

Modelos para Concorrência

Processamento de Alto Desempenho

Modelos para descrição de concorrência



Programa Interdisciplinar de Pós-Graduação em Computação Aplicada
Universidade do Vale do Rio dos Sinos

Resumo

- Apresentação de modelos
 - Aplicação
 - Máquina
 - Execução
 - Skillicorn e Talia
- Aplicação de modelos

Motivação 1/3

- Existência de modelos bem aceitos para a computação seqüencial:
 - Modelo de von Neumann
 - | Para a arquitetura
 - Máquina de Turing
 - | complexidade
- Com estes modelos fica fácil o desenvolvimento de programas...

Motivação 2/3

- Não existe um modelo universal para computação paralela;
- Existem duas características antagônicas:
 - simplicidade
 - realidade
 - | Modelos simples não aproximam o comportamento de máquinas reais;
 - | Modelo realistas são de difícil uso.

Motivação 3/3

■ Existem vários níveis de concorrência...

Muita fina	Intra-instrução	Proc Superescalares
Fina	Entre-Instruções	Proc Vetoriais
Fina/Média	Blocos	UMA
Média	Procedimentos	UMA/NUMA
Grossa	Processos	SC-NUMA/NORMA
Muito Grossa	Aplicações	Comp em grade ?

(Revisão)

Motivação 3/3

- Existem vários níveis de concorrência...

Muita fina	Intra-instrução	Proc Superescalares
Fina	Entre-Instruções	Proc Vetoriais
Fina/Média	Blocos	UMA
Média	Procedimentos	UMA/NUMA
Grossa	Processos	SC-NUMA/NORMA
Muito Grossa	Aplicações	Comp em grade ?

(Revisão)

Taxonomia: concorrência

- Programa concorrente:
 - Decomposto em atividades independentes
- Programa paralelo
 - Atividades são executadas em processadores independentes, potencialmente ao mesmo tempo
- Programa distribuído
 - As atividades executam em processadores independentes, cada um com sua própria memória

(Revisão)

Taxonomia: concorrência

- Programa concorrente:
 - Decomposto em atividades independentes
- Programa paralelo
 - Atividades são executadas em processadores independentes, potencialmente ao mesmo tempo
- Programa distribuído
 - As atividades executam em processadores independentes, cada um com sua própria memória

(Revisão)

Taxonomia: uma proposta

- Modelos para a aplicação:
 - Representam o paralelismo de um algoritmo.
- Modelos de máquina:
 - Descrevem as características das máquinas.
- Modelos de execução:
 - Detalham a forma de programação.

Criando uma aplicação paralela

- 1) Escolha de um modelo para a máquina;
- 2) Construa uma representação para o algoritmo;
- 3) Escolha de um modelo de execução;
- 4) Procura por um escalonamento;
- 5) Identifique a arquitetura;
- 6) Procure uma ferramenta de programação;
- 7) Implemente.

(Revisão)



Modelos para aplicação

Descrição da Concorrência

■ Paralelismo de dados

- O paralelismo é encontrado na execução de um mesmo cálculo sobre conjuntos distintos de dados

■ Paralelismo de tarefas

- O paralelismo é encontrado na viabilidade de execução simultânea de diferentes trechos de código.

Paralelismo de dados

- Tratamento paralelo dos dados pela execução do mesmo conjunto de operações

- Típico em arquiteturas:

 - SIMD

 - Vetoriais

- Pode ter apoio do compilador

 - Primitivas data-paralelas

 - | ForAll

 - Tipos de dados

 - | Vetoriais

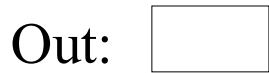
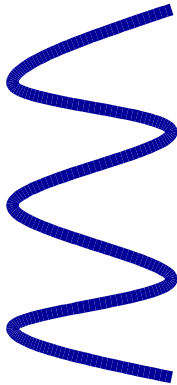
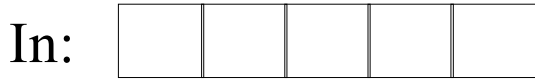
 - | Escalares

- Fortemente Síncrono!!!

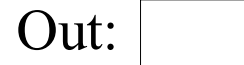
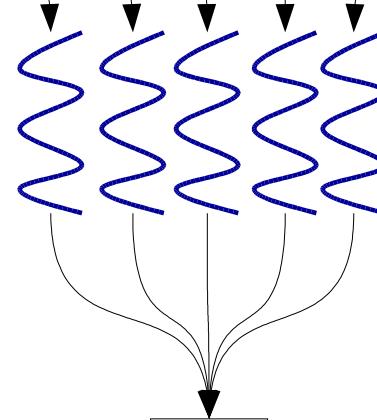
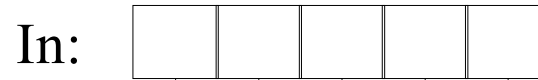
Paralelismo de dados

Exemplo de Esqueleto

■ Split-Compute-Merge



Processamento normal



Paralelismo de dados

Restrições

- Fortemente Síncrono!!!
- Compilador deve ser inteligente
- Distribuição dos dados
 - Complexo, especialmente em estruturas de dados não lineares
 - Distribuir dados é balancear a carga
- As linguagens disponíveis atualmente são antigas
 - Ex: HPF, baseado em Fortran 90, oferecendo unicamente primitivas data-paralelas (ForAll), e fornecendo primitivas para distribuição dos dados pelo programador
- Uso de esqueletos não consolidado nas linguagens
 - Split-Compute-Merge

Paralelismo de tarefas

- Maior complexidade de desenvolvimento

- Multithreading
- Processos comunicantes
- Suportes híbridos (cluster)

- Balanceamento de carga nas atividades

- Custo de manipulação de dados

- Atividades Assíncronas

- Mestre/Escravo
- SPMD
- Tarefas independentes
- Tarefas com dependências

- Princípio moderno de programação

- Oferecido, basicamente, por bibliotecas de programação

Grafo de precedência 1/3

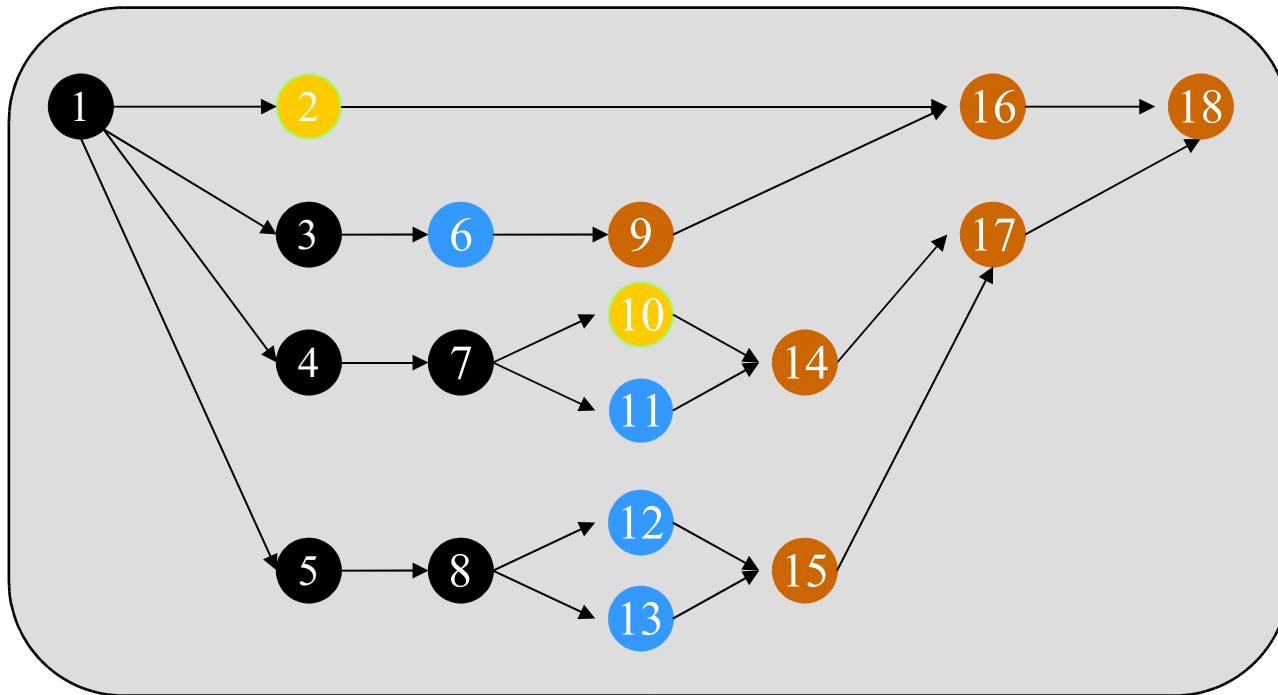
- Um grafo é a forma mais comum:
 - Grafos de fluxo de dados:
 - | Os arcos representam a evolução dos dados
 - | Pode-se transformar em grafo de precedência
 - Grafo de precedência;
 - | Grafo de dependência.
 - Grafo de tarefas

Grafo de precedência 2/3

- O algoritmo é dividido em tarefas:
 - os vértices representam as tarefas;
 - os arcos as relações de dependência.
- Quando o grafo é formado em tempo de execução:
 - Escalonamento dinâmico;
- Quando o grafo é conhecido antes do lançamento:
 - Escalonamento estático.

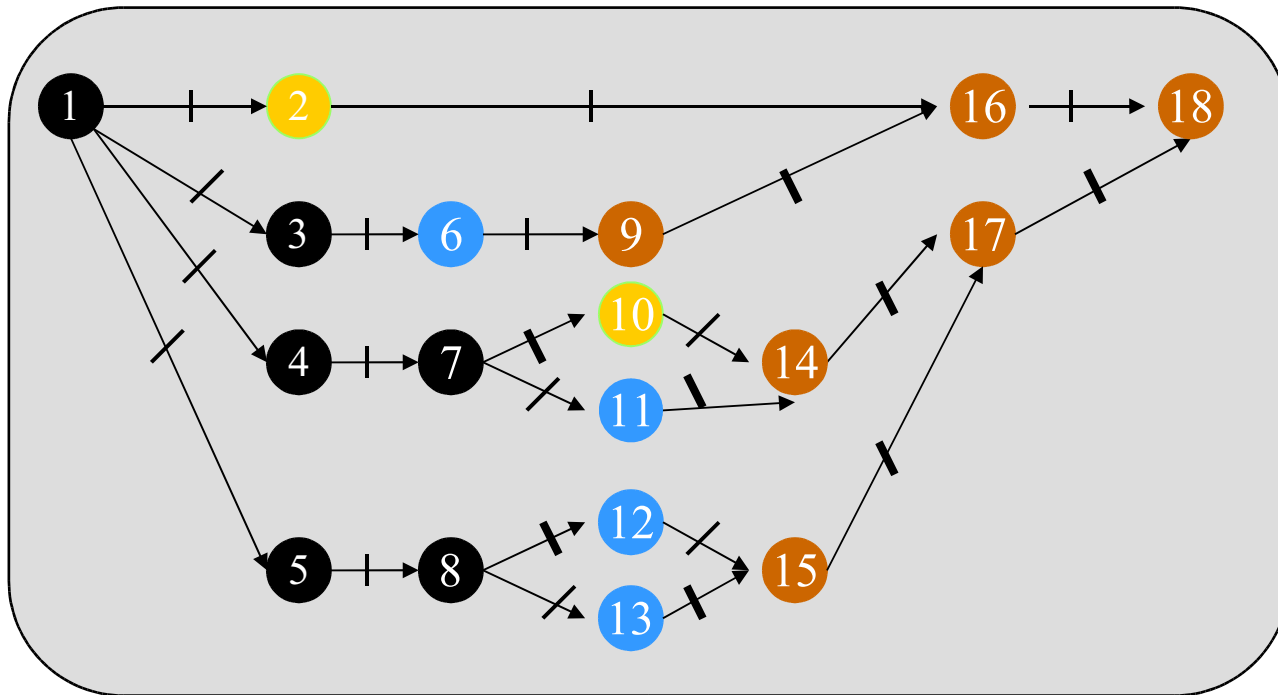
Grafo de precedência 3/3

Representa a ordem de execução entre as tarefas



Grafo de dependência

Além de representar a ordem, representa os dados trocados entre tarefas



Informações no grafo

- Sucessores
 - Quem executa após
- Predecessores
 - Quem deve ser executado antes
- Largura
 - Quantidade máxima de concorrência
- Caminho crítico
 - Maior caminho no grafo
- Granularidade
 - Relação entre execução e comunicação



Modelos para máquinas

Primeira proposta

- Feita por Flynn em 1966:
 - Classifica as máquinas conforme os fluxos:
 - | de instruções
 - | de dados

	Dados		
instr.		Simples	Múltiplo
Simples		SI SD sequencial	SI MD
Múltiplo		MI SD	MI MD

Outras classificações

- Velocidade de comunicação

- Multi-computadores
- Multi-processadores

- Tipo de acesso a memória

- UMA – uniforme
- NUMA – espaço de endereçamento compartilhado vs. distribuído

Memória compartilhada vs. Troca de mensagens



Modelos de execução

Modelos de execução

- Modelo usado no desenvolvimento
 - Predição de custos;
 - Planejamento da comunicação.
- Ligado ao modelo da máquina
- *Bridge models*
 - Modelos abstratos entre um programa real e uma dada arquitetura

Modelos de execução

Abstração do modelo de programação

- Oferecem uma visão da arquitetura e das ferramentas de exploração destas.
- Definem a semântica da execução dos programas.
- Permitem compreender o funcionamento dos algoritmos:
 - ◆ prever desempenho
 - ◆ comparar diferentes algoritmos
- Representações abstratas

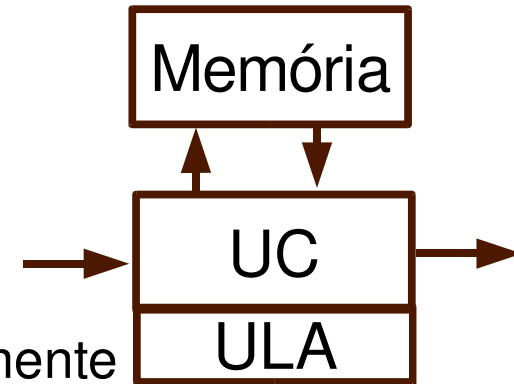
Modelos de execução

Abstração do modelo de programação

Exemplo:

- **Arquitetura Von Neumann**

- Abstração da arquitetura:
 - Memória: células endereçáveis individualmente
 - Instruções:
 - de acesso à memória (LD, STR)
 - de exploração da ULA (lógicas e aritméticas)
 - de E/S (read, write)
- Abstração da execução
 - Programas executam sequencialmente, em tempo finito.
- A memória pode ou não ser limitada.



Principais modelos

- PRAM
- Modelos com atraso de comunicação
- LogP
- BSP / CGM

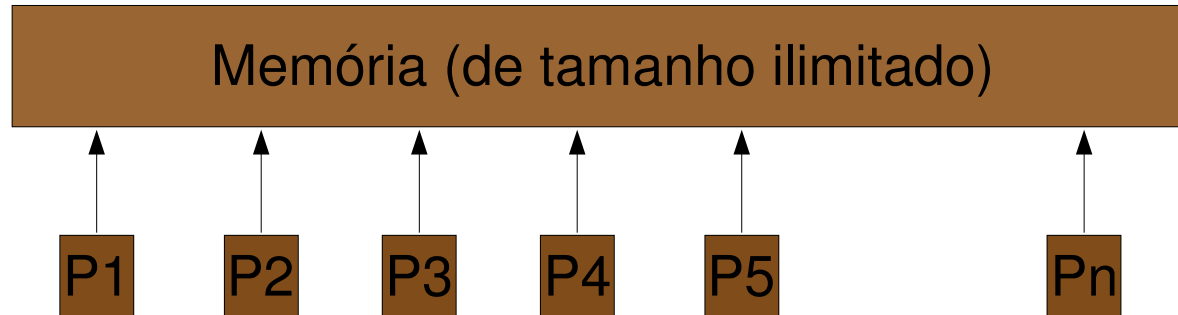
PRAM

- *Parallel Random Access Machine*
- Surgiu em 1978 e é uma referência
- Usado para análise de algoritmos paralelos

PRAM (características)

- Parallel Random Access Machine
- Primeiro modelo proposto
- Abstração de arquitetura:
 - Síncrono
 - número infinito de processadores
 - memória compartilhada e ilimitada
 - Tempo de acesso uniforme à memória
- Abstração de execução:
 - processadores executando de forma síncrona
 - operações lógicas e aritméticas e de acesso a memória
- Modelo não realista
 - manutenção da memória global não escalável

PRAM



Arquitetura:

- n processadores (ilimitado)
- compartilhamento de memória global (de tamanho finito)

Programação:

- Instruções de acesso a ULA
- De acesso a memória (read, write)

PRAM

- Modelos restritos de PRAM: mais realistas
- p-PRAM
 - limita o número de processadores
- Limitações no acesso à memória compartilhada:
 - EREW: Exclusive Read, Exclusive Write
 - Uma célula de memória pode ser acessada por somente um processador
 - CREW: Concurrent Read, Exclusive Write
 - Uma célula de memória pode ser acessada por vários processadores para leitura, mas em caso de escrita, apenas um pode ter acesso
 - CRCW: Concurrent Read, Concurrent Write
 - Vários processadores podem acessar (leitura ou escrita) uma mesma posição
 - Regras para escrita concorrente (Maior, Soma, ...)

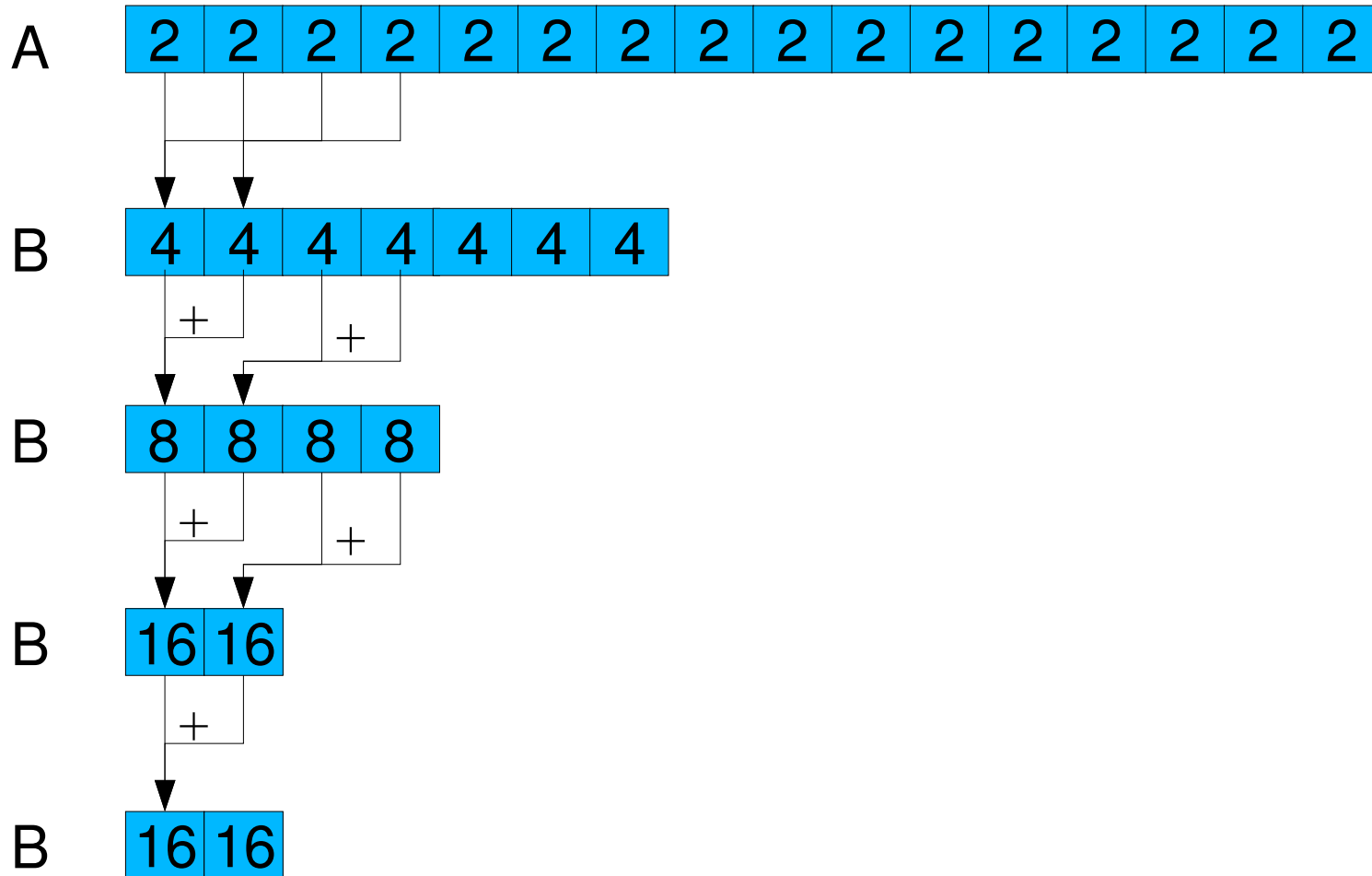
PRAM

- Dado um vetor A , com n elementos, tal que $n = 2^k$, encontrar a soma dos elementos deste vetor.
- Executar o programa em uma arquitetura com n processadores

```
Write( a, B(i) );  
for h = 1 to log n do  
    if i <= n/2 then  
        Read( B(2i-1), x );  
        Read( B(2i), y );  
        z = x + y;  
        Write( z, B(i) );  
    endif  
endfor  
if i = 1 then  
    Write( z, S );  
endif
```

PRAM

Exemplo de Programa



PRAM

Exemplo de Programa

Dados :

n = tamanho do array

p = n = numero de processadores

A[n] = Array de entrada e saida

Soluç o em 4 ciclos:

```
procedure Sort(var A: array 1  n of integer)
  for i in 1  n pardo
    K[i]:=0
  for i in 1  n pardo
    for j in 1  n pardo
      if A[i]<=A[j]
        then K[j]:=K[j]+1
  for i in 1  n pardo
    A[K[i]]:=K[i]
```

PRAM

Exemplo de Programa

i\j	3	1	4	2	5	Entrada
3						
1						
4						
2						
5						

Ciclo 1

PRAM

Exemplo de Programa

$i \backslash j$	3	1	4	2	5	Entrada
3	1	0	1	0	1	
1	1	1	1	1	1	
4	0	0	1	0	1	Processador P_{ij} compara $A[i]$ e $A[j]$:
2	1	0	1	1	1	$P_{ij} := "A[i] \leq A[j]"$
5	0	0	0	0	1	

=====

Ciclo 2

PRAM

Exemplo de Programa

$i \backslash j$	3	1	4	2	5	Entrada
3	1	0	1	0	1	
1	1	1	1	1	1	
4	0	0	1	0	1	
2	1	0	1	1	1	
5	0	0	0	0	1	
$K[i]$	3	1	4	2	5	Soma coluna em $K[i]$

Ciclo 3

PRAM

Exemplo de Programa

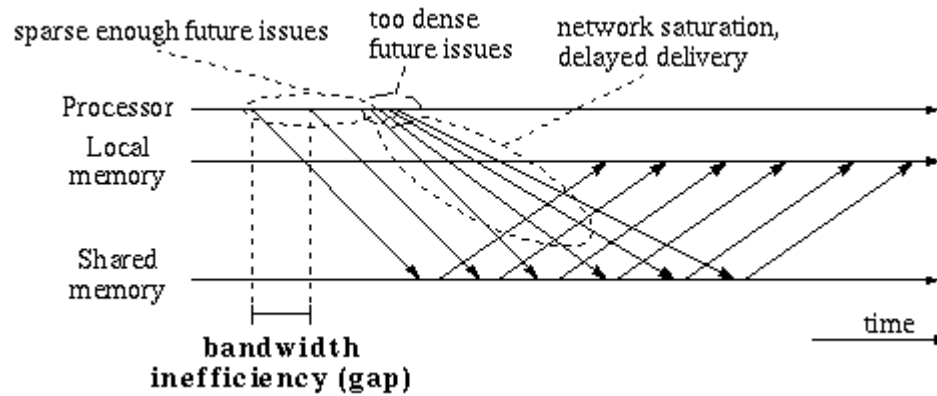
i\j	3	1	4	2	5	Entrada
3	1	0	1	0	1	
1	1	1	1	1	1	
4	0	0	1	0	1	
2	1	0	1	1	1	
5	0	0	0	0	1	
=====						
K[i]	3	1	4	2	5	
A[i]	1	2	3	4	5	$A[K[i]] := K[i]$

Ciclo 4

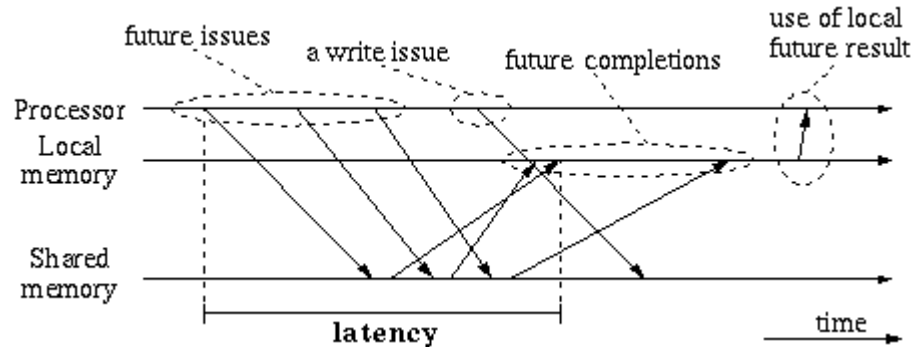
PRAM

Ineficiência

Gap entre duas
comunicações:



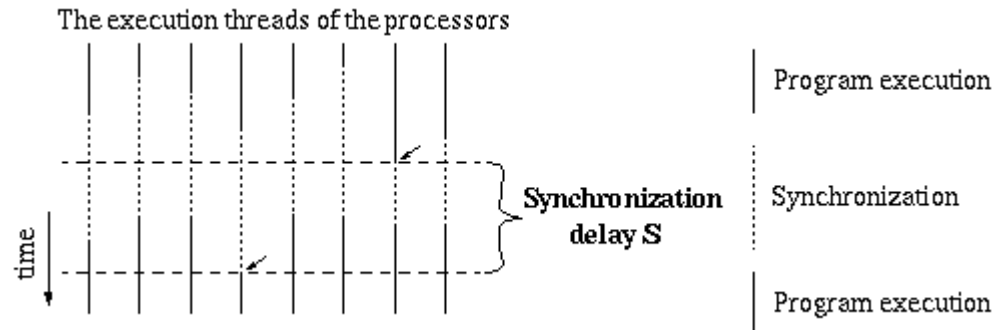
Latência entre
pedido e resposta:



PRAM

Ineficiência

Ciclos de cálculo definidos entre o processador cuja computação foi a mais lenta e o processador que esperou mais tempo pelo dado



Modelos com atraso de comunicação

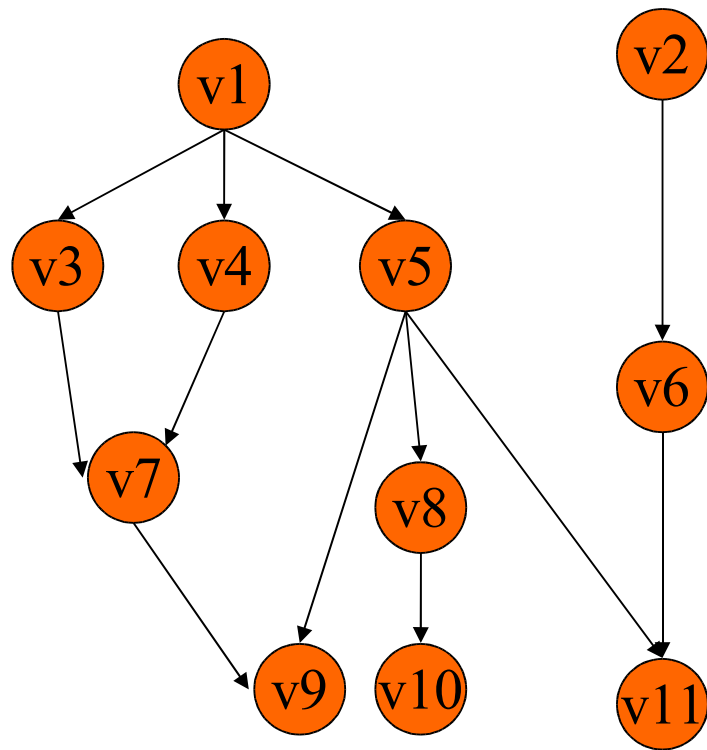
- Fornece um suporte para a comunicação
 - Processadores com memória local
- Vários resultados teóricos
- Considera o essencial das máquinas reais
 - Mas não considera
 - *Overhead* de comunicação, banda passante

Grafo para os exemplos

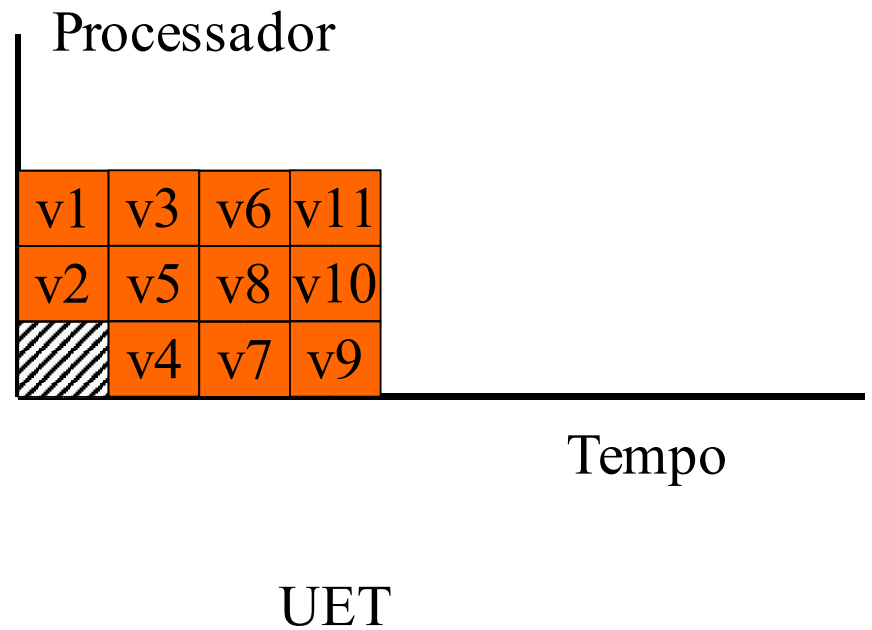
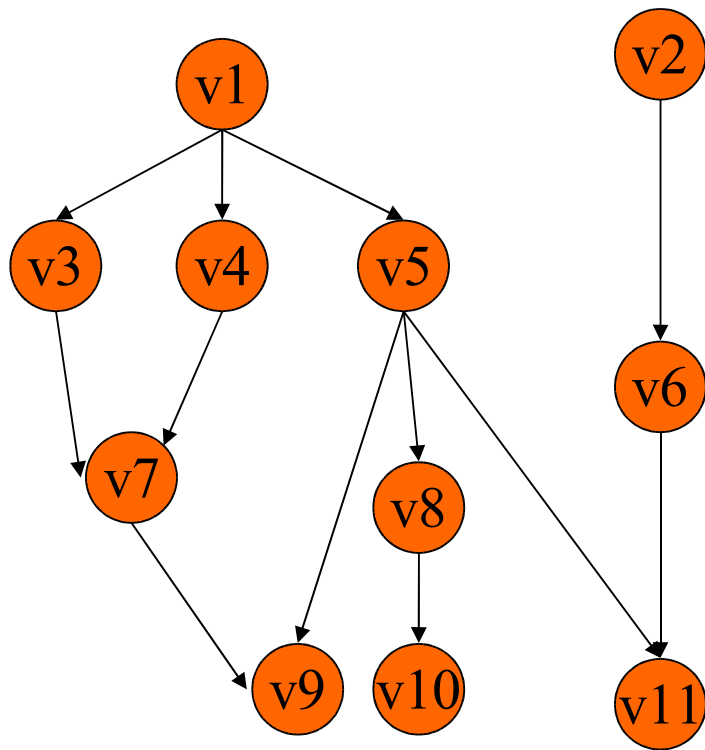
Modelos

- UET (Unit Execution Time)
- UET + UCT (+ Large Communication Time)
- SCT (Short Communication time)
- Modelo mais gerais consideram topologia

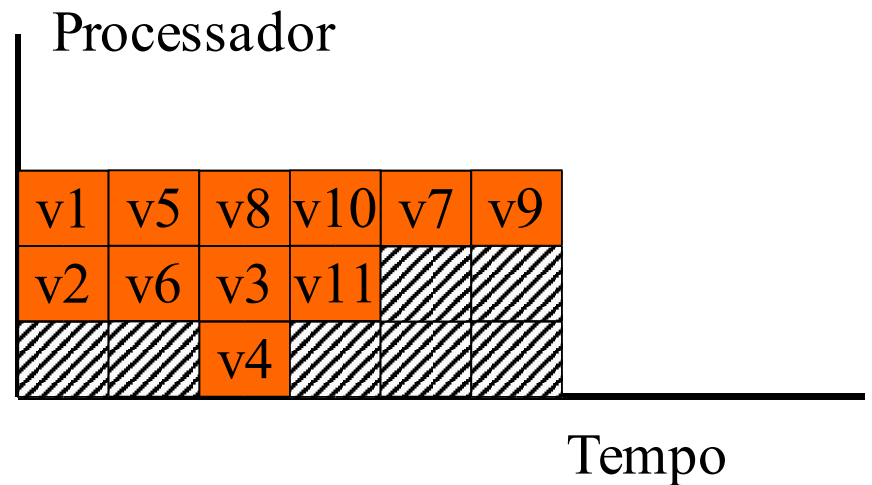
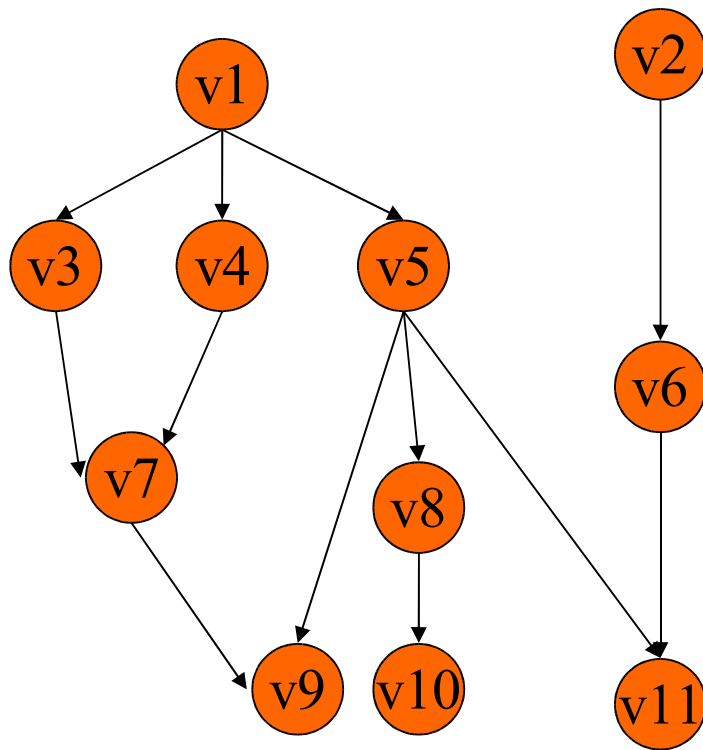
Grafo para os exemplos



Grafo para os exemplos

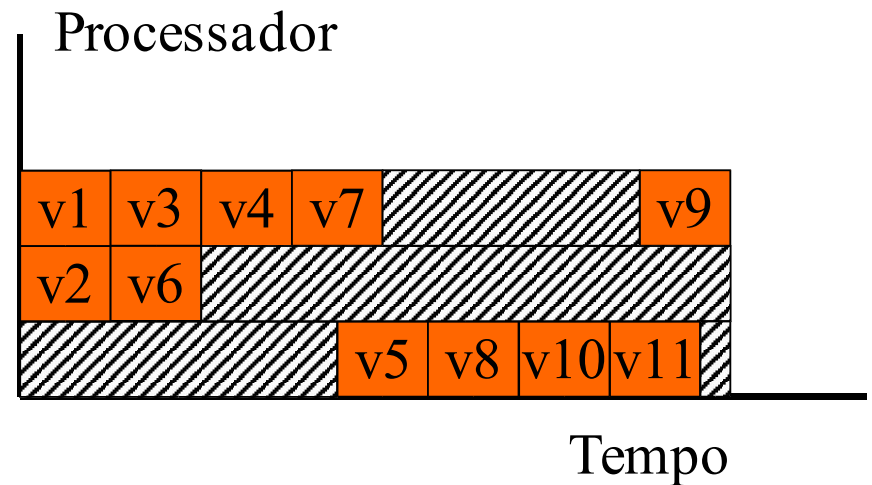
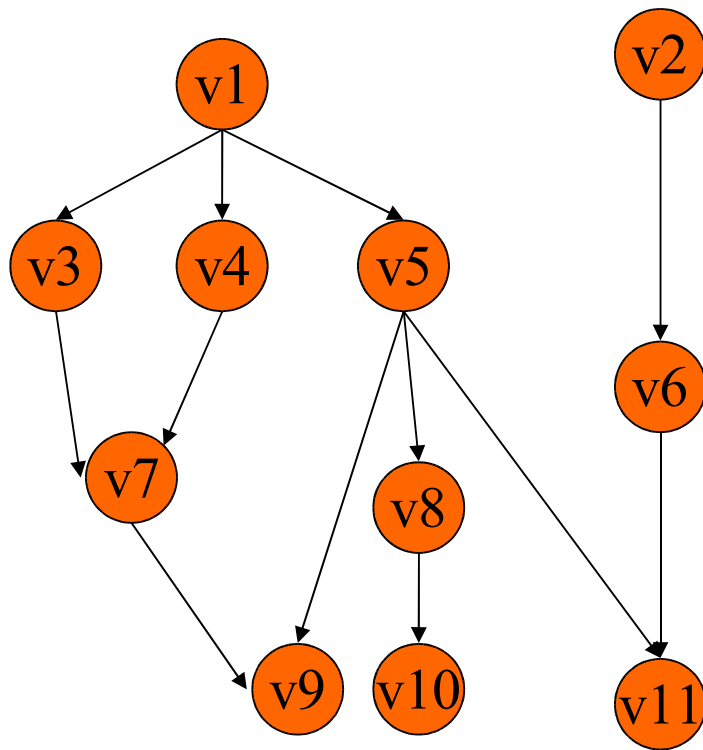


Grafo para os exemplos



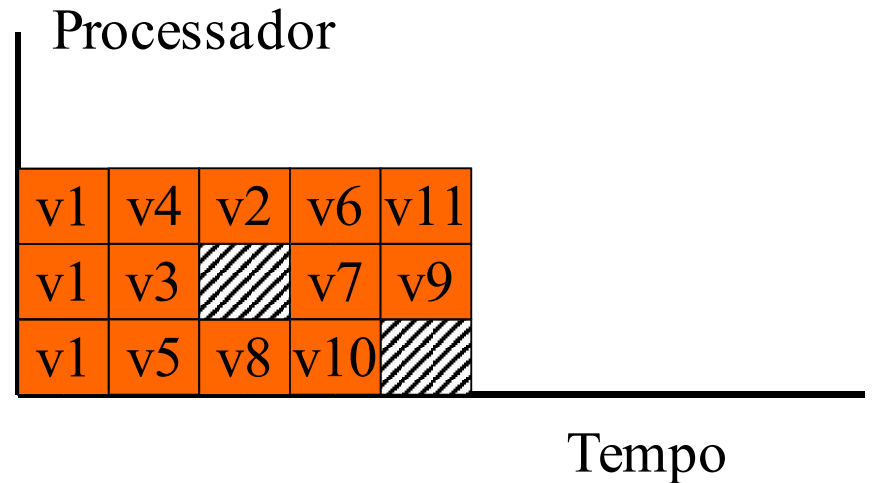
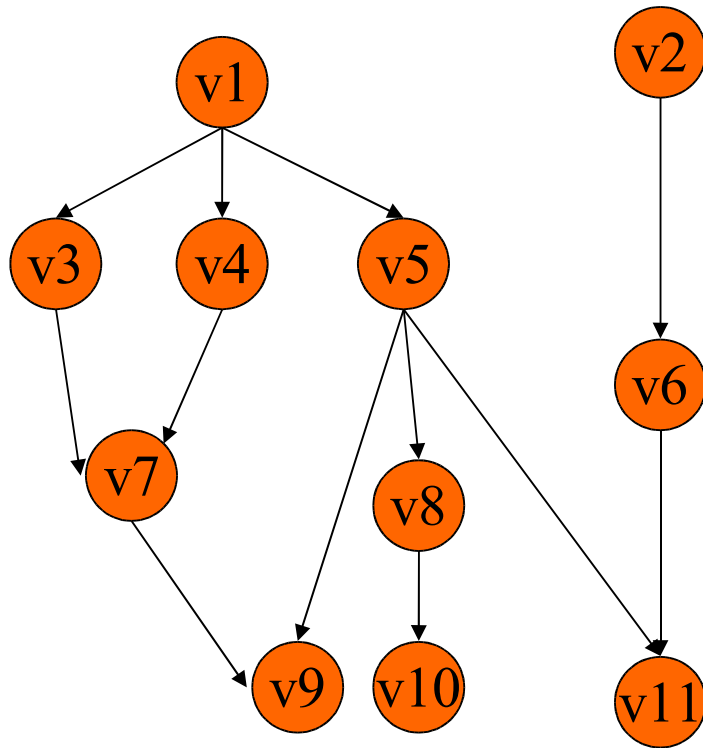
UET LCT

Grafo para os exemplos



UET LCT

Grafo para os exemplos



UET UCT
(com duplicação)

Modelos realísticos

- Duas categorias:
 - Aproximam máquinas de forma genérica
 - | LogP
 - Permitem algoritmos portáteis e eficazes
 - | BSP / CGM

LogP

- Número finito de processadores, custos de comunicação.
- L - Tempo de trânsito da mensagem
 - (pequena)
- o - custo adicional de envio e recepção
- g - intervalo entre envios/recepções seguidos
- P - número de processadores
- Máximo de L/g mensagens na rede

LogP

- Dados foram medidos em muitas máquinas reais;
- Pode-se prever o custo de programas com precisão;
- Existem extensões para mensagens grandes;
- É difícil obter resultados para grafos genéricos.

BSP e CGM

- Modelos semelhantes
 - Origens distintas:
 - BSP – Bulk Synchronous Parallel
 - | Modelo programação
 - CGM – Coarse Grained Multicomputer
 - | Modelo teórico que considera o tamanho do problema
- Separação explícita do cálculo e da comunicação

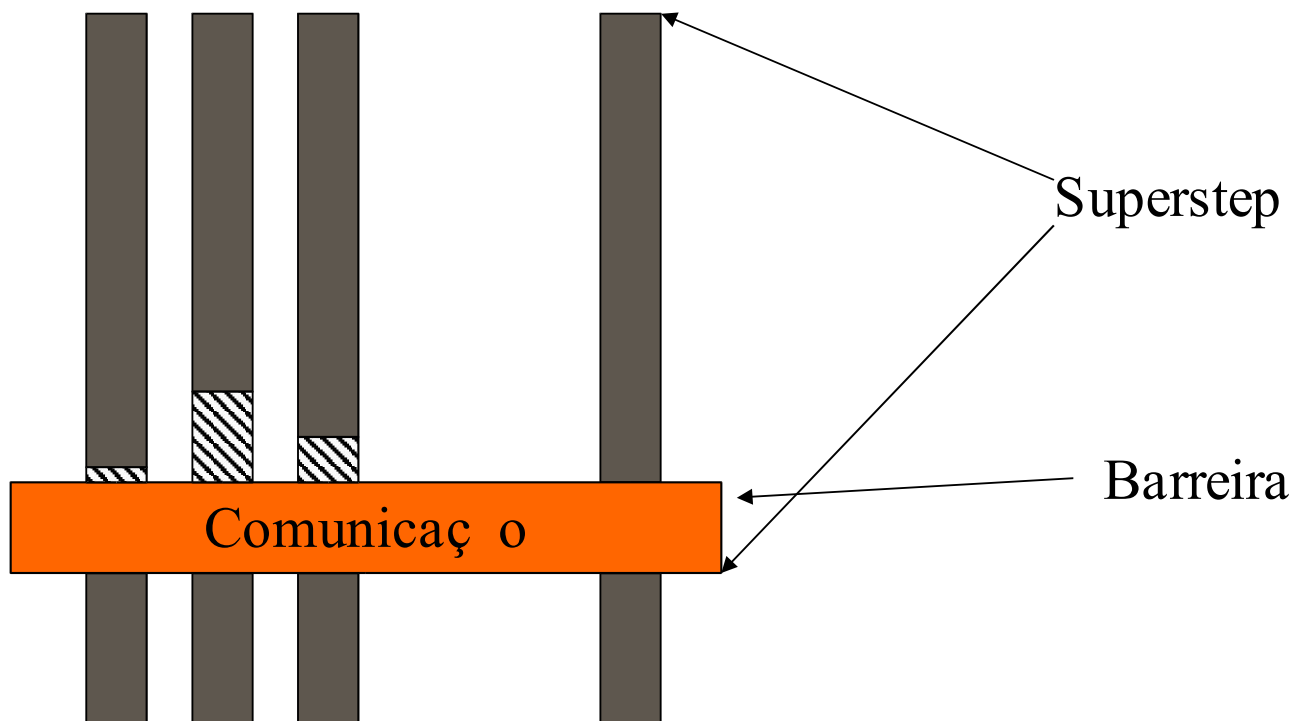
BSP – Princípios

- Super-Etapa
- Sincronização
- Parâmetros:
 - n - número de processadores
 - l - custo de sincronização (custo de uma barreira)
 - g - banda passante (total de operações realizadas por todos processadores em um segundo / número total de pacotes entregues na rede em um segundo)
 - w – trabalho de cada superstep (custo computacional)
 - comunicação: no máximo l/g -relação

BSP – Custo

- Complexidade de um superstep:
 - $\text{Max}\{l, w, gh_s, gh_r\}$
 - Onde:
 - h_s corresponde ao número de pacotes enviados
 - h_r corresponde ao número de pacotes recebidos
- Padrões de comunicação
 - 1-relation – um manda para todos
 - p-relation – todos mandam para um
 - h-relation – todos mandam para todos

BSP – Princípios



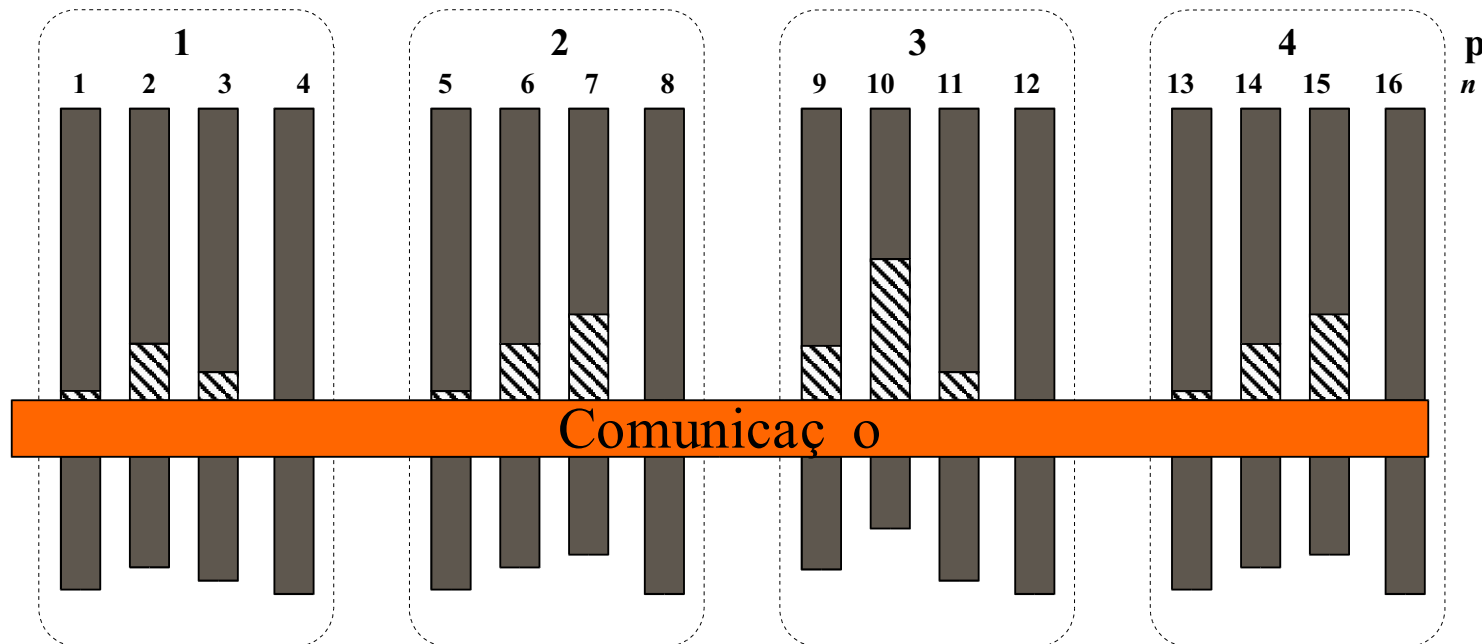
BSP – Exemplo

```
Prefix Sum Algorithm( $i, p, n, A[(i-1)n/p+1 : in/p]; S$ )
1  $T \leftarrow A[(i-1) * n/p + 1] + \dots + A[i * n/p];$ 
2 for  $j \leftarrow i+1$  to  $p$ 
3   do send( $T, p_j$ );
4 bspsync();
5 for  $k \leftarrow 1$  to  $i-1$ 
6   do receive( $p_k, T[k]$ );
7  $ST \leftarrow T[1] + \dots + T[i-1];$ 
8 for  $j \leftarrow (i-1) * n/p + 1$  to  $i * n/p$ 
9   do  $S[j] \leftarrow ST + \sum_{k=1}^j A[k];$ 
```

BSP – Parallel Slackness

Caso o número de processadores virtuais n ser muito maior que o número de processadores reais p , é possível obter sobreposição dos overheads de comunicação por cálculo efetivo

Existe parallel slackness quando $n \gg p$, por exemplo, $n = p \log p$



CGM (coarse grained computers)

- Modelo teórico
- Dois parâmetros:
 - p – número de processadores
 - n – tamanho do problema
- Suposições:
 - $n/p \gg p$
 - Memória local $O(n/p)$
 - n/p -relações
- Objetivo: minimizar a comunicação

Vantagens destes modelos

■ BSP/CGM

- Abstrai totalmente a máquina
- Os algoritmos elaborados em CGM são bons
- Existem várias bibliotecas disponíveis

BSP – BSPLib

void bsp begin (int maxProcs)

Inicializa

int bsp nprocs ()

int bsp pid ()

Número de processadores e Identificador do processador

void bsp pushregister (const void *ident, int size)

Disponibiliza um endereço para acesso remoto (escrita/leitura)

void bsp popregister (const void *ident)

Retira viabilidade de acesso remoto de um endereço

void bsp put (int pid, const void *src, void *dst, int offset, int nbytes)

void bsp get (int pid, void *src, int offset, void *dst, int nBytes)

Escreve/Le em/de um endereço remoto

void bsp sync ()

Barreira para todos os processadores

Classificação de Skillicorn e Talia

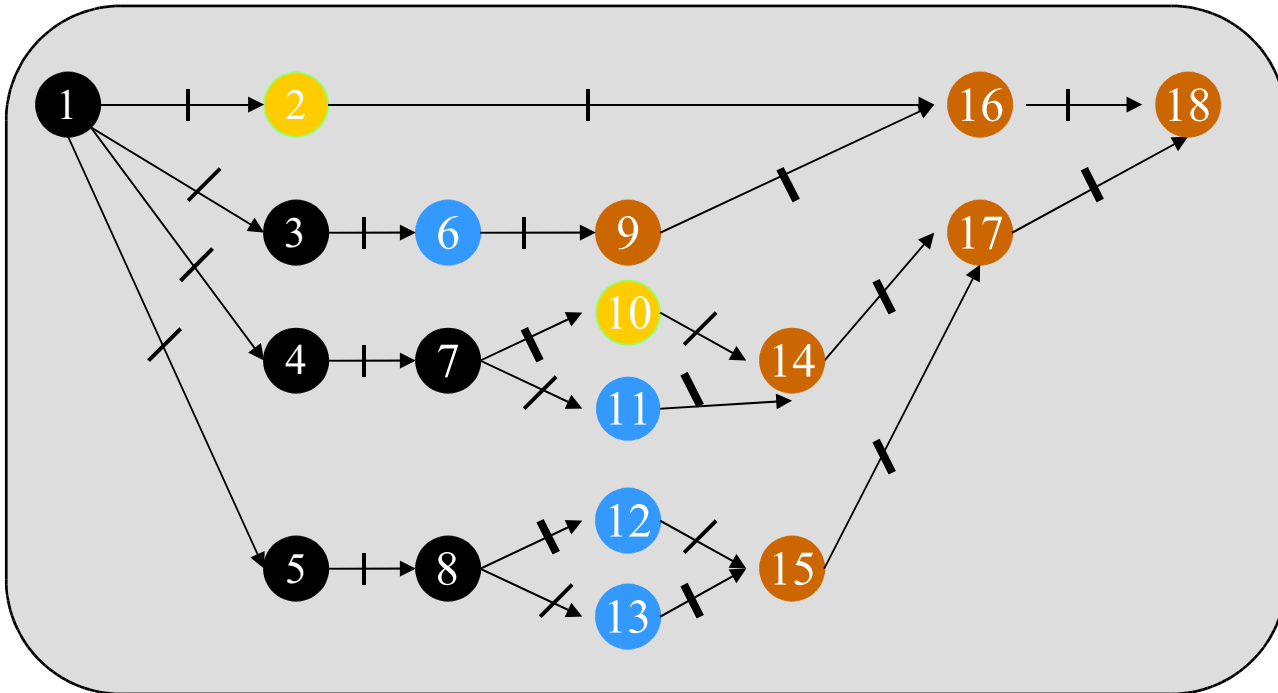
- Paralelismo implícito
- Paralelismo explícito, decomposição implícita
- Decomposição explícita, mapeamento implícito
- Mapeamento explícito, comunicações implícitas
- Comunicação explícita, sincronizações implícitas
- Programador faz tudo

■ Número de Processos vs. Comunicações

- Fixo/dinâmico
- Ilimitado vs. limitado

Nosso modelo de programa concorrente

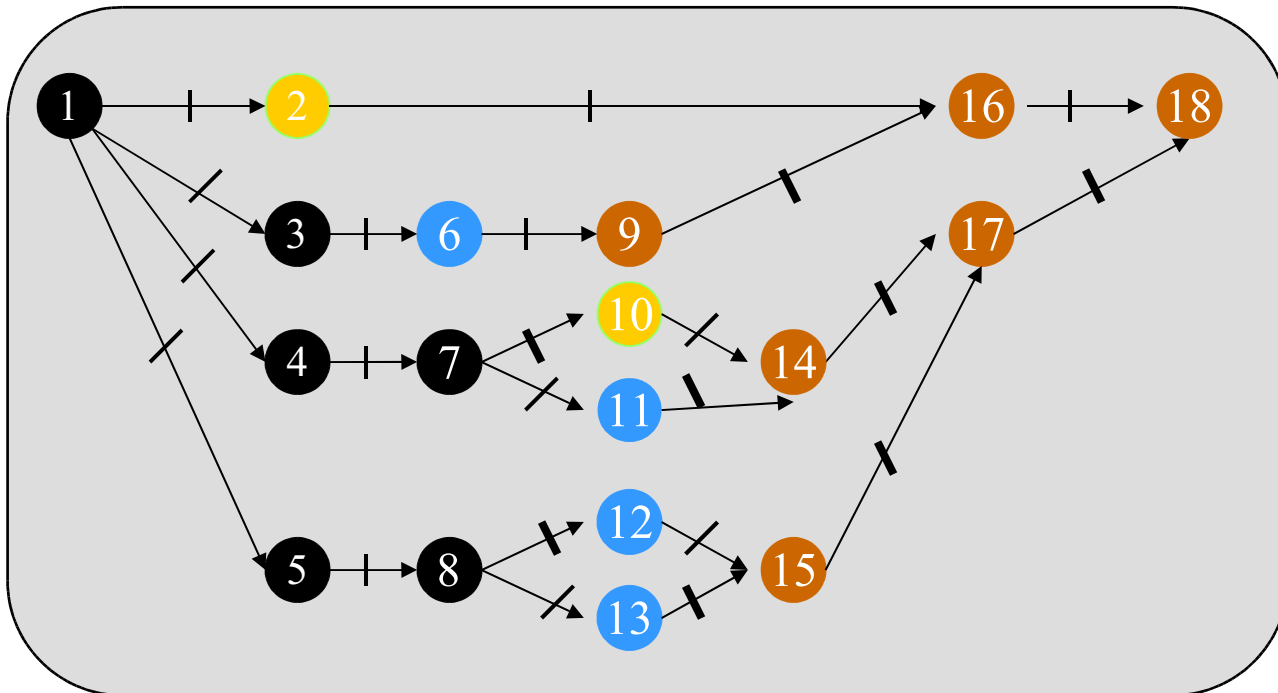
Um grafo G , composto de tarefas T e dados X representa a execução de uma



$$X = \{x_1, x_2, \dots, x_n\}$$
$$T = \{t_1, t_2, \dots, t_n\}$$

Nosso modelo de programa concorrente

Uma função F executa o grafo respeitando a ordem de precedência das tarefas



$t_1 \ll t_2 \ll t_{16} \ll t_{18}$



Ferramentas de Programação

Ferramentas de programação

- Modelos podem ser explorados em arquiteturas reais através de ferramentas de programação
- Compõem “níveis de abstração” dos recursos oferecidos pelas arquiteturas e pelos modelos



Ferramentas de programação

■ Modelo von Neumann

■ Tarefa: instruções

- | Executadas seqüencialmente;
- | Lógicas e aritméticas;
- | Leitura e escrita em memória.

■ Comunicação: memória

- | Informações transmitidas entre instruções.

■ Sincronização: dependência de dados

- | Uma instrução inicia enquanto a anterior não tenha terminado.

Ferramentas de programação

Execução Seqüencial: modelo von Neumann

```
I1: MOV DX, 0
I2: MOV AX, [313]
I3: ADD DX, AX
I4: DEC AX
I5: CMP AX, 0
I6: JNE I3
I7: MOV AX, [313]
I8: CALL RotinaImpressão
```

Efeito da execução de uma instrução:

Escrita em memória

Comunicação

Ferramentas de programação

Execução Seqüencial: modelo von Neumann

```
I1: MOV DX, 0
I2: MOV AX, [313]
I3: ADD DX, AX
I4: DEC AX
I5: CMP AX, 0
I6: JNE I3
I7: MOV AX, [313]
I8: CALL RotinaImpressão
```

**Premissa para executar
uma instrução:**

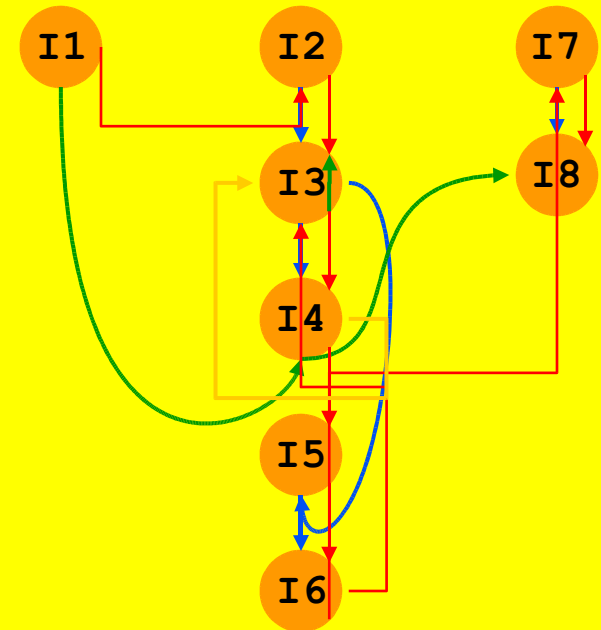
Dados de entrada
em memória

Sincronização

Ferramentas de programação

Execução Seqüencial: modelo von Neumann

```
I1: MOV DX, 0  
I2: MOV AX, [313]  
I3: ADD DX, AX  
I4: DEC AX  
I5: CMP AX, 0  
I6: JNE I3  
I7: MOV AX, [313]  
I8: CALL RotinaImpressão
```



Ferramentas de programação

O que deve ser feito:

Tarefa

Código

Seq-
encial

Parte do
todo

Produz
resultado

Sincronização

Controle

Depen-
dência

Exclusi-
vidade

Ferramentas de programação

O que deve ser feito:



De Modelos para Ferramentas

- Modelos para aplicação:
 - Estratégia mais comum: Grafos

Um grafo representando as relações entre tarefas pode incluir mais ou menos informações sobre a execução:

**Dependência de execução; ou,
Dependência de dados**

De Modelos para Ferramentas

- Modelos para aplicação:
 - No modelo von Neumann

O grafo representa a ordem em que as instruções devem ser executadas, considerando a ordem em que encontram-se definidas no código-fonte

Ordem Lexicográfica

De Modelos para Ferramentas

■ Modelos para aplicação:

■ Casos práticos na concorrência: **Cilk**

■ Linguagem:

`spawn(funcão)`

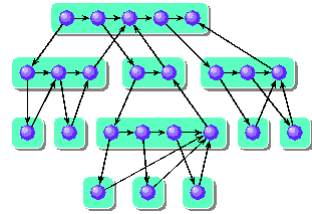
`sync // aguarda o término de todas as filhas`

■ Arquitetura:

n processadores executando

Memória compartilhada

■ Grafo controlando a ordem de execução das tarefas



De Modelos para Ferramentas

■ Modelos para aplicação:

■ Casos práticos na concorrência: **Athapascan-1**

■ Linguagem:

`shared<x>`

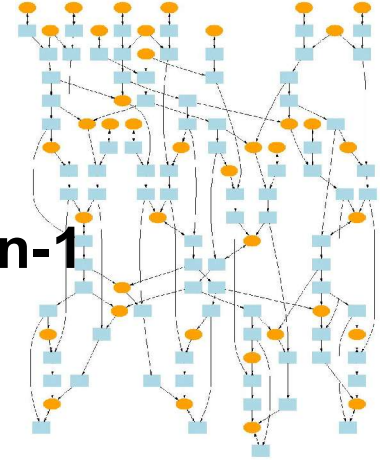
`fork<função>(parâmetros) // modos de acesso`

■ Arquitetura:

n processadores executando

Memória compartilhada

■ Grafo controlando a troca de dados entre tarefas



De Modelos para Ferramentas

■ Modelos para aplicação:

■ Casos práticos na concorrência: **Anahy**

■ Linguagem:

```
id = pthread_create(função, parâmetros)
```

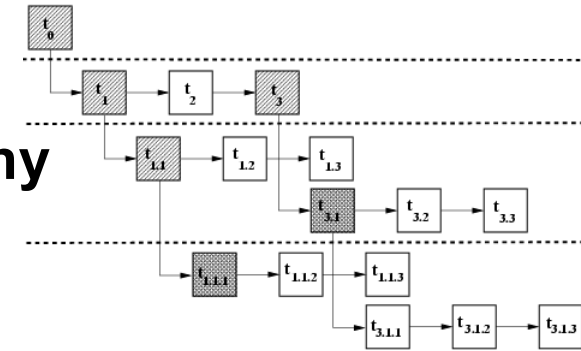
```
pthread_join(id, res) // sincroniza e recupera
```

■ Arquitetura:

n processadores executando

Memória compartilhada

■ Grafo controlando a ordem de execução, informando relações de troca de dados



De Modelos para Ferramentas

■ Modelos para máquinas:

- Casos práticos na concorrência: **multiprogramação leve**
 - Shared Memory Machines
- Implementação do modelo MIMD (Flynn)
- O programa em execução é composto possui **diversos fluxos de execução**, cada um executando sua própria seqüência de instruções e acessando **um conjunto de dados distintos** em uma memória compartilhada.
- Exemplo:
 - Threads POSIX, OpenMP, Cilk, Anahy

De Modelos para Ferramentas

■ Modelos para máquinas:

■ Casos práticos na concorrência: **SPMD**

■ Single Program, Multiple Data

■ Releitura do modelo SIMD (Flynn)

■ O programa em execução é composto de um único fluxo de execução, sobre o qual é executado em várias instâncias a mesma seqüência de instruções, sobre conjuntos disjuntos de dados.

■ Exemplo:

■ MPI (Message Passing Interface), PVM (Parallel Virtual Machine), aplicações escritas com *sockets*

De Modelos para Ferramentas

■ Modelos para máquinas:

■ Casos práticos na concorrência: **MPMD**

- Multiple Program, Multiple Data

■ Releitura do modelo MIMD (Flynn)

■ O programa em execução é composto de diversos fluxos de execução, sobre o qual é executado em várias instâncias a mesma seqüência de instruções, sobre conjuntos disjuntos de dados.

■ Exemplo:

- MPI-2 (Message Passing Interface), PVM (Parallel Virtual Machine), aplicações escritas com *sockets*

De Modelos para Ferramentas

- Modelos de execução

PRAM, BSP

- Representam a execução de programas em modelos de arquitetura;
- Análise do comportamento;
- Permitem programação

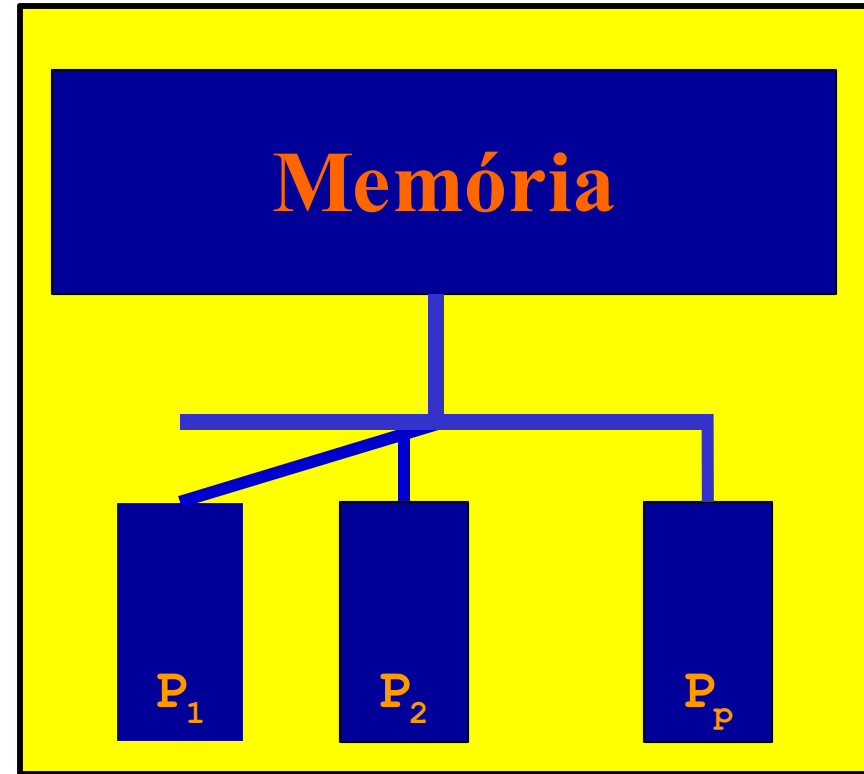
Exemplos: FORK, BSPLib

- Nossos exemplos: processos leves (threads POSIX) e MPI

Processos Leves

Arquitetura SMP

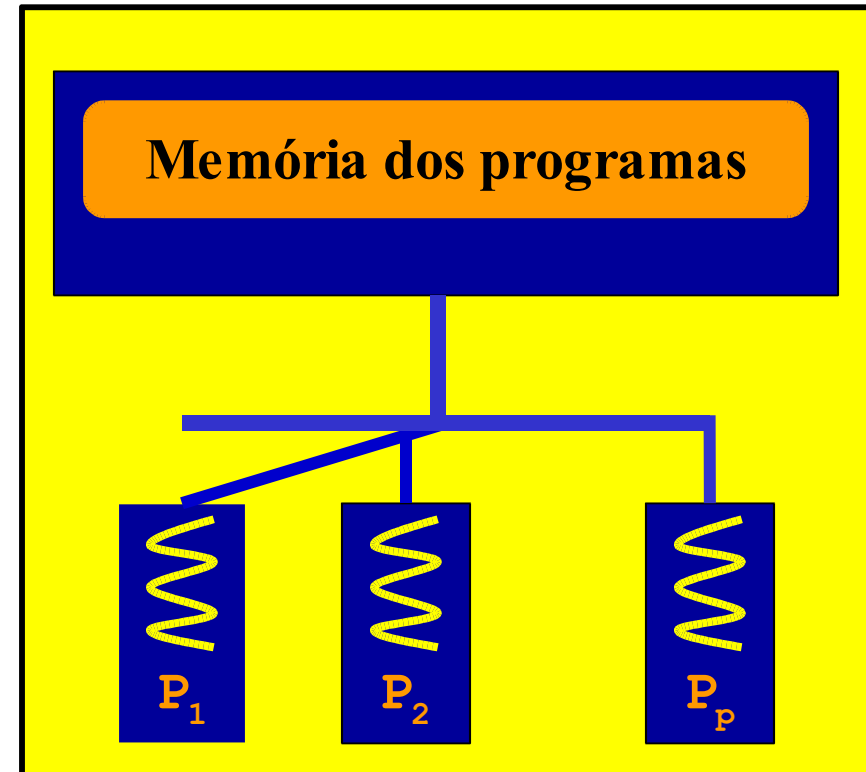
- p processadores;
- Memória compartilhada.



Processos Leves

Arquitetura SMP

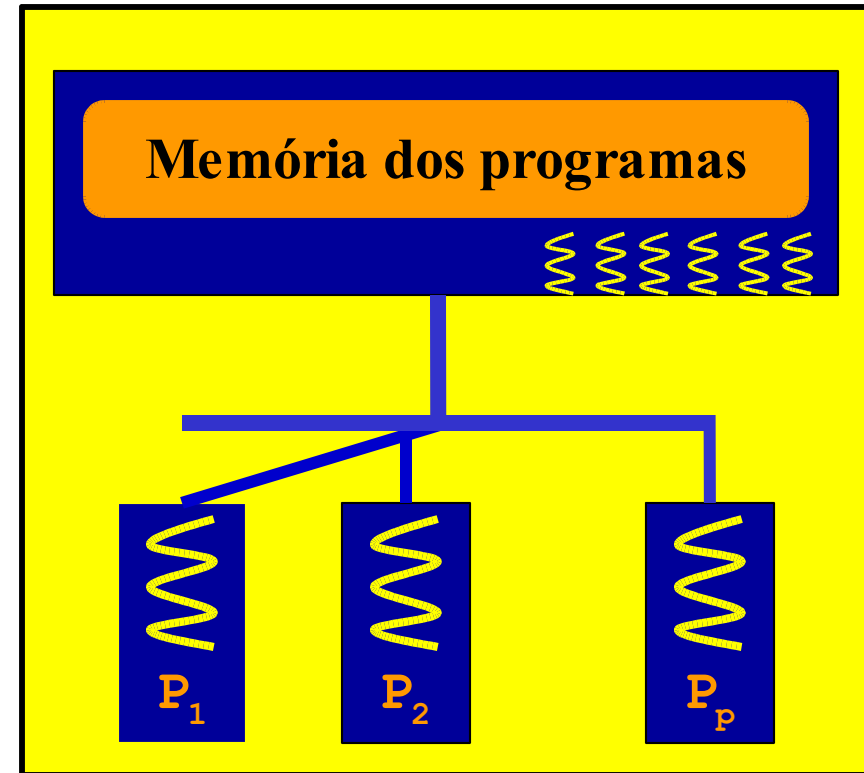
- p processadores.
- Memória compartilhada.
- Programa com múltiplos fluxos de execução;
- Execução independente sobre cada processador;
- A memória armazena dados do programa.



Processos Leves

Arquitetura SMP

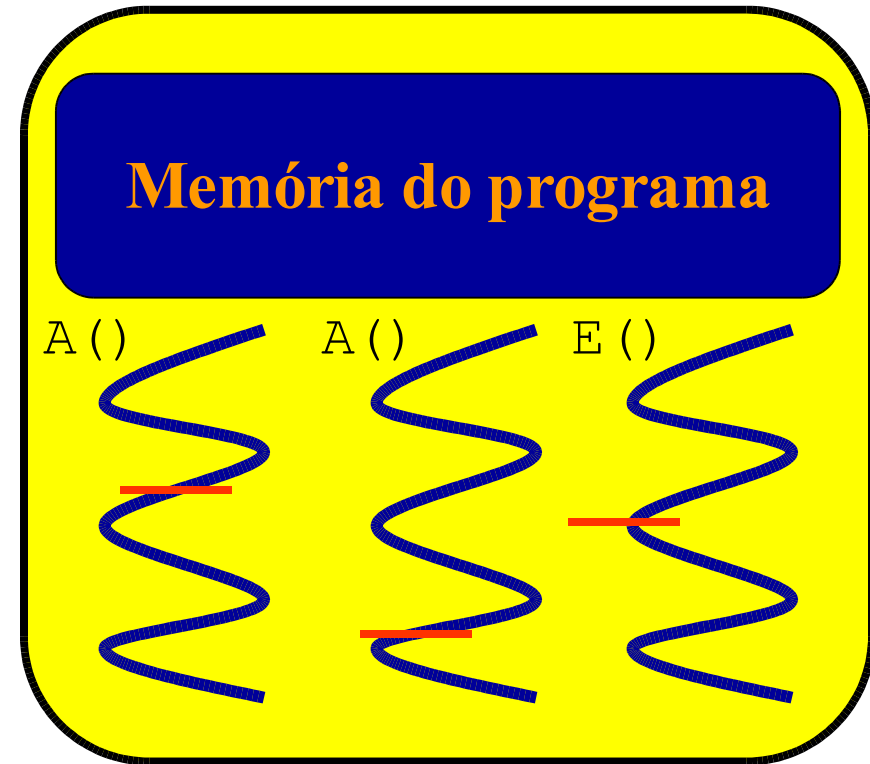
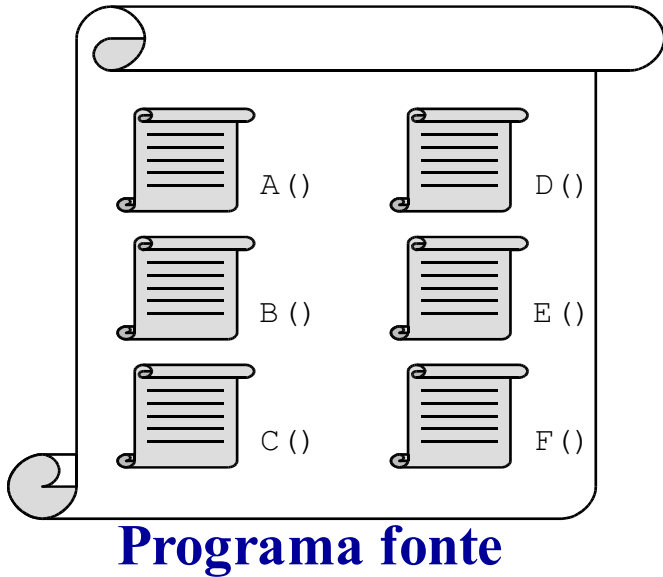
- p processadores;
- Memória compartilhada.
- Programa com múltiplos fluxos de execução;
- Execução independente sobre cada processador;
- A memória armazena dados do programa.



- Número de fluxos \gg número de processadores.

Processos Leves

Visões: programa vs. Execução



Processo com múltiplas *threads*

Um programa PRAM com threads

```
#define TAM 100
#define NTH 10
int vet[TAM]; // suponha inicializado
int somatotal = 0;
pthread mutex m;

void *func( void* in ) {
    int n = (int) in; // Quem sou ?

    for( i = (TAM/NTH)*i ; i < (TAM/NTH)/(i+1) ; i++)
        somalocal += vet[i]; // Realiza as somas parciais

    m lock(); // Escreve na memória com
    somatotal += somalocal; // a semântica de uma operação
    m unlock(); // de escrita concorrente

    return NULL;
}
```

Processos Comunicantes

Arquitetura com memória distribuída SPMD ou MPMD

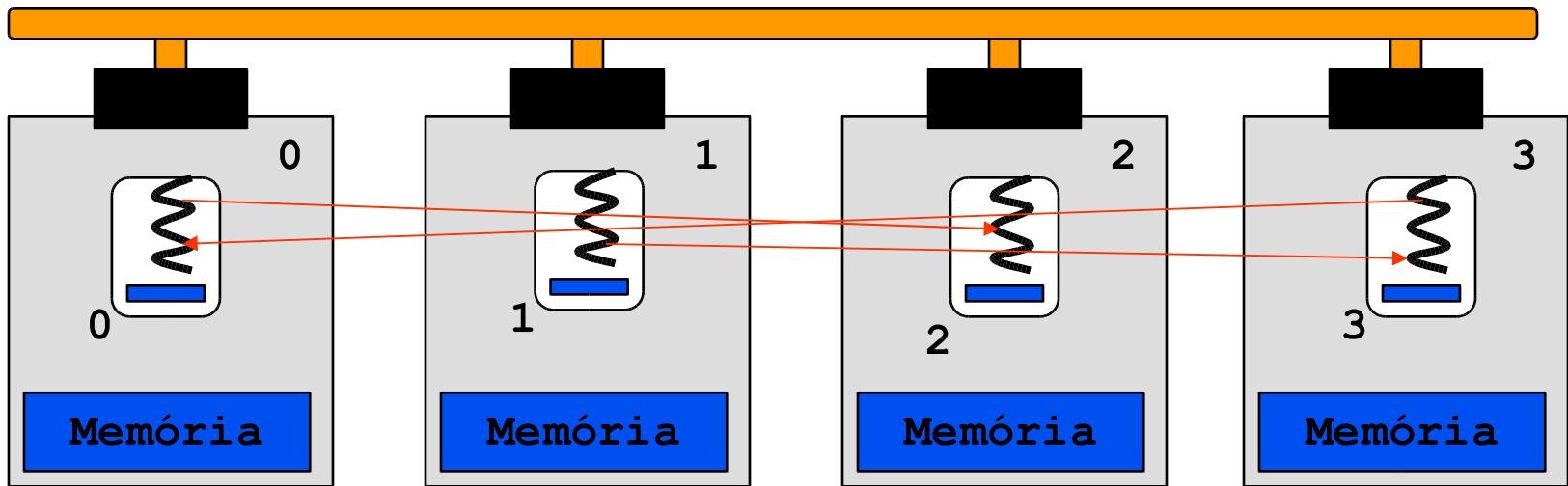
- p processadores;
- Memória local ao processador.

- Programa com múltiplos fluxos de execução;
- Execução independente sobre cada processador;
- A memória local a cada nodo armazena dados do programa.

- Trocas de informação entre os processadores através de mensagens.

Processos Comunicantes

Arquitetura com memória distribuída SPMD ou MPMD



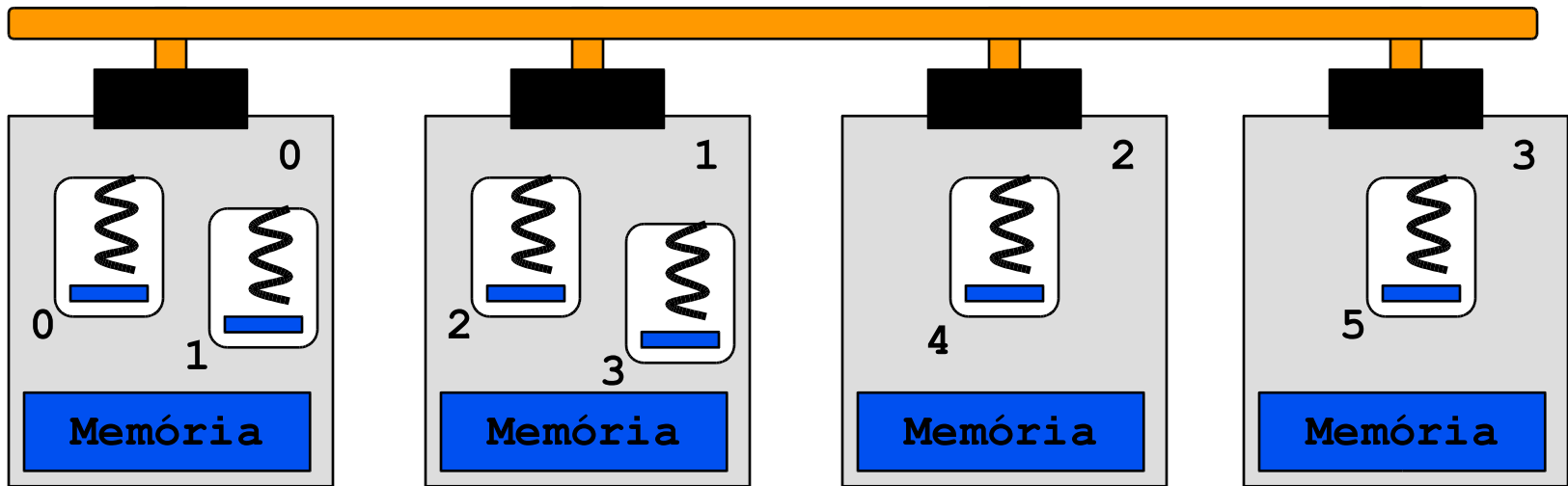
Processador do nodo: suporte para cálculo

Memória do nodo: armazenamento do código e dados do processo

Rede: meio para compartilhamento de dados

Processos Comunicantes

Arquitetura com memória distribuída SPMD ou MPMD



Um nodo pode suportar a execução de mais de um processo

Processos Comunicantes

- Toda a troca de dados é explícita.
- Toda a sincronização é explícita.
- Necessidade de endereçamento e conhecer os participantes em uma troca de dados.

Bastante popular

- Primitivas básicas: send/receive.
- Outras primitivas para comunicação em grupo.

Um programa BSP com MPI

```
#define TAM 100
#define NTH 10
int vet[TAM]; // suponha inicializado no nó 0 (considerado raiz)
int main( int argc, char** argv ) {
    int myrank, size;
    int i, somalocal, somatotal;
    MPI init( &argc, &argv );
    MPI Comm rank( MPI COMM WORLD, &myrank ); // Quem sou ?
    MPI Comm size( MPI COMM WORLD, &size ); // Quantos somos ?
    MPI Bcast( vet, MPI INT, TAM, 0, MPI COMM WORLD );

    for( i = (TAM/NTH)*myrank ; i < (TAM/NTH)*myrank+1 ; i++)
        somalocal += vet[i]; // Realiza as somas parciais

    if( myrank == 0 ) somatotal = somalocal;

    MPI Reduce( &somaparcial, &somatotal, MPI INT, 0, MPI COMM WORLD );
    return 0;
}
```

Conclusão

- Modelos são utilizados para auxiliar no desenvolvimento de aplicações concorrentes, oferecendo uma abstração dos recursos a serem explorados.
- A quantidade de modelos pode ser parcialmente explicada pela diversidade de arquiteturas paralelas disponíveis.
- Embora não haja um modelo universalmente aceito para a programação concorrente, os que existem podem ser utilizados na prática.