Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Algorytmy współbieżne
1. Wprowadzenie

Andrzej Karatkiewicz
Katedra Informatyki Stosowanej

Programowanie współbieżne

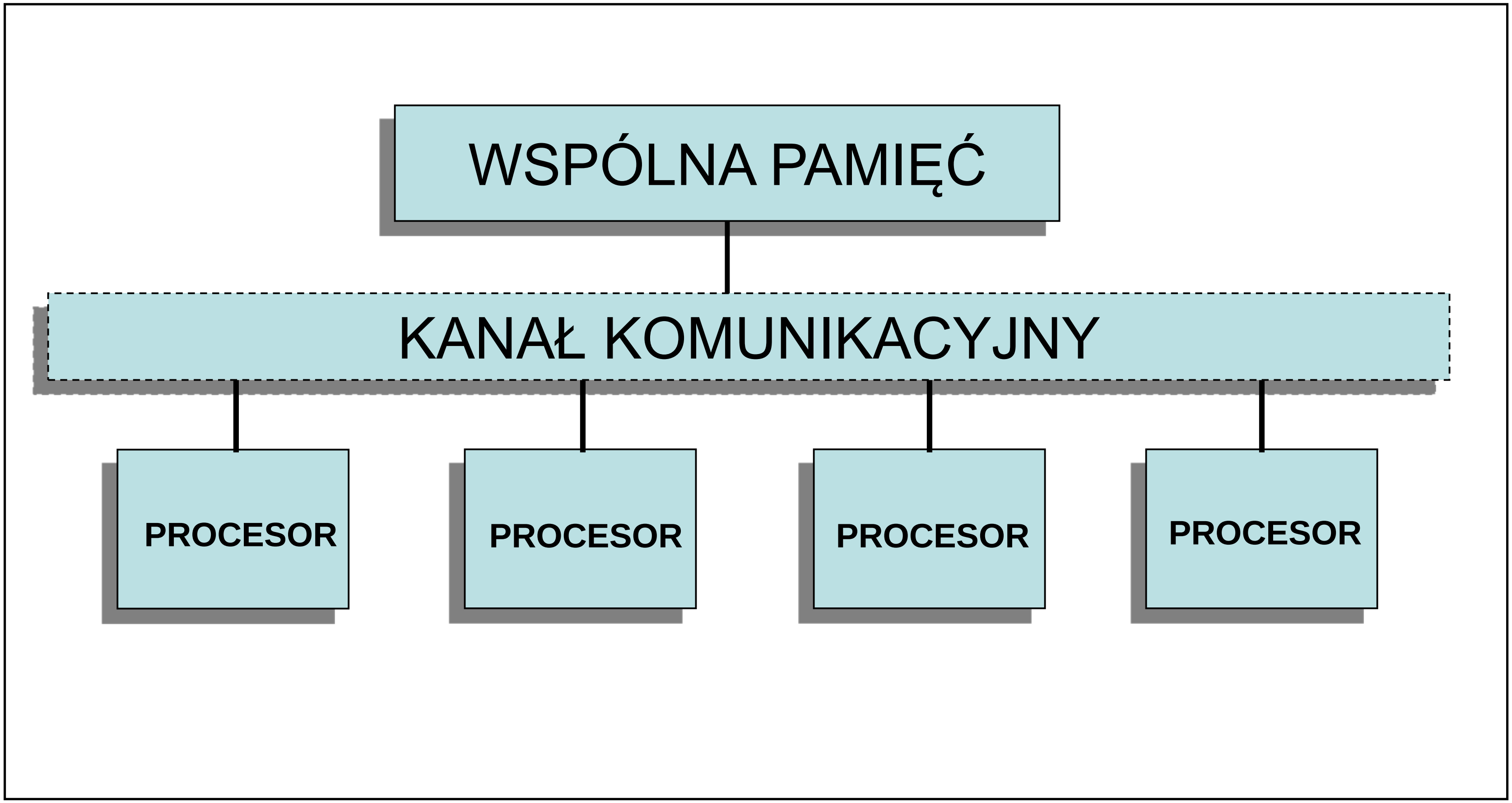
- Programowanie współbieżne dotyczy notacji i technik programowania umożliwiających specyfikację **potencjalnej równoległości** oraz rozwiązywanie zagadnień **synchronizacji i komunikacji**.
- Jak zrealizowana jest możliwość współbieżnego wykonywania programów jest problemem z dziedziny systemów komputerowych i w zasadzie wychodzi poza zakres programowania współbieżnego.
- Paradygmat programowania współbieżnego ustala pewne abstrakcyjne założenia do studiowania współbieżności bez wnikania w szczegóły implementacyjne.

Ben-Ari, „Podstawy programowania współbieżnego i rozproszonego”

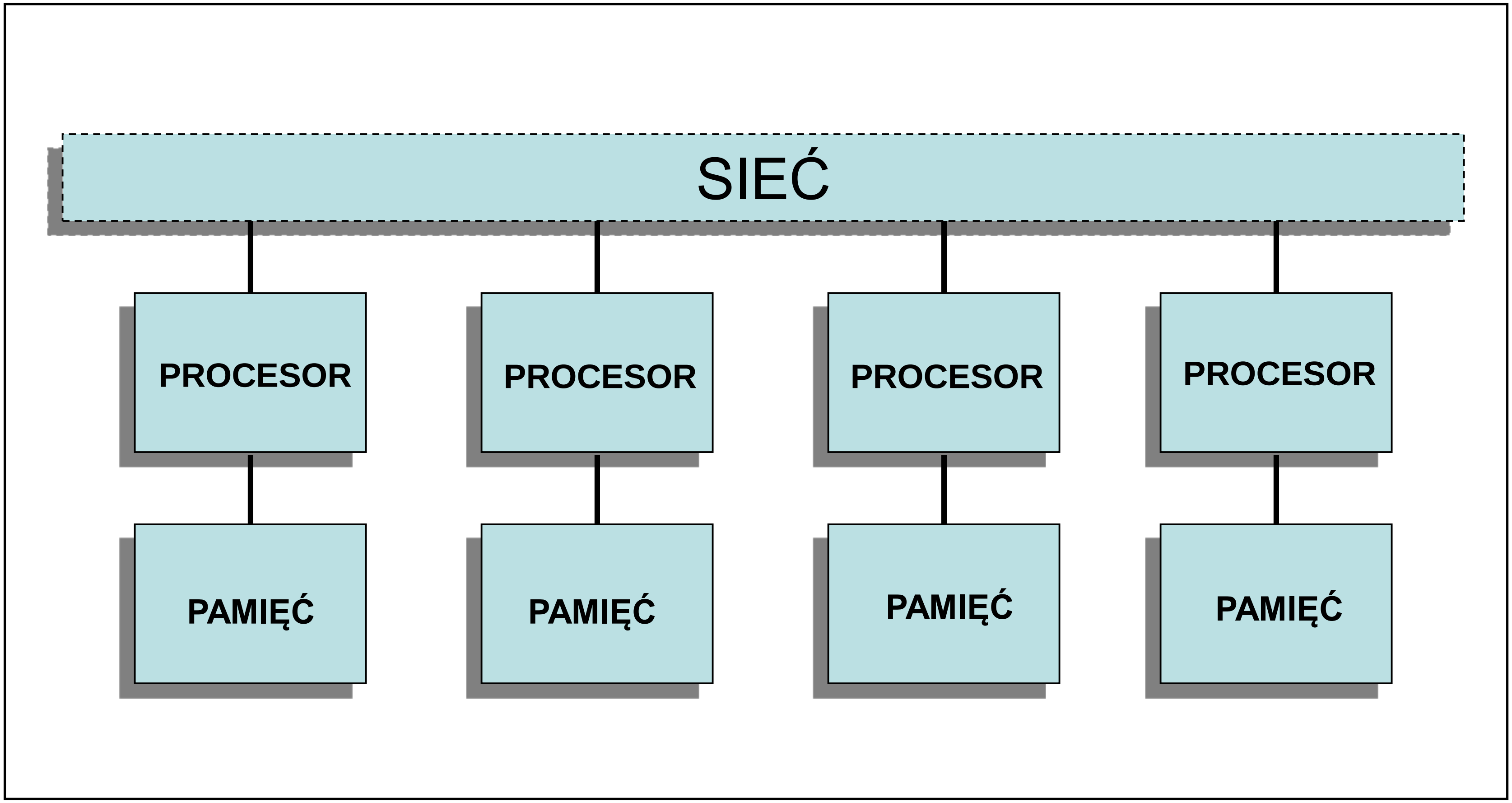
Algorytmy współbieżne

- **Algorytm współbieżny** (**równoległy**) – algorytm, który pozwala na wykonywanie w danej chwili więcej niż jednej operacji.
- **Program współbieżny** – program, w którym szereg instrukcji jest wykonywany jednocześnie.
- Algorytm/system współbieżny: procesy mają dostęp do wspólnej pamięci (zwykle pojedynczy komputer)
- Algorytm/system rozproszony: każdy proces ma osobną pamięć, komunikacja odbywa się za pomocą zdarzeń (events) (zwykle sieć komputerowa)

System współbieżny



System rozproszony



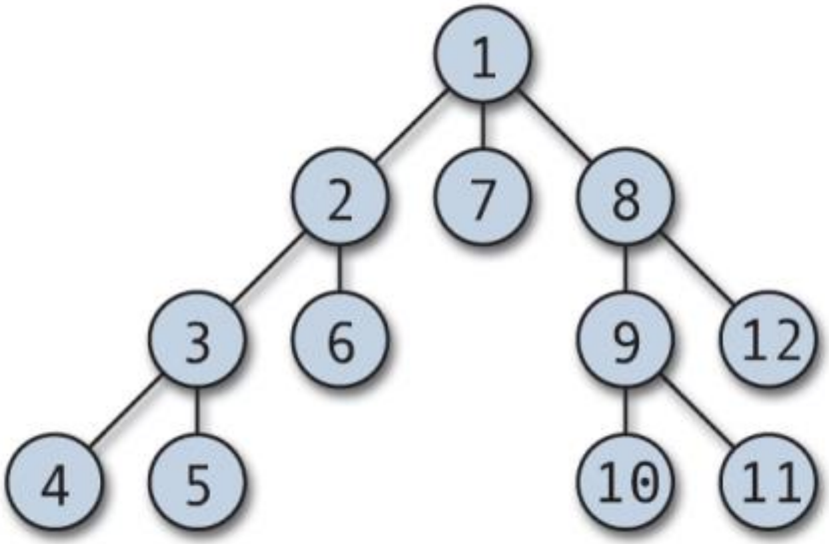
Algorytmy współbieżne

- Po co?
  - Nowoczesne komputery mają możliwość współbieżnego wykonania operacji.
  - Systemy operacyjne wspierają współbieżność i wielowątkowość.
  - Architektury wieloprocesorowe i sieci komputerowe stwarzają dodatkowe możliwości wykonania współbieżnego.
  - Współbieżne wykonanie operacji często pozwala na przyśpieszenie wykonania algorytmu.
  - Współbieżność jest potrzebna do sterowania współbieżnymi systemami.
- Czy to zawsze pomaga?
  - Jeśli sekwencyjne wykonanie zadania wymaga czasu T, czy wykonanie go przez n współbieżnych procesów będzie wymagało czasu T/n?
  - Nie (praktycznie nigdy)
  - Ale często ten czas okazuje się znacznie mniejszy.

Przykłady na „nie”

- Obliczanie kolejnych znaków liczby π
- Przeszukiwanie grafu w głąb
- Maksymalny przepływ w grafach

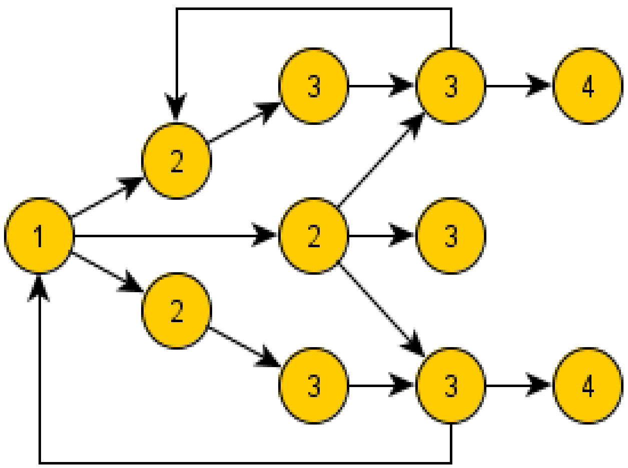
Oraz inne algorytmy, które prawie albo wcale nie ulegają podziałowi na równolegle operacje



Przykłady na „tak”

- Przeszukiwanie grafu wszerz
- Sortowanie
- Ustalanie porządku obiektów na liście
- Obliczenie minimum lub maksimum
- Wykonanie operacji łącznych (asocjatywnych) na szeregu danych

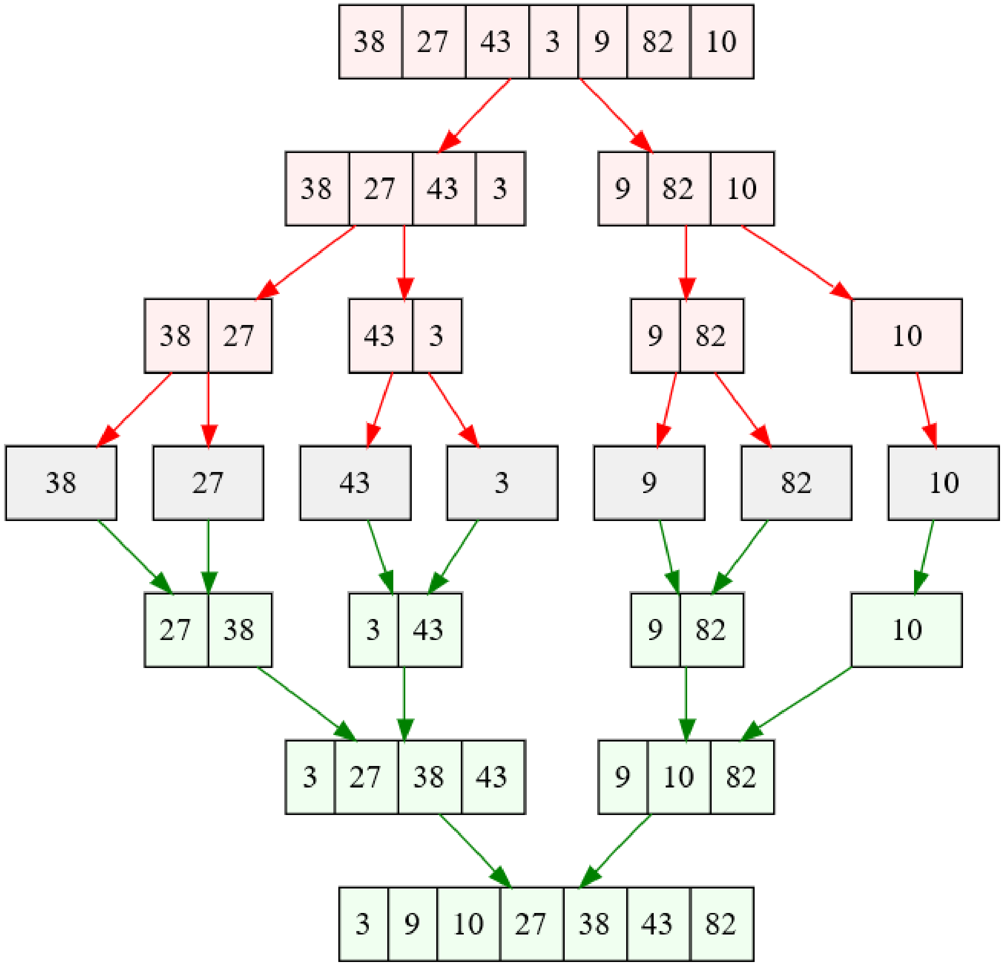
…oraz wiele innych…



Sortowanie (przez scalanie, Merge sort)

- Sekwencyjny algorytm sortowania nie może mieć złożoność czasową lepszą niż $O(n \log n)$.
- Sortowanie przez scalanie ma taką właśnie złożoność zarówno w najlepszym, jak i w najgorszym przypadku.

Sortowanie (przez scalanie, Merge sort)



Sortowanie (przez scalanie, Merge sort)

```
algorithm mergesort(A, lo, hi) is
    If lo < hi then // Two or more elements.
        mid = ⌊(lo + hi) / 2⌋;
        mergesort(A, lo, mid);
        mergesort(A, mid+1, hi);
        merge(A, lo, mid, hi);
```

Współbieżność w pseudokodzie

- Dla wielu procesów:

  for all $x \in A$ **in parallel do** instruction (x)

- Dla małej liczby procesów:

  fork

  instruction 1;

  instruction 2;

  join

Mergesort, wersja współbieżna:

```
algorithm mergesort(A, lo, hi) is
    If lo < hi then // Two or more elements.
        mid = ⌊(lo + hi) / 2⌋;
        fork
            mergesort(A, lo, mid);
            mergesort(A, mid+1, hi);
        join
        merge(A, lo, mid, hi);
```

Sortowanie (przez scalanie, Merge sort)

Jaką złożoność obliczeniową ma wersja współbieżna?

n + n/2 + n/4 + n/8 + … = 2n

Wersja współbieżna ma złożoność **liniową** – *O*(*n*)

Problemy NP

- W przypadku ogólnym nie są znane (i nie wiadomo czy istnieją) algorytmy wielomianowe
- Wszystkie znane algorytmy są (w najgorszym przypadku) wykładnicze
- Sprawdzenie poprawności zaproponowanego rozwiązania wymaga czasu wielomianowego
- Rozwiązanie za pomocą *niedeterministycznej maszyny Turinga* wymaga czasu *wielomianowego*

Problemy NP: sprawdzanie spełnialności formuły logicznej

- Problem spełnialności

czy dla danej formuły logicznej istnieje takie podstawienie zmiennych, żeby formuła była prawdziwa?

Algorithm satisfability(*f*) **is**
    for every *x* – argument *f*
        podstaw 0 zamiast *x*;
        oblicz *f'* – czyli *f* po podstawieniu 0 zamiast *x*;
        if *f'* = 1 **then return** *true*;
        if satisfability(*f'*) = ***true*** **then return** *true*;
        podstaw 1 zamiast *x*;
        oblicz *f'* – czyli *f* po podstawieniu 1 zamiast *x*;
        if *f'* = 1 **then return** *true*;
        if satisfability(*f'*) = ***true*** **then return** *true*;
    return *false*;

Problemy NP: sprawdzanie spełnialności formuły logicznej (wersja mocno współbieżna)

- Problem spełnialności

czy dla danej formuły logicznej istnieje takie podstawienie zmiennych, żeby formuła była prawdziwa?

Algorithm parallel_satisfability(*f*) **is**
    for every *x* – argument *f*
        fork
        {
            podstaw 0 zamiast *x*;
            oblicz *f'* – czyli *f* po podstawieniu 0 zamiast *x*;
            if *f'* = 1 **then return** *true*;
            if satisfability(*f'*) = ***true*** **then return** *true*;
        }
        {
            podstaw 1 zamiast *x*;
            oblicz *f'* – czyli *f* po podstawieniu 1 zamiast *x*;
            if *f'* = 1 **then return** *true*;
            if satisfability(*f'*) = ***true*** **then return** *true*;
        }
        join;
    return *false*;

Warianty implementacji współbieżności

- „prawdziwa” współbieżność – różne operacje są fizycznie wykonywane w tym samym czasie (możliwa do zrealizowania w systemie wieloprocesorowym);
- wykonanie „w przeplocie” – w danym momencie jest wykonywana jedna operacja, ale zachodzi przełączanie między wykonywanymi procesami, które są wykonywane na zmianę krótkimi fragmentami

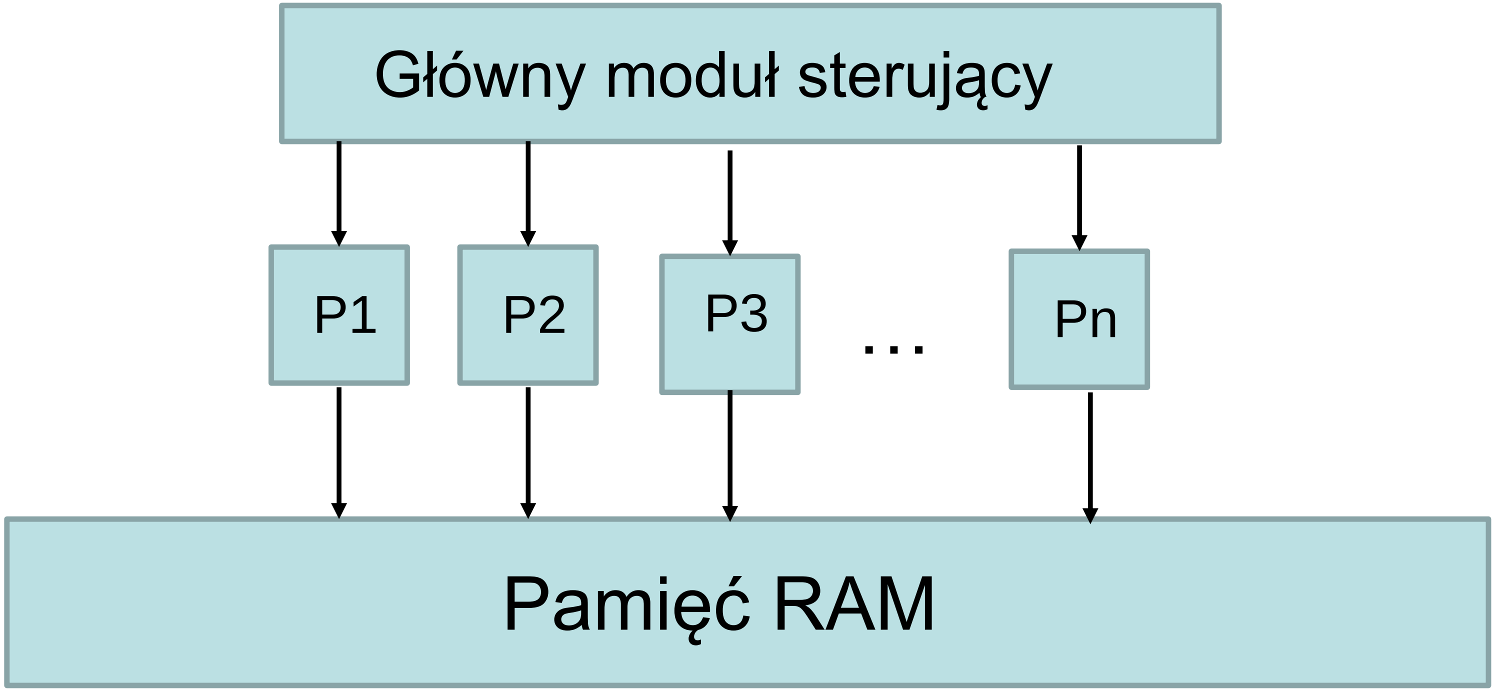
Algorytm współbieżny jest abstrakcją – nie uwzględnia konkretnego sposobu implementacji.

Oszacowywać można:

- czas działania algorytmu;
- **pracę**, wykonywaną przez algorytm: sumaryczny czas działania wszystkich procesorów.

Model obliczeń równoległych

P-RAM (parallel random access machine)



Warianty modelu obliczeniowego

- Podstawowy: **P-RAM**
- Warianty P-RAM:
  - **EREW** - wyłączny odczyt i wyłączny zapis;
  - **CREW** - jednoczesny odczyt i wyłączny zapis;
  - **ERCW** - wyłączny odczyt i jednoczesny zapis (w praktyce się nie zdarza);
  - **CRCW** - jednoczesny odczyt i jednoczesny zapis
- **W-RAM**: ograniczona wersja CRCW
  - Jednoczesny zapis do wybranych lokacji
  - Lokacje przechowują wartości binarne
  - Współbieżnie można zapisać tylko 1
- Zakładamy SIMD (single instruction, multiple data)

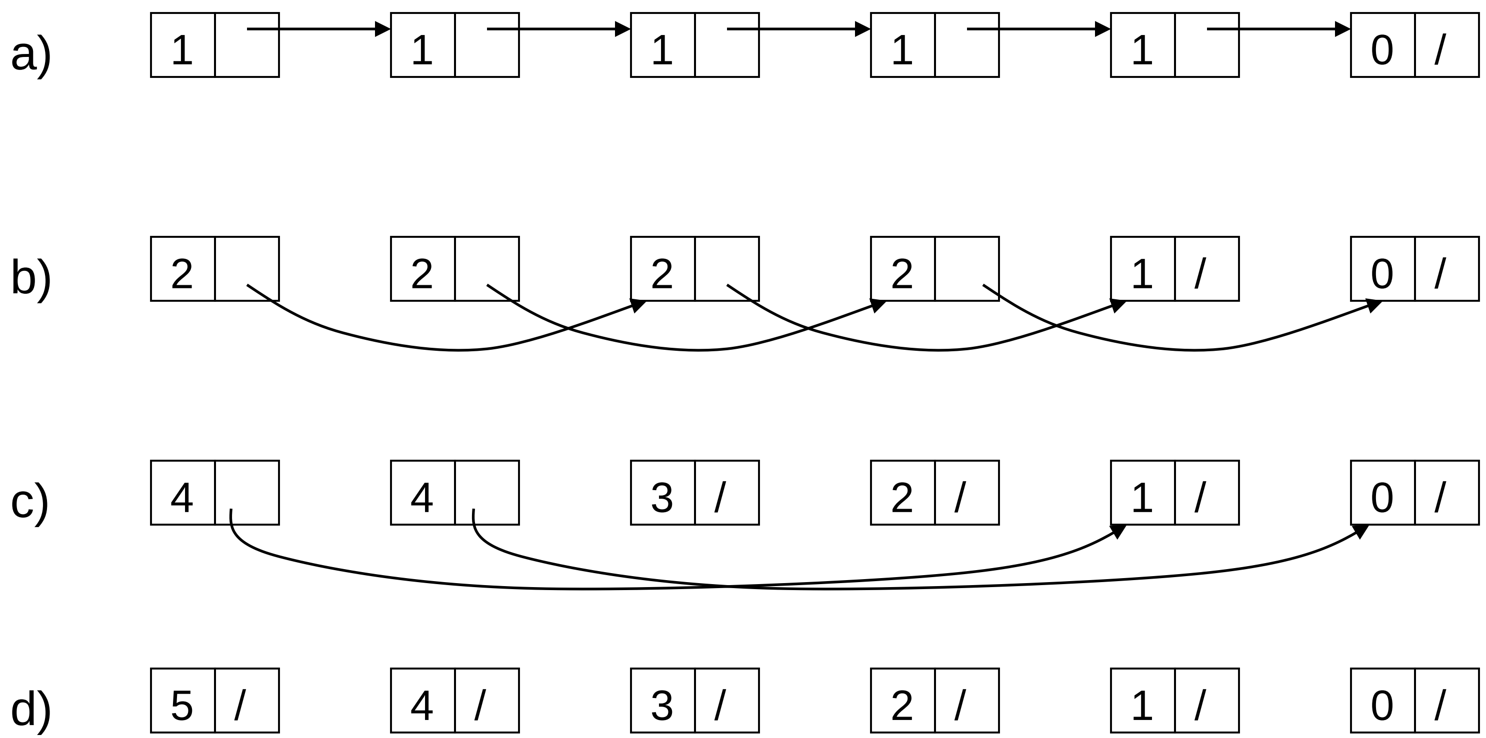
Ustalanie porządku obiektów na liście

Dana n-obiektowa lista L, chcemy dla każdego obiektu obliczyć jego odległość od końca listy. Algorytm sekwencyjny potrzebuje czasu $\Theta(n)$

LIST-RANK(L)

for każde i **in parallel**
    do if $next[i]$ = NIL
        then d[i] := 0
        else d[i] := 1
while istnieje obiekt i taki, że $next[i] \neq$ NIL
    do for każde i **in parallel**
        do if $next[i] \neq$ NIL
            then $d[i] := d[i]+d[next[i]]$
                $next[i] := next[next[i]]$

Ustalanie porządku obiektów na liście



Analiza algorytmu ustalania porządku

W algorytmie jest zachowywany następujący niezmiennik: dla każdego obiektu *i* na początku każdej iteracji pętli **while** suma wartości *d* w podliście o początku w *i* jest równa odległości obiektu *i* od końca pierwotnej listy *L*.

W każdej iteracji przeskoki wskaźników powodują przekształcenie każdej listy na dwie: z obiektów na pozycjach parzystych i nieparzystych. Każdy krok podwaja liczbę list i zmniejsza o połowę ich długości. Dlatego po wykonaniu lg *n* iteracji wszystkie listy zawierają po jednym obiekcie.

Czas działania procedury wynosi $\Theta(\log n)$, a liczba zastosowanych procesorów wynosi $\Theta(n)$, więc procedura wykonuje *pracę* $\Theta(n \log n)$.

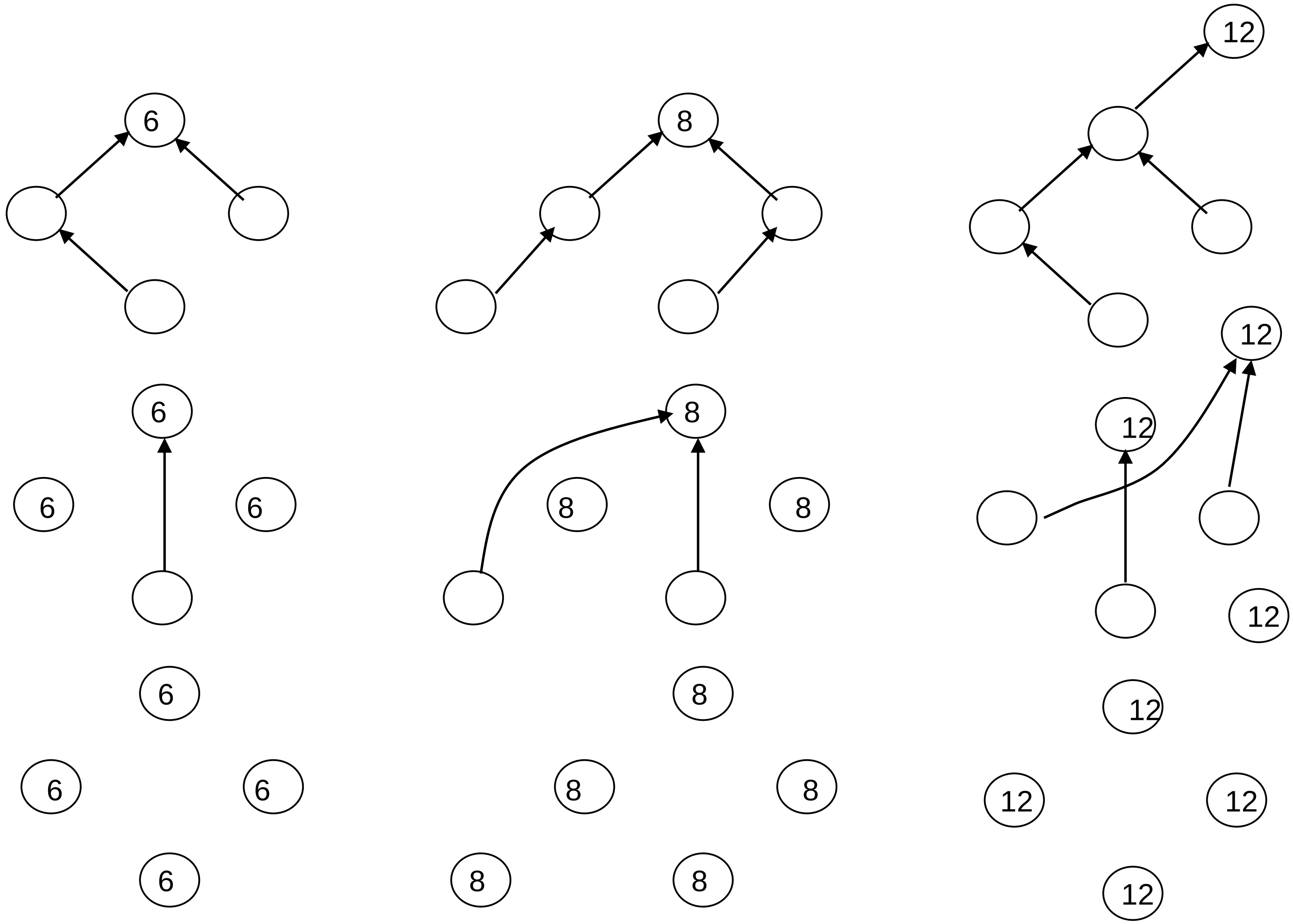
Znajdowanie korzeni w lesie drzew binarnych (algorytm typu CREW)

Niech dany jest las drzew binarnych, w którym każdy węzeł ma wskaźnik do swojego ojca, i chcemy znaleźć dla każdego węzła identyfikator korzenia drzewa, do którego ten węzeł należy.

FIND-ROOTS(*F*)

for każde *i* **in parallel**
    do if *parent*[*i*] = NIL
        then *root*[*i*] := *i*
while istnieje węzeł *i* taki, że *parent*[*i*] ≠ NIL
    do for każde *i* **in parallel**
        do if *parent*[*i*] ≠ NIL
            then *root*[*i*] := *root*[*parent*[*i*]]
                 parent[*i*] := *parent*[*parent*[*i*]]

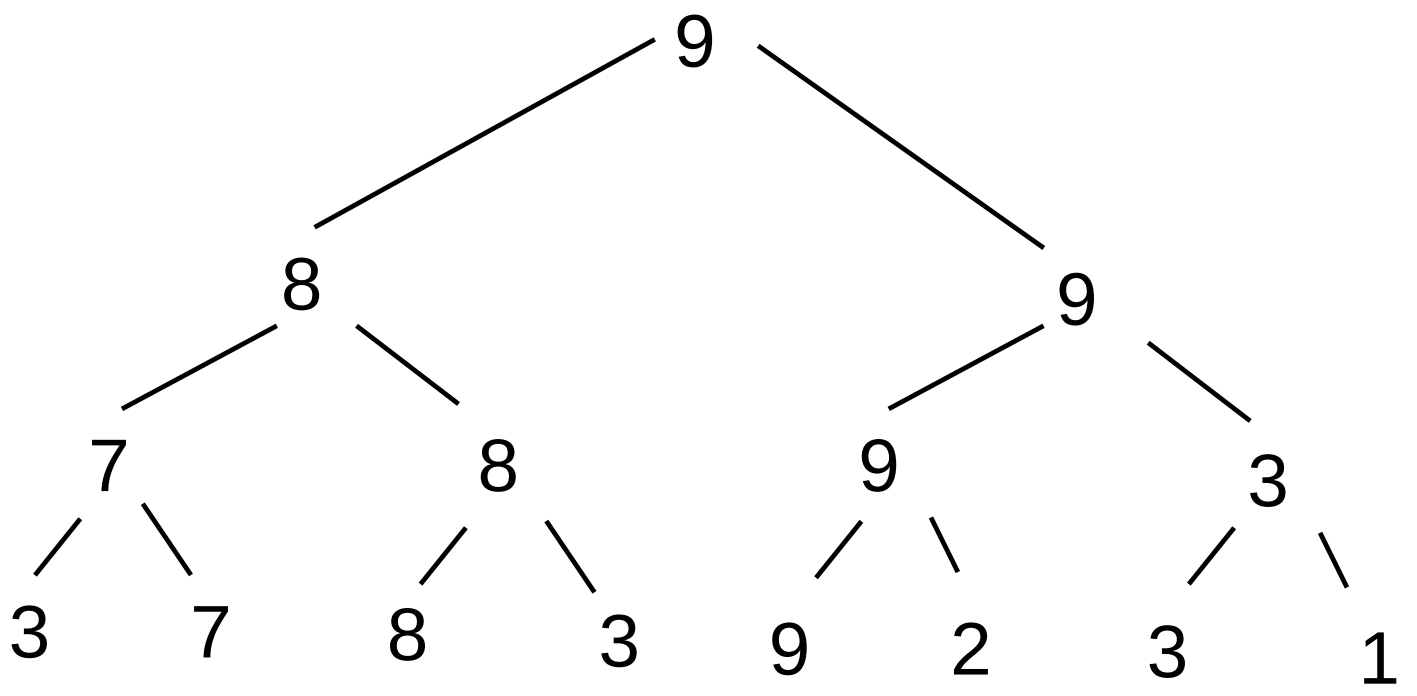
Znajdowanie korzeni w lesie drzew binarnych



Generalne techniki algorytmów współbieżnych

- Metoda drzewa binarnego
- Metoda podwajania (the doubling technique, pointer jumping)
- Metoda „dziel i zwyciężaj”
- Metoda kompresji

Metoda drzewa binarnego



Podwajanie, „dziel i zwyciężaj”

- Metoda podwajania: stosuje się do list i podobnych struktur (przykłady powyżej – ustalanie porządku, znajdowanie korzeni)

- „Dziel i zwyciężaj” – szeroka klasa przypadków, m. in. Parallel merge sort, obliczenie wielomianów

Metoda kompresji (collapsing technique)

$[3,7,8,3,9,2,3,1] \rightarrow [7,8,9,3] \rightarrow [8,9] \rightarrow [9]$

Jest skuteczna dla szeregu zagadnień, w tym grafowych (e.g. znajdowanie spójnych podgrafów)

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Algorytmy współbieżne
2. Sortowanie współbieżne
(wybrane algorytmy)

Andrzej Karatkiewicz

Sortowanie współbieżne

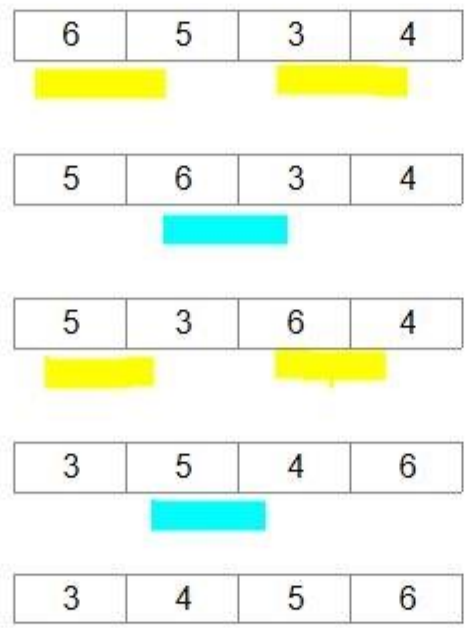
Najbardziej oczywiste podejście (mając k procesorów):

- Dzielimy sortowany ciąg na k podciągów o jednakowej długości

```
size = int(math.ceil(float(len(data)) / processes))
data = [data[i * size:(i + 1) * size] for i in range(processes)]
```

- Każdy procesor sortuje swoja część
- Scalamy wyniki (parami)
- Wąskie gardło: scalanie

Współbieżne sortowanie bąbelkowe



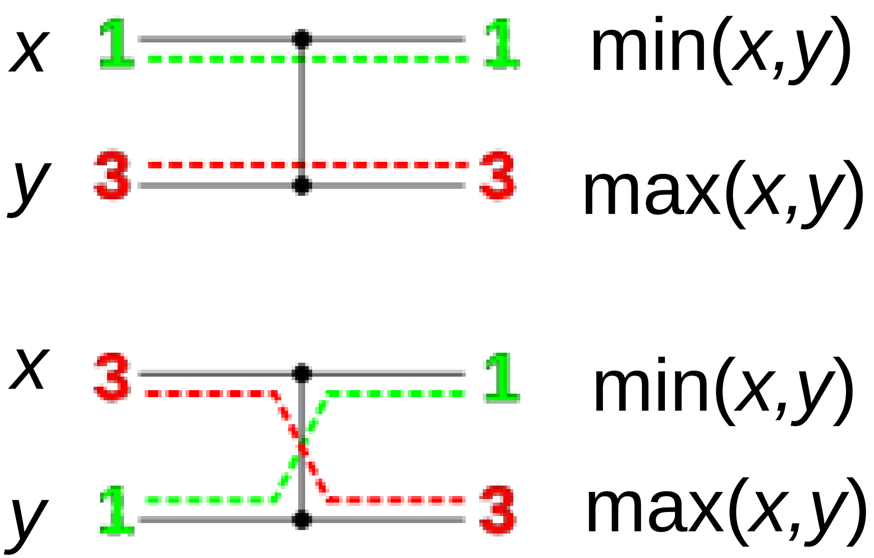
Wykonujemy porównanie par na przemian
(i,i-1); (i,i+1)

Współbieżne scalanie (jedna z możliwości)

```
algorithm merge(A[i...j], B[k...ℓ], C[p...q]) is
    inputs A, B, C : array
           i, j, k, ℓ, p, q : indices

    let m = j - i,
        n = ℓ - k

    if m < n then
        swap A and B  // ensure that A is the larger array: i, j still belong to A; k, ℓ to B
        swap m and n

    if m ≤ 0 then
        return  // base case, nothing to merge

    let r = ⌊(i + j)/2⌋
    let s = binary-search(A[r], B[k...ℓ])
    let t = p + (r - i) + (s - k)
    C[t] = A[r]

    in parallel do
        merge(A[i...r], B[k...s], C[p...t])
        merge(A[r+1...j], B[s...ℓ], C[t+1...q])
```

Sieć sortująca

- Abstrakcyjny, matematyczny model sieci, składającej się z przewodów i modułów porównujących (komparatorów), używanej do sortowania sekwencji liczb.
- Komparator:

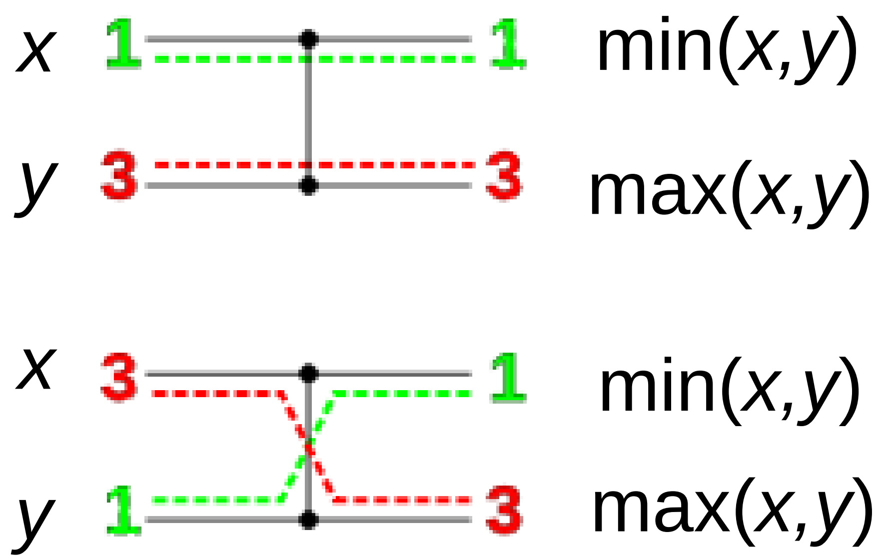


Komparator i odpowiednia operacja

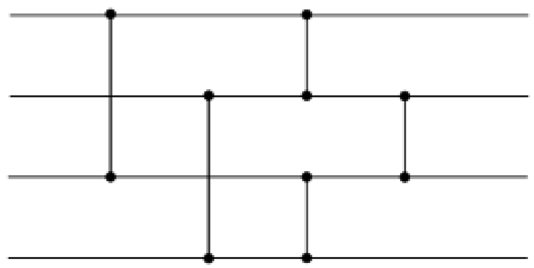
compare-exchange(*i*,*j*)

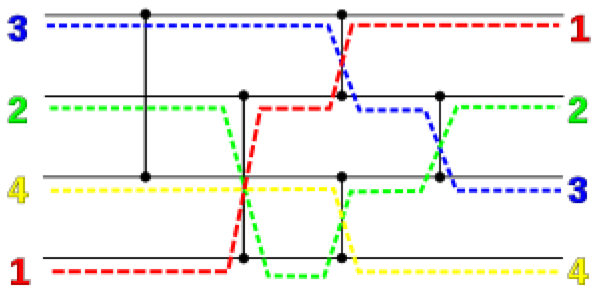
if *Key*[*i*]>*Key*[*j*] then

exchange values of *Key*[*i*], *Key*[*j*]

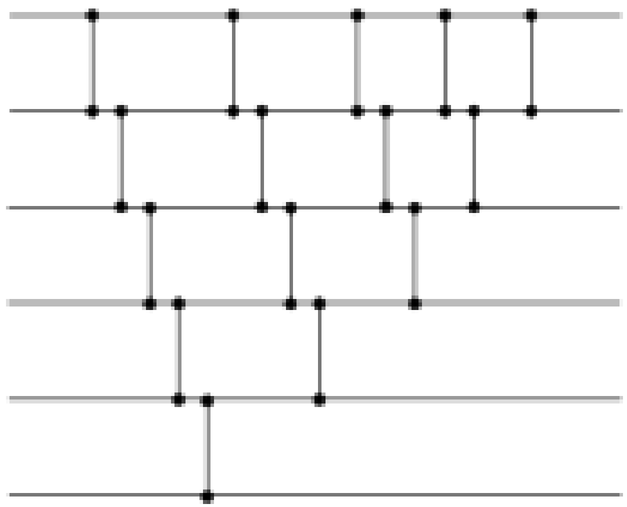


Sieć sortująca

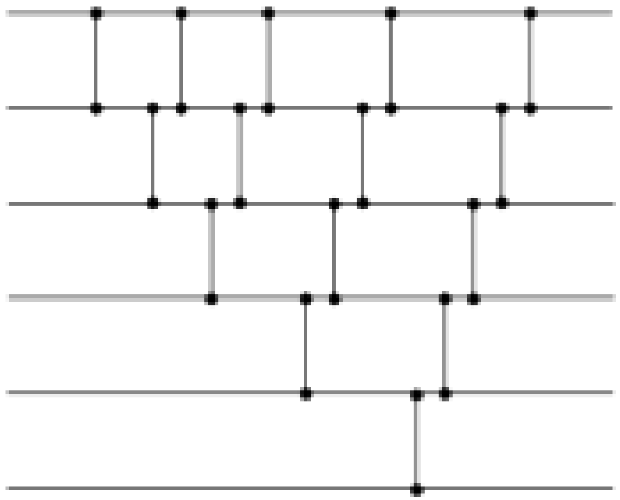




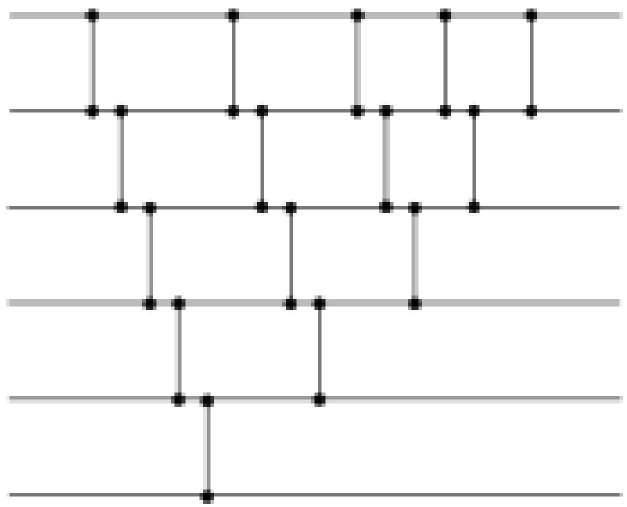
Sieć sortująca bąbelkowo

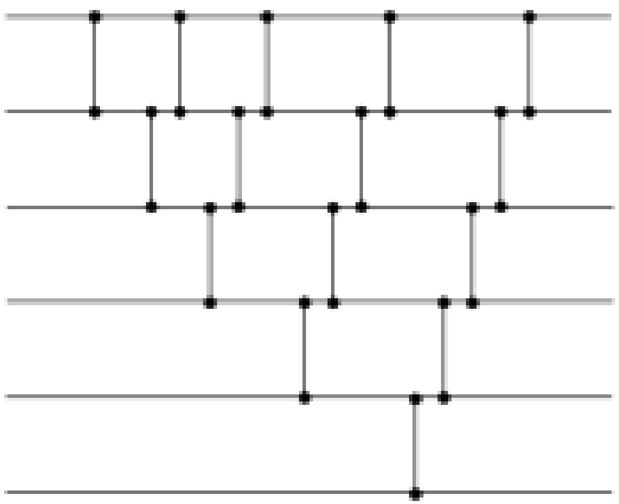


Sieć sortująca przez wstawianie

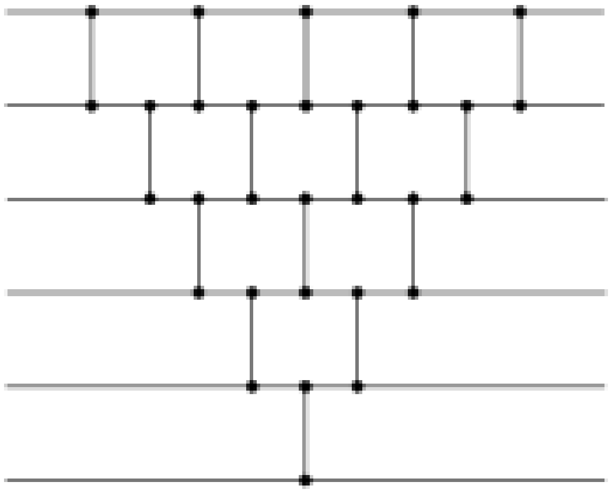


Porównujmy:

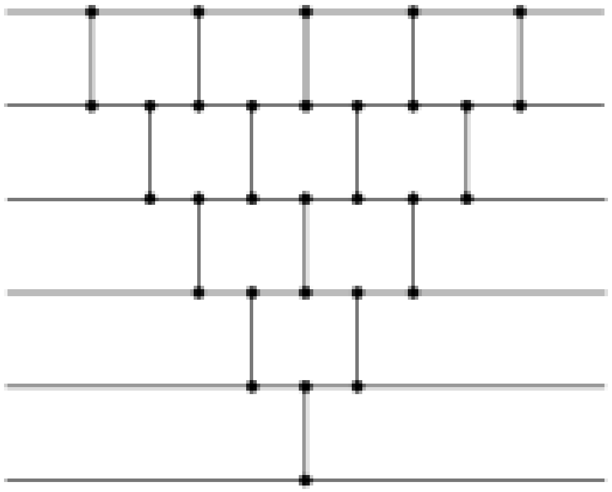




Jeśli możemy to robić współbieżnie:

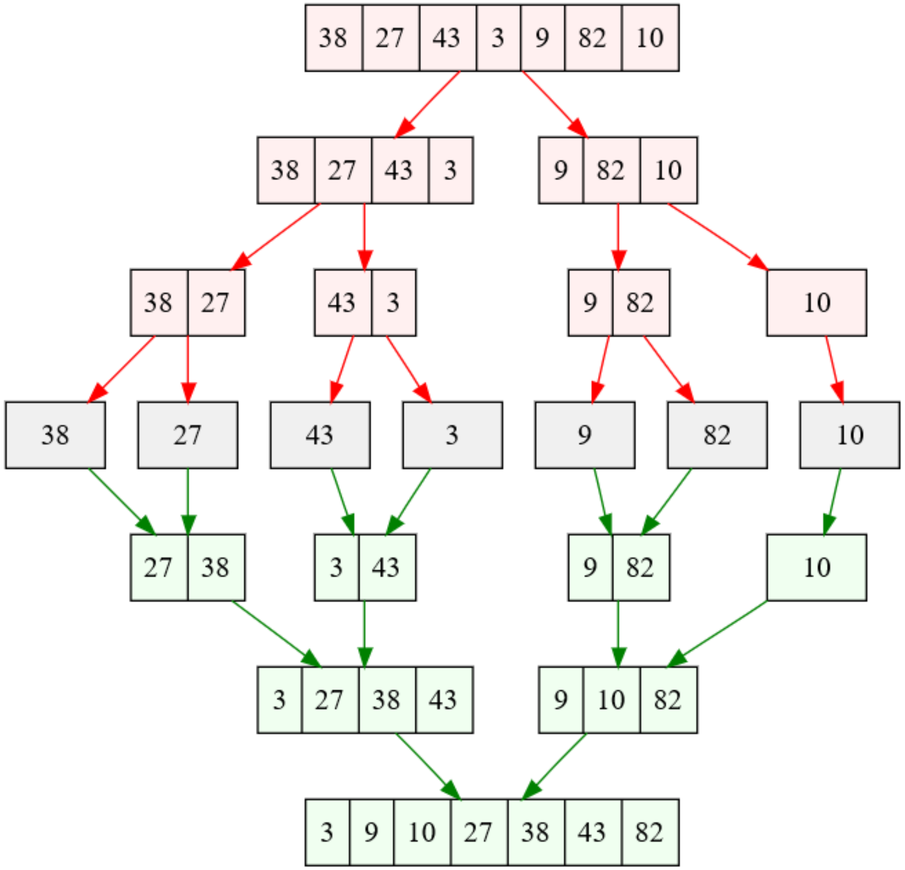


Jeśli możemy to robić współbieżnie:



$O(n)$ zamiast $O(n^2)$

Przypomnijmy sobie Mergesort:



Wąskie gardło: scalanie

Operacje, potrzebne w dalszym ciągu:

- *odd*(S) – podciąg S z elementów nieparzystych

  odd(2,3,4,5) = (2,4)

- *even*(S) – podciąg S z elementów parzystych

  odd(2,3,4,5) = (3,5)

- *interleave*(S_1,S_2) – podciąg S z elementów naprzemiennie z S_1 i S_2

  interleave((2,3,4,5),(6,7,8,9)) = (2,6,3,7,4,8,5,9)

Operacje, potrzebne w dalszym ciągu:

- *odd-even*(*S*) – współbieżne wykonanie *compare-exchange(i, i+1)* dla każdego parzystego *i* w zakresie *S* (indeksacja od 1)

  odd-even(1,6,2,4,3,10) = (1,2,6,3,4,10)

- *join1*(S_1,S_2) – *odd-even*(*interleave*(S_1,S_2))

  join1((1,2,3,4),(5,6,7,8)) = *odd-even*(1,5,2,6,3,7,4,8) =

  = (1,2,5,3,6,4,7,8)

Scalanie Batchera

odd-even-merge(*S*,*S'*) // *S*,*S'* - posortowane ciągi

begin

if len(S) = len(S') = 1 **then** merge them using one *compare-exchange*

else

compute S_{odd} and S_{even} **in parallel**:

S_{odd} := *odd-even-merge*(*odd*(*S*),*odd*(*S'*))

S_{even} := *odd-even-merge*(*even*(*S*),*even*(*S'*))

return *join1*(S_{odd}, S_{even})

end

Scalanie Batchera

odd-even-merge(*S*,*S'*) // *S*,*S'* - posortowane ciągi

begin

if len(S) = len(S') = 1 **then** merge them using one *compare-exchange*

else

compute S_{odd} and S_{even} **in parallel**:

S_{odd} := *odd-even-merge*(*odd*(*S*),*odd*(*S'*))

S_{even} := *odd-even-merge*(*even*(*S*),*even*(*S'*))

return *join1*(S_{odd}, S_{even})

end

S = (2,6,10,15)

S' = (3,4,5,8)

Scalanie Batchera

odd-even-merge(*S*,*S'*) // *S*,*S'* - posortowane ciągi

begin

if len(S) = len(S') = 1 **then** merge them using one *compare-exchange*

else

compute S_{odd} and S_{even} **in parallel**:

S_{odd} := *odd-even-merge*(*odd*(*S*),*odd*(*S'*))

S_{even} := *odd-even-merge*(*even*(*S*),*even*(*S'*))

return *join1*(S_{odd}, S_{even})

end

S = (2,6,10,15)

S' = (3,4,5,8)

1 wywołanie rekurencyjne:

odd-even-merge((2,10),(3,5))

Scalanie Batchera

odd-even-merge(*S*,*S'*) // *S*,*S'* - posortowane ciągi

begin

if len(S) = len(S') = 1 **then** merge them using one *compare-exchange*

else

compute S_{odd} and S_{even} **in parallel**:

S_{odd} := *odd-even-merge*(*odd*(*S*),*odd*(*S'*))

S_{even} := *odd-even-merge*(*even*(*S*),*even*(*S'*))

return *join1*(S_{odd}, S_{even})

end

S = (2,6,10,15)

S' = (3,4,5,8)

1 wywołanie rekurencyjne:

odd-even-merge((2,10),(3,5))

Następne wywołania:

odd-even-merge((2),(3)) = (2,3)

odd-even-merge((10),(5)) = (5,10)

join1((2,3),(5,10)) =

= *odd-even*(2,5,3,10) = (2,3,5,10)

Scalanie Batchera

odd-even-merge(*S*,*S'*) // *S*,*S'* - posortowane ciągi

begin

if len(S) = len(S') = 1 **then** merge them using one *compare-exchange*

else

compute S_{odd} and S_{even} **in parallel**:

S_{odd} := *odd-even-merge*(*odd*(*S*),*odd*(*S'*))

S_{even} := *odd-even-merge*(*even*(*S*),*even*(*S'*))

return *join1*(S_{odd}, S_{even})

end

S = (2,6,10,15)

S' = (3,4,5,8)

2 wywołanie rekurencyjne:

odd-even-merge((6,15),(4,8))

Następne wywołania:

odd-even-merge((6),(4)) = (4,6)

odd-even-merge((15),(8)) = (8,15)

join1((4,6),(8,15)) =

= *odd-even*(4,8,6,15) = (4,6,8,15)

Scalanie Batchera

odd-even-merge(*S*,*S'*) // *S*,*S'* - posortowane ciągi

begin

if len(S) = len(S') = 1 **then** merge them using one *compare-exchange*

else

compute S_{odd} and S_{even} **in parallel**:

S_{odd} := *odd-even-merge*(*odd*(*S*),*odd*(*S'*))

S_{even} := *odd-even-merge*(*even*(*S*),*even*(*S'*))

return *join1*(S_{odd}, S_{even})

end

Głębokość logarytmiczna

$O(\log n)$

Scalanie bitoniczne (też Batchera)

// Potrzebne procedury

shuffle(*S*)   // tasowanie
begin
Podziel *S* na dwa ciągi:
    S_1 – pierwsze n/2 elementy
    S_2 - pozostałe n/2 elementy
S := interleave(S_1, S_2)
end

shuffle(1,2,3,4, 5,6,7,8) = (1,5,2,6,3,7,4,8)

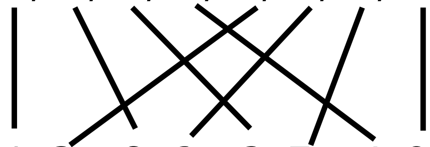
Scalanie bitoniczne

// Potrzebne procedury

join2(*S*)
begin
shuffle(*S*)
for all odd i, 1≤*i*≤*n*, **in parallel do**
compare-exchange(*i, i+1*)

end

1, 8, 2, 4, 3, 6, 5, 9



1-3 8-6 2-5 4-9
1, 3, 6, 8, 2, 5, 4, 9

Ciąg bitoniczny

Ciąg który najpierw jest niemalejący, a potem nierosnący

(3,5,10,4,1), (1,5), (10,14,5,-1,-4) - tak

(4,6,1,9,2) - nie

Monotoniczny ciąg też jest ciągiem bitonicznym

Każdy ciąg o długości 1 lub 2 jest ciągiem bitonicznym.

Każdy ciąg jest konkatenacją ciągów bitonicznych o długości 2!

Scalanie bitoniczne

// S to ciąg bitoniczny długości n, niemalejąca i nierosnąca części tej samej długości – n/2

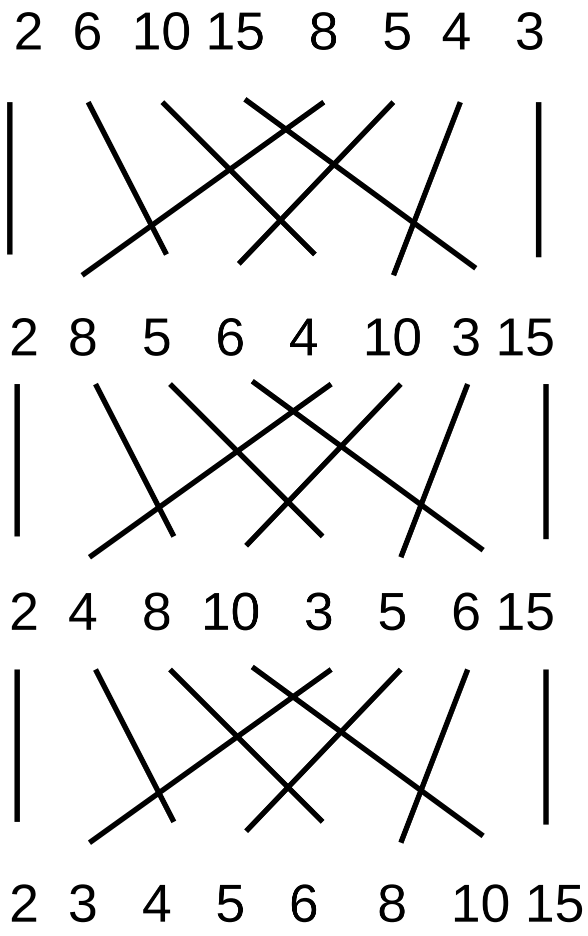
Bitonic-merge(S)

begin

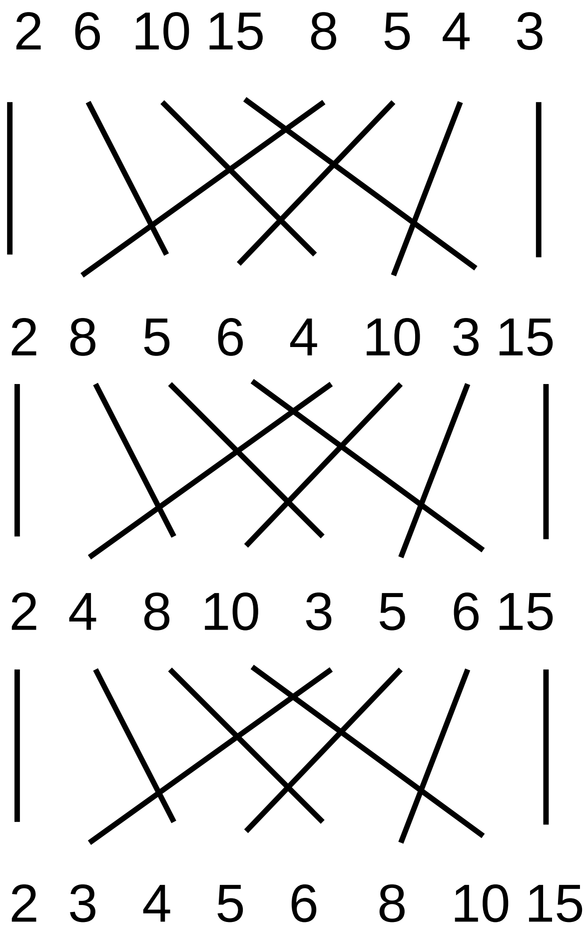
repeat *log n* times join2(S)

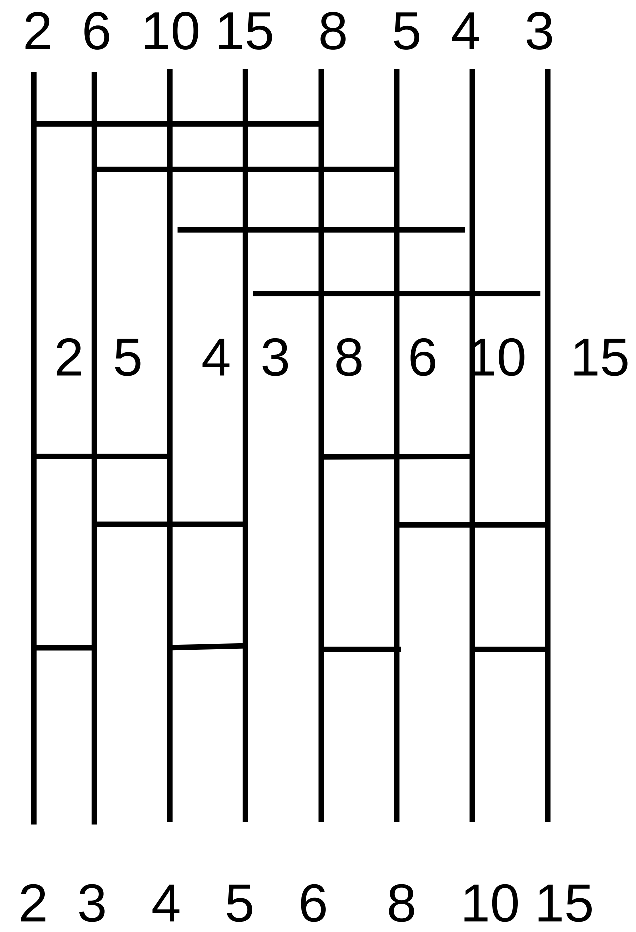
end

Scalanie bitoniczne: przykład

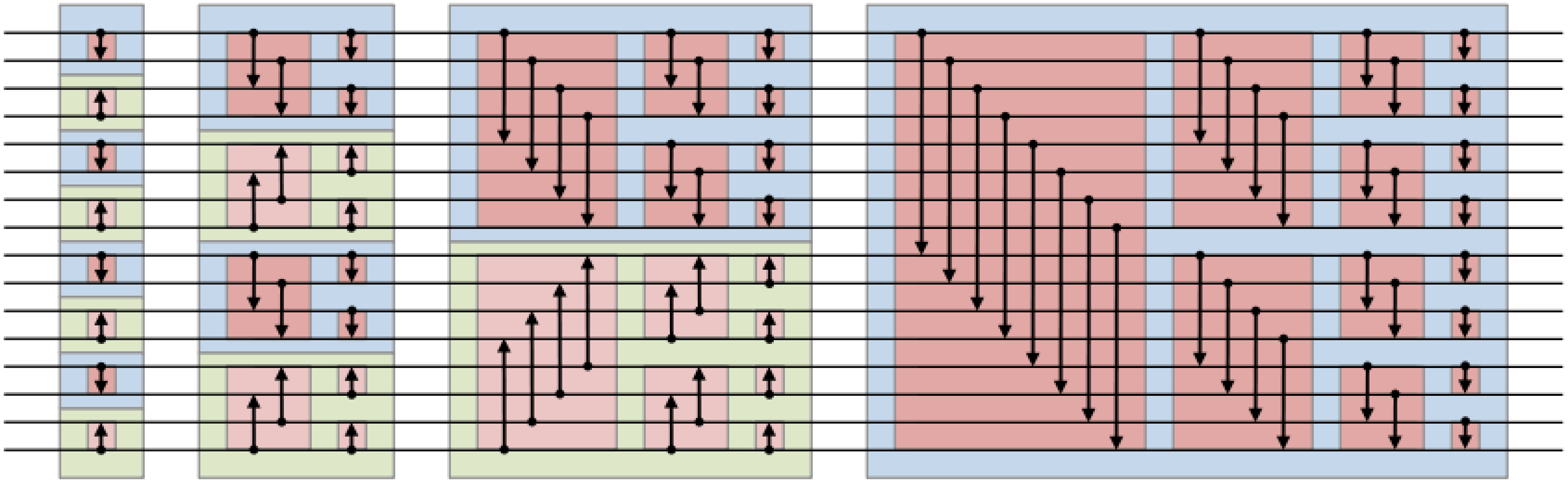


Scalanie bitoniczne: czemu to działa?





I sortowanie na tej podstawie:



Każdy ciąg jest konkatenacją ciągów bitonicznych o długości 2!

Sortowanie bitoniczne

- Klasyczny algorytm sortowania współbieżnego
- Złożoność – $O(log^2\ n)$
- Często stosowany w procesorach graficznych i PLD (układach programowalnych)

Współbieżny Python

- Pakiety:
  - threading – dla wykonania w sieci (albo wielowątkowości na jednym procesorze)
  - multiprocessing – pozwala na wykonanie współbieżne na komputerze mającym kilka procesorów

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Algorytmy współbieżne
Współbieżność w Pythonie

Andrzej Karatkiewicz

Współbieżny Python

- Pakiety:
  - threading – dla wykonania w sieci (albo wielowątkowości na jednym procesorze)
  - multiprocessing – pozwala na wykonanie współbieżne na komputerze mającym kilka procesorów

- W ramach jednego procesu może być kilka wątków
  - Wątek – najmniejsza część programu (sekwencja instrukcji), która może zostać przydzielona do wykonania przez system operacyjny
  - Proces ma swoje zasoby takie jak przestrzeń adresową, wątki wewnątrz procesu współdzielą te zasoby
- Proces wykonuje się na jednym procesorze

Ile może być współbieżnych procesów?

- Fizycznie – najwyżej tyle ile jest procesorów w komputerze (logicznie – znacznie więcej)
- Jak się dowiedzieć, ile jest procesorów?

```
import multiprocessing as mp
print("Number of processors: ", mp.cpu_count())
```

Multiprocessing: getting started

```
import multiprocessing
from multiprocessing import Pool

def f(x):
    return x*x

inputs = [6,7,8,9,0,10]

if __name__ == '__main__':           # „entry point”
    p = Pool(5)                      # multiprocessing.Pool() zawiera
                                     # m.in. funkcje apply() i map(),
    T = p.map(f, inputs)             #pozwalające na współbieżne wykonanie
    p.close()                        # map() przekazuje argumenty z listy
    p.join()              #close() nie pozwala przydzielać nowych zadań
    print(T)                         #join() czeka na koniec procesu
```

Multiprocessing

- Pool(processes)   (pula)

„Process pool” – obiekt, kontrolujący pulę procesów, do których może być przydzielona praca.

Domyślnie procesów tyle, ile zwraca cpu.count

- pool.map(funkcja, inputs)

Dzieli listę „inputs” na kawałki, przydzielane do procesów. Funkcja jest wywoływana przez któryś z procesów dla każdego z elementów listy

Multiprocessing

- Uruchomienie pojedynczego procesu:

```
p = Process(funkcja, args)   #tworzy obiekt
p.start()                    #uruchamia
```

- Pamięć dzielona: Value i Array

```
num = Value('d', 0.0)        #'d' – double
arr = Array('i', range(10))  #'i' – integer
```

- Szczegóły:

https://docs.python.org/3/library/multiprocessing.html