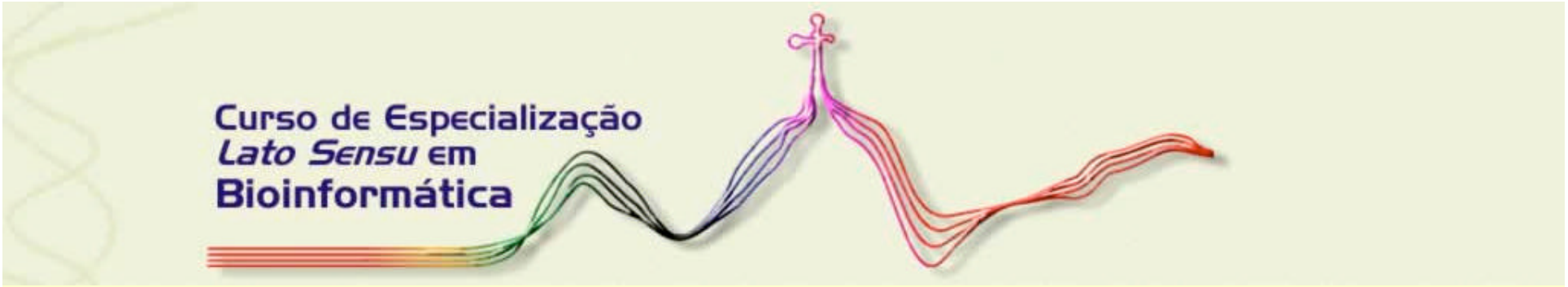


Programação Paralela

Simone de Lima Martins

Agosto/2002

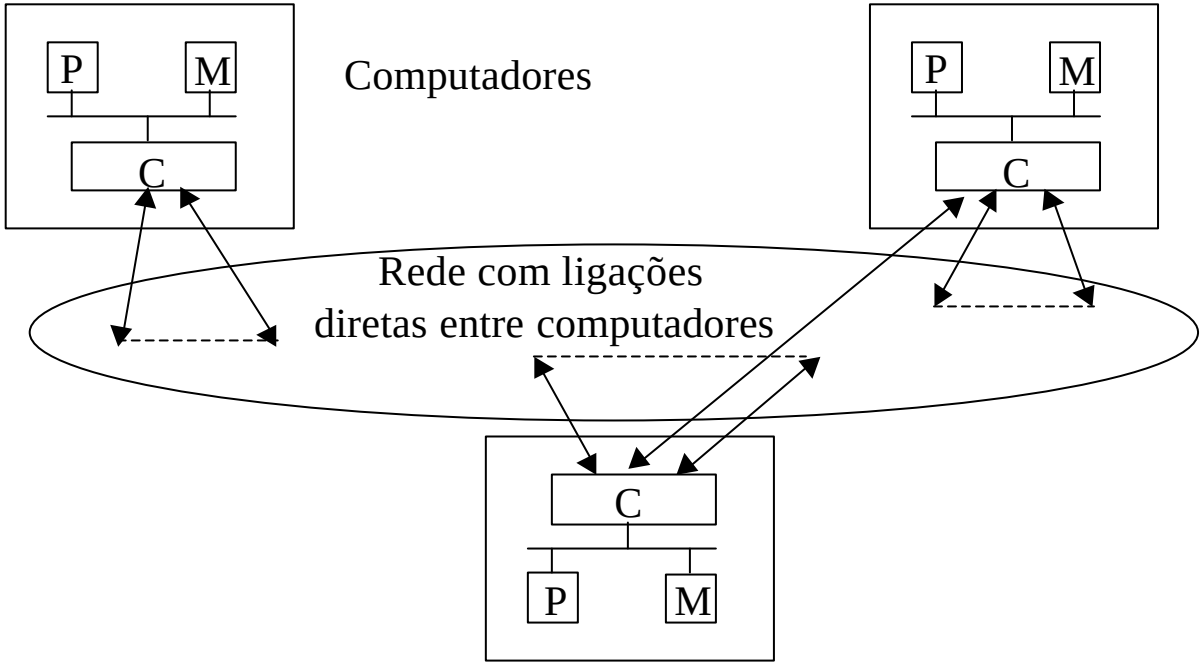
Computação por Passagem de Mensagens

Arquitetura de computadores que utilizam passagem de mensagens

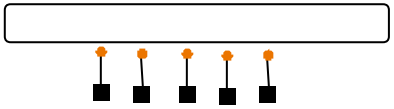
- **Computadores conectados por uma rede estática com passagem de mensagens**
- **Computadores conectados por redes locais ou geograficamente distribuídas**

Multiprocessadores conectados por rede

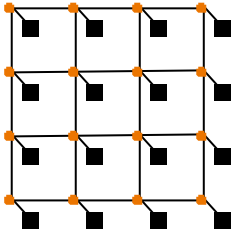
- **Processadores conectados por uma rede estática com passagem de mensagens**



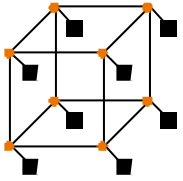
Topologia de interconexão



Anel



Grade bidimensional ou malha de 16 nós

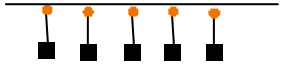


Árvore n-cubo de 8 nós

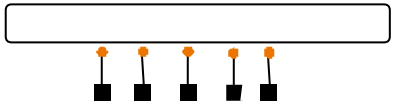
Computadores ligados em rede

- **Nós de multiprocessadores similares a computadores desktop**
- **Aumento da velocidade das redes locais**
- **Aparecimento de clusters de computadores de prateleira interligados por redes de alta velocidade**

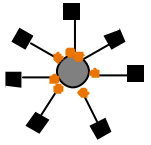
Topologia de interconexão



Barra



Anel



Estrela

Clusters

- **Aparecimento de clusters de computadores de prateleira interligados por redes de alta velocidade**
- **Rede de computador dedicada**
- **Exemplos:**
  - Sandia Cplant (592 Compaq XP100 workstations interconectadas pela Myrinet)
  - IBM RS/6000 SP2 (512 nós, switch própria)
  - Celera (700 CPUs conectadas por Gigabit Ethernet)

Clusters - Exemplos

- **http://www.top500.org/**
- **IBM ASCI White, SP (mais rápida)**
  - 8192 processadores
  - 6 Tbytes de memória e 160 Tbytes de disco
  - Interconexão por switch própria
- **Celera**
  - 10 estações com 6 a 12 processadores cada e com 24 Gbytes de memória
  - Um servidor com 16 processadores e 64 Gbytes de memória
  - 150 estações com 4 processadores cada e com memória variando de 2 a 32 Gbytes de memória
  - 70 Tbytes de disco

Clusters - Exemplos

- **BioCluster (COMPAQ)**
  - 25 estações ALPHA com 4 processadores cada e 4 Gbytes de memória
  - 1 nó com 16 Gbytes de memória
  - 1 servidor de arquivo com 4 Gbytes de memória e 1 Tbytes de disco
  - BLAST, FASTA, CLUSTALW

Programação por passagem de mensagens

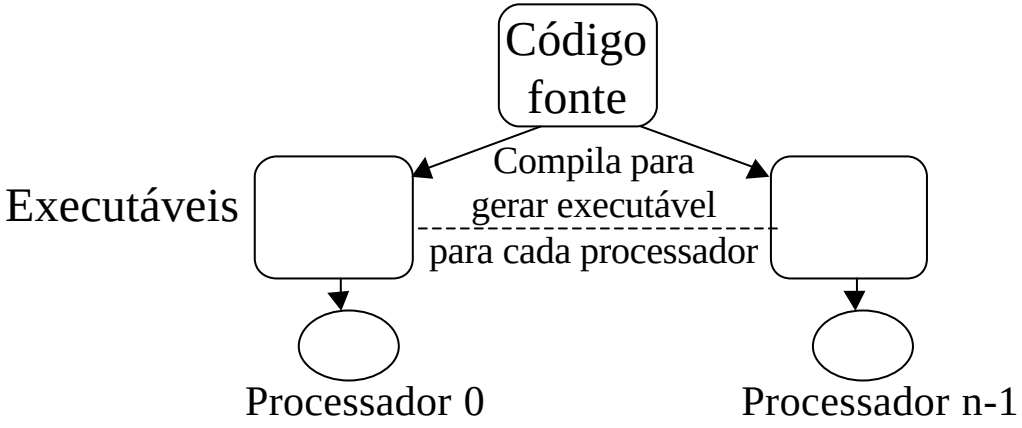
- **Programação de multiprocessadores conectados por rede pode ser realizada:**
  - criando-se uma linguagem de programação paralela especial
  - estendendo-se as palavras reservadas de uma linguagem seqüencial existente de alto nível para manipulação de passagem de mensagens
  - utilizando-se uma linguagem seqüencial existente, provendo uma biblioteca de procedimentos externos para passagem de mensagens

Biblioteca de rotinas para troca de mensagens

- **Necessita-se explicitamente definir:**
  - quais processos serão executados
  - quando passar mensagens entre processos concorrentes
  - o que passar nas mensagens
- **Dois métodos primários:**
  - um para criação de processos separados para execução em diferentes processadores
  - um para enviar e receber mensagens

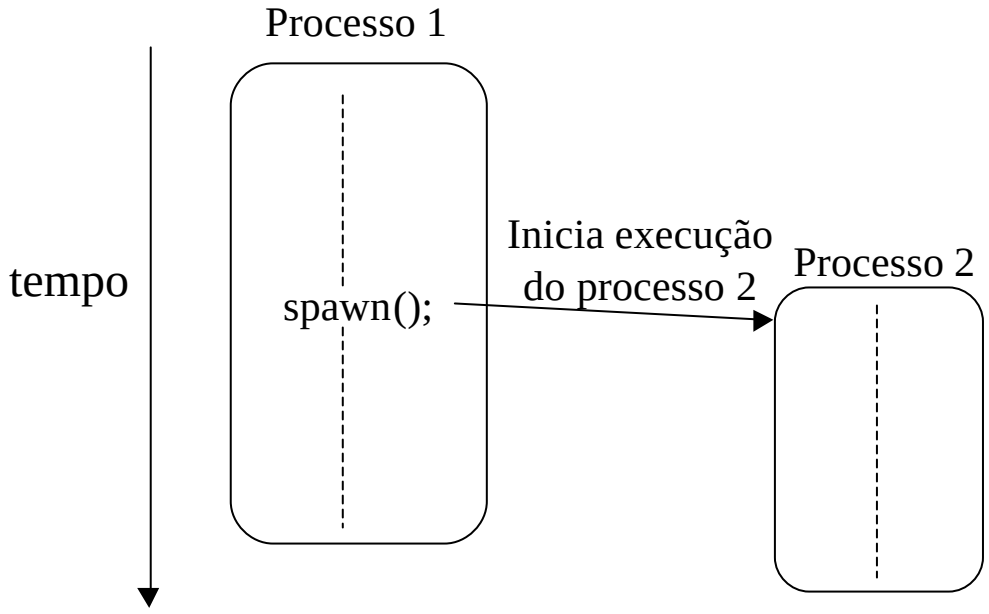
Modelo Single Program Multiple Data (SPMD)

- **Diferentes processos são unidos em um único programa e dentro deste programa existem instruções que customizam o código, selecionando diferentes partes para cada processo por exemplo**

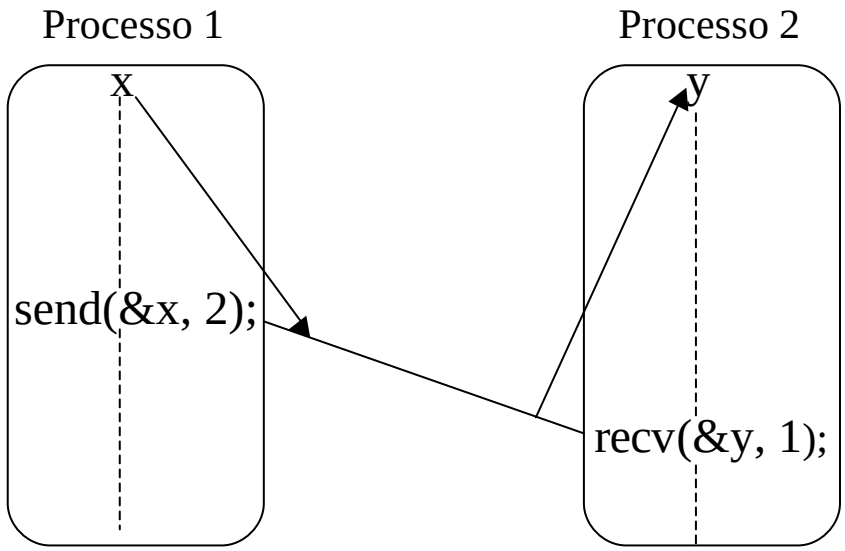


Modelo Multiple Program, Multiple Data (MPMD)

- **Programas separados escritos para cada processador, geralmente se utiliza o método mestre-escravo onde um processador executa o processo mestre e os outros processos escravos são inicializados por ele.**

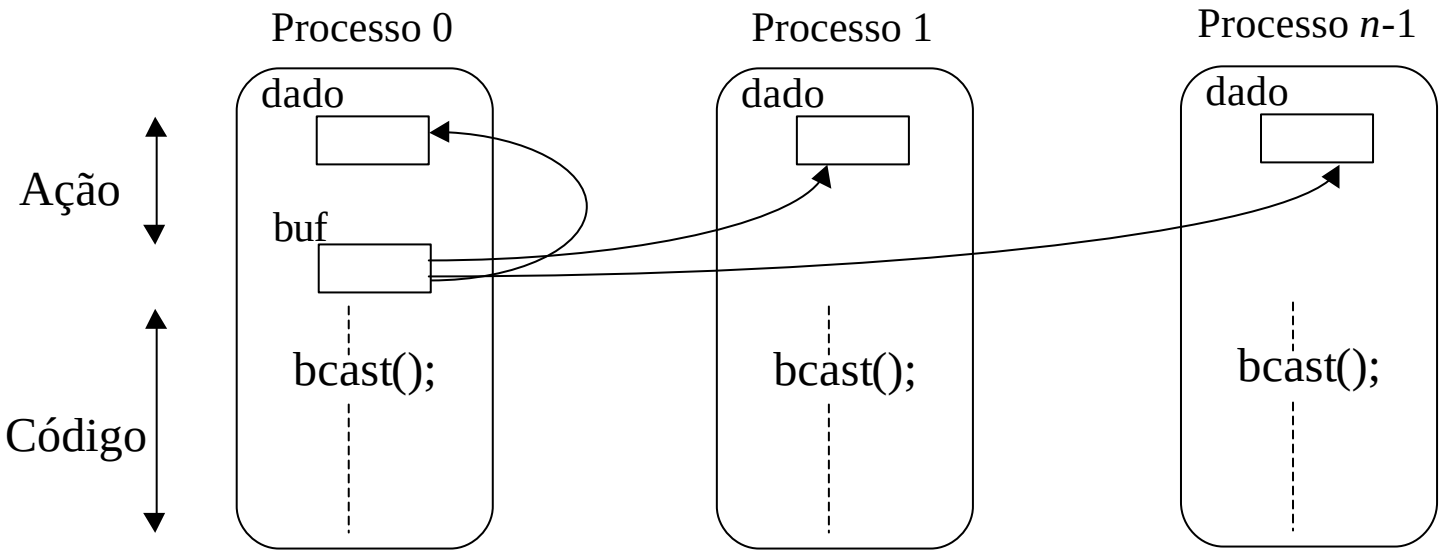


Rotinas básicas de envio e recebimento



Broadcast

- **Envio da mesma mensagem para todos os processos**
- ***Multicast*: envio da mesma mensagem para um grupo de processos**



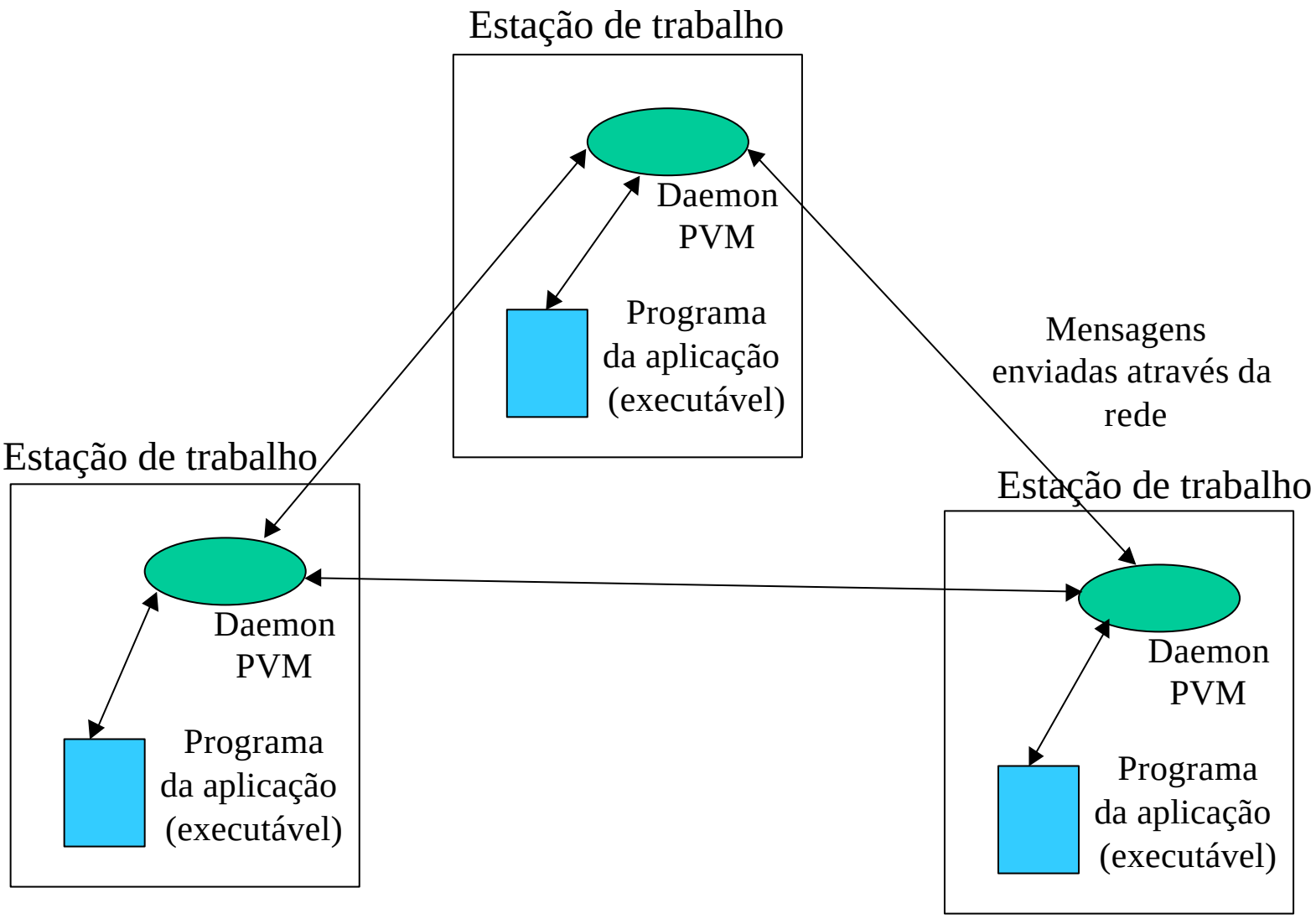
Ferramentas de software que utilizam biblioteca de troca de mensagens

- **PVM (Parallel Virtual Machine)**
  - desenvolvida pelo Oak Ridge National Laboratories para utilizar um cluster de estações de trabalho como uma plataforma multiprocessada
  - provê rotinas para passagem de mensagens entre máquinas homogêneas e heterogêneas que podem ser utilizadas com programas escritos em C ou Fortran
- **MPI (Message Passing Interface)**
  - padrão desenvolvido por um grupo de parceiros acadêmicos e da indústria para prover um maior uso e portabilidade das rotinas de passagem de mensagens

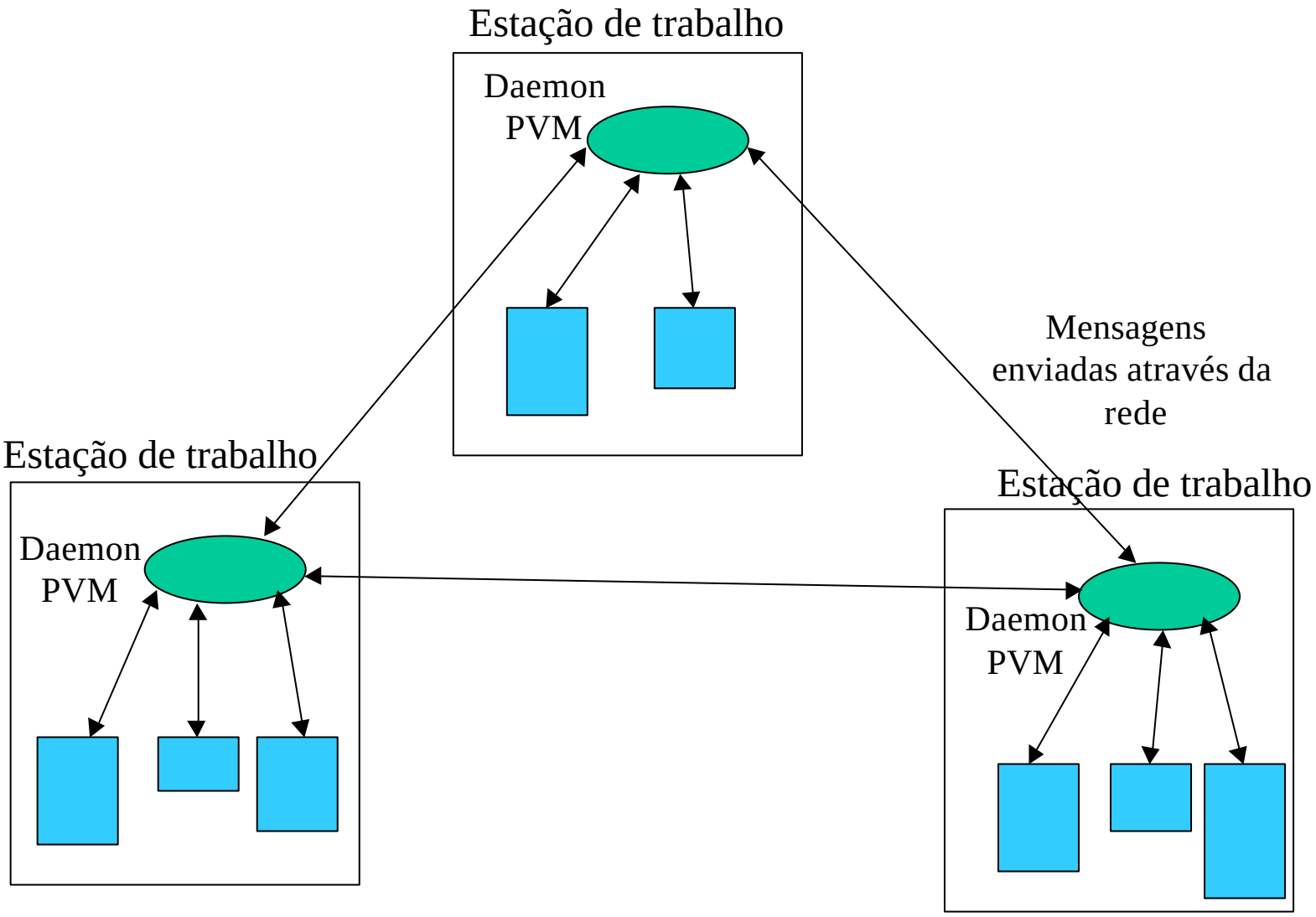
PVM

- **O programador decompõe o programa em programas separados e cada um deles será escrito em C ou Fortran e compilado para ser executado nas diferentes máquinas da rede**
- **O conjunto de máquinas que será utilizado para processamento deve ser definido antes do início da execução dos programas**
- **Cria-se um arquivo (*hostfile)* com o nome de todas as máquinas disponíveis que será lido pelo PVM**
- **O roteamento de mensagens é feito pelos processos *daemon* do PVM**

Passagem de mensagens utilizando o PVM



Passagem de mensagens utilizando o PVM



MPI

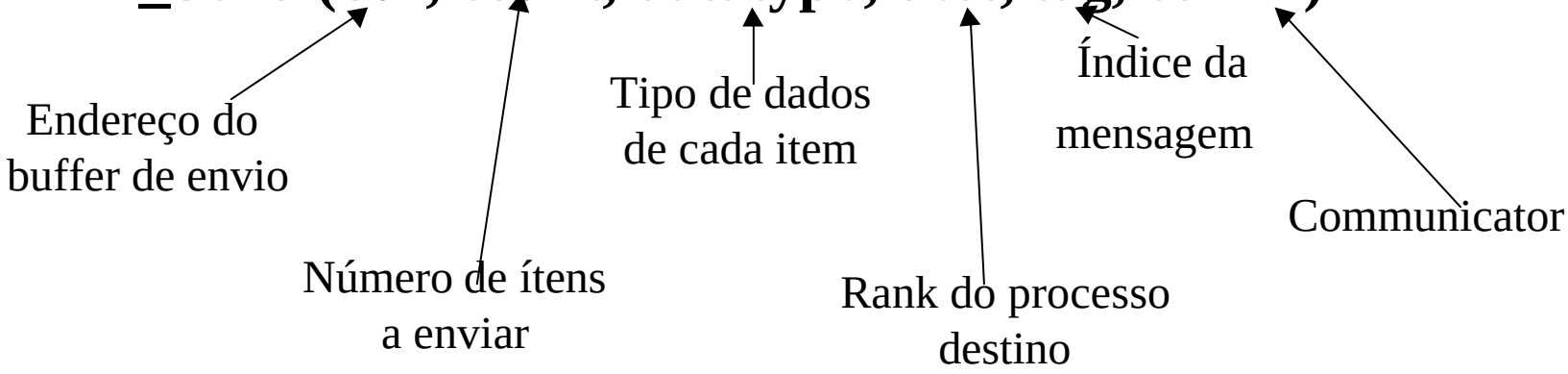
- **Criação e execução de processos**
  - Propositalmente não são definidos e dependem da implementação
  - MPI versão 1
    - Criação estática de processos: processos devem ser definidos antes da execução e inicializados juntos
    - Utiliza o modelo de programação SPMD

Utilizando o modelo SPMD

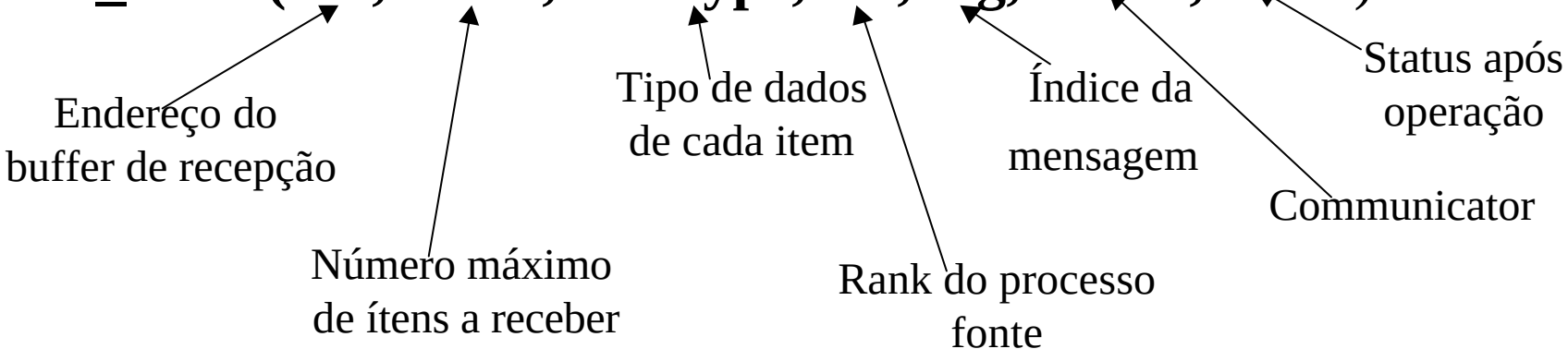
```
main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    .
    .
    MPI_Comm_Rank(MPI_COMM_WORLD, &myrank);
     if (myrank ==0)
        master();
     else
         slave();
     .
     .
     MPI_Finalize();
}
```

Rotinas com bloqueio

MPI_Send (buf, count, datatype, dest, tag, comm)



MPI_Recv (buf, count, datatype, src, tag, comm,status)



Exemplo de rotina com bloqueio

Envio de um inteiro *x* do processo 0 para o processo 1

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank ==0) {
   int x;
   MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
}
else  if (myrank ==1) {
   int x;
   MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

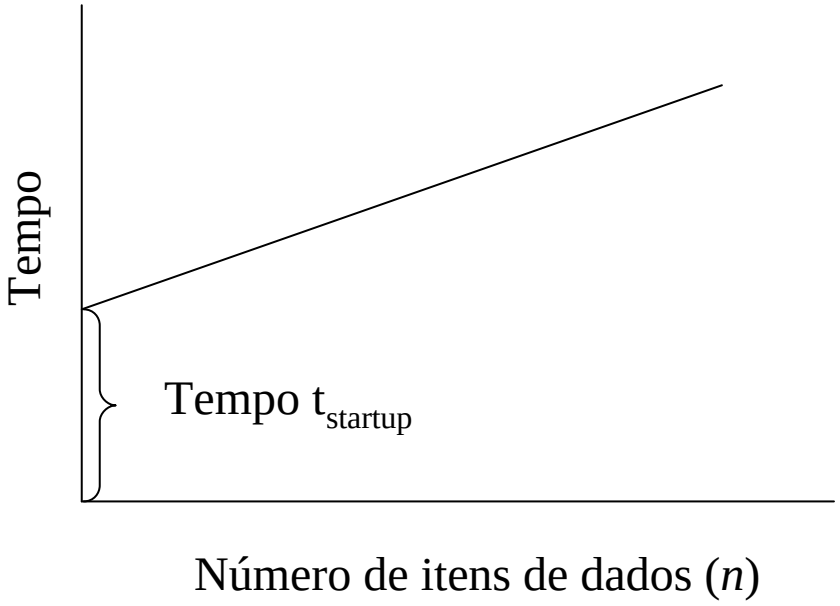
Comunicação coletiva

- MPI_Bcast()**: envio do processo raíz para todos os outros**
- MPI_Alltoall()**:envia dados de todos os processos para todos os processos**

Tempo de execução paralela

- **Composto de duas partes:**
  - t_{comp} : parte de computação
  - t_{comm} : parte de comunicação
  - $t_p = t_{comp} + t_{comm}$
- **Tempo de computação estimado como em algoritmo seqüencial**
- **Tempo de comunicação:**
  - $t_{comm} = t_{startup} + nt_{data}$
  - $t_{startup}$: tempo para enviar uma mensagem sem dados
  - t_{data} : tempo para transmitir uma palavra de dados
  - n: número de palavras a transmitir

Tempo teórico de comunicação

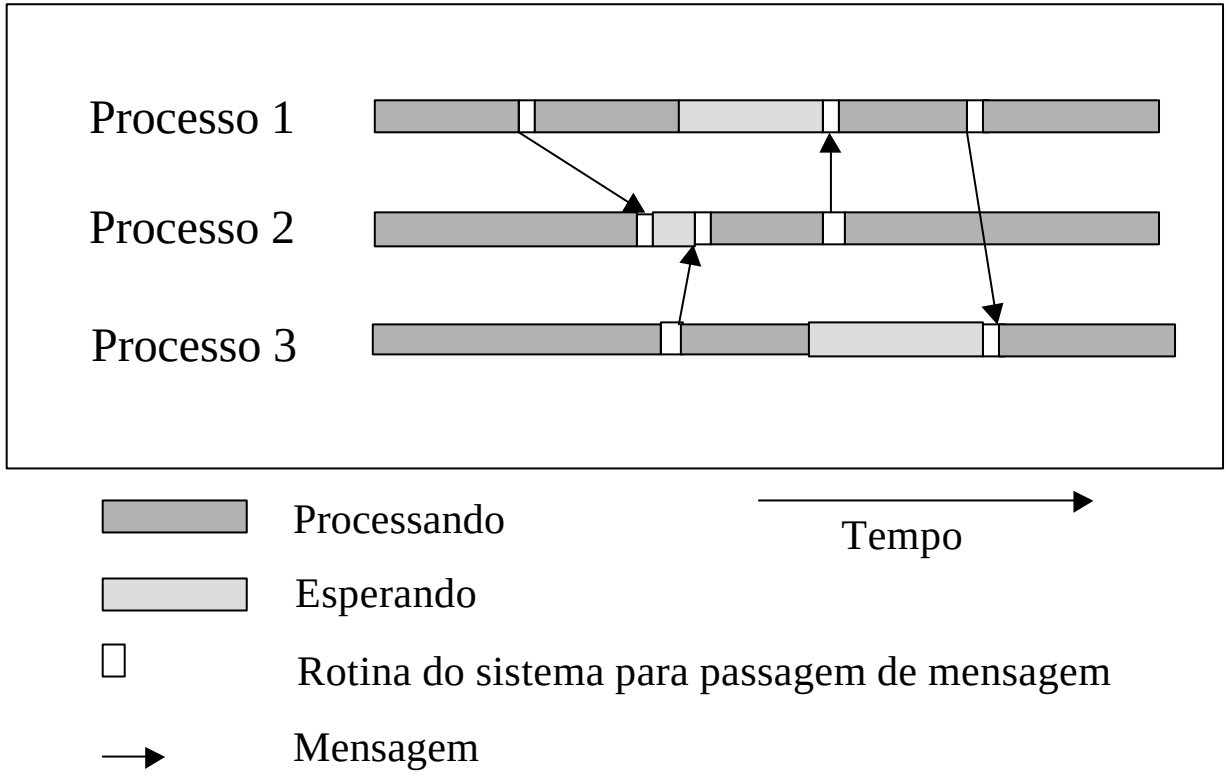


Algoritmos com custo ótimo

- **Custo é considerado ótimo quando o custo de resolver um problema é proporcional ao tempo de execução em um único processador utilizando o mais rápido algoritmo seqüencial conhecido**
  - Custo = t_p ? n = k ? t_s

Depurando e avaliando os programas paralelos

- **Diagrama de espaço-tempo**



Estratégias de depuração

- **Sugestão de Geist et al. Para depuração de programas paralelos:**
  1. Execute o programa como um único processo e depure como um programa seqüencial (se possível)
  2. Execute o programa utilizando dois ou quatro processos em uma mesma máquina e verifique se mensagens estão sendo enviadas de forma correta (erros são comuns nos índices e destinos das mensagens)
  3. Execute o programa utilizando os mesmos dois ou quatro processos em várias máquinas para tentar descobrir problemas na sincronização e temporização no programa que podem aparecer devido a atrasos na rede

Avaliando programas empiricamente

- **Medindo tempo de execução**

```
L1:  time(&t1);
.
.
L2: time(&t2);
.
elapsed_time = difftime (t2,t1);
printf("Elapsed_time = %5.2f segundos", elapsed_time);
```

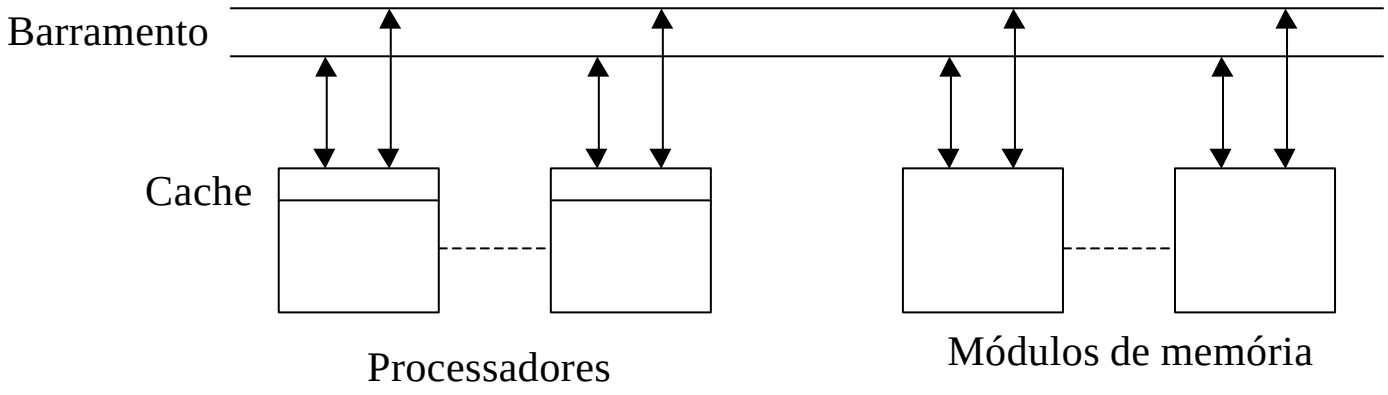
- **Rotina** MPI_ Wtime() provê hora em segundos

Memória Compartilhada

Programação com memória compartilhada

- **Nos sistemas multiprocessadores com memória compartilhada, qualquer localização da memória pode ser acessada por qualquer processador**
- **Existe um espaço de endereçamento único, ou seja, cada localização de memória possui um único endereço dentro de uma extensão única de endereços**

Arquitetura com barramento único



Alternativas para programação

- **Utilizar uma nova linguagem de programação**
- **Modificar uma linguagem seqüencial existente**
- **Utilizar uma linguagem seqüencial existente com rotinas de bibliotecas**
- **Utilizar uma linguagem de programação seqüencial e deixar a cargo de um compilador paralelizador a geração de um código paralelo executável**

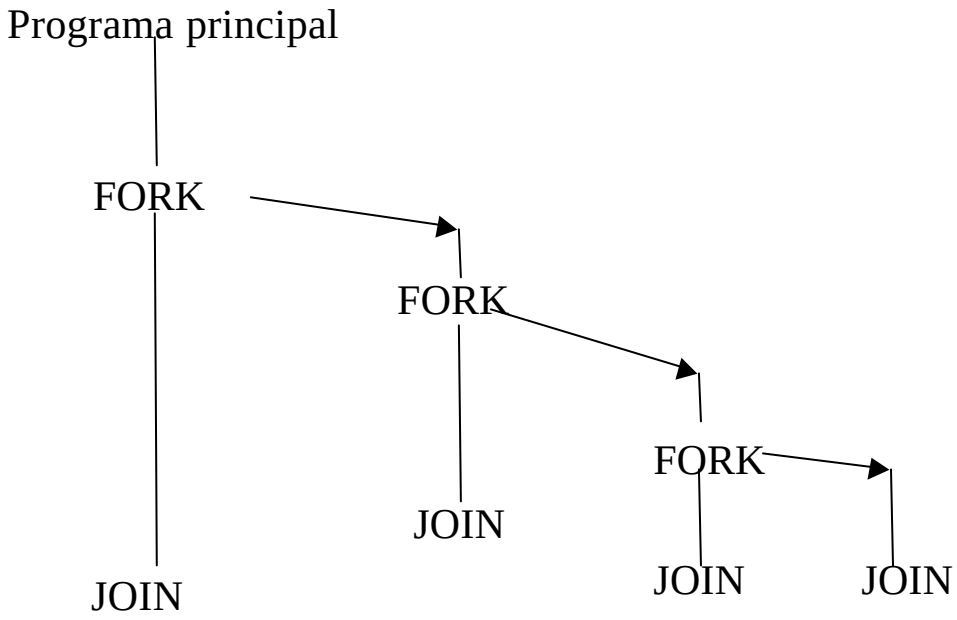
Algumas linguagens de programação paralelas

Linguagem	Criador/data	Comentários
Ada	Depto. de defesa americano	Nova linguagem
C*	Thinking Machines, 1987	Extensão a C para sistemas SIMD
Concurrent C	Gehani e Roome, 1989	Extensão a C
Fortran F	Foz et al., 1990	Extensão a Fortran para programação por paralelismo de dados
Modula-P	Braünl, 1986	Extensão a Modula 2
Pascal concorrente	Brinch Hansen, 1975	Extensão ao Pascal

Elementos básicos

- **Processos**
- **Threads**

Criação de processos concorrentes utilizando a construção *fork-join*



Processos UNIX

- **A chamada do UNIX *fork*() cria um novo processo, denominado processo filho**
- **O processo filho é uma cópia exata do processo que chama a rotina *fork*(), sendo a única diferença um identificador de processo diferente**
- **Esse processo possui uma cópia das variáveis do processo pai inicialmente com os mesmos valores do pai**
- **O processo filho inicia a execução no ponto da chamada da rotina**
- **Caso a rotina seja executada com sucesso, o valor 0 é retornado ao processo filho e o identificador do processo filho é retornado ao processo pai**

Processos UNIX

- **Os processos são "ligados" novamente através das chamadas ao sistema:**
  - wait(status): atrasa a execução do chamador até receber um sinal ou um dos seus processos filhos terminar ou parar
  - exit(status): termina o processo
- **Criação de um único processo filho**

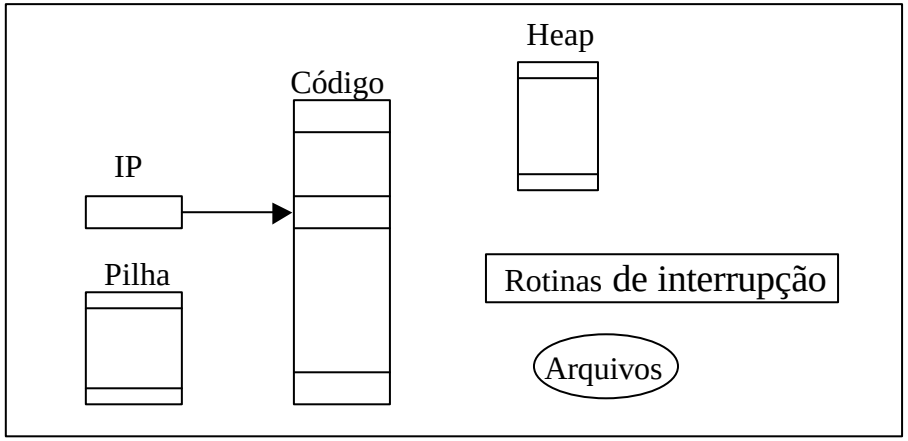
```
pid  =  fork();
código a ser executado pelo pai e filho
if (pid == 0) exit (0); else wait (0)
```

- **Filho executando código diferente**

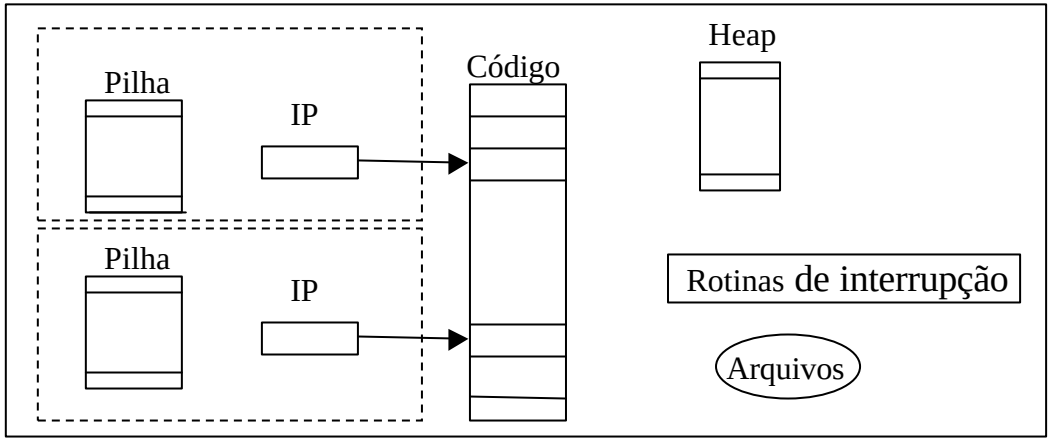
```
pid = fork();
if (pid == 0) {
   código a ser executado pelo filho
} else {
        código a ser executado pelo pai
}
if (pid == 0) exit (0); else wait (0);
```

Threads

Processos: programas completamente separados com suas próprias variáveis, pilha e alocação de memória

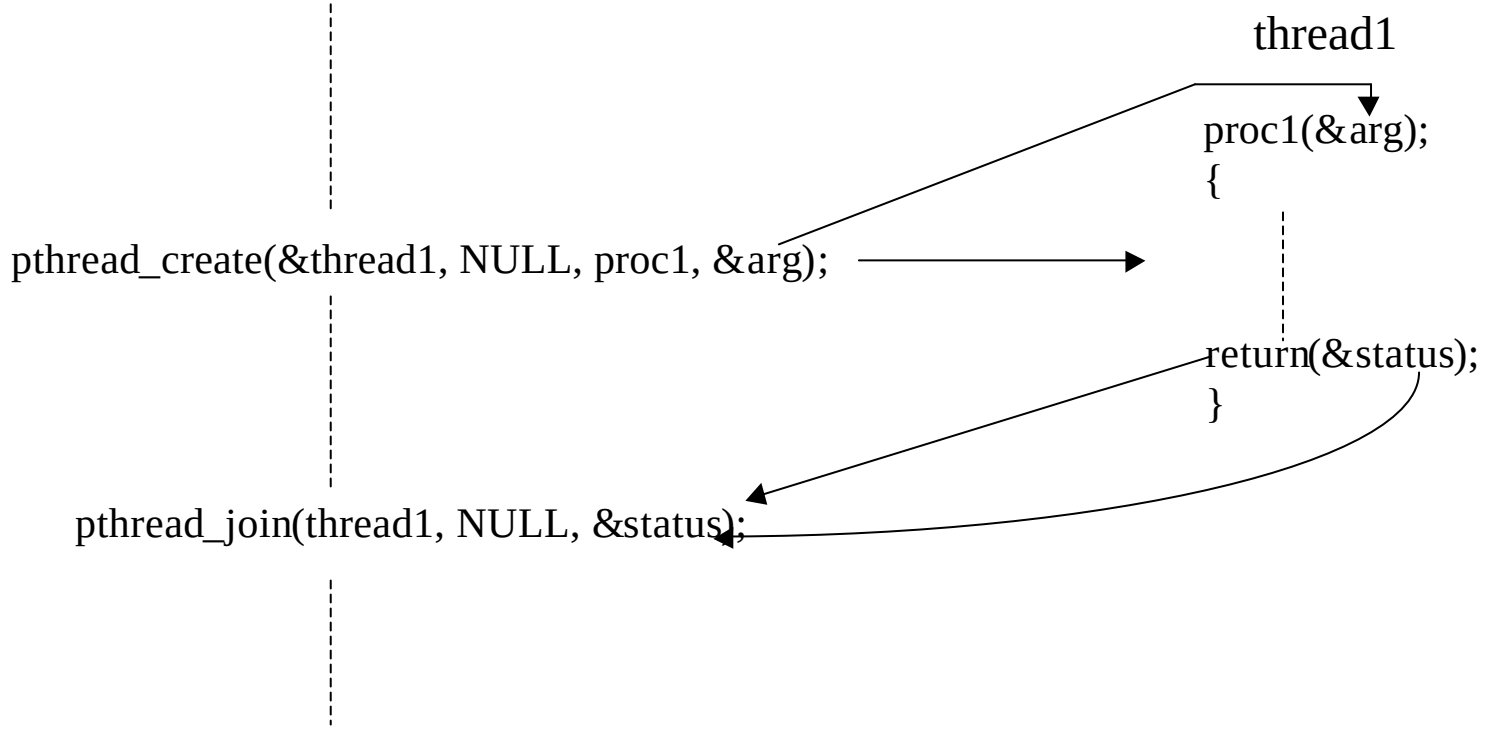


Threads: as rotinas compartilham o mesmo espaço de memória e variáveis globais



Pthreads

Interface portável do sistema operacional, POSIX, IEEE

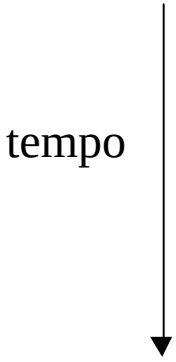


Rotinas seguras para threads

- **As chamadas ao sistema ou rotinas de bibliotecas são denominadas seguras para threads, quando elas podem ser acionadas por várias threads ao mesmo tempo e sempre produzem resultados corretos**
- **Rotinas padrão de E/S: imprimem mensagens sem permitir interrupção entre impressão de caracteres**
- **Rotinas que acessam dados compartilhados ou estáticos requerem cuidados especiais para serem seguras para threads**
- **Pode-se forçar a execução da rotina somente por uma thread de cada vez, ineficiente**

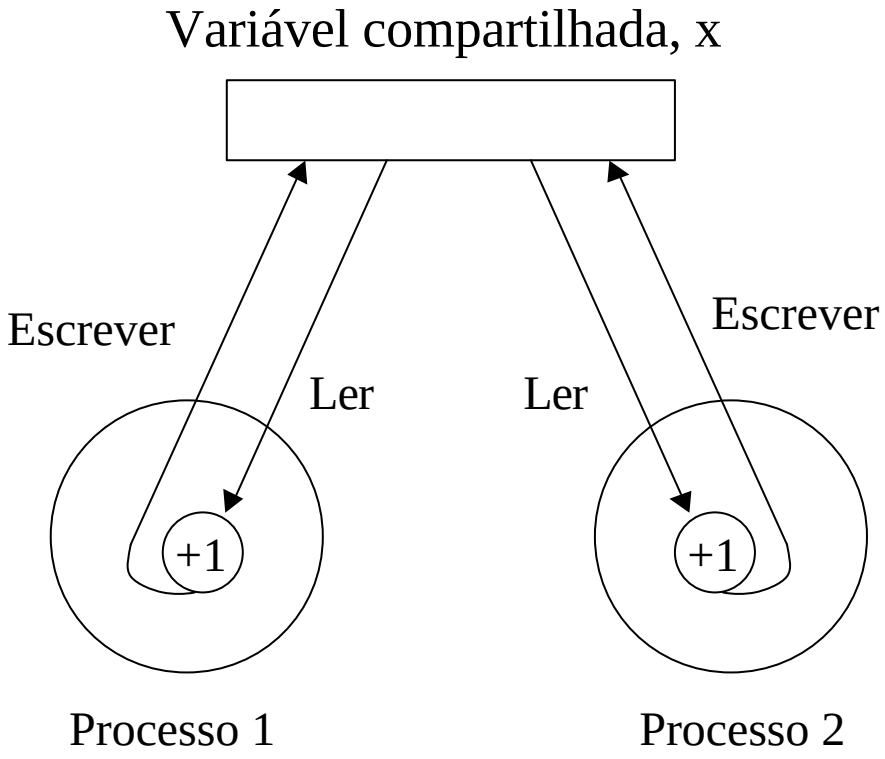
Acessando dados compartilhados

- **Considere dois processos que devem ler o valor de uma variável *x*, somar 1 a esse valor e escrever o novo valor de volta na variável *x***



Instrução	Processo 1	Processo 2
x = x + 1	ler x	ler x
	calcular x + 1	calcular x + 1
	escrever x	escrever x

Conflito em acesso a dados compartilhados

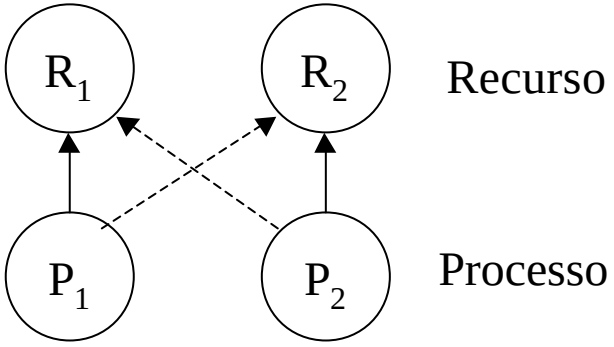


Rotinas de lock

- **Para poder se acessar estas variáveis compartilhadas sem problemas, devem existir rotinas que bloqueiam o acesso a elas enquanto uma das threads está executando**
- **Rotinas de lock**

Deadlock

- **Pode ocorrer com dois processos quando um deles precisa de um recurso que está com outro e esse precisa de um recurso que está com o primeiro**

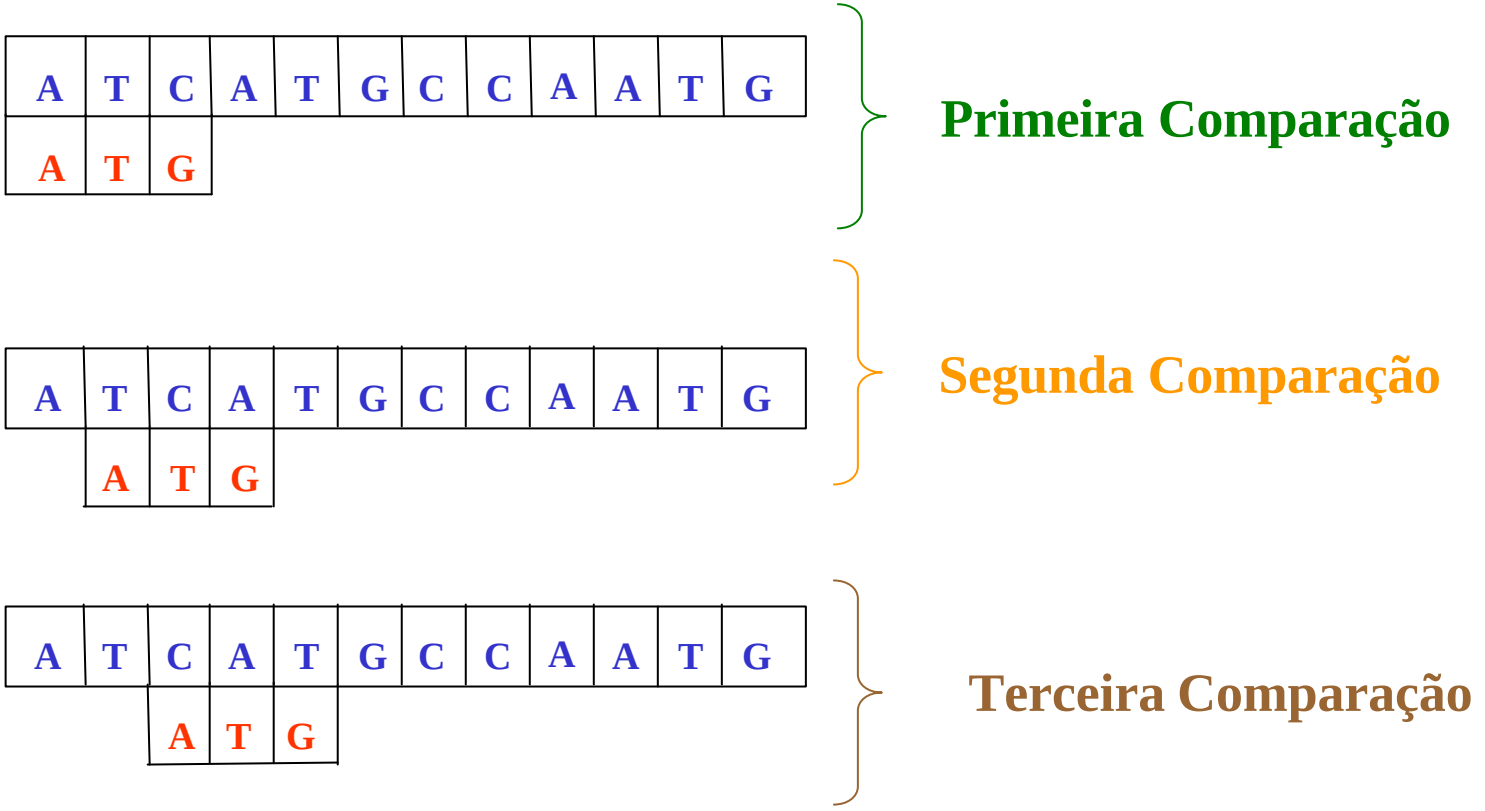


Exemplo de implementação

- **A procura de um padrão em seqüências de DNA**
- **Algoritmo com granulosidade fina**
- **Algoritmo com granulosidade média**
- **Implementação com troca de mensagens**
- **Implementação com threads (memória compartilhada)**

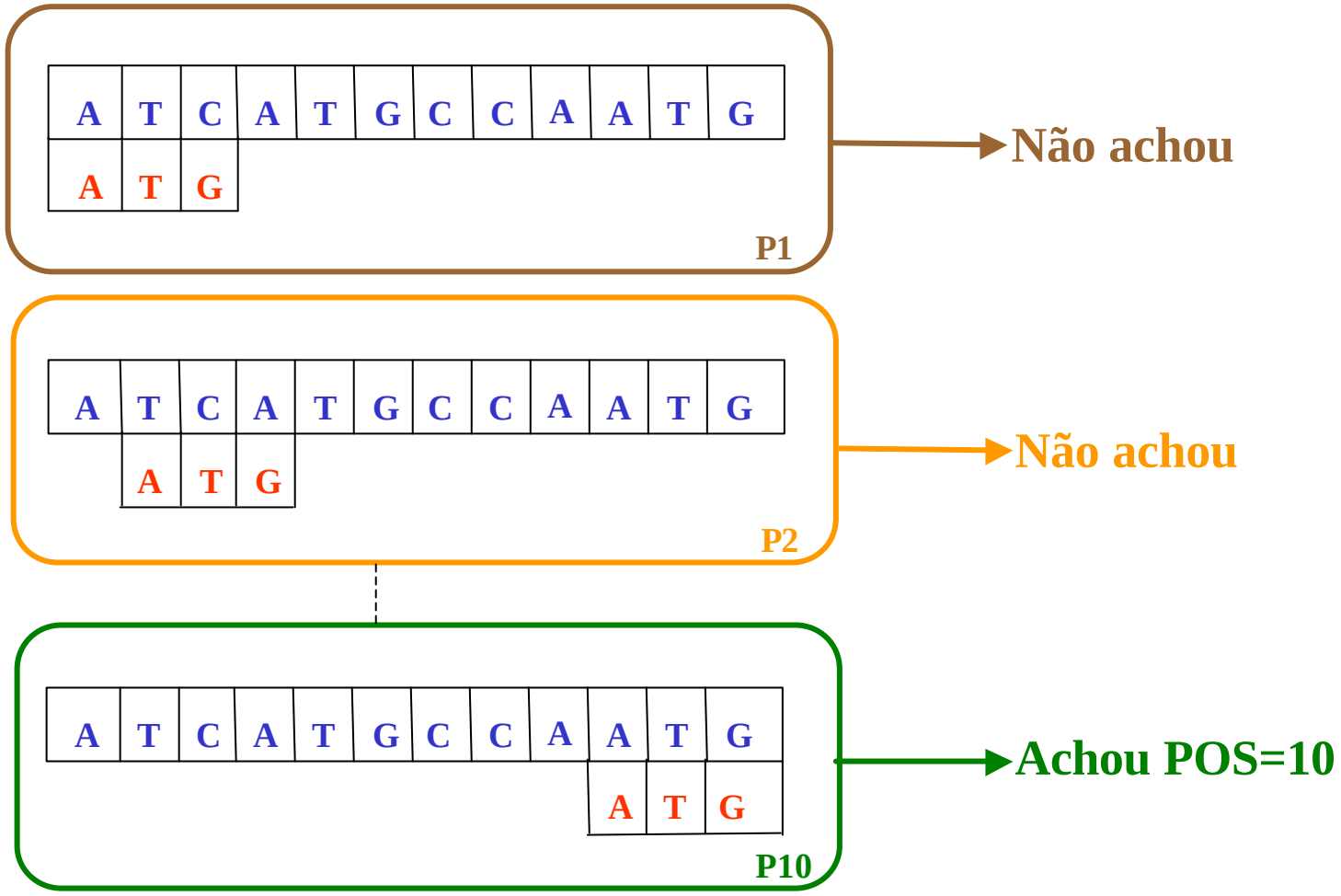
Exemplo de implementação

- **Algoritmo seqüencial simples que não é o mais eficiente**



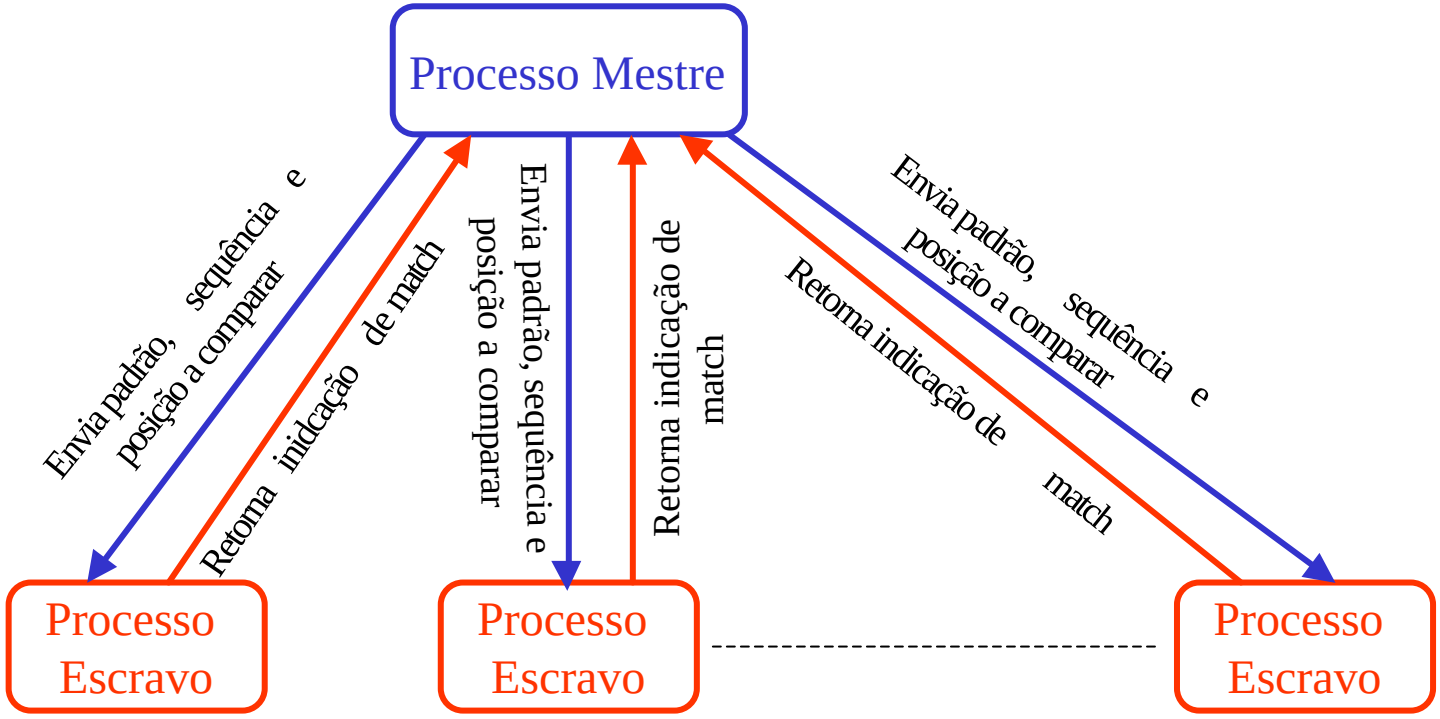
Granulosidade fina

- **Cada processador faz uma comparação**



Granulosidade fina com troca de mensagens

- **Um processo mestre controla as operações**
- **Processos escravos executam as comparações e devolvem resultados**



Granulosidade fina com troca de mensagens - Processo mestre

- **Envia padrão P[1..N] aos escravos**
- **Para cada seqüência S_i[1..M] faça**
  - Envia S_i para cada processador
  - Verifica quantas comparações devem ser feitas por cada processador
    - (M-N+1)/NP
  - Envia o intervalo de comparações para cada processador
  - Espera resultado de cada processador
- **Envia indicação de fim de processo para cada processador**

Granulosidade fina com troca de mensagens - Processo escravo

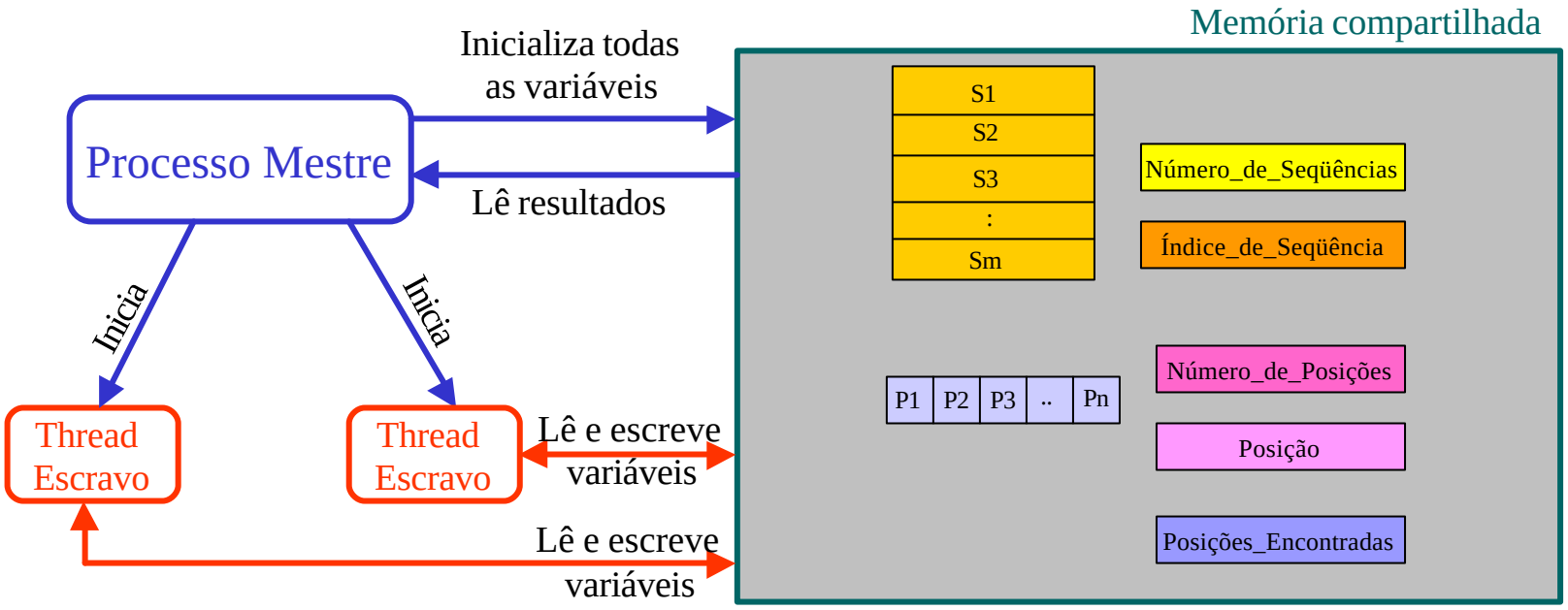
- **Recebe padrão P[1..N] do mestre**
- **Enquanto não recebe indicação de fim de processo do mestre faça**
  - Recebe S_i
  - Recebe as comparações que devem ser feitas
  - Executa as comparações
  - Envia resultado para mestre

Granulosidade fina com troca de mensagens - Exemplo

- **P[1..3]=ATG**
- **300 seqüências de tamanho 12**
- **5 processadores**
- **Devem ser executadas 10 comparações**
- **Cada processador executa 2 comparações**
- **Análise de desempenho**

Granulosidade fina com threads

- **Um processo mestre controla as operações através de variáveis compartilhadas**
- **Processos escravos executam as comparações e devolvem resultados através de variáveis compartilhadas**



Granulosidade fina com threads - Processo mestre

- **Lê Padrão P[1..N], Número_de_Seqüências e Seqüências S_i[1..M]**
- **Inicializa variáveis de controle**
  - Índice_de_Seqüência=1; Posição=1; Número_de_Posições = (M-N+1)/NP
- **Inicia threads escravas**
- **Enquanto Índice_de_Seqüência < Número_de_Seqüências faça**
  - Espera todas as threads lerem Índice_de_Seqüência
  - Espera threads executarem comparações
  - Imprime resultados que estão em Posições_Encontradas
  - Incrementa Índice_de_Seqüência

Granulosidade fina com threads - Thread escrava

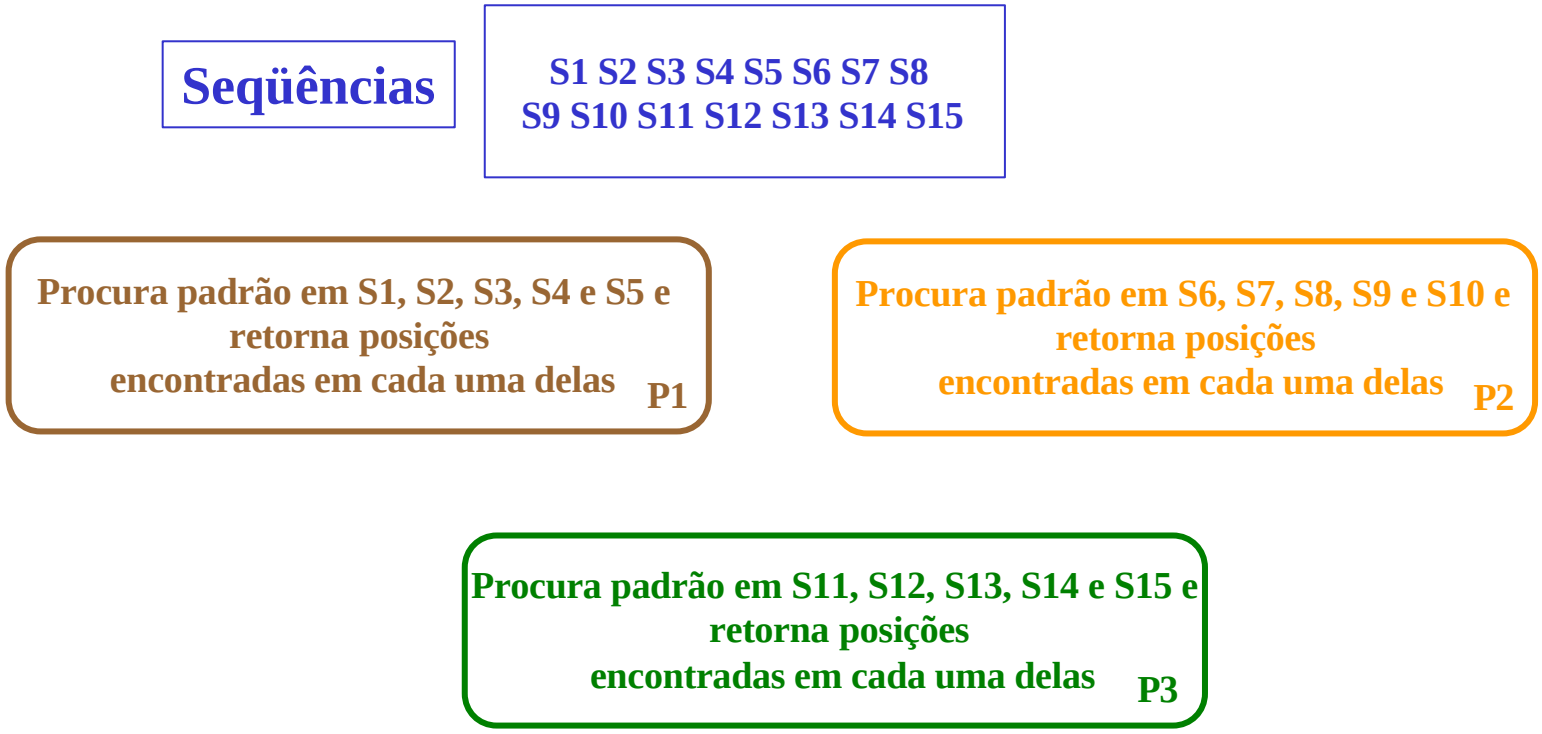
- **Repita**
  - Lê Índice_de_Seqüência
  - Espera todas as threads lerem Índice_de_Seqüência
  - Espera Posição estar desbloqueada
  - Bloqueia Posição
  - Lê Posição
  - Incrementa Posição de Número _de_Posições
  - Desbloqueia Posição
  - Executa as comparações com $\text{Seqüência}_{\text{Índice\_de\_Seqüência}}$
  - Coloca resultado em Posições_Encontradas
- **Até Índice_de_Seqüência > Número_de_Seqüências**

Granulosidade fina com threads - Exemplo

- **P[1..3]=ATG**
- **300 seqüências de tamanho 12**
- **5 processadores**
- **Devem ser executadas 10 comparações**
- **Cada processador executa 2 comparações**
- **Análise de desempenho**

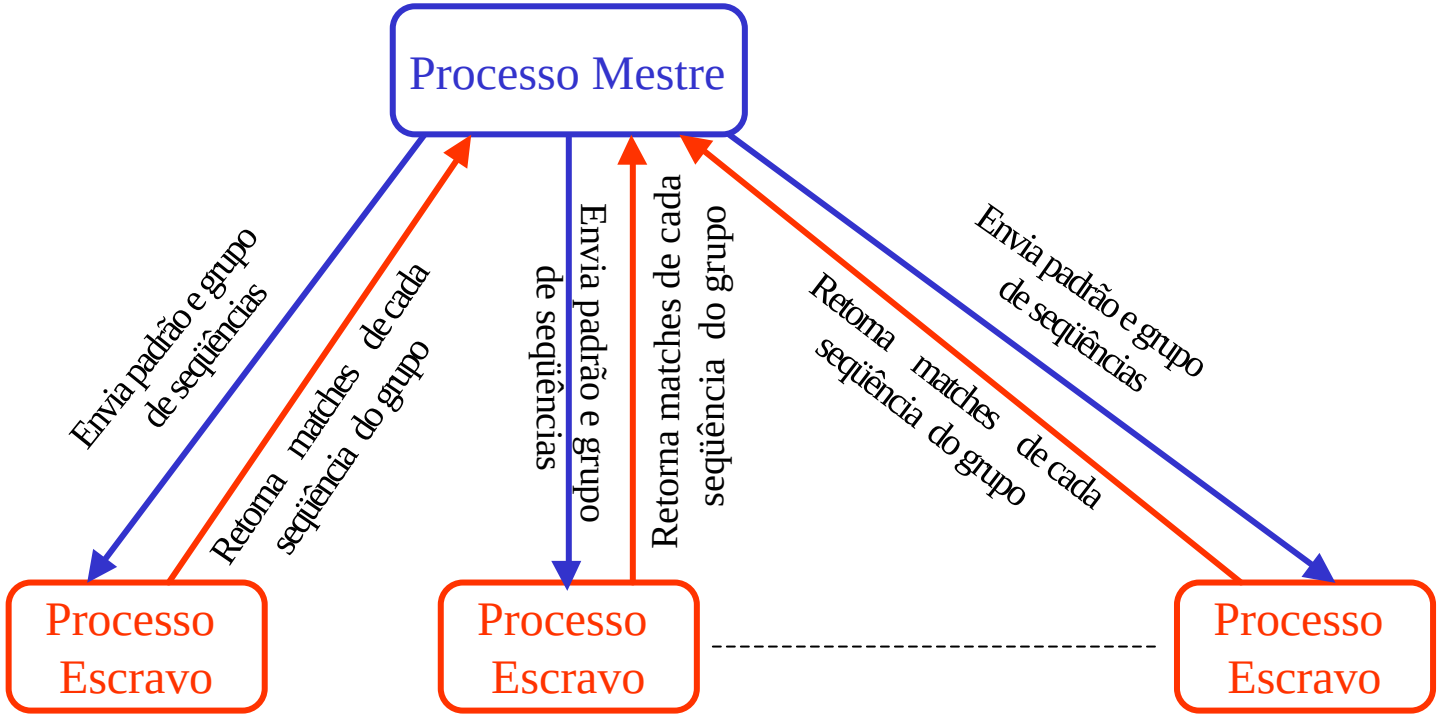
Granulosidade média

- **Divide as seqüências a serem procuradas entre os processadores e cada processador procura o padrão nas suas seqüências**



Granulosidade média com troca de mensagens

- **Um processo mestre envia grupos de seqüências para os processadores escravos**
- **Processos escravos procuram o padrão em seu grupo de seqüências e enviam resultados para o processo mestre**



Granulosidade média com troca de mensagens - Processo mestre

- **Envia (Número_de_Seqüências/NP) seqüências S_i[1..M] e padrão P[1..N] para cada processador**
- **Espera resultados de cada processador**

Granulosidade média com troca de mensagens - Processo escravo

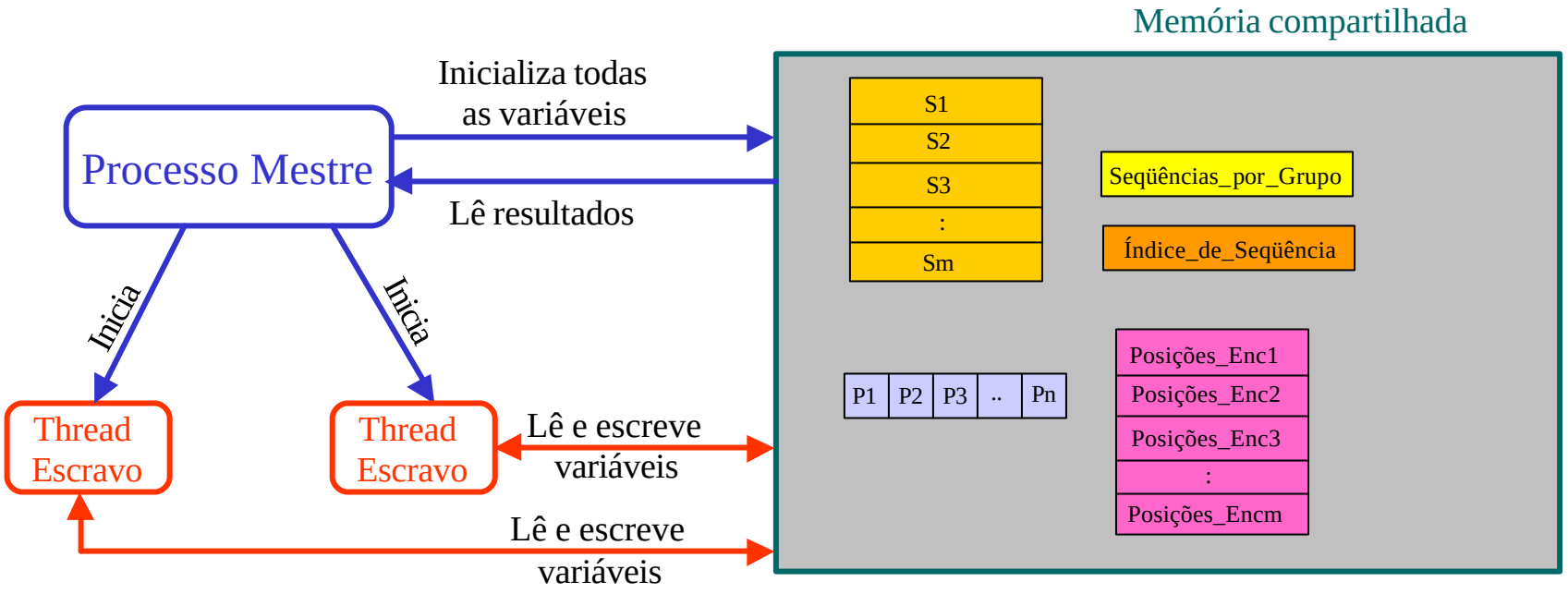
- **Recebe grupo de seqüências $S_i[1..M]$ e padrão P[1..N] do mestre**
- **Para cada seqüência do grupo**
  - Encontra posições do padrão e coloca em resultado
- **Envia resultado para mestre**

Granulosidade média com troca de mensagens - Exemplo

- **P[1..3]=ATG**
- **300 seqüências de tamanho 12**
- **5 processadores**
- **Cada processador procura o padrão em um grupo de 60 seqüências**
- **Análise de desempenho**

Granulosidade média com threads

- **Um processo mestre controla as operações através de variáveis compartilhadas**
- **Processos escravos executam as comparações e devolvem resultados através de variáveis compartilhadas**



Granulosidade média com threads - Processo mestre

- **Lê Padrão P[1..N] e Seqüências S_i[1..M]**
- **Inicializa variáveis de controle**
  - Seqüências_por_Grupo = Número_de_Seqüências/NP
  - Índice_de_Seqüência=1
- **Inicia threads escravas**
- **Espera todas as threads executarem as comparações**
- **Imprime resultados que estão em Posições_Encontradas**

Granulosidade fina com threads - Phread escrava

- **Lê Seqüências_por_Grupo**
- **Espera Índice_de_Seqüência estar desbloqueada**
- **Bloqueia Índice_de_Seqüência**
- **Lê Índice_de_Seqüência**
- **Incrementa Índice_de_Seqüência de Seqüências_por_Grupo**
- **Desbloqueia Índice_de_Seqüência**
- **Executa as comparações**
- **Coloca resultado em Posições_Encontradas**

Granulosidade média com troca de mensagens - Exemplo

- **P[1..3]=ATG**
- **300 seqüências de tamanho 12**
- **5 processadores**
- **Cada processador procura o padrão em um grupo de 60 seqüências**
- **Análise de desempenho**

Bibliografia

- Barry Wilkinson; Michael Allen. *Parallel Programming; Techniques and Applications using networked workstations and parallel computers*. Prentice Hall, 1999.
- Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.
- Ian Foster *Designing and Building Parallel Programs*. MIT Press 1999, disponivel em www.unix.mcs.anl.gov/dbpp
- Lou Baker and Bradley J. Smith. *Parallel Programming.*McGraw-Hill, 1996
- George S. Almasi and Allan M. Gottlieb *Higly Parallel Computing* . Benjamin/Cummings 1989
- Michael J. Quinn. *Parallel Computing: theory and practice*. McGraw-Hill, 1994.