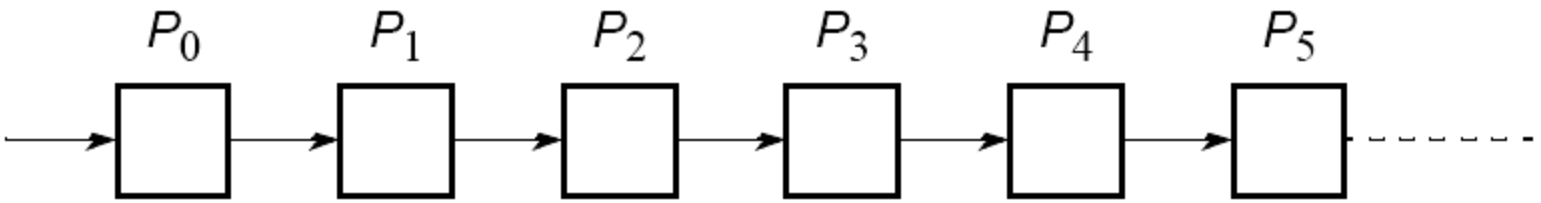
Técnicas de Paralelização

3 – Linha de Produção (Pipelining)

Computação em pipeline

Divisão do problema num conjunto de tarefas que têm de ser concluídas uma após outra (base da programação sequencial). Cada tarefa é executada por um processo.



Exemplo

Somar os elementos de um array:

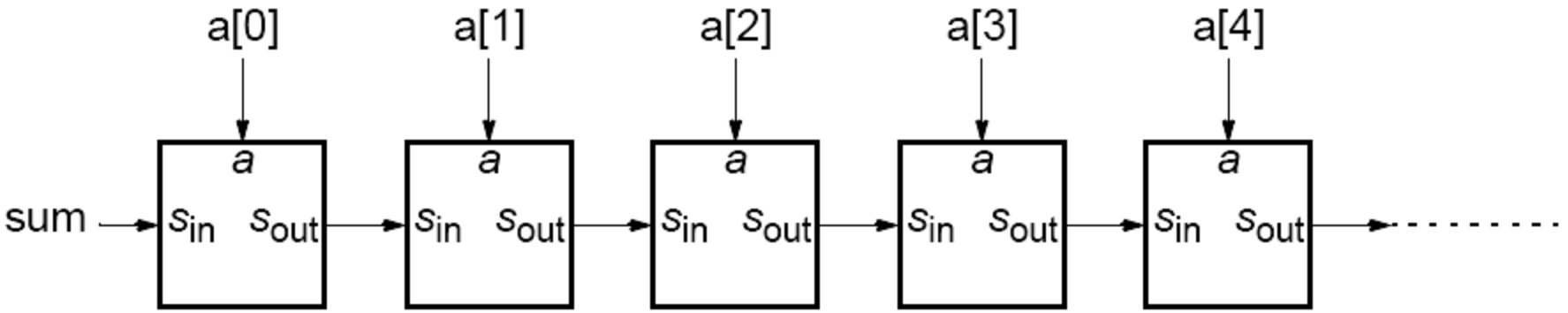
Versão sequencial:

```
for (i = 0; i < n; i++)
sum = sum + a[i];
```

O ciclo pode ser desdobrado em:

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
    .
    .
    .
```

Pipeline para um ciclo desdobrado:



Pseudo -código para processo i

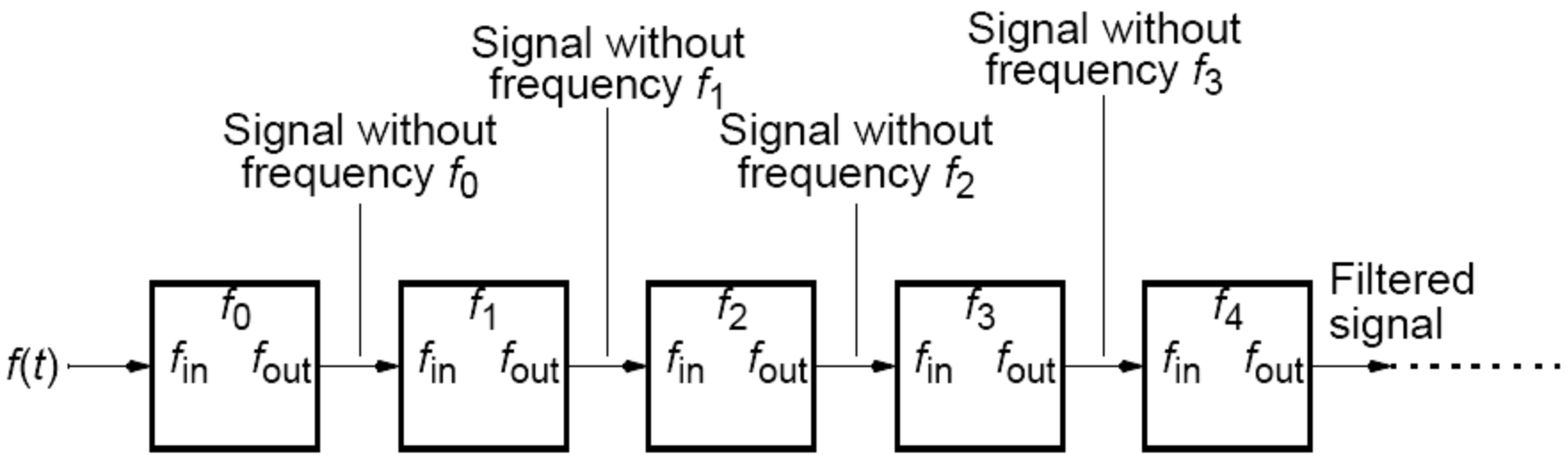
Recv da esquerda a Soma

Soma = Soma + a[i]

Send Soma para a direita

Outro Exemplo

Filtro de frequências – Programa para remover frequências específicas (f_0, f_1, f_2,f_3, etc.) de um sinal digitalizado $f(t)$.



Utilização do pipeline

Assumindo que um problema pode ser dividido numa sequência de tarefas sequenciais, a estratégia de pipeline pode dar resultado em <u>três tipos de problemas</u>:

1. Se mais do que uma instância do problema vai ser executada.

2. Se uma série de itens de dados têm de ser processados, e cada item requer várias operações.

3. Se a informação para iniciar o próximo processo pode ser passada para a frente antes do processo completar todos as operações.

Diagrama espaço-tempo para pipeline "Tipo 1"

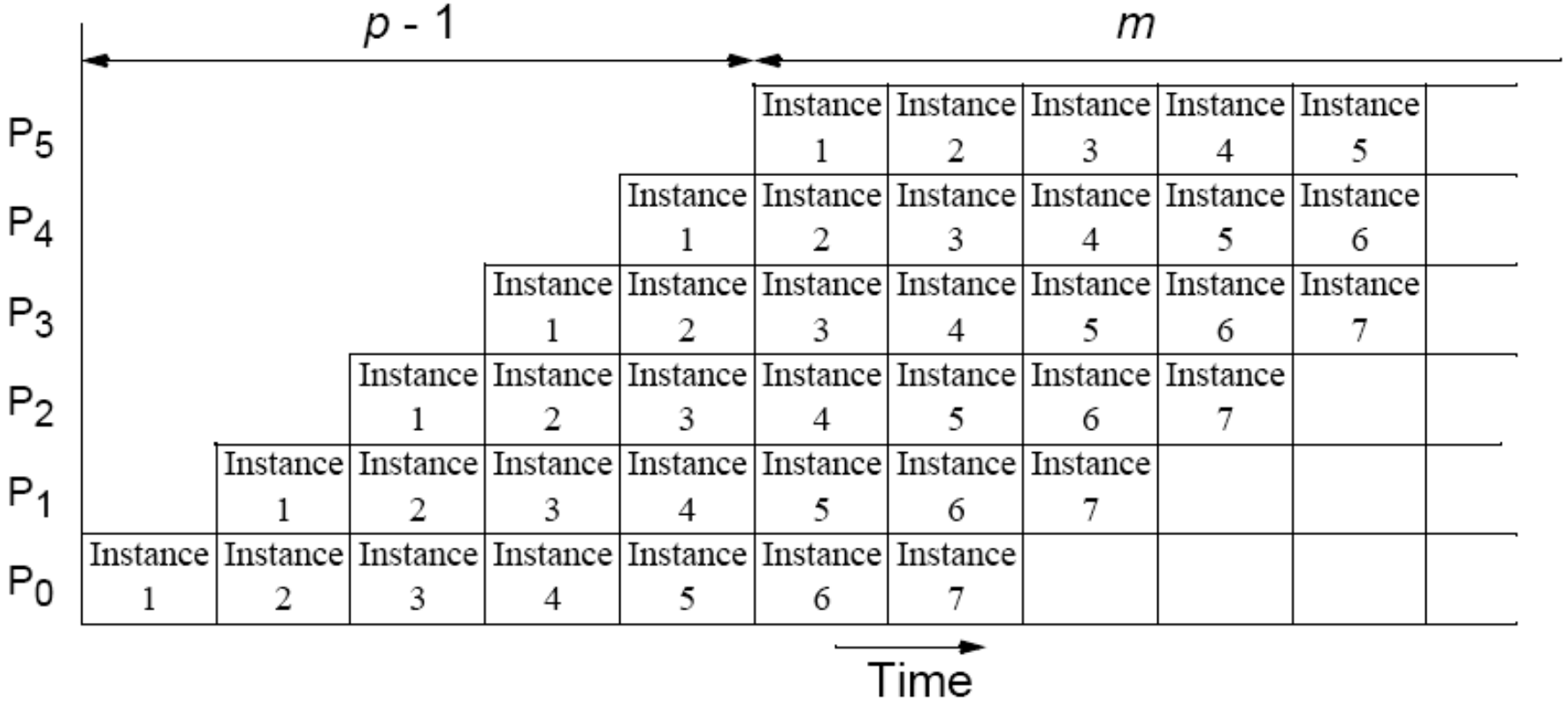


Diagrama espaço tempo alternativo

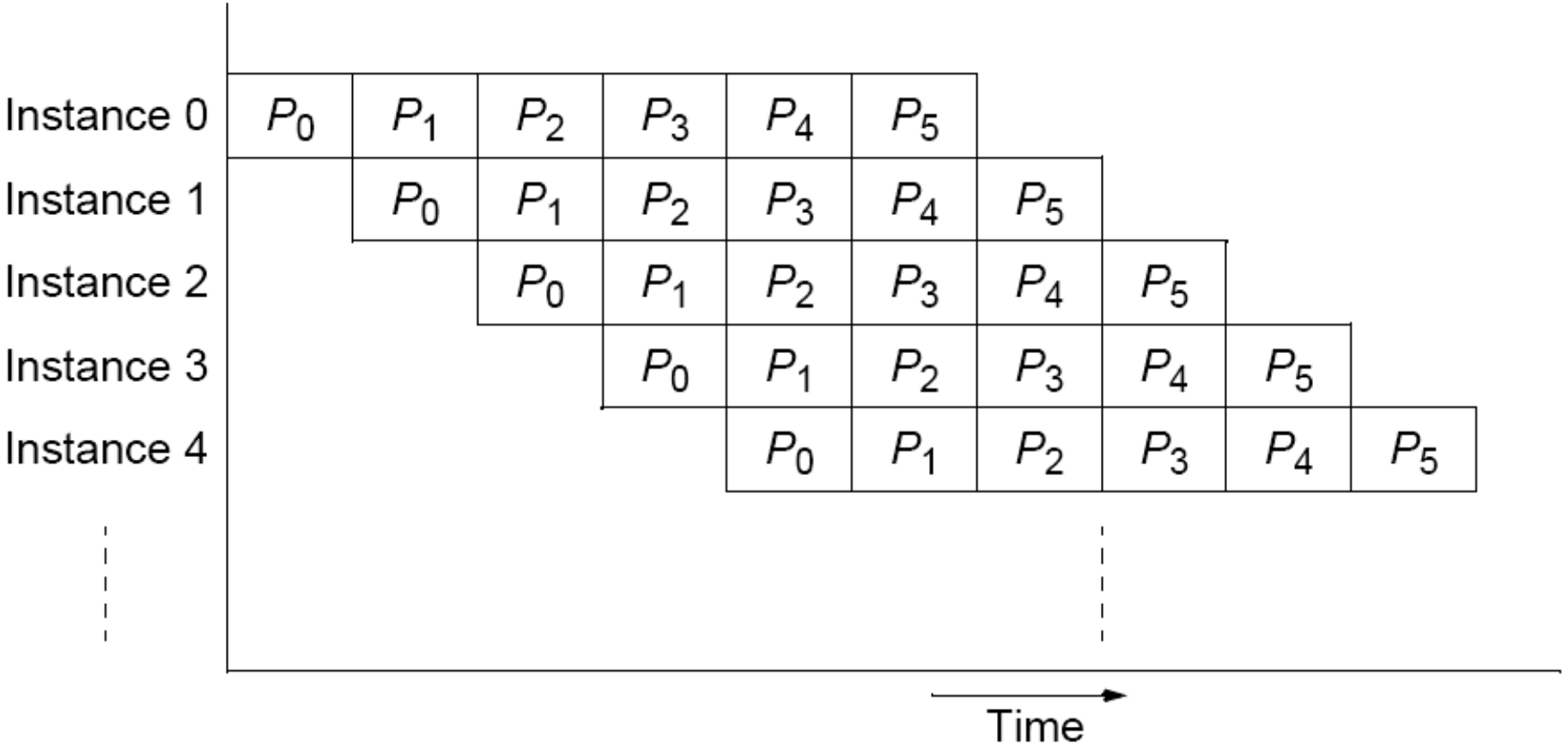
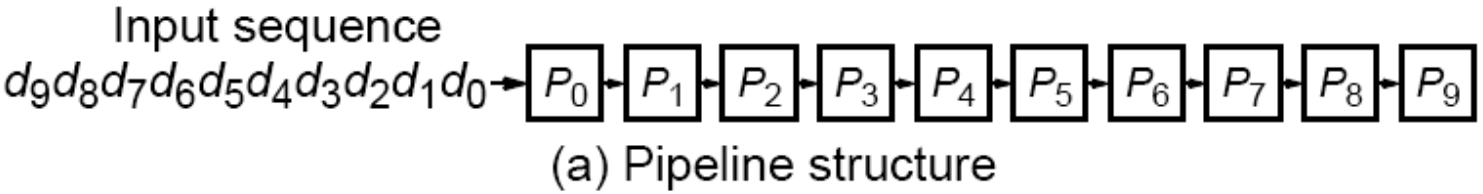
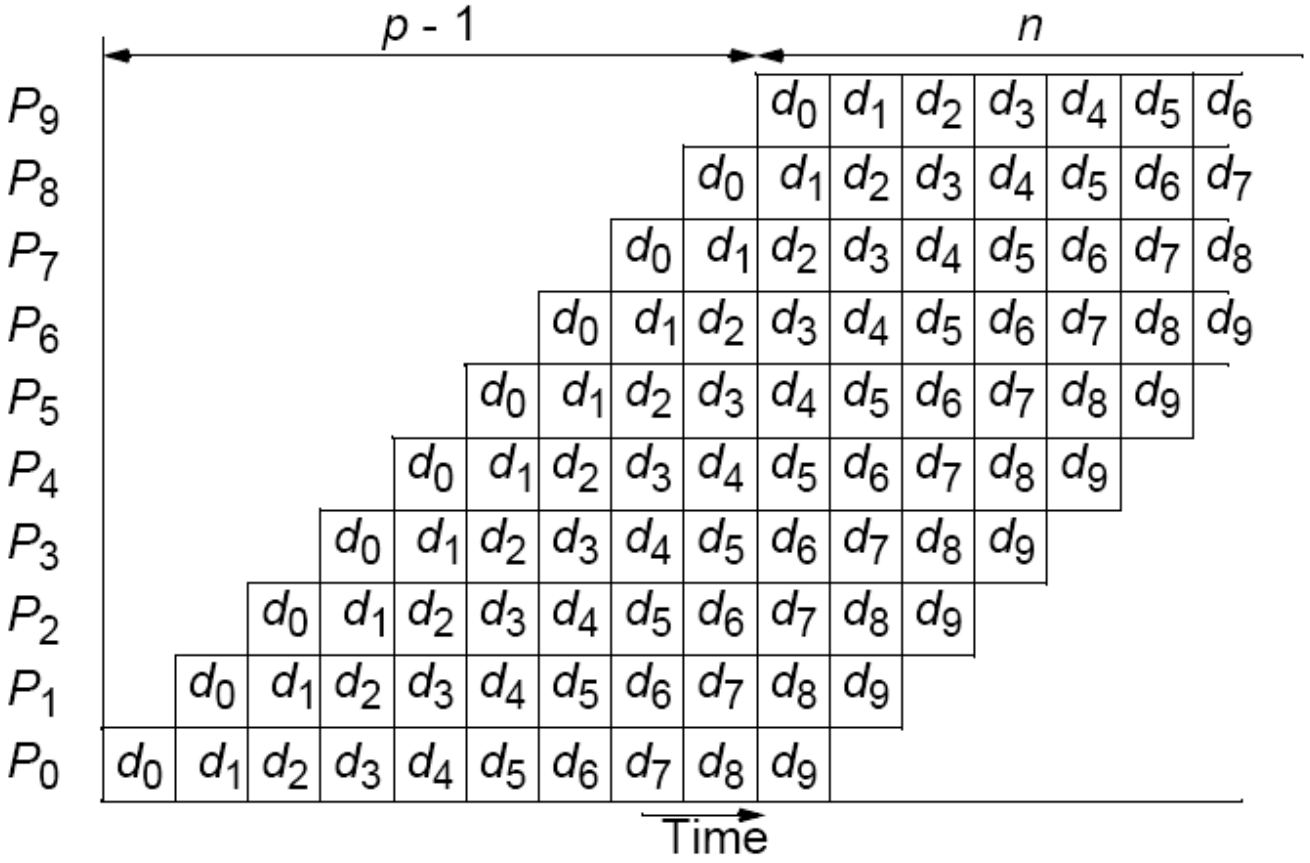


Diagrama espaço-tempo para pipeline “Tipo 2”

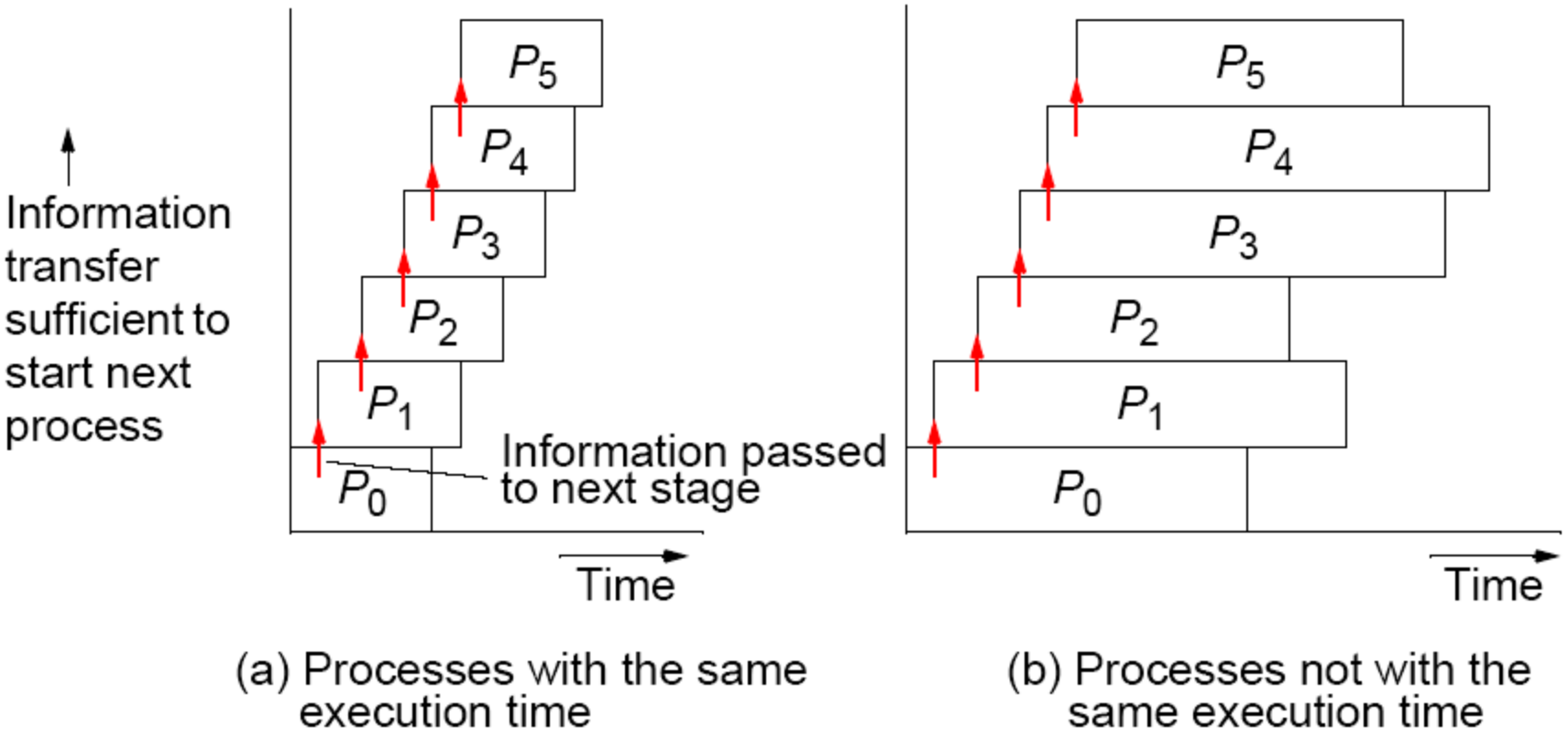


(a) Pipeline structure



(b) Timing diagram

Diagrama espaço-tempo para pipeline “Tipo 3”

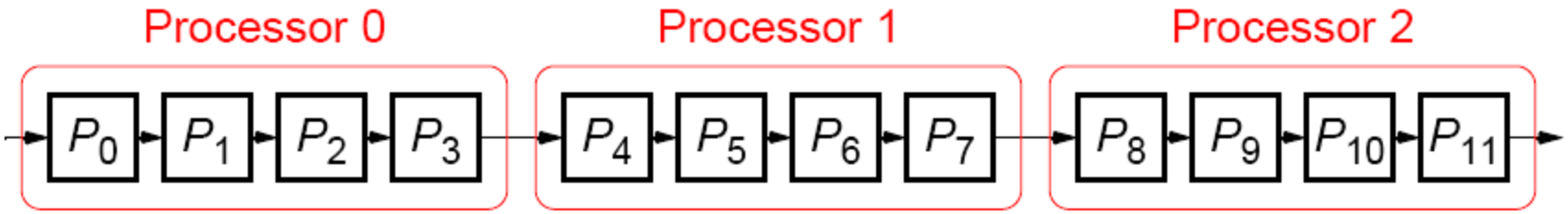


(a) Processes with the same execution time

(b) Processes not with the same execution time

Processamento em pipeline quando há informação que passa para o próximo antes do anterior terminar.

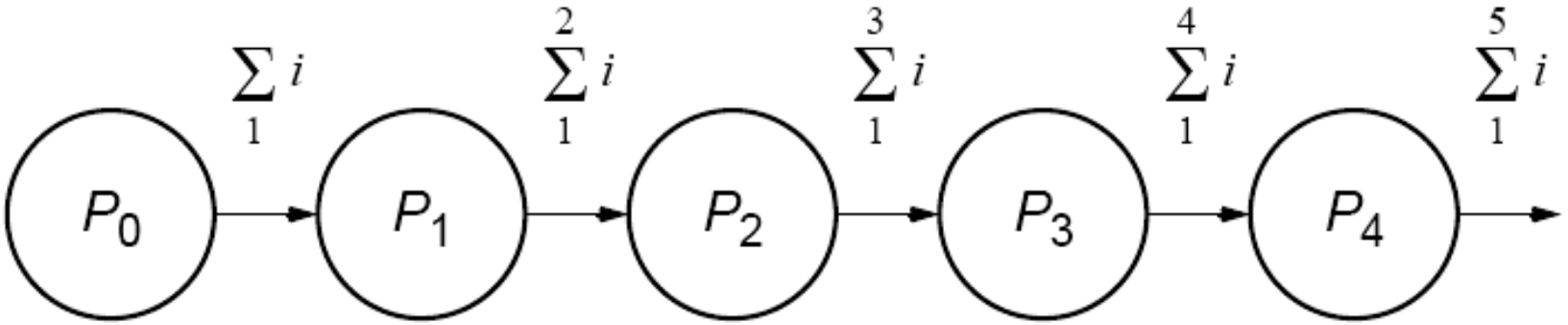
Se o número de etapas é maior que o número de processadores, um grupo de etapas pode ser atribuído a cada processador.



Exemplos de soluções pipeline

Exemplo 1

Somatório



Computação pipeline tipo 1

Código para o processo i:

```
recv(&accumulation, Pi-1);
accumulation = accumulation + number;
send(&accumulation, Pi+1);
```

Processo *P*0:

```
send(&number, P1);
```

Último processo, *Pn*-1:

```
recv(&number, Pn-2);
accumulation = accumulation + number;
```

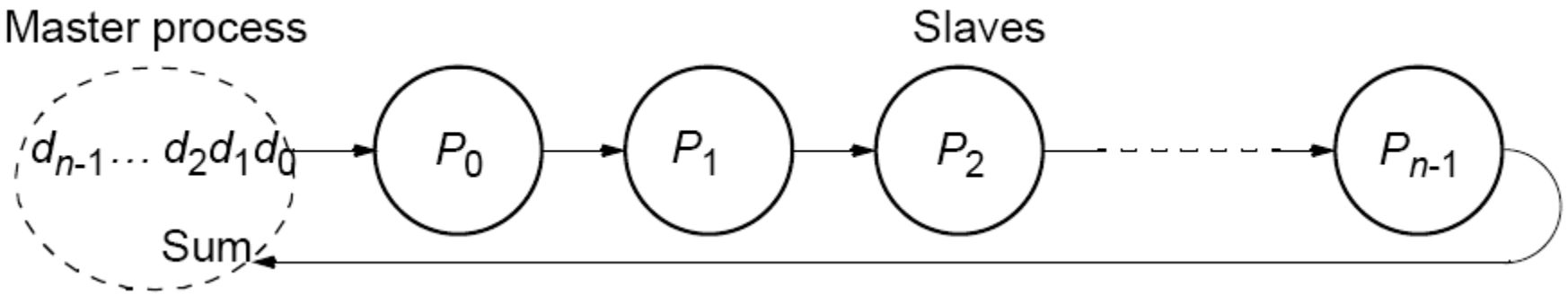
SPMD program

```
if (process > 0) {
        recv(&accumulation, Pi-1);
        accumulation = accumulation + number;
}
if (process < n-1)
        send(&accumulation, P i+1);
```

O resultado final está no último processo

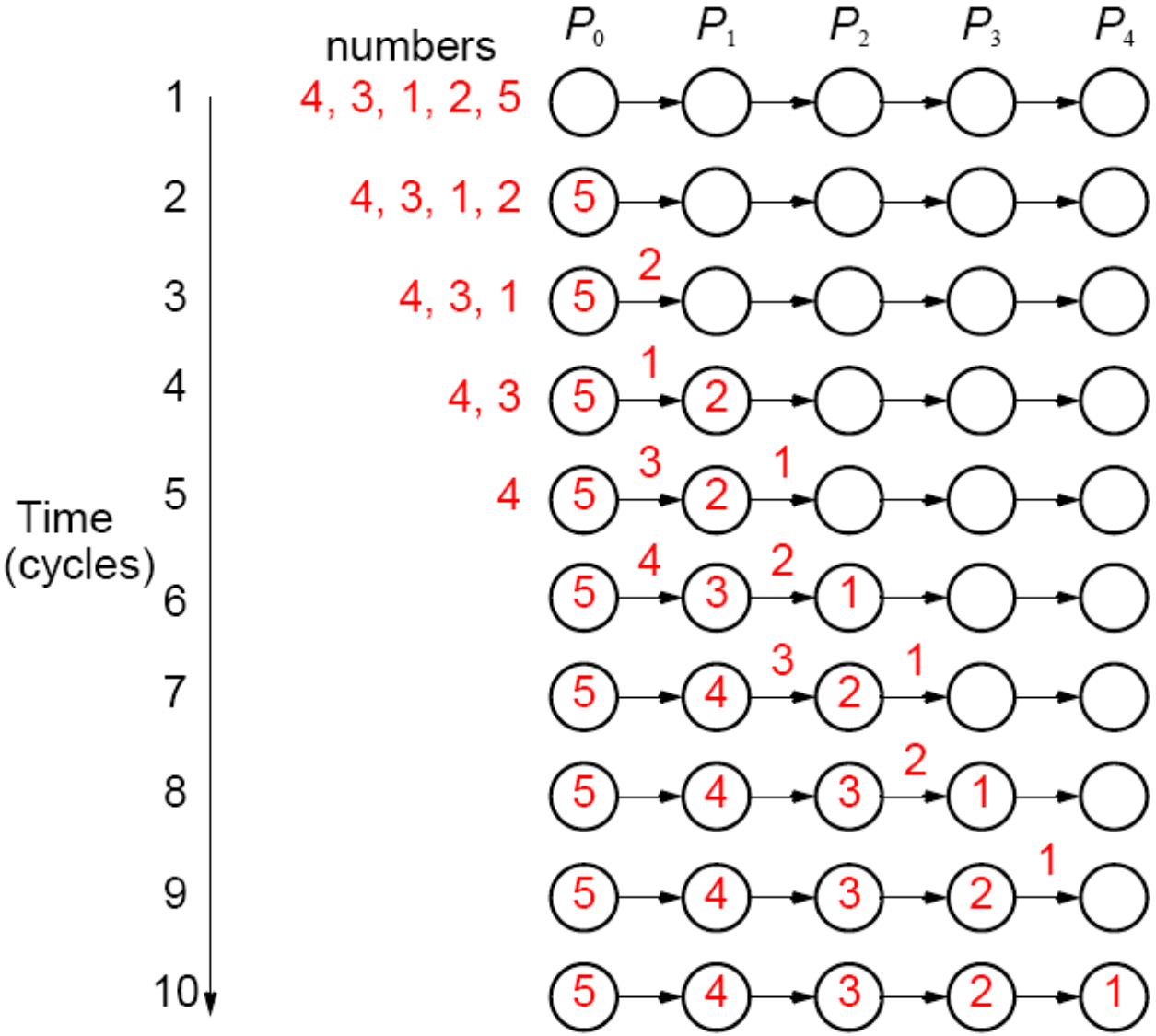
Em vez da operações de adição, outras operações podem ser feitas.

Processo master e configuração em anel:

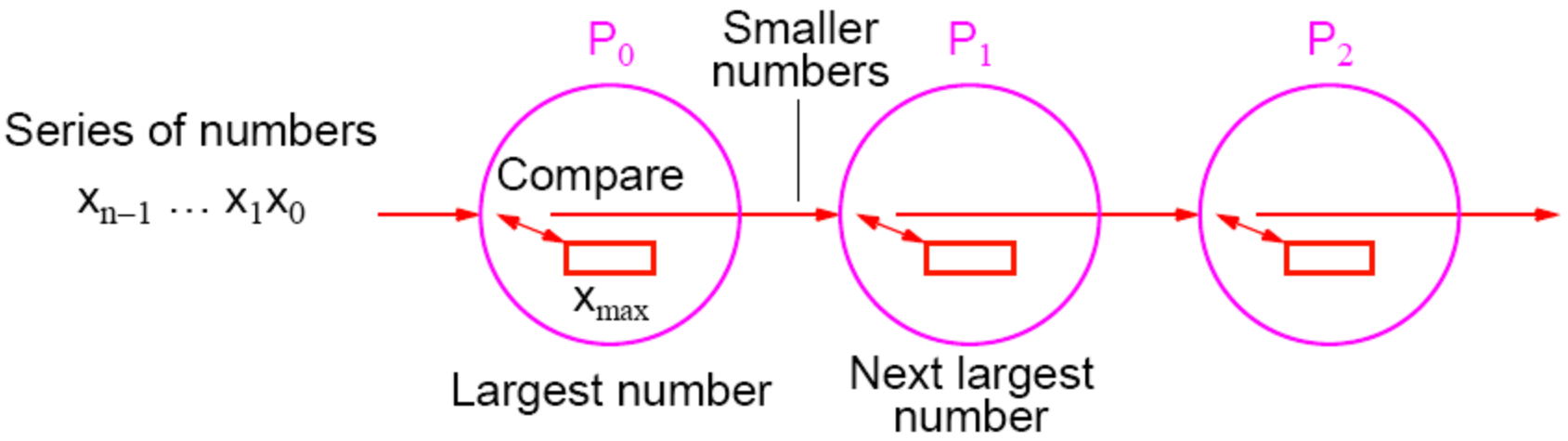


Ordenação

Versão paralela de ordenação por inserção:



Pipeline para ordenar por inserção



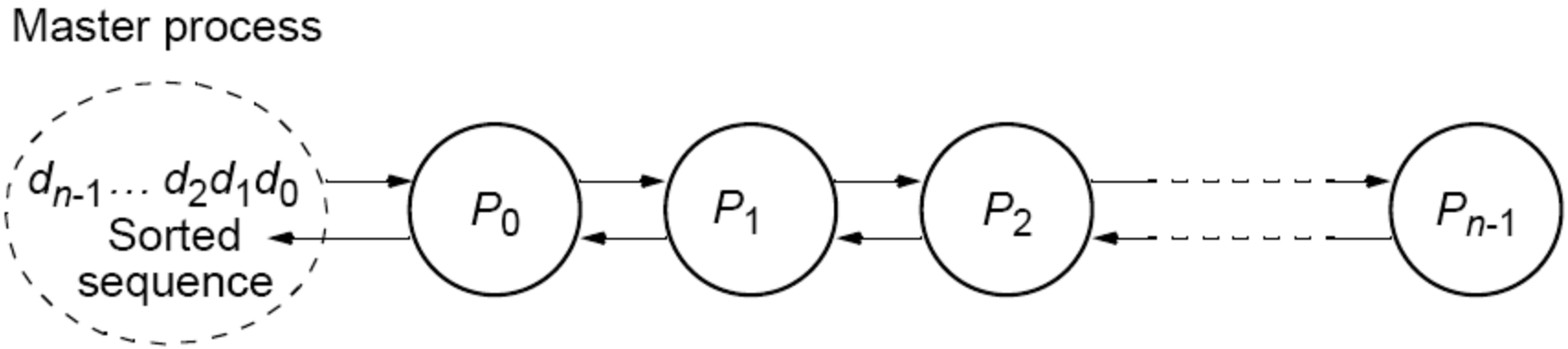
Computação pipeline tipo 2

Código para processo *Pi:*

```
recv(&number, Pi-1);
if (number > x) {
        send(&x, Pi+1);
        x = number;
} else send(&number, Pi+1);
```

Com *n* números, o processo *i* aceita *n-i* números e passa à frente *n-i-1* números.

Ordenação por inserção com resultados devolvidos ao master usando comunicação bidirecional



“”i==rank””

Pseudo-código

```
right_procno=n-i-1;
recv(&x, P_{i-1});
for (j = 0; j < right_procno; j++) {
    recv(&number, P_{i-1});
    if (number > x) {
                send (&x, P_{i+1});
                x = number;
    }
     else send (&number, P_{i+1});
}
```

Cada Ciclo

1 Recv

1 Send

Cada Ciclo

1 Comparação e 1 Troca

```
send (&x, Pi-1);
for (j = 0; j < right_procno; j++) {
  recv(&number, Pi+1);
  send (&number, Pi-1);
}
```

Envio dos dados ordenados para esquerda – para o processo mestre

Exemplo 3 : Geração de números primos (Crivo de Eratóstenes)

• Para encontrar os números primos entre 2 e *n*, gera-se a série de todos os números inteiros até *n.*

• O número 2 é o primeiro número primo e todos os múltiplos de 2 são removidos da lista, pois não podem ser primos.

• Considera-se o próximo número da lista e removem-se os seus múltiplos até chegar a *n.*

•Somente se analisam os números até $\sqrt{n}$, porque os números maiores que são múltiplos de algum número menor que $\sqrt{n}$ já foram examinados.

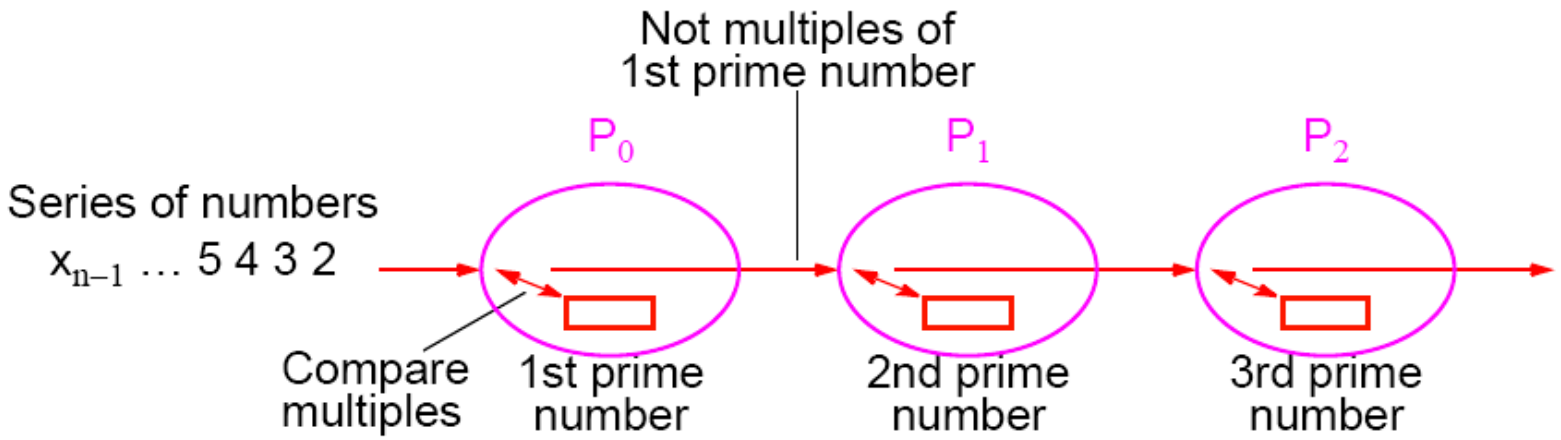
Código sequencial

```
for (i =2; i < n; i++)              // inicializar array – todos os numeros como primos
    prime[i] = 1;                   // 1→ o numero i é primo

for (i =2; i < =sqrt_n; i++)        // para cada número
   if (prime[i] == 1)               // se for primo

        for (j = i + i; j < n; j = j + i)  // eliminar múltiplos deste primo
            prime[j] = 0;               // eliminação: 0→ não é primo
```

Exemplo 3 : Geração de números primos
Crivo de Eratóstenes



Computação pipeline tipo 2

Pseudo-código

- **A sequência de inteiros é gerada e alimenta a primeira etapa do pipeline. Esta etapa elimina todos os múltiplos de 2 e passa os números restantes para a segunda etapa, …**

- **Para cada processador P_i:**

  recv(&x, P_{i-1});

  // repetir para cada número

  recv(&number, P_{i-1});

  if ((number %x) != 0) send (&number, P_{i+1});

Pseudo-código

- Como a quantidade de números não é a mesma para cada processador, utiliza-se uma mensagem de finalização:

```
recv(&x, P_{i-1});
for (j= 0; j < n; j++) {
  recv(&number, P_{i-1});
  if (number == terminator) break;
  if (number % x ) != 0) send (&number, P_{i+1});
}
```

- Otimização : considerar só números impares

Resolver um sistema de equações lineares

Na forma triangular superior

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad \ldots \quad + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

$$\vdots$$

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 = b_1$$

$$a_{0,0}x_0 = b_0$$

Com a's e b's contantes e x's as variáveis a calcular

Resolução por substituição

Calcula-se primeiro o x_0 a partir da última equação:

$$x_0 = \frac{b_0}{a_{0,0}}$$

O valor obtido para x_0 é substituído na equação seguinte para calcular x_1:

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

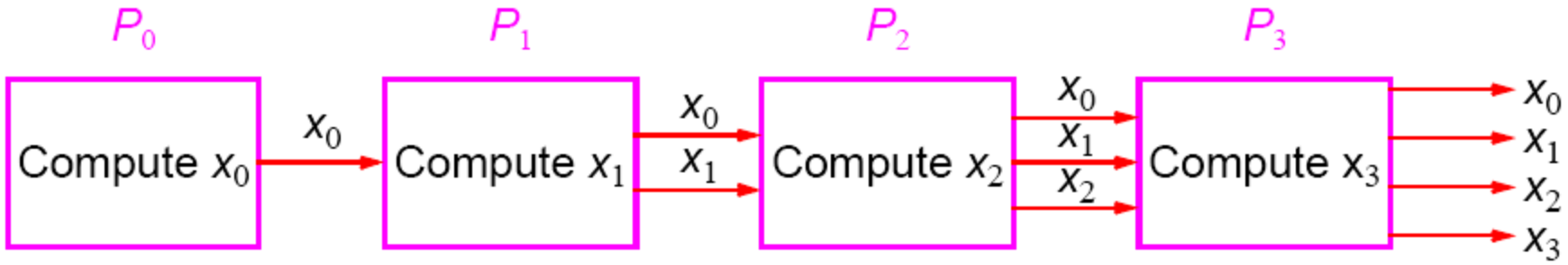
A assim por diante:

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

...

Solução em pipeline

A primeira etapa calcula x_0 e passa x_0 para o segunda etapa, esta calcula x_1 a partir de x_0 e passa ambos, x_0 e x_1 para a etapa seguinte, que calcula x_2 a partir de x_0 e x_1, e por aí adiante.



Computação pipeline “tipo 3”

O processo i ($0 < i < n$) recebe os valores x_0, x_1, x_2, …, x_{i-1} e calcula x_i *pela equação*:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j} x_j}{a_{i,i}}$$

Código Sequencial

Dadas as constantes $a_{i,j}$ *e* b_k armazenadas nos arrays **a[ ][ ]** and **b[ ]**, e o array **x[ ] onde serão guardadas as variáveis:**

```
x[0] = b[0]/a[0][0];
for (i = 1; i < n; i++) {
        sum = 0;
        For (j = 0; j < i; j++
                sum = sum + a[i][j]*x[j];
        x[i] = (b[i] - sum)/a[i][i];
}
```

Código paralelo

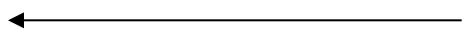
P_i $(1 < i < n)$:

```
for (j = 0; j < i; j++) {
        recv(&x[j], Pi-1);
        send(&x[j], Pi+1);
}
sum = 0;
for (j = 0; j < i; j++)
        sum = sum + a[i][j]*x[j];
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], Pi+1);
```

Podemos reescrever:

Código paralelo

```
sum = 0;
for (j = 0; j < i; j++)
{
    recv(&x[j], Pi-1);
    send(&x[j], Pi+1);
    sum = sum + a[i][j]*x[j];
}
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], Pi+1);
```



Non-Blocking para aumentar *performance*

Pipeline processing using back substitution