Algorytmy sortujące

Sortowanie

- Jeden z najczęściej występujących, rozwiązywanych i stosowanych problemów.
  - Ułożyć elementy listy (przyjmujemy: tablicy) w rosnącym porządku
- Sortowanie może być oparte na porównaniach lub też wykorzystujące specjalne właściwości porównywanych liczb.
- Dolne ograniczenie na złożoność sortowania opartego na porównaniach wynosi O(nlogn).
- Święty Graal teoretyków obliczeń równoległych: osiągnięcie złożoności O(log n) na n procesorach.
  - Udało się osiągnąć, ale stałe ukryte w notacji O() są tak duże, że niezwykle utrudniają praktyczne wykorzystanie.
- W praktyce interesuje nas p-krotne przyspieszenie sortowania n-elementowej sekwencji na p procesorach.

Operacja porównania - wymiany (ang. compare and exchange)

- Stanowi podstawę większości algorytmów opartych na porównaniach.
- Dwie liczby, A i B, są porównywane i jeżeli A>B zamieniane:

```
if (A>B) {
    temp=A
    A=B
    B=temp;
}
```

- Jak ją zaimplementować, jeżeli A i B przechowywane są na różnych procesorach w systemie z przekazywaniem komunikatów ?

Porównanie i wymiana na systemie z przesyłaniem komunikatów - wersja 1

- Niech proces P_1 przechowuje A, a proces P_2 przechowuje B.
- P_1 wysyła A do P_2. który porównuje A z B i odsyła mniejszą z dwóch liczb z powrotem do P_1. Pseudokod:

```
// Pseudokod dla procesu P1
send(&A,P2);
recv(&A,P2);


// Pseudokod dla procesu P2
recv(&A,P1);
if (A>B) {
   send(&B,P1);
   B=A;
} else
  send(&A,P1);
```

Porównanie i wymiana na systemie z przesyłaniem komunikatów - wersja 2

- P_1 wysyła A do P_2, a P_2 wysyła B do P_1. Obydwa procesy dokonują porównania zostawiając sobie odpowiednio mniejszą i większą liczbę . Pseudokod:

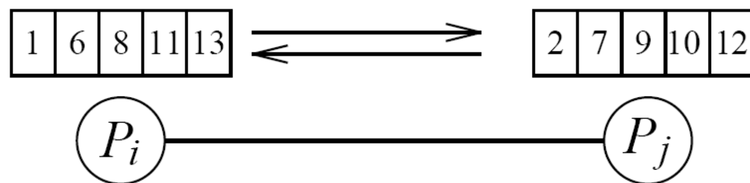
```
// Pseudokod dla procesu P1
send(&A,P2);
recv(&B,P2);
if (A>B) A=B;
```

```
// Pseudokod dla procesu P2
recv(&A,P1);
send(&B,P1);
if (A>B) B=A;
```

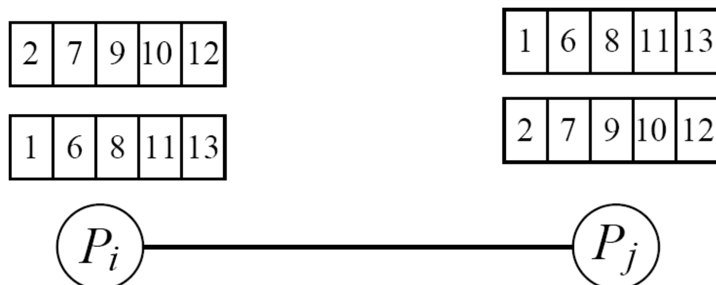
- Aby uniknąć ryzyka blokady P2 ma zamienioną kolejność wysyłania i odbioru.

Operacja compare-and-split

- W praktycznych zastosowaniach mamy p procesów do posortowania n liczb, gdzie p‹‹n. Każdy proces otrzymuje n/p liczb.
- Zakładamy że lista lokalna każdego procesu jest uporządkowana (koszt O(n/plog(n/p)) jednorazowo na starcie algorytmu. Operacja compare-and-exchange przekształca się wtedy w operację compare-and-split. Może być ona zaimplementowana w czterech krokach:
- Krok 1. Procesy wysyłają nawzajem posortowane listy

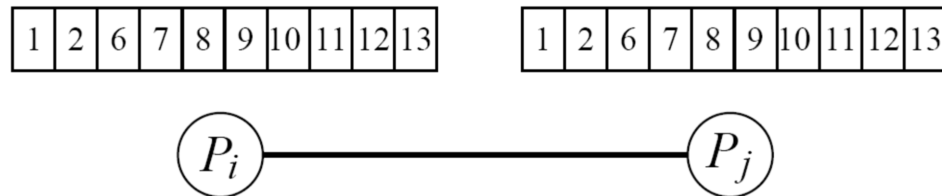


- Krok 2. Każdy z procesów posiada dwie listy.



Operacja compare-and-exchange

- Krok 3. Obydwa procesy scalają dwie listy w jedną posortowaną.



- Krok 4. Proces P_i zatrzymuje połowę najmniejszych elementów, a proces P_j połowę największych.

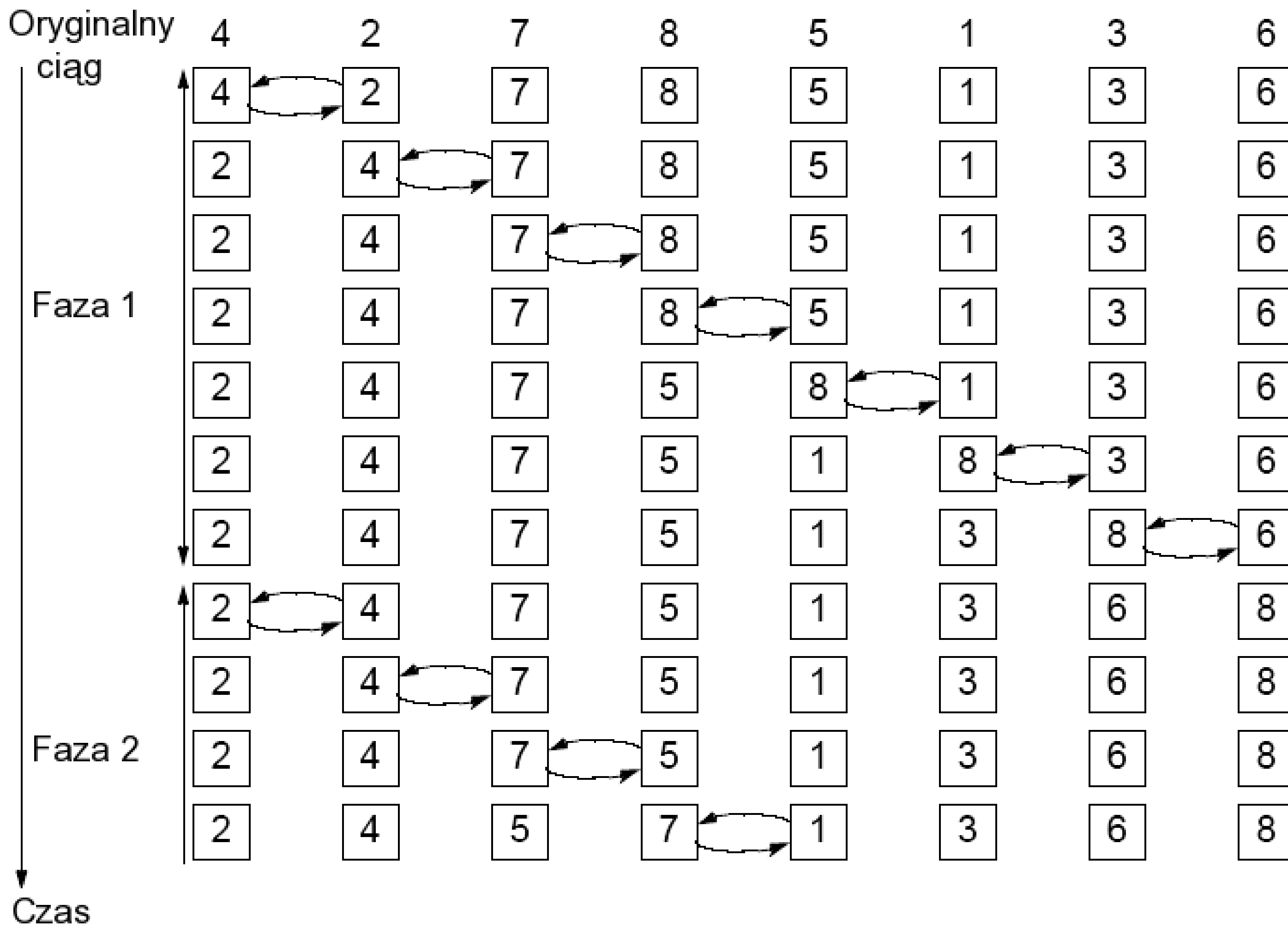
- W ten sposób, poprzez podmianę operacji compare-and-exchange na compare-and-split, możemy przerobić algorytm wyspecyfikowany dla n procesów i n liczb na algorytm wyspecyfikowany dla p procesów i operujący na n liczbach.

Sortowanie bąbelkowe - kod

```
1.  procedure BUBBLE_SORT(n)
2.  begin
3.      for i := n − 1 downto 1 do
4.          for j := 1 to i do
5.              compare-exchange(a_j, a_{j+1});
6.  end BUBBLE_SORT
```

- Dwie zagnieżdzone pętle - złożoność $O(n^2)$.
- W pierwszej iteracji wewnętrznej pętli for największy element przemieszczany na koniec tablicy, w drugiej drugi co do wielkości itd.

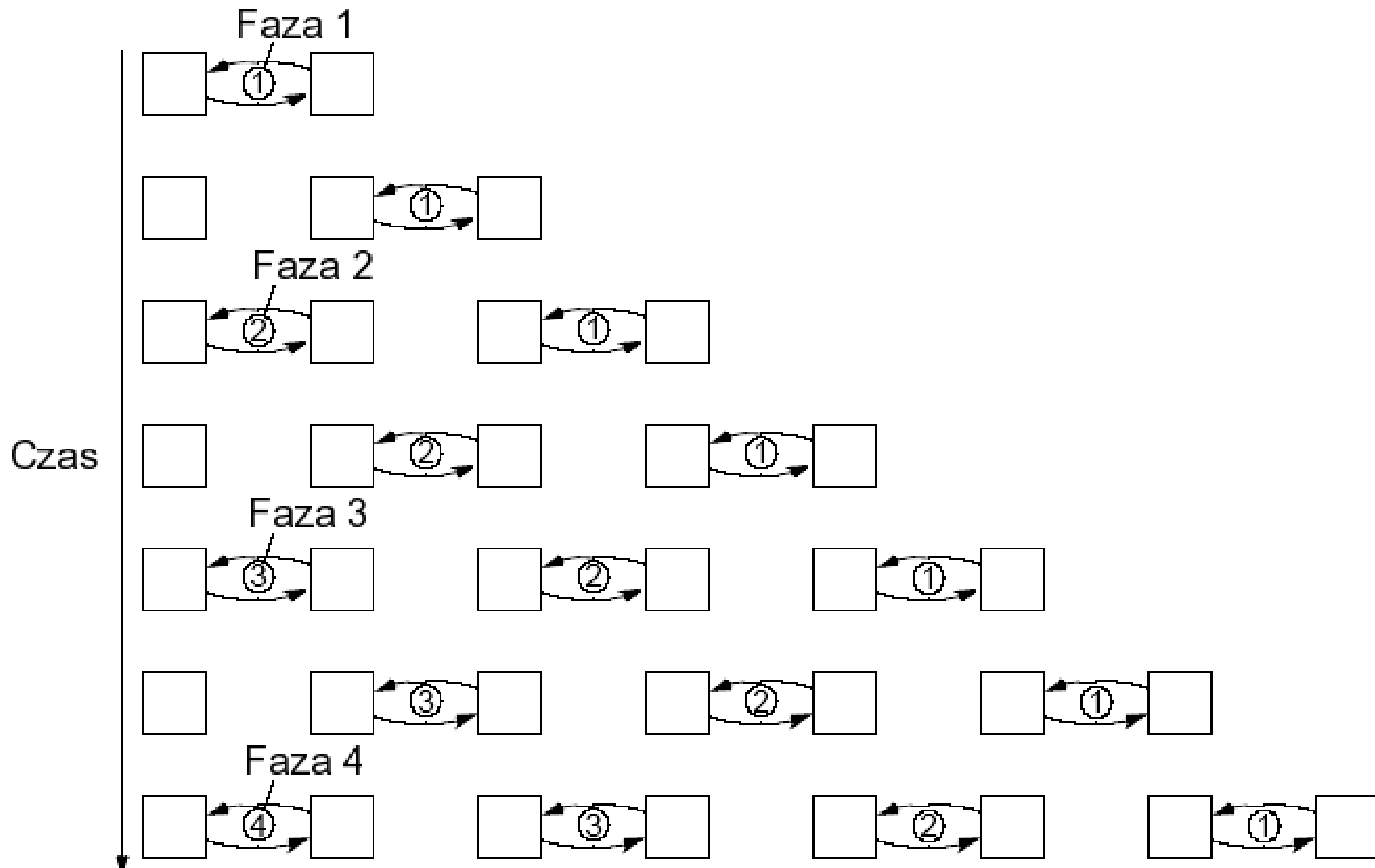
Sortowanie bąbelkowe



- Faza 1 - największy element.
- Faza 2 - drugi co do wielkości.

- Porównywane są sąsiednie elementy; zamiana następuje jeżeli lewy jest większy.

Sortowanie bąbelkowe - zrównoleglenie



- Kolejny procesor rozpoczyna iterację, natychmiast jak może, zanim poprzednia iteracja zakończy się.
- Zakładamy maszynę ze wspólną pamięcią.
- Złożoność O(n) dla n procesorów.

Sortowanie odd-even transposition

- Jest wariantem sortowania bąbelkowego. N procesów; Na razie zakładamy, że każdy przechowuje jedną liczbę.
- Algorytm operuje na przemian w dwóch fazach:
  - Faza parzysta: procesy o numerze parzystym wykonują operację compare -and-exchange ze swoim sąsiadem po prawej stronie.
  - Faza nieparzysta: procesy o numerze nieparzystym wykonują operację compare -and-exchange ze swoim sąsiadem po prawej stronie.
- Po n/2 fazach parzystych i n/2 fazach nieparzystych sekwencja jest posortowana.
- Łącznie mamy n faz.
  - W implementacji sekwencyjnej faza zajmuje O(n) co prowadzi do złożoności $O(n^2)$.
  - W implementacji równoległej z procesem P_i przechowującym jedną pozycję tablicy a[i] faza zajmuje O(1), co prowadzi do złożoności O(n).

Sortowanie odd-even transposition: przykład dla ośmiu liczb

Krok	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Faza parzysta 0	4	2	7	8	5	1	3	6
Faza nieparzysta 1	2	4	7	8	1	5	3	6
Faza parzysta 2	2	4	7	1	8	3	5	6
Faza nieparzysta 3	2	4	1	7	3	8	5	6
Faza parzysta 4	2	1	4	3	7	5	8	6
Faza nieparzysta 5	1	2	3	4	5	7	6	8
Faza parzysta 6	1	2	3	4	5	6	7	8
Faza nieparzysta 7	1	2	3	4	5	6	7	8

- Wersja z jednym procesem przechowującym jedną liczbę.

Sortowanie odd-even transposition: wersja dla n liczb i p procesów

- Każdy proces przechowuje n/p liczb.
- Na początku, z kosztem O(n/p*log(n/p)) każdy proces sortuje (quicksort albo mergesort swoją sekwencję).
- Następnie przeprowadzane jest (naprzemian) p/2 faz parzystych oraz p/2 faz nie parzystych.
  - W każdej fazie operacja compare-and-exchange zastąpiona zostaje operacją compare-and-split o koszcie O(n/p)
  - Zatem koszt łączny koszt O(n)
- Złożoność obliczeniowa:
  - O(n/p*log(n/p)) + O(n) + O(n).

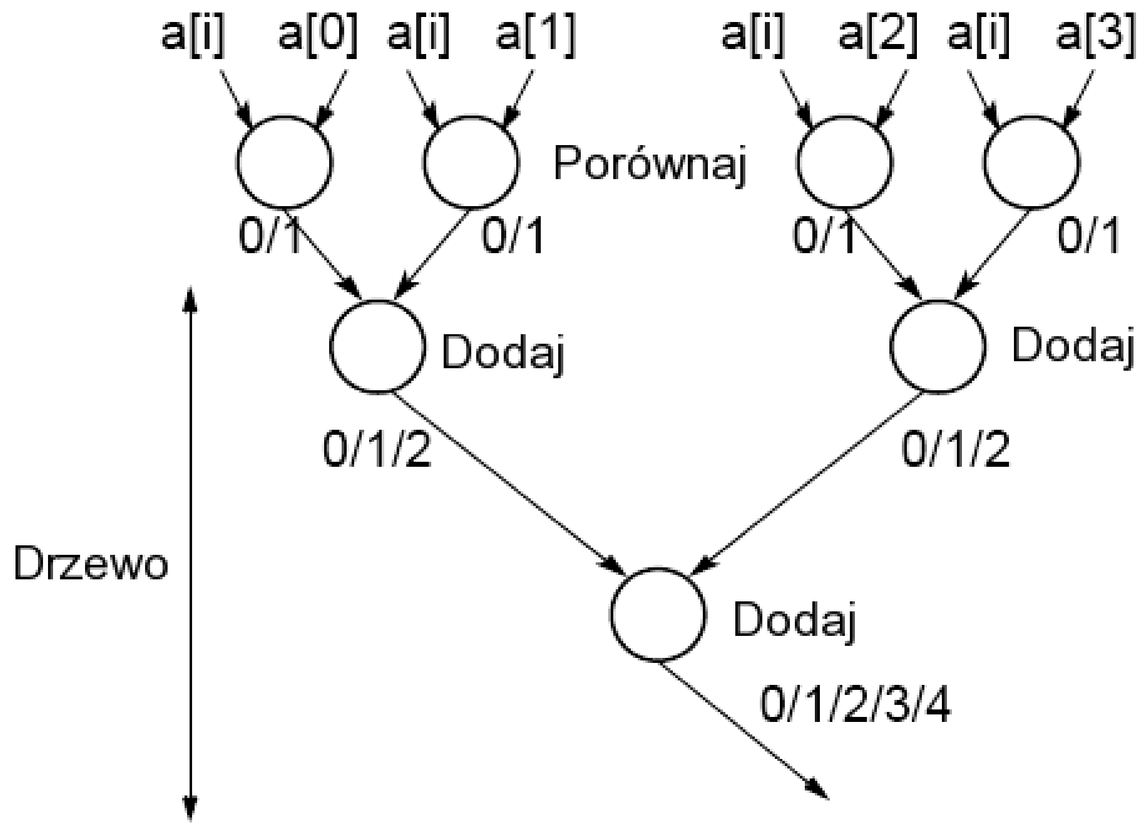
    obliczenia komunikacja
- Algorytm słabo wydajny ma sens jedynie w sytuacji, w której mamy do czynienia z niewielką liczbą procesów. Koszt wstępnego lokalnego sortowania jest wtedy dominujący.

Rank sort

```
for(int i=0;i<n;i++) {
   x=0;
   for(j=0;j<n;j++)
      if (a[i]>a[j])
          x++;
   b[x]=a[i];
}
```

- Bardzo prosty algorytm. Zliczamy w zmiennej x liczbę elementów tablicy mniejszych od a[i]. Następnie a[i] umieszczamy w tablicy wynikowej b na pozycji x.
- Założenie: elementy ciągu a nie mają powtórzeń.
- Zlożoność sekwencyjna $O(n^2)$.
- Algorytm nie wydaje się być wart głębszej uwagi do momentu gdy nie zauważymy, że możemy przydzielić jeden proces do każdej liczby, i obliczać wartości x równolegle.
- Prowadzi do kosztu O(n) dla n procesorów.

Rank sort o złożoności log(n) z n^2 procesorami



- Do obliczenia wartości x używamy n procesorów.
- W praktyce nie do zastosowania (n^2 procesorów !!!).

Counting sort

- Zakłada że liczby w tablicy są liczbami całkowitymi z zakresu 1...m. Wykorzystuje dodatkową tablicę c o rozmiarze m (m+1 w języku C), w której konstruowany jest histogram sekwencji.
- Histogram, to znaczy c[i] jest liczbą wystąpień elementu i w sekwencji.
- Konstrukcja histogramu:

```
for(int i=1;i<=m;i++)
   c[i]=0;
for(int i=0;i<n;i++)
   c[a[i]]++;
```

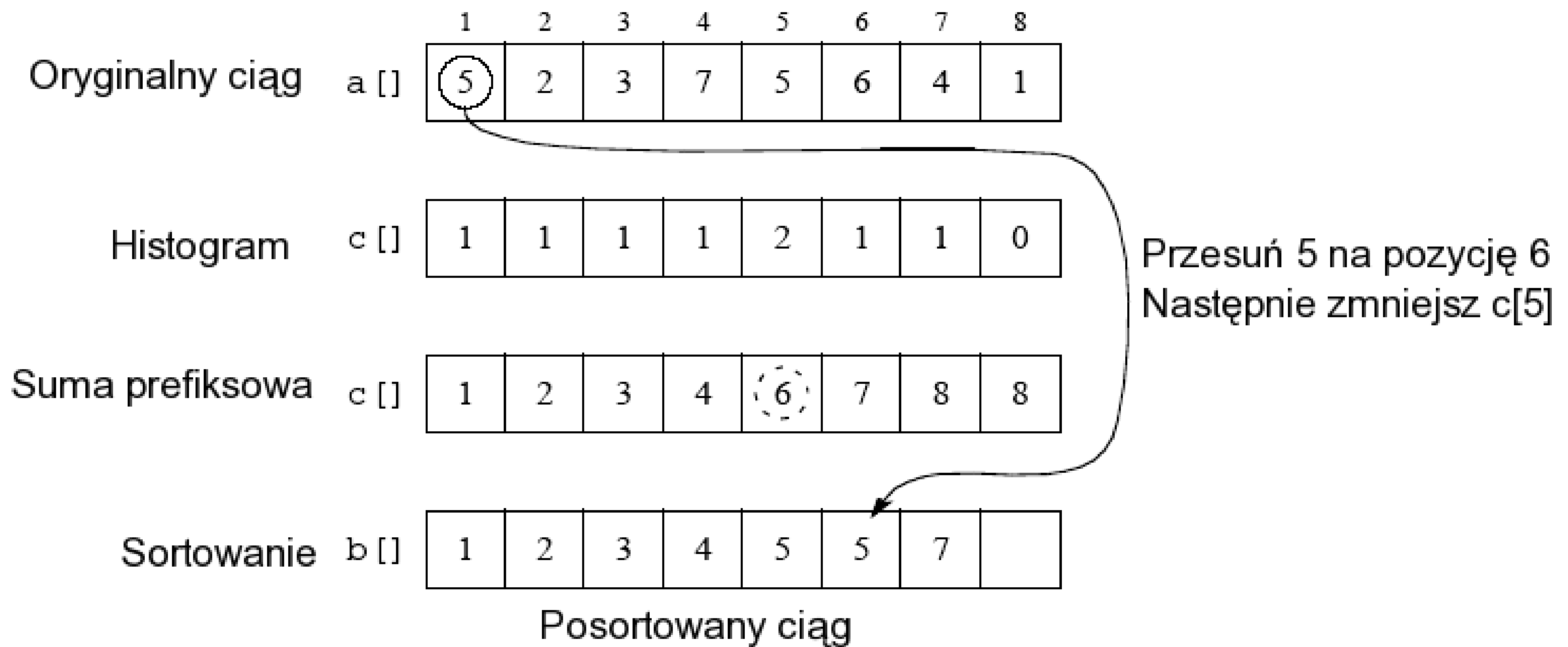
- Następnie liczba elementów mniejszych od c[i] jest obliczana poprzez operację sumy prefiksowej

```
for(int i=2;i<=,;i++)
   c[i]=c[i]+c[i-1];
```

- Następnie elementy są umieszczane w posortowanej tablicy b.

```
for(i=n-1;i>=0;i--) {
   b[c[a[i]]=a[i];
   c[a[i]]--;
}
```

Counting sort - przykład



- Algorytm ma złożoność O(m+n) Jeżeli m jest O(n) to złożoność upraszcza się do O(n).

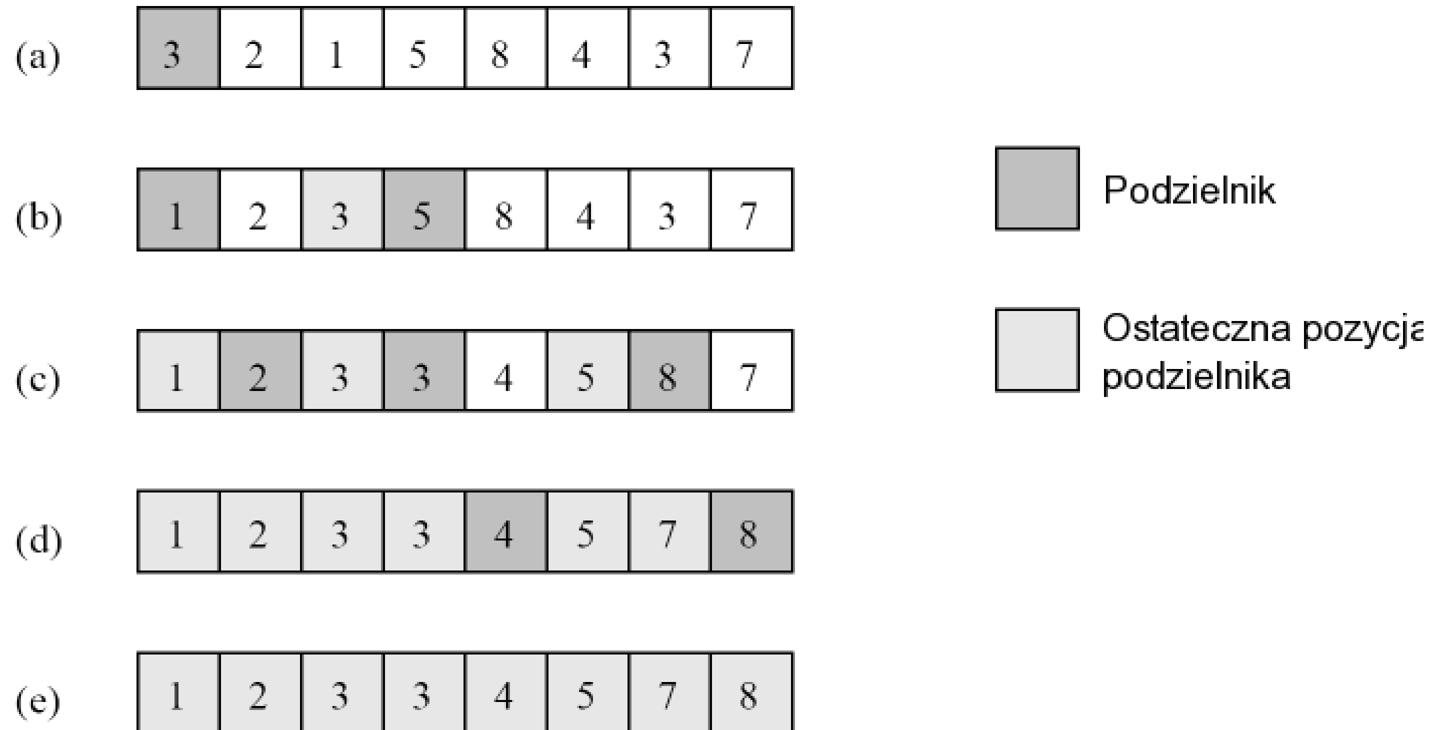
Counting sort - zrównoleglenie

- Zakładamy dalej że m jest O(n).
- Koszt poszczególnych faz algorytmu:
  - Wyzerowanie tablicy c: O(1) na na procesorach. Każdy procesor zeruje jeden element.
  - Wyliczenie tablicy c: O(1) na n procesorach. Każdy procesor dokonuje inkrementacji elementu tablicy c odpowiadającemu wartości a[i].
  - Operacja sumy prefiksowej O(logn) na n procesorach (przy wykorzystaniu struktury drzewiastej podobnej do tej w rank sort).
  - Finalne posortowanie O(1) na n procesorach. Każdy procesor wykonuje jedną instrukcję finalnej pętli.
- Łączny koszt O(log(n)) na n procesorach.

Algorytm quicksort

- Szybki algorytm o oparty na regule „dziel i zwyciężaj”
- Lista elementów dzielona jest na dwie podlisty, tak że wszystkie elementy na jednej podliście są mniejsze od wszystkich elementów na drugiej.
- Jest to osiągane przez wybór jednego elementu tzw. podzielnika. Następnie przeglądamy całą listę i elementy mniejsze od podzielnika umieszczamy na jednej podliście a elementy większe na drugiej.
- Podział na dwie listy jest wykonywany w miejscu. Nie wykorzystujemy dodatkowych struktur danych, a przesuwamy elementy większe od podzielnika na prawo a mniejsze na lewo.
- Następnie procedura jest rekurencyjnie wywoływana dla obydwu podlist.
- Rekurencja się kończy gdy długość podlisty jest równa 1.
- Wybór podzielnika ma krytyczny wpływ na wydajność algorytmu. W idealnym przypadku gdy podzielnik jest medianą ciągu lista dzielona jest na dwie podlisty. Przy kiepskim (pechowym) wyborze podzielnika (np. zawsze jest to element najmniejszy na liście) dostajemy pesymistyczną złożoność $O(n^2)$. Jednakże złożoność średnia jest $O(n\log(n))$

Quicksort - demonstracja



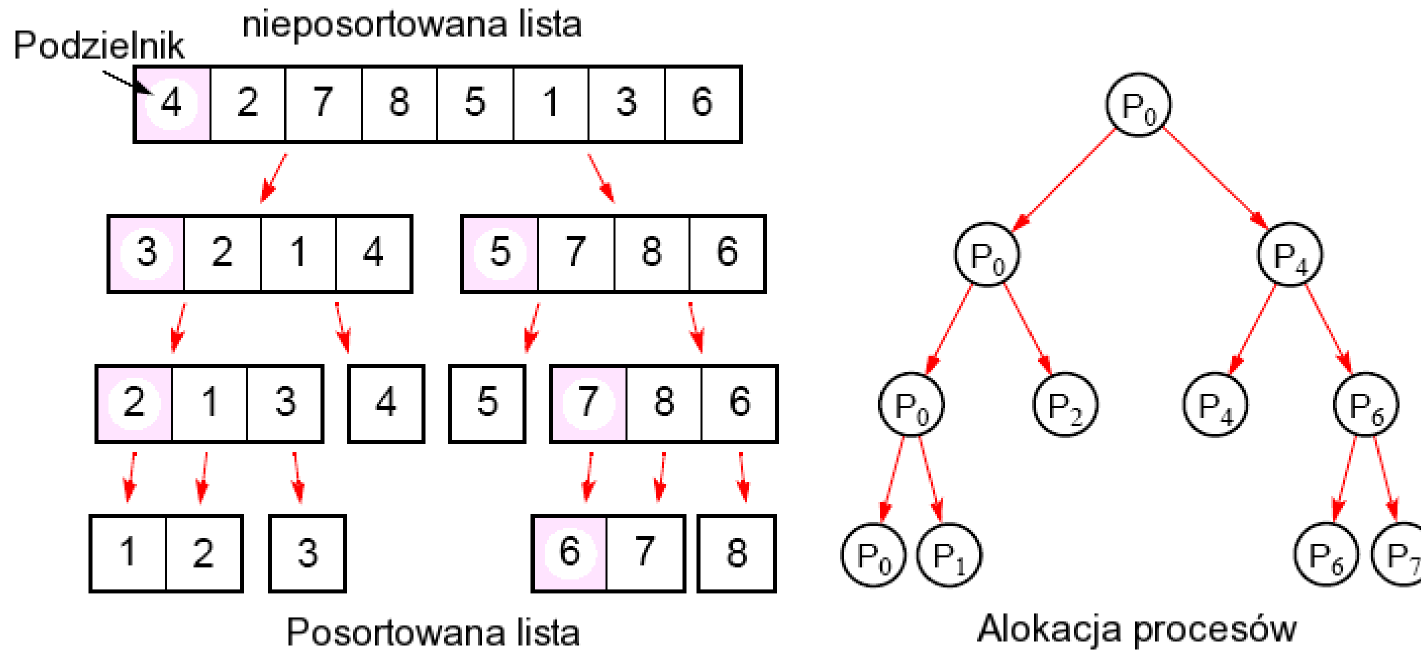
- Przykład algorytmu quicksort sortującego sekwencję ośmiu znaków.

Quicksort - pseudokod wersji szeregowej

```
1.   procedure QUICKSORT (A, q, r)
2.   begin
3.       if q < r then
4.       begin
5.           x := A[q];
6.           s := q;
7.           for i := q + 1 to r do
8.               if A[i] ≤ x then
9.               begin
10.                  s := s + 1;
11.                  swap(A[s], A[i]);
12.              end if
13.          swap(A[q], A[s]);
14.          QUICKSORT (A, q, s);
15.          QUICKSORT (A, s + 1, r);
16.      end if
17.  end QUICKSORT
```

- x - wartość podzielnika. q,r - lewy i prawy kraniec podciągu w tablicy.

Quicksort - naiwne zrównoleglenie



- Idea: Zacznij od jednego procesu. Do każdego wywołania rekurencyjnego zatrudniaj nowy proces. (Dekompozycja rekurencyjna).
- Słabość. W pierwszej fazie (koszt O(n)) aktywny jest jeden proces, pozostałe są bezczynne.
- Efektywna metoda musi zrównoleglać sam podział na dwie podlisty.

Wydajna wersja dla maszyny z przesyłaniem komunikatów (A. Grama i wsp.)

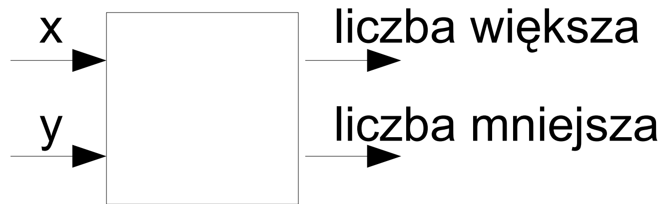
- Zakladamy n liczb i p procesorów. Każdy procesor przechowuje n/p elementów.
- Dzielimy maszynę równoległą na dwie części: dolną i górną połowę. Każdy procesor w dolnej połowie ma jednego partnera w górnej.
- Jeden procesor wybiera podzielnik i wysyła go pozostałym procesorom (operacja rozgłaszania).
- Każdy dzieli swoją lokalną listę na dwie części: L_i przechowującą elementy mniejsze od podzielnika oraz U_i przechowującą elementy większe od podzielnika.
- Procesor z dolnej połowy przesyła swojemu partnerowi z górnej połowy listę U_i. Procesor partner z górnej połowy przesyła procesorowi z dolnej połowy listę L_i.
- Łatwo zauważyć, że po wykonaniu tego kroku wszystkie elementy mniejsze od podzielnika znajdują się w dolnej połowie, a elementy większe w górnej połowie.

Wydajna wersja dla maszyny z przesyłaniem komunikatów c.d.

- Rekurencja kończy się gdy połowa ma rozmiar pojedynczego procesora. W tym momencie możemy uporządkować zbiór procesorów w ciąg $P_1, P_2, \ldots, P_N$, tak że dla $i<j$ wszystkie elementy przechowywane w procesorze P_i są mniejsze od elementów przechowywanych w procesorze P_j.
- Dla pełnego posortowania potrzebne jeszcze jest aby każdy procesor uporządkował jeszcze swoją lokalną listę w czasie O(n/p log(n/p))
- Czas pojedynczego kroku O(n/p) (podział listy) + O(log p) (rozgłaszanie podzielnika).
- Średnio mamy O (log p) kroków, do tego należy dodać czas sortowania sekwencyjnego.
- Łączny czas: $O(n/p \log(n/p)) + O(n/p \log(p)) + O(\log^2 p)$.
- Uwaga: jednoczesna reorganizacja elementów bardzo obciąża sieć połączeń maszyny.

Co pominęliśmy

- Sieci sortujące: sieci składające się z kolumn elementarnych komparatorów postaci:



- Pierwotnie zaprojektowano je z myślą o sprzętowym rozwiązaniu problemu, ale istnieje możliwość ich zamapowania na komputer równoległy ze wspólną pamięcią lub z przesyłaniem komunikatów.

- Najpopularniejsza sieć bitoniczna pozwala na sortowanie w czasie $O(\log^2 n)$.