Introdução a Programação Paralela e Distribuída

Liria Matsumoto Sato
Edson Toshimi Midorikawa
Hermes Senger
{liria, emidorik, senger}@lsi.usp.br

Laboratório de Sistemas Integráveis
Escola Politécnica da Universidade de São Paulo

Av. Prof. Luciano Gualberto, travessa 3, n°158
Cidade Universitária, São Paulo, SP - 05508-900

Fone: (011) 818-5589

FAX: (011) 818-5664 / 211-4574

WWW: http://www.lsi.usp.br./~liria/jai96.html

Capítulo 1 - INTRODUÇÃO

A demanda crescente de processamento tem motivado a evolução dos computadores, viabilizando implementações de aplicações que envolvem um intenso processamento e grandes volumes de dados.

Na busca pelo alto desempenho tem-se as alternativas:

- aumentar o desempenho do processador. Este aumento pode ser obtido através de:
  - aumento da velocidade do relógio: esta alternativa envolve o desenvolvimento de tecnologia de confecção de circuitos integrados, trazendo entretanto, problemas, tal como, o aumento de temperatura;
  - melhorias na arquitetura: este objetivo motivou o surgimento dos processadores RISC, vetoriais e super-escalares;
  - melhoria no acesso à memória, por exemplo, através da exploração da hierarquia de memória, utilizando registradores, memória cache e memória pricipal;
- utilizar vários processadores, distriibuindo, entre estes, o processamento do programa.

A implementação de algoritmos sobre um computador paralelo requisita recursos de programação paralela que permitam expressar o paralelismo e incluir mecanismos para sincronização e comunicação. A programação paralela é a programação concorrente orientada para computadores paralelos, incorporando os seus requisitos de sincronização e comunicação, e buscando a utilização adequada dos recursos de processamento para otimizar o seu desempenho.

Dada a diversidade de arquiteturas paralelas, linguagens, compiladores e bibliotecas especiais para gerenciar o paralelismo foram propostos.

O texto está estruturado nos seguintes capítulos:

Capítulo 2 - Processamento Paralelo e Distribuído e Medidas de Desempenho

Neste capítulo será apresentado como o alto desempenho pode ser obtido e quais as métricas mais utilizadas. Será feita uma apresentação sucinta das arquiteturas paralelas.

Capítulo 3 - Modelos de Programação Paralela

Serão apresentados aqui os modelos de programação paralela que têm sido propostos, sendo detalhados os modelos de programação procedural baseados em variáveis compartilhadas e passagem de mensagens. Também serão abordados suportes fornecidos para a implementação de tais modelos (sistemas DSM e a biblioteca PVM).

Capítulo 4 - Exemplos de sistemas de programação

Serão apresentados aqui exemplos que mostram concretamente o que oferecem os sistemas de programação de máquinas paralelas, através da descrição dos ambientes da SGI Power 4D e Meiko CS-2.

Capítulo 5 - Algumas linguagens e bibliotecas de programação paralela

Serão apresentadas aqui algumas linguagens propostas e em uso para programação paralela, em particular, serão descritas com mais detalhes as linguagens HPF e CPAR. É estudada também a biblioteca PVM.

Capítulo 6 - Algoritmos e implementações para sistemas com memória compartilhada

Será apresentado como devem ser explorados os recursos de sistemas com memória compartilhada para obter bom desempenho. Serão utilizados como exemplo os algoritmos de multiplicação de matrizes e decomposição LU e suas implementações em CPAR.

Capítulo 7 - Algoritmos e implementações para sistemas com memória distribuída

Será apresentado como devem ser explorados os recursos de sistemas com memória distribuída para obter bom desempenho. Será utilizado como exemplo de análise e estudo o algoritmo de multiplicação de matrizes e sua implementação em PVM.

Capítulo 2 - PROCESSAMENTO PARALELO E DISTRIBUÍDO E MEDIDAS DE DESEMPENHO

A busca do alto desempenho, visando atender a demanda crescente de processamento, motivou o surgimento de vários modelos de arquiteturas paralelas. Segundo Duncan [Dun90] arquiteturas paralelas são caracterizadas pela presença de múltiplos processadores que cooperam entre si.

2.1 - Arquiteturas

Diversas taxonomias para arquiteturas paralelas foram propostas. Visando abranger os computadores escaláveis, Gordon Bell apresentou uma taxonomia [Bel94], mostrada na figura 2.3, que inclui as arquiteturas mais recentes.

Sistemas compostos de estações de trabalho interconectadas por um chaveamento de alta velocidade como o ATM podem ser classificados como computadores escaláveis.

Utilizando múltiplos pares microprocessador-memória primária interconectados por um chaveamento de alta velocidade [Bel94a][Kof94], pode-se atingir altos níveis de desempenho.

A taxonomia apresentada por Gordon Bell [Bel94] classifica as arquiteturas, como segue:

- Fluxo de Instruções Único
  - ♦ fluxo de dados único
    - ◊ CISC
    - ◊ RISC
    - ◊ Superescalar e VLIW RISC
    - ◊ Processadores de sinais digitais
    - ◊ “Multi-threaded”
  - ♦ fluxos de dados múltiplos
    - ◊ Vetoriais
    - ◊ SIMD
- Fluxos de Instruções Múltiplos
  - ♦ multiprocessador (memória compartilhada)
    - ◊ acesso à memória não uniforme
      - ∗ somente cache (COMA)
      - ∗ “memory coherent”
      - ∗ nenhum cache
    - ◊ acesso à memória uniforme
      - ∗ “multi-threaded”
      - ∗ supercomputadores e “mainframes”
      - ∗ múltiplos barramentos
        - ♦ multicomputador (passagem de mensagem)
    - ◊ chaveamento distribuído e centralizado
      - ∗ granularidade fina homogênea
      - ∗ granularidade média não homogênea
      - ∗ estações interconectadas na rede
    - ◊ rede distribuída
      - ∗ aglomerados
      - ∗ estações em rede local
      - ∗ estações em área externa

2.1.1. Fluxo de instruções único

Esta categoria é caracterizada pela execução de um único fluxo de instruções em cada processador. Estão incluídos desde os computadores CISC até computadores com palavras de instruções bem longas, como os superescalares e VLSIW RISC, e computadores “multi-threaded”, onde múltiplos “threads” processam o mesmo fluxo de instruções.

Os processadores vetoriais SIMD são o núcleo dos supercomputadores e são largamente utilizados em aplicações com uso intenso de ponto flutuante e em aplicações incluindo transformações gráficas.

2.1.2. Fluxos de instruções múltiplos

Esta categoria denominada de “MIMD” é caracterizada pela execução de múltiplos fluxos de instruções sobre fluxos distintos de dados, pelos seus múltiplos processadores. São classificados em duas classes, conforme a forma de comunicação entre os processadores: os multiprocessadores e os multicomputadores.

2.1.3. Multiprocessadores

Nesta taxonomia, de acordo com o tipo de acesso à memória, os multiprocessadores que apresentam memória compartilhada, são classificados em duas categorias:

- acesso à memória uniforme (UMA);
- acesso à memória não uniforme (NUMA).

A memória “cache” é caracterizada pelo seu acesso rápido. Os multiprocessadores classificados nesta categoria como “somente cache”(COMA) é um tipo de NUMA, onde a memória local de cada nó de processamento é transformada em memória cache.

Uma subclasse de NUMA, aqui classificada como “coerência de memória”, é caracterizada por apresentar uma memória “cache” pequena, utilizada para acessos a dados presentes na memória de outro nó. Os sistemas DASH [Len92] e FLASH [Kus94] e o CEDAR [VIP94] são exemplos dessa classe de sistemas.

Esta hierarquia de acesso presente em multiprocessadores NUMA deve ser explorada pelos compiladores e pelo programador para reduzir o custo de comunicação.

2.1.4. Multicomputadores

Esta categoria se caracteriza por apresentar recursos de comunicação por passagem de mensagem. Estações de trabalho distribuídas utilizando chaves ATM, com interfaces de comunicação bem projetadas, podem apresentar um custo de comunicação da ordem dos multicomputadores como o CM5 [Bel94].

Por outro lado, estações de trabalho conectadas em uma rede local são uma boa forma de se obter computadores escaláveis, uma vez que podem ser utilizadas até algumas dezenas de estações para executar um único trabalho.

2.2 Formas de Paralelização de Programas

Dada a diversidade de arquiteturas paralelas, linguagens, compiladores e bibliotecas especiais foram propostos, visando gerenciar o paralelismo.

O paralelismo em um programa pode ser explorado implicitamente através de compiladores paralelizantes, como o Parafrase2 [Pol88] e o Suif , que paralelizam automaticamente programas escritos em C. ou explicitamente.

A experiência tem mostrado que a programação paralela é significativamente mais difícil que a programação seqüencial, principalmente porque envolve a necessidade de sincronização entre tarefas, como também a análise de dependência de dados. A utilização de um sistema paralelizante minimiza estas dificuldades, e permite também o reaproveitamento de programas seqüenciais já implementados.

Por outro lado, a programação paralela, expressando o paralelismo, permite que fontes de paralelismo não passíveis de detecção por um sistema paralelizante, possam ser explorados. Aplicações onde formas de paralelismo específicas são requisitadas não podem ser paralelizadas automaticamente.

O paralelismo expresso pelo usuário pode ser especificado em um programa através de:

- expressão por comandos específicos de uma linguagem de programação paralela;
- chamadas a rotinas de uma biblioteca que gerencia os recursos de paralelismo da máquina;
- diretivas do compilador.

Entre estas formas, a utilização de uma linguagem de programação paralela é a que oferece um ambiente mais adequado de programação.

A seguir é apresentado o exemplo de um programa escrito na linguagem CPAR.

```
shared double total_pi = 0.0;
void main ()
{
int i, nrets = MAXRET; /* # de retangulos */
double intervalo, largura = 1/MAXRET;
double x, local_pi = 0.0;
semaforo semaforo;

create_sem (&semaforo, 1);
intervalo = nrets/num_procs;

/* execucao do calculo parcial */
forall i = 0 to MAXRET    {
  x = ((i-0.5) * largura);  /* calcula x */
  local_pi = local_pi + (4.0 / ( 1.0 + x * x));
  local_pi = local_pi * largura;
  /* atualização da variavel global */
  lock (&semaforo);
  total_pi = total_pi + local_pi;
  unlock (&semaforo);
  }
printf (”valor de pi = %d\n”, total_pi);
}
```

Bibliotecas específicas também têm sido uma forma de implementar programas muito utilizada. Bibliotecas específicas para multiprocessadores como a oferecida pelo sistema Balance da Sequent [Tha88], e outras voltadas para várias plataformas como a biblioteca do sistema PVM (Parallel Virtual Machine) [Gra92] e as bibliotecas do sistema Quarks [Uta95] e do sistema Treadmarks [Kel93] que oferecem uma memória distribuída compartilhada, estão disponíveis.

A seguir é apresentado um exemplo de um programa que efetua chamadas da biblioteca oferecida pelo sistema Balance da Sequent.

```
#define MAXRET   1000000
double total_pi = 0.0;
void main()
  {
  int i, nrets = MAXRET; /* # de retangulos */
  int num_procs;
  double intervalo, largura = 1/MAXRET;
  void parte_pi();
  num_procs = m_get_numprocs();
  intervalo = nrets/num_procs;
  m_fork (parte_pi, nrets); /* execução paralela */
  m_kill_procs();
  printf (”valor de pi = %d\n”, total_pi);
```

```
  }

void parte_pi ( int nrets )
  {
  double local_pi = 0.0;
  int my_id, inicio, fim;
  my_id = m_get_myid();
  /* calculo de inicio e fim */
  inicio = my_id * num_procs;
  fim = inicio + intervalo;
  /* execucao do calculo parcial */
  for ( i = inicio; i < fim; i++)
    {
    x = ((i-0.5) * largura);     /* calcula x */
    local_pi = local_pi + (4.0 / ( 1.0 + x * x));
    }
  local_pi = local_pi * largura;
  /* atualização da variavel global */
  m_lock();
  total_pi = total_pi + local_pi;
  m_unlock();
  }
```

Compiladores como o Power C [Bau92] oferecem diretivas especiais para utilizar o paralelismo. A seguir é apresentado um exemplo de um programa escrito em Power C Language.

```
#define MAXRET   1000000
double total_pi = 0.0;
void main()
  {
  int i;
  double x, largura, local_pi;
  largura = 1/MAXRET;
#pragma parallel
#pragma shared (total_pi,largura)
#pragma local(i,x,local_pi)
{
#pragma pfor iterate (i=0; MAXRET; 1)
 for (i = 0; i < MAXRET; i++)
    {
    x = ((i-0.5) * largura);          /* calcula x */
    local_pi = local_pi + (4.0 / ( 1.0 + x * x));
    }
  local_pi = local_pi * largura;
#pragma critical
  {
  total_pi = total_pi + local_pi;
  }
}
  printf (”valor de pi = %d\n”, total_pi);
}
```

2.3 - Medidas de Desempenho e Métricas

Medidas de desempenho, que permitam a análise do ganho obtido com o aumento do total de processadores utilizados, são necessárias. Duas medidas usuais são: o “speedup” (ganho de desempenho) e a eficiência.

2.3.1. Speedup

O “speedup” (S) obtido por um algoritmo paralelo executando sobre p processadoes é a razão entre o tempo levado por aquele computador executando o algoritmo serial mais rápido (t_s) e o tempo levado pelo mesmo computador executando o algoritmo paralelo usando p processadores (t_p).

$$S \geq \frac{t_s}{t_p}$$

2.3.2. Eficiência

A eficência (E) é a razão entre o “speedup” obtido com a execução com p processadores e p. Esta medida mostra o quanto o paralelismo foi explorado no algoritmo. Quanto maior a fração inerentemente seqüencial menor será a eficiência.

$$E = \frac{S}{p}$$

2.3.3. Lei de Amdahl’s

Se f é a fração inerentemente seqüencial de uma computação a ser resolvida por p processadores, então o “speedup” S é limitado de acordo com a fórmula.

$$S \leq \frac{1}{f + \frac{1-f}{p}}$$

gráfico da lei de Amdahl’s

Capítulo 3 - MODELOS E LINGUAGENS DE PROGRAMAÇÃO PARALELA

Um programa seqüencial especifica a execução de uma lista de comandos. Um processo pode ser definido como a execução de um programa seqüencial.

A comunicação entre processos que podem ser atribuídos aos múltiplos processadores de um computador com arquitetura paralela, pode ser efetuada através de variáveis compartilhadas ou por passagem de mensagem.

Linguagens de programação paralela, baseadas em passagem de mensagem, como a Concurrent C [Geh86] [Geh88][Geh92] e a Ada [Geh88] foram propostas, assim como bibliotecas que oferecem rotinas para a passagem de mensagem [Tur93], como a biblioteca do sistema PVM (Parallel Virtual Machine) [Gra92].

Modelos de programação baseados em variáveis compartilhadas permitem implementações com menor complexidade em relação aos modelos com passagem de mensagem. Entretanto, as leituras e escritas dessas variáveis devem ser feitas, considerando algumas restrições. Uma leitura e escrita ou múltiplas escritas não podem ser executadas simultaneamente. Isto exige a utilização de uma seção crítica envolvendo o acesso a variáveis compartilhadas. A exclusão mútua é um mecanismo que implementa a seção crítica, garantindo que uma seqüência de comandos seja executada exclusivamente por um processo.

3.1 - Linguagens de Programação

As linguagens podem ser classificadas em 2 tipos básicos: as linguagens imperativas e declarativas.

As linguagens imperativas são caracterizadas por modificar um estado implícito do programa através de comandos. Estão incluídas nesta classe: as linguagens procedurais e orientadas a objetos.

As linguagens declarativas são caracterizadas por uma programação realizada através de expressões. Dentro desta classe estão incluídas as linguagens funcionais e lógicas.

3.1.1. Linguagens procedurais

As linguagens procedurais são caracterizadas por subdividir o programa em subprogramas também mencionados na literatura como rotinas, subrotinas ou procedimentos. As linguagens C e Pascal são exemplos desta classe.

Dados os efeitos colaterais que podem ser gerados, como a alteração em um procedimento de dados que são lidos em outros procedimentos, a análise de dependência de dados torna-se uma tarefa complexa. Existe uma dependência de dados entre dois comandos, quando um deles altera um dado que é lido ou escrito pelo outro. Neste caso os dois comandos não podem ser executados simultaneamente. Na programação através de uma linguagem que permite que o usuário expresse o paralelismo, este deve efetuar uma análise de dependência de dados. No caso de existir uma dependência de dados, mecanismos de exclusão mútua devem ser utilizados, envolvendo o acesso ao dado.

Entretanto, como as linguagens procedurais são largamente utilizadas, é importante que haja extensões das mais conhecidas como a linguagem C e Fortran. Extensões da linguagem C foram propostas, entre as quais podem ser citadas: a Concurrent C [Geh86] [Geh92], a Jade[Rin93] e a CPAR [Sat92], descrita neste trabalho. Extensões da linguagem Fortran foram propostas, podendo ser citadas a linguagem Fortran D [Fox92] e a HPF (High Performance Fortran) [Lov93].

3.1.2. Linguagens orientadas a objetos

O encapsulamento das informações em objetos, fornecido pelo paradigma de orientação a objetos, e

o seu acesso apenas através de chamadas a seus métodos, é a principal característica que torna este paradigma atraente para a programação paralela.

Devido a este encapsulamento é garantida a inexistência de dependência de dados entre objetos, existindo, desta forma, um paralelismo implícito entre eles.

Paralelismos entre métodos do mesmo objeto podem ser explorados. Entretanto, neste caso como acessos a um mesmo dado podem ocorrer, deve ser efetuada uma análise de dependência entre os métodos. Caso haja acessos a um mesmo dado, mecanismos de exclusão mútua devem ser utilizados.

Várias linguagens para arquiteturas paralelas com memória compartilhada ou distribuída foram propostas. Podem ser citadas a COOL [Cha90] [Cha94], a Mentat [Gri93], a Orca [Bal92] e a Ágata [Sal94] [Sal95a] [Sal95b].

3.1.3. Linguagens funcionais

Nos programas escritos em linguagens funcionais as funções comportam-se como funções matemáticas: o resultado obtido depende apenas de seus argumentos [Bal89].

Não são gerados efeitos colaterais, pois não é permitido que uma função interaja na execução de outra função, através de variáveis globais ou do uso de ponteiros.

Esta ausência de efeitos colaterais é uma característica relevante para a programação para processamento paralelo, pois não existe dependência entre as funções, proporcionando um paralelismo implícito entre elas. Entretanto, quando uma função utiliza o resultado retornado por uma outra função é obrigada a esperá-lo, onerando o desempenho do programa.

Embora, possa existir uma grande quantidade de paralelismos, podem estar incluídas paralelizações de granularidades muito finas. Devido ao custo da implementação do paralelismo, tem-se que quanto maior a granularidade das porções de código paralelas, maior é o desempenho obtido. Para não permitir a ocorrência de paralelizações de granularidades muito finas é necessário dispor de uma forma para indicar as expressões que serão processadas paralelamente.

3.1.4. Linguagens lógicas

A programação lógica é um paradigma de programação onde os programas são expressos como regras lógicas. Uma propriedade das linguagens de programação lógica, presente em sua semântica, é a independência da ordem em que são processadas diferentes operações durante a execução do programa. Esta independência permite que estas operações sejam executadas paralelamente[Pon94].

Algumas propostas de linguagens lógicas para processamento paralelo têm sido apresentadas, explorando implicitamente o AND-paralelismo e o OR-paralelismo [Pon94].

3.2 - Programação Procedural baseada em Variáveis Compartilhadas

O paralelismo pode ser explorado em diversas granularidades. Entende-se aqui como granularidade . Tem-se em um programa implementado em uma linguagem procedural as seguintes fontes de paralelismo:

- Subrotinas: módulos independentes de computação podem ser implementados como subrotinas independentes, que podem ser executadas paralelamente. Tais módulos podem ser especificados como macrotarefas. Tem-se aqui um paralelismo de granulação grossa;
- Laços: iterações de laços independentes podem ser executadas paralelamente. Dependendo do processamento envolvido em cada iteração tem-se um paralelismo de granularidade grossa ou média;
- Blocos básicos: blocos podem ser executados paralelamente se não existir dependências entre blocos. Um bloco básico é uma sequência de comandos que não contém desvios. Conforme os processamentos envolvidos tem-se um paralelismo de granularidade grossa ou média.
- Comandos ou operações: comandos ou operações podem ser processadas simultaneamente,

através de um paralelismo de granularidade fina.

- Instruções: a execução paralela de múltiplas instruções é usualmente providenciada dentro de cada processador através de múltiplas unidades funcionais “pipelining”. Tem-se aqui um paralelismo de granularidade fina.

3.2.1. Macrotarefas e Microtarefas

A paralelização de um programa pode ser realizada particionando-o em múltiplos subprogramas, sendo esta paralelização denominada "multitasking"[Geh92] [Guz86] ou "macrotasking". Várias linguagens têm sido propostas oferecendo este modelo.

Uma paralelização do programa em uma granularidade mais fina é oferecida pelo modelo que utiliza a técnica denominada “microtasking” [Tha88] que divide o trabalho a ser executado em microtarefas. Esta paralelização está presente nos laços paralelos e blocos paralelos oferecidos por algumas linguagens de programação paralela.

3.2.2. “Multitasking”

A paralelização de um programa pode ser realizada particionando-o em múltiplos subprogramas, sendo esta paralelização denominada “multitasking” [Geh92] [Guz96]. Várias linguagens têm sido propostas oferecendo este modelo. A figura 2.1 ilustra o modelo.

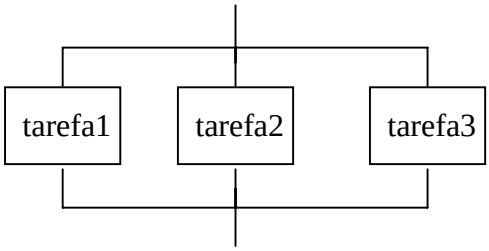


Figura 3.1 - “Multitasking”.

3.2.3. “Microtasking”

Uma paralelização do programa em uma granularidade mais fina oferecida pelo modelo que utiliza a técnica denominada “microtasking” [Tha88] que divide o trabalho a ser executado em microtarefas. Microtarefa é uma porção de código seqüencial, contida em um laço, cujas iterações são executadas paralelamente, ou em um bloco de comandos que é executado paralelamente a outros blocos. A figura 2.2 ilustra o modelo.

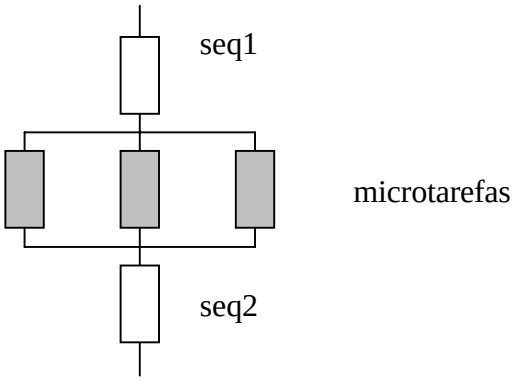


Figura 3.2 - “Microtasking”.

3.2.4. “Macrotasking” e “microtasking”

Pode-se particionar um programa em múltiplas tarefas, sendo cada uma das tarefas particionadas em múltiplas microtarefas. Neste modelo que combina “multitasking”com “microtasking”o particionamento em múltiplas tarefas é conhecido como “macrotasking”e as tarefas como “macrotarefas. A figura 3 ilustra o modelo.

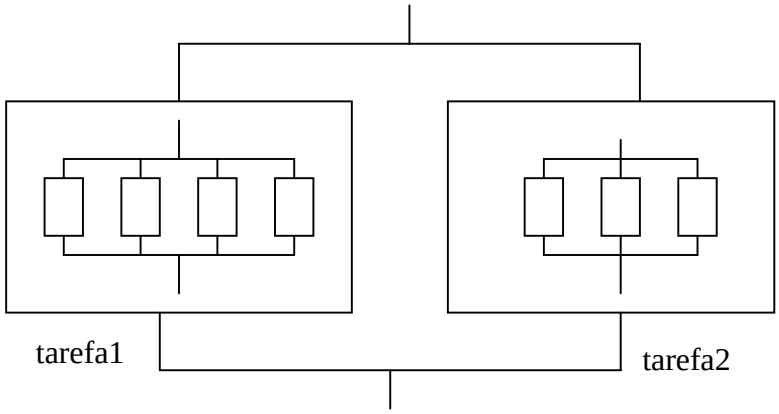


Figura 3.3: “Macrotasking” e “Microtasking”.

3.2.5. Sistemas com memória compartilhada distribuída (DSM)

Dada a menor complexidade da programação baseada em modelos com variáveis compartilhadas em relação a modelos com passagem de mensagem, há cerca de uma década, estão em desenvolvimento pesquisas que buscam oferecer mecanismos que suportam variáveis compartilhadas em multicomputadores.

Em sistemas multicomputadores, cópias nas memórias locais dos dados compartilhados permitem que seus acessos sejam efetuados eficientemente. Esta abordagem, chamada “caching”, cria entretanto, o problema denominado consistência de “cache”, que ocorre quando um processador atualiza um dado compartilhado. Como cópias deste dado podem estar presentes em outros nós, estes devem ser mantidos consistentes com a versão do dado mais recente, não permitindo que um processador obtenha um valor que não foi atualizado.

Modelos de consistência, visando a redução de mensagens e do impacto da latência da comunicação, têm sido propostos.

Três abordagens têm sido utilizadas na implementação de sistemas DSM:

- implementação por “hardware”: estendem técnicas tradicionais de “caching” para arquiteturas escaláveis;
- implementação de bibliotecas e pelo sistema operacional: o compartilhamento e a coerência são obtidos através de mecanismos do gerenciamento de memória virtual;
- implementação pelo compilador e bibliotecas: acessos compartilhados são automaticamente convertidos em primitivas de coerência e sincronização.

Levando em conta que muitos programas paralelos definem seus próprios requisitos de consistência de mais alto nível, requisitos de consistência de memória podem ser relaxados.

Para a construção correta de um programa em um sistema com memória compartilhada distribuída, o programador deve conhecer sobre como as atualizações são propagadas no sistema.

Considerando a visão para o programador as semânticas de coerência de memória podem ser apresentadas em modelos de consistência.

A semântica para coerência de memória mais intuitiva é a apresentada pela consistência estrita (“strict consistency”), onde uma leitura retorna o valor mais recente do dado [Nit91].

Como “o mais recente” é um conceito ambíguo em um sistema distribuído, alguns sistemas DSM providenciam uma forma reduzida de coerência de memória. Entretanto, se o usuário necessita de uma forma de coerência forte, não deve utilizar sistemas que oferecem coerência fraca.

Os modelos de consistência propostos e apresentados na literatura, podem ser listados, como segue:

- **Consistência estrita:** uma leitura retorna o valor mais recentemente escrito [Nit91];
- **Consistência seqüencial:** o resultado de uma execução é semelhante ao obtido pelo entrelaçamento de operações dos nós individuais quando executado em uma máquina seqüencial “multithreaded”. Os acessos devem ser ordenados considerando os acessos de todos os processadores;
- **Consistência de processador:** escritas de um único processador devem ser ordenadas como elas ocorrem. Entretanto, escritas de diferentes processadores podem ser vistas em ordens diferentes;
- **Consistência fraca:** os acessos aos dados são tratados separadamente dos acessos de sincronização, mas requerem que todos os acessos aos dados anteriores sejam feitos antes que o acesso de uma sincronização seja obtido. Cargas e armazenagens entre acessos de sincronizações são livres de ordenação;
- **Consistência “release”:** é uma consistência fraca com dois tipos de operadores: “acquire” e “release”. É garantido que cada tipo de operador é consistente de processador. Um operador “acquire”no início de uma seção crítica é usado para adquirir o direito exclusivo à execução da seção crítica, e o operador “release”para liberá-la, e exportar os dados atualizados para outros nós. O operador “acquire” em um outro nó garante que todas as atualizações realizadas pelo processador estejam consistentes antes que a variável de sincronização tenha sido obtida. Entre os sistemas que oferecem este modelo de consistência, estão o Quarks [Uta95] e o Dash [Len92];
- **Consistência “lazy release”:** é um tipo de consistência “release” que busca reduzir o número de mensagens e a quantidade de dados exportados por acessos remotos à memória. Neste modelo as modificações são exportadas apenas quando necessário [Kel92], ou seja, apenas quando ocorre um acesso aos dados, através do operador “acquire”. Esta consistência garante apenas que um processador verá todas as modificações que precedem o acesso “acquire”. Uma modificação precede um “acquire” se ocorre antes de algum “release”, tal que, existe uma cadeia de operações “release-acquire” sobre a mesma variável de sincronização (“lock”) terminando no “acquire” corrente. O sistema Treadmarks [Kel92] [Kel93] oferece este modelo de consistência;
- **Consistência “entry”:** é utilizada a relação entre variáveis de sincronização específicas que protegem as seções críticas e os acessos aos dados compartilhados nelas efetuados [Ber91]. Uma seção crítica é delimitada por um par de acessos de sincronização à uma variável de sincronização s. Um acesso “acquire” a *s* é usado para obter o acesso a um conjunto de dados compartilhados. Um acesso “release” à variável *s* no final da seção crítica providencia a sua liberação. São permitidos múltiplos acessos a dados compartilhados para leitura, através dos acessos de sincronização que podem ser especificados como exclusivos ou não exclusivos.

Uma vantagem do modelo de consistência “entry” em relação à consistência “release” é que somente os acessos guardados pela variável de sincronização precisam ser liberados para a permissão de acessos remotos, enquanto na consistência “release” todas as atualizações devem ser efetuadas antes que a liberação seja efetivada. Isto reduz o tempo para executar uma seção crítica.

Uma outra vantagem do modelo “entry” é a redução de comunicações, pois apenas os processadores que utilizarão os dados irão receber as suas atualizações. Não se tem entretanto esta vantagem, em relação à consistência “lazy release”, pois neste modelo as modificações também são importadas quando necessárias. Contudo, o fato de importar as modificações apenas quando necessárias, adiciona ao custo de acesso a latência da comunicação das modificações. Para reduzir este custo adicional são necessários mecanismos de pré-busca (“pre-fetching”).

Um exemplo de implementação de memória compartilhada distribuída através de uma biblioteca e do compilador, é o sistema SAM desenvolvido em Stanford University [Sca93], que oferece uma biblioteca implementada utilizando o sistema PVM (Parallel Virtual Machine) e providencia o suporte para o compilador da linguagem Jade [Rin93].

3.3 - Programação Procedural baseada em Passagem de Mensagem

Para que uma aplicação (programa) possa ser executado por um computador paralelo com memória distribuída é necessário distribuir trechos de seu código entre os processadores. tem-se um modelo chamado SPMD (*Single Program Multiple Data*). Algumas bibliotecas de programação utilizam o modelo SPMD, enquanto que outras permitem que trechos heterogêneos de código sejam distribuídos pelos processadores.

As subunidades de um programa comumente recebem denominações como *tarefas*[1], ou processos. Ambos os termos serão utilizados indistintamente neste capítulo. Tais subunidades fazem parte de um mesmo programa, e portanto devem trocar dados, resultados e informações de sincronização.

Nos computadores com memória compartilhada, todos os processadores podem ler e escrever dados sobre uma área de memória comum (fig.3.4). Nesses computadores, a memória compartilhada é utilizado para armazenar variáveis compartilhadas entre os processos de um programa paralelo.

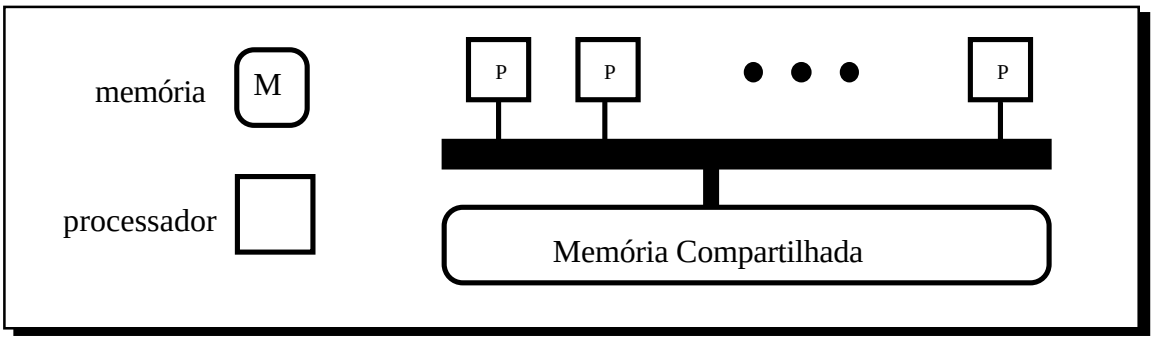


Figura 3.4 - Modelo de arquitetura de um multiprocessador.

Nos multiprocessadores com memória fisicamente compartilhada todos os mecanismos de comunicação entre as tarefas de um programa paralelo (*semáforos*, *monitores*, *eventos*, *mensagens* e variáveis compartilhadas entre tarefas) são abstrações do modelo de variáveis (ou seja, posições de memória) compartilhadas.

[1] do inglês *tasks.*

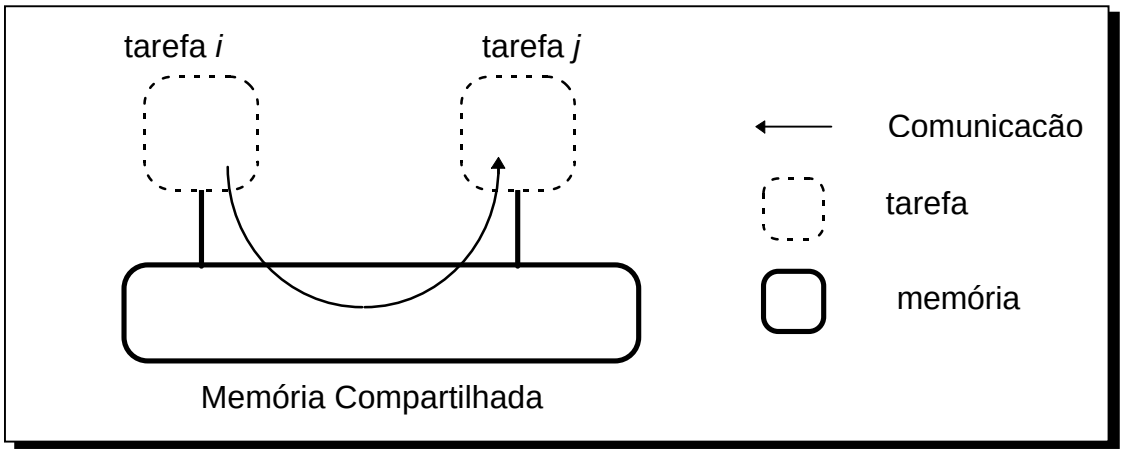


Figura 3.5 - Comunicação entre tarefas, através da memória compartilhada.

Nos chamados **multicomputadores,** os processadores encontram-se distribuídos fisicamente, e somente podem acessar a sua própria memória local. Cada par composto de um processador e sua respectiva memória local constituem um **nó**. Todos os nós de um multicomputador são interligados por algum tipo de barramento, ou sistema de interconexão, através do qual pode-se enviar e receber mensagens. Assim, o modelo mais elementar de programação de multicomputadores é aquele no qual os processos que compõem um programa paralelo são distribuídos pelos nós da rede, e trocam *mensagens* entre si. Esta modalidade comumente é chamada de programação distribuída.

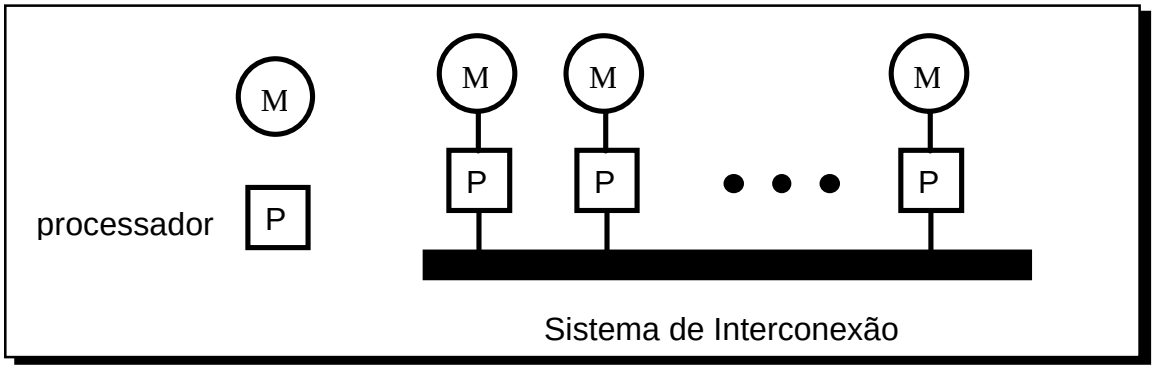


Figura 3.6 - Modelo de arquitetura com memória distribuída.

Um multicomputador pode ser ter seus nós contidos em uma mesma placa, como os multicomputadores compostos por uma rede de *transputer*, ou em placas ligadas por barramentos/chaveadores contidas em um único gabinete, como os sistema Intel *Paragon* e Meiko-CS2. Uma terceira forma, bastante interessante sob vários aspectos, é a implementação de multicomputadores como redes locais de estações de trabalho e de PC's, comumente chamados de **aglomerados**. Esta última é a forma mais barata e comum de implementar um multicomputador, e diversas linguagens e bibliotecas de programação distribuída têm sido implementadas para tais sistemas.

3.3.1. Comunicação por passagem de mensagem

Existem outros modelos de programação distribuída, tais como *Rendezvous* e RPC (*Remote Procedure Call*), entretanto, esta seção trata da comunicação entre tarefas através de passagem explícita de mensagens. Esta forma de comunicação é a mais utilizada em bibliotecas de programação distribuída.

Uma troca de mensagem envolve pelo menos dois processos, o *transmissor* que envia a mensagem, e o *receptor*, que a recebe. Geralmente isto é feito através de primitivas do tipo SEND e RECEIVE [9], em duas principais modalidades: a comunicação *síncrona* e a comunicação *assíncrona*.

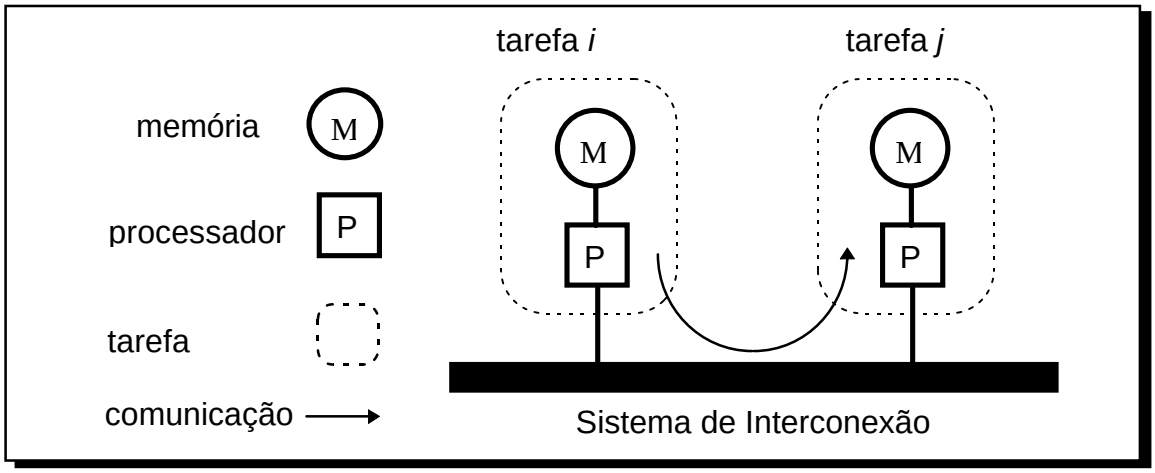


Figura 3.7 - Comunicação por passagem de mensagens, através dos canais de comunicação.

3.3.2. Comunicação síncrona (com bloqueio)

Nesta modalidade, o transmissor envia a mensagem para o receptor, e aguarda até que este último sinalize o recebimento da mesma. Se o receptor não estiver pronto para receber a mensagem, o transmissor é bloqueado temporariamente. Se o receptor quiser receber a mensagem antes que esta tenha sido enviada, ele também será bloqueado temporariamente. Quando ambos estiverem prontos, e só então, o transmissor envia a mensagem, e o receptor envia de volta um sinal confirmando seu recebimento. Imediatamente os dois são desbloqueados e podem seguir seu fluxo de execução normalmente.

Esta modalidade de comunicação não exige a presença de um *buffer* para armazenamento temporário da mensagem.

3.3.3. Comunicação assíncrona (sem bloqueio)

Esta modalidade permite que o transmissor envie a mensagem, e prossiga em seu fluxo normal de execução sem sofrer nenhum tipo de bloqueio. Caso o receptor ainda não esteja pronto para receber a mensagem, esta deve permanecer armazenada temporariamente em um *buffer*. Caso o receptor esteja pronto para receber uma mensagem que ainda não foi enviada, ele então deverá permanecer bloqueado temporariamente. Os sistemas que implementam esta modalidade geralmente oferecem primitivas adicionais que apenas verificam se alguma mensagem chegou ou não, sem bloquear o processo receptor.

Apesar de esta modalidade aparentar melhor desempenho porque não impõe tantas situações de bloqueio, existem custos adicionais para fazer o tratamento dos *buffers* que são necessários.

3.3.4. Suporte à programação distribuída

Os objetivos da programação distribuída podem variar, como por exemplo, detectar possíveis falhas parciais[2] e recuperar o último estado consistente do sistema, obter máximo desempenho, ou ainda, garantir que o tempo de resposta do sistema obedeça certos parâmetros estabelecidos[3].

A programação de sistemas com memória distribuída pode ser feita de três maneiras :

- Programar diretamente sobre as primitivas de controle do *hardware* como vetores de interrupções ou interfaces de comunicação com dispositivos.
- Utilizar uma linguagem seqüencial disponível, fazendo chamadas às primitivas de um sistema operacional ou bibliotecas de rotinas. O sistema operacional pode ser um núcleo (dispondo

[2] Sistemas tolerantes a falhas.
[3] Sistemas de tempo real.

apenas do gerenciamento e intercomunicação de processos), um sistema operacional de rede, ou ainda, um sistema operacional distribuído completo. São exemplos dessa modalidade, o sistema operacional AMOEBA [Bal93], que dispõe de primitivas para a comunicação entre processos, as bibliotecas PVM, MPI, P4, TCGMSG e outros, que trabalham sobre ambientes do tipo UNIX acrescidos de *software* de rede local.

- Utilizar linguagens especialmente projetadas para a programação distribuída. Além de libertarem o programador de qualquer contato direto com o *hardware* e o sistema operacional, essa alternativa oferece um modelo abstrato de mais alto nível, sobre o sistema distribuído.

3.3.5. Bibliotecas de programação distribuída

Com a finalidade de explorar a capacidade de processamento oferecida pelos aglomerados de estações de trabalho, surgiram diversos ambientes de programação [Duk93] visando a exploração do paralelismo, tais como PVM, MPI, TCGMSG, P4 e Express.

3.4 - Programação Procedural baseada no Paralelismo de Dados

O paradigma de programação paralela baseada no paralelismo de dados (em inglês, data parallel programming) representa uma forma de exploração de operações simultâneas sobre grandes conjuntos de dados, ao invés de especificar várias tarefas paralelas de controle. Este estilo de programação é adotada em máquinas contendo centenas a milhares de processadores.

A concorrência então não é especificada na forma de diferentes operações simultâneas, mas na aplicação da mesma operação sobre múltiplos elementos de uma estrutura de dados. Um exemplo de uma destas operações é “some 2 sobre todos os elementos da matriz X” (fig.3.8). Um programa paralelo baseado no paralelismo de dados é um conjunto destas operações. Como cada operação é uma unidade computacional distinta das demais, pode-se afirmar que a granularidade deste paradigma é fina.

Este estilo de programação é adotado em máquinas paralelas do tipo SIMD, ou MIMD programadas sob o estilo SPMD. Uma das primeiras máquinas comerciais a adotarem este paradigma de programação foi o Connection Machine Modelo CM-1, introduzido pela Thinking Machines Corporation em 1986. A programação no CM-1 utilizava a linguagemde programação C*. Outros exemplos de linguagens de programação que suportam o paralelismo de dados são o DataParallel C, pC++ e, mais recentemente, o HPF [Fos95].

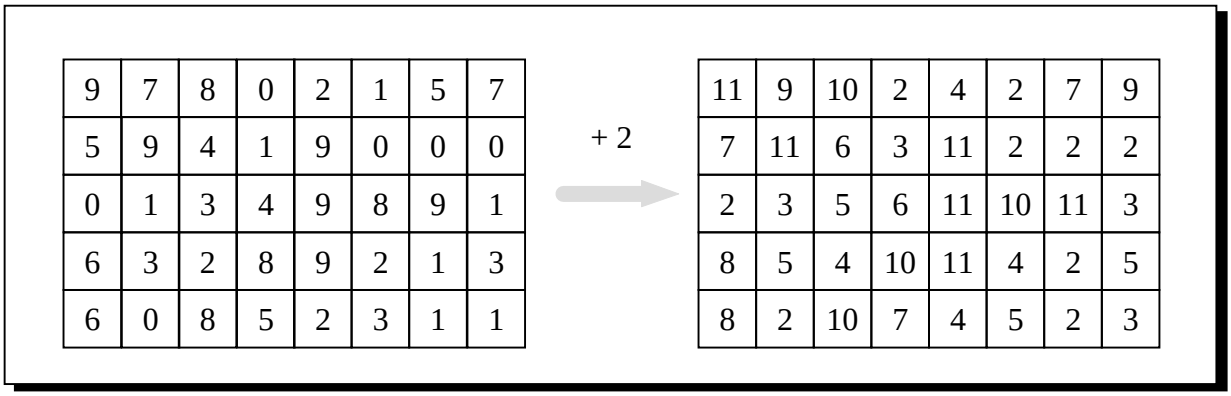


Figura 3.8 - Exemplo de uma operação com paralelismo de dados.

3.4.1. Alguns aspectos da linguagem C*

C* é uma extensão da linguagem de programação ANSI C e foi projetada para suportar a programação paralela baseada no paralelismo de dados. Os recursos extra incluídos na linguagem incluem recursos para broadcasting, redução e comunicação entre processadores. Uma varíavel pode ser definida como sendo uma variável paralela. Associado a cada variável paralela está o seu molde

(shape), que define como os elementos desta variável paralela é distribuída pelos processadores da máquina. Por exemplo,

```
shape [16384] empregados;
```

é um comando que define um molde unidimensional com 16348 posições chamado empregados.

Após um molde ser especificado, as variáveis paralelas podem ser especificadas de acordo com certo molde. Por exemplo, o comando

```
unsigned int: empregados cadastro;
```

especifica uma variável paralela cadastro como sendo uma variável do tipo unsigned int.

Uma operação com paralelismo de dados é especificado através do comando with. Adotando o molde corrente, with é adotado para a maioria das operações sobre variáveis paralelas. Por exemplo,

```
shape [16384] empregados;
unsigned int: empregados cadastro;
with (empregados)
    cadastro = 0;
```

Outro recurso acrescido a C* são os operadores de redução. Por exemplo, no trecho abaixo

```
int: numeros x;
soma =  (+=x);
```

a variável soma conterá a soma de todos os elementos ativos da variável paralela x.

3.4.2. Um exemplo de um programa com paralelismo de dados em C*

Apresenta-se aqui um pequeno exemplo onde é calculado o valor aproximado de π, através do cálculo da área da curva $\frac{4}{1+x_i^2}$ entre 0 e 1 realizado pela computação da somatória

$$\pi \approx \frac{1}{N}\sum_{i=0}^{N}\frac{4}{1+x_i^2}, \quad \text{onde } x_i = \frac{\left(1+\frac{1}{2}\right)}{N}.$$

A solução com paralelismo de dados para este problema associa um processador virtual a cada intervalo. Cada um destes processadores virtuais calcula a área do retângulo associado e então todas as áreas individuais são somadas para obter-se a área total.

```
/* Estimativa de pi pelo método dos retângulos */
#define INTERVALOS 400000
shape [INTERVALOS] trecho; /* declaração do molde */
double: trecho x;          /* ponto médio do retângulo no eixo x */
main ()
{
     double soma;          /* soma total das áreas */
     double largura;       /* largura dos intervalos */
     largura = 1.0 / INTERVALOS;
     with (trecho)         /* seleciona molde corrente */
     {
          x = (pcord(0)+0.5) * largura;
          soma = (+= (4.0 / (1.0+x*x)));
     }
     soma *= largura;
     printf ("Estimativa de pi = %14.12f\n", soma);
}
```

Este exemplo mostra a execução paralela do cálculo da área de um retângulo em todos os processadores da máquina, para posterior somatória para a obtenção do cálculo da aproximação do valor de π.

Capítulo 4 - EXEMPLOS DE SISTEMAS DE PROGRAMAÇÃO

A programação de computadores paralelos requer o auxílio de linguagens e de uma série de ferramentas. Este capítulo mostra o que está disponível em duas máquinas comerciais: a Silicon Graphics Power 4D [Bau92] e a Meiko CS-2 [Mei95].

4.1 - Programação na SGI Power 4D

A linha de máquinas da Silicon Graphics Power 4D e, mais recentemente, a linha Power Challenge oferecem um conjunto de ferramentas de apoio a programação paralela que inclue compiladores, rotinas de biblioteca, analisadores de código e paralelizadores automáticos, gerentes para compilação paralela, analisadores de desempenho, utilitários para profiling e depuradores de código.

O sistema operacional IRIX oferece uma série de recursos adicionais em relação ao ambiente UNIX padrão. Desta forma, descreve-se aqui apenas aqueles tópicos que são relacionados ao desenvolvimento de programas paralelos para a plataforma Silicon Graphics.

4.1.1. Analisadores de código e paralelizadores automáticos

A forma mais simples para a paralelização de código sequencial é submeter o programa a um analisador e paralelizador automático. O IRIX oferece duas ferramentas para tal estratégia: Power C e Power Fortran Analysers. Estas ferramentas são:

- pré-processadores de código que auxiliam na otimização e paralelização de código C e Fortran
- analisam e reconhecem fontes de paralelismo; podem ser ativados como um passo no processo de compilação tradicional
- inserem automaticamente construções de paralelismo (usam um subconjunto das Power Languages)
- executam várias otimizações sequenciais com o intuito de melhorar o desempenho global do programa
- podem ser aplicadas somente para analisar um código fonte ou como uma ferramenta para paralelização automática de programas
- podem gerar relatórios de análise e manter o código paralelizado para posterior análise pelo programador

De uma maneira geral, o PCA ou PFA pode ser ativado de duas maneiras:

1. diretamente: com a chamada direta do comando
   % /usr/lib/pca prog.c
2. através do compilador
   % cc -pca keep prog.c -o prog.o

De uma maneira geral, o processo de análise e paralelização pode ser esquematizado como apresentado pela figura x abaixo.

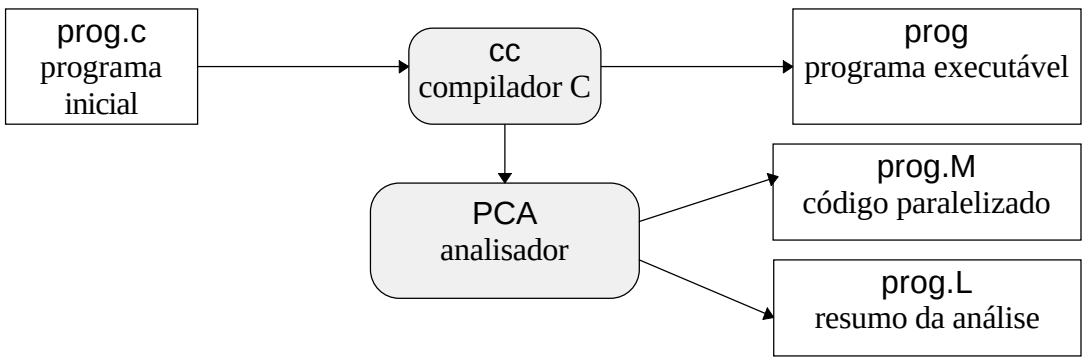


Figura 4.1 - Esquema do uso de uma ferramenta de análise e paralelização automática.

4.1.2. Compiladores

Outra forma de paralelização é o programador especificar as fontes de paralelismo explicitamente através do uso de diretivas (C) ou comentários (Fortran). Este recurso é implementado através das chamadas Power Languages. São características deste recurso:

- suporte a programação paralela tipo SPMD
- paralelização de laços
- especificação de seções independentes de código
- especificação de seções críticas
- suporte de mecanismos sincronização

Os principais pragmas de compilação disponíveis para programas C são apresentadas na tabela 4.1 abaixo.

Tabela 4.1 - Alguns pragmas de compilação do IRIX.

pragma	descrição
parallel	especifica uma seção de código paralelo
pfor	especifica um laço paralelo
independent	cria um bloco de código que é executado por um único thread
critical	especifica uma seção crítica
synchronize	cria uma barreira

O seguinte pequeno trecho de código ilustra o uso dos pragmas.

```
double total_pi = 0.0;
int main()
  {
  ...
#pragma parallel
#pragma shared (total_pi)
#pragma local(i,x,local_pi)
{
#pragma pfor iterate (i=0;
MAXRET; 1)
 for ( i = 0; i< MAXRET; i++)
    {
    x = ((i-0.5) * largura);
      /* calcula x */
    local_pi = local_pi + (4.0 /
( 1.0 + x * x));
    }
  local_pi = local_pi * largura;
#pragma critical
  {
  total_pi = total_pi + local_pi;
  }
}
  printf ("valor de pi = %d\n",
total_pi);
}
```

4.1.3. Rotinas de biblioteca

A terceira forma de se obter um programa paralelo no IRIX é através do uso de rotinas de biblioteca. Estas rotinas são uma extensão da biblioteca fornecida pelo sistema Sequent. Podendo ser utilizadas em programas C ou Fortran, as rotinas de biblioteca podem paralelizar as seguintes situações:

- programas com particionamento de dados
- chamadas de subrotinas e de funções
- E/S em programas C
- algumas situações de dependência de dados com o uso de sincronização

As principais rotinas de biblioteca de paralelismo disponíveis são apresentadas na tabela 4.2 abaixo.

Tabela 4.2 - Algumas rotinas da biblioteca de paralelismo do IRIX.

rotina	descrição
m_fork	cria processos paralelos
m_get_myid	retorna identificador do processo paralelo
m_kill_procs	termina execução dos processos paralelos
m_next	atualiza contador global do sistema utilizado para sincronização dos processos
m_sync	sincroniza todos os processos paralelos em algum ponto do código do programa
m_get_numprocs	retorna número de processadores disponíveis
m_set_procs	determina quantos processadores serão utilizados na execução paralela do programa
m_lock	trava semáforo global
m_unlock	destrava semáforo global

Um pequeno trecho de código que ilustra o uso destas rotinas é mostrado a seguir.

```
double total_pi = 0.0;
int main()
  {
  int I, nrets = MAXRET; /* # de
retangulos */
  ...
  num_procs = m_get_numprocs();
  chunk = nrets/num_procs;
  /* execução paralela */
  m_fork (parte_pi, nrets);
  m_kill_procs();
  printf (“valor de pi = %d\n”,
\
     total_pi);
void parte_pi ( int nrets)
   {
  double local_pi = 0.0;
  ...
  my_id = m_get_myid();
  ...   /* calcula inicio e fim
*/
  for ( i = start; I< fim; I++)
    {
    x = ((i-0.5) * largura);
     /* calcula x */
    local_pi = local_pi + (4.0 /
( 1.0 + x * x));
    }
  local_pi = local_pi * largura;
  m_lock();
  total_pi = total_pi +
local_pi;
  m_unlock();
}
```

4.1.4. Gerentes de compilação paralela

O sistema IRIX oferece uma extensão ao gerente de compilação padrão do UNIX (pmake) que permite a especificação e a execução paralela do processo de compilação dos módulos de um programa. Após a declaração das relações de dependência (makefile), o utilitário pmake pode criar vários programas simultaneamente.

O objetivo de pmake é aproveitar os recursos de paralelismo disponíveis em ambientes multiprocessadores e agilizar o processo de compilação. Por default, pmake cria 2 tarefas em ambientes monoprocessadores e 4 tarefas em um multiprocessador. À medida que novas dependências são definidas, novas tarefas são criadas. Se não houver restrição de memória, podem ser criadas até (3 x número de processadores) tarefas paralelas.

4.1.5. Analisadores de desempenho

Para a medida do tempo de execução de uma aplicação paralela, IRIX oferece o utilitário timex. Ele é uma extensão do utiltário padrão time. Sua saída oferece uma resolução de tempo de 10ms, e é dividida em:

- *real*: tempo total de execução em segundos.
- *user*: quantidade de tempo de processamento em modo usuário.
- *sys*: quantidade de tempo de processamento de serviços de sistema.

Por exemplo

```
% timex cholesky
<saída do programa de decomposição>
real 5.45
user 3.12
sys  0.86
```

Opcionalmente, timex pode ser utilizado para a obtenção do uso de memória do sistema.

Outras ferramentas que permitem monitorar o uso de memória pelos diversos programas são o top e osview. O comando top monitora continuamente quais processos estão em execução e quais os recursos em uso. As estatísticas mais interessantes são:

- estado do processo
- tempo total de execução (em segundos de CPU)
- tamanho do processo (em páginas)
- conjunto residente (RSS) - número de páginas residentes em memória
- prioridade do processo

O intervalo de amostragem default é de 5 segundos, mas pode ser modificado através da opção -I. A seguir tem-se um exemplo de saída

```
IRIX wizard 5.3 11091812 IP22 Load[0.28, 0.25, 0.27] 23:18:07   60
procs
    user  pid   pgrp    %cpu proc  pri  size   rss     time  command
malmeida  850    835    5.60    5  +39  1247   682    13:16
gr_osview
malmeida  909    896    1.01    *   26  3565  2445     0:27
maker4X.exe
    root  792    762    0.88    0   26  3441  2547     8:42  Xsgi
malmeida  950    947    0.05    *   26   741   279     0:02  Xwsh
```

O comando osview permite monitorar a utilização corrente dos recursos de sistema.Sua saída é agrupada em informações sobre: CPU, memória e disco. Por exemplo,

```
 Osview 2.1 : One Second Average    wizard   03/21/95  23:55:04    #8
int=5s
 Load Average         *Wait Ratio              *System Activity
   1 Min      0.475    System Memory           *Swap
   5 Min      0.322      Phys         96.0M    *System VM
   15 Min     0.218       kernel       6.2M    *Memory Faults
 CPU 0 Usage               heap      612.0K    *TLB Actions
   %user       2.93          stream    28.0K    *Graphics
   %sys        3.52         zone      768.0K    *TCP
   %intr       0.39         ptbl        1.0M    *UDP
   %gfxc       0.00        fs ctl       1.4M    *IP
   %gfxf       0.00        fs data      6.6M    *NetIF[ec0]
   %sxbrk      0.00         delwri         0    *NetIF[lo]
   %idle      93.16        free        30.2M    *Scheduler
 CPU 1 Usage               userdata    51.6M    *Interrupts
   %user       20.09       pgallocs         0    *PathName Cache
  ...
```

4.1.6. Profiling

Para auxiliar o processo de otimização de programas, IRIX oferece um conjunto de utilitários para uma coleta detalhada de informações sobre os padrões de execução de programas. Com isto é possível:

- encontrar as seções de código que consomem mais tempo de processamento
- concentrar esforços de paralelização nestas porções

São disponíveis duas técnicas:

- Amostragem do contador de programa (PC Sampling): é um método estatístico, onde o código é interrompido periodiamente e é armazenado o valor corrente do contador de programa para posterior análise.
- Contagem de blocos básicos (Basic block counting): é um método determinístico. O programa é alterado de forma que seja introduzido um código responsável pela contagem do número de ativações de cada um dos blocos básicos.

O processo de análise com o método da amostragem do contador de programa é ilustrado na figura 4.2 abaixo.

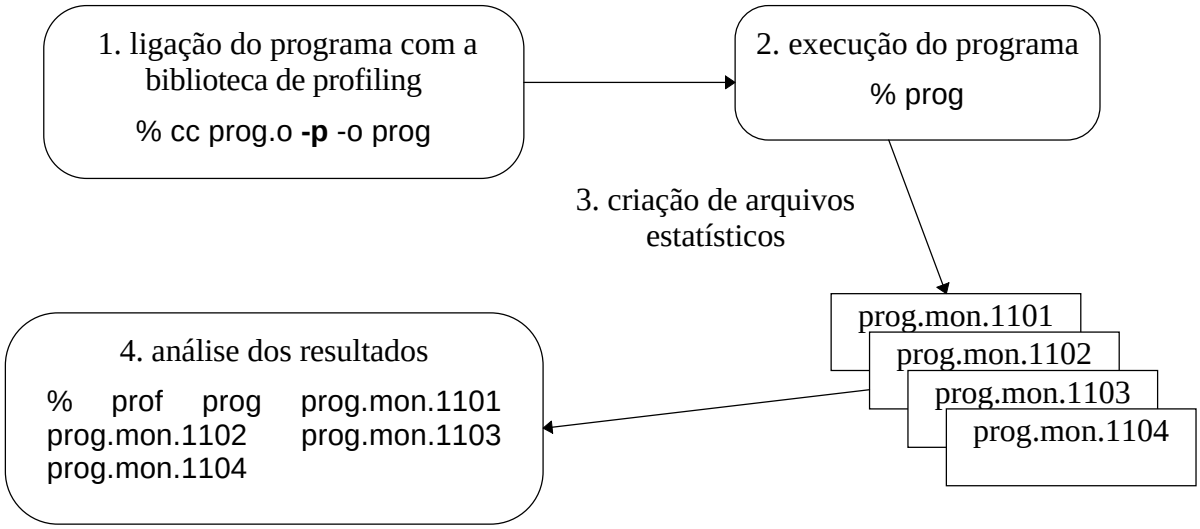


Figura 4.2 - Profiling por amostragem do contador de programa.

Um exemplo de saída da análise é o seguinte:

```
------------------------------------------------------------------------
Profile listing generated Fri Apr 22 08:01:52 1996
    With:       prof prog prog.mon.(1101,1102,1103,1104)
------------------------------------------------------------------------

samples    time    CPU    FPU    Clock  N-cpu    S-interval   Countsize
    414    4.1s  R4400  R4010  100.0MHz  4       10.0ms       2(bytes)

Each sample covers 4 bytes for every 10.0 ms ( 0,24% of 4.1400 sec)
------------------------------------------------------------------------
-p[rocedures] using pc-sampling
 Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.
------------------------------------------------------------------------
file prog.mon.1101:
samples   time(%)      cum time(%)      procedure (file)
    332   3.3s( 80.2)  3.3s( 80.2)    piece_of_pi (prog:prog.c)
     40   0.4s(  9.7)  3.7s( 89.9)       _sprocsp (libc.so.1:sproc.s)
     25  0.25s(  6.0)    4s( 95.9)        barrier (libc.so.1:barrier.c)
     10   0.1s(  2.4)  4.1s( 98.3)         memset (libc.so.1:memset.s)
    414   4.1s(100.0)  4.1s(100.0)          TOTAL
file prog.mon.1102:
samples   time(%)      cum time(%)      procedure (file)
    333   3.3s( 97.4)  3.3s( 97.4)    piece_of_pi (prog:prog.c)
      9  0.09s(  2.6)  3.4s(100.0)        barrier (libc.so.1:barrier.c)
    242   3.4s(100.0)  3.4s(100.0)          TOTAL
...
```

O método da contagem de blocos básico tem um procedimento similar, contendo os seguintes passos:

1. compilação normal do programa
   % cc -o prog prog.c
2. execução do comando pixie
   % pixie prog
3. preparação para execução do programa
   % setenv LD_LIBRARY_PATH .
   % prog.pixie
4. geração dos arquivos de contagem de blocos básicos
   prog.Counts.2225, prog.Counts.2226
5. análise dos resultados
   % prof -pixie prog prog.Count.*

4.1.7. Depuração de código paralelo

O IRIX oferece uma versão melhorada do utilitário dbx para a tarefa de depuração. A estratégia recomendada pela Silicon para a depuração de código paralelo é

- depurar primeiro o código sequencial
- depurar o código paralelo controlando sua execução em um único processador, se possível
- depurar a versão multithreaded

Alguns dos comandos adicionais do dbx são os seguintes:

comando	descrição
set $promptonfork=valor	a variável controla a criação de novos processos e sua notificação ao usuário
active [pid]	imprime processos ativos, ou torna pid um processo ativo
showproc [pid | all]	mostra processos disponíveis para depuração
delproc pid	remove processo do conjunto de processos
addproc pod	acrescenta processo ao conjunto de processos
waitall	espera por todos os processos em execução alcançarem um breakpoint ou terminar sua execução

Uma versão gráfica é disponível através de um pacote opcional chamado Workshop Pro MPF.

4.2. Programação na Meiko CS-2

O multicomputador CS-2 da Meiko é composto de um conjunto de placas com processadores HyperSparc. Estes processadores são organizados em uma estrutura do tipo “fat-tree”.

Existem diversas maneiras de utilizar os recursos de processamento paralelo no sistema Meiko CS-2. Uma delas é um biblioteca proprietária, criada pela própria Meiko, especificamente para o sistema CS-2 : o CSN (*Computing Surface Network*). A biblioteca CSN possui um conjunto de rotinas, com interface para as linguagens de programação C e FORTRAN. CSN se baseia no modelo de troca de mensagens entre processos paralelos, que são executados nos diversos processadores da rede. Tais processos se comunicam através de abstrações, que são chamadas de **transporte** (*transport*). Um transporte é uma conexão lógica que permite a conexão ponto-a-ponto entre dois processos.

A biblioteca CSN dispõe de rotinas para a manipulação (criação, destruição, etc.) de transportes e para o envio (bloqueante e não-bloqueante) e recebimento de mensagens. Para depuração e acompanhamento de programas, sistemas como Paragraph e Alog/Upshot são oferecidos.

4.2.1 PVM na Meiko

A Meiko fornece uma implementação própria da biblioteca PVM, parcialmente compatível com a versão 3.2. O objetivo dessa implementação é aproveitar os recursos de comunicação do sistema CS-2, proporcionando ganhos de desempenho em comunicação. A principal diferença é e eliminação dos *daemons* existentes na implementação original. Em seu lugar, pode-se usar recursos oferecidos por outra biblioteca, o CS-2 *Resource Management System*. Em troca do ganho de desempenho perde-se algumas funcionalidades, como por exemplo, a possibilidade de ter parte da aplicação em um processador externo ao CS-2, e ausência de algumas rotinas existentes no PVM padrão original.

4.2.2 Outras bibliotecas disponíveis

Existem outras bibliotecas disponíveis, como o RMS *Resource Management System*, que oferece serviços de gerenciamento (consulta, alocação, e controle) de recursos do sistema. A biblioteca Elan também é oferecida com o sistema CS-2, e oferece recursos para a programação paralela, tais como o gerenciamento de processadores de DMA, manipulação de processos, manipulação de

eventos, etc.Um programa pode utilizar serviços do RMS, Elan e PVM separadamente ou de maneira combinada.

4.2.3 Outras ferramentas

A Meiko CS-2 dispõe de todas as ferramentas padrão UNIX: compiladores C e Fortran, depuradores de código, ferramentas de profiling, etc.

Capítulo 5 - ALGUMAS LINGUAGENS E BIBLIOTECAS DE PROGRAMAÇÃO PARALELA

Este capítulo apresenta duas linguagens de programação paralela, HPF e CPAR, descrevendo os recursos disponíveis para a expressão de paralelismo. Além de apresentar a sintaxe dos comandos, exemplos ilustram suas aplicações. A biblioteca PVM é descrita a seguir, onde se apresenta uma alternativa para o desenvolvimento de progromas paralelos.

5.1 - High Performance Fortran (HPF)

Como visto anteriormente, há uma grande variedade de máquinas paralelas, cada uma delas com seu próprio modelo de programação paralela e sua respectiva linguagem. Isto faz com que o desenvolvimento de aplicações portáveis sobre um grande número de máquinas paralelas seja uma tarefa muito difícil, senão impossível. Tendo isto em mente, um conjunto de grupos de pesquisa e de empresas iniciaram um esforço comum para a definição de uma linguagem portável para a programação de aplicações científicas para multiprocessadores em geral.

Depois de uma série de reuniões e discussões, em 1993, foi definida a primeira versão da linguagem High Performance Fortran (HPF). Desde então várias empresas têm lançado diversos produtos, como compiladores, conversores e treinamentos. IBM, Fujitsu, Hitachi e Intel são alguns exemplos de companhias com produtos HPF anunciados. [Fos95] [Lov93]

O modelo de programação paralela adotado pela HPF é o paralelismo de dados. O paralelismo é proveniente de operações independentes sobre os elementos de grandes estruturas de dados. Como a HPF é baseada na linguagem Fortran 90, estas estruturas de dados são normalmente *arrays* e as operações paralelas, aritmética sobre *arrays*. Como tipicamente as máquinas possuem dezenas (ou centenas) de processadores e as aplicações científicas, milhares (ou milhões) de elementos, a execução é realizada com a distribuição destes elementos entre os processadores. Isto é especificado em programas HPF através de diretivas de mapeamento de dados de alto nível.

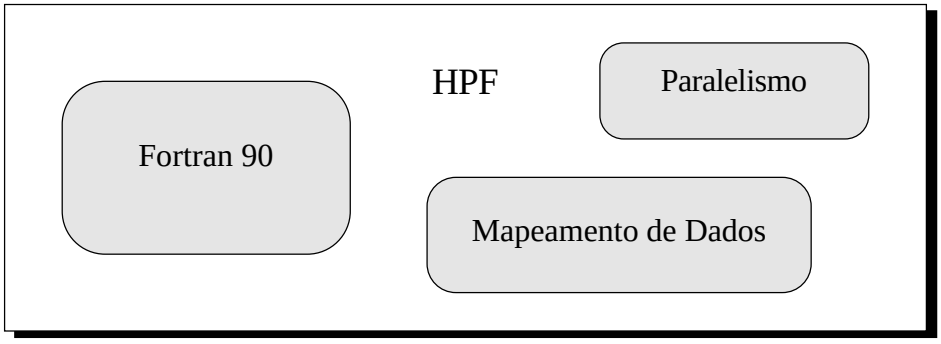


Figura 5.1 - Recursos da linguagem HPF.

De uma forma esquemática (fig.5.1), os recursos da linguagem HPF pode ser descrita como sendo constituída por:

- recursos da linguagem Fortran 90
- suporte para especificação de paralelismo
- suporte para mapeamento de dados

Durante o ano passado (1995), uma série de reuniões iniciaram os trabalhos para a definição da HPF 2.0, onde se tentou acrescentar novas características à linguagem. Espera-se a divulgação de uma versão preliminar do documento de especificação (draft) na conferência Supercomputing'96 a ser realizada em Pittsburgh.

5.1.1. Descrição sucinta da linguagem Fortran 90

A linguagem Fortran 90, o último padrão ISO para Fortran, é uma grande extensão sobre a Fortran 77 (ANSI X3.9-1978). De uma maneira geral, estas extensões tornaram a Fortran uma linguagem mais atualizada como as linguagens C e C++, além de oferecer recursos exclusivos, como as operações sobre *arrays*.

As extensões Fortran 90 sobre a Fortran 77 podem ser classificadas em:

- construções de arrays
- alocação dinâmica de memória e variáveis automáticas
- apontadores
- novos tipos de dados e estruturas
- novas funções intrínsecas
- novas estrutruras de controle
- interfaces de procedimentos

Construções de arrays

As construções de arrays permitem programas Fortran 90 especificarem operações sobre todos os elementos dos arrays. Isto permite que o compilador possa traduzir estas construções em instruções vetoriais ou paralelas, dependendo da arquitetura da máquina. A linguagem especifica somente que a computação é executada em paralelo. Por exemplo, se se deseja somar os elementos de dois vetores, A e B, basta em Fortran 90, especificar esta operação como uma simples soma ao invés de implementá-la com um tradicional laço. Ou seja, basta escrever

```
      A = A + B
```

no lugar do laço Fortran 77

```
      DO 10 I=1,N
10      A(I) = A(I) + B(I)
```

Este recurso não é limitado para arrays unidimensionais, podendo ser empregado para arrays genéricos.

Uma das confusões mais comuns sobre esta característica se refere ao comando

```
      REAL A(N,N), B(N,N), C(N,N)
      ...
      C = A * B;
```

onde é calculado o produto elemento a elemento das matrizes A e B e não a multiplicação de ambas as matrizes. Este caso é coberto por uma função intrínseca Fortran 90.

Caso não se deseje operar sobre todos os elementos, é possível se especificar uma "seção de array". Uma seção é composto por um subconjunto de elementos de array, possivelmente não contíguos. Sua especificação utiliza a notação de ternos sob a forma a:b:c, que se lê "elementos entre a e b, tomados com incrementos de c". Por exemplo

```
      REAL  X(10,10), Y(100), Z(5,5,5)
      ...
      X(10,1:10)      = Y(91:100)
      X(10,:)         = Y(91:100)
      Z(10:1:-2,5)    = Y(1:5)
```

o primeiro comando atribui os últimos dez elementos de Y à 10ª linha de X. O segundo comando especifica o mesmo, de um modo diferente. O sinal ":" indica ao compilador a faixa completa da 2ª dimensão da matriz.O terceiro comando atribui os primeiros cinco elementos de Y aos elementos 10, 8, 6, 4 e 2 da coluna 5 de Z.

Alocação dinâmica de memória

No passado, programadores Fortran adotavam a estratégia de pré-alocar um grande espaço extra para tratar as necessidades de alocação temporária de memória. Isto fazia com que se desperdiçasse

memória. Fortran 90 elimina a necessidade deste artifício com o acréscimo dos arrays alocáveis dinamicamente.

A alocação dinâmica de arrays permite ao programador especificar o espaço a ser alocado em função do conteúdo de outras variáveis do programa. Por exemplo, é possível ao programa determinar as dimensões antes de alocar memória.

```
INTEGER M,N
REAL, ALLOCATABLE, DIMENSION (:,:) :: X
...
WRITE(*,*) 'DIMENSOES DE X'
READ (*,*) M,N
ALLOCATE (X(M,N))
...
processamento utilizando X
...
DEALLOCATE (X)
...
```

No exemplo acima, o comando ALLOCATE cria uma matriz MxN,que é liberada mais tarde pelo comando DEALLOCATE. Como na linguagem C, é muito importante devolver espaço de memória logo após a sua utilização devido ao risco de se esgotar a memória virtual disponível ao programa.

Funções intrínsecas

As novas funções intrínsecas introduzidas por Fortran 90 se referem principalmente a operações sobre arrays. Além da extensão natural das antigas funções intrínsecas para operarem sobre arrays, como SQRT, LOG, as novas funções podem ser agrupadas principalmente em:

- reduções: funções que operam sobre arrays e retornam um único valor. São exemplos desta classe as funções MAXVAL, MINVAL. SUM que retornam o maior, o menor e a soma dos elementos de um vetor. Além disto, é definida também a função DOT_PRODUCT para a obtenção do produto interno de dois vetores.
- manipulação de matrizes: as operações MATMUL e TRANSPOSE manipulam matrizes inteiras calculando o produto e a transposição.
- informação: SHAPE, SIZE, LBOUND, UBOUND são disponíveis para a obtenção de informação sobre um array.

Novas estruturas de controle

A primitiva de atribuição condicional WHERE é uma das novas estruturas de controle. Ela permite o controle da atribuição baseado em uma máscara, conforme ilustrado no exemplo seguinte:

```
REAL A(2,2), B(2,2), C(2,2)
DATA B/1,2,3,4/, C/1,1,5,5/
...
WHERE (B .EQ. C)
    A = 1.0
    C = B + 1.0
ELSEWHERE
    A = -1.0
ENDWHERE
...
```

Nos elementos onde a máscara é avaliada TRUE, A recebe 1.0 e C, B+1.0. Nos outros elementos (ELSEWHERE), A recebe -1.0. Ao término deste comando, as matrizes A e C têm os seguintes valores

$$A = \begin{bmatrix} 1.0 & -1.0 \\ -1.0 & 1.0 \end{bmatrix} \qquad C = \begin{bmatrix} 2.0 & 5.0 \\ 1.0 & 5.0 \end{bmatrix}$$

Fortran 90 também inclui a construção CASE similar ao comando equivalente da linguagem C para a 9seleção de expressões inteiras, lógicas e de caracteres.

```
SELECT CASE (RESPOSTA)
    CASE (‘S’)
      CALL EXECUTA
    CASE (‘N’)
      STOP ‘DONE’
    DEFAULT
      WRITE(*,*) ‘A resposta deve ser S ou N.’
ENDSELECT
```

5.1.2. Especificação de paralelismo em HPF

As operações paralelas são especificadas em HPF através:

- comando FORALL, uma generalização do comando de atribuição de elementos de array
- funções PURE, que podem ser chamadas dentro de um FORALL
- diretiva INDEPENDENT, que indica a possibilidade de execução paralela de um laço

Comando FORALL

O FORALL foi adotado na HPF a partir de um comando semelhante disponível no compilador da Connection Machine. Embora seja semelhante ao comando DO, seu significado é bem diferente. O trecho de código seguinte

```
FORALL ( I = 2 : 4 )
      A(I) = A(I-1) + A (I+1)
      C(I) = B(I) * A(I+1)
ENDFORALL
```

é executado seguindo os seguintes passos:

1. Avalia-se a expressão A(I-1) + A(I+1) para I = 2, 3 e 4.
2. Executa-se a atribuição aos respectivos elementos A(I).
3. Avalia-se B(I) * A(I+1) para os três valores de I.
4. Atribui-se os novos resultados aos elementos C(I).

Se os valores iniciais dos vetores forem

```
A=[0, 1 ,2, 3, 4], B=[0, 10, 20, 30, 40] e C=[-1, -1, -1, -1, -1]
```

ao término do trecho de código acima tem-se

```
A=[0, 2, 4, 6, 4], B=[0, 10, 20, 30, 40] e C=[-1, 40, 120, 120, -1]
```

Funções PURE

Uma das restrições existentes ao comando FORALL é a de que seu corpo pode conter apenas atribuições. Esta restrição é devido ao fato de que uma chamada a subrotina dentro do corpo do comando pode alterar o valor de uma variável global (efeito colateral), levando a um resultado errôneo no processamento. Caso haja uma função sem tais efeitos colaterais, basta ao programador especificá-la como uma função PURE para que esta possa ser ativada dentro de um FORALL.

Diretiva INDEPENDENT

A diretiva INDEPENDENT representa um outro enfoque para a especificação de paralelismo de dados. Ao invés de definir uma nova semântica a um novo comando, INDEPENDENT informa ao compilador que um laço DO não apresenta dependência entre as iterações, ou seja, que uma iteração não afeta o resultado de outra iteração. Por exemplo, no trecho de código seguinte

```
!HPF$ INDEPENDENT
DO J = 1, 3
      A(J) = A ( B(J) )
      C(J) = A(J) * B(A(J))
ENDDO
```

cada iteração pode ser executado em um processador diferente e nenhuma operação de sincronização é necessária.

5.1.3. Mapeamento de dados em HPF

O mapeamento de dados em HPF é realizado através da especificação da forma como os dados são distribuídos entre os processadores. Sua importância está no fato de que uma das partes essenciais dentro de um programa com paralelismo de dados se refere à localidade dos dados. Isto é, a localização dos dados apresenta uma grande influência no desempenho da aplicação, visto que o acesso a um dados remoto envolver a necessidade de comunicação entre processadores.

HPF usa uma estratégia dividida em duas fases para o mapeamento dos dados:

- a diretiva ALIGN efetua o alinhamento dos elementos de arrays distintos; desta forma, se dois elementos são alinhados juntos, eles sempre serão armazenados no mesmo processador.
- a diretiva DISTRIBUTE particiona os elementos de array pelos processadores; este mapeamento leva em conta os outros arrays com os quais foram alinhados.

Os estágios de mapeamento são especificados como as declarações, assim elas devem ser escritas de forma que o compilador possa processá-las corretamente.

Alinhamento de arrays

A diretiva ALIGN especifica como elementos individuais se relacionam com elementos de outros arrays. Por exemplo, a diretiva

```
!HPF$ ALIGN A(I) WITH B(I)
```

faz com que o 1° elemento de A seje ligado ao 1° elemento de B, e assim sucessivamente.

A escolha do melhor alinhamento é muito dependente do programa e pode variar dependendo do trecho do código. Desta forma, é muito difícil ao programador fazer esta escolha. Desta forma, muitos preferem ignorar este recurso da linguagem.

Exemplos mais complexos são

```
!HPF$ ALIGN A(I) WITH B(I+2)
```

que associa o i-ésimo elemento de A com o (i+2)-ésimo elemento de B.

```
!HPF$ ALIGN X(I,J) WITH Y(J,I)
```

onde X é alinhado com a transposta de Y.

```
!HPF$ ALIGN Y(K) WITH W(K,*)
```

onde cada elemento de Y é alinhado com uma linha inteira de W. A figura 5.2 abaixo ilustra um caso mais complexo de alinhamento.

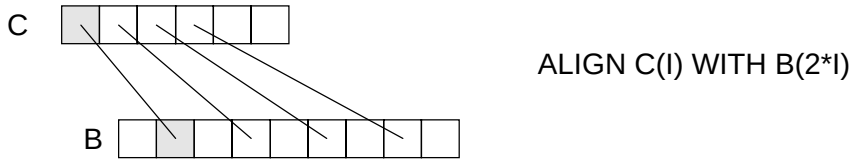


Figura 5.2 - Exemplo de alinhamento de arrays.

Distribuição de arrays

A distribuição dos elementos é especificada através da definição de um padrão regular de distribuição para cada uma das dimensões. A forma básica da diretiva DISTRIBUTE para uma matriz bi-dimensional é

```
!HPF$ DISTRIBUTE nome_da_matriz ( padrão_1, padrão_2)
```

O padrão pode ser especificado com sendo uma das opções seguintes:

*	sem distribuição
BLOCK(n)	distribuição blocada (default: n = N/P)
CYCLIC(n)	distribuição cíclica (default: n = 1)

Seja N o número de elementos em uma dimensão do array e P o número de processadores, a figura 5.3 abaixo ilustra alguns casos exemplo de distribuição. Os elementos marcados são aqueles mapeados ao processador 1. É considerada a distribuição de uma matriz 8x8 em 4 processadores.

5.1.4 - Características da HPF Versão 2.0

Embora a linguagem HPF já disponha de uma série de compiladores comerciais e já tenha uma certa aceitação na comunidade científica, ela apresenta uma série de limitações. A nova versão da linguagem HPF, ainda em fase de definição, procurará incorporar uma série de recursos, que incluem:

- suporte a paralelismo de tarefas (ou de controle)
- suporte a distribuição irregular de dados e estruturas de dados mais complexas
- suporte a E/S paralela

Espera-se a publicação de uma versão preliminar da HPF versão 2.0 durante a realização do Supercomputing'96 a ser realizada nos Estados Unidos. Os comentários preliminares informam que os objetivos principais desta nova versão é atender aos diversos requisitos que os usuários e as empresas vem reinvindicando de forma a tornar a HPF uma linguagem de grande aceitação.

5.2 - O Sistema CPAR

Buscando o equilíbrio entre alto desempenho, facilidades de programação e portabilidade, a linguagem CPAR foi projetada visando oferecer construções simples para a exploração do paralelismo em múltiplos níveis, e permitir a otimização do uso da localidade de memória.

Em computadores com arquiteturas que apresentam uma hierarquia de memória, tais como sistemas com multiprocessadores com memória compartilhada e memória local, ou sistemas com aglomerados de multiprocessadores com memória compartilhada local em cada aglomerado e uma memória compartilhada global, a exploração da localidade de memória é um aspecto importante na obtenção do alto desempenho.

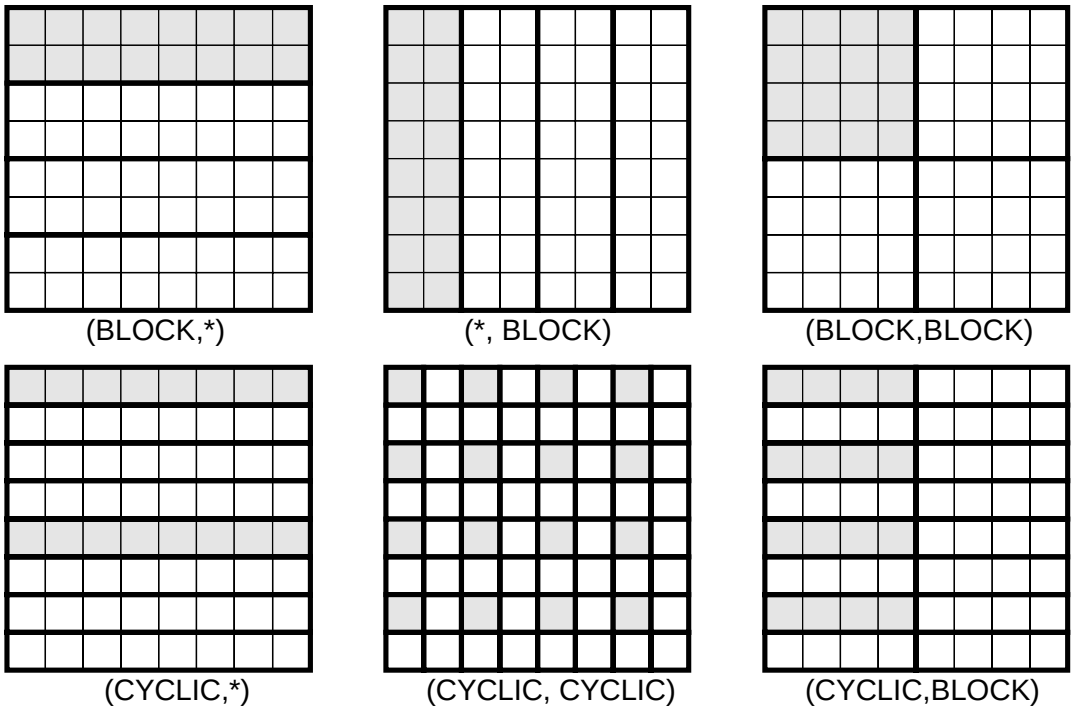


Figura 5.3 - Exemplos de distribuição de elementos de array em HPF.

A linguagem CPAR é uma extensão da linguagem C, na qual foram acrescentadas construções para expressar o paralelismo. Em seu projeto algumas características tiveram a sua origem baseada na linguagem Concurrent C [Geh86] e na linguagem ADA [Geh88], que oferecem um modelo de programação multitarefas com passagem de mensagem.

Nesta seção são apresentados o modelo de programação suportado pela linguagem CPAR, a descrição das construções oferecidas e algumas aplicações.

5.2.1. Modelo de programação

Paralelizar um programa é distribuir o seu trabalho entre os processadores disponíveis, através da sua partição em múltiplas tarefas que podem ser executadas simultaneamente. O modelo de programação suportado pela linguagem CPAR permite o uso de múltiplos níveis de paralelismo.

Blocos contendo elementos que devem ser executados seqüencialmente, podem ser executados simultaneamente. Cada elemento de um bloco pode ser um bloco, promovendo assim, múltiplos níveis de paralelismo. Esta é uma paralelização de granularidade grossa.

O modelo de programação adotado no sistema CPAR oferece a paralelização da função principal ("main") em múltiplos níveis através dos blocos paralelos. A figura 5.4 ilustra esta característica do modelo.

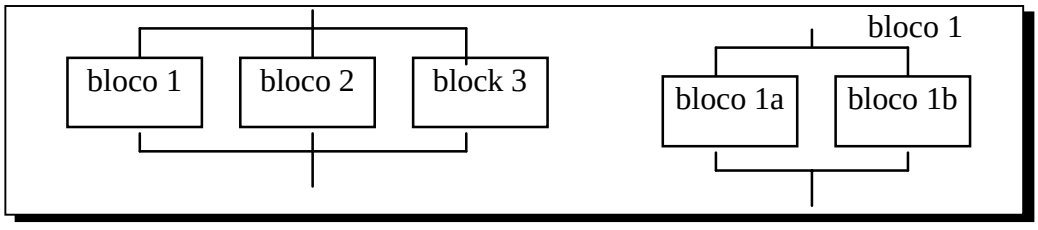


Figura 5.4 - Processo principal com múltiplos níveis de paralelismo.

No exemplo apresentado na figura 5.4, considerando-se uma quantidade de processadores suficiente para a execução simultânea de todos os blocos, e nenhum custo de paralelização tem-se:

```
t = maximo(t_bloco 1, t_bloco 2, t_bloco 3)
t_bloco 1 = maximo (t_bloco 1a, t_bloco 1b)
```

Para uma execução seqüencial, tem-se:

```
t= t_bloco 1a + t_bloco 1b  + t_bloco 2 + t_bloco 3
```

Em uma avaliação mais exata é necessário adicionar o custo referente à paralelização, incluindo os tempos gastos na partição em múltiplas tarefas, na criação dos processos e nas sincronizações necessárias. Considerando tais custos, tem-se uma influência relevante da granularidade de paralelismo no desempenho do programa. Os custos referentes ao acesso à memória compartilhada e às comunicações entre tarefas também devem ser considerados.

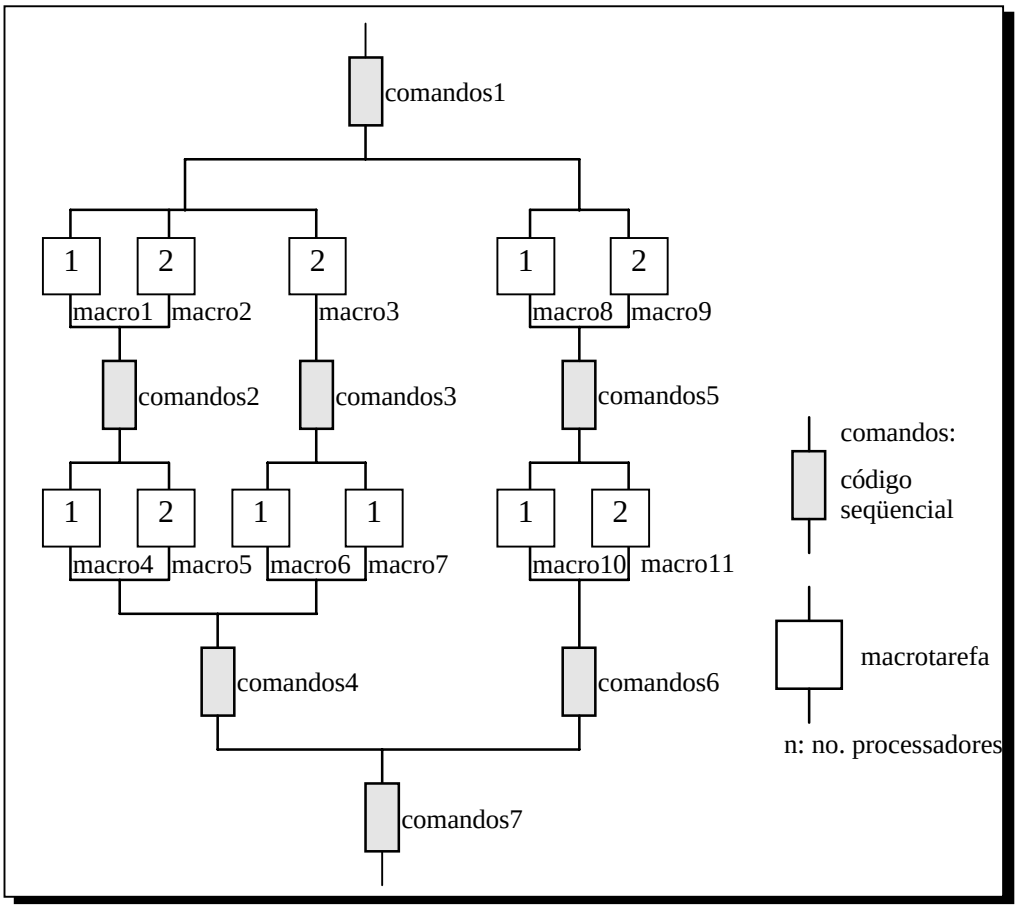
No sistema CPAR, os custos referentes à criação de processos e ao acesso à memória compartilhada, foram amenizados, pela adoção de uma combinação das técnicas aqui referenciadas como “microtasking” e “macrotasking”. A técnica “macrotasking”ou “multitasking”, consiste em particionar o programa em múltiplas macrotarefas, as quais devem envolver uma computação significativa, caracterizando uma granularidade grossa, tal que o custo adicional não inviabilize a obtenção de um bom desempenho. Em cada macrotarefa podem estar presentes múltiplas microtarefas, que exploram o paralelismo em uma granularidade média ou fina, no nível de laços ou blocos de instruções paralelos.

Neste modelo as variáveis referenciadas apenas pelas microtarefas de uma macrotarefa devem ocupar a área de memória compartilhada local ao aglomerado ao qual pertencem os processadores responsáveis pela execução da macrotarefa. Somente as variáveis compartilhadas entre várias macrotarefas devem ocupar a área de memória compartilhada global. Esta hierarquia na memória compartilhada global, promove uma redução no custo referente ao acesso à memória compartilhada, visto que o custo gasto em um acesso à memória compartilhada local é menor do que à memória compartilhada global. Tem-se que locações visíveis para um maior número de processadores têm um acesso mais caro do que aquelas visíveis para um menor número [Har91].

A figura 5.5 ilustra o modelo de programação do sistema CPAR, apresentando os níveis de paralelismo. Neste exemplo, considerando-se uma quantidade de processadores suficiente para a execução simultânea dos blocos e das macrotarefas, e nenhum custo adicional devido à paralelização, tem-se:

```
t= t_comandos1+
       máximo(
           (máximo(
                ( máximo( t_macro1, t_macro2) +
                        t_comandos2+
                        máximo( t_macro4,t_macro5)),
                ( t_macro3 +
                  t_comandos3 +
                  máximo(t_macro6 , t_macro7))) +
             t_comandos4),
            (máximo(t_macro8,t_macro9)+
             t_comandos5+
             máximo(t_macro10,t_macro11) +
             t_comandos6))+
           t_comandos7
```

Na primeira versão do sistema CPAR , utilizou-se a estratégia “microtasking” usada em alguns sistemas propostos e utilizados por volta de 1987 , e baseou-se em um projeto desenvolvido em 89 no Laboratório de Sistemas Integráveis cuja etapa inicial foi realizada por Roberto Diniz Branco, e que consistia em uma biblioteca de rotinas para implementação da técnica “microtasking” em um programa escrito na linguagem C sobre um computador com memória compartilhada. Nesta versão da CPAR, a criação de uma macrotarefa envolve a criação dos processos responsáveis pela execução das suas microtarefas, minimizando o custo envolvido.



As estratégias adotadas exploram a hierarquia de memória fornecida pelo modelo de programação, minimizam o custo referente à criação de processos e proporcionam uma melhor utilização dos processos criados.

Resumindo-se tem-se no modelo de programação os seguintes elementos:

- *elemento seqüencial*: é uma porção de comandos, que deve ser executada seqüencialmente;
- macrotarefa: é uma porção de código, ao nível de subrotina, que pode conter um nível mais fino de paralelismo, as microtarefas. Macrotarefas podem ser executadas simultaneamente;
- *microtarefa*: é uma porção de código seqüencial, contida em um laço, cujas iterações são executadas paralelamente, ou em um bloco de comandos que é executado paralelamente a outros blocos;
- *blocos paralelos*: são porções do código da função principal (“main”), do programa que são executadas paralelamente. Um bloco pode conter blocos paralelos, ou seja é permitido o aninhamento de blocos. Um bloco pode conter elementos seqüenciais, macrotarefas e blocos paralelos;
- *função principal*: é a função principal do programa (“main”). Pode conter blocos, elementos seqüenciais e macrotarefas.

5.2.2. Descrição da Linguagem

A linguagem CPAR [Sat92] apresenta as seguintes características:

- oferece primitivas para explicitar blocos paralelos e macroblocos que encapsulam blocos paralelos;
- permite declarar e colocar em execução as macrotarefas;
- oferece primitivas para explicitar microtarefas;

- permite declarar variáveis locais, variáveis compartilhadas globais e locais;
- a comunicação entre macrotarefas pode ser efetuada por memória compartilhada ou passagem de mensagem;
- microtarefas podem efetuar acessos a variáveis compartilhadas locais à mcrotarefa ou globais às macrotarefas;
- a sincronização entre macrotarefas, através do uso de semáforo ou evento, é efetuada por rotinas de biblioteca;
- fornece um mecanismo de exclusão mútua, denominado monitor, que permite a utilização segura da memória compartilhada. Este mecanismo está presente na linguagem Pascal concorrente.

5.2.3. Macrotarefas

As macrotarefas são porções do programa que podem ser executadas assincronamente (paralelismo a nível de subrotina)

Declaração e criação de macrotarefas

Na criação de uma macrotarefa são alocados os recursos necessários e providenciada a sua execução. Em seu término todos os recursos são liberados.

- declaração da tarefa

```
task spec nome_task();

task  body nome_task()
declaração de parametros
{declararação de variáveis
  ...
}
```

- criação e execução da tarefa:

```
create n,nome_task(parametros)
```

onde n: total de processadores para da tarefa.

Exemplo :

```
task spec teste();
task body teste()
{
char a;
a="A";
printf( a );
}
void main()
{
create 1, teste();
printf( "fim teste" );
}
```

Sincronização das Macrotarefas

Comandos específicos são oferecidos para a sincronização das macrotarefas no programa principal .

`wait_task  (nome_task)`

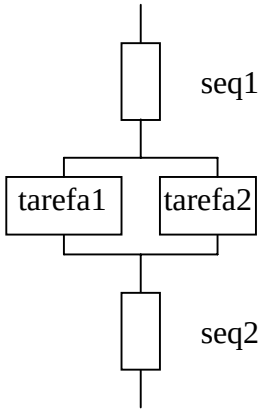
- espera pelo término da macrotarefa especificada

`wait_all`

- espera pelo término de todas as macrotarefas ativas

Exemplo:

```
task spec tarefa1();
task spec tarefa2();
void mostra( carac, lim )
char carac;
int lim;
{
  int i;
  for( i=1; i<lim; i++ )
    printf( carac );
}
task body tarefa1()
{
  mostra( 'a', 100 );
}
task body tarefa2()
{
  mostra( 'b', 100 );
}
void main()
{
  alloc_proc( 2 );
  printf( "inicio" ); /* secao 1 */
  create 1, tarefa1();
  create 1, tarefa2();
  waitall;
  printf( "fim" ); /* secao 2 */
}
```



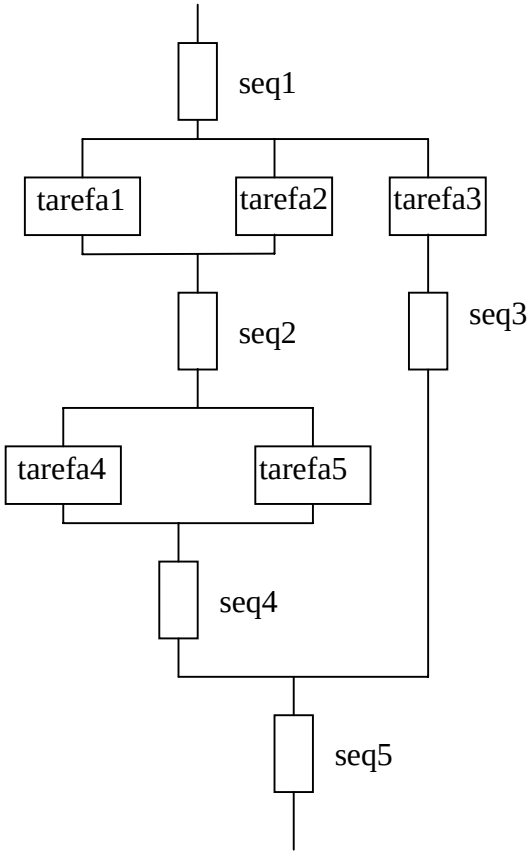
5.2.4. Blocos Paralelos no Programa Principal

A criação de blocos paralelos pode ser especificada no corpo da rotina main através da construção seguinte.

```
cobegin
  ...
also
  ...
also
  ...
coend
```

Exemplo:

```
...
void main()
{
  printf( "inicio" );
  create 1, tarefa1();
  create 1, tarefa2();
  create 1, tarefa3();
  cobegin
    wait_task ( tarefa1);
    wait_task ( tarefa2);
    printf( "secao 2" );'
    create 1, tarefa4();
    create 1, tarefa5();
    wait_task ( tarefa4);
    wait_task  (tarefa5);
    printf( "secao 4" );
  also
```



```
    wait _task (tarefa3);
    printf( "secao 3" );
  coend
  printf( "secao 5 " );
}
```

5.3 - Parallel Virtual Machine (PVM)

A biblioteca PVM - *Parallel Virtual Machine* [Gue94] foi projetada para sistemas heterogêneos, compostos de estações de trabalho e PC's, sendo a alta portabilidade uma de suas características mais atraentes. A maior parte do projeto tem sido realizada por iniciativa do *Oak Ridge National Laboratory*, e da Universidade do Tenessee.

O ambiente PVM é constituído de uma biblioteca de rotinas com interface para C e FORTRAN, cujo principal serviço é a troca de mensagens entre processos. O usuário pode escolher o protocolo de rede a ser utilizado, TCP ou UDP, no envio e recebimento de mensagens. Em ambos os casos, as mensagens devem ser empacotadas antes do envio, e desempacotadas pelo processo receptor após seu recebimento. O empacotamento gera um *overhead* extra, mas esse é o preço pago pela comodidade que o ambiente oferece em resolver o problema dos diferentes formatos de armazenamento de dados em arquiteturas heterogêneas. Para o programador, toda a manipulação dos *sockets* é invisível, sendo feita automaticamente pelo processo *daemon* do PVM. A biblioteca PVM oferece ainda, primitivas para criação e manipulação de processos dinamicamente, e relativa tolerância a falhas.

5.3.1. Configuração do ambiente

A principal idéia por trás do PVM é utilizar um conjunto de computadores heterogêneos interconectados, como um recurso virtualmente único. Cada computador existente na rede pode ser utilizado como sendo um nó da máquina paralela virtual. O papel de console da máquina paralela é assumido pelo próprio nó local onde o usuário está localizado fisicamente. O usuário pode alocar qualquer nó da rede local, ou a longa distância, desde o mesmo esteja devidamente autorizado. A partir do console, o usuário pode criar sua própria configuração. Uma configuração é composta de um conjunto de nós, e pode ser determinado de duas maneiras :

- Após a criação da máquina paralela virtual antes de disparar uma aplicação paralela. Isto é feito ao nível da linha de comando do console, através do comando ADD, seguido do nome da máquina.
- Durante o tempo de execução da aplicação paralela, pela função pvm_addhosts ( ), como no exemplo abaixo :

```
int num_hosts = 4;
static char *h_name[]={ “no1", "no2”, "no3", "no4" }; /* especifica o
nome                  das máquinas da rede que serão alocadas */
int h_status[]={0,0,0,0};
int info;
. . .
if ((info = pvm_addhosts(h_name, num_hosts, h_status)) < 0) {
     printf("ERRO : Impossivel acrescentar hosts\n");
     pvm_exit ( );
}
. . .
```

Os argumentos passados à rotina pvm_addhosts são, na ordem : um vetor contendo os nomes dos nós a serem alocados, a quantidade de nós, e, por último, um vetor de inteiros onde a função irá retornar os códigos de *status* de cada nó alocado, indicando sucesso ou insucesso. Somente uma única máquina paralela virtual pode ser criada para cada usuário em um dado instante, e cada uma delas não tem qualquer relação com a máquina de outro usuário. Isto evita a possibilidade de uma aplicação causar problemas nas aplicações de outros usuários.

5.3.2. Formato de um programa (interface c/ ling. C)

Apesar da biblioteca PVM possuir interfaces para linguagem C e FORTRAN, apenas a primeira será mostrada neste texto.

```
#include  ”pvm3.h”
...
main (  )
{
     int info;
     info = pvm_mytid ( );
     ...
     pvm_exit (  );
}
```

No exemplo acima, a função pvm_mytid ( ) devolve o identificador de tarefa (*task id*), ao mesmo tempo que cria um vinculo entre o processo do usuário e um processo especial (*daemon*) que gerencia o ambiente PVM. Essa função deve ser chamada no início de todo programa PVM, antes de qualquer outra chamada ou troca de mensagem. Do mesmo modo, a função pvm_exit ( ) deve ser chamada no final de todo programa PVM, para encerrar a conexão deste com o *daemon* do PVM.

5.3.3. Criação e controle de processos.

Uma vez criada uma configuração, uma aplicação paralela pode criar diversos processos nos vários nós. No ambiente PVM esses processos que compõem uma aplicação paralela são chamadas de **tarefas** (*tasks*). Existem três níveis de abstração para criação das tarefas.

No primeiro modo, as tarefas são criadas em um nó qualquer do aglomerado, a critério do PVM. Desse modo o usuário não precisa se preocupar com a localização física de seus processos, e novas políticas de escolha do “nó mais adequado” para receber uma nova tarefa podem ser implementadas, sem que o usuário precise alterar as aplicações já existentes.

```
. . .
pvm_spawn (”prog1”, NULL, PvmTaskDefault, 0, n_tasks, tids );
 /* executa n_tasks  cópias do arquivo “prog1” */
. . .
```

Um segundo modo permite que o usuário crie suas tarefas apenas em alguns tipos específicos de computadores (fabricante, família, etc).

```
. . .
pvm_spawn (”prog3”, NULL, PvmTaskArch, “SUN4”, 5, tids );
  /* executa 5 cópias do arquivo ”prog3” nas máquinas do tipo SUN4 */
. . .
```

No terceiro modo, considerado de mais baixo nível, o programador especifica “onde” as tarefas serão criadas. Neste caso, o usuário tem pleno controle sobre a topologia assumida pela sua aplicação. Também é possível destruir tarefas criadas e verificar seu estado.

```
. . .
pvm_spawn (”prog2”, NULL, PvmTaskHost, “dumbo”, 1,tids );
/* executa ”prog2” na máquina denominada “dumbo”*/
. . .
```

É possível matar processos filhos, passando seus números (*task id*) como argumento da função pvm_kill ( ).

```
. . .
pvm_kill ( tid ); /* termina a tarefa cujo task id é tid */
. . .
```

O exemplo a seguir ilustra o procedimento completo para criar uma tarefa.

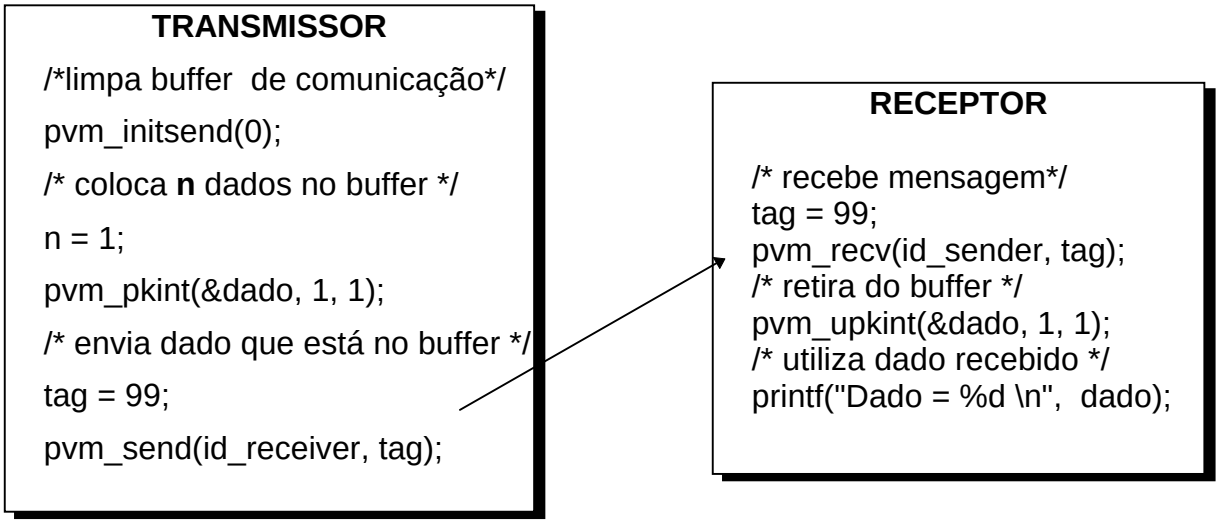
```
#include <stdio.h>
#include </user/pvm/pvm3/include/pvm3.h>

main (  )
{    int tid, kid,  dado, cc;
```

```
      char buf[100];

      printf("Processo pai tid = %x\n", pvm_mytid());
      cc = pvm_spawn("filho", (char**)0, 0, "", 1, &tid);
      if (cc == 1) {
           printf("OK : Criou processo filho \n");
           pvm_exit();
           exit(0);
      } else
           printf("Nao foi possível criar processo  Erro %d\n",cc);
      pvm_exit();
      exit(0);
}
```

A função pvm_spawn retorna a quantidade de processos filhos criados. Note que nenhum nó foi alocado explicitamente no decorrer do programa, através da função pvm_addhosts, de modo que o processo filho será criado no nó local, ou possíveis nó alocados antes do início do programa, ao nível da console da máquina virtual paralela.

5.3.4. Comunicação entre processos

No ambiente PVM, existem diversas funções de comunicação entre tarefas. A mais simples é realizada através da função pvm_send( ), e a recepção da mensagem é feita pela função pvm_recv().



A figura acima ilustra o procedimento de ambos os processos, transmissor e receptor, para uma troca de mensagem. No exemplo abaixo, o programa ”pai.c” cria o processo “filho.c”, e ambos trocam dados entre si.

Pai.C

```
#include <stdio.h>
#include </user/pvm/pvm3/include/pvm3.h>
main ()
{
      int mytid, tid, dado, dado_rec, info;

      mytid = pvm_mytid ();
      info = pvm_spawn("filho", (char**)0, 0, "", 1, &tid);
      if (info == 1) {
           dado = rand();       * produz um dado aleatório */

           pvm_initsend(0);     /* envia dado para o filho */
```

```
            pvm_pkint(&dado, 1, 1);
            pvm_send(tid, 63);

            pvm_recv(tid, 63);   /* recebe dado enviado pelo filho */
            pvm_upkint(&dado_rec, 1, 1);
            printf("Dado local = %d Soma = %d\n",dado, dado + dado_rec);
      } else
            printf("Nao conseguiu criar processo filho\n");
      pvm_exit();
}
```

FILHO.C

```
#include <stdio.h>
#include </user/pvm/pvm3/include/pvm3.h>
main ()
{
   int pid,  dado, dado_rec;
   pvm_mytid();
   pid = pvm_parent();

   pvm_recv(pid, 63);
   pvm_upkint(&dado_rec, 1, 1);

   dado = rand();

   pvm_initsend(0);
   pvm_pkint(&dado, 1, 1);
   pvm_send(pid, 63);
   pvm_exit();
}
```

5.3.5. PVM para microcomputadores

A bilioteca PVM pode ser utilizada em uma rede de microcomputadores do tipo PC com o sistema operacional LINUX.

Mais recentemente, foi portada uma versão para plataforma Win32 da Microsoft. Um dos objetivos é permitir o uso combinado de computadores com Win NT e UNIX, com aplicações PVM. Inicialmente, a bilioteca foi portada para Win NT, devendo ocorrer o mesmo em seguida, para Windows 95.

Capítulo 6 - ALGORITMOS E IMPLEMENTAÇÃO PARA SISTEMAS COM MEMÓRIA COMPARTILHADA

Neste capítulo faz-se a apresentação de dois algoritmos clássicos em um ambiente de computação paralela com o paradigma de programação com memória compartilhada. Primeiramente descreve-se cada um dos casos estudados, seus algoritmos paralelos e finalmente, a implementação paralela e medidas de desempenho.

6.1 - Ambiente de Execução e Bibloteca de Paralelismo

A execução dos programas foi realizada em uma maquina Silicon Graphics Power Series 4D/480 VGX, que é composto por 8 processadores MIPS R3000A. Cada processador possui uma memória cache de instruções de 64Kbytes, uma cache de dados de primeiro nível de 64Kbytes e uma cache de dados secundário de 1Mbytes. A memória principal do sistema é de 256 Mbytes. A medição de tempo foi efetuada instrumentando-se os programas com a chamada times. A versão do sistema operacional é o IRIX 5.3.

6.2 - Multiplicação de Matrizes

O caso da multiplicação de matrizes clássica consta do produto de duas matrizes quadradas de ordem 512, implementado segundo o algoritmo tradicional baseado no cálculo do produto interno das linhas da primeira matriz com as colunas da segunda matriz, segundo

$$c_{i,j} = \sum_{k=1}^{512} a_{i,k} \times b_{k,j}$$

Neste algoritmo, é calculado o produto interno para cada par composto por um linha da matriz A e uma coluna da matriz B. Esta estratégia pode ser esquematizada segundo ilustrado na figura 6.1.

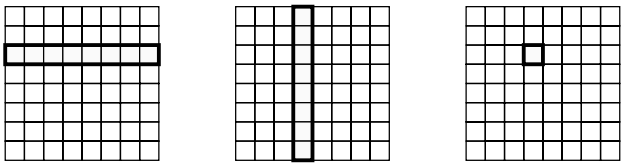


Figura 6.1 - Algoritmo tradicional para multiplicação de matrizes.

Este algoritmo de multiplicação é normalmente implementado sob a forma de três laços aninhados para o cálculo dos produtos internos.

```
int i,j,k;
double a[N][N], b[N][N], c[N][N];
...
for (I=0; i<N; I++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

A estratégia de paralelização que pode ser adotada é paralelizar o loop mais externo, distribuindo as iterações pelos processadores de forma estática (*pre-scheduling*). Desta forma, a matriz B tem seus elementos acessados por todos os processadores e as matrizes A e C podem ter seus elementos distribuídos pelos processadores segundo as linhas. A figura 6.2 abaixo ilustra o esquema adotado.

O código CPAR para esta paralelização é o seguinte:

```
int i, j, k;
shared double a[N][N], b[N][N], c[N][N];
...
forall I=0 to N
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Ve-se então que esta estratégia de paralelização enfoca a distribuição dos cálculos dos produtos internos pelos processadores. Ou seja, procurou-se particionar a carga computacional de forma igualitária.

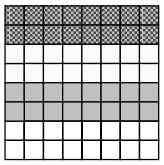


Figura 6.2 - Divisão dos acessos aos elementos das matrizes A e C
do programa de multiplicação de matrizes.

Esta estratégia de paralelização não necessita de nenhum mecanismo de exclusão mútua para a atualização da matriz produto, visto que seus elementos foram distribuídos pelos processadores, que tinham acesso exclusivo às suas respectivas linhas.

6.2.1 - Análise da paralelização da multiplicação de matrizes

O estudo da paralelização visou analisar o efeito da paralelização com relação ao tempo de execução total em função do número de processadores. Os tempos de execução obtidos para cada uma destas versões é mostrada na tabela 6.1 e resumidos no gráfico 6.1.

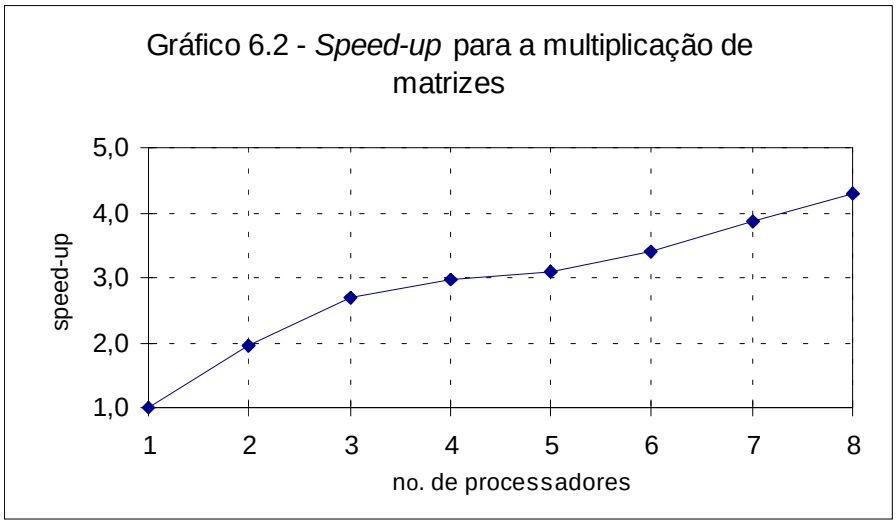
Tabela 6.1 - Tempos de execução das versões do programa de multiplicação de matrizes.

Nº de processadores	tempo de execução (s)
1	480.0
2	245.0
3	178.0
4	161.9
5	154.5
6	140.6
7	124.5
8	111.9

Constata-se uma diminuição do tempo de execução à medida que aumentamos o número de processadores. Nota-se que o tempo diminui a uma taxa maior entre 1 e 3 processadores. Este fenômeno pode ser examinado de forma mais clara em outro gráfico apresentado mais adiante.

Outro fato que pode ser concluído é a adequação da paralelização à plataforma testada. Embora o número de processadores testado seja relativamente pequeno, constata-se ainda um ganho de desempenho, que pode não ser válido em um outro ambiente com um número elevado de processadores (por exemplo, em um máquina com 32 processadores).

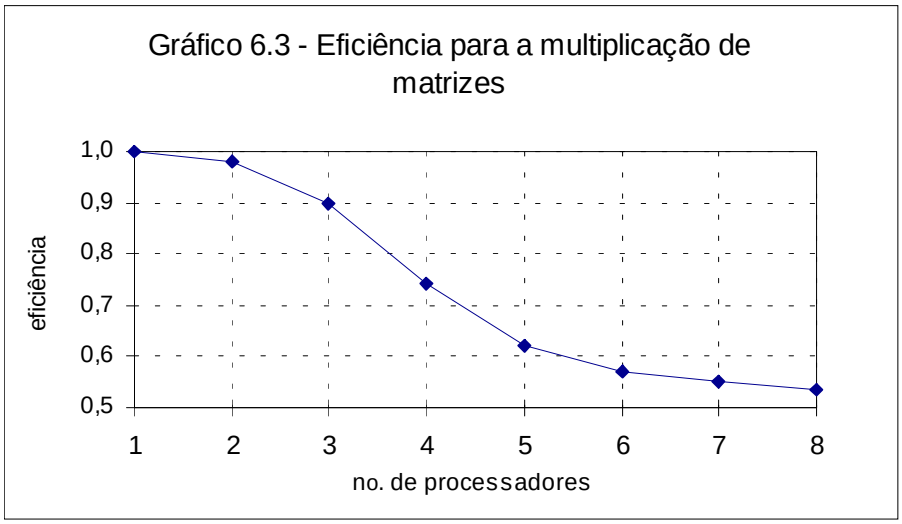
O gráfico 6.2, onde está apresentado a curva de speed-up para cada uma das versões, descreve melhor o desempenho do programa paralelizado. No início do gráfico ve-se claramente que a paralelização está muito próxima do ideal, com um ganho de velocidade próximo ao número de processadores empregados. A partir daí, à medida que se aumenta o número de processadores, a curva se afasta da curva ideal. Ao final, o programa executado em 8 processadores é apenas 4,23 vezes mais rápido em relação à execução com 1 processador.



O mesmo efeito pode ser observado através da curva da eficiência, apresentada no gráfico 6.3. Nota-se uma queda inicial a partir do ponto relativo a 2 processadores. Esta queda passa a diminuir a partir do ponto referente a 5 processadores, até atingir um valor de 53% para 8 processadores.

Conclui-se então que, apesar da estratégia de paralelização não ser a ideal, os ganhos de desempenho obtidos com o aumento de processadores é vantajoso, visto que se obteve um tempo de execução melhor.

Outras análises podem ser realizadas neste cont7exto. Por exemplo, em [Mid95], é feita uma comparação, similar à realizada neste capítulo, entre versões diferentes de programas de multiplicação de matrizes paralela. Através da aplicação de diversas transformações de programa, obteve-se ao final do processo de otimização uma versão que apresenta uma característica de escalabilidade muito superior à versão inicial. Nesta versão otimizada, para a execução em 8 processadores, o speed-up e a eficiência foram, respectivamente, 10,8 e 60%. Este speed-up super-linear pode ser explicado pelo fato das transformações de programa efetuarem adicionalmente otimizações de acesso à memória.



6.3 - Decomposição LU

O programa de decomposição LU executa a computação de duas matrizes triangulares inferior e superior, L e U respectivamente, a partir de uma matriz quadrada A de ordem 512, tal que A = L × U. A implementação segue o algoritmo tradicional sem pivotamento (fig.6.3).

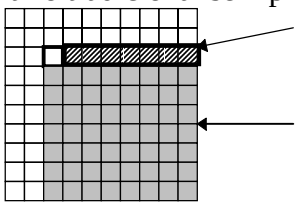


Figura 6.3 - Algoritmo tradicional para decomposição LU sem pivotamento.

A estratégia de paralelização adotada seguiu a seguinte estratégia: para cada passo do algoritmo quando uma nova linha é processada, a atualização da respectiva sub-matriz foi executada pelos diversos processadores da máquina. E a divisão de processamento para a computação desta submatriz é a mesma empregada para a multiplicação de matrizes (particionamento das linhas). A figura 6.4 ilustra o esquema adotado.

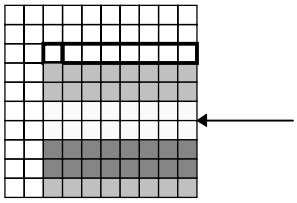


Figura 6.4 - Esquema de paralelização adotado para o programa de decomposição LU.

6.3.1. Análise da paralelização da decomposição LU

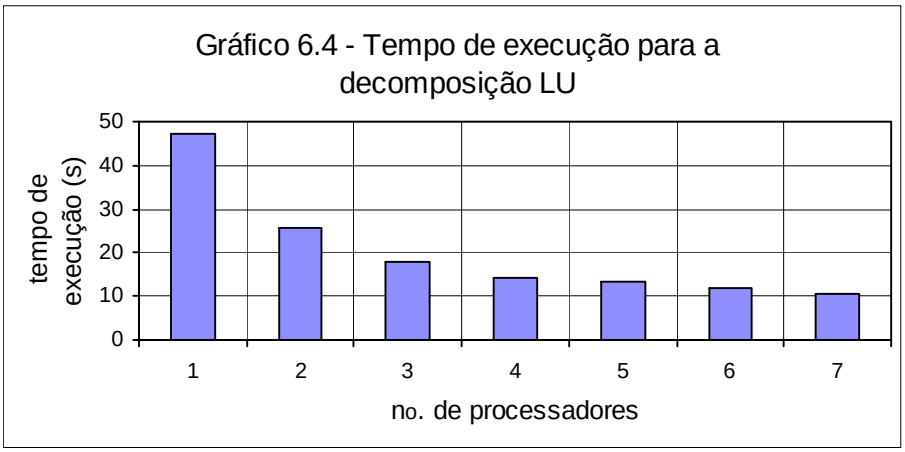
O procedimento experimental foi executado em 7 processadores. Não foi possível utilizar todos os processadores da máquina devido a alta carga de utilização no período de avaliação do caso de estudo. O programa de decomposição LU foi executada com vários processadores e os tempos de execução medidos são mostrados na tabela 6.2 abaixo.

Tabela 6.2 - Tempos de execução das versões do programa de decomposição LU (em seg).

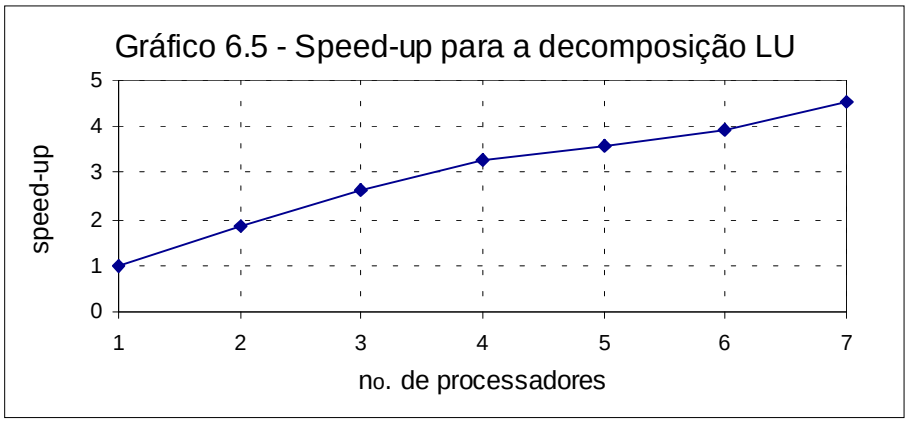
Nº de	tempo de

processadores	execução (s)
1	47.20
2	25.50
3	17.90
4	14.35
5	13.20
6	12.00
7	10.40

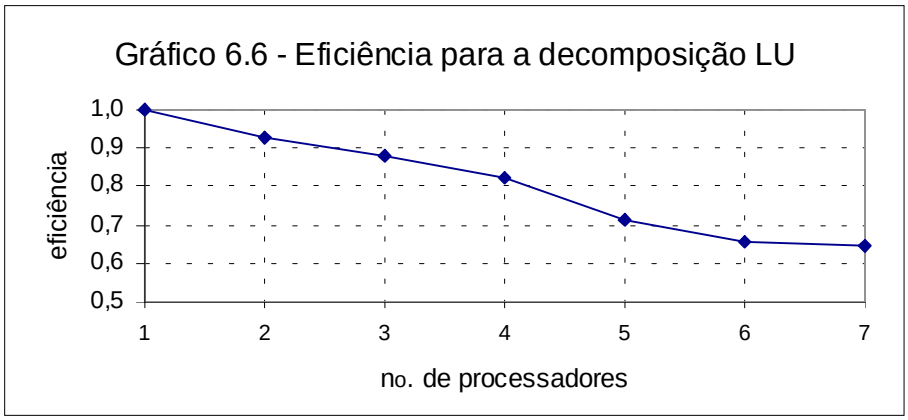
Nota-se um ganho de desempenho em todos os casos estudados. A execução em 7 processadores apresentou um ganho de 78% em relação à execução monoprocessada. Este ganho é bem significativo, e pode ser melhor analisado observando-se a curva de speed-up.



Analisando-se também o efeito das transformações sobre o *speed-up*, ilustrado no gráfico 6.5, verifica-se um ganho de desempenho quase linear com o aumento do número de processadores. O speed-up com 7 processadores é de 4,53.



O mesmo efeito é observado com relação à eficiência. O gráfico 6.6 mostra que de uma maneira geral. o comportamento do programa de composição LU foi similar ao de multiplicação de matrizes..



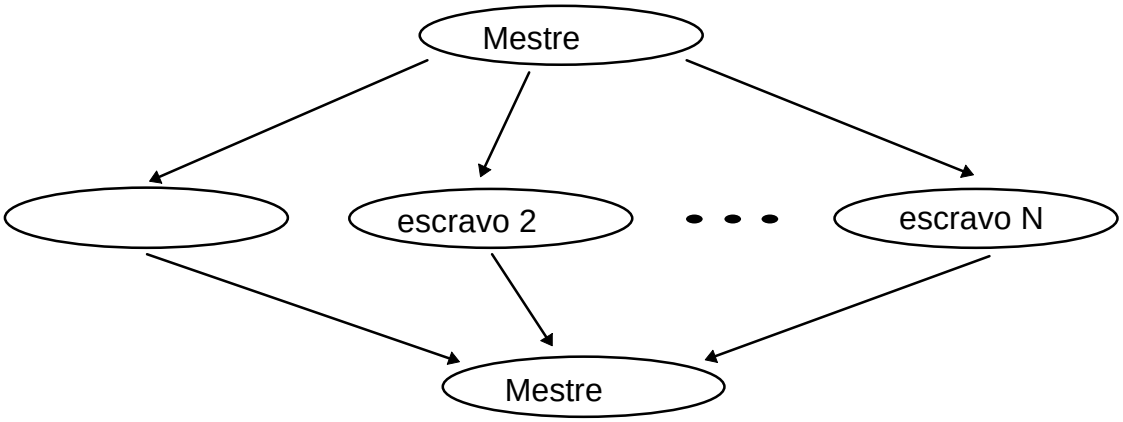
Resultados similares podem ser obtidos para outros programas.

Capítulo 7 - ALGORITMOS E IMPLEMENTAÇÕES PARA SISTEMAS COM MEMÓRIA DISTRIBUÍDA

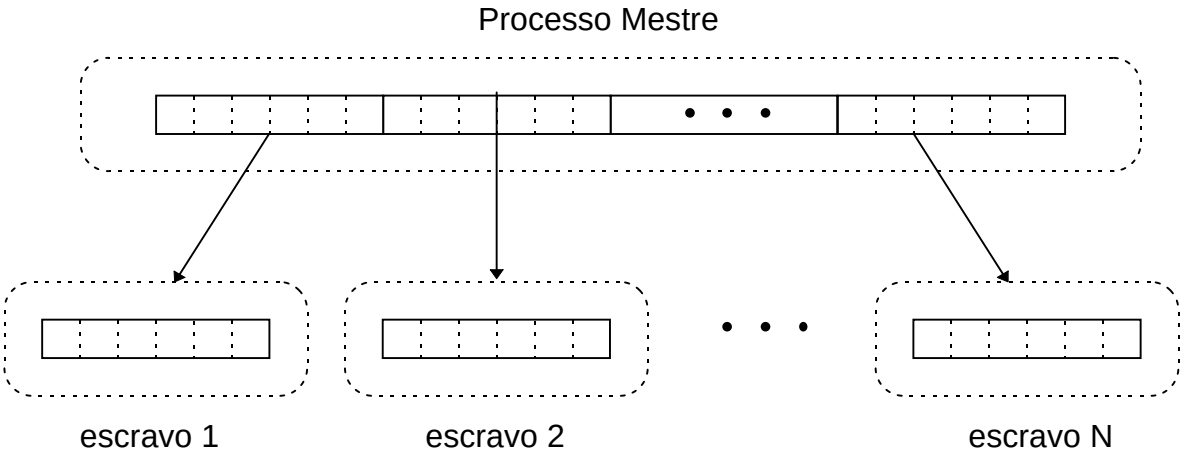
Nos computadores atuais, o tempo gasto para realizar uma troca de mensagem pode variar de duas a seis ordens de grandeza maior do que o tempo necessário para que o processador execute uma instrução. Um dos principais fatores, senão o principal, de desempenho de um algoritmo baseado em troca de mensagens é a quantidade de mensagens geradas durante sua execução. Além do número de mensagens, a distância (quantidade de nós intermediários) entre o que transmite e o que recebe a mensagem é importante. Assim, existem algoritmos bastante específicos, projetados para trabalhar em computadores com topologias como hipercubos, mesh, e outros. Não cabe neste texto, uma discussão mais profunda sobre os custos de cada algoritmo com comunicação, mas até por uma questão didática serão abordados algoritmos bastante simples.

7.1 - Modelo de computação mestre-escravo

Um modelo bastante simples de computação é o chamado **mestre-escravo**[4], no qual, um processo denominado *mestre* é responsável pela criação, monitoração e controle das tarefas *escravas*, distribuição dos dados e coleta dos resultados.



Para calcular a soma dos elementos de um vetor, por exemplo, pode-se dividi-lo em intervalos que serão distribuídos pelos nós de processamento. Cada nó irá calcular seu resultado parcial localmente, e enviá-lo ao mestre, para que este possa compor o resultado global.



[4] *master-slave*

O processo mestre divide os dados do vetor em partes iguais, e as envia aos escravos. Cada um dos escravos calcula a soma local de seus dados locais, e envia esse resultado para o mestre. Após coletar os resultados parciais dos escravos, o mestre calcula o resultado global.

```
#include <stdio.h>
#include </user/pvm/pvm3/include/pvm3.h>
#define TAM_VET 100             /* tamanho do vetor */
#define N_TASKS                 /* numero de processos escravos */
main ()
{
  int i, j, mytid, tid[N_TASKS], info, v[TAM_VET];
  long soma_parcial, soma_global;

  mytid = pvm_mytid ();
  /* cria escravos */
  info = pvm_spawn("escravo",(char**)0,0, "",N_TASKS, tid);
  if (info == N_TASKS) {        /* gera os dados do vetor */
    for ( i=0; i<TAM_VET; i++) {
      v[i] = i+1;
      printf("v[%d] = %5d\n ",i,v[i]);
    }
    printf("\n");
    j = 0;
    for ( i=0; i<N_TASKS; i++) { /* distribui os trechos do vetor*/
      pvm_initsend(0);
      pvm_pkint(&(v[i*(TAM_VET/N_TASKS)]),(TAM_VET/N_TASKS),1);
      pvm_send(tid[i], 63);
    }
    soma_global = 0; /* recebe resultados parciais dos escravos */
    for ( i = 0; i < N_TASKS; i++) {
      pvm_recv(tid[i], 63);
      pvm_upklong(&soma_parcial, 1, 1);
      printf(”Soma parcial recebida do processo [%d] = \
              %d\n”,tid[i],soma_parcial);
      soma_global += soma_parcial;  /* calcula a soma global */
    }
    printf("Soma global  = %d\n",soma_global);
    pvm_exit();
    exit(0);
  }
  else
    printf("Nao conseguiu criar processo filho\n");
  pvm_exit();
  exit(0);
}
```

O processo escravo recebe apenas os dados do intervalo que o mestre lhe enviou, soma e devolve o resultado parcial. Nesse exemplo, o processo pai direcionou a saída de erro dos filhos, para a sua sadia padrão. Isto permite que os escravos possam exibir dados em tela. Tal procedimento não é usual, e nem mesmo necessário, exceto às vezes, para fazer depuração.

```
#include <stdio.h>
#include </user/pvm/pvm3/include/pvm3.h>
#define    TAM_VET   100
#define   N_TASKS    5

main ()
{
  int i, pid, v_local[TAM_VET / N_TASKS];       /* vetor local */
  long soma_local;

  pvm_mytid();
  pid = pvm_parent();     /* coleta o identificador de seu pai */
```

```
  pvm_recv(pid, 63);      /* recebe os dados que o pai enviou */
  pvm_upkint(v_local,TAM_VET / N_TASKS, 1);

  soma_local = 0;              /* soma os elementos */
  for ( i  = 0; i < TAM_VET/N_TASKS; i++)
    soma_local += v_local[i];

  pvm_initsend(0);
  pvm_pklong(&soma_local, 1, 1);
  pvm_send(pid, 63);

  pvm_exit();
  exit(0);
}
```

Abaixo, segue uma amostra da saída que o programa produz na tela. A título de ilustração, cada escravo mostra sua soma local. O número que aparece entre colchetes é o *process id* de cada escravo.

```
Soma parcial recebida do processo [400B0] = 210
Soma parcial recebida do processo [400AF] = 610
Soma parcial recebida do processo [400AE] = 1010
Soma parcial recebida do processo [400AD] = 1410
Soma parcial recebida do processo [400AC] = 1810
Soma global  = 5050
```

Para realizar o produto de um vetor por uma matriz, pode-se, por exemplo replicar o vetor em todos os nós de processamento, enquanto que a matriz é distribuída linha a linha entre os nós de processamento.

mv_mestre.c

```
#include <stdio.h>
#include </user/pvm/pvm3/include/pvm3.h>
#define DIM     3
#define N_TASKS DIM
main ()
{
  int i,k,j, mytid, tid[N_TASKS], info, u[DIM], mat[DIM][DIM];
  long v[DIM];

  mytid = pvm_mytid ();
  info = pvm_spawn("mv_escravo", (char**)0, 0, "", N_TASKS, tid);
  if (info == N_TASKS) {/* se conseguiu criar todos as tarefas ... */
    /* Gera dados da matriz */
    k = 0;
    for ( i=0; i< DIM; i++)
      for ( j=0; j<DIM; j++)
        mat[i][j] = ++k;
    /* Gera os dados do vetor */
    k = 0;
    for ( i=0; i<DIM; i++)
      u[i] = ++k;
    printf("\n");
    /* Envia uma linha da matriz para cada processo */
    for ( i=0; i<N_TASKS; i++) {
      pvm_initsend(0);
      pvm_pkint(&(mat[i][0]),DIM, 1);
      pvm_send(tid[i], 63);
    }
```

```
    /* envia uma copia do vetor a cada processo filho */
    pvm_mcast ( tid, N_TASKS, 63 );

    /* Recebe um elemento do vetor U de cada processo */
    for ( i = 0; i < N_TASKS; i++) {
      pvm_recv(tid[i], 63);
      pvm_upklong(&(v[i]), 1, 1);
      printf ("v[%d] recebido do processo [%X] = %d\n", \
               i,tid[i], v[i]);
    }
  }
  else
    printf("Nao conseguiu criar processo filho\n");
  pvm_exit();
  exit(0);
}
```

mv_escravo.c

```
#include <stdio.h>
#include </user/pvm/pvm3/include/pvm3.h>
#define DIM     3
#define   N_TASKS    DIM

main ()
{
  int i, pid, linha_mat[DIM]= {0,0,0}, u_local[DIM];
  long elemento;

  pvm_mytid();
  pid = pvm_parent();

  /* recebe uma linha da matriz para processar */
  pvm_recv(pid, 63);
  pvm_upkint(linha_mat, DIM, 1);

  /* recebe uma copia do vetor u */
  pvm_recv(pid, 63);
  pvm_upkint(u_local, DIM, 1);

  /* calcula o produto do vetor pela linha da matriz */
  elemento = 0;
  for ( i = 0; i < DIM; i++)
    elemento += u_local[i] * linha_mat[i];

  /* envia o resultado obtido */
  pvm_initsend(0);
  pvm_pklong(&elemento, 1, 1);
  pvm_send(pid, 63);

  pvm_exit();
  exit(0);
}
```

A seguir, veja a saída em tela do programa acima :

```
v[0] recebido do processo [400E4] = 14
v[1] recebido do processo [400E3] = 32
v[2] recebido do processo [400E2] = 50
```

Como exercício, elabore um modelo de particuinamento dos dados e codifique um programa para calcular o produto escalar de dois vetores A e B, de tamanho N.

$$\text{Produto escalar} = \sum_{i=1}^{N} a_i \times b$$

7.2 - Modelo SPMD

No exemplo anterior, todos os processos de trabalho (escravos) são idênticos, mas poderiam ter sido criados vários outros, com funções diferenciadas, como entrada de dados, divisão da carga de trabalho, processamento, junção de resultados parciais e saída de resultados.

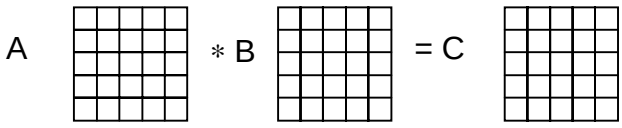
Há também um outro modelo, no qual trechos idênticos de código podem ser replicados, e vir a trabalhar sobre diferentes conjuntos de dados. No PVM, isto pode ser feito por um programa que cria cópias de seu próprio código.

```
#define NPROC 4
#include <stdio.h>
#include <sys/types.h>
#include "pvm3.h"
main( char *argv[], int argc )
{
  int myid;
  int tids[NPROC];

  myid = pvm_mytid();
  if ( pvm_parent( )==PvmNoParent )
    pvm_spawn(argv[0], (char**)0, 0, "", NPROC-1, &tids[1]);

  . . .
  /* corpo da tarefa */
  . . .

  pvm_exit();
  exit(0);
}
```

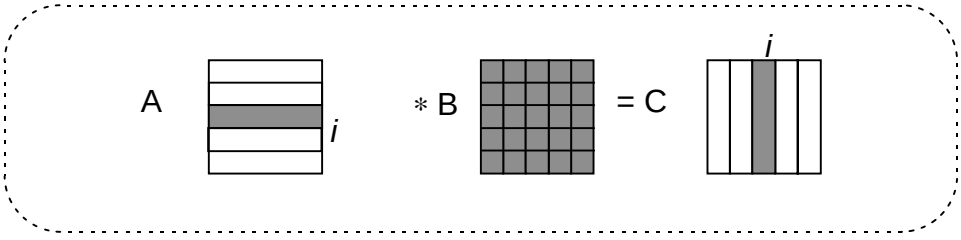
O exemplo acima testa o número de identificação do processo pai. Caso não exista um processo pai (isto significa que ele próprio é o pai), então serão criados os processos filhos, a partir do mesmo arquivo executável.

7.3 - Multiplicação de matrizes

Uma vez resolvido o problema de multiplicação de um veetor por uma matriz, torna-se fácil e imediato criar uma solução para a multiplicação entre duas matrizes.



Existem inúmeros algoritmos, baseados em diferentes maneiras de particionar e dividir os dados entre os nós de processamento. Certamente a maneira apresentada aqui não visa máxima eficiência, mas apenas a simplicidade. Na verdade, cada processador irá receber uma linha de A, e todos os dados da matriz B. Por exemplo, um certo nó *i* irá receber a *i-ésima* linha de A, uma cópia completa da matriz B, e deverá computar a *i-ésima* coluna da matriz C.Na figura abaixo as regiões escuras representam as partes das matrizes que residem em cada nó de processamento



A seguir, apresenta-se o código dos processos mestre e escravo para a multiplicação de matrizes com PVM.

mm_escravo.c

```
#include <stdio.h>
#include </user/pvm/pvm3/include/pvm3.h>
#define DIM     3
#define    N_TASKS    DIM

main ()
{
  int i,j,k, pid,soma;
  int  linha_matA[DIM]= {0,0,0};
  int  linha_matC[DIM]= {0,0,0};
  int B[DIM][DIM];
  long elemento;

  pvm_mytid();
  pid = pvm_parent();

  /* recebe uma linha da matriz A para processar */
  pvm_recv(pid, 63);
  pvm_upkint(linha_matA, DIM, 1);

  /* recebe uma copia da matriz B */
  pvm_recv(pid, 63);
  pvm_upkint(&(B[0][0]), DIM*DIM, 1);

  /* calcula o produto do vetor pela linha da matriz */
  elemento = 0;
  for ( j  = 0; j < DIM; j++) {
    soma = 0;
    for (k=0; k<DIM; k++)
      soma += B[k][j] * linha_matA[k];
    linha_matC[j]=soma;
  }
  /* envia o resultado obtido */
  pvm_initsend(0);
  pvm_pkint(linha_matC, DIM, 1);
  pvm_send(pid, 63);

  pvm_exit();
  exit(0);
}
```

mm_mestre.c

```
#include <stdio.h>
#include </user/pvm/pvm3/include/pvm3.h>
#define DIM     3
#define N_TASKS DIM
main ()
{
  int i,k,j, mytid, tid[N_TASKS], info;
  int A[DIM][DIM] = {1, 4, 2, 0, 1, 2, 3, -1, 1}; /*dados p/ teste */
```

```
  int B[DIM][DIM] = {1, 0, 2, -1, 1, 3, 0, 4, 1};
  int C[DIM][DIM];

  mytid = pvm_mytid ();
  info = pvm_spawn("filhomm", (char**)0, 0, "", N_TASKS, tid);
  if (info == N_TASKS) {
    /* Envia uma linha da matriz A para cada processo */
    for ( i=0; i<N_TASKS; i++) {
      pvm_initsend(0);
      pvm_pkint(&(A[i][0]),DIM, 1);
      pvm_send(tid[i], 63);
    }

    /* Envia copia da matriz B para cada processo */
    pvm_initsend(0);
    pvm_pkint(&(B[0][0]),DIM*DIM, 1);
    pvm_mcast(tid,N_TASKS, 63);

    /* Recebe uma linha da matriz resultado de cada processo */
    for ( i = 0; i < N_TASKS; i++) {
      pvm_recv(tid[i], 63);
      pvm_upkint(&(C[i][0]), DIM, 1);
      for (j =0; j<DIM; j++)
        printf ("  C[%d][%d] = %3d",i,j, C[i][j]);
      printf("\n");
    }
  }
  else
      printf("Nao conseguiu criar processo filho\n");
  pvm_exit();
  exit(0);
}
```

7.4 Bibliografia complementar.

Para quem deseja utilizar a biblioteca PVM, é fundamental ter à mão o tutorial [Gue94], publicado pelo MIT Press, que também pode ser obtido através da rede em versão html através da URL **http://www.netlib.org/pvm3/pvm-book.html**. Para os que desejam se aprofundar um pouco mais, [Ben90] faz uma introdução sobre programação distribuída, apresentando outros mecanismos de programação que vão além da troca explícita de mensagens. Uma visão geral sobre os diversos paradigmas existentes, e linguagens de programação distribuída pode ser vista em [Bal93]. Outro aspecto importante em programação distribuída, é a elaboração dos algoritmos, onde a questão da geração das mensagens costitui um fator preponderante na obtenção de alto desempenho [Kum94]. Um grupo de biliotecas e ferramentas são citados e referenciados por [Duk93], que também fornece um tutorial sobre a área de processamento distribuído baseado em redes de estações.

BIBLIOGRAFIA

[Bal89] BAL, H.E. et alii. Programming languages for distributed computing systems **ACM Computing Surveys**. v.21, n.3, p.261-322, 1989.

[Bal92] BAL, H.E. et alii. Orca: a language for parallel programming for distributed systems. **IEEE Trans. on Software Engineering**. v.18, n.3, 1992.

[Bau92] BAUER, B.E. **Practical parallel programming**. San Diego, Academic Press, 1992.

[Bel94] BELL, G. Scalable, parallel computers: alternative, issues and challenges**. Intl. Journal of Parallel Programming**. v.22, n.1, p.3-47. 1994.

[Ben90] BEN-ARI, M. **Principles of concurrent and distributed programming**. Cambridge, Prentice-Hall, 1990.

[Ber91] BERSHAD, B.N.; ZEKAUSKAS, M.J. **Midway: shared memory parallel programming with entry consistency for distributed shared memory multiprocessors**. Technical report CMU-CS-91-170, School of Computer Science, Carnegie-Mellon University, 1991.

[Cha90] CHANDRA, J.B. et alii. COOL: a language for parallel programming. In: **Programming Languages and Compilers for Parallel Computing**. MIT Press, England, 1990.

[Cha94] CHANDRA, J.B. et alii. COOL: an object-based language for parallel programming. **IEEE Computer**. p.13-26, 1994.

[Duk93] DUKE, D.W; ELIAS, D.; LIVNY, M.; TURCOTTE, L. Clustered Workstations Environments. In : SUPERCOMPUTING, Portland, 1993. **Tutorials**. Los Alamitos, IEEE Computer Society Press, 1993.

[Dun90] DUNCAN, R. A survey of parallel computer architectures. **IEEE Computer**. v.23, n.2, p.5-16. Feb. 1990.

[Fos95] FOSTER, I. **Designing and building parallel programs**: concepts and tools for parallel software engineering. Addison Wesley, 1995.

[Fox92] FOX, G. et alii. **Fortran D language specification**. Department of Computer Science, Rice University, 1992.

[Geh86] GEHANI, N.H.; ROOME, W.D. Concurrent C. **Software Practice and Experience**, v.16, n.9, p.821-44. Sept. 1986.

[Geh88] GEHANI, N.H.; ROOME, W.D. Rendezvous facilities: Concurrent C and Ada language. **IEEE Trans. on Software Engineering**. v.14, n.11. Nov.1988.

[Geh92] GEHANI, N.H.; ROOME, W.D. Implementing Concurrent C. **Software Practice and Experience**, v.22, n.3, p.267-85. March 1992..

[Gra92] GRANT, B.K.; SKELLUM, A. **The PVM system**: an in-depth analysis and documenting study-concise edition. Numerical Mathematics Group, Lawrence Livermore National Laboratory, 1992.

[Gri93] GRIMSHAW, A.S. Easy-to-use object-oriented parallel processing with Mentat. **IEEE Computer**, v.26, n.5, p.39-51, 1993.

[Gue94] GUEIST, A. et al. **PVM** : parallel virtual macine. A user's guide and tutorial for networked parallel computing. Cambridge, MIT Press, 1994.

[Gus86] GUZZI, M.D. **Multitasking runtime systems for the CEDAR multiprocessor**. Technical Report CSRD n.604, Center for Supercomputing Research and Development, University of Illinois at Urbana Champaign, 1986.

[Har91] HARRISON, L.; AMMARGUELAT, Z. A comparison of automatic versus manual parallelization of the Boyer-Moore Theorem Prover. In**: Languages and Compilers for Parallel Computing**. Pitman Publishing, 1991.

[Kel92] KELEHER, P. et alii. **Lazy release consistency for software distributed shared memory**. Department of Computer Science, Rice University. 1992.

[Kel93] KELEHER, P. et alii. **Treadmarks**: distributed shared memory on standard workstations and operating systems. Technical Report Rice COMP TR-93-214. Department of Computer Science, Rice University. 1993.

[Kof94] KOFUJI, S.T. **Considerações de projeto e análise do SPADE: um multiprocessador de larga escala baseado no padrão ANSI/IEEE SCI**. Tese de doutorado. Escola Politécnica da Universidade de São Paulo. 1994.

[Kum94] KUMAR, V. et al**. Introduction to Parallel Computing**. The Benjamin Cummings Publishing Company Inc., 1994.

[Kus94] KUSKIN, J. et alii. **The Stanford Flash multiprocessor**. Computer Systems Laboratory, Stanford University. 1994.

[Len92] LENOSKI, D. The Stanford DASH multiprocessor. **IEEE Computer**. v.25, n.3, p.63-79. March 1992.

[Lov93] LOVEMAN, D.B. **High Performance Fortran**. IEEE Parallel and Distributed Technology, Feb. 1993.

[Mei95] Meiko World Inc. **CS-2 Documentation Set**. 1995.

[Mid95] MIDORIKAWA, E.T.; SATO, L.M. Integrando as otimizações de acesso a dados e paralelismo. In: Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho, 7, Canela, RS. **Anais**. p.49-60. 1995.

[Nit91] NITZBERG, B.; LO, V. Distributed shared memory: a survey of issues and algorithms. **IEEE Computer**, p.52-60. August 1991.

[Pol88] POYCHRONOPOULOS, C.D. **Parallel programming and compilers**. Kluwer Academic Publishers. 1988.

[Pon94] PONTELLI, E. et alii. **A high-performance parallel Prolog system**. Laboratory for Logic and Databases, Department of Computer Science, New Mexico University, 1994.

[Rin93] RINARD, M.C. et alii. Jade: a high-level, machine-independent language for parallel programming. **IEEE Computer**, v.26, n.6, p.28-38. June 1993.

[Sal94] SALVADOR, L.N.; SATO, L.M. Uma linguagem de programação orientada a objetos para ambientes paralelos. In: Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho, 6, Caxambu, MG. **Anais**. p.107-22. 1994.

[Sal95a] SALVADOR, L.N. **Aspectos de paralelismo de uma linguagem baseada em objetos**. Dissertação de mestrado. Escola Politécnica da Universidade de São Paulo, 1995.

[Sal95b] SALVADOR, L.N.; SATO; L.M. Uma linguagem de programação baseada a objetos para ambientes paralelos. In: Simpósio Integrado de Software e Hardware, 22, Canela, RS. **Anais**. 1995.

[Sat92] SATO, L.M. Um sistema de programação e processamento para sistemas multiprocessadores. In: Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho, 4, São Paulo, SP. **Anais**. p.95-107. 1992.

[Sca93] SCALES, D.J.; LAM, M.S**. A flexible shared memory system for distributed memory machines**. Computer Systems Laboratory, Stanford University, 1993.

[Tha88] THAKKAR, S. The Balance multiprocessor system. **IEEE Micro**, v.8, n.1, p.57-69. Feb. 1988.

[Uta95] Univerity of Utah. **Application programming with Quarks**. Computer Systems Laboratory, 1995.

[Tur93] TURCOTTE, L.H. **A survey of software environment for exploiting networked computing resources**. Enginneering Research Center for Computational Field Simulation, Mississipi State, EUA, 1993.

Nome do arquivo: TUDO.DOC
Diretório: C:\USER\HS
Modelo: C:\WINWORD\MODELOS\JAI96.DOT
Título: Capítulo x - nonono
Assunto:
Autor: lsi
Palavras-chave:
Comentários:
Data de criação: 20/08/96 17:24
Número de revisões: 7
Última gravação: 20/08/96 18:12
Gravado por: lsi
Tempo total de edição: 36 Minutos
Última impressão: 20/08/96 18:14
Como a última impressão
Número de páginas: 58
Número de palavras: 16.280 (aprox.)
Número de caracteres: 92.797 (aprox.)