Algoritmo de Ordenação Bubble Sort e derivados

Computação Paralela
2018

Ordenação em Paralelo

Por quê?

- é uma operação frequente em muitas aplicações

Objetivo

- ordenar uma sequência de números (valores) por ordem crescente usando n processadores

Speedup potencial?

- melhor algoritmo sequencial tem complexidade $\mathcal{O}(n \log n)$
- o melhor a que podemos aspirar com um algoritmo paralelo, usando n processadores é:
  complexidade óptima do algoritmo paralelo: $\mathcal{O}(n \log n)/n = \mathcal{O}(\log n)$

Compare e Troque

Uma operação que forma a base de vários algoritmos clássicos de ordenação é a operação **compare-e-troque** (ou **compare-e-inverta**).

Em uma operação **compare-e-troque**, dois números, por exemplo, A e B são comparados e, se eles não estão ordenados, eles são intercambiados; caso contrário, permanecem não modificados.

```
if (A > B) { // sorting in increasing order
  temp = A;
  A = B;
  B = temp;
}
```

Questão: como se pode fazer o **compare-e-troque** em paralelo?

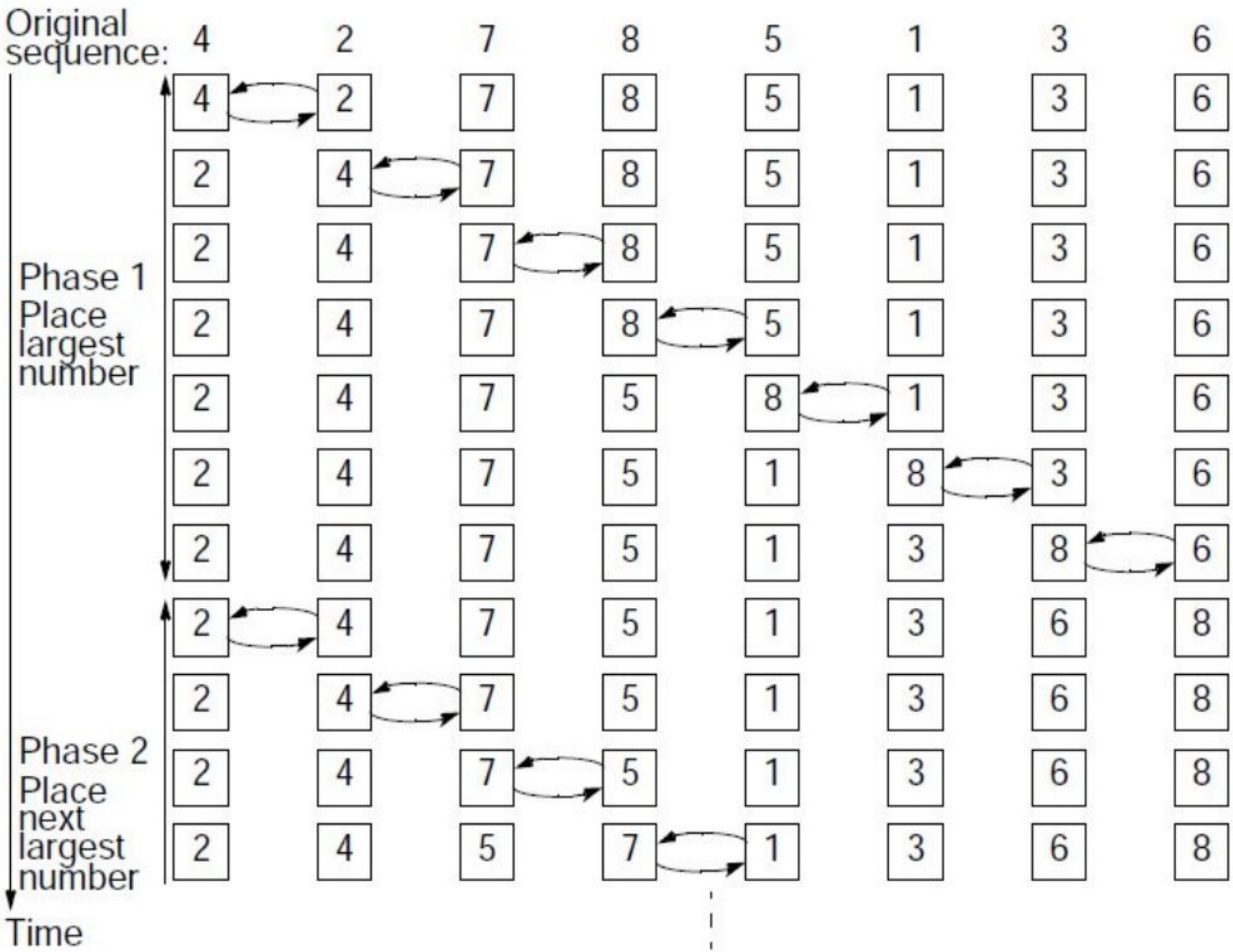
Bubble Sort (método da bolha)

- compara dois a dois e troca se estiverem fora de ordem.
- maiores valores vão sendo deslocados para o final da lista.
- número de comparações e trocas: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

o que corresponde a uma complexidade $\mathcal{O}(n^2)$.

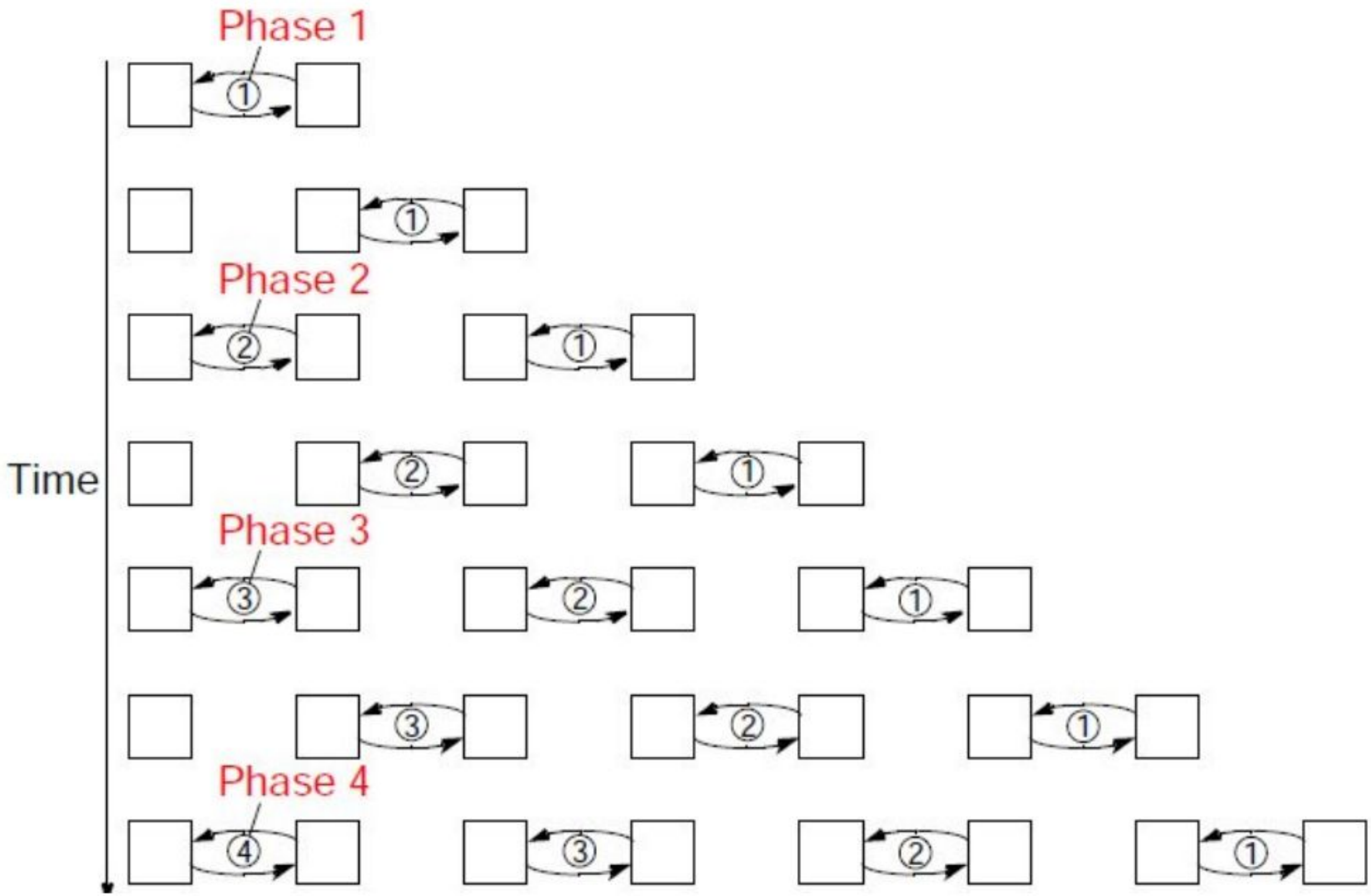
```
for (i = N - 1; i > 0; i--)
  for (j = 0; j < i; j++) {
    k = j + 1;
    if (a[j] > a[k]) {
      temp = a[j];
      a[j] = a[k];
      a[k] = temp;
    }
  }
```

Exemplo bubble-sort



Bubble Sort Paralelo

Ideia é ter várias iterações a correr em paralelo.



Par-Ímpar com transposição (1/2)

- é uma variante do bubble-sort
- opera em duas fases alternadas:

Fase-par:

- os processos par trocam números com os seus vizinhos direitos.

Fase-ímpar:

- os processos ímpar trocam números com os seus vizinhos direitos.

Par-Ímpar com transposição (2/2)

Step	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	4	2	7	8	5	1	3	6
1	2	4	7	8	1	5	3	6
2	2	4	7	1	8	3	5	6
3	2	4	1	7	3	8	5	6
4	2	1	4	3	7	5	8	6
5	1	2	3	4	5	7	6	8
6	1	2	3	4	5	6	7	8
7	1	2	3	4	5	6	7	8

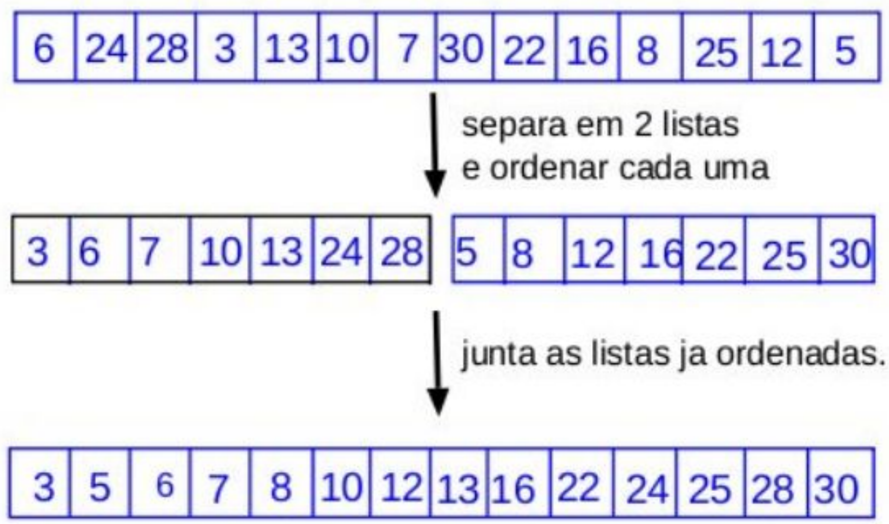
Time

Algoritmo paralelo - Par-Ímpar com transposição

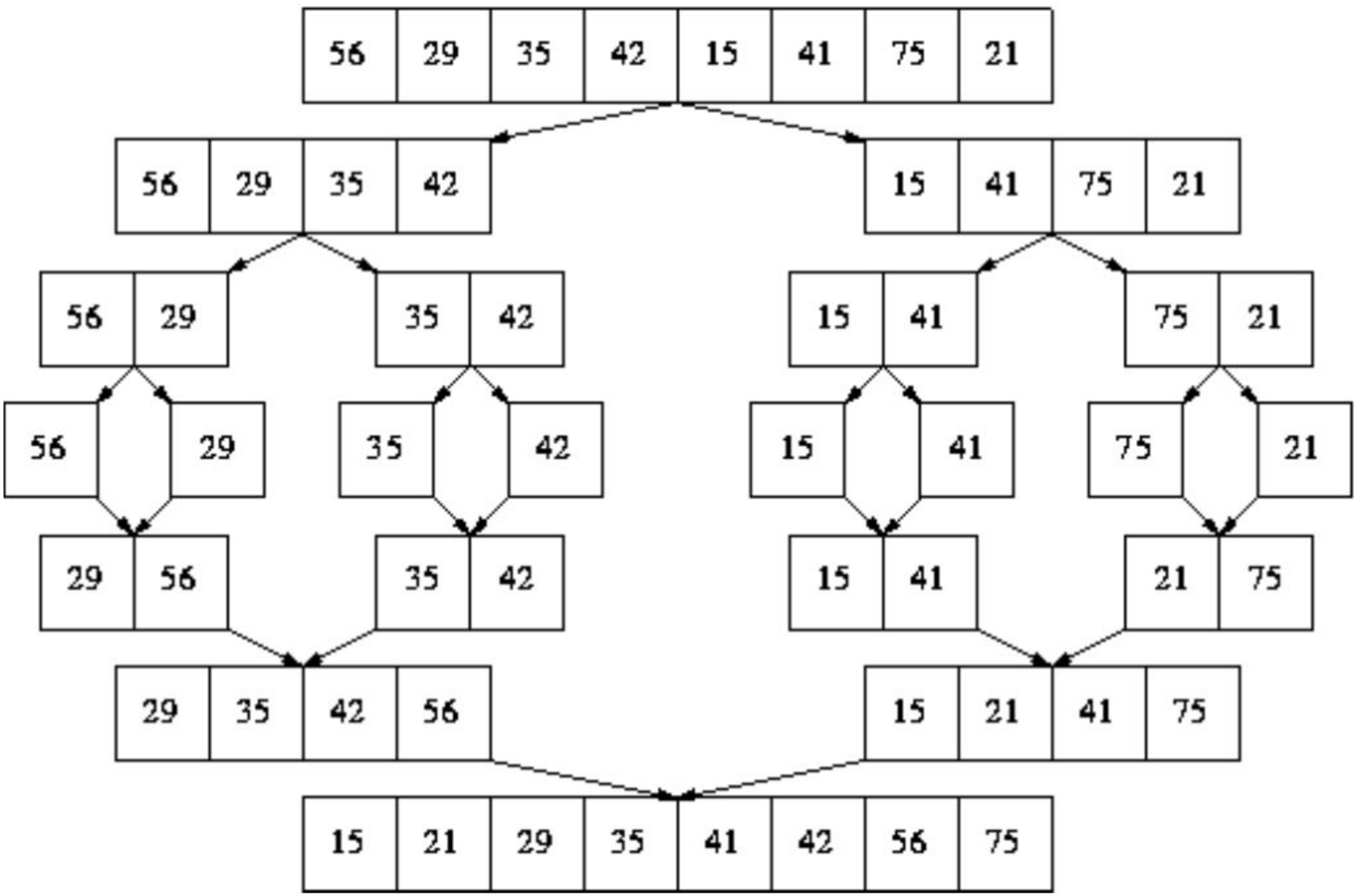
```
void ODD-EVEN-PAR(n)
{
  id = process label
  for (i= 1; i<= n; i++) {
    if (i é ímpar)
      if (id for ímpar)
        compara-e-troca-min(id+1);
      else
        compara-e-troca-max(id-1);
    if (i é par)
      if (id for par)
        compara-e-troca-min(id+1);
      else
        compara-e-troca-max(id-1);
  }
```

Mergesort (1/2)

- Exemplo de um algoritmo *divide-and-conquer*
- Método de ordenação em que para ordenar um vector, subdivide-o em duas partes, aplica novamente o método a cada uma das partes e quando estas estiverem ordenadas (2 vectores ordenados) com m e n elementos, faz-se a junção para produzir um vector ordenado que contém os $m + n$ elementos do vector inicial.
- A complexidade é, em média, $\mathcal{O}(n \log n)$.



Mergesort



56 29 35 42 15 41 75 21
56 29 35 42
15 41 75 21
56 29
35 42
15 41
75 21
56
29
35
42
15
41
75
21
29 56
35 42
15 41
21 75
29 35 42 56
15 21 41 75
15 21 29 35 41 42 56 75

Mergesort

Computações (cálculos) somente ocorrem quando se está mesclando as sublistas. No pior caso, leva 2s - 1 passos para mesclar duas sublistas ordenadas de tamanho **s**. Se nós temos m = n/s sublistas ordenadas em uma etapa de mesclagem, levará

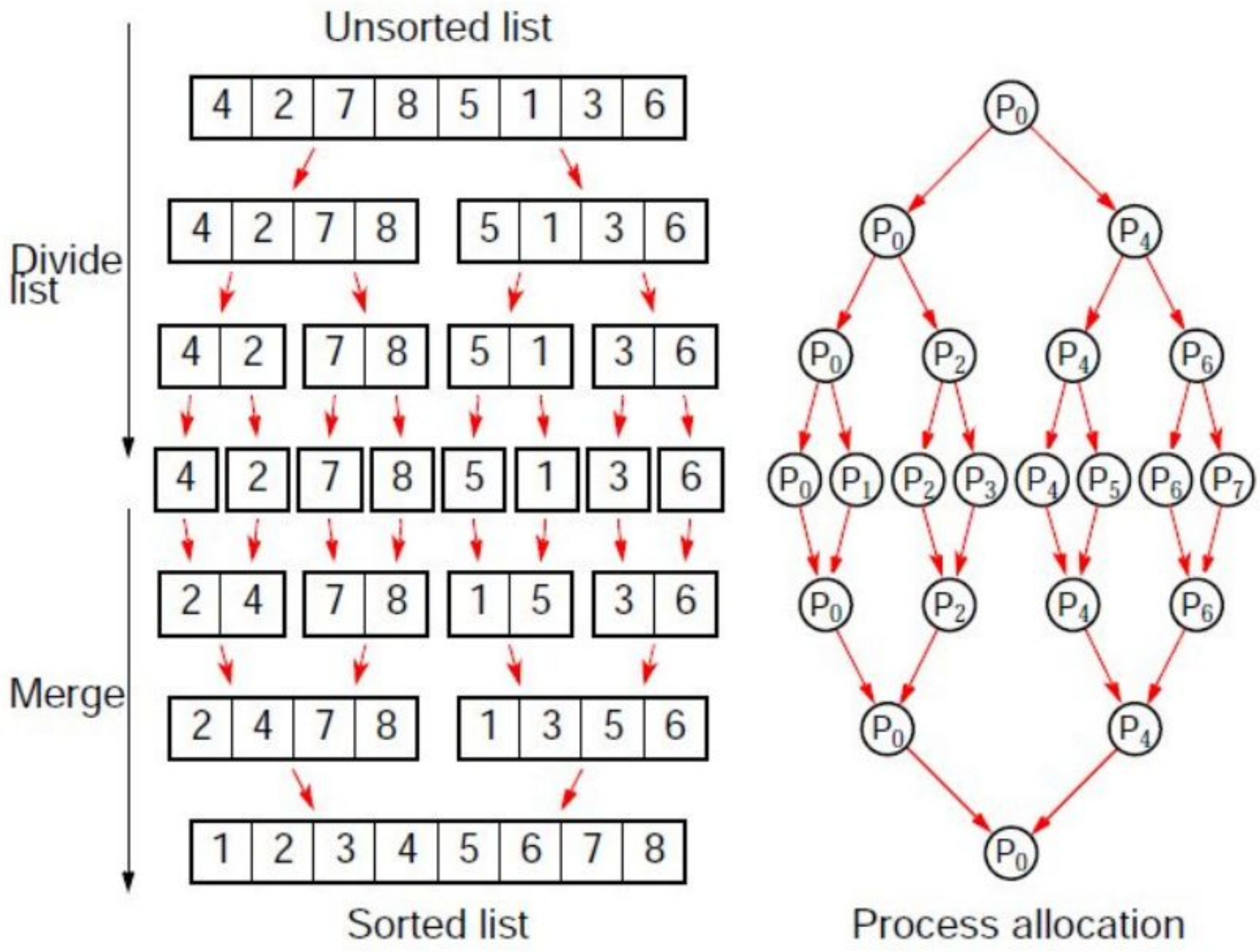
$$\frac{m}{2}(2s-1) = ms - \frac{m}{2} = n - \frac{m}{2}$$

passos para mesclar todas as sublistas (duas a duas).

Como ao todo temos log(n) etapas de mesclagem, isto corresponde a uma complexidade de tempo de *O(n log(n))*

Mergesort em paralelo (2/2)

Usando uma atribuição de trabalho a processos em árvore.



Mergesort Paralelo

Se nós ignoramos o tempo de comunicação, computações ocorrem somente quando estamos mesclando as sublistas. Mas agora, no pior caso, levamos 2**s** - 1 passos para mesclar todas as sublistas (duas a duas) de tamanho **s** em um passo da mesclagem.

Como no total temos log(n) passos de mesclagem, levará:

$$\sum_{i=1}^{\log(n)} (2^i - 1)$$

passos para se obter a lista final ordenada em uma implementação paralela, que corresponde a uma complexidade de tempo de *O(n)*.

Quicksort

Quicksort é um algoritmo de ordenação popular que também utiliza a abordagem **dividir-e-conquistar**.

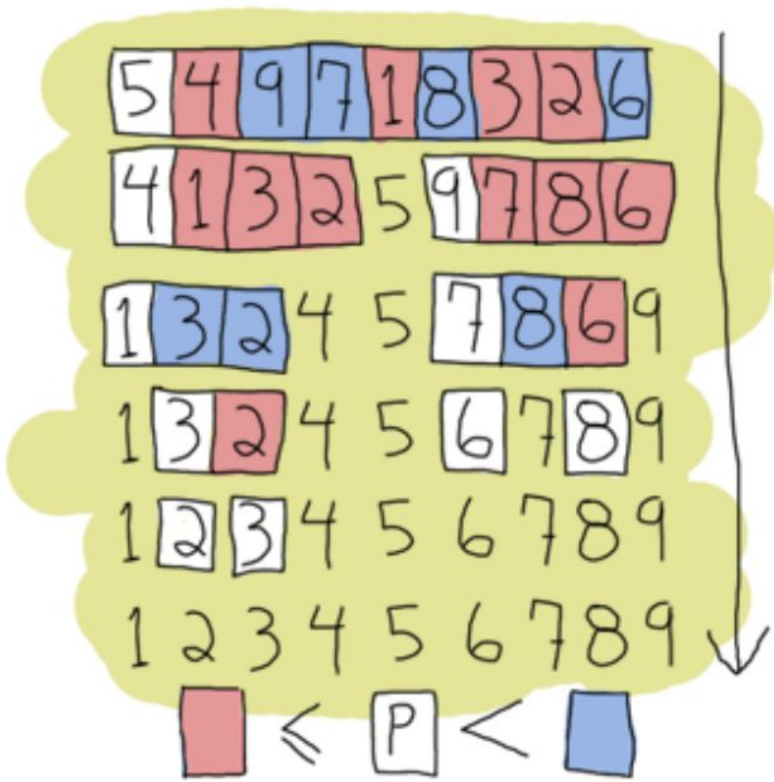
A lista não ordenada inicial é primeiramente dividida em duas sublistas de modo que **todos os elementos na primeira sublista são menores que todos os elementos na segunda sublista**.

Isto é conseguido através da seleção de um elemento chamado **pivô**, contra o qual cada um dos outros elementos é comparado (o pivô pode ser qualquer elemento da lista, mas frequentemente o primeiro ou o último são escolhidos).

Cada sublista é então aplicado o mesmo método de divisão até que elementos individuais são obtidos. Com ordenação apropriada das sublistas, a lista final ordenada é então obtida.

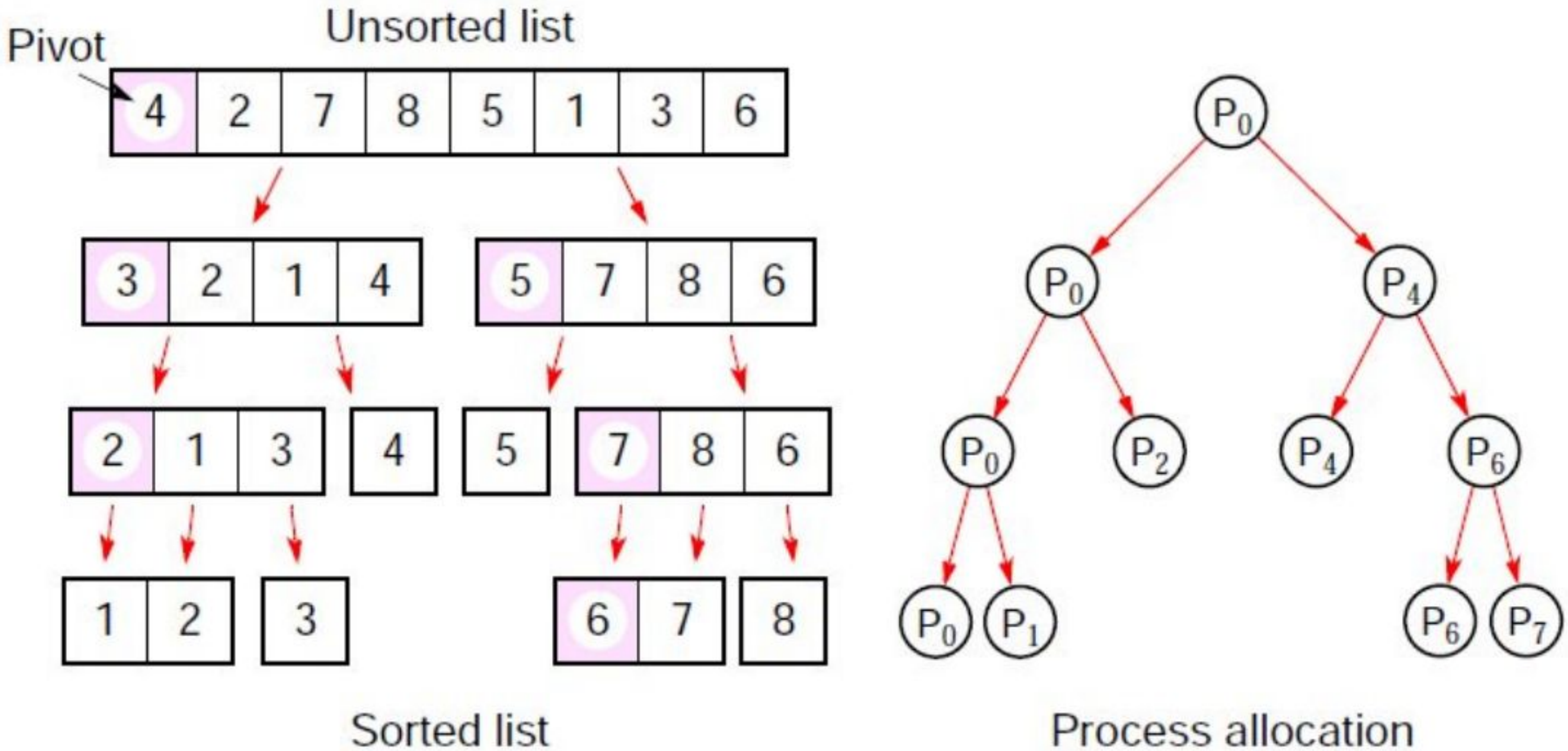
Quicksort

Na média, o algoritmo quicksort apresenta uma complexidade de tempo de $O(n \log(n))$, e, no pior caso, apresenta uma complexidade de $O(n^2)$, apesar de este comportamento ser raro.



Quicksort em paralelo

Usando uma atribuição de trabalho a processos em árvore.



Quicksort paralelo

Alocar processos em uma estrutura de árvore leva a dois problemas fundamentais:

- Em geral, a **árvore de partição não é perfeitamente balanceada** (selecionar bons candidatos a pivô é crucial para a eficiência)
- O processo de atribuir o trabalho a processos **limita seriamente o uso eficiente dos processos disponíveis** (a partição inicial somente envolve um processo, então a segunda partição envolve dois processos, então quatro processos, e assim por diante…)

Quicksort Paralelo

Novamente, se nós ignoramos o tempo de comunicação e consideramos que a seleção dos pivôs é ideal, i.e., criamos sublistas de tamanho igual, então levaremos:

$$\sum_{i=0}^{\log(n)} \frac{n}{2^i} \approx 2n$$

passos para obter a lista ordenada final em uma implementação paralela, que corresponde a complexidade de tempo *O(n)*. O pior caso da seleção de pivô degenera para uma complexidade de tempo de $O(n^2)$.