# Fontes principais

1. E. Cáceres, H. Mongeli, S. Song: Algoritmos paralelos usando CGM/PVM/MPI: uma introdução
   `http://www.ime.usp.br/~song/papers/jai01.pdf`
2. L. Gonda: Algoritmos BSP/CGM para ordenação, dissertação de mestrado, UFMS, 2004.

# Modelos Realísticos

# Modelos Realísticos

Anos 80: crise na área de computação paralela

▷ Vários resultados teóricos para máquinas específicas (Malhas e hipercubos).

▷ Resultados desapontadores, quando implementados em máquinas reais.

# Modelos Realísticos

Anos 90: Surgem os modelos computação de granularidade grossa

- BSP - Bulk Synchronous Parallel Model.
- CGM - Coarse Grained Multicomputers.

# Modelo BSP

# Modelo BSP

O modelo BSP (Bulk Synchronous Parallel) foi proposto por Valiant em 1990

▷ Foi um dos primeiros modelos a considerar custo de comunicação.

▷ O modelo BSP consiste de um conjunto de p processadores com memória local.

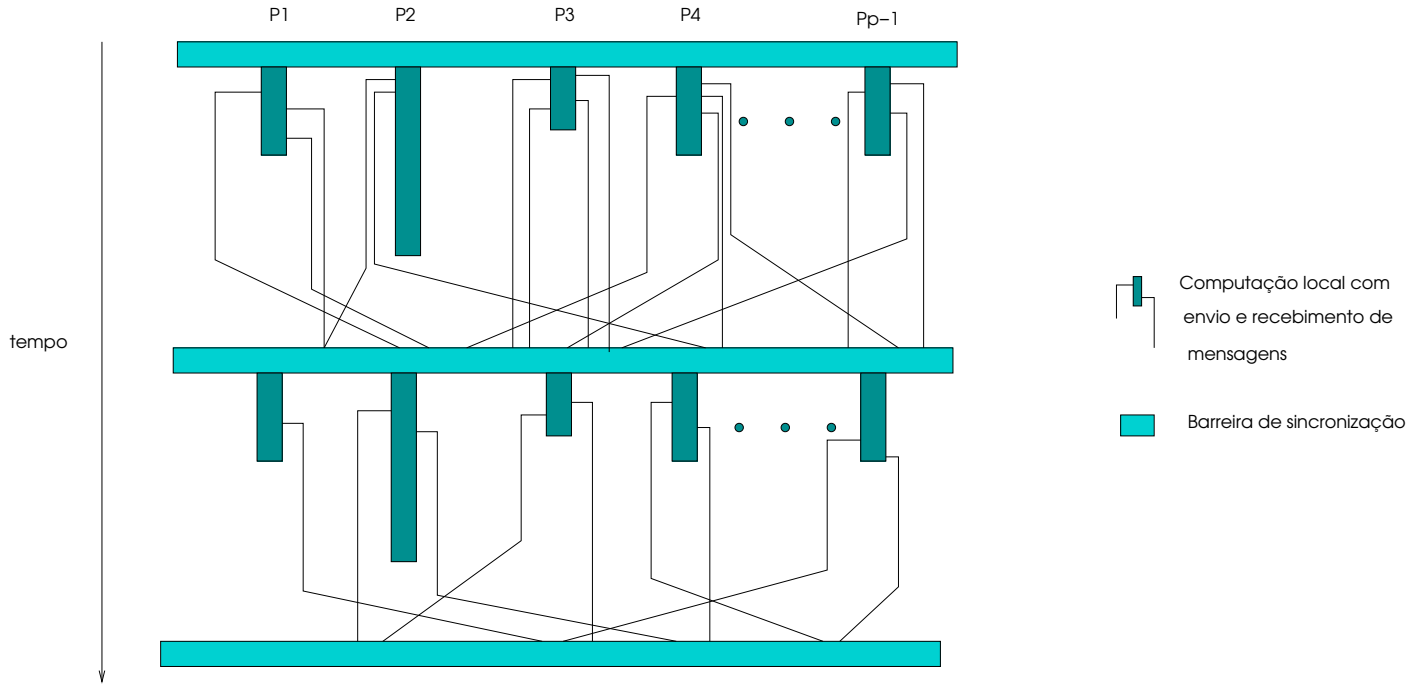
▷ A comunicação é feita por meio de rede de interconexão, gerenciados por roteador e com facilidades de sincronização global.

# Modelo BSP

Um algoritmo BSP consiste de:

- ▷ Uma sequência de superpassos separados por barreiras de comunicação.
- ▷ Em cada superpasso cada processador executa uma combinação de:
  - ◦ passos de computação (computações locais), e;
  - ◦ passos de comunicação (através da transmissão e recebimentos de mensagens).

# Modelo BSP


P1
P2
P3
P4
Pp−1
tempo
Computação local com
envio e recebimento de
mensagens
Barreira de sincronização

# Modelo BSP

O modelo possui os seguintes parâmetros:

▷ $n$ : tamanho do problema;

▷ $p$ : número de processadores disponíveis, cada um com sua memória local;

▷ $L$ : tempo mínimo de um superpasso (latência);

▷ $g$ : taxa de eficiência da computação/comunicação.

Custo (superpasso $i$): $w_i + gh_i + L$, $w_i = \{L, t_1, t_2, \cdots, t_p\}$ e $h_i\{L, c_1, c_2, \cdots, c_p\}$

Custo Total: $W + gH + L$, $W = \sum_{i=0}^{T} w_i$ e $H = \sum_{i=0}^{T} h_i$, onde T é o número de superpassos

# Modelo CGM

# Modelo CGM

O modelo CGM foi proposto por Frank Dehne e é derivado do BSP. O CGM é definido em apenas dois parâmetros:

1. $n$ : tamanho do problema
2. $p$ : número de processadores $P_1, P_2, \cdots, P_p$, cada um com uma memória local de tamanho $O(n/p)$.

# Modelo CGM

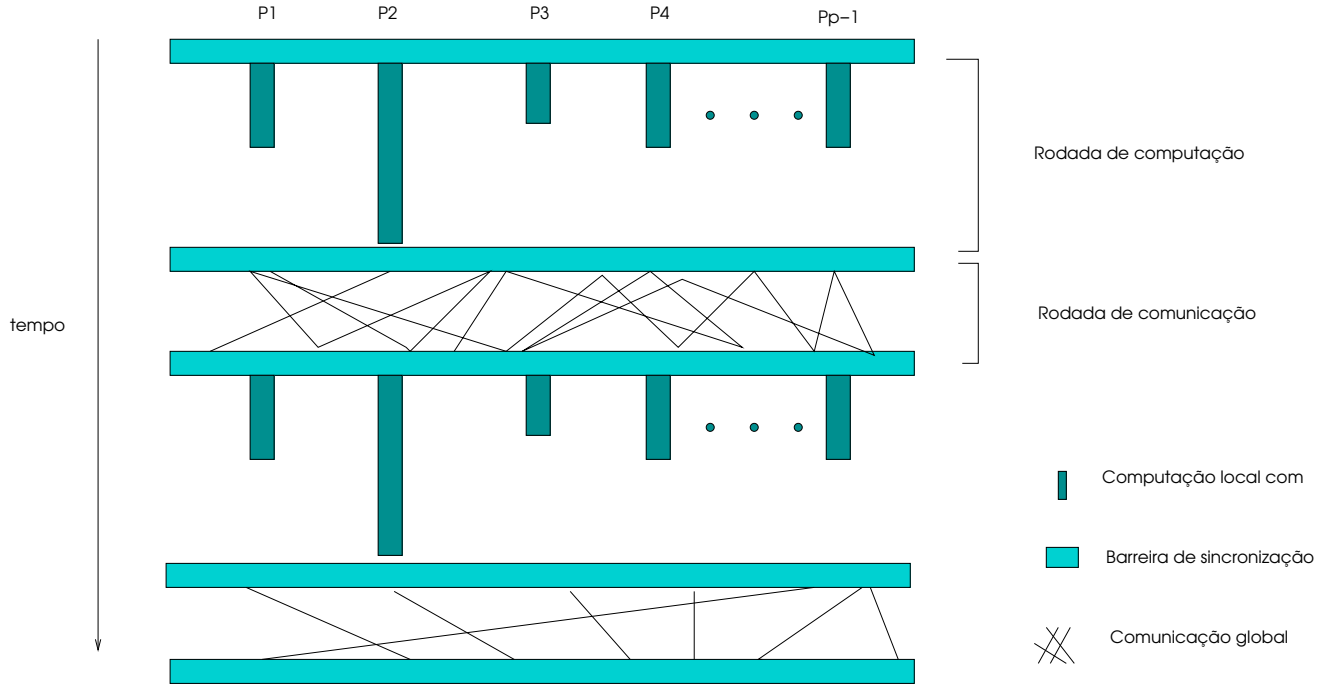
Um algoritmo CGM consiste de uma sequência alternada de **rodadas de computação** e **rodadas de comunicação** separadas por uma barreira de sincronização.

Na fase de comunicação quantidade de dados trocados por cada processador deve ser $O(n/p)$.

# Modelo CGM

O objetivo de um algoritmo CGM é minimizar o número de superpassos e a quantidade de computação local.

# Modelo CGM


P1
P2
P3
P4
Pp−1
tempo
Rodada de computação
Rodada de comunicação
Computação local com
Barreira de sincronização
Comunicação global

# Soma de um vetor no Modelo BSP/CGM

# Soma de um vetor no Modelo BSP/CGM

Seja A um vetor de ordem $n$, considere o problema de computar a soma $S = A(1) + \cdots + A(n-1)$ no modelo com $p$ processadores, onde $p << n$.

Seja $r = n/p$. A é particionado como segue: $A = (A_1, A_2, \cdots, A_{p-1})$, onde cada $A_i$ tem tamanho r.

# Soma de um vetor no Modelo BSP/CGM

Para determinar a soma S, cada processador $P_i$ computa a $i-$ésima soma parcial $s_i = A_i((i-1)r+1) + \cdots + A_i(ir)$, para $1 \leq i \leq p$, e envia $s_i$, através de uma mensagem, para o processador $P_1$, que computa o total das somas parciais.

## Soma de um vetor no Modelo BSP/CGM

**Entrada:**(1) O número do processador $i$; (2) O número $p$ de processadores; (3) O $i$-ésimo sub-vetor $B = A((i-1)r + 1 : ir)$ de tamanho r, onde $r = n/p$.

**Saída:** Processador $P_i$ calcula o valor $S = s_1 + \cdots + s_i$ e envia o resultado para $P_1$. Quando o algoritmo termina, $P_1$ terá a soma S.

# Soma de um vetor no Modelo BSP/CGM

**Algoritmo**

1 $z := B[1] + \cdots B[r]$

2 **se** $i = 1$ **então** $S := z$
**senão** $envia(z, P_1)$

3 **se** $i = 1$ **então**
**para** $i := 2$ **até** $p$ **faça**
$recebe(z, P_i)$
$S := S + z$

# Soma de um vetor no Modelo BSP/CGM

Complexidade:

▷ Passo 1: Cada $P_i$ efetua $r$ operações.

▷ Passo 2: $P_1$ efetua uma operação e os demais processadores $P_i$ enviam uma mensagem.

▷ Passo 3: $P_1$ recebe $p-1$ mensagens e efetua $p-1$ operações.

# Soma de um vetor no Modelo BSP/CGM

Complexidade:

▷ Tempo de computação: $O(n/p)$

▷ $P_1$ recebe $p-1$ mensagens, todas no mesmo superpasso(BSP) ou rodada(CGM), logo o algoritmo utiliza $O(1)$ rodadas de comunicação.

# Soma de prefixos de um vetor no Modelo BSP/CGM

# Soma de prefixos de um vetor no Modelo BSP/CGM

A solução do problema de soma de prefixos no modelo BSP/CGM é semelhante ao da soma de $n$ números.

# Soma de prefixos de um vetor no Modelo BSP/CGM

A idéia é o de dividir a entrada em $p$ (número de processadores) subconjuntos, cada um com $n/p$ elementos e distribuir esses subconjuntos entre os processadores (um subconjunto para cada processador).

## Soma de prefixos de um vetor no Modelo BSP/CGM

**Entrada:** (1) O número do processador $i$; (2) O número $p$ de processadores; (3) O $i$-ésimo sub-vetor $B = A(ir : (i+1)r - 1)$ de tamanho r, onde $r = n/p$.

**Saída:** Cada processador $P_i$ contém o valor das somas de prefixos $S[i * r + j]$, $0 \leq j \leq n/p - 1$

# Soma de prefixos de um vetor no Modelo BSP/CGM

**Algoritmo**

1 $s_i := B[0] + \cdots B[r-1]$
2 $broadcast(s_i, p_j \neq i)$
3 $S[i * r] := s_0 + \cdots + s_i$
4 $S[i * r] := S[i * r] - s_i + B[0]$
5 **para** $k := 1$ **até** $r - 1$ **faça**
$S[i * r + k] := S[i * r + k - 1] + B[k]$

# Soma de prefixos de um vetor no Modelo BSP/CGM

Complexidade:

▷ Passo 1: Cada $P_i$ efetua $r$ operações.

▷ Passo 2: Os processadores executam um *broadcast* de $s_i$ para os demais processadores. Essa comunicação pode ser feita em uma única rodada de comunicação.

▷ Passo 3: Cada processador calcula o valor da soma dos $A(0 : (i+1)r-1)$ elementos do vetor.

▷ Passo 4 e 5: Utiliza o valor computado no passo 3 para calcular as somas dos prefixos dos $A(ir : (i+1)r-1)$ elementos do vetor $A$.

# Soma de prefixos de um vetor no Modelo BSP/CGM

Complexidade:

▷ Tempo de computação: $O(n/p)$

▷ Cada processador $P_i$ tem que receber $p-1$ mensagens, todas no mesmo superpasso(BSP) ou rodada(CGM), logo o algoritmo utiliza $O(1)$ rodadas de comunicação.

# Algoritmo de ordenação no BSP/CGM

# Algoritmo de ordenação split sort no BSP/CGM

O algoritmo *split sort*, ou ordenação por divisão, consiste em dividir um conjunto de números em cestos, e distribuir os cestos de forma adequada, para que se possa ordenar $n$ números divididos em $p$ processadores, utilizando $O(1)$ rodadas de comunicação para $\frac{n}{p} \geq p^2$

# Algoritmo de ordenação split sort no BSP/CGM

Na divisão dos cestos, utilizamos a idéia de calcular um conjunto de separadores (*splitters*), denominados de $p - quartis$, baseado no cálculo de medianas de um conjunto de elementos.

## Algoritmo de ordenação split sort no BSP/CGM

**Entrada:** (1) Um vetor $A$ com $n$ elementos. (2) $p$ processadores $p_0, p_1, p_2 \cdots, p_{p-1}$. (3) Os elementos do vetor A são distribuídos entre os $p$ processadores ($n/p$ elementos por processador).

**Saída:** Todos os elementos ordenados dentro de cada processador e por processador, ou seja, se $i < j$, temos que os elementos em $p_i$ são menores que os elementos pertencentes a $p_j$.

# Split sort no BSP/CGM

## Algoritmo

1 Compute um conjunto divisor $S = \{s_1, s_2, \cdots, s_{p-1}\}$
2 $broadcast(S, p_i) \triangleright p_0$ envia $S$ para todos os processadores
3 Particionar os elementos de $p_i$ em buckets $B_j^i$ de acordo com S
4 $envia(B_j^i, p_j) \triangleright$ cada processador $P_i$ envia $B_j^i$ para $p_j$, $1 \leq i, j \leq p$
5 Ordene $B_i^k = B_i^0 \cup B_i^1 \cup \cdots \cup B_i^{p-1}$

# Split sort no BSP/CGM

É fácil veriricar que este algoritmo ordena qualquer entrada, visto que não foi efetuado nenhuma restrição ao tamanho dos buckets.

Como no modelo CGM temos que cada processador tem $O(n/p)$ memória local, devemos escolher cuidadosamento o conjunto $S$, pois isso influenciará no tamanho dos buckets.

# Split sort no BSP/CGM

Vamos apresentar um algoritmo CGM para computar o conjunto S (conjunto splitter), que utiliza apenas $O(p)$ espaço de memória por processador.

O método divide a entrada em $p$ subconjuntos de mesmo tamanho como segue.

# Split sort no BSP/CGM

**Definição 1.** *A **mediana** de um conjunto ordenado de $n$ números é o $(n+1)/2$-ésimo elemento de $n$ para $n$ ímpar ou a média do $n/2$-ésimo com $(n+1)/2$-ésimo elemento para $n$ par.*

# Split sort no BSP/CGM

**Definição 2.** *Os **p-quartis** de um conjunto ordenado $A$ de tamanho $n$ são os $p-1$ elementos, de índice $\frac{n}{p}, \frac{2n}{p}, \cdots, \frac{(p-1)n}{p}$, que dividem $A$ em $p$ partes de igual tamanho.*

Os p-quartis podem ser facilmente computados de forma sequencial usando um algoritmo recursivo em tempo $O(n \log p)$

## Algoritmo p-quartis sequencial

**Entrada:** (1) Um vetor $A$ com $n$ elementos. (2) $p$ o número de quartis

**Saída:** O conjunto A dividido em p-quartis

# Algoritmo p-quartis sequencial

## Algoritmo

1 Compute a mediana de $A$
2 Usando a mediana, divida $A$ em dois subconjuntos $A_1$ e $A_2$
3 Aplique o algoritmo recursivamente, até que $p-1$ splitter sejam encontrados

## Algoritmo p-quartis no BSP/CGM

**Entrada:** (1) Um vetor $A$ com $n$ elementos. (2) $p$ processadores $p_0, p_1, \cdots, p_{p-1}$ (3) Os elementos do vetor $A$ são distribuídos entre os $p$ processadores ($n/p$ elementos por processador)

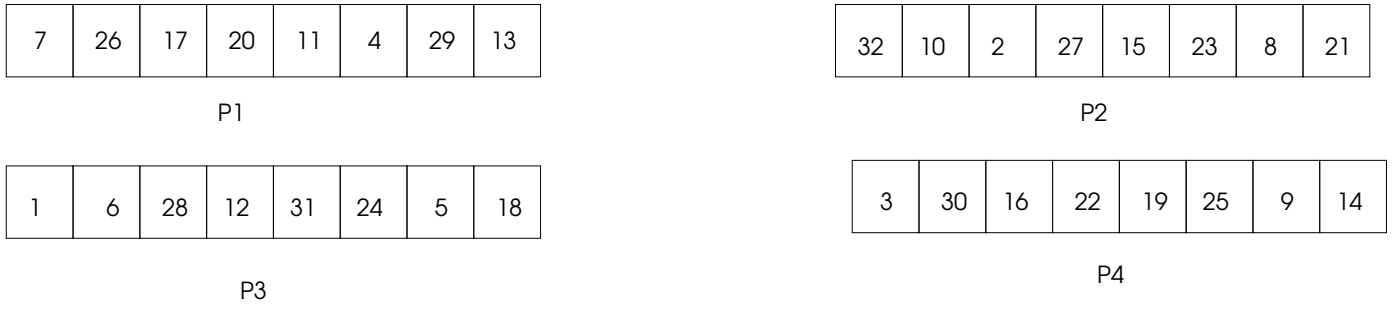
**Saída:** O conjunto A dividido em p-quartis

# Algoritmo p-quartis no BSP/CGM

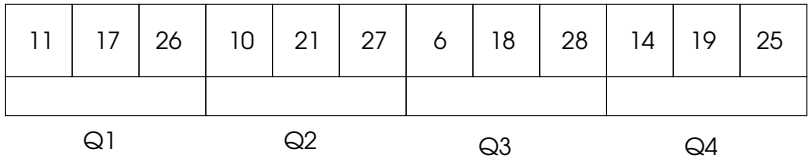
**Algoritmo**

1 $Q_i := p - quartis(A_i)$ ▷ Cada processador $p_i$ calcula
▷ sequencialmente seus $p - quartis$
2 $envia(Q_i, p_0)$ ▷ Todos os processador $p_i$ enviam $Q_i$ para $p_0$
3 **se** $i = 0$ **então**
4 $Q := Ordena(Q_0 \cup Q_1 \cup \cdots \cup Q_{p-1})$
5 $S := p - quartis(Q)$
6 $broadcast(S, p_i)$

# Algoritmo p-quartis no BSP/CGM



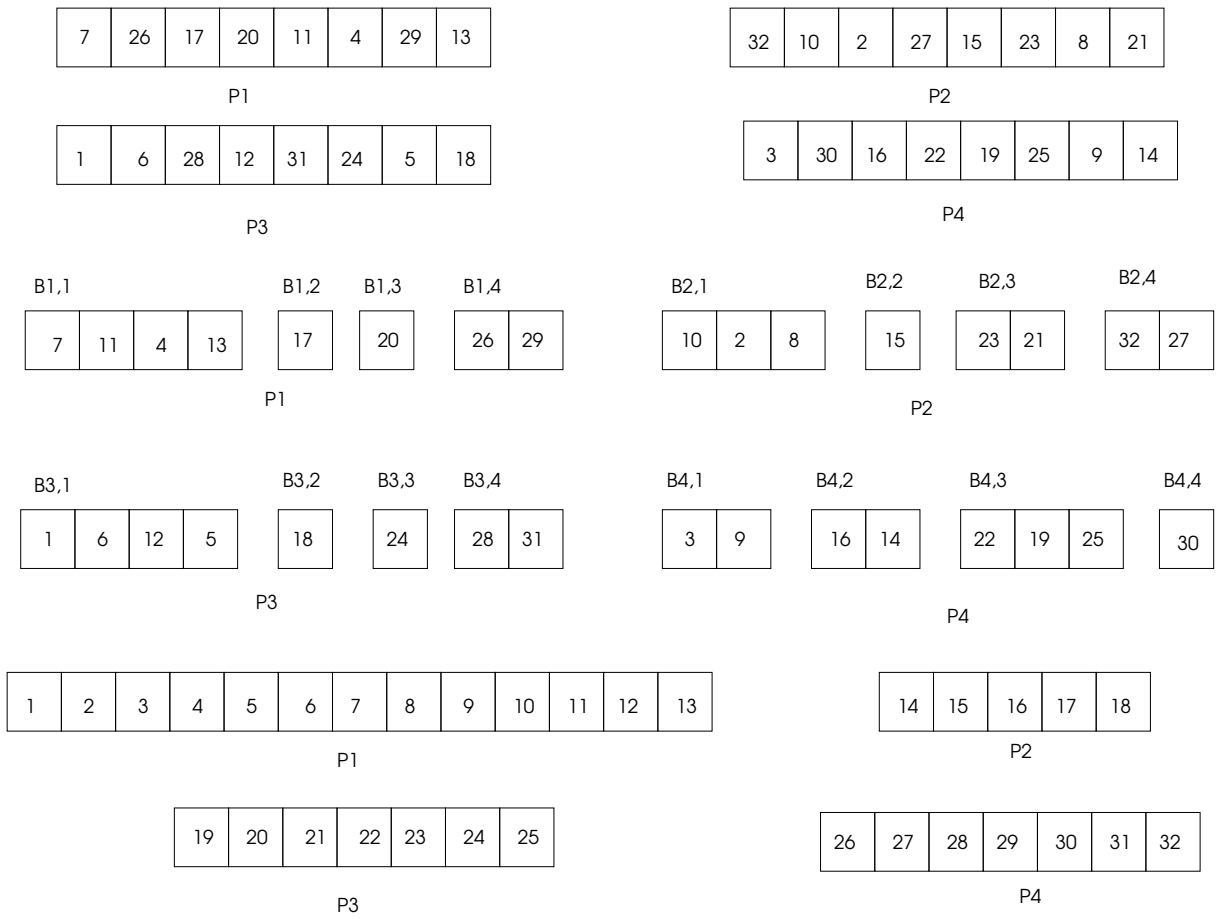

(a)




(b)


(c)

# Algoritmo split-sort no Modelo BSP/CGM



# Algoritmo p-quartis no BSP/CGM

Complexidade:

- ▷ Tempo de computação local: $O(\frac{n \log p}{p})$, onde $\frac{n}{p} \geq p^2$
- ▷ Rodadas de comunicação: $O(1)$

# Algoritmo p-quartis no BSP/CGM

Complexidade:

- ▷ Tempo de computação local: $O(\frac{n \log p}{p})$, onde $\frac{n}{p} \geq p^2$
- ▷ Rodadas de comunicação: $O(1)$

Este tempo de computação local pode ser melhorado de tal forma que $\frac{n}{p} \geq p$

# Algoritmo CGM de ordenação por inserção

# Algoritmo CGM de ordenação por inserção

**Entrada:** (1) O número de processadores; (2) O número $i$ do processador, onde $0 \leq i \leq p-1$, e (3) Um conjunto de $n$ números distintos, $S = \{s_0, s_1, \cdots, s_{n-1}\}$.

**Saída:** $S' = \{s'_0, s'_1, \cdots, s'_{n-1}\}$, onde $s_{i-1} < s_i$

**Algoritmo**

1 $r = n/p$
2 **se** $i = 0$ **então**
3 leia $S_0 = \{s_0, s_1, \cdots, s_{r-1}\}$
4 $S'_0 \leftarrow Ordene(S_0)$
5 **senão**
6 $S'_0 \leftarrow \emptyset$
7 **para** $k = 1$ **até** $p - 1$ **faça**
8 **se** $i = 0$ **então**
9 leia $S_k = \{s_{kr}, s_{kr+1}, \cdots, s_{(k+1)r-1}\}$
10 $S'_k \leftarrow Ordene(S'_{k-1} \cup S_k)$
11 $R_k \leftarrow s'_r, s'_{r+1}, \cdots, s'_{2r-1}$
12 $S'_k \leftarrow s'_0, s'_1, \cdots, s'_{r-1}$
13 $envia(R_k, P_1)$

14 **se** $i \neq 0$ **então**

15 $recebe(R_k, P_{i-1})$

16 $S'_k \leftarrow Ordene(S'_{k-1} \cup S_k)$

17 $R_k \leftarrow s'_r, s'_{r+1}, \cdots, s'_{2r-1}$

18 $S'_k \leftarrow s'_0, s'_1, \cdots, s'_{r-1}$

19 **se** $i \neq p-1$ **então**

20 $envia(R_k, P_{i+1})$

# Algoritmo CGM de ordenação por inserção

▷ Entrada de $n/p$ números de cada vez, ao invés de um número por processador.

▷ Requer $O(p)$ rodadas de comunicação.

▷ Computação local em cada rodada: ordenar $2n/p$ números.

▷ Portanto $O(\frac{n}{p} \log \frac{n}{p}) = O(\frac{n \log n}{p})$ (em cada rodada).

▷ A computação local total (considerando $O(p)$ rodadas) é $O(n \log n)$

# Características da computação "pipeline"

▷ A ordenação por inserção é um exemplo de computação sistólica ou "pipeline"

▷ Computação prossegue em *frente de ondas*.

▷ Vantagens:
- ▷ Exige comunicação com poucos vizinhos
- ▷ No início e no final poucos processadores trabalham.

# Algoritmo Bucket Sort no Modelo BSP/CGM

# Ordenação por partição ou Bucket Sort

▷ Não é baseado no paradigma de *comparar* e *trocar de posição*.

▷ Usa uma operação importante: *divisão*.

▷ Supõe que os números a serem ordenados estão uniformemente distribuídos dentro de um intervalo de 0 a $a - 1$.

# Ideia do Bucket Sort

Particionar os números (uniformemente distribuído em $[0, a-1]$) em $m$ intervalos (buckets).

Os buckets são portanto de

- de $0$ a $\frac{a}{m} - 1$,
- de $\frac{a}{m}$ a $2\frac{a}{m} - 1$,
- de $2\frac{a}{m}$ a $3\frac{a}{m} - 1$, etc.

# Algoritmo Bucket Sort no Modelo BSP/CGM

Com a distribuição uniforme no intervalo $[0, a-1]$, a quantidade de números dentro de cada bucket será aproximadamente igual.

Cada partição é então ordenada. (Novamente nos sub-buckets podem ser usados, dentro do paradigma de divisão e conquista).

# Como particionar em Buckets

Sejam $m$ buckets. O número $x$ será colocado no bucket $\lfloor \frac{x}{a/m} \rfloor$ (divisão).

Se $m$ é uma potência de $2$, i.e. $m = 2^k$, então a divisão resume em escolher os $k$ bits mais significativos do número binário $x$.

Por exemplo: $m = 2^3 = 8$. O número $x = 1100101$ é colocado no bucket $110$ (3 bits mais significativos de $x$).

# Bucket Sort sequencial

Dados $n$ números a serem ordenados, o particionamento em buckets leva tempo $O(n)$.

Cada ordenação do bucket leva tempo $O(\frac{n}{m} \log \frac{n}{m})$.

# Bucket Sort sequencial

Para ordenar todos os $m$ buckets: $O(m\frac{n}{m}\log\frac{n}{m}) = O(n\log\frac{n}{m})$.

- ▷ O tempo total será $O(n+\log\frac{n}{m})$.
- ▷ Se fizermos $k = \frac{n}{m} =$ constante, então o Bucket Sort leva tempo $O(n)$.

Lembrete: é necessário ter distribuição uniforme dos números e o uso de divisão.

# Bucket Sort paralelo

Considere $p$ processadores com memória local $O(n/p)$.

Usamos $m = p$ buckets correspondentes aos $p$ processadores.

Supondo a distribuição uniforme dos números no intervalo $0$ a $a - 1$, temos aproximadamente $n/p$ números em cada bucket, portanto cada bucket cabe num processador.

Bucket $i$ corresponde ao processador $i$, $0 \leq i \leq p - 1$.

# Bucket Sort paralelo

**Entrada:**(1) vetor $A$ de $n$ elementos. (2) $p$ processadores $p_0.p_1, \cdots, p_{p-1}$. (3) Elementos de $A$ distribuídos entre os $p$ processadores ($n/p$ elementos por processador).

**Saída:** Todos os elementos ordenados dentro de cada processador e por processador, i.e. se $i < j$, então todos os elementos de $p_i$ são menores que os de $p_j$.

# Bucket Sort paralelo

algoritmo bucket-sort

(1) Compute $max$ e $min$ de $A$; $b = (max - min)/p$;

(2) Compute um conjunto divisor

$D = \{min + b, min + 2b, \cdots, min + (p-1)b\}$;

(3) Particionar os elementos de $p_i$ em

buckets $B_i^j$ de acordo com $D$;

(4) $envia(B_i^j, p_j)$;

(5) $recebe(B_j^i, p_j)$;

(6) Ordene $\bigcup_{j=0}^{p-1} B_j^i = B_0^i \cup B_1^i \cdots B_{p-1}^i$;

# Algoritmo Bucket Sort no Modelo BSP/CGM

Cada processador tem inicialmente $n/p$ números. Cada processador determinar os buckets (0 a $p-1$) de seus números.

Cada processador $i$ envia os seus números do bucket $j$ ao processador $j$. Recebe por sua vez dos outros processadores os números do bucket $i$.

# Algoritmo Bucket Sort no Modelo BSP/CGM

Cada processador ordena seus números. (O processador $p_0$ terá os menores números, o processador $p_1$ terá os próximos menores, etc.)

- ▷Rodada de comunicação: $O(1)$
- ▷Computação local: $O(\frac{n}{p} \log \frac{n}{p})$

Lembre-se da suposição de distribuição uniforme da entrada.

# Ordenação Bitônica

# Ordenação Bitônica

Sequência bitônica: uma sequência de números $(s_0, s_1, \cdots, s_{n-1})$ que cresce (decresce) monotonicamente, atinge um único máximo (mínimo), e então decresce (cresce) monotonicamente:

$s_0 < s_1 < \cdots s_{i-1} < s_i > s_{i+1} > s_{i+2} > \cdots > s_{n-2} > s_{n-1}$,
$0 \leq i \leq n$

# Ordenação Bitônica

Também é uma sequência bitônica se existe um deslocamento cíclico $\sigma$ de $(0, 1, \cdots, n-1)$ tal que a sequência $(s_{\sigma(0)}, s_{\sigma(1)}, \cdots, s_{\sigma(n-1)})$ satisfaça a condição anterior.

# Ordenação Bitônica

Sequências bitônicas podem ser formadas pela concatenação de sequencias em ordem crescente (decrescente) e outra em ordem decrescente (crescente).

**Exemplo:** $\{2, 5, 7, 9, 8, 4, 3, 0\}$

# Propriedade de Split Bitônico

Dada uma sequência bitônica de números distintos $(s_0, s_1, \cdots, s_{n-1})$, então as sequências

$$S_{\min} = (\min(s_0, s_{\frac{n}{2}}), \min(s_1, s_{\frac{n}{2}+1}), \cdots, \min(s_{\frac{n}{2}-1}, s_{n-1})) \text{ e}$$
$$S_{\max} = (\max(s_0, s_{\frac{n}{2}}), \max(s_1, s_{\frac{n}{2}+1}), \cdots, \max(s_{\frac{n}{2}-1}, s_{n-1}))$$

são também bitônicas e, mais ainda, todos os elementos de $S_{\min}$ são menores que todos os elementos de $S_{\max}$.

# Propriedade de Split Bitônico

Exemplo: sequência bitônica

| 2 | 5 | 7 | 9 | 8 | 4 | 3 | 0 |
|---|---|---|---|---|---|---|---|
| ↑ | | | | ↑ | | | |

produz

- $S_{min} = (2, 4, 3, 0)$
- $S_{max} = (8, 5, 7, 9)$

# Ordenação Bitônica

Podemos ordenar uma sequência bitônica

$$(s_0, s_1, \cdots, s_{n-1})$$

usando sucessivos splits bitônicos.

# Ordenação Bitônica

**Exemplo:**

| passos | elementos | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 5 | 7 | 9 | 8 | 4 | 3 | 0 |
| | ↑ | | | | ↑ | | | |

# Ordenação Bitônica

**Exemplo:**

| passos | elementos | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 ↑ | 5 | 7 | 9 | 8 ↑ | 4 | 3 | 0 |
| 1 | 2 ↑ | 4 | 3 | 0 | 8 ↑ | 5 | 7 | 9 |

# Ordenação Bitônica

**Exemplo:**

| passos | elementos | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 5 | 7 | 9 | 8 | 4 | 3 | 0 |
| | ↑ | | | | ↑ | | | |
| 1 | 2 | 4 | 3 | 0 | 8 | 5 | 7 | 9 |
| | ↑ | | ↑ | | ↑ | | ↑ | |

# Ordenação Bitônica

**Exemplo:**

| passos | elementos | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 ↑ | 5 | 7 | 9 | 8 ↑ | 4 | 3 | 0 |
| 1 | 2 ↑ | 4 | 3 ↑ | 0 | 8 ↑ | 5 | 7 ↑ | 9 |
| 2 | 2 ↑ | 0 | 3 ↑ | 4 | 7 ↑ | 5 | 8 ↑ | 9 |

# Ordenação Bitônica

**Exemplo:**

| passos | elementos | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 ↑ | 5 | 7 | 9 | 8 ↑ | 4 | 3 | 0 |
| 1 | 2 ↑ | 4 | 3 ↑ | 0 | 8 ↑ | 5 | 7 ↑ | 9 |
| 2 | 2 ↑ | 0 ↑ | 3 ↑ | 4 ↑ | 7 ↑ | 5 ↑ | 8 ↑ | 9 ↑ |

# Ordenação Bitônica

**Exemplo:**

| passos | elementos | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 ↑ | 5 | 7 | 9 | 8 ↑ | 4 | 3 | 0 |
| 1 | 2 ↑ | 4 | 3 ↑ | 0 | 8 ↑ | 5 | 7 ↑ | 9 |
| 2 | 2 ↑ | 0 ↑ | 3 ↑ | 4 ↑ | 7 ↑ | 5 ↑ | 8 ↑ | 9 ↑ |
| 3 | 0 ↑ | 2 ↑ | 3 ↑ | 4 ↑ | 5 ↑ | 7 ↑ | 8 ↑ | 9 ↑ |

# Ordenação Bitônica

A técnica descrita anteriormente funciona para ordenar uma sequência bitônica.

**Pergunta:** para ordenar uma sequência de números quaisquer (não bitônica), como tranformá-la em uma sequência bitônica?

# Transformando uma sequência qualquer em Bitônica

Seja uma sequência de 8 números $(s_0, s_1, \cdots, s_7)$.

- ▷ Ordene pares de números e produza
  $(s'_0, s'_1)$ crescente e $(s'_2, s'_3)$ decrescente;
  $(s'_4, s'_5)$ crescente e $(s'_6, s'_7)$ decrescente;

- ▷ Ordene a sequência bitônica $(s'_0, s'_1, s'_2, s'_3)$ em ordem crescente. Ordene a sequência bitônica $(s'_4, s'_5, s'_6, s'_7)$ em ordem de crescente.

# Transformando uma sequência qualquer em Bitônica

Temos então uma sequência bitônica de 8 elementos:

| passos | elementos | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 8 | 3 | 4 | 7 | 9 | 2 | 0 | 5 |
| 1 | 3 | 8 | 7 | 4 | 2 | 9 | 5 | 0 |
| 2 | 3 | 4 | 7 | 8 | 5 | 9 | 2 | 0 |
| 3 | 3 | 4 | 7 | 8 | 9 | 5 | 2 | 0 |

A ordenação bitônica de uma sequência qualquer de $n$ números leva tempo $O(\log^2 n)$

# Ordenação Bitônica no modelo CGM

Vamos assumir $n$ elementos do conjunto $S$ a serem ordenados seja uma potência de dois. Assuminos que o número de processadores $p$ também é potência de dois.

A cada rodada do algoritmo, cada processador conterá $\frac{n}{p}$ elementos.

# Ordenação Bitônica no modelo CGM

**Idéia:** Divisões bitônicas sucessivas e ordenações locais, até que toda a sequência esteja ordenada.

Cada operação de divisão bitônica é sempre executada entre pares de processadores.

Após cada operação, $S_{min}$ fica armazenado em um dos processadores e $S_{max}$ em outro.

# Ordenação Bitônica no modelo CGM

**Entrada:** uma sequência de $n$ elementos, distribuídos entre os $p$ processadores, rotulados de 0 a $p-1$ com $\frac{n}{p}$ elementos em cada processador. Durante o algoritmo os processadores são re-rotulados localmente da forma $p_{g,i}$, onde $g$ é o rótulo do grupo a quem pertence e $i$ é o rótulo do processador dentro do grupo.

**Saída:** A sequência ordenada de elementos distribuída entre os $p$ processadores.

# Ordenação Bitônica no modelo CGM

**algoritmo** ordenação_bitônica

(1) Cada processador ordena seus dados localmente. Os processadores de identificador par ordena em ordem crescente e os processadores de identificador ímpar ordenam os dados em ordem decrescente.

(2) **para** $i$ **de** 1 **até** $\log p - 1$ **faça**

(2.1) $k = 2^{i-1}$

(2.2) Agrupe os processadores em $g = \frac{p}{2k}$ grupos, contendo $t = 2k$ processadores adjacente cada. Os grupos são rotulados de 0 a $g - 1$ e os processadores são identificados dentro do seu grupo através de índices de 0 a $t - 1$.

(2.3) Execute em paralelo, em cada um dos $g$ grupos, a operação de divisão bitônica entre $p_{g,j}$ e $p_{g,j+k}$, onde $0 \leq j < k$. Nos grupos de rótulo par, $S_{min}$ ficará armazenada no processador de menor índice global e $S_{max}$ ficará armazenada no processador de maior índice global. Nos grupos de rótulo ímpar, $S_{min}$ ficará armazenada no processador de maior índice global e $S_{max}$ ficará armazenada no processador de menor índice global.

(2.4) Cada processador ordena seus dados localmente, onde os processadores que pertencem aos grupo de rótulo par, ordenam os dados em ordem crescente e os processadores que pertencem aos grupo de rótulo ímpar, ordenam os dados em ordem decrescente.

(3) **para** $i$ **de** $1$ **até** $\log p$ **faça**

(3.1) $k = \frac{p}{2^i}$

(3.2) Agrupe os processadores em $g = \frac{p}{2k}$ grupos, contendo $t = 2k$ processadores adjacentes cada. Os grupos são rotulados de $0$ a a $g - 1$ e os processadores são identificados dentro do seu grupo através de índices de 0 a $t - 1$.

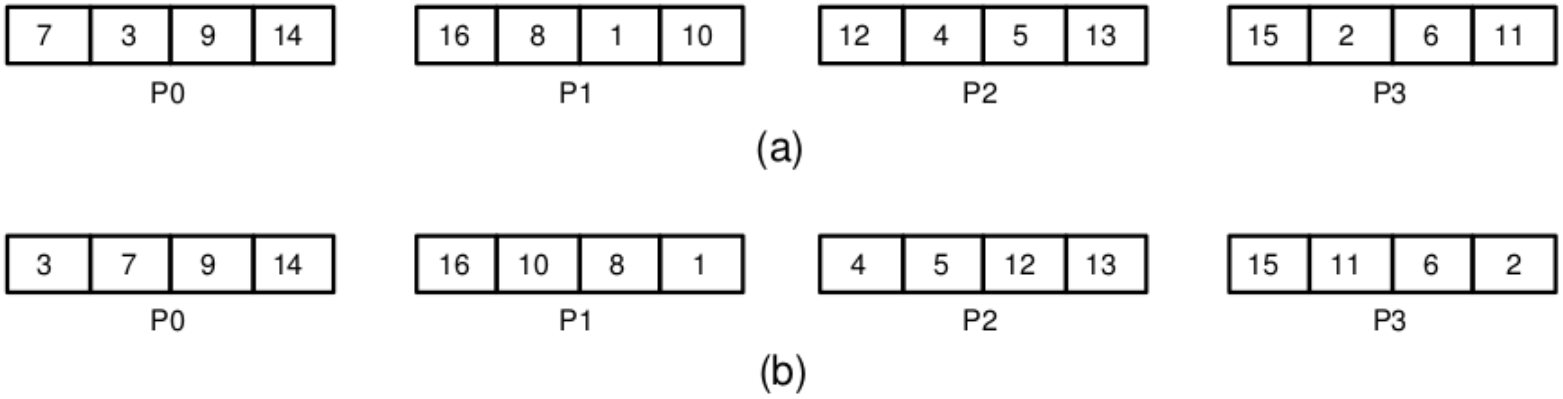
(3.3) Execute em paralelo, em cada um dos $g$ grupos, a operação de divisão bitônica entre $p_{g,j}$ e $p_{g,j+k}$, onde $0 \leq j < k$. Nos grupos de rótulo par, $S_{min}$ ficará armazenada no processador de menor índice global e $S_{max}$ ficará armazenada no processador de maior índice global. Nos grupos de rótulo ímpar, $S_{min}$ ficará armazenada no processador de maior índice global e $S_{max}$ ficará armazenada no processador de menor índice global.

(4) Cada processador ordena localmente seus elementos em ordem crescente.

# Ordenação Bitônica no modelo CGM (Exemplo)

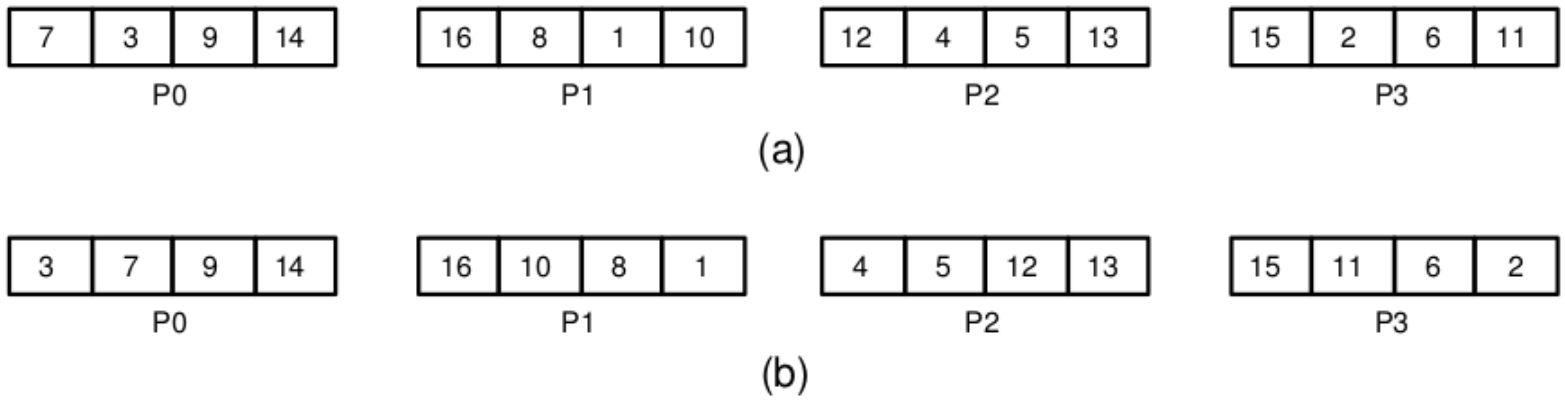
Sequência: (7, 3, 9, 14, 16, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11)

Número de processadores: 4



# Ordenação Bitônica no modelo CGM (Exemplo)

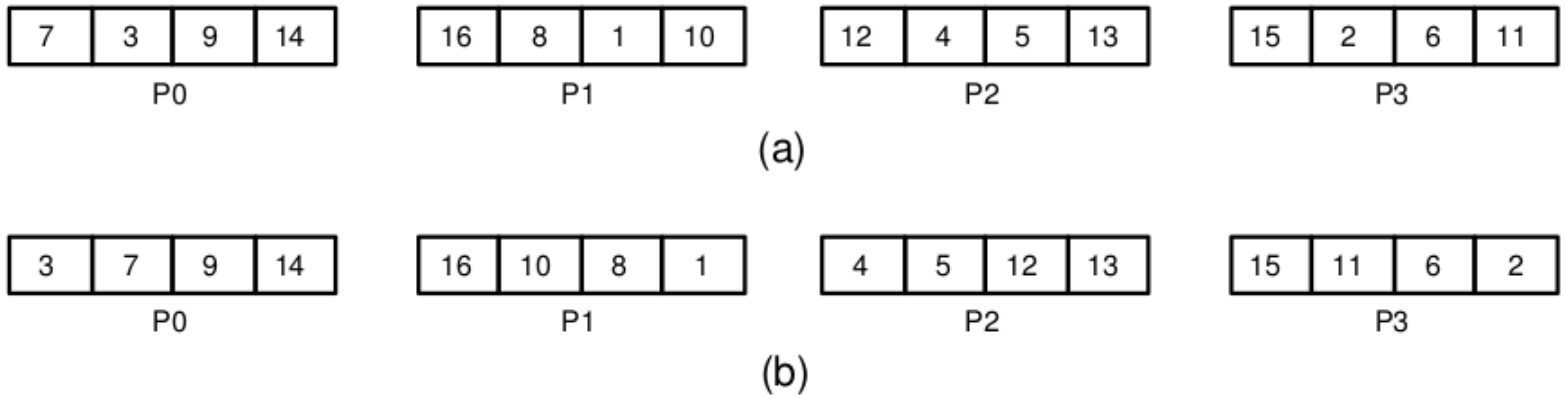
Sequência: (7, 3, 9, 14, 16, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11)




(a) distribuir os elementos pelos processadores.

# Ordenação Bitônica no modelo CGM (Exemplo)

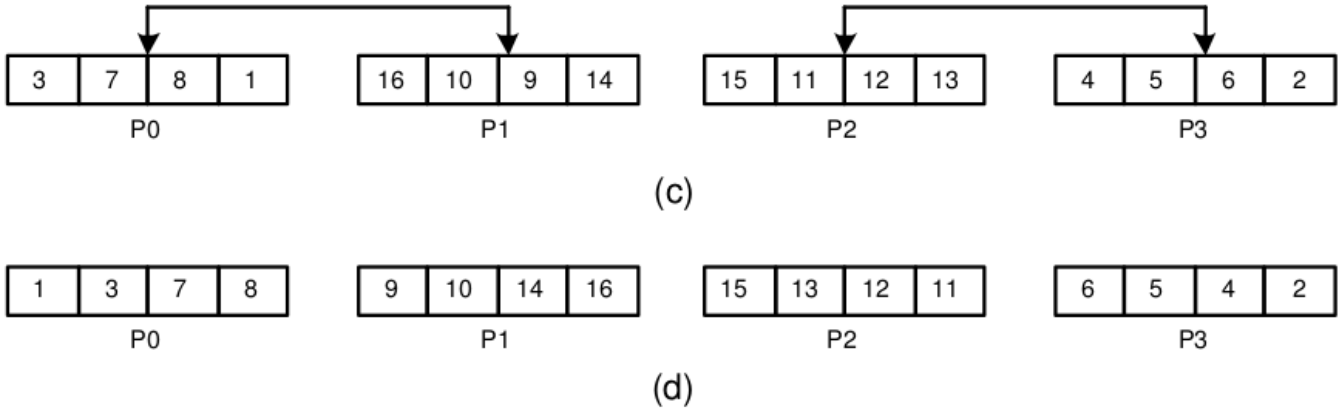
Sequência: (7, 3, 9, 14, 16, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11)




(b) ordenação local. Processadores pares ordenam em ordem crescente e processadores ímpares ordenam e ordem decrescente.

# Ordenação Bitônica no modelo CGM (Exemplo)

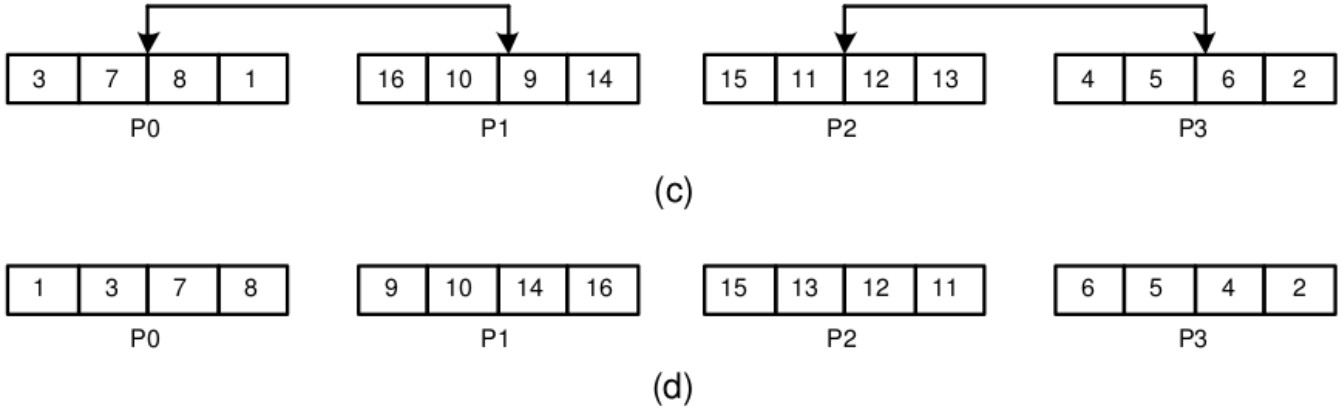
Sequência: (7, 3, 9, 14, 16, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11)




(c) Executa uma rodada de comunicação, duas operações de divisão bitônicas são executadas entre $p_0$ e $p_{0+1}$ e outra entre $p_2$ e $p_{2+1}$, pois $k = 1$.

# Ordenação Bitônica no modelo CGM (Exemplo)

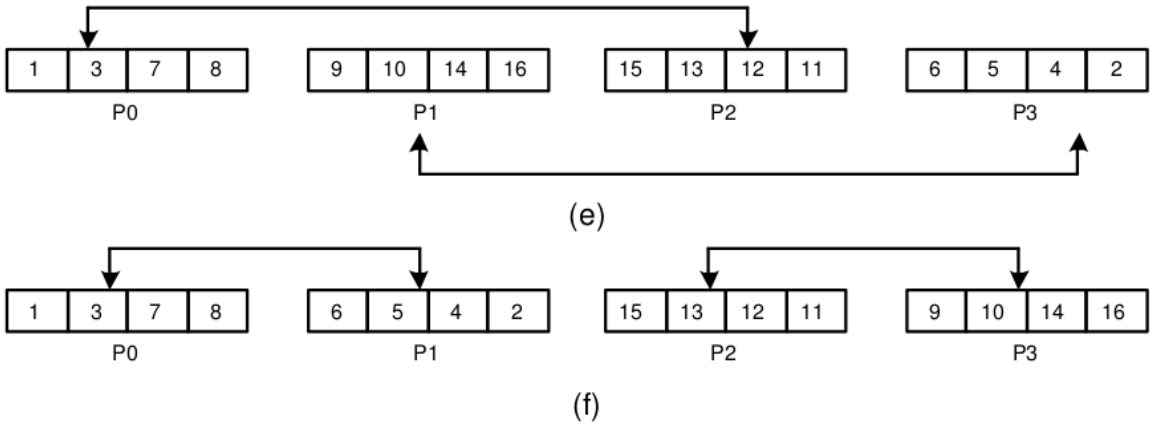
Sequência: (7, 3, 9, 14, 16, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11)




(d) ordenação local. processadores que pertencem ao grupo de rótulo par, ordenam em ordem crescente e os processadores que pertencem ao grupo de rótulo ímpar, ordenam em ordem decrescente.

# Ordenação Bitônica no modelo CGM (Exemplo)

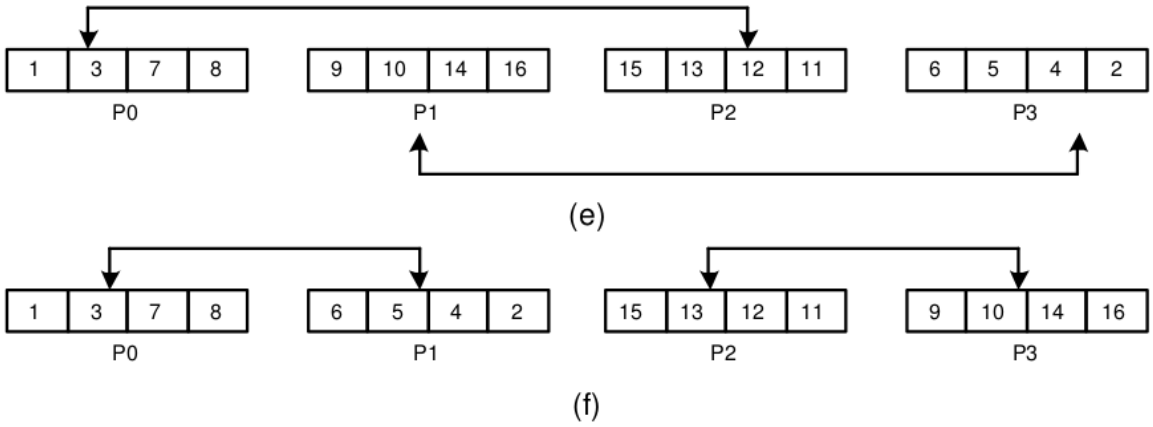
Sequência: (7, 3, 9, 14, 16, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11)




(e) Como $k = 2$ executar operação de divisão bitônica entre $p_0$ e $p_2$ e outra entre $p_1$ e $p_3$.

# Ordenação Bitônica no modelo CGM (Exemplo)

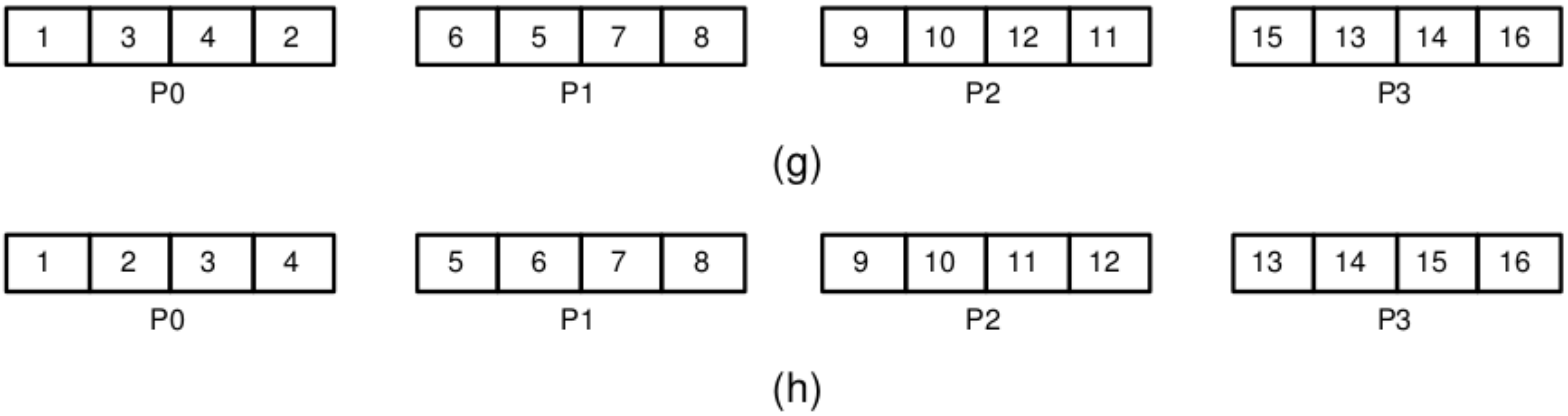
Sequência: (7, 3, 9, 14, 16, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11)




(f) $k$ passa a ser igual a 1, executar operação de divisão bitônica entre $p0$ e $p_1$ e outra entre $p_2$ e $p_3$.

# Ordenação Bitônica no modelo CGM (Exemplo)

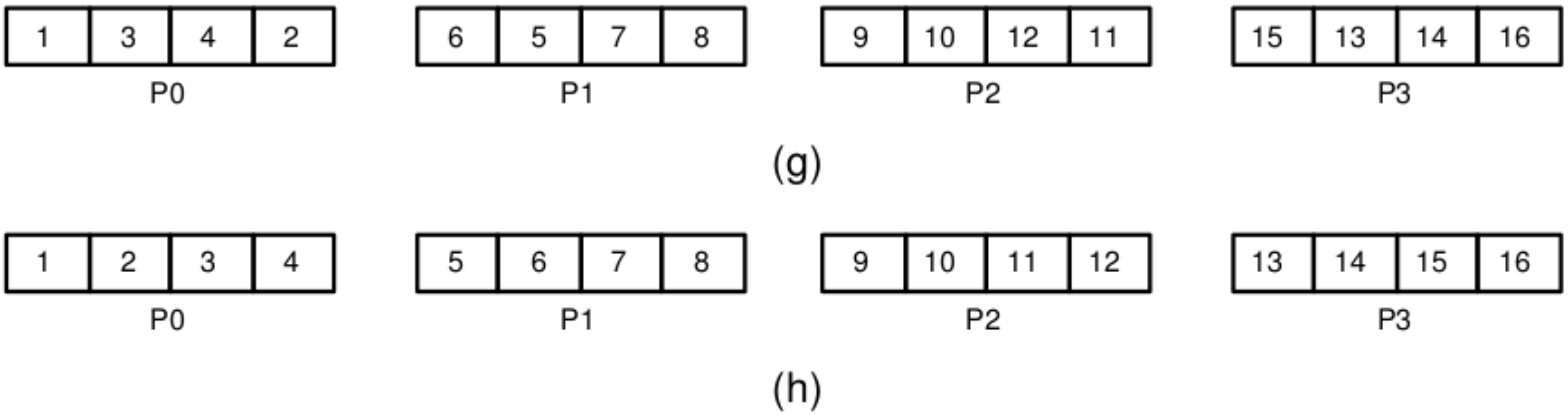
Sequência: (7, 3, 9, 14, 16, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11)




(g) Resultado das operações de divisão bitônica do passo anterior.

# Ordenação Bitônica no modelo CGM (Exemplo)

Sequência: (7, 3, 9, 14, 16, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11)




(h) Ordenação local, todos os elementos estão ordenados e distribuídos nos processadores.

# Ordenação Bitônica no modelo CGM

Características:

- Trocas de mesagens ocorrem sempre entre pares de processadores.

- Em cada rodada de comunicação são enviados $\frac{n}{p}$ elementos.

# Radix Sort para Ordenação de Inteiros

# Radix Sort para Ordenação de Inteiros

Dados $a_0, a_1, \cdots, a_n$ inteiros no intervalo $[0, m-1]$. Se $m$ é grande, podemos usar o algoritmo radix sort para ordenar os $n$ elementos inteiros.

# Radix Sort para Ordenação de Inteiros

Considere $k$ dígitos decimais. Uma forma simples é fazer radix sort com base em cada dígito (começando com o mais significativo) para separar os números em buckets de 0 a 9.

# Radix Sort para Ordenação de Inteiros

**Exemplo:** $45, 24, 39, 58, 23, 32, 25, 47, 18, 54, 51, 42, 43$

Ordenar pelo primeiro digito:

| | | | | |
|---|---|---|---|---|
| 18 | 24 | 39 | 45 | 58 |
| | 23 | 32 | 47 | 54 |
| | 25 | | 42 | 51 |
| | | | 43 | |

Ordenar pelo segundo digito:

| | | | | |
|---|---|---|---|---|
| 18 | 23 | 32 | 42 | 51 |
| | 24 | 39 | 43 | 54 |
| | 25 | | 45 | 58 |
| | | | 47 | |

# Algoritmo CGM de Chan e Dehne

# Algoritmo CGM de Chan e Dehne

Os próximos slides apresentam dois algoritmos de Chan e Dehne.

▷ algoritmo 1: ordenação por amostragem determinística (deterministic sample sorting)

▷ algoritmo 2: distribuição em grupos e aplicação do algoritmo 1.

# Algoritmo CGM de Chan e Dehne (1)

Dados $n$ inteiros no intervalo $[1..n^c]$, para constante $c$.

- ▷ Mistura split sort com radix sort na fase sequencial.
- ▷ Não há nenhuma suposição quanto a distribuição dos números.
- ▷ $p$ processadores com restrição $n/p \geq p^2$. (Essa restrição pode ser reduzida para $n/p \geq p$.)

# Algoritmo CGM de Chan e Dehne (1)

**Idéia:**

Cada processador ordena seus $n/p$ números usando radix sort.

Cada processador seleciona uma *amostra local* de $p$ números: considera os $n/p$ números já ordenados e escolhe um número a cada intervalo de $n/p^2$

# Algoritmo CGM de Chan e Dehne (1)

**Idéia: (cont.)**

Cada processador envia sua *amostra local* de $p$ números ao processador $p_1$. Note que $p_1$ vai receber $p^2$ números. Portanto memória local deve satisfazer $n/p \geq p^2$.

$p_1$ ordena todos os $p^2$ números recebidos usando radix sort.

# Algoritmo CGM de Chan e Dehne (1)

**Idéia: (cont.)**

$p_1$ considera todos os $p^2$ números ordenados e seleciona uma amostra global de $p$ números pegando um número a cada intervalo de $p$.

$p_1$ envia a *amostra global* a todos os processadores.

# Algoritmo CGM de Chan e Dehne (1)

**Idéia: (cont.)**

Cada processador $p_i$ particiona os seus $n/p$ números em $p$ Buckets $B_{i,1}, B_{i,2}, \cdots, B_{i,p}$. $B_{i,j}$ contém valores entre $(j-1)$-ésima e a $j$-ésima amostra global.

Cada processador $p_i$ envia Buckets $B_{i,j}$ para o processador $p_j$, para todo $j$.

# Algoritmo CGM de Chan e Dehne (1)

**Entrada:** $n$ inteiros no intervalo $1 \cdots n^c$, para a constante fixa $c$, armazenados em $p$ processadores, $\frac{n}{p}$ inteiros por processador. $\frac{n}{p} \geq p^2$.

**Saída:** Os inteiros ordenados.

# Algoritmo CGM de Chan e Dehne (1)

**algoritmo** Ordenação_CD

(1) Cada processador ordena localmente seus $\frac{n}{p}$ inteiros, usando o *radix sort*.

(2) Cada processador seleciona de seus inteiros localmente ordenados, uma amostra (*sample*) de $p$ inteiros com ranks $i(\frac{n}{p^2})$, $0 \leq i \leq p-1$. Denominaremos estes inteiros selecionados como amostras locais. Todas as amostras locais são enviadas ao processador $p_1$. (Note que $p_1$ receberá um total de $p^2$ inteiros). Para isso, necessitamos que $O(\frac{n}{p}) \geq p^2$).

(3) $p_1$ ordena as $p^2$ amostras locais recebidas no passo 2 (usando radix sort) e seleciona uma amostra de $p$ inteiros com ranks $ip$, $0 \leq i \leq p-1$. Chamaremos estes inteiros selecionados de amostras globais. As $p$ amostras globais são enviadas $broadcast$ a todos os processadores.

(4) Baseado nas amostras globais recebidas, cada processador $p_i$, particiona seus $\frac{n}{p}$ inteiros em Buckets $B_{i,1}, B_{i,2}, \cdots, B_{i,p}$ onde $B_{i,j}$ são inteiros locais com valores entre $(j-1)$-ésimo e $j$-ésimo elemento das amostras globais.

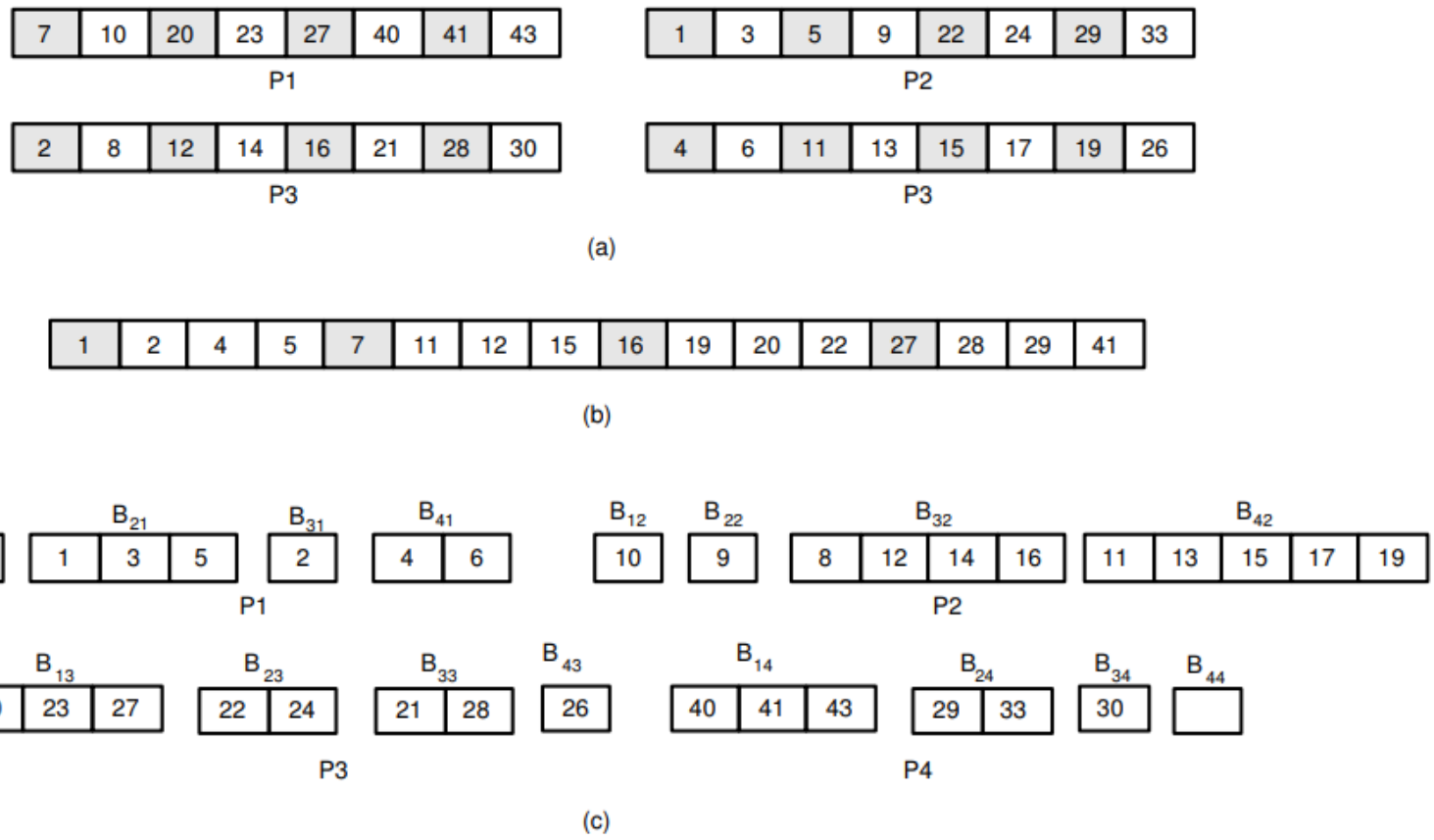
(5) Em uma (combinada) $h$-relação, cada processador $p_i$, $1 \leq i \leq p$, envia $B_{i,j}$ para o processador $p_j$, $1 \leq i \leq p$. Seja $R_j$ o conjunto dos inteiros recebidos pelo processador $p_j$, $1 \leq i \leq p$, e seja $r_i = |R_i|$.

# Algoritmo CGM de Chan e Dehne (1)

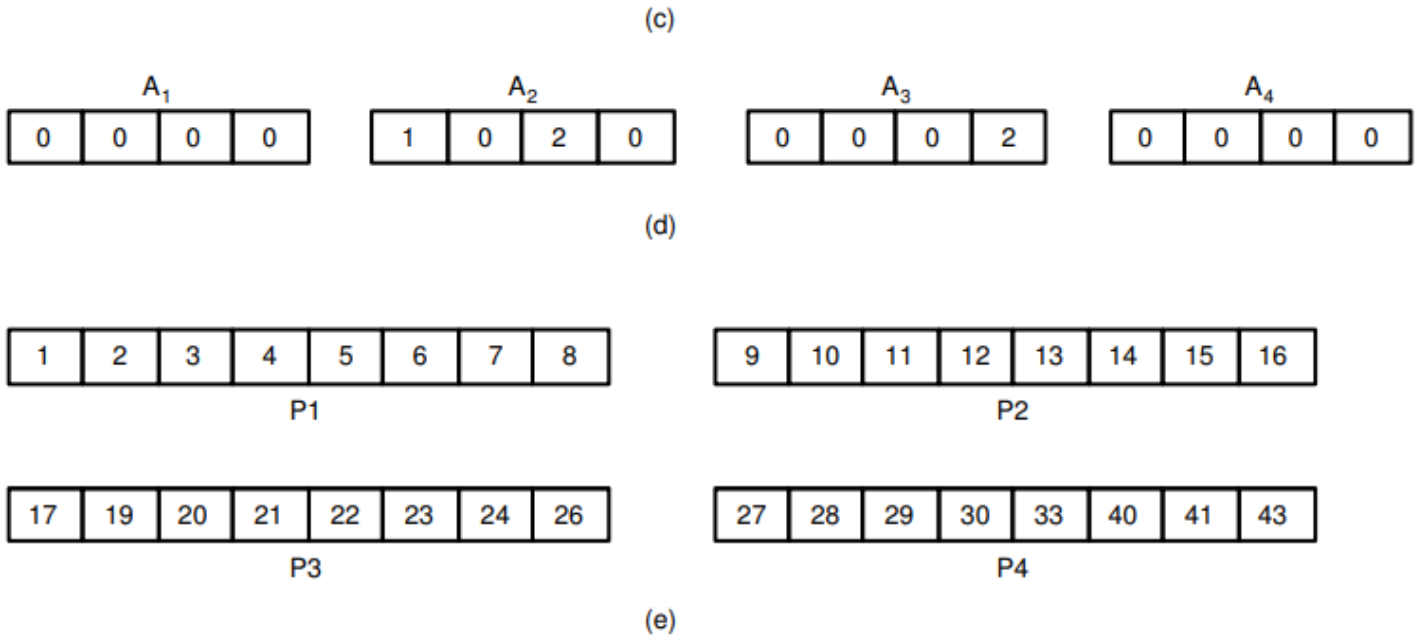
(6) Cada processador $p_i$, $1 \leq i \leq p$, ordena localmente $R_i$ usando *radix sort*.

(7) Uma operação de balanceamento (balancing shift) que distribui igualmente todos os inteiros entre os processadores sem alterar sua ordem é realizada como segue: Cada processador $p_i$, $1 \leq i \leq p$, envia $r_i$ para $p_1$. O processador $p_1$ calcula para cada $p_j$ um vetor $A_j$ de $p$ números indicando quantos de seus inteiros devem ser removidos para os respectivos processadores. Em uma $h$-relação, todo $A_j$ é enviado a $p_j$, $1 \leq j \leq p$. O balanceamento é então executado em uma $h$-relação subsequente de acordo com os valores de $A_j$.

# Algoritmo CGM de Chan e Dehne (1)



# Algoritmo CGM de Chan e Dehne (1)



# Algoritmo CGM de Chan e Dehne (1)

Usa 6 rodadas de comunicação

- ▷ sendo 2 rodadas com $h = n/p$

  Passo 5 e 7

- ▷ e 4 rodadas com $h = p^2$ na $h$-relação

  Passo 2 e 3

  Passo 7 requer duas rodadas

Computação local: $O(n/p)$.

# Algoritmo CGM de Chan e Dehne (2)

# Algoritmo CGM de Chan e Dehne (2)

**Idéia:** Particionar $p$ processadores em $\sqrt{p}$ grupos de mesmo tamanho e realizar permutações entre elementos a serem ordenados de tal forma que teremos todos os elementos do grupo $G_i$ menores que os elementos do grupo $G_j$, se $i < j$.

Para realizar a ordenação, aplicamos o algoritmo 1 (ordenação por amostragem) em cada $\sqrt{p}$ grupos. Este algoritmo utiliza a idéia de dividir os processadores em grupos para garantir que $\frac{n}{p} \geq p$

# Algoritmo CGM de Chan e Dehne (2)

**Entrada:** $n$ inteiros de $1 \cdots n^c$, para uma constante $c$, armazenada em $p$ processadores, $n/p$ inteiros por processadores. $\frac{n}{p} \geq p$.

**Saída:** Os elementos ordenados, divididos em grupos de processadores.

# Algoritmo CGM de Chan e Dehne (2)

**algoritmo** Ordenação_CD2

(1) Agrupe os $p$ processadores em $\sqrt{p}$ grupos $G_1, G_2 \cdots G_{\sqrt{p}}$.

(2) Em cada um dos grupos obtidos, aplique o algoritmo ordenacao_CD.

(3) Cada processador obtém seu menor elemento, denominado *mínimo local* e envia para $p_1$.

(4) $p_1$ ordena os elementos recebidos no passo 3 e seleciona $\sqrt{p}$ inteiros com rank $i\sqrt{p}$, denominados *divisores globais*. Em seguida, envia os $\sqrt{p}$ divisores globais para todos os processadores.

(5) Cada processador $p_i$ divide seu conjunto de $\frac{n}{p}$ inteiros em $\sqrt{p}$ cestos $B_{i,j}$, contendo os elementos entre $(j-1)$-ésimo e o $j$-ésimo elementos dos divisores globais, onde $1 \leq j \leq \sqrt{p}$.

(6) $p_i$ envia $B_{i,j}$ para um processador no grupo $G_j$, onde $1 \leq i \leq p$ e $1 \leq j \leq \sqrt{p}$. Denominaremos $R'_j$, o conjunto de inteiros enviados para o grupo $G_j$.

(7) Em cada um dos grupo $G_i$, calcula-se o tamanho do conjunto de inteiros enviados ao grupo $G_j$, $1 \leq j \leq \sqrt{p}$, denominado de $t_{i,j}$.

(8) Todos $t_{i,j}$ e os tamanhos de $B_{i,j}$ são enviados para um processador em cada grupo, denominado de processador principal.

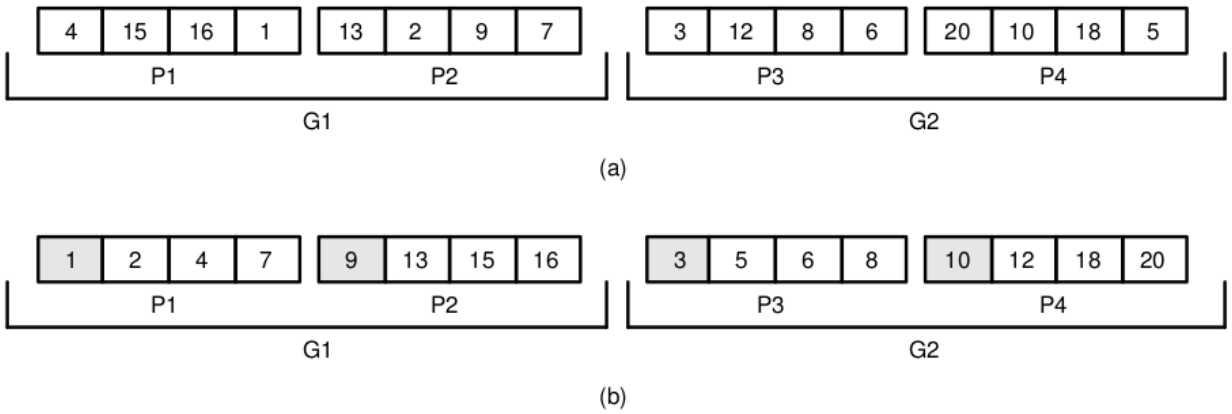
(9) Em cada grupo, o processador principal computa uma escala de roteamento para realizar balanceamento na distribuição dos dados entre os processadores do grupo e envia aos demais processadores do grupo.

(10) Usando a ordenação_CD, cada grupo $G_j$, $1 \leq j \leq \sqrt{p}$, ordena $R'_j$.

(11) Uma operação de balanceamento (balancing shift) distribui todos os inteiros igualmente entre os processadores sem mudar sem mudar sua ordem. Este passo é análogo ao passo 7 do algoritmo ordenação_CD, porém utilizando duas fases como nos passos 7, 8 e 9 deste algoritmo.

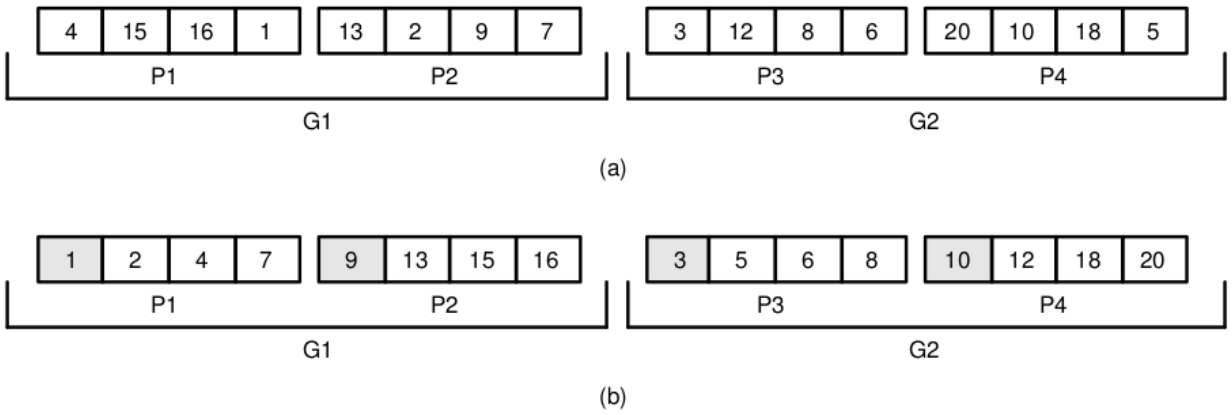
**– fim do algoritmo**

# Algoritmo CGM de Chan e Dehne (2) - Exemplo



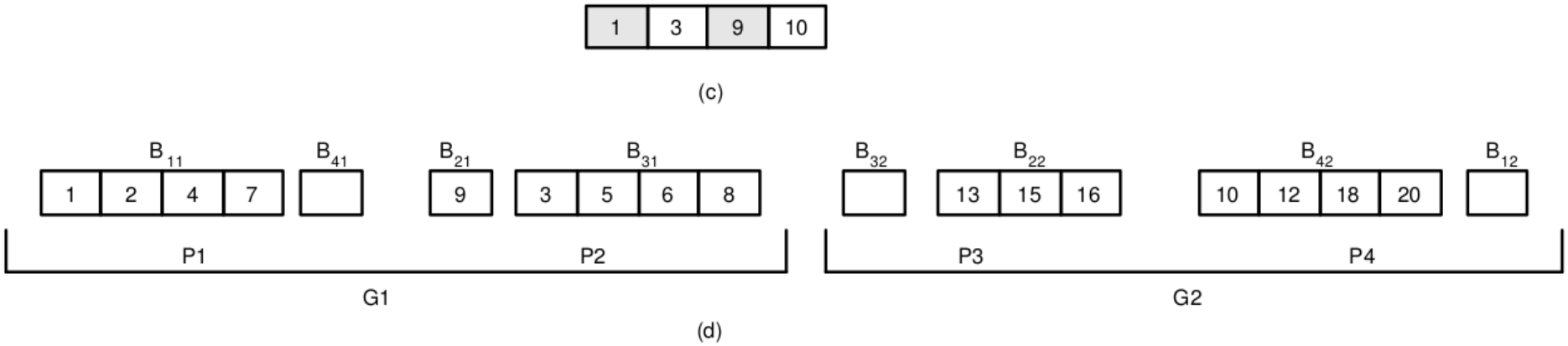

(a) $G_1$, composto por $p_1$ e $p_2$; e $G_2$ composto por $p_3$ e $p_4$. Depois, o algoritmo ordenação_CD é aplicado em cada grupo separadamente.

# Algoritmo CGM de Chan e Dehne (2) - Exemplo



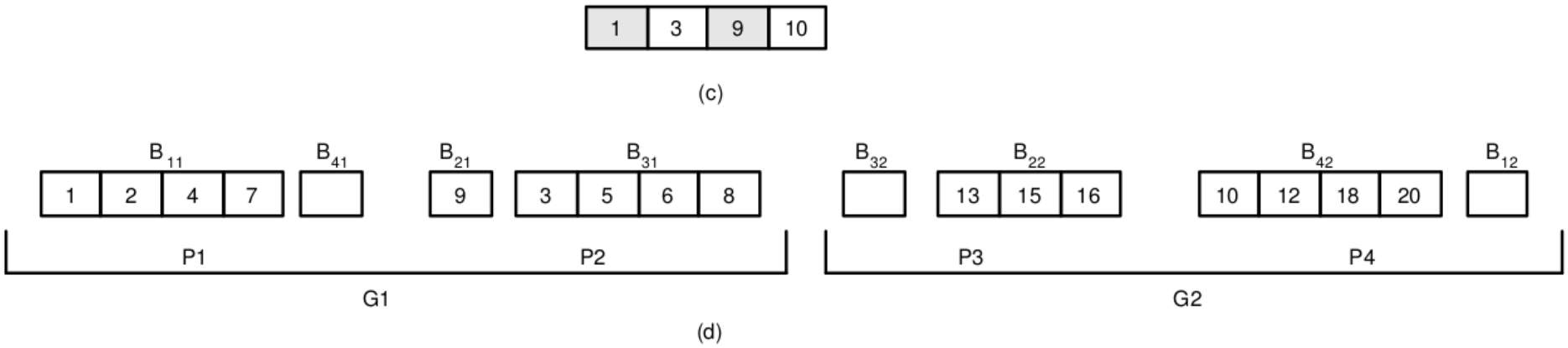

(b) Cada processador seleciona o *mínimo local* e envia para o processador $p_1$.

# Algoritmo CGM de Chan e Dehne (2) - Exemplo



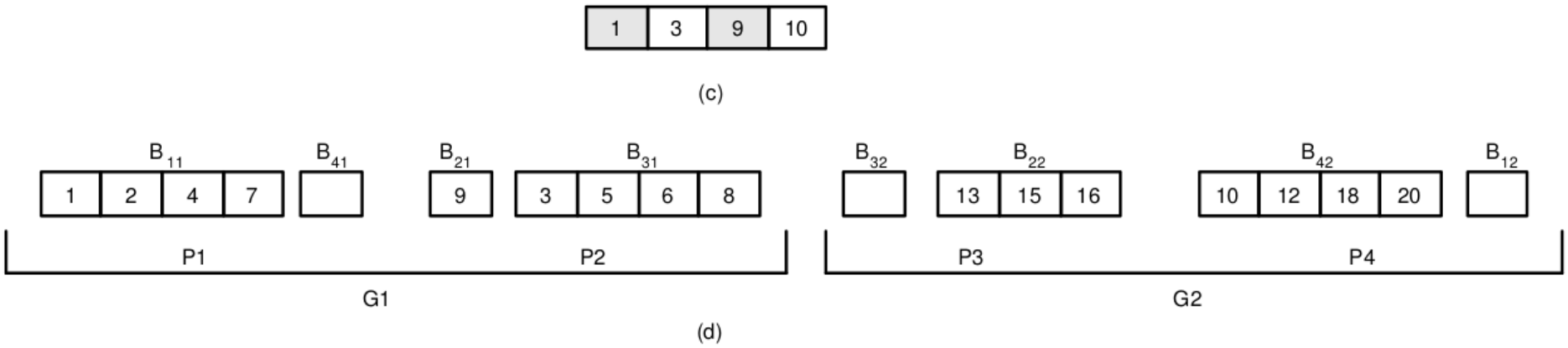

(c) o processador $p_1$ ordena os elementos recebidos e escolhe uma amostra de $\sqrt{p}$ inteiros, e envia para todos os processadores.

# Algoritmo CGM de Chan e Dehne (2) - Exemplo



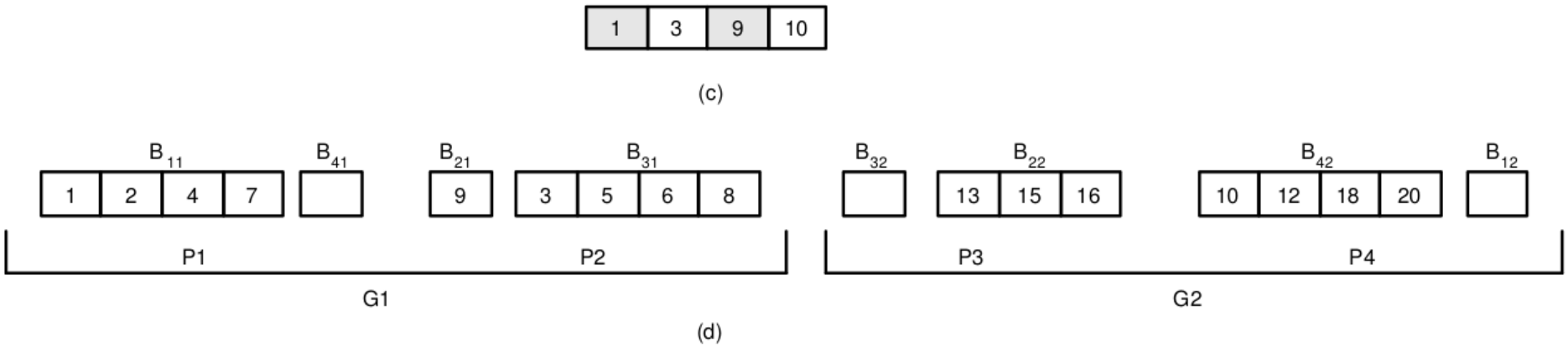

(d) cada processador divide seus $\frac{n}{p}$ dados em cestos $B_{i,j}$, envia para um processador no grupo $G_j$.

# Algoritmo CGM de Chan e Dehne (2) - Exemplo



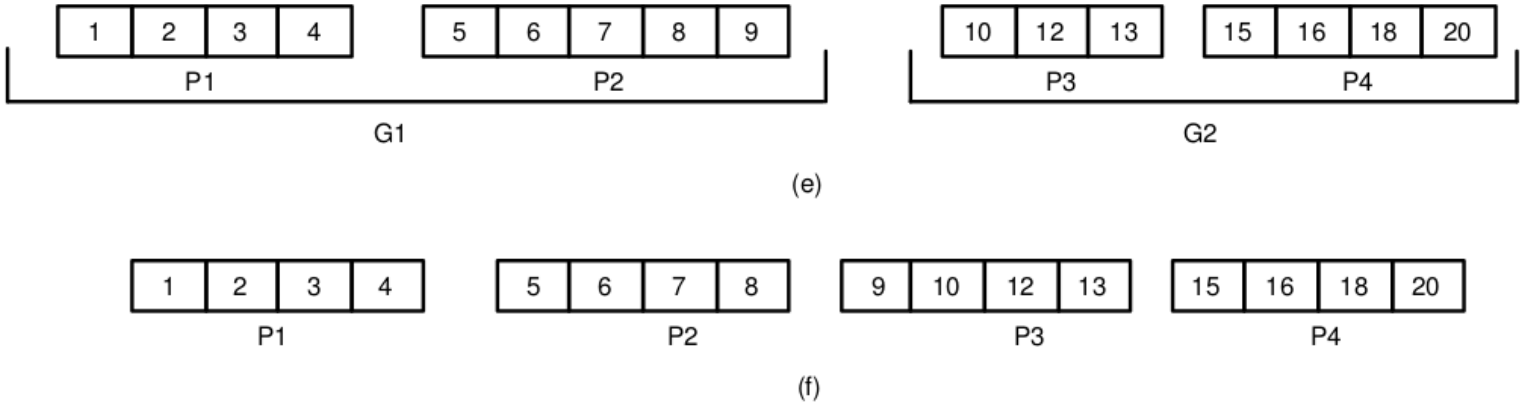

(d) $p_1$ permaneceu com o cesto $B_{1,1}$ e enviou o cesto $B_{1,2}$ para o processador $p_4$, que pertence ao grupo $G_2$.

# Algoritmo CGM de Chan e Dehne (2) - Exemplo



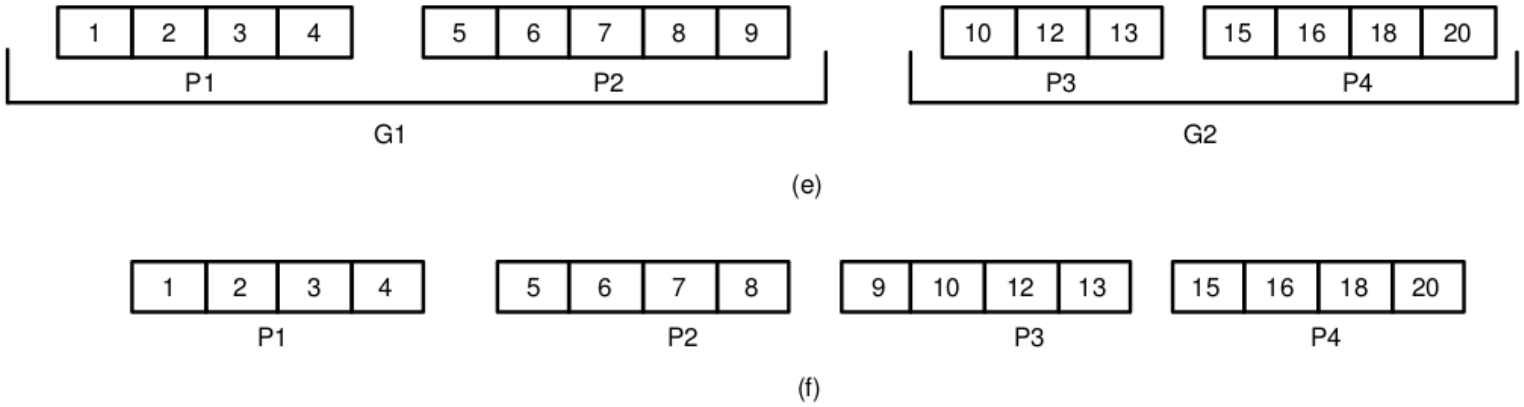

(d) Analogamente, $p_3$ enviou o cesto $B_{3,1}$ para o processador $p_2$, que pertence ao grupo $G_1$.

# Algoritmo CGM de Chan e Dehne (2) - Exemplo




(e) Cada um dos grupos $G_1$ e $G_2$ executa novamente o algoritmo de ordenação_CD.

# Algoritmo CGM de Chan e Dehne (2) - Exemplo




(f) Por fim, é realizado um balanceamento de modo que todos os processadores tenham $\frac{n}{p}$ elementos.

# List Ranking no Modelo BSP/CGM

# List Ranking

Seja $L$ uma lista representada por um vetor $s$ tal que $s[i]$ é o nó sucessor de $i$ na lista $L$, para $u$, o último elemento da lista $L$, $s[u] = u$. Denominamos $i$ e $s[i]$ por vizinhos.

A **distância** entre $i$ e $j$, $d_L(i, j)$, é o número de nós entre $i$ e $j$ mais 1.

# List Ranking

O problema do **list ranking** consiste em computar para cada $i \in L$, a distância entre $i$ e o último elemento $j$, denotado por $rank_L(i) = d_L(i, j)$.

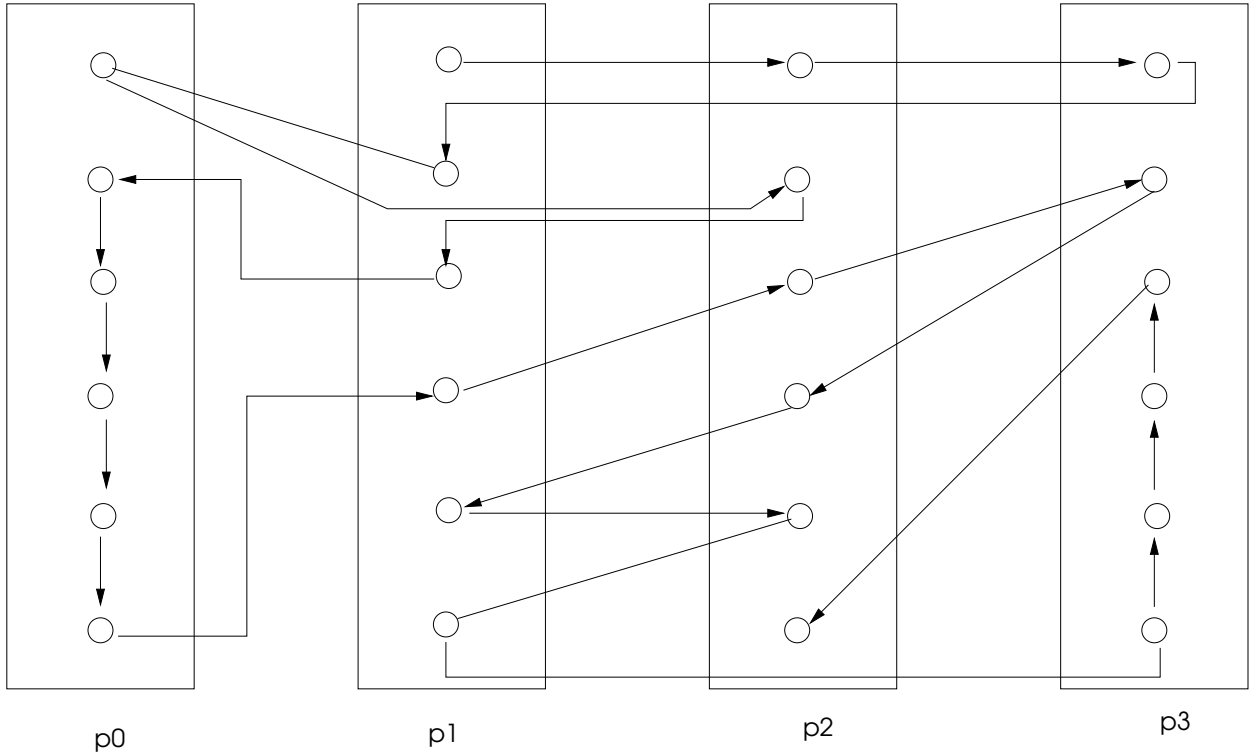
# List Ranking

Não é possível aplicar a idéia do teorema de Brent no problema do list ranking.

- O número de nós da lista cujos sucessores não estão armazenados no mesmo processador pode variar de 0 a $n/p$.

- Mesmo se todos os sucessores estiverem em um dado processador, após a aplicação da duplicação recursiva (*pointer jumping*), não há garantia que isto ocorra nos passos seguintes.

# List Ranking




▷ $n = 24$ elementos armazenados em $p = 4$ processadores. Cada processador armazena $n/p = 6$ elementos.

# List Ranking

O número de rodadas de comunicação pode chegar a $O(\log n)$, uma vez que pode ser necessária a comunicação para obter o sucessor de um dos seus elementos.

A simples aplicação da duplicação recursiva não leva a um algoritmo CGM eficiente.

# List Ranking

Para diminuir o número de rodadas de comunicação, a idéia é a de selecionar um conjunto de elementos $i^* \in L$ bem distribuido em L, de tal forma que a distância de qualquer $i \in L$ a $i^*$ possa ser computada em $O(\log^k p)$ aplicações de pointer jumping.

# r-ruling set

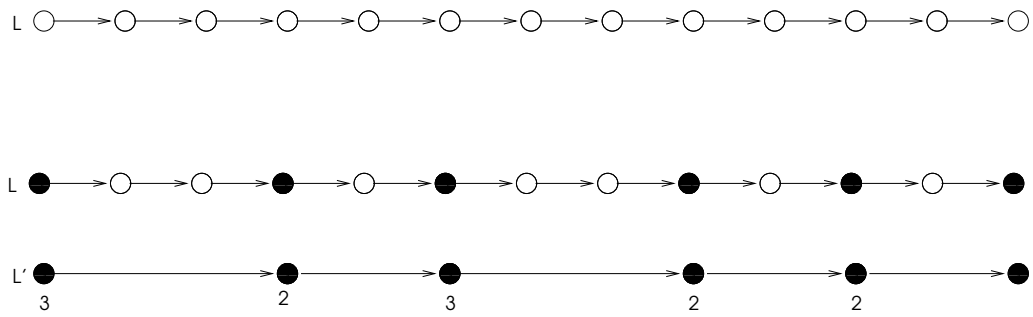
Um **r-ruling set** de L é um subconjunto de elementos selecionados da lista $L$ com as seguintes propriedades:

(1) Dois vizinhos nunca são selecionados.

(2) A distância entre qualquer elemento não selecionado ao próximo elemento selecionado é no máximo $r$.

# r-ruling set

Uma lista L e um 3-ruling set.



# Algoritmo List Ranking

# Algoritmo List Ranking determinístico

Entrada: Uma lista ligada L de comprimento $n$ onde cada processador armazena $n/p$ elemento $i \in L$ e seus respectivos ponteiros $s_L[i]$.

Saída: Para cada elemento $i$ da lista obter seu $rank_L(i)$ em $L$.

# Algoritmo List Ranking determinístico

1. Calcular $O(p^2)$-*ruling set* R com $|R| = O(n/p)$.

2. Fazer um *broadcast* de $R$ para todos os processadores. O subconjunto $R$ é uma lista ligada onde cada elemento $i$ é atribuído um ponteiro para o próximo elemento $j \in R$ com respeito à ordem induzida por $L$.

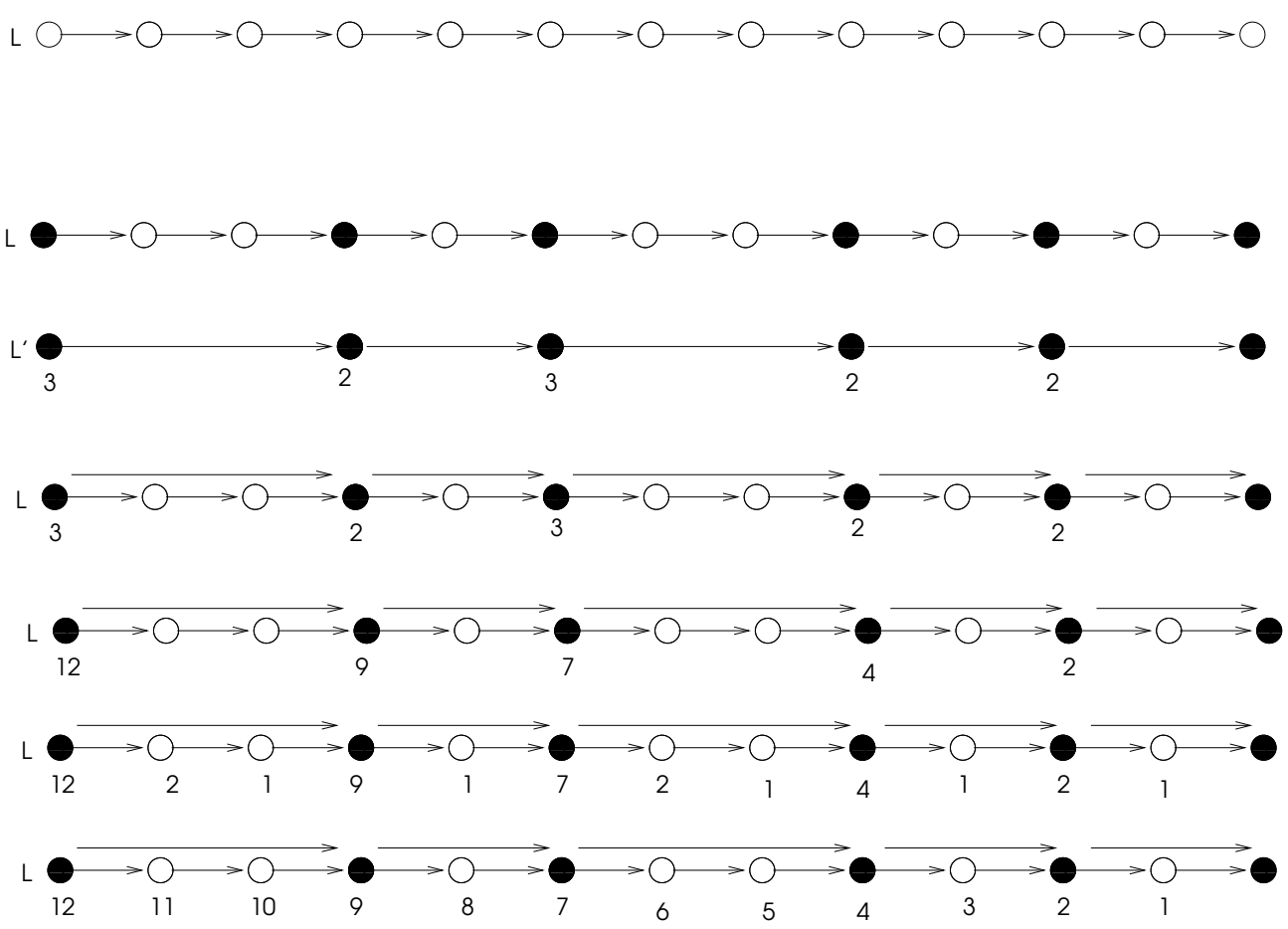
3. Calcular sequencialmente em cada processador o *List Ranking* de $R$, isto é, calcular para cada $j \in R$ seu $rank_L(j)$ em $L$.

4. Obter para cada elemento $i \in L - R$ sua distância $d_L(i, s_R[i])$ ao próximo elemento $s_R[i]$ em $R$ através da duplicação recursiva.

5. Calcular em cada processador os *ranks* dos seus elementos $i \in L - R$ com:

$$rank_L[i] = d_L(i, s_R[i]) + rank_L(s_R[i])$$

# Algoritmo List Ranking determinístico



# Compressão determinística de listas

Para computar um $O(p^2)$-*ruling set* em $O(\log p)$ rodadas de comunicação, usaremos uma técnica chamada **compressão determinística de lista**.

# Compressão determinística de listas

Na compressão determinística da lista aplica-se uma sequência alternada de fases de **compressão** e de **concatenação**.

# Compressão determinística de listas

Na fase de compressão, seleciona-se um subconjunto de elementos da lista L, utilizando um esquema de rotulação (*deterministic coin tossing*).

A fase de concatenação consiste da construção de uma lista ligada, através da duplicação recursiva, com os elementos selecionados na fase de compressão.

# Compressão determinística de listas

O rótulo $l(i)$, $\forall i \in L$, na fase de compressão é o número do processador $p$ que armazena o nó $i$.

Neste esquema, cada elemento de L tem no máximo $p$ rótulos distintos.

Seja $M = \{i, i+1, \cdots, i+k\} \subseteq L$, tal que $l(i) \neq l(s[i])$, $\forall i \in L$, onde o $s[i]$ é o máximo local se $l(i) < l(s[i]) > l(s[s[i]])$.

# Compressão determinística de listas

Selecionando apenas máximos locais não há garantia de distância menor que $O(p)$.

Pode haver $L' = \{j, j+1, \cdots, j+k\} \subseteq L$, onde $l(s[j]) = l(j), \forall j \in L'$ e $k > p$

Para contornar esse problema, sempre que tivermos um subconjunto com esta característica, selecionamos todos os segundos elementos.

# Algoritmo $p$-*ruling set*

**Entrada:** (1) Uma lista ligada $L$ representada pelo vetor $s$ onde $s[i]$ é o sucessor de $i$ na lista $L$. (2) $p$ processadores $p_0, p_1, \cdots, p_{p-1}$.

**Saída:** Um subconjunto $R \subset L$ de nós selecionados.

# Algoritmo $p$-*ruling set*

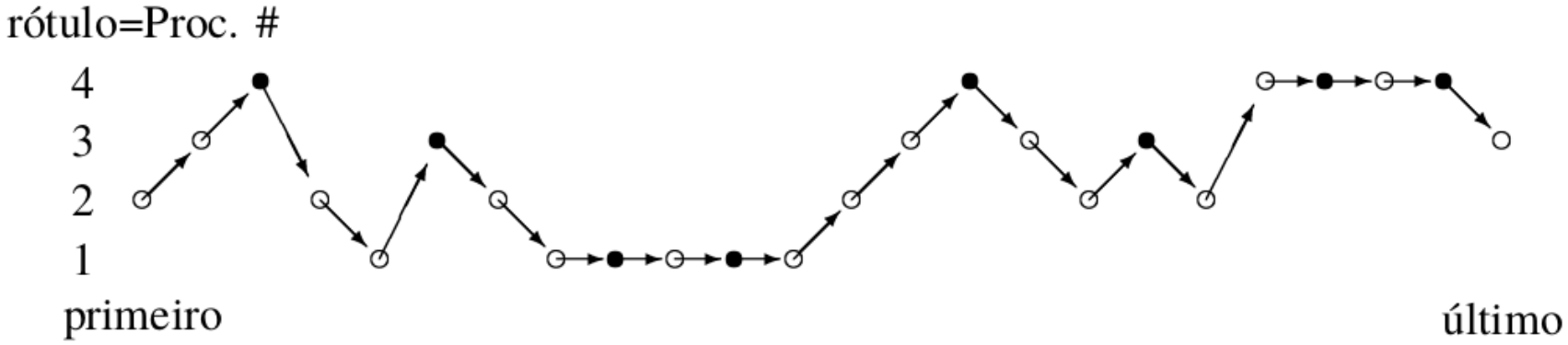
**algoritmo** $p -$ rulling set

(1) **para** $i = p_j * (n/p) + 1$ **até** $(p_j + 1) * (n/p)$ **faça**
    (1.1) $sel[i] \leftarrow$ não selecionado

(2) **para** $i = p_j * (n/p) + 1$ **até** $(p_j + 1) * (n/p)$ **faça**
    (2.1) **se** $l(i) < l(s[i]) > l(s[s[i])$ **então** $sel[s[i]] \leftarrow$ selecionado
    (2.2) **se** $l(i) = l(s[i])$ **então**
        **se** $l(s[i]) = l(s[s[i]])$ **então**
            $sel[s[i]] \leftarrow$ selecionado
        **senão se** $l(s[i]) > l(s[s[i]])$ **então**
            $sel[i] \leftarrow$ selecionado

    **– Fim do algoritmo**

# Algoritmo $p$-*ruling set*



# Algoritmo $p$-*ruling set*

O algoritmo computa $O(p)$-ruling set $R$, mas $|R|$ pode ser igual a $O(n)$, pois se os $n/p$ elementos da lista $L$ em cada processador $p_j$ tiverem rótulo igual a $j$, serão selecionados $n/2$ elementos.

Para selecionar $O(n/p)$ elementos necessitamos executar o algoritmo $\log p$ vezes.

# Algoritmo $p$-*ruling set*

Para obter uma $p^2$-ruling set $R$ de elementos selecionados com $O(n/p)$ elementos, necessitamos desmarcar elementos que foram selecionados.

Para isso, construímos uma nova lista ligada com elementos selecionados e aplicamos o procedimento de marcação (máximos locais e segundos elementos de sublista no mesmo processador)

# Algoritmo $p$-*ruling set*

Se dois elementos selecionados estão a uma distância $\Theta(p)$ a um dado momento, então não é necesário aplicar novamente a compressão para reduzir o número de elementos selecionados.

Abordagem básica: intercalar duplicação recursiva (concatenação) com compressão.

Não aplicar duplicação recursiva aos elementos que estão apontando para elementos selecionados.

# Algoritmo $p^2$-*ruling set*

**Entrada:** $L$ representada pelo vetor $s$ onde $s[i]$ é o sucessor de $i$ na lista $L$. (2) processadores $p_0, p_1, \cdots, p_{p-1}$ e $LC$ uma cópia de $L$.

**Saída:** $R \subset L$ de nós selecionados e $|R| = O(n/p)$.

# Algoritmo $p^2$-*ruling set*

(1) $R \leftarrow p - ruling\_set(LC)$

(2) **para** $k = 1$ **até** $\log p$ **faça**

(2.1) **para todo** $i \in L$ **faça em paralelo**

**se** $s[i] =$ não selecionado então $s[i] = s[s[i]]$

(2.2) **para todo** $i \in L$ **faça em paralelo**

**se** ($i, s[i]$ e $s[s[i]]$ são selecionado) **e não** ($l(i) < l(s[i]) > l(s[s[i]])$) **e** ($l(s[i]) \neq l(s[s[i]])$) **então**

$s[i] \leftarrow$ **não-selecionado**.

(2.3) Sequencialmente, cada processador processa as sublistas de elementos subsequentes que estão armazenadas no mesmo processador. Para cada sublista, marque todo segundo elemento como não selecionado. Se uma das sublistas possui apenas dois elementos, marque ambos como não selecionado.

(3) Selecione o último elemento.

# Algoritmo $p^2$-*ruling set*

O algoritmo computa um $p^2$-*ruling set* $R$ onde $|R| = O(n/p)$ usando $O(\log^2 p)$ rodadas de comunicação e $O(n/p)$ computação local por rodada.

# Algoritmo List Ranking determinístico

O problema do list ranking para uma lista L com $n$ vértices pode ser resolvido no modelo CGM com $p$ processadores e $O(n/p)$ memória local por processador usando $O(\log p)$ rodadas de comunicação e $O(n/p)$ computação local por rodada.

Fim