas não são reconfiguráveis, de maneira que a sua estrutura de conexão de processadores não pode ser modificada após a sua construção. Apresentam menor custo e maior simplicidade para o seu projeto, visto que a conexão de seus elementos é regular, mas essa regularidade implica na utilização destas topologias para aplicações com padrões simples e previsíveis de comunicação. Essas topologias têm sido utilizadas com eficiência no projeto de computadores de propósito específico. Porém, é difícil a construção de um multi-processador de propósito geral, que se baseie numa topologia estática, visto que, para aplicações com padrões de comunicação variáveis, é interessante a utilização de topologias que adequem a sua estrutura de conexão de acordo com a carga de trabalho nos vários processadores. Nesses casos, é mais útil a utilização de topologias dinâmicas ou topologias estáticas que, em nível de *software*, possibilitem padrões de comunicação dinâmicos.

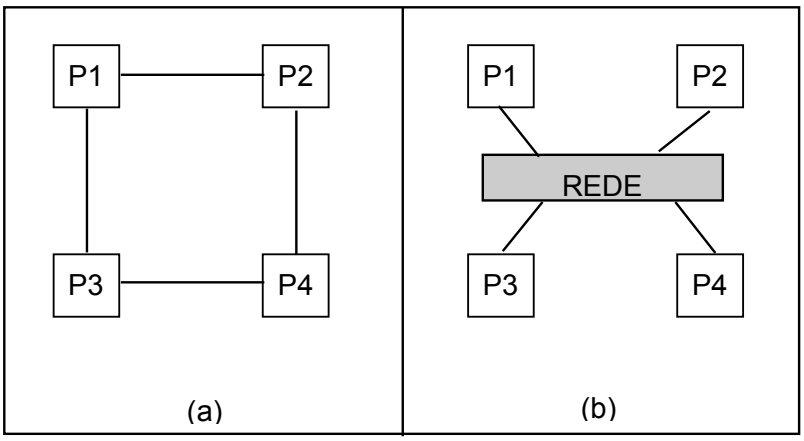

Figura 3.7 - Topologias. a) estática b) dinâmica

Apesar de topologias dinâmicas apresentarem, em geral, melhor desempenho, são mais caras e de mais difícil construção. E também, geralmente impõe restrições a futuros ampliações, isto é, o acréscimo de novos elementos de processamento a topologia pode diminuir drasticamente o desempenho.

Topologias estáticas são muito utilizadas para computadores SIMD e MIMD com memória distribuída, e exemplos para a sua organização são: linear, anel, estrela, árvore, hipercubo, entre outros. Por outro lado, topologias dinâmicas são muito utilizadas em arquiteturas MIMD com memória centralizada, em virtude de possibilitarem demora constante para todos os processadores no acesso à memória, e exemplos de sua utilização são: barramento, redes de chaves multifásicas, entre outros.

Dentre as arquiteturas em geral, o modelo MIMD tem se destacado (principalmente o modelo de memória distribuída), devido à sua flexibilidade e por representar uma boa opção para o desenvolvimento de algoritmos paralelos de granulações média e grossa [ZAL91] [BLE94]. Assim, nas próximas seções serão discutidos alguns tópicos sobre a programação nesse tipo de arquitetura.

4. Programação Concorrente

Um programa seqüencial é composto por um conjunto de instruções que são executadas seqüencialmente, sendo que a execução dessas instruções é denominada um **processo**. Um programa concorrente especifica dois ou mais programas seqüenciais que podem ser executados concorrentemente como **processos paralelos** [AND83].

A programação concorrente existe para que se forneçam ferramentas para a construção de programas paralelos, de maneira que se consiga melhor desempenho e melhor utilização do *hardware* paralelo disponível. A computação paralela apresenta muitas vantagens em relação à computação seqüencial, como foi exposto no Capítulo 1, e todas essas vantagens podem ser citadas como pontos de incentivo para o uso da programação concorrente.

Um programa seqüencial é constituído basicamente de um conjunto de construções já bem dominadas pelo programador em geral, como por exemplo, atribuições, comandos de decisão (if... then... else), laços (for... do), entre outras. Um programa concorrente, além dessas primitivas básicas, necessita de novas construções que o permitam tratar aspectos decorrentes da execução paralela dos vários processos.

Segundo Almasi [ALM94], para a execução de programas paralelos, deve haver meios de:

- Definir um conjunto de tarefas a serem executadas paralelamente;
- Ativar e encerrar a execução dessas tarefas;
- Coordenar e especificar a interação entre essas tarefas.

A fase de definição da organização das tarefas paralelas é de extrema importância, pois o ganho de desempenho adquirido da paralelização depende fortemente da melhor configuração das tarefas a serem executadas concorrentemente. Alguns aspectos que devem ser considerados no desenvolvimento de algoritmos paralelos são discutidos na Seção 4.1.

Definido o algoritmo, é necessário um conjunto de ferramentas para que o programador possa representar a concorrência, definindo quais partes do código serão executadas seqüencialmente e quais serão paralelas. As construções para definir, ativar e encerrar a execução de tarefas concorrentes são discutidas na Seção 4.2.

Além disso, processos cooperando para a resolução de determinado problema devem comunicar-se e sincronizar-se, a fim de que haja interação entre eles. A maneira como se implementa a comunicação entre processos depende da arquitetura onde se executa a aplicação: em caso de memória centralizada, utilizam-se variáveis

compartilhadas; em caso de memória distribuída, é utilizada troca de mensagens. As formas de comunicação/sincronização entre processos são discutidos na Seção 4.3.

4.1. Projeto de um Algoritmo Paralelo

Há, pelo menos, três maneiras de se construir um algoritmo paralelo [QUI87]:

- Detectar e explorar algum paralelismo inerente a um algoritmo seqüencial existente: essa abordagem é muito utilizada, apesar de apresentar baixo *speedup*, visto que não há necessidade de nova análise do algoritmo;
- Criar um algoritmo paralelo novo: possibilita melhor desempenho, necessitando porém de reestruturação completa do algoritmo;
- Adaptar outro algoritmo paralelo que resolva problema similar: nesse caso, têm-se bom desempenho, exigindo menor trabalho do programador em relação à construção completa do algoritmo.

Três aspectos importantes devem ser considerados quando se projeta um algoritmo paralelo. Primeiro, o processo de escolha da abordagem a ser seguida, entre as três citadas acima, deve ser feita de maneira cuidadosa, para que se consiga a melhor eficiência, levando em conta o tempo de escrita do algoritmo e o desempenho obtido.

Além disso, deve-se pesar o custo da comunicação entre processos em relação ao tempo de execução efetiva, visto que operações de comunicação geram uma sobrecarga que pode degradar o desempenho, tornando o algoritmo menos eficiente que o seqüencial.

Por último, deve-se considerar a arquitetura na qual se executará o algoritmo, visto que a sua eficiência pode variar de maneira drástica de acordo com o tipo de arquitetura.

Alguns exemplos de algoritmos paralelos são:

1. Seja a expressão (a * b + c * d ^ 2) + (g + f * h). Uma possível organização da execução de cada uma das operações, respeitando-se a precedência, é mostrada na Figura 4.1 (a), onde cada tarefa (A, B, C, ..., G) executa uma operação matemática entre dois números. Pode-se organizar a execução de cada tarefa através do diagrama apresentado em b. O sinal || representa tarefas sendo executadas em paralelo.

   Supondo-se que se possua três processadores (P1, P2 e P3), pode-se representar a distribuição das tarefas entre eles através de um diagrama *processadores X tempo*, apresentado na Figura 4.2. Para a construção do diagrama, deve-se respeitar a dependência entre as diversas tarefas. Por exemplo, a tarefa D necessita

de dados das tarefas A e B e, portanto, necessita ser iniciada posteriormente ao término delas.

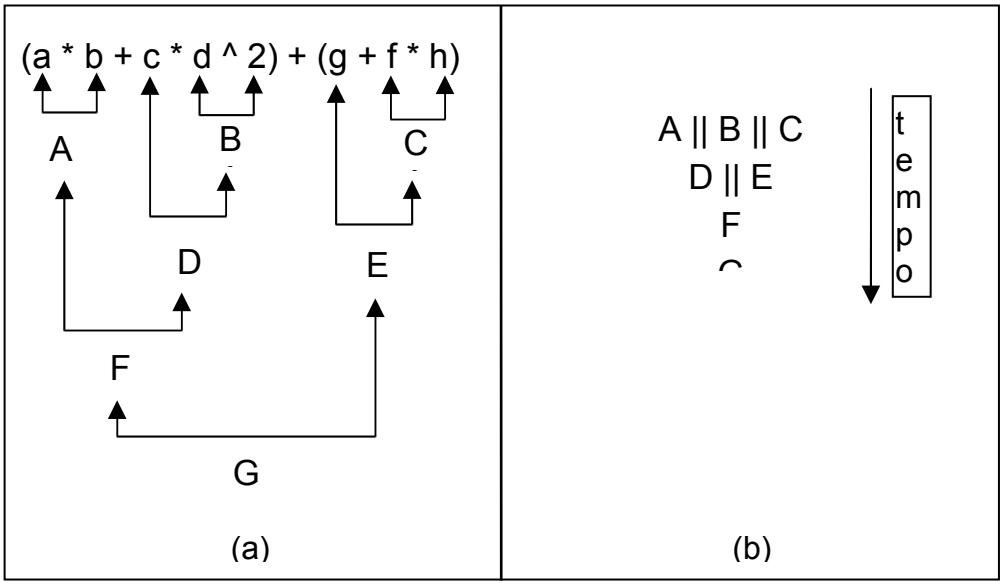

Figura 4.1 - Organização para execução concorrente

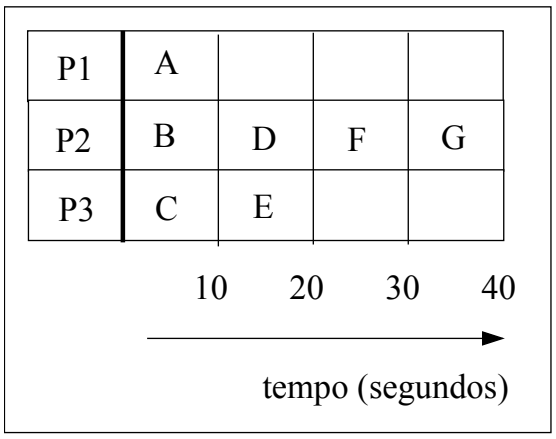

Figura 4.2 - Distribuição das tarefas

Considerando os tempos de execução de todas as tarefas igual a 10 segundos (tA, tB, tC ..,, tG = 10 segundos), têm-se:

Tempo sequencial (tseq) = 70 segundos

Tempo paralelo (tpar) = 40 segundos

$$Speedup = \frac{40}{70} = 0,57$$

Por outro lado, sejam 20,.10 e 5 os tempos da exponenciação (^), multiplicação (*) e adição (+) respectivamente, então o diagrama seria o da Figura 4.3.

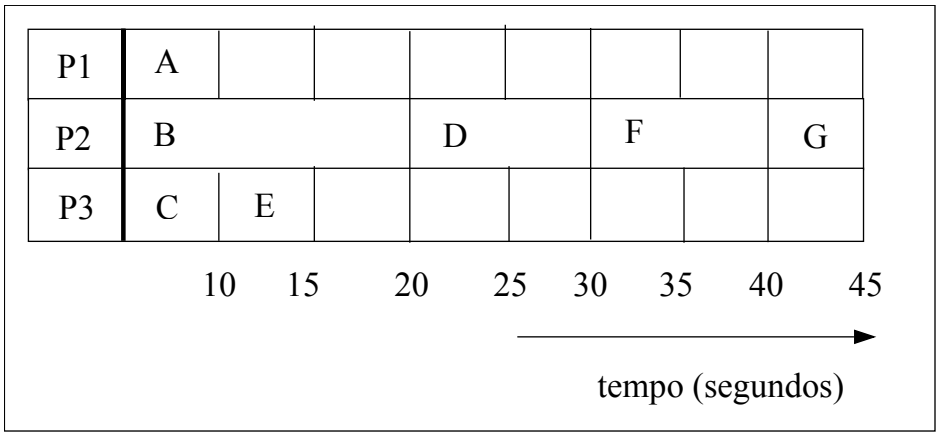

Figura 4.3 - Distribuição das tarefas por tempo de execução das operações

Nesse caso, o *speedup* seria de 45/70, ou 0,64. Uma melhor configuração de tarefas, que geraria um melhor balanceamento e uma menor complexidade, seria a transferência das tarefas C e E para serem executadas em P1, visto que não modificaria o tempo de execução e possibilitaria a utilização de menos processadores, como mostra a Figura 4.4.

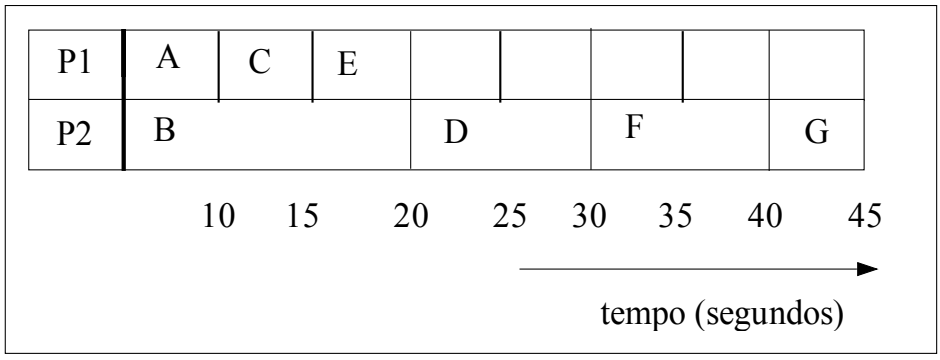

Figura 4.4 - Novo arranjo na distribuição das tarefas

2. Seja o loop:

for i = 1 *to* n
$a_i = 0$

Pode-se paralelizá-lo completamente, iniciando todas as *i* variáveis ao mesmo tempo, tendo-se: $a_1 = 0 \,||\, a_2 = 0 \,||\, a_3 = 0 \,||\, ... \,||\, a_n = 0$.

3. Seja o seguinte bloco de comandos, dentro de uma instrução *for*:

```
for i=1 to n
        xi = 0;
        ai = bi + 1.
```

Pode-se paralelizar em primeiro nível (dentro de cada uma das intruções, como no exemplo acima) e também podem-se executar as duas instruções em paralelo, tendo-se: $(x_1 = 0 \,||\, x_2 = 0 \,||\, ... \,||\, x_n = 0) \,||\, (a_1 = b_1 + 1 \,||\, a_2 = b_2 + 1 \,||\, ... \,||\, a_3 = b_3 + 1)$.

4. Seja o seguinte pedaço de código:

```
soma = 0;
for i = 1 to n
        soma = soma + ai
```

Este trecho não é paralelizável, visto que a toda iteração da instrução *for* depende da iteração anterior.

5. Como foi exposto, um aspecto muito importante no desenvolvimento de um algoritmo paralelo trata-se da relação entre tempo de comunicação e tempo de execução. Por exemplo, sejam dois algoritmos para soma paralela de vários elementos a*i* (a=1, 2, 3, ...,n).

   Executando-se o algoritmo em *n*/2 processadores (A_1, A_2, ... $A_{n/2}$) , tem-se o esquema da Figura 4.5.

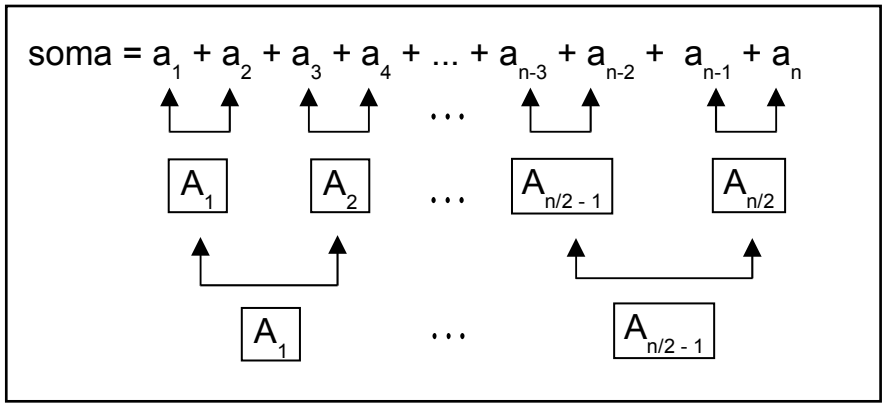

Figura 4.5 - Esquema de paralelização

Neste algoritmo, o número de elementos a serem somados cai pela metade a cada iteração (supondo que, em passos onde o número de elementos é ímpar, o último

processador executa uma soma de três números). Então, após o passo 1, restarão n/2 elementos a serem somados, no passo 2, n/4, e assim por diante.

Além disso, antes de cada passo *k*, um processador deve receber um dado de seu vizinho a fim de executar a sua soma. Por exemplo, o processador A_1, após o primeiro passo, deve receber o valor da soma feita no processador A_2. Um exemplo da estrutura de comunicação é mostrado na Figura 4.6. Em cada passo, todos os processadores trabalham paralelamente entre si e todas as operações de comunicação, representadas por setas, também serão feitas em paralelo.

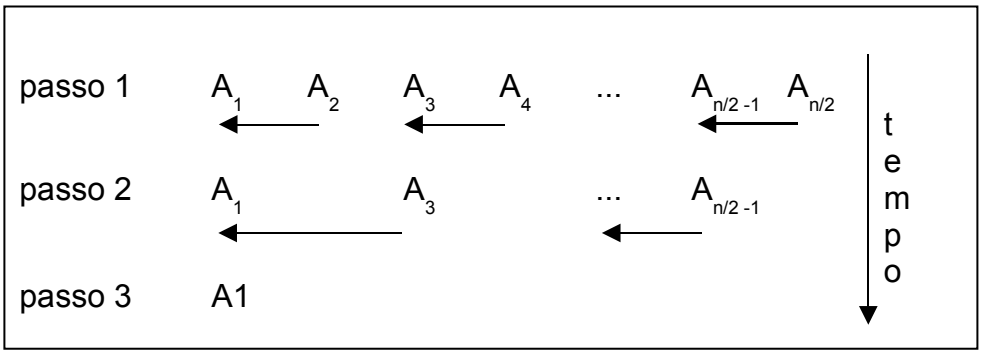

Figura 4.6 - Estrutura de comunicação

Para um exemplo de um seqüência de 100 números, o tempo total de execução seria calculado da seguinte maneira:

$A_1 \,||\, ... \,||\, A_{50}$

$A_1 \,||\, ... \,||\, A_{25}$

$A_1 \,||\, ... \,||\, A_{13}$

$A_1 \,||\, ... \,||\, A_7$

$A_1 \,||\, ... \,||\, A_2$

Soma

Seja t_o o tempo de execução de uma operação de soma, e tc o tempo de comunicação entre processadores, então o algoritmo número 1 teria um tempo de execução igual a $6t_o + 5t_c$.

Considere uma variação do algoritmo 1, utilizando 2 processadores, com a configuração mostrada na Figura 4.7. Apenas uma operação de comunicação será

necessária, de A_2 para A_1. Então, para a mesma seqüência de 100 elementos, o tempo de execução será de $50t_o + t_c$.

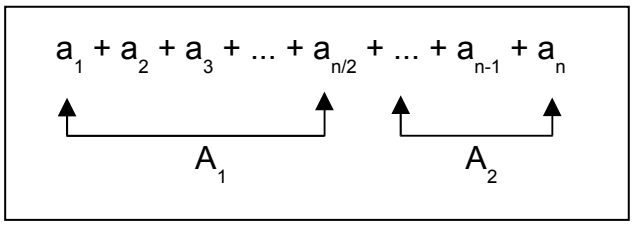

Figura 4.7 - Nova configuração

Qual algoritmo é melhor? Depende da plataforma onde estão sendo executados ambos os algoritmos e da relação apresentada entre tempo de execução e tempo de comunicação. Em plataformas adequadas à granulação fina, com pouca sobrecarga nas operações de comunicação, os valores de t_o têm peso muito grande no resultado da equação, de maneira que o algoritmo 1 é mais eficiente. Caso contrário, se a sobrecarga de comunicação é muito significativa, então o valor de t_c determina o valor da equação e o algoritmo 2 é mais eficiente. Conclui-se então, que um algoritmo paralelo deve ser projetado cuidadosamente, levando-se em consideração a relação *execução X comunicação* e também a plataforma a ser utilizada.

4.2 - Desenvolvimento de Algoritmos Paralelos

No desenvolvimento de algoritmos paralelos, várias etapas, não utilizadas na programação seqüencial, devem ser seguidas. Em resumo, pode-se dividir o processo de desenvolvimento de algoritmos paralelos em 4 etapas:

1. Identificação do paralelismo inerente ao problema, que consiste no estudo do problema a fim de se determinar possiveis eventos paralelizáveis;
2. Organização do trabalho, englobando a escolha do estilo de paralelismo que será utilizado (os estilos de paralelismo são apresentados na próxima Seção) e a divisão de tarefas entre os diversos processadores a fim de maximizar o ganho de desempenho;
3. Desenvolvimento do algoritmo, utilizando ferramentas para expressar o paralelismo e para comunicação e sincronismo entre os processos;
4. Implementação através da utilização de um método de programação adequado. Esta etapa engloba a escolha de uma linguagem de programação adequada, tratamento de possíveis *deadlocks* e mapeamento de tarefas.

4.3 - Estilos de Paralelismo

Um algoritmo paralelo pode ser organizado de várias maneiras, de acordo com o tipo de plataforma onde será executado. Genericamente, os modelos computacionais paralelos são divididos em dois tipos: memória compartilhada e troca de mensagens.

O modelo paralelo baseado em memória compartilhada divide-se de acordo com o tipo de acesso à memória. No modo síncrono (ou *PRAM - Parallel Random Acess Memory* - Acesso Aleatório Paralelo à Memória), os processadores atuam sincronamente no acesso à memória, enquanto no modo assíncrono esse sincronismo de acesso não existe.

O modelo paralelo, baseado em troca de mensagens, é dividido em três estilos, que são: paralelismo geométrico, paralelismo algorítmico ou *processor farming*.

Paralelismo geométrico

Também conhecido como paralelismo de resultados ou paralelismo de dados (*Data Parallelism*), caracteriza-se pela divisão do conjunto de dados a serem trabalhados igualmente entre todos os processadores. Dessa maneira, cada processador executa uma cópia do programa completo, porém em um subconjunto de dados. Este é considerado o modo mais fácil de desenvolvimento de algoritmos paralelos e é intensamente utilizado em computadores massivamente paralelos[2].

Considere, por exemplo, que se deseja construir uma parede de tijolos, onde cada um dos pedreiros é um processador. No paralelismo geométrico, o muro é dividido em partes iguais, e cada uma dessas partes são distribuídas para um pedreiro (T1, T2, T3 e T4), como mostra a Figura 4.8.

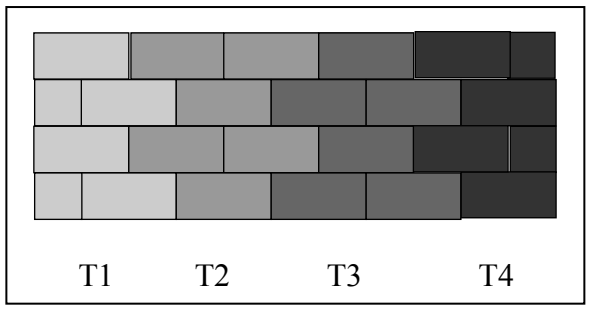

Figura 4.8 - Paralelismo geométrico

Paralelismo “Processor Farm”

Caracteriza-se pela existência de um processador “mestre” que supervisiona um grupo de processadores “escravos”, cada um processando assincronamente tarefas submetidas a ele pelo processador “mestre”. Este modelo também é conhecido como

[2]Computadores com muitos processadores (geralmente, mais de 1000 processadores)

paralelismo pela pauta, visto que é definido um conjunto de tarefas (pauta) e a partir daí são distribuídas as tarefas pelo processador mestre.

No exemplo da parede de tijolos, uma construção baseada em "*processor farm*" (Figura 4.9) pode ser organizada criando-se três tarefas básicas: pegar o cimento, pegar o tijolo e colocar no próximo lugar disponível. Essas três tarefas são submetidas a cada um dos pedreiros pelo mestre de obras (processador mestre), que ao colocar o tijolo no lugar apropriado, está disponível para novas tarefas ordenadas pelo mestre de obras.

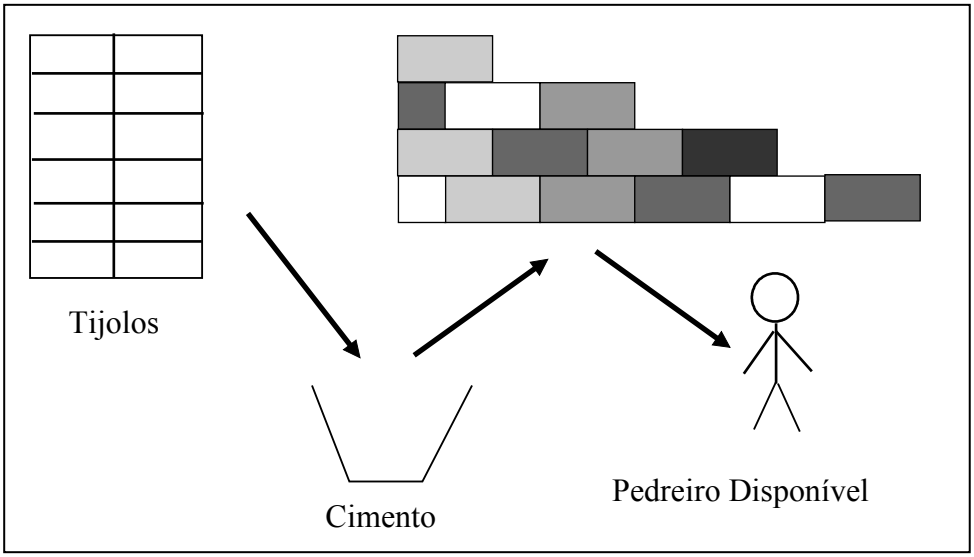

Figura 4.9 - Paralelismo "*Processor Farm*"

O estilo *"processor farm"* possui várias vantagens, como por exemplo: facilidade de ampliação do sistema, o que pode ser conseguido através do aumento de trabalhadores; facilidade de programação; balanceamento de carga mais natural, visto que as tarefas vão sendo submetidas aos processadores de acordo com a disponibilidade. Como desvantagens, podem ser citados a sobrecarga de comunicação e a possibilidade de gargalo no processador "mestre".

Paralelismo *Pipeline*

Também conhecido por paralelismo especialista ou algorítmico, caracteriza-se pela divisão de uma aplicação em várias tarefas específicas, que são distribuídas aos processadores de forma *pipeline.* Nesse caso, quando apenas uma unidade de dados atravessa o *pipeline* não há paralelismo.

Um mecanismo *pipeline* para a construção de uma parede de tijolos é a divisão da parede a ser construída em várias camadas horizontais, de maneira que cada pedreiro seja responsável pela construção de 1 camada. Dessa maneira, o início das camadas superiores dependem do término das primeiras camadas, como mostra a Figura 4.10.

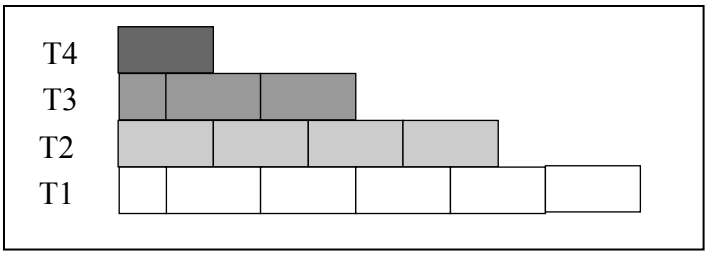

Figura 4.10 - Paralelismo "*Pipeline*"

Muitas vezes, este é o estilo mais natural para desenvolver paralelismo a partir de um programa seqüencial, sendo até possível a sua detecção por compiladores e analisadores de código fonte. Porém, algumas desvantagens que podem ser citadas são: pouca flexibilidade, visto que modificações no algoritmo podem requerer mudanças drásticas na rede de processadores; balanceamento de carga deve ser feito de maneira cuidadosa, a fim de que subtarefas lentas não tornem a execução de todo o *pipeline* lenta; a relação entre comunicação e execução deve ser baixa, a fim de que a sobrecarga de comunicação não torne o algoritmo ineficiente.

Durante o desenvolvimento de algoritmo paralelo, deve-se procurar escolher o estilo de paralelismo mais natural ao problema. Desenvolvido o algoritmo utilizando o método de programação mais natural ao estilo, pode-se conseguir resultados não satisfatórios em relação ao desempenho. Deve-se então transpor o algoritmo para um estilo de paralelismo mais eficiente. Podem-se utilizar métodos definidos na literatura para facilitar essa transição entre estilos de paralelismo.

Estendendo esta discussão para um problemas mais concreto, suponha uma multiplicação de matrizes **A(*n* X *k*) * B(*k* X *m*)**. Neste caso, multiplicam-se duas matrizes A e B, de dimensões n,k e k,m respectivamente. Serão citados exemplos de organização para a execução das tarefas paralelas baseado nas características próprias de cada estilo.

Na abordagem geométrica, pode-se determinar que cada elemento da matriz resultante seja determinado por um processador. Então, é possível a obtenção de um alto grau de paralelismo, com grande sobrecarga de comunicação em virtude da fina granulação apresentada por essa solução.

Na abordagem *processor farm*, o processador mestre envia aos escravos ociosos a próxima posição da matriz produto a ser calculada. Os processadores escravo, por outro lado, enviam ao processador mestre o resultado, e tornam-se disponível para calcular um novo elemento da matriz resultante. Esta abordagem apresenta flexibilidade quanto ao número de processadores e balanceamento automático de carga. Por outro lado, a possibilidade de gargalo no processador mestre pode ser um problema.

Na abordagem *pipeline*, divide-se a operação de obtenção de um elemento da matriz resultado em vários estágios, cada um designado a um processador. Os estágios podem ser, por exemplo: entrada de dados, multiplicação, soma e saída do

resultado. Essa organização de tarefas apresenta pouca flexibilidade e tempo de latência significativo para a obtenção de um elemento da matriz resultado.

4.4. Ativação de Processos Paralelos

Várias construções para a ativação e término de processos concorrentes são discutidas na literatura, apresentando características e finalidades distintas [ALM94] [AND83] [KIR89] [QUI87] [SNO92]. Modelos representativos dessas construções são apresentados a seguir.

Corotinas

Corotinas são subrotinas que possuem um modo de transferência de controle não hierárquico. Uma subrotina comum, ativada através de uma chamada *call subrotina*, ao executar um comando *return*, retorna o controle ao módulo de programa que a ativou e termina a sua execução. Além disso, toda vez que esta é ativada, será executada desde o seu início.

Corotinas transferem controle entre si de maneira livre, através do comando *resume corotina*. E sempre que são ativadas, executam a partir do ponto onde foi executado a última chamada à *resume* (Figura 4.11). Cada corotina pode ser vista como implementando um processo, e elas são executadas intercaladamente. Sempre existe apenas uma corotina ativa em cada instante, o que implica na adequação desta estrutura para a organização de programas concorrentes que compartilhem uma única CPU.

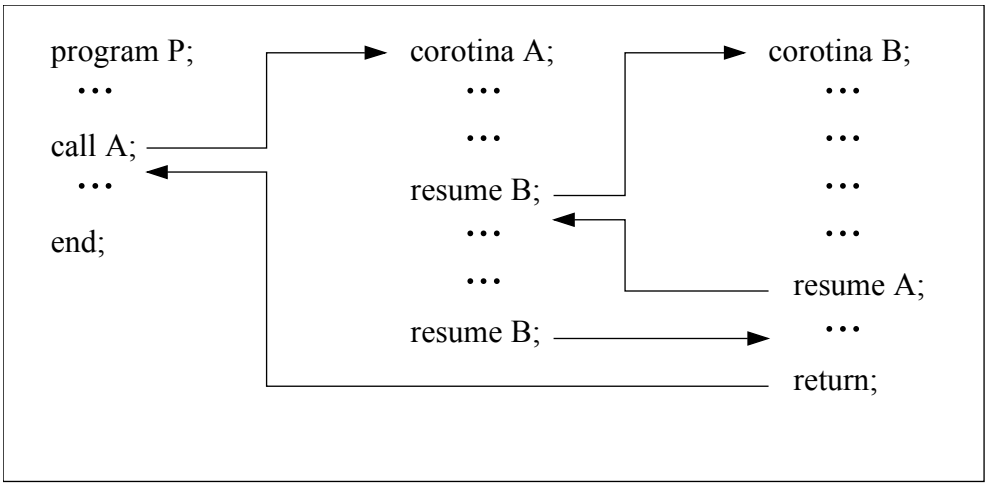

Figura 4.11 - Exemplo da utilização de Corotinas

Fork/Join

O comando *fork* implica que um determinado conjunto de instruções (processo filho) deve iniciar a sua execução em paralelo com o processo que o executa (processo pai). O comando *join* é utilizado para a sincronização do processo pai com os filhos gerados. Um exemplo de sintaxe para os comandos *fork/join* é:

- ***Fork*** *end*: Executa a partir do endereço *end*, concorrentemente ao processo que executa a chamada *fork*;
- ***Join*** *num, end1, end2*: Decrementa a variável *num*, que contém o número de processos que devem sincronizar-se. Se *num*=0, execute a partir de *end1*. Caso contrário, execute a partir de *end2* (geralmente é um comando *quit*, que termina a execução).

A utilização de *fork/join* é um meio poderoso e flexível de se especificar o processamento concorrente (Figura 4.12). Porém, programas escritos utilizando tais comandos devem ser escritos de maneira disciplinada, visto que a organização do código fonte obtido é desestruturada. Com o objetivo de proporcionar maior estruturação, à custa de perda de flexibilidade, foram propostos alguns novos modelos que são apresentados à seguir, como *cobegin/coend* e *doall*.

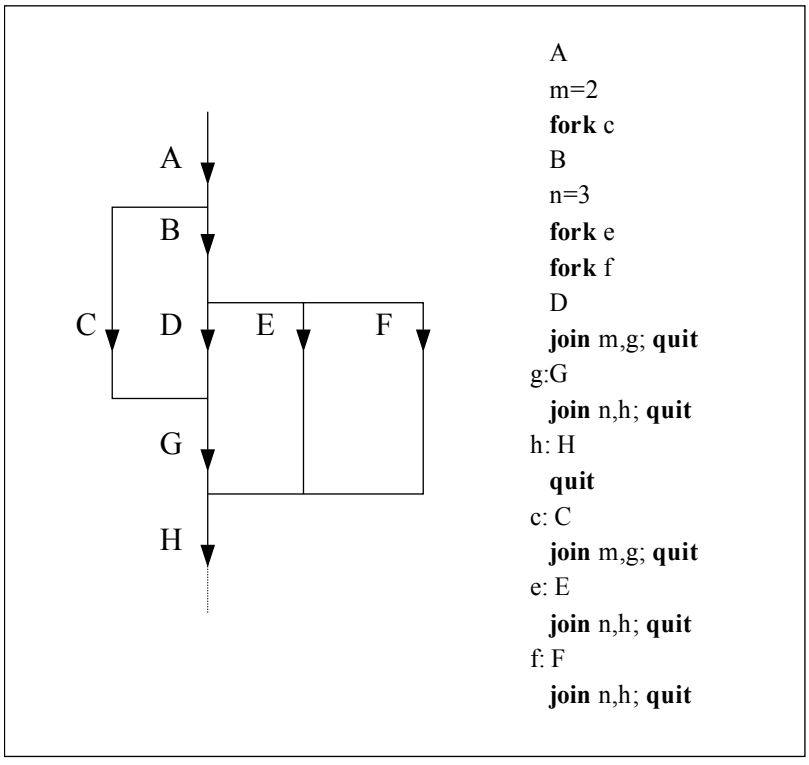

Figura 4.12 - Exemplo da utilização de *Fork/Join*

Cobegin/Coend

Também chamados de *parbegin/parend*, estes comandos oferecem uma maneira estruturada de ativação de um conjunto de instruções que devem ser executadas concorrentemente (Figura 4.13). A execução concorrente das declarações S1, S2, ..., Sn pode ser ativada através da estrutura:

Cobegin S1 // S2 // S3 // ... // Sn ***Coend***

O processo pai será bloqueado até que S1, S2, ..., Sn estejam terminadas.

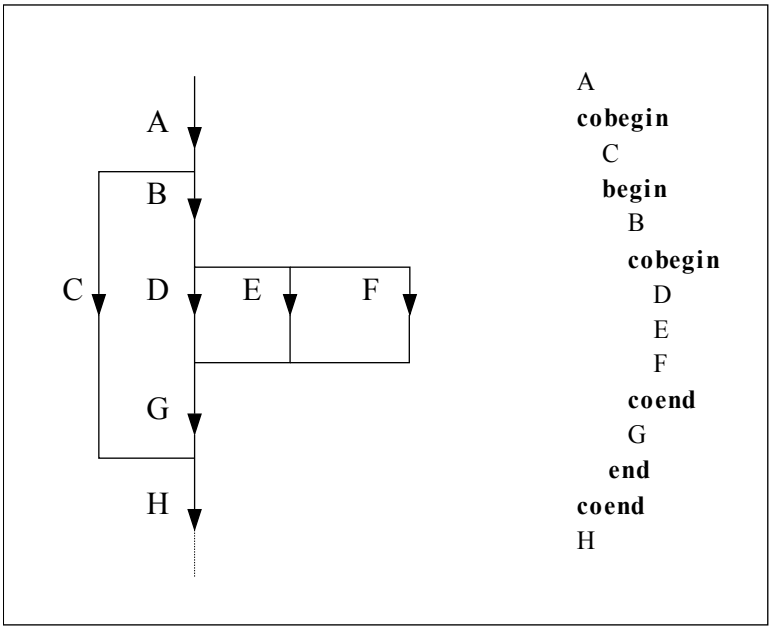

Figura 4.13 - Exemplo da utilização de *Cobegin/Coend*

Doall

Este comando pode ser visto como um comando *cobegin/coend* onde as instruções executadas em paralelo são as diversas instâncias de um bloco de comandos dentro de um comando de *loop* (Figura 4.14). Alguns comandos de função semelhante são: *forall, pardo* e *doacross*.

A escolha do método de ativação de processos concorrentes deve ser feita de acordo com os objetivos do programador. Corotinas são utilizadas para ativação de processos concorrentes, *fork/join* e *cobegin/coend* para a ativação de processos paralelos e *doall* para a ativação paralela de instâncias de *loops.* Em relação ao contraste flexibilidade/estruturação, *fork/join* oferece um mecanismo flexível porém desestruturado, enquanto *cobegin/coend* e *doall* apresentam maior estruturação, o que diminui a flexibilidade.

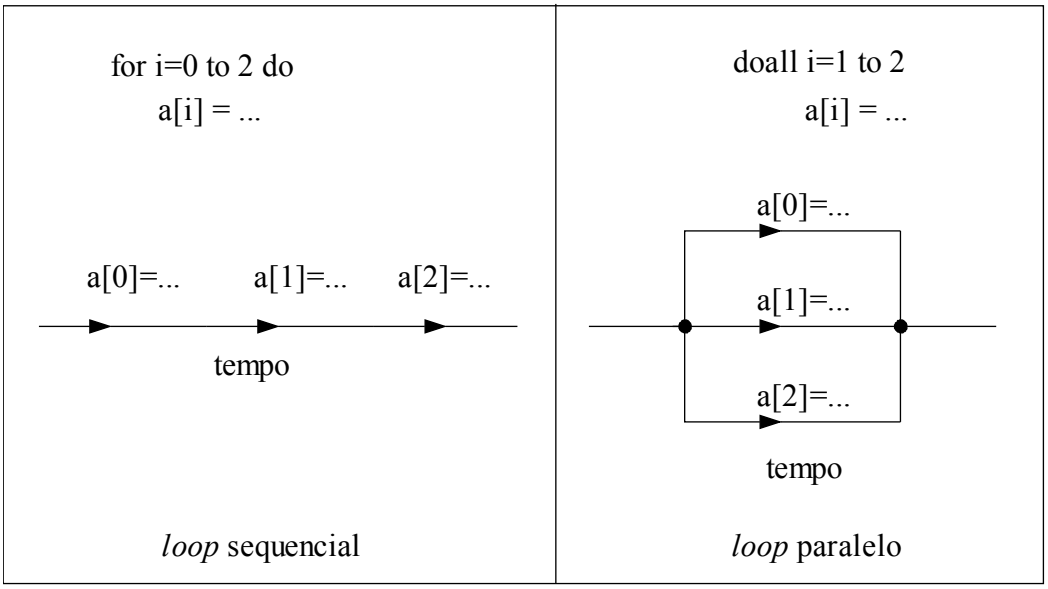

Figura 4.14 - Exemplo da utilização de *Doall*

4.5. Comunicação e Sincronismo

Comunicação é necessária para que processos interagindo na resolução de determinada aplicação troquem informações. E quando há comunicação, devem existir operações de sincronização, para fornecer controle de acesso e controle de seqüência.

Controle de seqüência (também chamado de sincronização condicional ou sincronização de atividades) é utilizado para que se determine uma ordem na qual os processos (ou partes deles) devem ser executados. Controle de acesso é necessário quando há competição entre processos para a manipulação de algum recurso. Deve-se garantir que acessos concorrentes a esses recursos sejam controlados, para que se mantenha a consistência [ALM94] [AND83] [KIR89] [QUI87] [SNO92].

Comunicação e Sincronismo em Memória Centralizada

Comunicação e sincronismo (controle de acesso e seqüência) em memória centralizada são implementados através da utilização de variáveis compartilhadas entre os diversos processos concorrentes.

O controle de acesso é geralmente implementado através de exclusão mútua. São definidas seções críticas, isto é, conjuntos de instruções que devem ser executadas de maneira mutuamente exclusiva, de maneira que os processos não sofram interferência durante a execução dessas seções.

Para a implementação de exclusão mútua e controle de seqüência utilizando-se de variáveis compartilhadas, vários métodos podem ser utilizados, entre eles: *busy-waiting*, semáforos e monitores [ALM94] [AND83] [KIR89] [QUI87] [SNO92].

Busy-waiting: utilizando-se, por exemplo, uma variável compartilhada cujo valor pode ser manipulado (modificado e testado/*test-and-set*) através de uma primitiva indivisível, cria-se um modo simples para a sincronização de processos concorrentes. Um processo querendo entrar em uma região crítica deve executar esta primitiva continuamente até conseguir permissão de entrada. São gastos ciclos de CPU enquanto se está testando a variável, caracterizando este método de sincronização como um exemplo de *busy-waiting*, ou espera ocupada.

Busy-waiting possui várias desvantagens, como por exemplo, o gasto supérfluo de CPU. Além disso, programas utilizando tais primitivas são difíceis de se entender, depurar e provar que estão corretos, em virtude de serem implementados em baixo nível.

Semáforos: um semáforo é uma variável compartilhada inteira e não negativa sobre a qual estão definidas duas operações atômicas (indivisíveis): **p** e **v** (também chamados de **down** e **up**, respectivamente). Dado um semáforo *s*, implementa-se **p** e **v** como:

```
p(s):  Se (s = 0)                          v(s):  s = s + 1
         então bloqueia-se o processo
         senão  s = s - 1
```

Semáforos oferecem um meio de sincronização de nível mais elevado do que *busy-waiting*, além de evitar o desperdício de CPU. Um exemplo de sua utilização, é a implementação de exclusão mútua. Considere o semáforo s, iniciado com o valor 1. Então, a seqüência

p(s); *região crítica*; **v(s)**;

garante que apenas um único processo esteja executando esta região crítica em um determinado instante.

Monitores: uma desvantagem de semáforos é o fato de serem pouco estruturados, o que pode levar a erros. Monitores oferecem uma maneira estruturada para a implementação de exclusão mútua.

Um monitor consiste de variáveis representando o estado de algum recurso compartilhado e procedimentos que implementam operações sobre esses recursos. Essas variáveis podem ser obtidas (ter-se acesso a elas) somente pelos procedimentos internos a cada monitor, e a execução desses procedimentos é feita de maneira mutuamente exclusiva.

Comunicação e Sincronismo em Memória Distribuída

Comunicação e sincronismo em arquiteturas de memória distribuída devem ser implementados através de troca de mensagens entre processos [ALM94] [AND83] [KIR89] [QUI87] [SNO92]. Uma operação de comunicação via mensagens, de maneira

genérica, é realizada pela utilização das primitivas *send/receive* (envia/recebe), cujas sintaxes, por exemplo, podem ser:

Send mensagem ***to*** processo_destino

Receive mensagem ***from*** processo_fonte

Uma transferência de mensagem pode ser realizada de duas maneiras. Uma operação bloqueante (síncrona - Figura 4.15 (a)) implica que o processo que transmite a mensagem (transmissor) é bloqueado até que receba uma confirmação de recebimento da mensagem pelo processo receptor. Caso contrário, tem-se uma operação não bloqueante (assíncrona - Figura 4.15 (b)), isto é, o processo transmissor envia a mensagem (que deve ser armazenada em um *buffer*) e continua a sua execução.

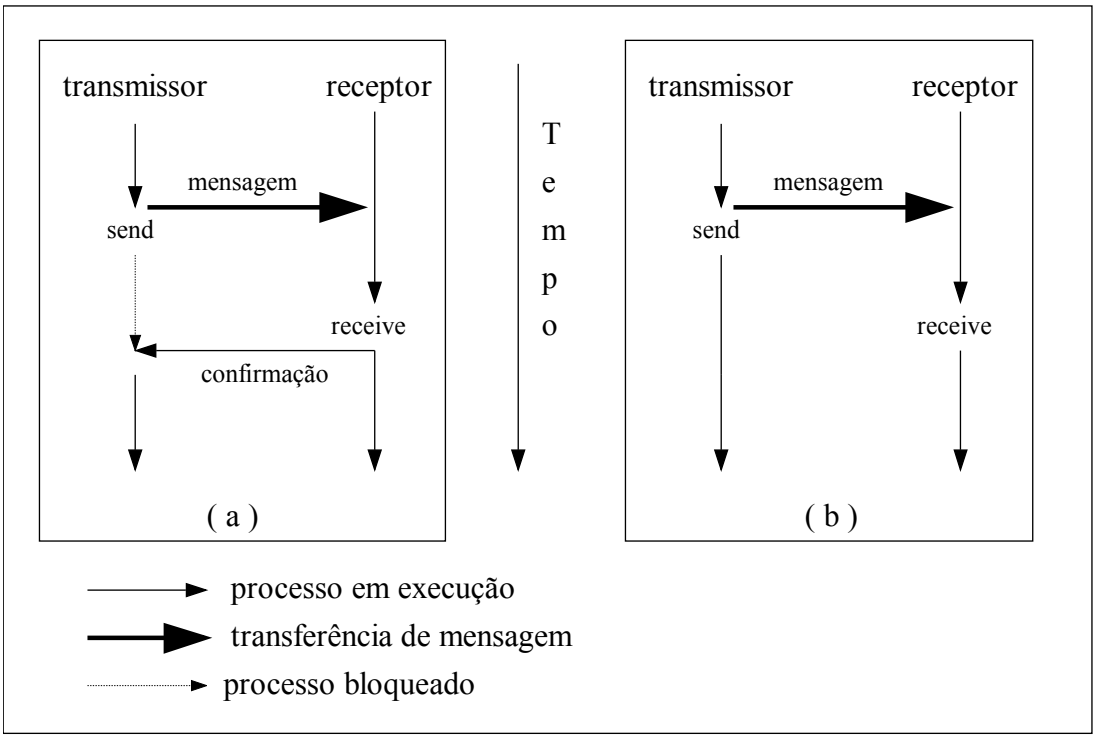

Figura 4.15 - *Send/Receive*. (a) Bloqueante (b) Não-Bloqueante

As primitivas *send/receive* podem ser organizadas para efetuar trocas de mensagens, gerando três mecanismos básicos: comunicação ponto a ponto, *rendezvous* e RPC.

Comunicação Ponto-a-Ponto: caracteriza-se pelo uso de uma operação *send/receive* bloqueante, de maneira que os processos se sincronizem (Figura 4.16 (a)). Tem-se então, comunicação unidirecional.

Rendezvouz:: estrutura de comunicação bidirecional que permite que um processo, por exemplo, comande a execução de um trecho de programa em outro processo. Isso é conseguido através do uso de dois conjuntos de operações *send/receive* bloqueantes (Figura 4.16 (b)).

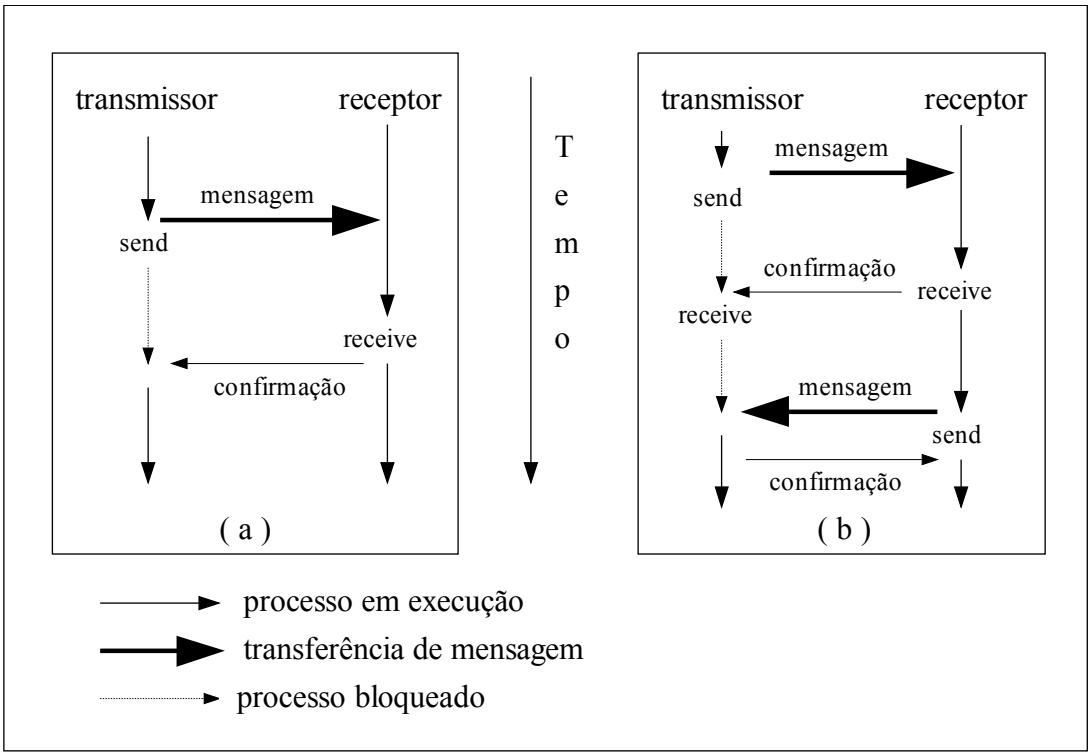

Figura 4.16 - Mecanismos de Comunicação. (a) Ponto-a-Ponto (b) *Rendezvouz*

RPC (**R**emote **P**rocedure **C**all **-** Chamada de Procedimento Remoto): caracteriza-se pela execução de um procedimento não local a um determinado processo, utilizando uma sintaxe semelhante às chamadas de procedimentos locais, sendo que o processo requisitador do serviço é bloqueado até que se obtenham os resultados desejados (Figura 4.17). O procedimento remoto é iniciado quando uma requisição de execução é recebida.

Os mecanismos de comunicação e sincronismo em memória compartilhada diferenciam-se pelo contraste entre estruturação e flexibilidade. Enquanto *busy-waiting* oferece flexibilidade e pouca estruturação, monitores são implementados em um nível mais alto, sendo mecanismos mais estruturados e menos flexíveis. Semáforos são mecanismos que apresentam um nível de estruturação e flexibilidade intermediário. Quanto aos mecanismos de memória distribuída, o que os diferencia é simplesmente a forma com que os comandos *send* e *receive* são utilizados.

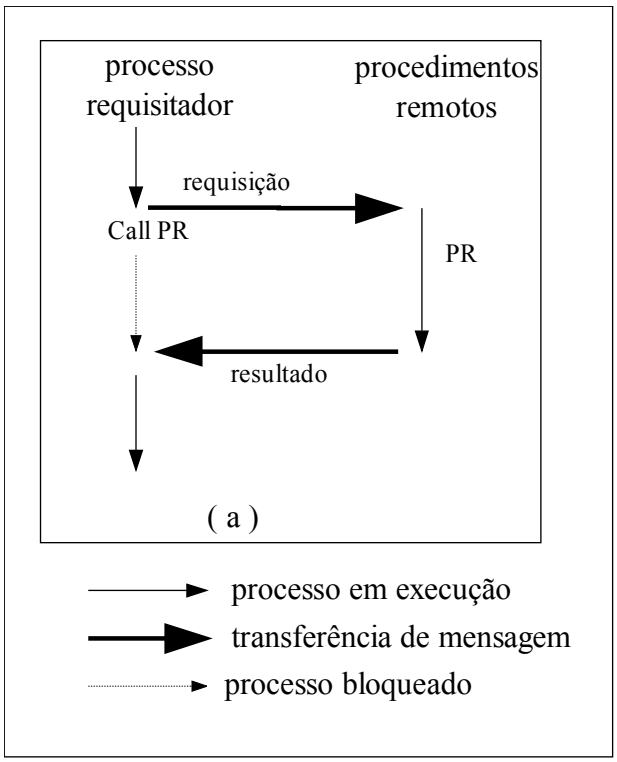

Figura 4.17 - RPC

4.6. Suporte para Programação Paralela

Conhecidos os mecanismos de ativação de processos concorrentes e de comunicação e sincronismo citados nas seções anteriores, uma decisão extremamente importante é a escolha do tipo de ferramenta (que implemente esses mecanismos) que será usada para a construção de um programa fonte que represente o algoritmo paralelo definido.

Vários aspectos devem ser levados em consideração nessa escolha. Entre eles, o tipo de aplicação, o tipo de usuário que utilizará a ferramenta e a arquitetura que executará os códigos gerados. Baseado no tipo de aplicação e na arquitetura, define-se a granulação desejada, que deve ser considerada também no processo de escolha. Segundo Almasi [ALM94], dois fatores principais devem ser considerados: o tempo de trabalho do programador (e aqui inclui-se o tempo de aprendizado da ferramenta), e o desempenho obtido.

Almasi [ALM94], relaciona três tipos de ferramentas para construção de programas paralelos: ambientes de paralelização automática, extensões paralelas para linguagens seriais e linguagens concorrentes.

Compiladores paralelizadores, que caracterizam ambientes de paralelização automática, são responsáveis por gerarem automaticamente versões paralelas de programas não paralelos. Tais compiladores exigem o mínimo de trabalho do usuário,

mas o desempenho obtido geralmente é modesto. Além disso, nem sempre esses compiladores são disponíveis, principalmente para sistemas de memória distribuída [BLE94].

Extensões paralelas são bibliotecas que contêm um conjunto de instruções que complementam linguagens seriais já existentes. Esses ambientes requerem algum trabalho do programador, mas ainda evitam a necessidade de aprendizagem de uma nova linguagem ou a completa rescrita do código fonte (no caso onde se paraleliza um programa já implementado). Geralmente, o desempenho obtido é superior ao obtido pelos compiladores paralelizadores. Existem extensões para arquiteturas de memória distribuída (MPI, PVM, P4) e para memória compartilhada (Pascal Concorrente). Dentro das extensões paralelas, é interessante citar os ambientes de passagem de mensagens, que são ambientes de programação paralela para memória distribuída portáteis, que permitem o transporte de programas paralelos entre diferentes arquiteturas (e sistemas distribuídos) de maneira transparente. Os ambientes de passagem de mensagens serão discutidos com maiores detalhes no próximo capítulo, o qual aborda a utilização da computação paralela sobre sistemas distribuídos.

O terceiro tipo de ferramenta relaciona as linguagens criadas especialmente para processamento concorrente, o que implica em tempo de aprendizagem de uma linguagem totalmente nova e rescrita total do código fonte. Essas ferramentas tendem a fornecer melhores desempenhos, de maneira que se compense a sobrecarga sobre o programador. Outra vantagem dessas linguagens é que geralmente elas possibilitam a construção de códigos bem estruturados, tornando fácil a identificação dos processos que estão executando em paralelo e a comunicação entre eles. Exemplos dessas linguagens são: *Occam* e *Ada*.

De maneira sucinta, a escolha de uma ferramenta para programação paralela, deve ser feita de acordo com os objetivos do programador. Compiladores paralelizadores oferecem desempenho ruim, porém com sobrecarga nula sobre o programador. Por outro lado, linguagens concorrentes oferecem melhor desempenho, mas oferecem uma sobrecarga considerável sobre o programador. Num patamar intermediário de desempenho e sobrecarga sobre o programador, situam-se as extensões paralelas.

5. Computação Paralela sobre Sistemas Distribuídos

O motivo principal para a criação e desenvolvimento de sistemas distribuídos foi, inicialmente, a necessidade de se compartilhar recursos, normalmente de alto custo e separados fisicamente. A computação paralela, entretanto, teve como objetivo fundamental aumentar o desempenho observado na implementação de problemas específicos [ZAL91]. Embora as duas áreas tenham surgido por razões diferentes, observou-se uma rápida convergência ao longo da última década, culminando atualmente com um forte inter-relacionamento caracterizado por muitos aspectos em comum.

Essa convergência deve-se principalmente ao avanço tecnológico e às linhas de pesquisa ocorridos a partir da década de 80. A computação paralela que empregava quase que na totalidade arquiteturas SIMD, começou também a utilizar (devido à sua versatilidade e alto desempenho) máquinas MIMD com memória distribuída. Essas máquinas passaram a contar com processadores de propósito geral, tornando possível a sua visualização como um conjunto de computadores autônomos (com Unidade de Controle, Unidade de Processamento e Memória), interligados por uma rede de comunicação.

Os sistemas distribuídos, por sua vez, foram cada vez mais aperfeiçoados, destacando-se como mudanças significativas: maior eficácia dos meios de comunicação e protocolos, e computadores com maior potência computacional.

Balanceamento de carga e perda de desempenho devido ao congestionamento no meio de comunicação são exemplos de problemas comuns às duas áreas e que são, ainda, amplamente pesquisados [ZAL91].

Devido às características em comum, vários trabalhos foram desenvolvidos com o objetivo de utilizar os sistemas distribuídos para computação paralela. A idéia básica é ter um grupo de computadores interligados, funcionando como os elementos de processamento de uma máquina paralela.

Apesar de possuir um meio de comunicação mais lento, tornando-se um fator que degrada rapidamente o desempenho, tais sistemas têm sido utilizados com sucesso para paralelizar aplicações que possuem granulação grossa e pouca necessidade de comunicação entre os processos [ZAL91].

Para a realização da computação paralela sobre sistemas distribuídos são utilizados ambientes de passagem de mensagens (ou interfaces de passagem de mensagens). Esses ambientes têm sido aperfeiçoados nos últimos anos para serem utilizados por uma grande quantidade de equipamentos diferentes, ganhando com isso popularidade e aceitação, visto que proporcionam o desenvolvimento de aplicações paralelas a um custo relativamente baixo em relação às máquinas paralelas [BEG94].

5.1. Ambientes de Passagem de Mensagens

Um ambiente de passagem de mensagem consiste basicamente em uma biblioteca de comunicação que, atuando como uma extensão das linguagens seqüenciais (como C e Fortran), permite a elaboração de aplicações paralelas.

Os ambientes de passagem de mensagens não foram desenvolvidos especificamente com o intuito de utilizar os sistemas distribuídos para o desenvolvimento de aplicações paralelas. Eles foram desenvolvidos inicialmente para máquinas com processamento maciçamente paralelo (*Massively Parallel Processing* - MPP) onde, devido à ausência de um padrão, cada fabricante desenvolveu seu próprio ambiente, sem se preocupar com portabilidade. Com o passar dos anos, muita experiência foi adquirida, pois os diferentes projetos de interfaces de passagem de mensagens enfatizavam aspectos diferentes para o seu sistema. Exemplos desses sistemas são *nCUBE* PSE, *IBM EUI*, *Meiko CS System* e *Thinking Machines CMMD* [MCB94].

Com o objetivo de acabar com o problema de portabilidade, vários grupos de pesquisa desenvolveram ambientes de passagem de mensagens com plataforma portátil. A idéia é definir um conjunto de funções independentes da máquina que está sendo utilizada e implementá-las em várias plataformas de hardware.

As aplicações puderam, com isso, ganhar a portabilidade perdida e serem executadas em todos os equipamentos para os quais o ambiente foi desenvolvido. Recentemente os ambientes portáteis também têm sido desenvolvidos para sistemas heterogêneos, onde dois ou mais tipos de computadores diferentes cooperam para resolver um problema. Exemplos de plataformas portáteis para equipamentos heterogêneos são: P4, PARMACS, Express, PVM, MPI, entre outros [MCB94] [CAR94] [WAL94] [MAT93] [BUT94] [CAL94] [FLO94] [SUN94].

Computação Paralela em Sistemas Heterogêneos: Problemas e Vantagens

Em uma arquitetura maciçamente paralela, todos os processadores são exatamente iguais em capacidade, recursos, software e velocidade de comunicação. Isso não acontece em uma rede. Os computadores disponíveis em uma rede podem ser de diferentes fabricantes com diferentes compiladores. Portanto, quando se deseja explorar um conjunto de computadores interligados por uma rede, poderá haver vários tipos de heterogeneidade, como por exemplo [BEG94]:

- arquiteturas;
- formato de dados;
- potência computacional;
- carga de trabalho em cada máquina;
- carga de trabalho na(s) rede(s).

O conjunto de computadores disponíveis pode incluir vários tipos arquiteturais tais como: computadores CISC (PC 386/486/pentium), computadores RISC (*Sun SPARCstations*, *DECstation*, entre outros), multiprocessadores com memória compartilhada, etc. Cada tipo de arquitetura tem suas próprias características para o desenvolvimento de aplicações. A máquina virtual paralela pode ser composta por computadores paralelos e mesmo que sejam empregados apenas computadores seriais, haverá ainda o problema da incompatibilidade nos códigos binários, sendo preciso compilar uma tarefa paralela em cada máquina diferente.

Computadores diferentes freqüentemente possuem formatos de dados incompatíveis. Isso é um ponto importante para os sistemas distribuídos visto que o dados enviados de um computador podem não ser reconhecidos por outro. Há portanto, a necessidade de ambientes de passagem de mensagens que traduzam os formatos de dados incompatíveis, permitindo assim a comunicação.

Máquinas com potências computacionais diferentes podem gerar desempenhos fracos. Como exemplo disso pode-se considerar o problema de executar tarefas paralelas em uma máquina virtual, composta por um supercomputador e uma estação de trabalho. O projeto da aplicação deve assegurar que o supercomputador não fique ocioso aguardando o processamento da estação de trabalho para poder continuar.

Como sistemas distribuídos normalmente possuem vários usuários, cada um executando suas aplicações, a carga de trabalho atribuída a cada máquina e à(s) rede(s) podem variar muito, afetando aplicações que desejam explorar o paralelismo no sistema.

Apesar de algumas dificuldades, a computação paralela sobre sistemas distribuídos (e principalmente sobre sistemas heterogêneos) oferece potencialmente muitas vantagens:

- custo reduzido, devido a utilização do hardware já existente, o qual era empregado apenas para a execução de aplicações seqüenciais;
- desempenho alto, por atribuir cada tarefa para a arquitetura mais apropriada;
- explora a heterogeneidade natural de certas aplicações, permitindo acesso a bancos de dados diferentes e a processadores especiais para determinadas partes do problema;
- os recursos da máquina virtual podem aumentar gradativamente permitindo a assimilação de tecnologias de ponta mais facilmente (com custo menor);
- utilização de recursos conhecidos para o desenvolvimento de aplicações. Programadores podem utilizar editores, compiladores e periféricos disponíveis para os equipamentos seriais;
- a computação distribuída pode facilitar o trabalho corporativo.

Todos esses fatores têm por objetivo reduzir o tempo de desenvolvimento e de depuração, otimizar a utilização dos recursos, reduzir custos e tornar as aplicações mais eficientes. Esses são os benefícios que todos os ambientes de passagem de mensagens devem explorar.

5.2. Exemplos de Ambientes de Passagem de Mensagens

Essa seção descreve algumas interfaces de passagem de mensagens com plataformas portáteis para sistemas heterogêneos.

P4

O sistema P4 desenvolvido no *Argonne National Laboratory* foi a primeira tentativa de desenvolvimento de uma plataforma portátil. Seu projeto começou a ser elaborado em 1984 (chamado na época de Monmacs) e a partir da sua terceira geração (em 1989) ele foi reescrito com o objetivo de produzir um sistema mais robusto para as necessidades recentes de passagem de mensagens entre máquinas heterogêneas.

O ambiente P4 é uma biblioteca de macros projetada para expressar uma grande variedade de algoritmos paralelos portáteis, com eficiência e simplicidade. Para atingir a eficiência, cada implementação do P4 é voltada para conseguir o melhor desempenho da plataforma escolhida. A simplicidade é obtida com um número pequeno de conceitos, suficientes entretanto, para permitir o desenvolvimento dos algoritmos paralelos projetados [BUT94].

A principal característica do P4 é a possibilidade de ser utilizado por múltiplos modelos de computação paralela, podendo ser empregado em arquiteturas MIMD com memória distribuída, com memória compartilhada e em *clusters* onde há uma coleção de máquinas com ambos os tipos de memória.

Para as arquiteturas MIMD com memória compartilhada (não máquinas com memória compartilhada virtual) ele fornece o modelo de monitores para coordenar o acesso aos dados compartilhados. Exemplos de equipamentos são: *Alliant FX/2800* e *Sequent symmetry*.

Para as arquiteturas MIMD com memória distribuída o sistema P4 fornece funções para a enviar e receber mensagens. Essas operações são bloqueantes, sendo isso uma grande restrição do P4.

As mensagens podem ser enviadas para plataformas heterogêneas. XDR (*External Data Representation*) é usado para traduzir as mensagens entre máquinas com diferentes formatos de dados. Devido à necessidade de desempenho, essa tradução é feita somente quando é absolutamente necessária. Exemplos de plataformas que executam o P4 são: *Intel Touchstone*, *CM-5* e redes heterogêneas de estações de trabalho.

Nos *clusters* os processos podem ser coordenados tanto pelos monitores quanto pela passagem de mensagens, dependendo da organização da memória. Além disso, há funções para identificar o processo principal de cada *cluster*, determinar o número de *clusters* e obter os identificadores dos processos pertencentes a um *cluster*.

O P4 é bastante flexível no modo em que os processos são iniciados. Dependendo da plataforma, são iniciados vários processos ou apenas um, o qual irá

iniciar os outros. É utilizado um arquivo que especifica as características dos processos que estão em execução como: as máquinas em que eles estão sendo executados, os arquivos executáveis e os grupos de *clusters* de processos que compartilham memória.

Dois paradigmas, o **SPMD** e o **mestre-escravo** podem ser utilizados no ambiente P4. O SPMD (*Single Program - Multiple Data*) consiste de um único programa sendo executado em todos os processadores de uma máquina MIMD, mas com diferentes dados e com possíveis execuções diferentes em cada processador. O modelo mestre-escravo possui um processo mestre que é chamado pelo usuário e é o responsável por iniciar os outros processos (que são diferentes), denominados escravos.

Parmacs

O Parmacs (*PARallel MACroS*) é uma biblioteca de comunicação portátil, a qual tem sido implementada na maioria das máquinas com arquitetura MIMD, em máquinas com MPP e em redes de estações de trabalho. Ele herdou muitas características do P4 e começou a ser desenvolvido em 1987 no *Argonne National Laboratory.*

Na primeira versão foi implementado um conjunto básico de instruções para passagem de mensagens utilizando a linguagem C. Atualmente o Parmacs está sendo comercializado na versão 6.0, cuja principal diferença em relação às versões anteriores, é a substituição das macros por uma biblioteca de funções, onde a chamada a subrotinas tornaram o Parmacs mais próximo das linguagens de programação atuais.

Seu modelo de programação é baseado em memória local, ou seja, processos paralelos podem ter acesso somente ao seu espaço de endereçamento. Não existem variáveis globais compartilhadas [CAL94].

A ativação dos processos paralelos com Parmacs é feita através de um processo principal (*host*). Sua tarefa é criar processos (*nodes*) e distribuí-los através do hardware. Todos os *nodes* são criados ao mesmo tempo e é atribuído um identificador a cada um. Os *nodes* podem ser finalizados pelo *host*, o qual pode criar outros processos *nodes*.

A comunicação entre os processos é feita pela troca de mensagens as quais são seções de memória contínuas caracterizadas pelo seu endereço inicial e comprimento. Uma mensagem pode ser enviada para outro processo especificando o processo destino e a mensagem que se deseja enviar, sendo que a recepção pode também obter o identificador do processo que enviou.

A comunicação pode ser síncrona ou assíncrona. No primeiro modo, o processo que enviou e o que recebeu a mensagem realizam um *rendezvous*. No segundo caso, o processo que enviou a mensagem continua a sua execução tão logo a mensagem tenha sido copiada para o *buffer* do usuário (origem) e esteja apta para a transmissão.

Assim como o P4, a comunicação entre processadores com diferentes representações de dados, é feita através do padrão XDR.

Express

O sistema Express é um produto da empresa Parasoft, desenvolvido para ser um ambiente de passagem de mensagens atuando sobre várias arquiteturas MIMD com memória distribuída [ALM94], visando obter o máximo de desempenho de cada plataforma utilizada [FLO94].

Com a sua evolução, o Express passou a ser considerado também como um conjunto de ferramentas para desenvolvimento de software paralelo [FLO94]. A sua interface com o usuário está sendo melhorada, procurando-se esconder a maioria de detalhes internos. Isso levou ao desenvolvimento de mapeamentos e bibliotecas de comunicação para os diferentes tipos de topologias empregadas, para se obter o melhor desempenho possível em cada uma.

Uma das características principais do Express é abordar problemas como o balanceamento dinâmico de carga e de desempenho em I/O paralelos. Esses dois fatores são quase totalmente ignorados pela maioria dos ambientes de passagem de mensagens.

Linda

Linda é um ambiente independente de máquina para implementação de códigos paralelos e começou a ser desenvolvido em 1980 pela *Yale University*. Atualmente possui duas versões comerciais: uma para máquinas paralelas com memória compartilhada e distribuída, e outra para estações de trabalho em rede [MAT93].

Linda cria e coordena múltiplos processos de maneira diferente dos demais ambientes de passagem de mensagens (como Parmacs, Express e outros). O ambiente está baseado em um conjunto de operações que implementam o conceito do *Tuple Space* (TS).

TS é um modelo de memória, descrito por uma linguagem de alto nível (como C e Fortran). Ele fornece um nível de abstração onde é possível a construção lógica de uma memória compartilha sobre memórias distribuídas, ou seja, memória compartilhada virtual.

As tarefas de gerenciamento de processos, sincronização e comunicação são realizadas aplicando-se as operações ao TS [MCB94].

Objetos de dados são conhecidos como *tuples*, os quais podem ser de dois tipos diferentes: *Process Tuples* (PT) ou *Data Tuples* (DT). O PT troca dados através da criação, leitura e alteração de DT.

As seis operações que manipulam as *tuples* são [MAT93]:

- out : insere dados (DT) no TS;
- eval : ativa um processo (PT);
- in : procura uma tuple no TS e remove a primeira encontrada;
- rd : semelhante ao in mas a tuple não é removida;

- inp : versão não bloqueante do in;
- rdp : versão não bloqueante do rd.

Para ocorrer uma comunicação, essa não pode ser diretamente entre os processos e sim através do TS. Por exemplo, para um processo A transmitir dados para o processo B, A envia uma *tuple* de dados (DT) para o TS e B procura a *tuple* em TS.

MPI

Diversas plataformas portáteis atuais apresentam, ainda, alguns dos problemas identificados nos ambientes de passagem de mensagens propostos inicialmente, desenvolvidos por fabricantes para execução apenas nas suas máquinas. A maioria das plataformas portáteis existentes possui apenas um subconjunto das características necessárias para os mais diversos equipamentos fabricados [MCB94], dificultando a tarefa de implementação em diferentes máquinas.

Devido a esses problemas foi criado o Comitê MPI (em 1992), para definir um padrão para os ambientes de passagem de mensagens denominado MPI (*Message Passing* Interface). Os principais objetivos são a padronização para a maioria dos fabricantes de hardware e plataformas portáteis, e eficiência.

O comitê MPI reúne membros de aproximadamente 40 instituições e inclui quase todos os fabricantes de máquinas com MPP, universidades e laboratórios governamentais pertencentes à comunidade envolvida na computação paralela mundial [MCB94] [WAL94].

O MPI é um padrão de interface de passagem de mensagens para aplicações que utilizam computadores MIMD com memória distribuída. Ele não oferece nenhum suporte para tolerância a falhas e assume a existência de comunicações confiáveis. O MPI não é um ambiente completo para programação concorrente, visto que ele não implementa: I/O paralelos, depuração de programas concorrentes, canais virtuais para comunicação e outras características próprias de tais ambientes [WAL94].

Um conjunto de rotinas responsáveis pela comunicação ponto-a-ponto entre os pares de processos forma o núcleo do MPI. São implementadas rotinas bloqueantes e não bloqueantes para enviar e receber mensagens.

A rotina que envia a mensagem no modo bloqueante não retorna enquanto a mensagem contida no *buffer* não estiver segura. No modo não bloqueante a rotina que envia a mensagem pode retornar enquanto a mensagem ainda está “volátil”.

Para receber uma mensagem no modo bloqueante, a rotina não retorna até que a mensagem seja inserida no *buffer.* No modo não bloqueante a rotina pode retornar antes da mensagem ter chegado.

MPI possui grupos de processos e rotinas para o gerenciamento dos grupos. Os grupos podem ser usados para duas funções distintas. Na primeira os grupos especificam os processos envolvidos em uma operação de comunicação coletiva,

como um *broadcasting.* Toda a comunicação ocorre dentro e através dos grupos. Na segunda eles podem ser usados para introduzir o paralelismo dentro da aplicação, onde diferentes grupos realizam diferentes tarefas.

Cada grupo pode possuir códigos executáveis diferentes ou o mesmo código. Quando é utilizado o mesmo código está sendo utilizado o paradigma SPMD [WAL94].

PVM

O PVM (*Parallel Virtual Machine*) é um conjunto integrado de bibliotecas e de ferramentas de software, cuja finalidade é emular um sistema computacional concorrente heterogêneo, flexível e de propósito geral [BEG94].

Diferente de outros ambientes portáteis desenvolvidos inicialmente para máquinas com multiprocessadores (como o P4, Express e outros), o PVM nasceu com o objetivo de permitir que um grupo de computadores interconectados, possivelmente com diferentes arquiteturas, possa trabalhar cooperativamente formando uma máquina paralela virtual [GEI94].

O projeto PVM teve início em 1989 no *Oak Ridge National Laboratory* - ORNL. A versão 1.0 (protótipo), foi implementada por *Vaidy Sunderam* e *Al Geist* (ORNL) sendo direcionada ao uso em laboratório. A partir da versão 2 (1991), houve a participação de outras instituições (como *University of Tennessee, Carnegie Mellon University,* entre outras), quando começou a ser utilizado em muitas aplicações científicas. A versão 2 deu início à distribuição gratuita do PVM. Depois de várias revisões (PVM 2.1 - 2.4), o PVM foi completamente reescrito, gerando a versão 3.0 (em fevereiro de 1993).

Várias mudanças foram feitas na versão 3.0, com objetivo de retirar erros de programação e ajustar pequenos detalhes como oferecer interface com o usuário melhor e aumentar o desempenho de certas comunicações (como em multiprocessadores). A versão disponível mais recente é o PVM 3.3, sendo a versão discutida neste trabalho. Já foram realizadas várias revisões nessa versão tendo como objetivo a retirada de erros e a inclusão de novas arquiteturas.

O sistema PVM é composto por duas partes [GEI94]. A primeira é um *daemon*, chamado pvmd3 (ou simplesmente pvmd), que reside em todos os *hosts,* compondo a máquina virtual. O termo máquina virtual será utilizado para designar um computador lógico com memória distribuída e o termo *host* será para designar um dos computadores que formam a máquina virtual.

O pvmd foi projetado para ser instalado na máquina por qualquer usuário com um *login* válido. Quando um usuário deseja executar uma aplicação PVM, ele primeiro deve criar a máquina virtual iniciando o PVM. A aplicação pode então ser iniciada a partir do *prompt* do sistema operacional (normalmente, mas não necessariamente o UNIX), em qualquer computador pertencente à máquina virtual. Pode haver mais de uma máquina virtual utilizando os mesmos equipamentos simultaneamente, sendo que cada aplicação não interfere nas demais.

A segunda parte do sistema é uma biblioteca de rotinas da interface PVM. Ela contém um conjunto completo de primitivas que são necessárias para a cooperação entre as tarefas de uma aplicação. Essa biblioteca contém rotinas que podem ser chamadas pelo usuário para a passagem de mensagens, geração de processos, coordenação de tarefas e modificação da máquina virtual.

O sistema PVM permite que sejam escritas aplicações nas linguagens Fortran, C e C++. A escolha por esse conjunto de linguagens deve-se ao fato de que a maioria das aplicações passíveis de paralelização estão escritas nessas linguagens.

Atualmente o PVM está disponível para uma grande variedade de plataformas. A tabela abaixo mostra alguns equipamentos que podem ser utilizados pela versão 3.3.

Alguns dos principais equipamentos que executam o PVM:

Alliant FX/8
DEC Alpha
Sequent Balance
Bbn Butterfly TC2000
80386/486/Pentium com UNIX (Linux ou BSD)
Thinking Machines CM2 CM5
Convex C-series
C-90, Ymp, Cray-2, Cray S-MP
HP-9000 modelo 300, Hp-9000 PA-RISC
Intel iPSC/860, Intel iPSC/2 386 host
Intel Paragon
DECstation 3100, 5100
IBM/RS6000, IBM RT
Silicon Graphics
Sun 3, Sun 4, SPARCstation, Sparc multiprocessor
DEV Micro VAX

Através do PVM, uma coleção de computadores heterogêneos (seriais, paralelos e vetoriais) desempenham as funções de um computador com memória distribuída e com alto desempenho. O PVM fornece as funções que iniciam automaticamente as tarefas (*tasks*) na máquina virtual e permitem a comunicação e a sincronização entre elas. Uma tarefa é definida como uma unidade computacional em PVM análoga aos processos UNIX (freqüentemente, não necessariamente, é um processo UNIX) [SUN94] [GEI94] [BEG94].

O modelo computacional do PVM é , portanto, baseado na noção de que uma aplicação consiste de várias tarefas. Cada tarefa é responsável por uma parte da carga de trabalho da aplicação.

Uma aplicação pode ser paralelizada por dois métodos: o paralelismo funcional e o paralelismo de dados. No paralelismo funcional (também conhecido como paradigma mestre-escravo) a aplicação é dividida através das suas funções, isto é, cada tarefa desempenha um serviço diferente, como por exemplo entrada, processamento e saída.

O paralelismo de dados refere-se ao paradigma SPMD descrito anteriormente. O PVM permite qualquer um dos métodos, como também um método híbrido (uma mistura dos dois). A Figura 5.1 mostra um exemplo do modelo computacional do PVM e uma visão arquitetural destacando a heterogeneidade do sistema.

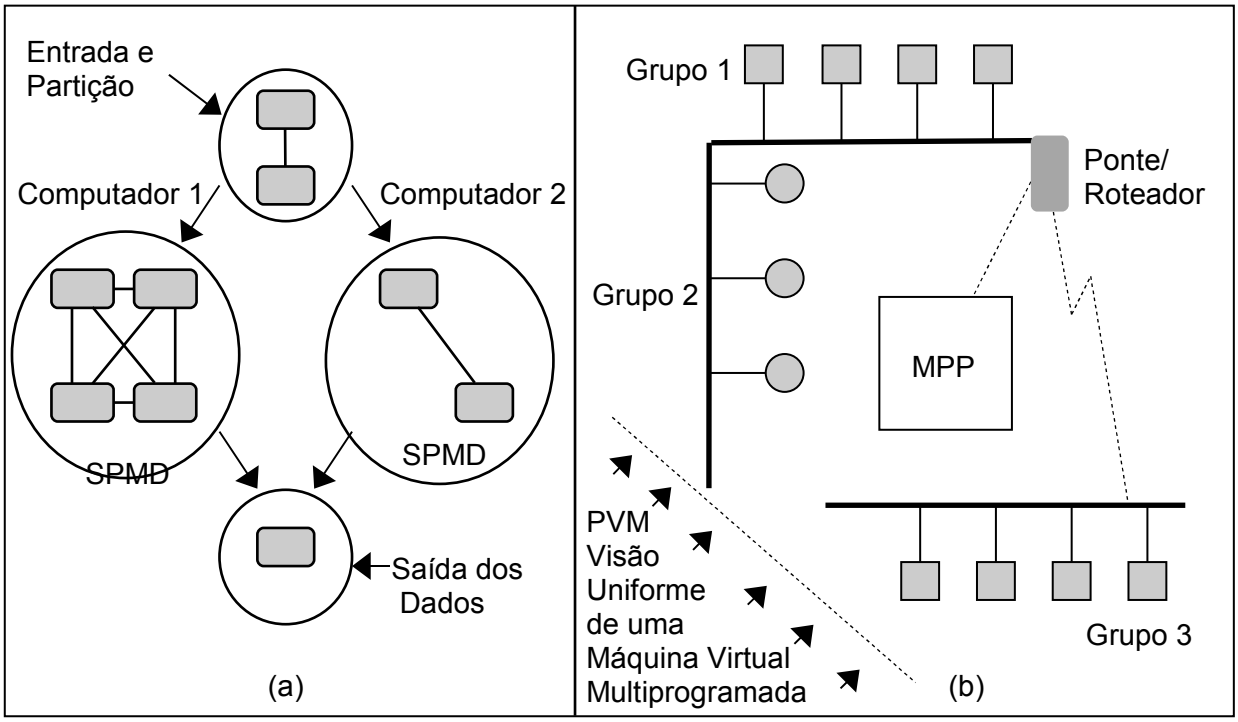

Figura 5.1 - O PVM. (a) Modelo Computacional (b) Visão Arquitetural

6. Considerações Finais

A filosofia seqüencial de von Neumann representou um paradigma computacional eficiente e suficiente para o panorama tecnológico das primeira décadas de existência da computação, onde ambos memória e processador eram recursos caros e preciosos. Na época atual, porém, essas restrições tecnológicas já não apresentam o mesmo peso, e a manufatura de computadores com vários processadores já não é uma opção economicamente inviável [KIR91].

As diversas áreas na qual a computação se aplica demandam cada vez mais poder computacional e vários autores acreditam que esse poder computacional só pode ser conseguido através do processamento paralelo. Além disso, o aumento de velocidade em processadores seqüenciais tende a alcançar um valor limite. Segundo Lenatti [LEN95], o processamento paralelo está rapidamente se tornando uma realidade dentro da comunidade empresarial, sendo a sua ótima relação custo/desempenho o seu fator de impulso comercial. Porém, substituir uma filosofia computacional de décadas de existência, como é a de von Neumann, não é uma tarefa fácil ou rápida [ALM94] [AMO88] [KIR91] [NAV89].

Em nível de *hardware*, muito já se desenvolveu em computação paralela, e muita experiência já foi adquirida. Porém, em nível de *software* paralelo, ainda existem muitas lacunas a serem preenchidas e muito ainda deve ser pesquisado até que se encontrem soluções ótimas, que impliquem na utilização dessas arquiteturas paralelas com grande desempenho [NAV89]. Segundo Zaluska [ZAL91], o custo de sistemas computacionais paralelos tende a ser dominado pelo custo de *software* e as características fundamentais que todo *software* paralelo deve oferecer abrangem: ser facilmente transportado entre plataformas de *hardware* diferentes, ser facilmente ampliado para acomodar grandes problemas sem dificuldade e ser de uso fácil.

Dentre as plataformas de execução de programas paralelos, destaca-se o modelo MIMD com memória distribuída, em função de sua grande flexibilidade e facilidade de ampliação [BLE94] [MCB94] [ZAL91]. Dentro do modelo MIMD, uma tendência atual é a utilização de sistemas distribuídos, interligados por redes de comunicação rápidas, como plataformas de programação paralela [BLE94]. Sistemas distribuídos representam uma solução de custo acessível, em comparação às arquiteturas paralelas de alto custo, o que de certa forma expande o uso da computação paralela para uma comunidade maior de usuários [BEG94] [BLE94] [ZAL91].

7. Referências Bibliográficas

[ALM94] ALMASI, G. S., Gottlieb A., *Highly Parallel Computing*, 2ª. ed., The Benjamin Cummings Publishing Company, Inc., 1994.

[AMO88] AMORIM, C. L., *Uma Introdução a Computação Paralela e Distribuída*, VI Escola de Computação, 1988.

[AND83] ANDREWS, G. R., Schineider, F. B., "Concepts and Notations for Concurrent Programming", *ACM Computing Survey*, v. 15, nº.1, pp. 3-43, 1983.

[BEG94] BEGUELIN, A., *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, The MIT Press, 1994.

[BEN93] BEN-DYKE, A. D., "Architectural taxonomy, A brief review", University of Birmingham, 1993.

[BLE94] BLECH, R. A., "An overview of parallel processing", Slides, *Parallel Computing with PVM Workshop*, Nasa Lewis Research Center, http://www.lerc.nasa.gov/Other_Groups/IFMD/2620/tutorialPP.html, 1994.

[BUT94] BUTLER, R. M., Lusk, E. L., "Monitors, messages and clusters: The P4 parallel programming system", *Parallel Computing*, v. 20, pp. 547-564, 1994.

[CAR94] CARRIERO, N., "The Linda alternative to message-passing systems", *Parallel Computing*, vol. 20, pp. 633-655, 1994.

[CAL94] CALKIN, R., "Portable programming with the PARMACS message-passing library", *Parallel Computing*, v. 20, pp. 615-632, 1994.

[DUN90] DUNCAN, R., "A Survey of Parallel Computer Architectures", *IEEE Computer*, pp.5-16, Fevereiro, 1990.

[FLO94] FLOWER, J., Kolawa A., "Express is not just a message passing system. Current and future directions in Express", *Parallel Computing*, v. 20, pp. 597-614, 1994.

[FLY72] FLYNN, M. J., "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, v. C-21, pp.948-960, 1972.

[GEI94] GEIST, A., Beguelin, A., Dongarra J., Jiang, W., Manchek, R., Sunderam V., *PVM 3 User's Guide and Reference Manual*, Oak National Laboratory, Setembro, 1994.

[HWA84] HWANG, K., Briggs, F. A., *Computer Architecture and Parallel Processing*, McGraw-Hill International Editions, 1984.

[KIR89] KIRNER, C., “Sistemas operacionais para ambientes paralelos”, *IX Congresso da SBC, Anais da VIII Jornada de Atualização em Informática*, 1989.

[KIR91] KIRNER, C., “Arquiteturas de sistemas avançados de computação”, *Anais da Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho*, pp. 307-353, 1991.

[LEN95] LENATTI, C., “Rethinking in parallel”, *Open Computing*, vol. 12, pp. 57-58, 1995.

[MAT93] MATRONE, A., “Linda and PVM: A comparison between two environments for parallel programming”, *Parallel Computing*, v. 19, pp. 949-957, 1993.

[MCB94] MCBRYAN, O. A., “An overview of message passing environments”, *Parallel Computing*, v. 20, pp. 417-444, 1994.

[NAV89] NAVAUX, P. O. A., “Introdução ao processamento paralelo”, *RBC- Revista Brasileira de Computação*, v. 5, nº 2, pp.31-43, Outubro, 1989.

[QUI87] QUINN, M.J., *Designing Efficient Algorithms for Parallel Computers*, McGraw Hill, 1987.

[SNO92] SNOW, C.R., *Concurrent Programming*, Cambridge University Press, 1992.

[SUN94] SUNDERAM, V. S., Geist, A., Dongarra, J., Manchek, R., “The PVM concurrent computing system: evolution, experiences and trends”, *Parallel Computing*, v. 20, pp. 531-545, 1994.

[TAN95] TANENBAUM, A. S., *Distributed Operating System*s , Prentice Hall, 1995.

[WAL94] WALKER, D. W., “The design of a standard message passing interface for distributed memory concurrent computers”, *Parallel Computing*, vol. 20, pp. 657-673, 1994.

[ZAL91] ZALUSKA E. J., “Research lines in distributed computing systems and concurrent computation”, *Anais do Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software*, pp. 132-155, 1991.