Modelos de Programação Paralela

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
UFC-Quixadá

Modelos Para Sistemas Paralelos

Paralelização de Programas

Níveis de Paralelismo

Distribuição de Dados para *Arrays*

Troca de Informação

Produto Vetor Matriz Paralelo

Sistema Computacional

Sistema Computacional é o conjunto de todos os componentes de *software* e *hardware* que são disponibilizados para o programador, formando a visão que o desenvolvedor tem da máquina.

- Sistema operacional;
- linguagem de programação;
- compilador;
- bibliotecas.

Diferentes sistemas computacionais podem levar a diferentes programas paralelos para o mesmo algoritmo. Precisamos de visões abstratas dos sistemas para podermos estudar os princípios gerais do projeto e análise de programas paralelos.

Modelos para Sistemas Paralelos

- Modelo de máquina: abstração básica, descrição de *hardware* e sistema operacional, linguagem *assembly*;
- modelo arquitetural: rede de interconexão, organização de memória, SIMD, MIMD, etc;
- modelo computacional: funções de custo para análise de execução em uma arquitetura, RAM (*random access machine*), análise assintótica, etc;
- modelo de programação: especifica a visão do programador sobre um computador paralelo ao definir como o desenvolvedor pode codificar um algoritmo.

Características de Um Modelo de Programação Paralelo

- Nível de paralelismo: instrução, declarações, procedural ou laços;
- paralelismo implícito ou explícito;
- o modo de execução das unidades paralelas: SIMD, SPMD, etc;
- padrão de comunicação entre as unidades paralelas;
- mecanismos de sincronização.

Uma visão muito importante do modelo é a organização do espaço de endereçamento. Como já vimos, ele pode ser distribuído, compartilhado ou híbrido.

Programa Paralelo

Um programa paralelo especifica computações. Uma computação é uma abstração que pode representar:

- Uma sequência de instruções;
- uma sequecia de declarações, cada uma capturando diversas instruções;
- uma função, método ou procedimento, com várias declarações;
- o corpo de um laço (*paralell loops*).

De acordo com o modelo, as computações pode ser divididas em tarefas (*tasks*). As tarefas são mapeadas para os processos (unidades de execução) e conseguem executar em paralelo.

Paralelização de Programas

Indenpendente do modelo de programação paralelo, em geral partimos de um programa sequencial, identificamos as computações e suas dependências de controle e dados.

- Decomposição de computações;
- atribuição de tarefas para processos e *threads*;
- mapeamento de processos ou *threads* para unidades de processamento;

Pense na computação como um nível mais alto de abstração, enquanto a tarefa já é uma descrição mais próxima do código. Em muitos problemas, ambos os termos acabam por identificar a mesma coisa.

Decomposições de Computações

Computações $\rightarrow$ tarefas.

- Computações definem tarefas, com suas dependêncais determinadas;
- tarefas podem ser a nível de instrução, paralelismo de dados ou paralelismo funcional;
- a tarefa é uma sequência de computações executadas por um única unidade de processamento;
- decomposição estática ou dinâmica;

Granularidade: tempo de execução de uma tarefa.

- Granularidade fina: muitas tarefas curtas;
- granularidade espessa: poucas tarefas longas.

Compromisso entre extremos para minimizar a sobrecarga do escalonamento e mapeamento.

Atribuição de Tarefas

Tarefas $\rightarrow$ processos, *threads*.

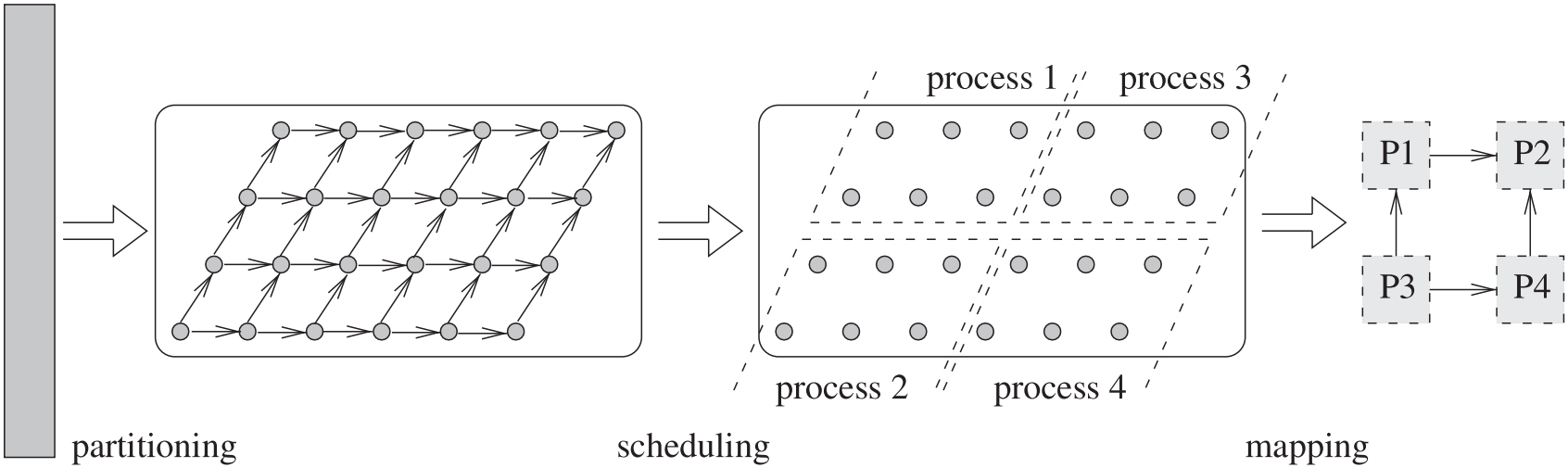
- Um processo ou *thread* representam um fluxo de execução, podendo executar tarefas diferentes uma após outra;
- o nº de processos não precisa ser igual ao número de processadores, mas costuma ser;
- balanceamento de carga: cada processo/*thread* deve receber aproximadamente o mesmo número de computações;
- a quantidade acessos de memória e operações de comunicação deve ser considerada.

Escalonamento (*scheduling*): atribuição de tarefas a processos.

Mapeamento de Processos ou *Threads*

Processos, *threads* $\rightarrow$ processadores, núcleos.

- Novamente, garantir o balanceamento de carga;
- evitar processadores ociosos;
- manter a comunicação entre processadores a menor possível;
- seja por mensagens ou variáveis compartilhadas, a comunicação causa bloqueios.



Copyright © 2010 Springer-Verlag GmbH

Desafios para o algoritmo de escalonamento:

- Restrições de precedência: dependência entre as tarefas;
- restrições de capacidade: quantidade de tarefas maior do que as unidades de execução.

Para cada tarefa, definir sua unidade de execução e quando iniciará, de forma a respeitar suas restrições. O problema geral é NP-completo e não é permitida a migração de tarefas.

Níveis de Paralelismo

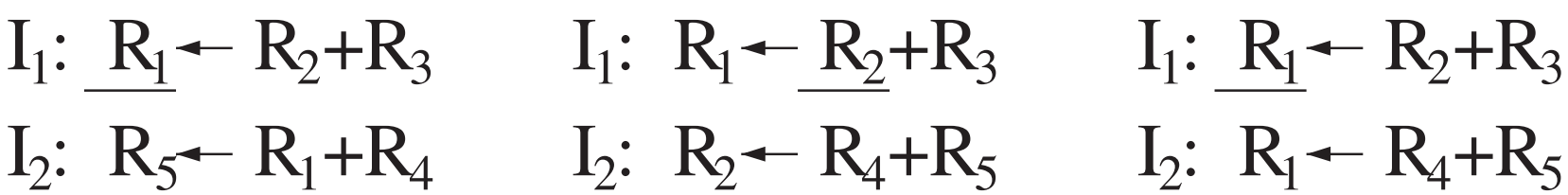
Nível crescente de granularidade (fina $\rightarrow$ espessa):

- Instrução;
- declaração;
- laço;
- funcional.

Paralelismo ao Nível de Instrução

Dependências de dados entre instruções I_1 e I_2:

- Dependência de fluxo: I_1 calcula um valor que I_2 usa como operando;
- anti-dependência: I_1 usa um operando que I_2 usa como saída;
- dependência de saída: tanto I_1 e I_2 usam mesmo registrador como saída.



Copyright © 2010 Springer-Verlag GmbH

Em qualquer caso, I_1 e I_2 não podem ser executadas em paralelo.

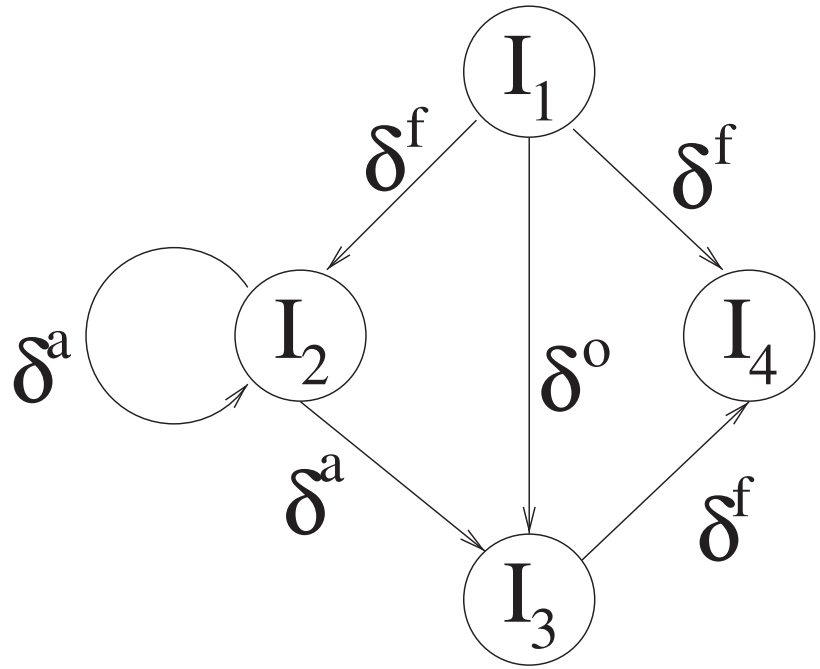
Paralelismo ao Nível de Instrução

I_1: $R_1 \leftarrow A$

I_2: $R_2 \leftarrow R_2 + R_1$

I_3: $R_1 \leftarrow R_3$

I_4: $B \leftarrow R_1$



Copyright © 2010 Springer-Verlag GmbH

Em geral, esse tipo de paralelismo é suportado pelo *hardware* e explorado pelo compilador, sendo implícito ao desenvolvidor final.

Paralelismo de Dados

Operações aplicadas a elementos diferentes de uma estrutura de dados.

- as operações são independentes;
- os elementos da estrutura de dados são distribuídos igualmente entre os processadores;
- algumas linguagens fornecem suporte nativo.

Em Fortran 90:

```
a(1:n) = b(0:n-1) + c(1:n)
```

é o mesmo que:

```
for (i = 1:n)
   a(i) = b(i-1) + c(i)
endfor.
```

Paralelismo de Dados

Na maioria das linguagens que fornecem suporte nativo, todos os acessos a *arrays* e operações no lado direito são realizados antes da avaliação do lado esquerdo.

```
a(1:n) = a(0:n-1) + a(2:n+1)
```

não é o mesmo que

```
for (i=1:n)
   a(i) = a(i-1) + a(i+1)
endfor.
```

Mesmo sem suporte nativo da linguagem, paralelismo de dados pode ser fornecido por SPMD.

Paralelismo de Dados

S*ingle* **P***rogram* **M***ultiple* **D***ata*: cada processador pode executar uma parte diferente do programa, dependendo do seu identificador.

```
local_size = size / p;
local_lower = me *  local_size;
local_upper = (me + 1) * local_size - 1;
local_sum = 0.0;

for (i = local_lower; i <= local_upper; i++)
   local_sum += x[i] * y[i];

Reduce(&local_sum, &global_sum, 0, SUM);
```

Paralelismo de Laço

Laço paralelo: se não há dependências entre as iteraçõe e podem ser executas em ordem arbitrária, elas também podem ser executadas em paralelo em diferentes processadores. Padrões:

- *forall*
- *dopar*

Paralelismo de Laço

Laço *forall*.

```
forall (i = 1:n)
   a(i) = a(i-1) + a(i+1)
endforall
```

é equivalente a

```
a(1:n) = a(0:n-1) + a(2:n+1)
```

Se o laço contém várias atribuições, elas são executadas uma após outra, de tal forma que uma atribuição não é iniciada antes que a anterior seja finalizada.

Paralelismo de Laço

No laço *dopar*, qualquer atribuição, declaração ou até mesmo outros laços são permitidos.

- Atualizações de variáveis em uma iteração não são visíveis pelas outras;
- após o fim do laço, as atualizações são combinadas para formar um novo estado global final;
- se duas ou mais iterações atualizam uma mesma variável, o comportamento é *não determinístico*.

Paralelismo de Laço

```
for (i=1:4)
 a(i) = a(i) + 1
 b(i) = a(i-1) + a(i+1)
endfor
```

```
forall (i=1:4)
 a(i) = a(i) + 1
 b(i) = a(i-1) + a(i+1)
endforall
```

```
dopar (i = 1:4)
 a(i) = a(i) + 1
 b(i) = a(i-1) + a(i+1)
enddopar
```

Iniciais		Após *for*	Após *forall*	Após *dopar*
$a(0)$ 1				
$a(1)$ 2	$b(1)$	4	5	4
$a(2)$ 3	$b(2)$	7	8	6
$a(3)$ 4	$b(3)$	9	10	8
$a(4)$ 5	$b(4)$	11	11	10
$a(5)$ 6				

Paralelismo Funcional

Paralelismo de tarefas ou funcional:

- Qualquer trecho independente do programa que pode ser encapsulado em uma tarefa;
- uma tarefa por processador, ou tarefa paralela em vários processadores;
- grafo de tarefas:
  - Aresta direcionada partindo de uma tarefa para outra;
  - a ligação atesta uma dependência;
  - uma tarefa só pode começar após a finalização de suas dependências;
- escalonamento estático ou dinâmico (*pool* de tarefas);

Muitos programas *multi-thread* são baseados no aproveitamento de paralelismo funcional, quando cada *thread* executa uma função.

Representação Explícita ou Implícita de Paralelismo

Paralelismo é explícito no código do programa ou não?

- Particionamento de tarefas;
- distribuição de dados;
- comunicação e sincronização.

Existe um *trade off* entre facilidade de desenvolvimento, complexidade do compilador e desempenho final.

Paralelismo Implícito

Paralelização via compilador:

- Programador cria programa serial, fica a cargo do compilador:
  - Análise de dependências;
  - balanceamento de carga;
  - comunicação.
- o uso de ponteiros e referências dinâmicas dificultam o trabalho do compilador;
- em geral, o tempo de execução não é satisfatório.

Linguagens funcionais:

- Ausência de efeitos colaterais;
- natural distribuir diferentes invocações para processadores distintos;
- fica a questão de em qual nível da recursão aplicar o paralelismo.

Paralelismo Implícito

O que o programador informa?

Some dois vetores.

O que o compilador faz?

- Define o esquema de criação das tarefas;
- determina qual partição dos vetores cada tarefa será responsável;
- designa um *thread* para cada tarefa;
- configura como cada partição somada é copiada de cada *thread* para consolidar o resultado final.

Paralelismo Explícito e Distribuição Implícita

O que o programador informa?

Some dois vetores usando quatro tarefas.

O que o compilador faz?

- Determina qual partição dos vetores cada tarefa será responsável;
- designa um *thread* para cada tarefa;
- configura como cada partição somada é copiada de cada *thread* para consolidar o resultado final.

Distribuição Explícita

O que o programador informa?

Some dois vetores de tamanho 100 usando quatro tarefas, uma por *thread*, a primeira da posição 0 a 24, a segunda da posição 25 a 49, a terceira da posição 50 a 74 e a quarta da posição 75 a 99.

O que o compilador faz?

- configura como cada partição somada é copiada de cada *thread* para consolidar o resultado final.

Comunicação e Sincronização Explícita

O que o programador informa?

Some dois vetores de tamanho 100 usando quatro tarefas, uma por *thread*, a primeira da posição 0 a 24, a segunda da posição 25 a 49, a terceira da posição 50 a 74 e a quarta da posição 75 a 99.
Espere cada *thread* terminar. Aloque um vetor de 100 posições, copie o resultado da primeira *thread* nas posições 0-24, da segunda *thread* nas posições 25-49, da terceira *thread* nas posições 50-74 e da quarta *thread* nas posições 75 a 99.

O que o compilador faz?

- apenas gera código...

Padrões de Programação Paralela

Padrões fornecem coordenação específica para processos ou *threads*.

- *fork-join*;
- *parbegin-parend*;
- SPMD/SIMD;
- mestre-escravo ou mestre-trabalhador;
- cliente-servidor;
- *pipelining*;
- *task pools*;
- produtor-consumidor.

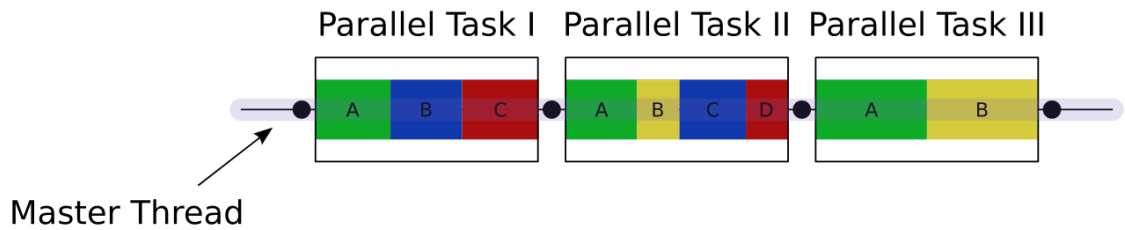
Criação de Processos e *Threads*

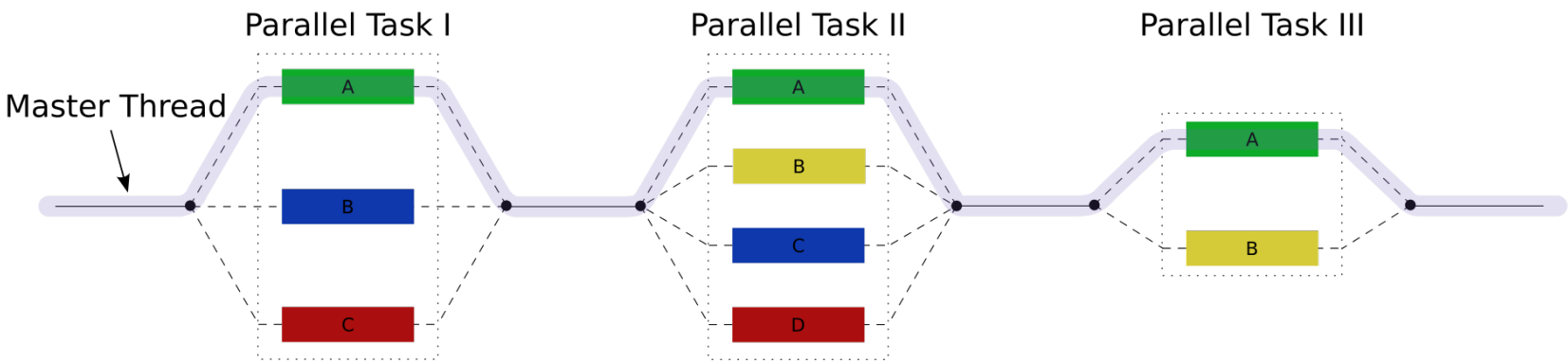
Os padrões organizam quando os *threads* devem ser criados, por quanto tempo executam, como se comunicação e são finalizados.

- Criação de *threads* estática;
- criação de *threads* dinâmica;

Vamos usar os termos *thread* e processo de forma intercambiável.

Fork-Join





https://en.wikipedia.org/wiki/Fork-join_model

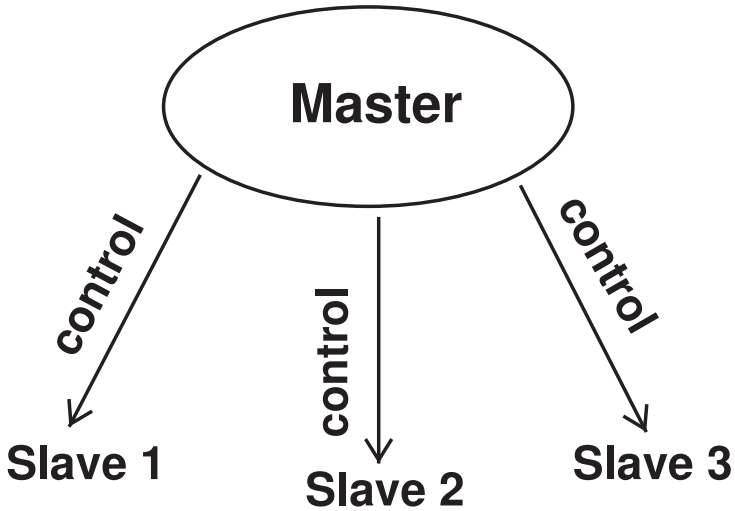
Parbegin-Parend

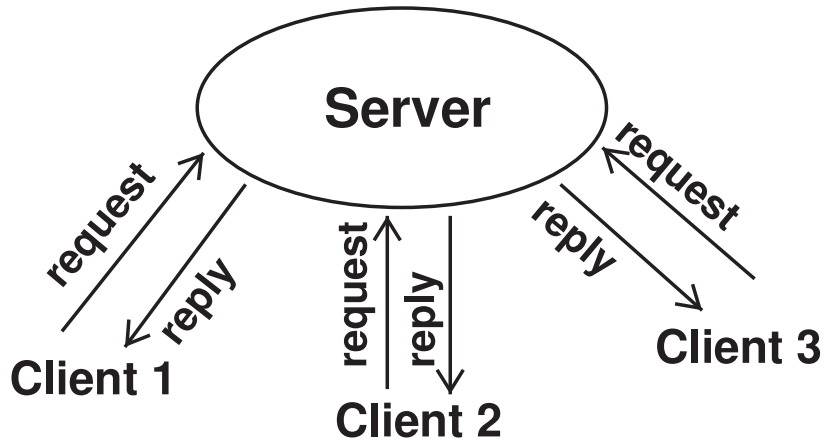
- Bastante similiar ao *fork-join*;
- permite ao programador definir blocos de código que são regiões paralelas;
- quando a execução serial atinge uma região paralela, vários *threads* são criados, cada um executando o mesmo bloco de código;
- as declarações posteriores a região paralela só executam quando todos os *threads* finalizam.

SPMD e SIMD

- Já discutimos SIMD;
- no SPMD, temos um mesmo trecho de programa, não instrução, copiado para os *threads*;
- cada *thread* pode executar partes diferentes do trecho em um dado momento;
- a sincronização é explícita.

Mestre-Escravo e Cliente-Servidor



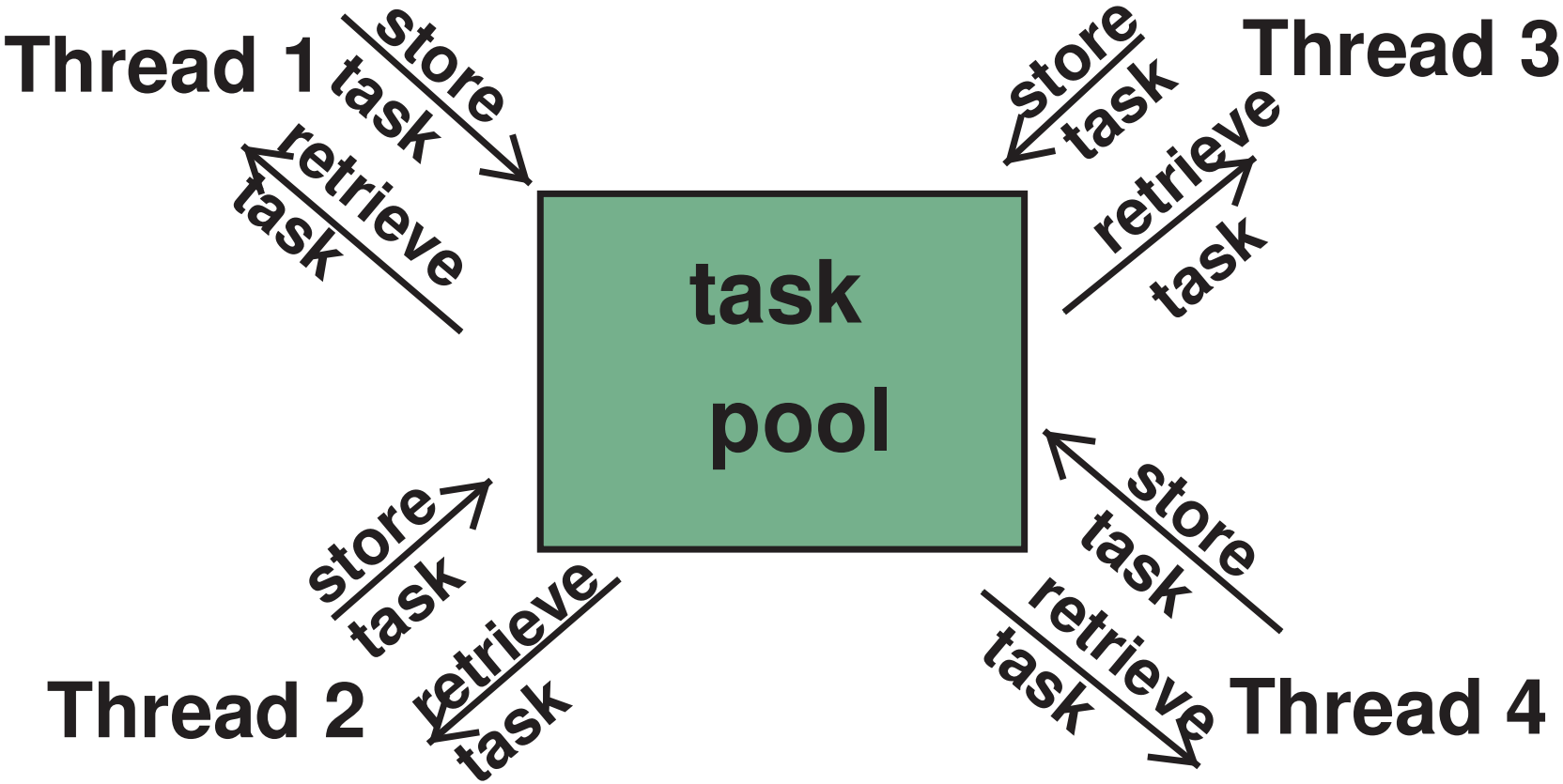


Copyright © 2010 Springer-Verlag GmbH

Pipelining

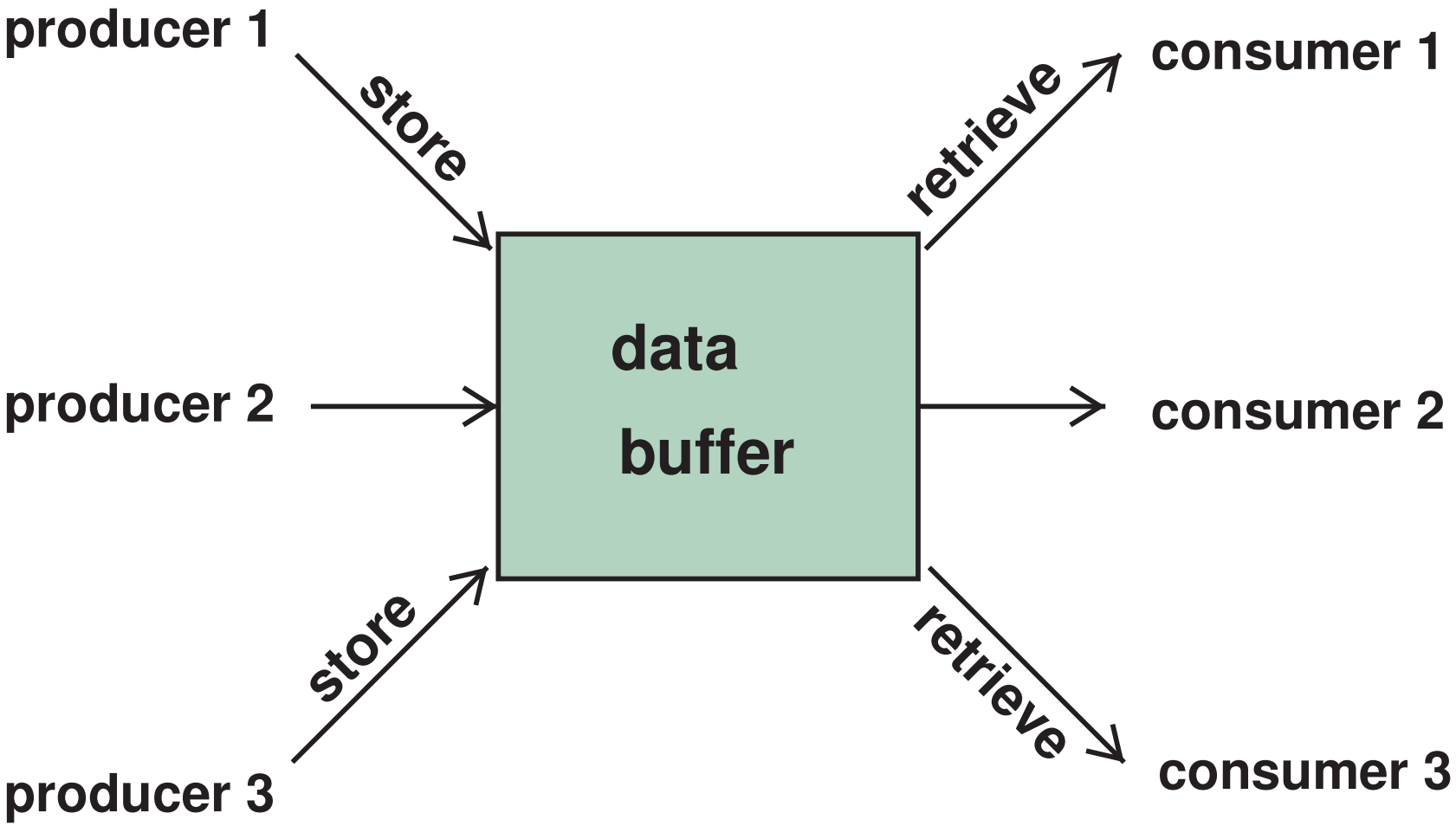
- Os *threads* são organizados em uma ordem serial;
- cada *thread* recebe o resultado do *thread* anterior;
- o primeiro *thread* recebe a entrada do programa;
- o último gera a saída;
- cada um pode ser mapeado para um processador diferente;
- em geral, o código de cada *thread* é diferente.

Task Pools



Copyright © 2010 Springer-Verlag GmbH

Produtor Consumidor



Copyright © 2010 Springer-Verlag GmbH

Distribuição de Dados para *Arrays*

- Algoritmos de análise numérica e computação científica;
- vetores e matrizes;
- um, duas, três ou mais dimensões;
- memória local, acesso sincronizado;
- memória distribuída, comunicação.

A questão é, considerando uma estrutura de N dimensões, como mapear os dados para um conjunto de processadores $P = \{P_1, P_2, ..., P_n\}$.

Distribuição de Dados para *Arrays* de Uma Dimensão

Distribuição em Blocos

- $v = (v_1, v_2, ..., v_n)$;
- p blocos cada um com $\lceil n/p \rceil$ elementos;
- bloco j, $1 \leq j \leq p$ contém os elementos $(j-1) \times \lceil n/p \rceil + 1, ...j \times \lceil n/p \rceil$ sendo atribuido ao processador P_j;
- como ficaria a distribuição de um vetor de 14 elementos em 4 processadores?

Distribuição de Dados para *Arrays* de Uma Dimensão

Distribuição Cíclica

- $v = (v_1, v_2, ..., v_n)$;
- elemento v_i é atribuído ao processador $P_{(i-1)mod(p+1)}, i = 1, ..., n$;
- P_j tem os elementos $j, j + p, ..., j + p \times (\lceil n/p \rceil - 1)$ para $j \leq (n)mod(p)$ e $j, j + p, ..., j + p \times (\lceil n/p \rceil - 2)$ para $(n)mod(p) < j \leq p$.
- como ficaria a distribuição de um vetor de 14 elementos em 4 processadores?

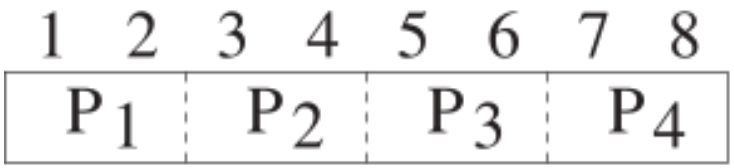
Distribuição de Dados para *Arrays* de Uma Dimensão

Distribuição Cíclica de Blocos

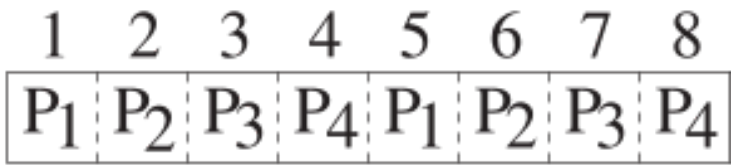
- Combinação das duas anteriores;
- elementos consecutivos são organizados em blocos de tamanho b;
- $b << n/p$;

Distribuição de Dados para *Arrays* de Uma Dimensão

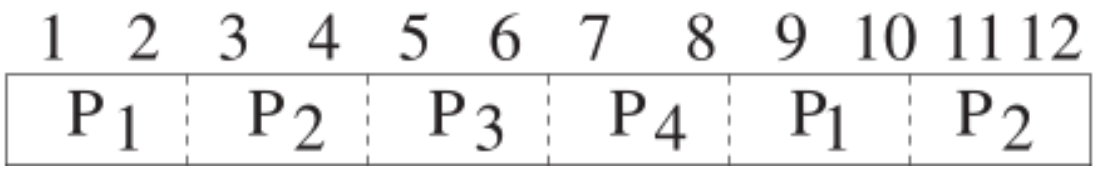
blockwise



cyclic

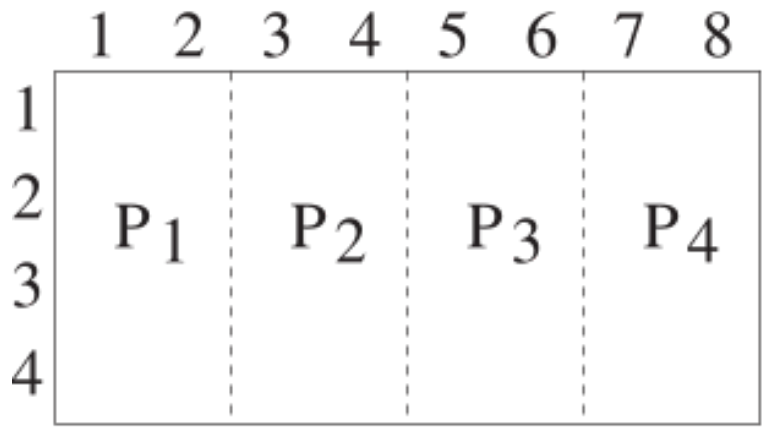


block–cyclic

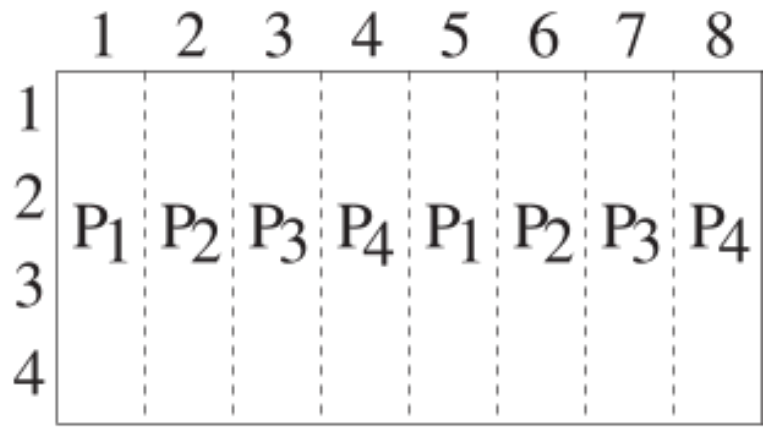


Distribuição de Dados para *Arrays* de Duas Dimensões

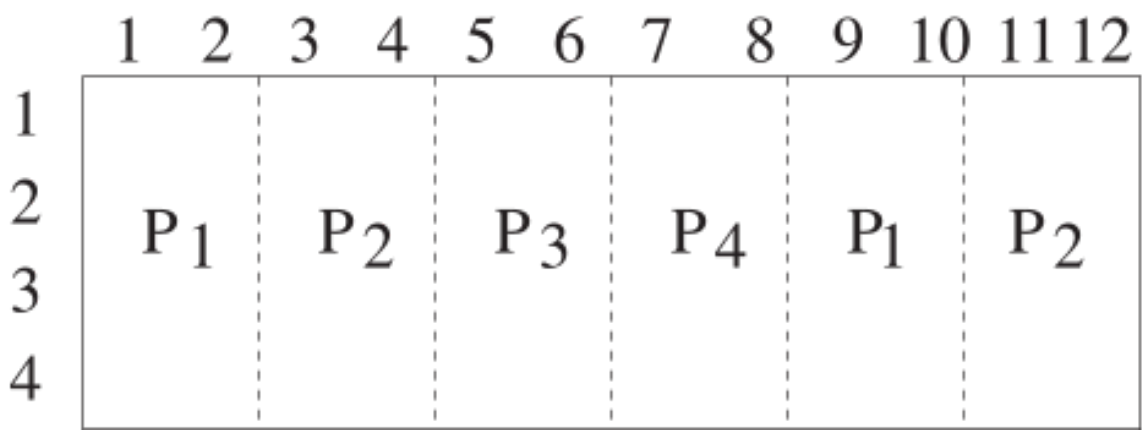
blockwise



cyclic



block–cyclic



Distribuição de Dados para *Arrays* de Duas Dimensões

Distribuição em Tabuleiro

- Os processadores são organizados em uma *mesh* virtual de tamanho $p_1 \times p_2 = p$;
- p_1 são as linhas e p_2 as colunas;
- os elementos (k, l) são mapeados para processadores $P_{i,j}, i = 1, ..., p_1, j = 1, ..., p_2$.

Distribuição de Dados para *Arrays* de Duas Dimensões

Distribuição em Tabuleiro de Blocos

- O *array* é decomposto em $p_1 \times p_2$ blocos;
- bloco (i, j) com $1 \leq i \leq p_1, 1 \leq j \leq p_2$ é designado para o processador com posição (i, j) no *mesh*;
- bloco (i, j) contém os elementos (k, l) com $k = (i-1) \times \lceil n_1/p_1 \rceil + 1, ..., i \times \lceil n_1/p_1 \rceil$ e $l = (j-1) \times \lceil n_2/p_2 \rceil + 1, ..., j \times \lceil n_2/p_2 \rceil$;
- como ficaria para $n_1 = 4, n_2 = 8$ e $p_1 \times p_2 = 2 \times 2 = 4$?

Distribuição de Dados para *Arrays* de Duas Dimensões

Distribuição em Tabuleiro Cíclica

- Atribuições cíclicas de linhas $k = 1, ..., n_1$ para processadores nas linhas $i = 1, ..., p_1$ e colunas $l = 1, ..., n_2$ para processadores nas colunas $j = 1, ..., p_2$;
- elemento (k, l) vai para processador $((k-1) mod(p_1) + 1, (l-1) mod(p_2) + 1)$;
- como ficaria para $n_1 = 4, n_2 = 8$ e $p_1 \times p_2 = 2 \times 2 = 4$?

Distribuição de Dados para *Arrays* de Duas Dimensões

Distribuição em Tabuleiro Cíclica de Blocos

- Blocos de tamanho $b_1 \times b_2$;
- elemento (m, n) percente ao bloco (k, l) com $k = \lceil m/b_1 \rceil$ e $l = \lceil n/b_2 \rceil$;
- bloco (k, l) é atribuído ao processador na posição $((k-1) mod(p_1) + 1, (l-1) mod(p_2 + 1)$;
- como ficaria para $n_1 = 4, n_2 = 12$ e $p_1 \times p_2 = 2 \times 2 = 4$?

Distribuição de Dados para *Arrays* de Duas Dimensões

blockwise

	1	2	3	4	5	6	7	8
1	P_1				P_2			
2								
3	P_3				P_4			
4								

cyclic

	1	2	3	4	5	6	7	8
1	P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2
2	P_3	P_4	P_3	P_4	P_3	P_4	P_3	P_4
3	P_1	P_2	P_1	P_2	P_1	P_2	P_1	P_2
4	P_3	P_4	P_3	P_4	P_3	P_4	P_3	P_4

block–cyclic

	1	2	3	4	5	6	7	8	9	10	11	12
1	P_1		P_2		P_1		P_2		P_1		P_2	
2												
3	P_3		P_4		P_3		P_4		P_3		P_4	
4												

Distribuição de Dados Parametrizada

Vamos considerar agora um *array* com d dimensões distintas:

- *Array* A com conjunto de índices $I_A \subset \mathbb{N}^d$;
- o tamanho do *array* é $n_1 \times n_2 \times ... \times n_d$;
- seus elementos são $A[i_1, ..., i_d]$ com $i = (i_1, ..., i_d) \in I_A$;
- p processadores, organizados em um *mesh* $p_1 \times ... \times p_d$;
- $p = \prod_{i=1}^{d} p_i$.

Distribuição de Dados Parametrizada

Função de Distribuição

Uma função que indica para qual processador p o elemento $i = (i_1, ..., i_d)$ está alocado.

- $\gamma_A : I_A \subset \mathbb{N}^d \rightarrow 2^p$;
- 2^p é o conjunto de todos os subconjuntos de p;
- todo elemento $A[i_1, ..., i_d]$ está mapeado para todos os processadores em $\gamma_A(i_1, ..., i_d)$;
- $\gamma_A(i) = P$ para todo $i \in I_A$ significa que todos os dados estão replicados;
- $|\gamma_A(i) = 1|$ para todo $i \in I_A$ significa que cada elemento está designado para apenas um processador;
- $L(\gamma_A) : P \rightarrow 2^{I^A}$, dado um processador, retorna quais elementos estão mapeados a ele.

Troca de Informação

- Controlar a coordenação entre diferentes partes de um programa paralelo;
- mecanismos de sincronização já estão disponíveis através do SO para memória compartilhada;
- para memória distribuída, precisamos de operações de comunicação bem definidas;

Operações de Comunicação

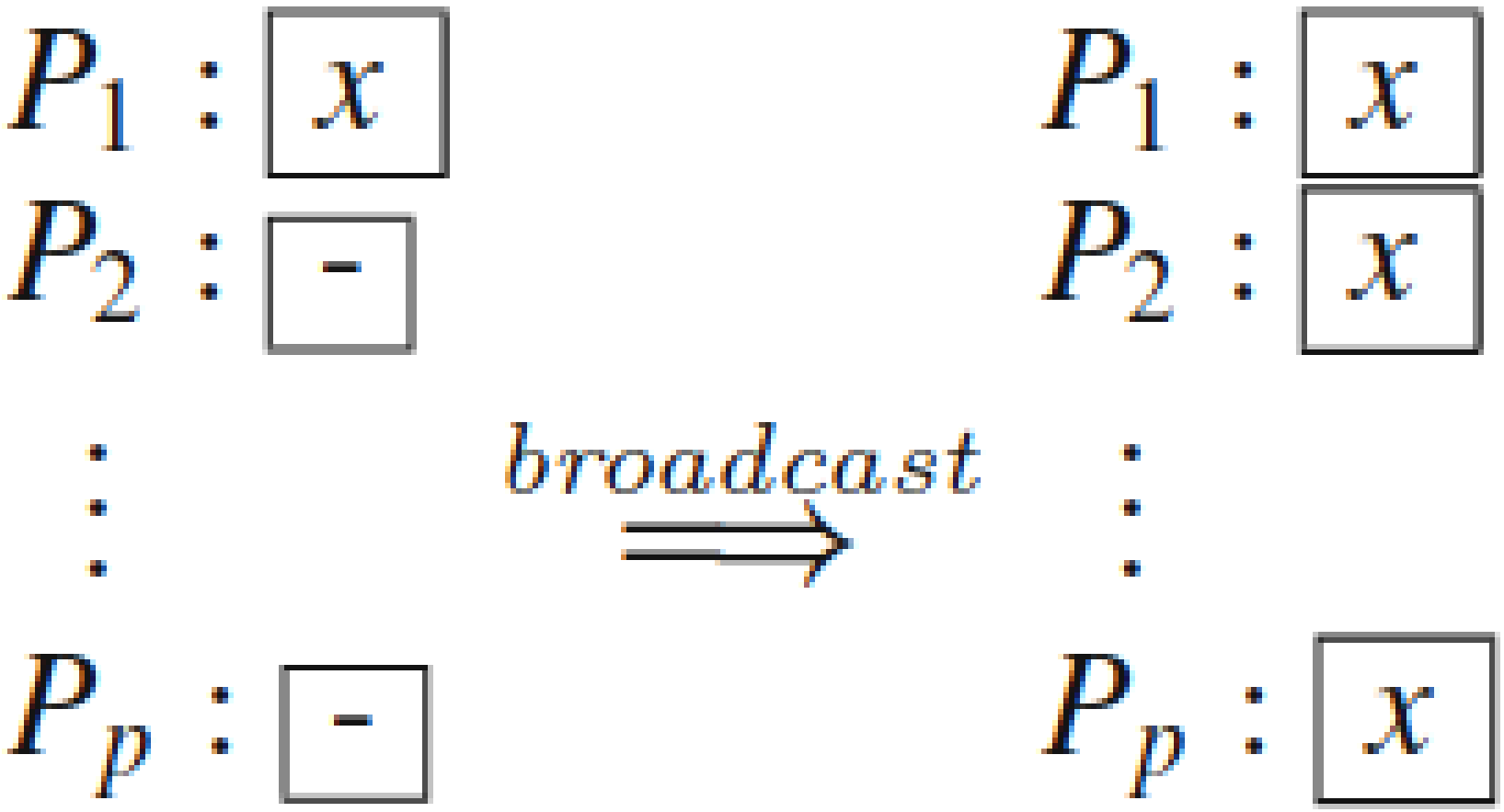
- São operações que os processadores invocam para trocar informações;
- o resultado de uma operação é um processador receber dados que estão na memória de outro processador;
- modelos de programação baseados em troca de mensagens;
- comunicação ponto-a-ponto: uma operação *send* e uma operação *receive* devem ser usadas em par;
- comunicação coletiva global: todos os processos são envolvidos e invocam a operação.

Transferência Única

- Processador P_i (*sender*) envia mensagem para o processador P_j (*receiver*), com $i \neq j$;
- P_i indica um *buffer* com os dados e o número identificador do processador destino (*rank*);
- P_j indica um *buffer* para receber os dados e o *rank* do *sender*;
- para cada operação invocada, é preciso que sua contraparte também seja.

Broadcast Único

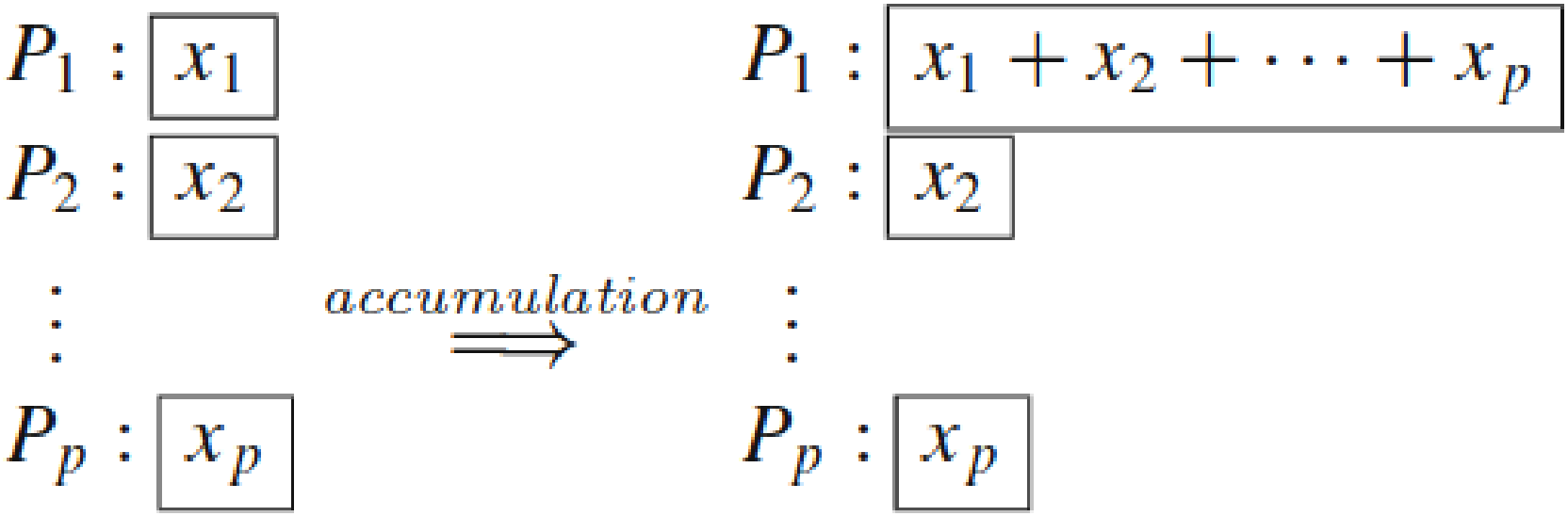
Um único processador P_i envia os mesmos dados para todos os outros processos.



Cada processador precisa invocar a operação.

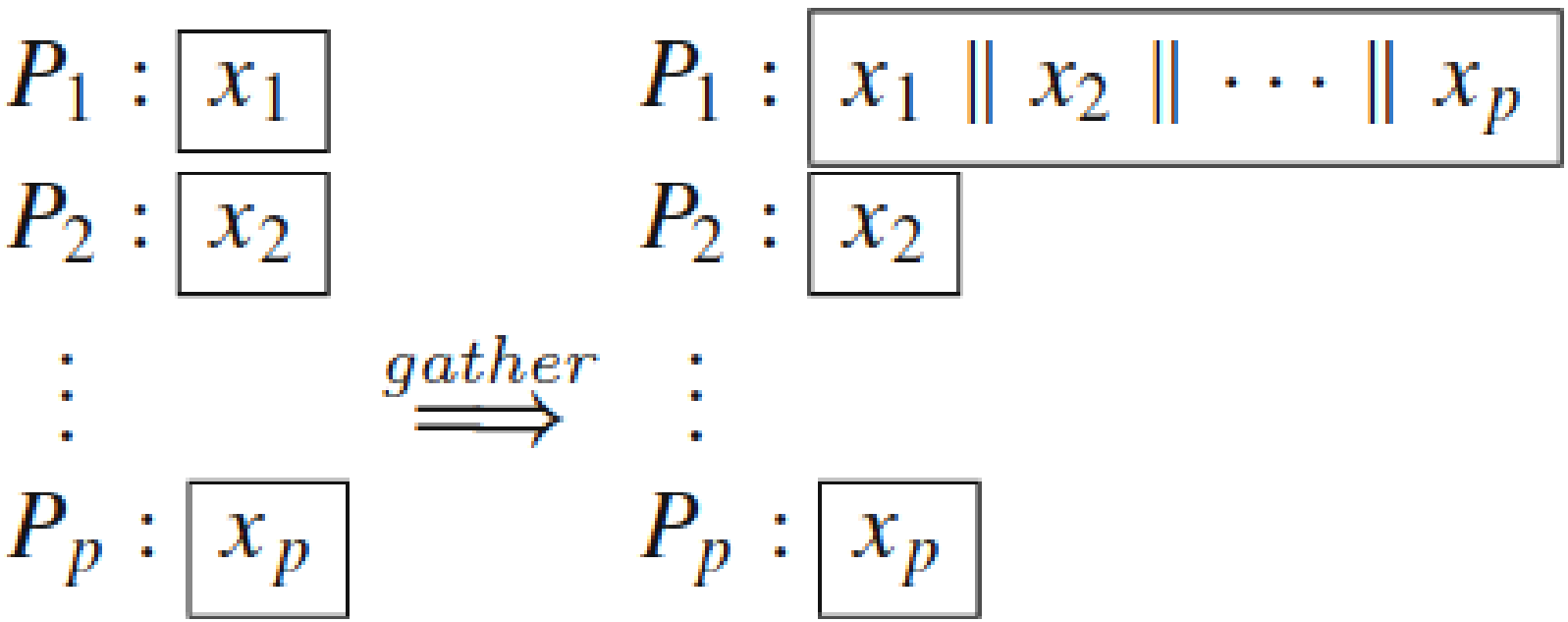
Acumulação Única

Operação de redução: associativa e acumulativa.

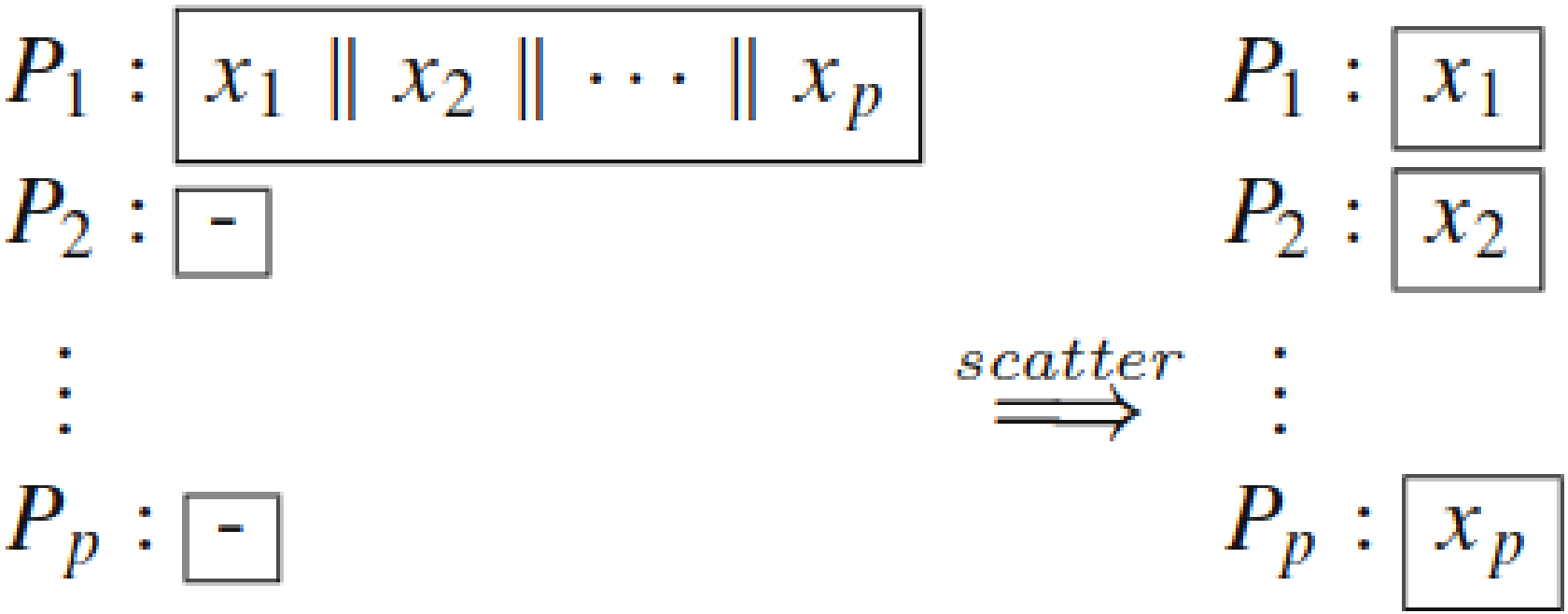


Gather

Concatenação de dados.

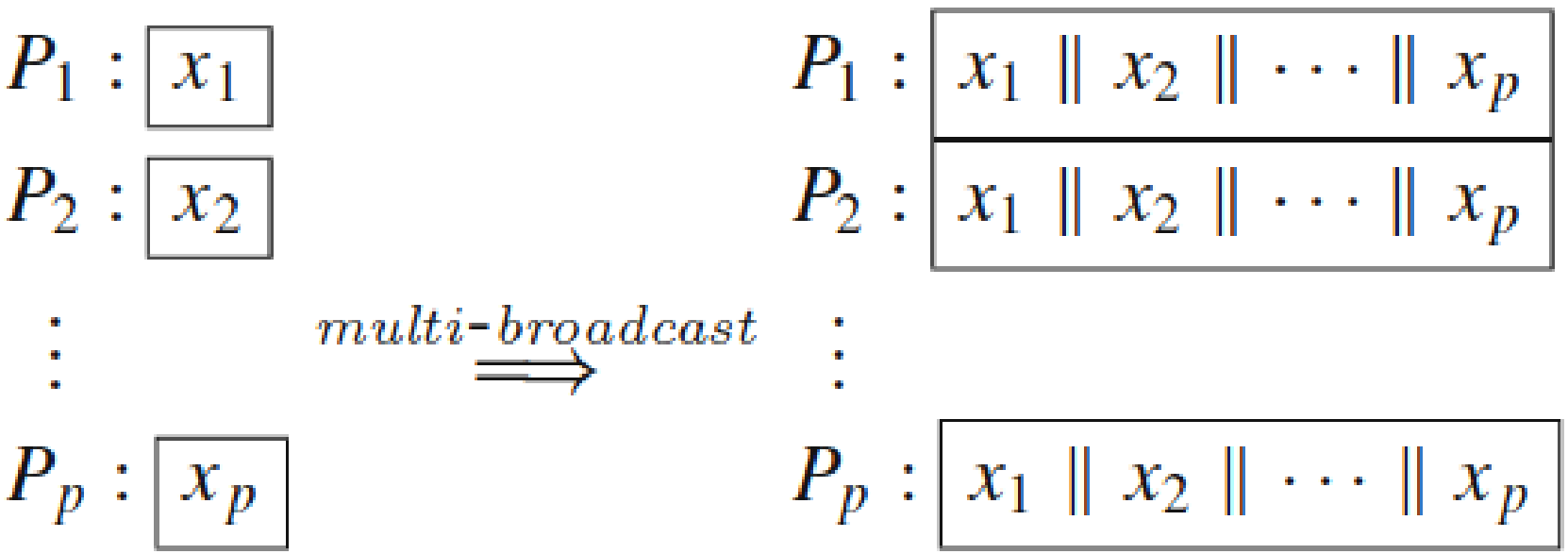


Scatter

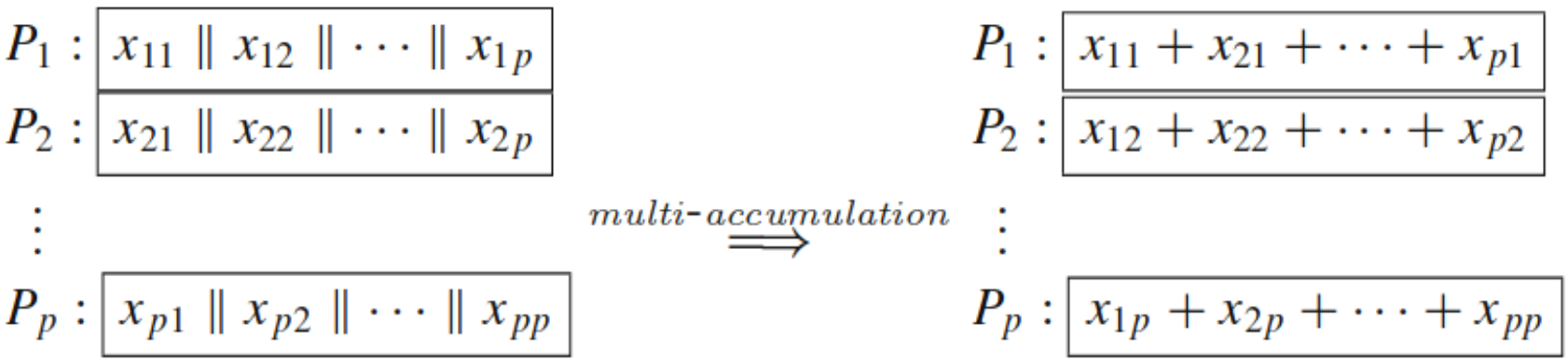

P_1 : $x_1 \parallel x_2 \parallel \cdots \parallel x_p$
P_2 : -
P_p : -
scatter
P_1 : x_1
P_2 : x_2
P_p : x_p

Multi-Broadcast

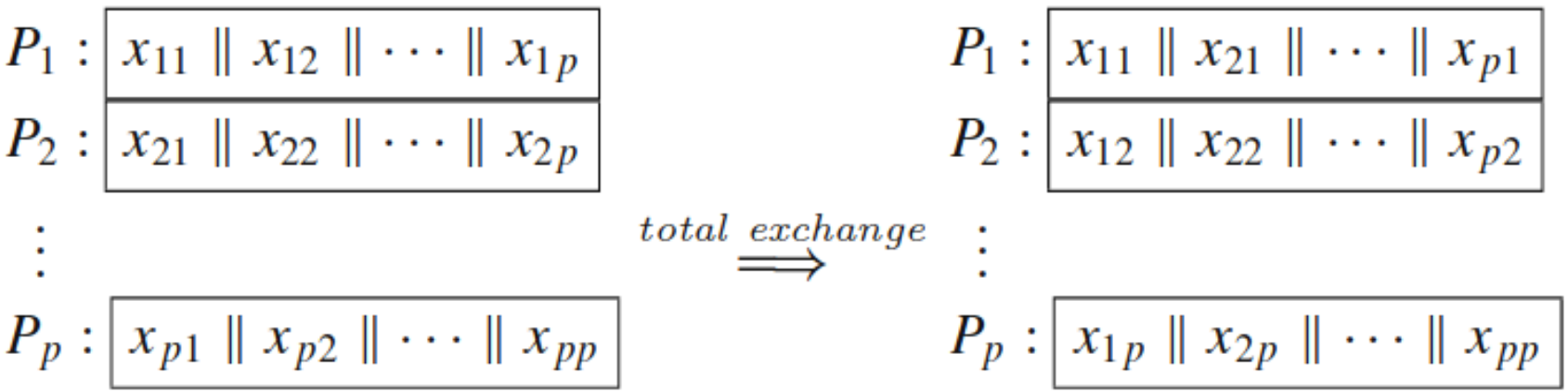
Não há processador raiz.



Acumulação Múltipla

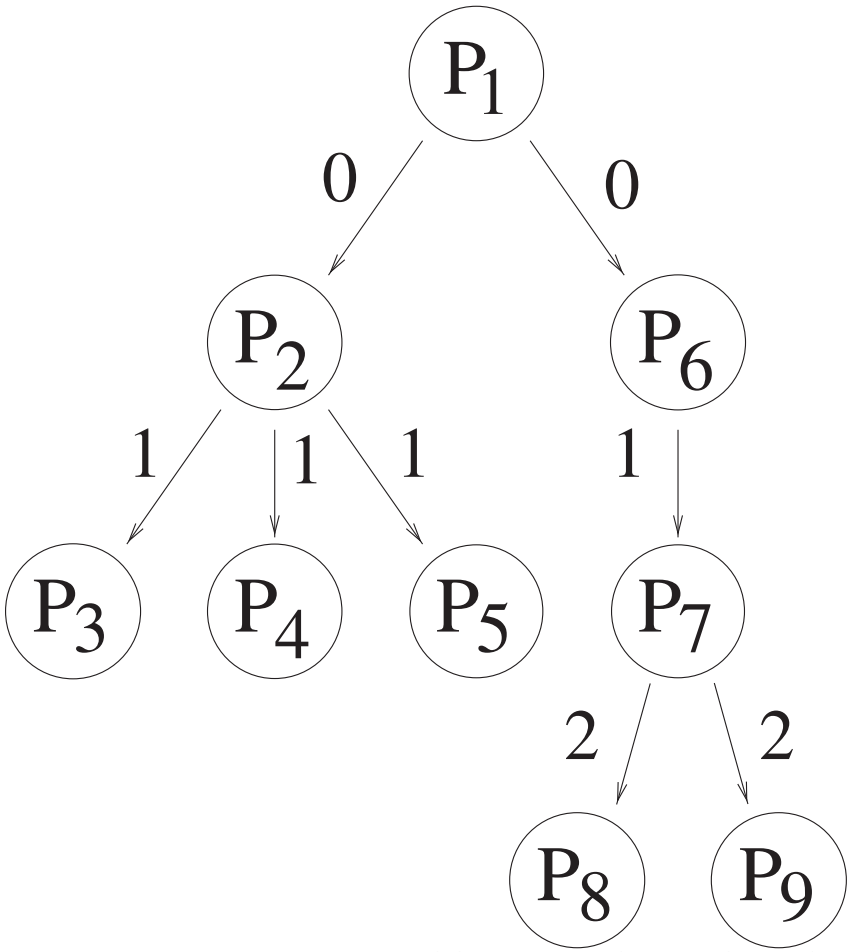


Troca Total

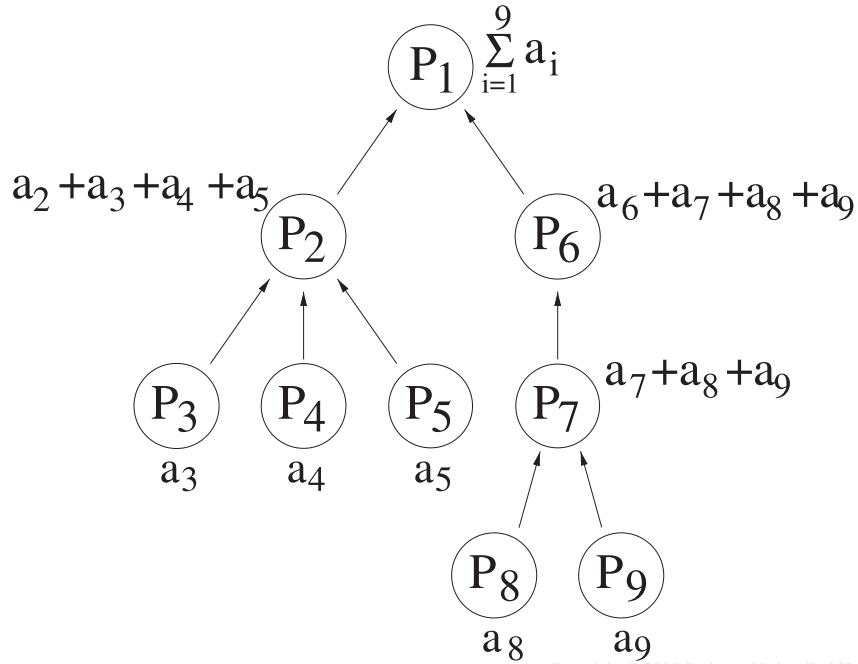

$P_1 : x_{11} \parallel x_{12} \parallel \cdots \parallel x_{1p}$
$P_2 : x_{21} \parallel x_{22} \parallel \cdots \parallel x_{2p}$
$P_p : x_{p1} \parallel x_{p2} \parallel \cdots \parallel x_{pp}$
$total\ exchange \Longrightarrow$
$P_1 : x_{11} \parallel x_{21} \parallel \cdots \parallel x_{p1}$
$P_2 : x_{12} \parallel x_{22} \parallel \cdots \parallel x_{p2}$
$P_p : x_{1p} \parallel x_{2p} \parallel \cdots \parallel x_{pp}$

Dualidade das Operações de Comunicação

Um *broadcast* usa o mesmo caminho que uma acumulação.



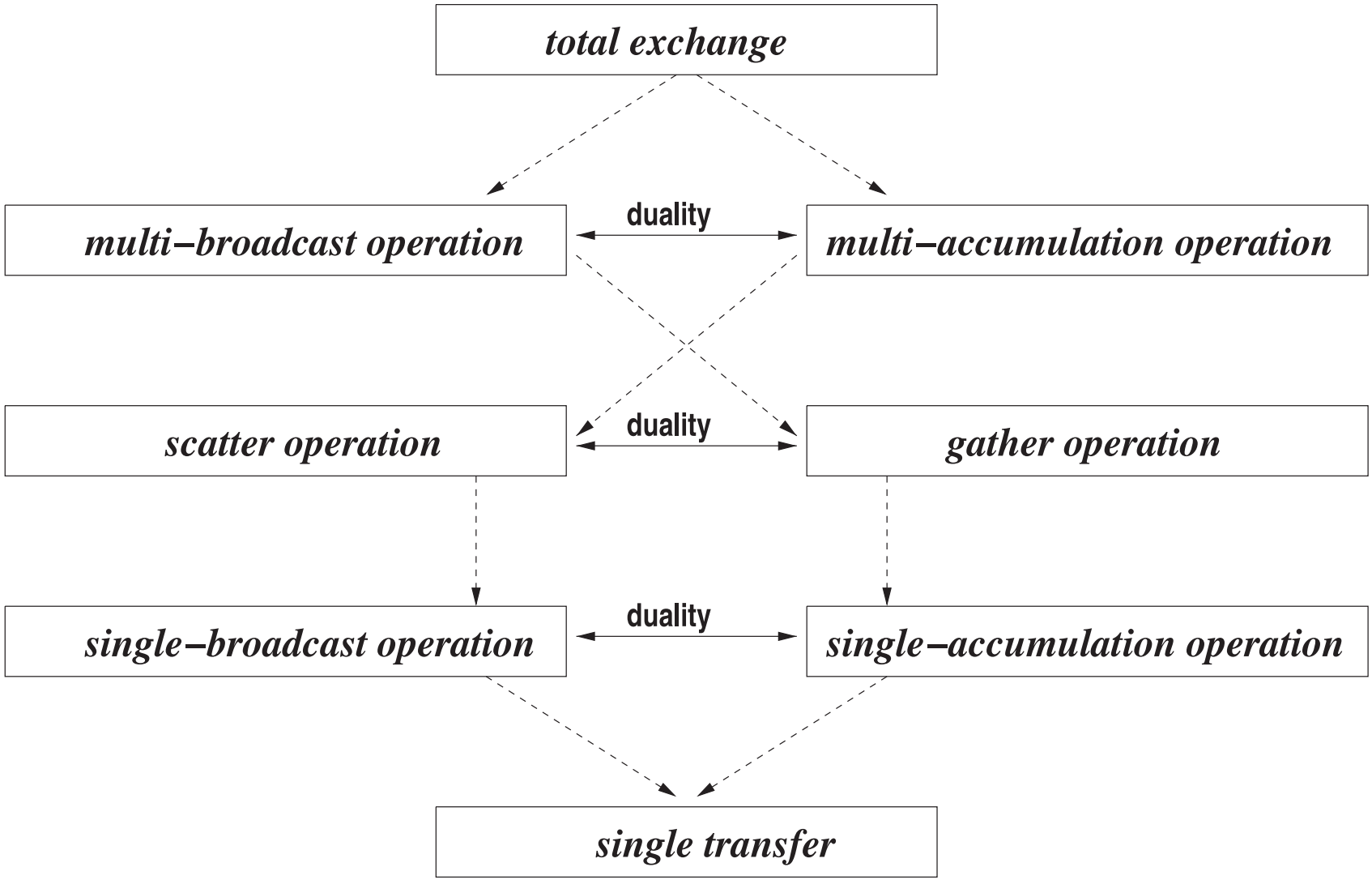
Copyright © 2010 Springer-Verlag GmbH



Copyright © 2010 Springer-Verlag GmbH

São operações duais.

Hierarquia das Operações de Comunicação



Copyright © 2010 Springer-Verlag GmbH

Produto Vetor Matriz Paralelo

- Produto $Ab = c$;
- $A \in \mathbb{R}^{n \times m}$, matriz $n \times m$;
- $b \in \mathbb{R}^m$, vetor de tamanho m;

$$c_i = \sum_{j=1}^{m} a_{ij} b_j, i = 1, ..., n$$

- $c = (c_1, ..., c_n)$;
- $A = (a_{ij})_{i=1,...,n, j=1,...,m}$.

Computação de Produtos Escalares

Calculo de n produtos escalares entre as linhas $a_1, ..., a_n$ de A e o vetor b.

$$A.b = \begin{pmatrix} (a_1, b) \\ . \\ . \\ . \\ (a_n, b) \end{pmatrix}$$

Na qual cada elemento $(x, y) = \sum_{j=1}^{m} x_j y_j$ para $x, y \in \mathbb{R}^m$ com $x = (x_1, ..., x_m)$ e $y = (y_1, ..., y_m)$ representando o produto escalar de dois vetores.

Computação de Produtos Escalares

Algoritmo serial:

```
for (i=0; i<n; i++) c[i] = 0;
for (i=0; i<n; i++)
   for (j=0; j<m; j++)
      c[i] = c[i] + A[i][j] * b[j];
```

A matriz $A \in \mathbb{R}^{n \times m}$ é um *array* multidimensional, sendo $b \in \mathbb{R}^m$ e $c \in \mathbb{R}^n$ *arrays* unidimensionais.

Combinação Linear

Combinação linear das colunas $\tilde{a}_1, ..., \tilde{a}_m$ de A com os coeficientes $b_1, ..., b_m$.

$$A.b = \sum_{j=1}^{m} b_j \tilde{a}_j$$

```
for (i=0; i<n; i++) c[i] = 0;
for (j=0; j<m; j++)
   for (i=0; i<n; i++)
      c[i] = c[i] + A[i][j] * b[j];
```

Para cada $j = 0, ..., m-1$, uma coluna $\tilde{a}_j$ é adicionada para a combinação linear.

Versões Paralelas

- A representação orientada a linhas no cálculo dos produtos escalares leva a uma implementação paralela na qual um conjunto de p processadores calcula aproximadamente n/p produtos escalares cada.
- A representação orientada a colunas no cálculo das combinações lineares leva uma implementação paralela na qual cada um dos processadores calcula uma combinação linear com aproximadamente m/p vetores coluna.

Computação Paralela de Produtos Escalares

Distribuição de Dados:

- O processador computando o produto escalar $(a_i, b), i \in 1, ..., n$ acessa apenas elelementos armazenados na memória local;
- linha a_i e vetor b;
- b é replicado;
- podemos usar tanto distribuição de blocos, cíclica ou cíclica em blocos para distribuir as linhas de a.

Computação Paralela de Produtos Escalares

- O vetor resultante, c, pode precisar estar disponível para todos os processadores;
- operação multi-*broadcast*;
- ao final dos cálculos, todos processadores invocam a operação, tendo uma barreira de sincronização.

Computação Paralela de Produtos Escalares

Função de Distribuição

Cara processador P_k armazena uma parte da matriz $A_{n \times m}$. Usando blocos, será uma matriz $local\_A$ com dimensões $local\_n \times m$. O resultado é armazenado no vetor $local\_c$ de dimensão $local\_n$.

$$local\_A[i][j] = A[i + (k-1) \times n/p][j]$$

- $i = 0, ..., n/p - 1$;
- $j = 0, ..., m - 1$;
- $k = 1, ..., p$

Computação Paralela de Produtos Escalares

Consolidação dos Resultados

A operação de comunicação

$$multi\_broadcast(local\_c, local\_n, c)$$

transfere o resultados parciais entre os processadores. O vetor c irá conter os valores:

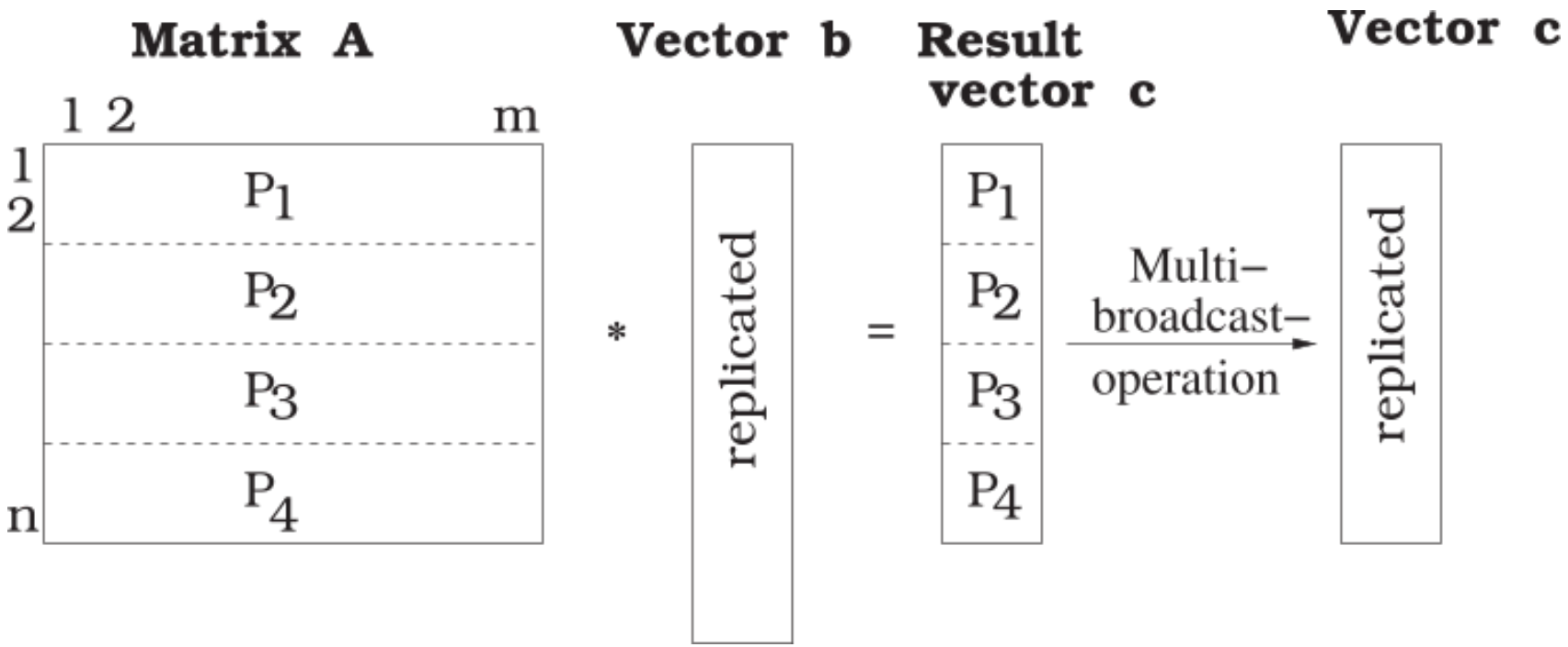
$$c[i + (k-1) \times n/p] = local\_c[i]$$

- $i = 0, ..., n/p - 1$;
- $k = 1, ..., p$

Computação Paralela de Produtos Escalares

```
/* Matrix-vector product Ab = c with parallel inner products*/
/* Row-oriented blockwise distribution of A */
/* Replicated distribution of vectors b and c */
local_n = n/p;
for (i=0; i<local_n; i++) local_c[i] = 0;
for (i=0; i<local_n; i++)
    for (j=0; j<m; j++)
        local_c[i] = local_c[i] + local_A[i][j] * b[j];
multi_broadcast(local_c,local_n,c);
/* Multi-broadcast operation of (c[0],...,c[local_n]) to global_c*/
```

Computação Paralela de Produtos Escalares



Computação Paralela de Produtos Escalares

- Até agora, o exemplo é para arquiteturas de memória distribuída;
- para a memória compartilhada, como A e b estão acessíveis a todos processadores (*threads*), o cenário é mais simples;
- como cada processo acessa uma parte diferente dos vetores, não há conflito e nem seção crítica na computação do vetor resultante;
- ao final, sincronização garante que todos os processos seguem adiante com c devidamente calculado.

Computação Paralela de Produtos Escalares

```
/* Matrix-vector product Ab=c with parallel inner products*/
/* Row-oriented distribution of the computation */
local_n = n/p;
for (i=0; i<local_n; i++) c[i+(k-1)*local_n] = 0;
for (i=0; i<local_n; i++)
    for (j=0; j<m; j++)
        c[i+(k-1)*local_n] =
        c[i+(k-1)*local_n] + A[i+(k-1)*local_n][j] * b[j];
synch();
```

Computação Paralela de Combinações Lineares

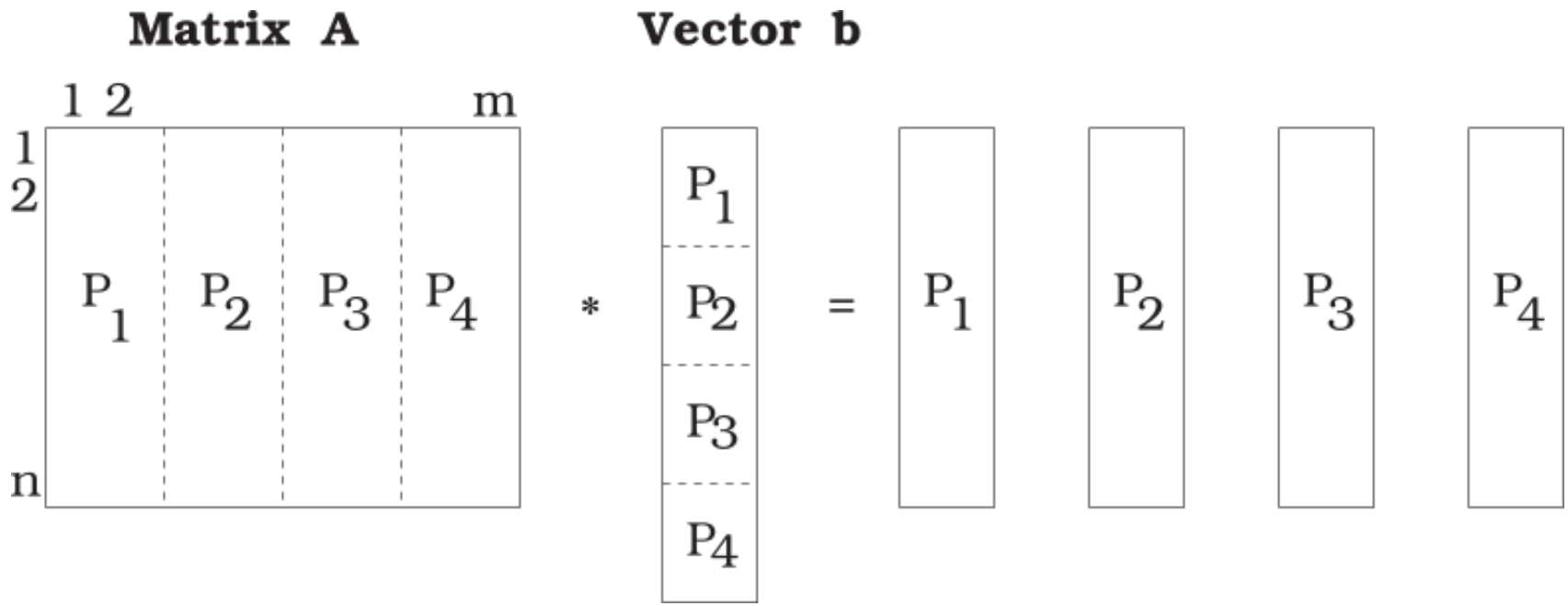
Para uma arquitetura de memória distribuída:

- Distribuição orientada a colunas;
- cada processador faz uma parte da combinação linear correspondente às colunas $\tilde{a}_i, i \in 1, ..., m$;
- para uma distribuição em blocos, o processador P_k é dono das colunas $\tilde{a}_i$ com $i = m/p \times (k-1) + 1, ..., m/p \times k$. Cada processador calcula o vetor n-dimensional d_k:

$$d_k = \sum_{j=m/p\times(k-1)+1}^{m/p\times k} b_j \tilde{a}_j$$

equivalente a combinação linear parcial do resultado toda, com $k = 1, ..., p$.

Computação Paralela de Combinações Lineares



Computação Paralela de Combinações Lineares

Ao final, os vetores d_k estarão em cada memória local dos processadores, precisam ser acumulados, ou seja,

$$c = \sum_{k=1}^{p} d_k$$

Os vetores locals *local_b* e *local_A* tem as seguintes distribuições:

$$local\_A[i][j] = A[i][j + (k-1) \times m/p]$$

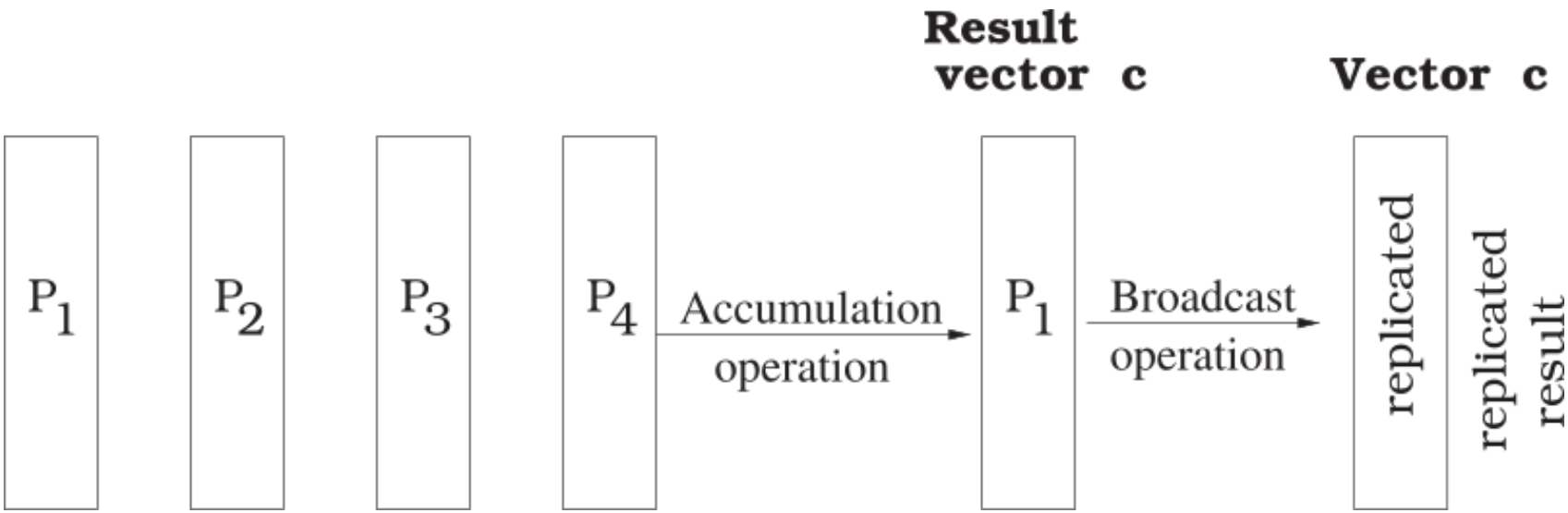
$$local\_b[j] = b[j + (k-1) \times m/p]$$

para cada processador P_k, com
$j = 0, ..., m/p - 1, i = 0, ..., n - 1, k = 1, ..., p$

Computação Paralela de Combinações Lineares

```
/* Matrix-vector product Ab=c with parallel linear combination*/
/* Column-oriented distribution of A */
/* Replicated distribution of vectors b and c */
local_m=m/p;
for (i=0; i<n; i++) d[i] = 0;
for (j=0; j<local_m; j++)
    for (i=0 ;i<n; i++)
        d[i] = d[i] + local_b[j] * local_A[i][j];
single_accumulation(d,n,c,ADD,1);
single_broadcast(c,1);
```

Computação Paralela de Combinações Lineares



Conclusão

- Apresentamos as características dos principais modelos;
- discutimos como pensar um programa paralela;
- entendemos a distribuição de dados em vetores;
- apresentamos um exemplo para discutir os conceitos;
- vamos agora para apresentar ambientes específicos.