PRZYKŁADOWE RÓWNOLEGŁE ALGORYTMY NUMERYCZNE

LESŁAW SIENIAWSKI © 2017

MNOŻENIE MACIERZY

DEFINICJA ILOCZYNU MACIERZY

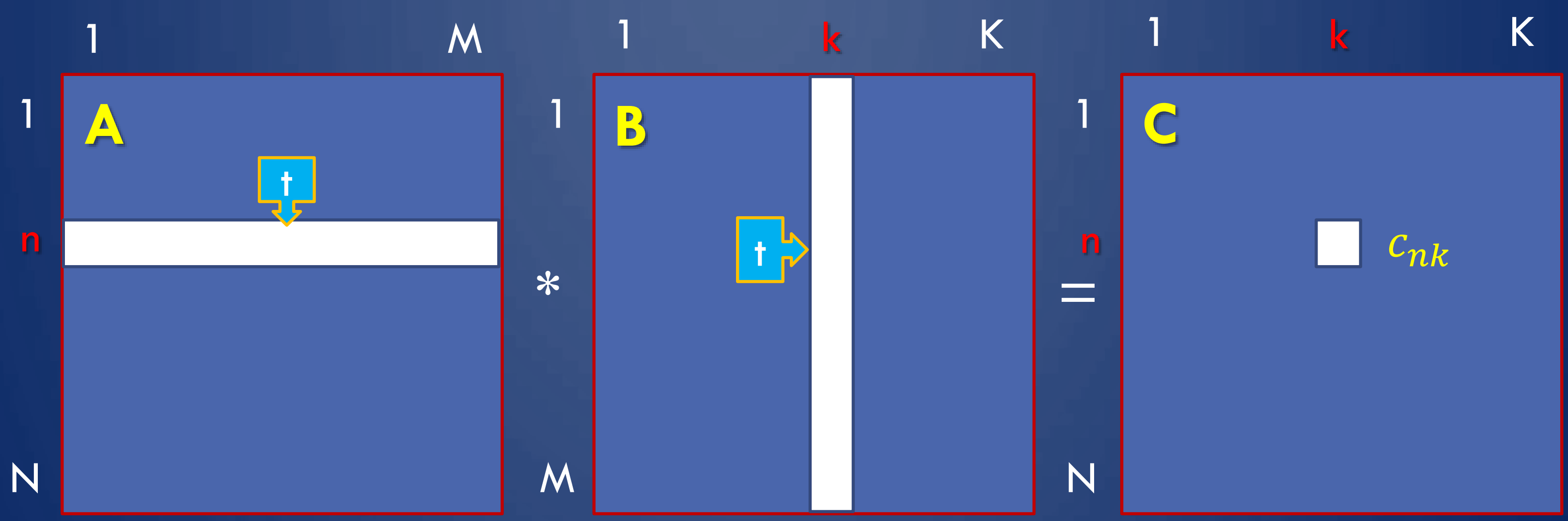
Dane są macierze:

- $A\ [N, M]$ – elementy a_{nm}
- $B\ [M, K]$ – elementy b_{mk}

Iloczynem macierzy A i B jest macierz $C\ [N, K]$, której elementy są określone jako

$$c_{nk} = \sum_{t=1}^{M} a_{nt}\, b_{tk}$$

SCHEMAT MNOŻENIA MACIERZY



$$c_{nk} = \sum_{t=1}^{M} a_{nt}\, b_{tk}$$

Do obliczenia jednego elementu macierzy C potrzeba *M* mnożeń oraz *M-1* dodawań. Dla wyznaczenia macierzy C wymagane jest łącznie *(2M-1)*N*K* działań zmiennoprzecinkowych.

MNOŻENIE MACIERZY

Algorytmy mnożenia macierzy są często definiowane dla zadania postaci $C = C + A*B$

Odpowiednio: $c_{nk} = c_{nk} + \sum_{t=1}^{M} a_{nt}\, b_{tk}$

dla $n=1 \ldots N$; $k=1 \ldots K$.

Do obliczenia jednego elementu macierzy C trzeba teraz M mnożeń i M dodawań, tj $2M$ operacji arytmetycznych. Obliczenie macierzy C wymaga $2MNK$ operacji.

Dla macierzy kwadratowych (M=K=N) złożoność algorytmu wynosi $2N^3$ operacji zmiennoprzecinkowych.

ROZWIĄZANIE IDEALNE

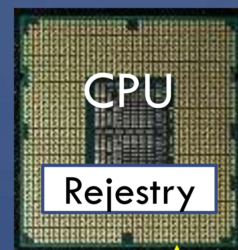
Ideał: obliczenie iloczynu macierzy $[N, N]$ na p procesorach w czasie równym $2N^3/p$, tj. proporcjonalne skrócenie czasu w stosunku do realizacji na jednym procesorze.

WYDAJNOŚĆ MNOŻENIA MACIERZY

Sposób dostępu do pamięci:

- Pamięć współdzielona
  - <u>Wąskie gardło</u>: pamięć buforowa (ang. *cache*) – konieczność przemieszczania danych pomiędzy poziomami hierarchii pamięci
- Pamięć rozproszona
  - <u>Wąskie gardło</u>: komunikacja pomiędzy procesami

HIERARCHIA PAMIĘCI



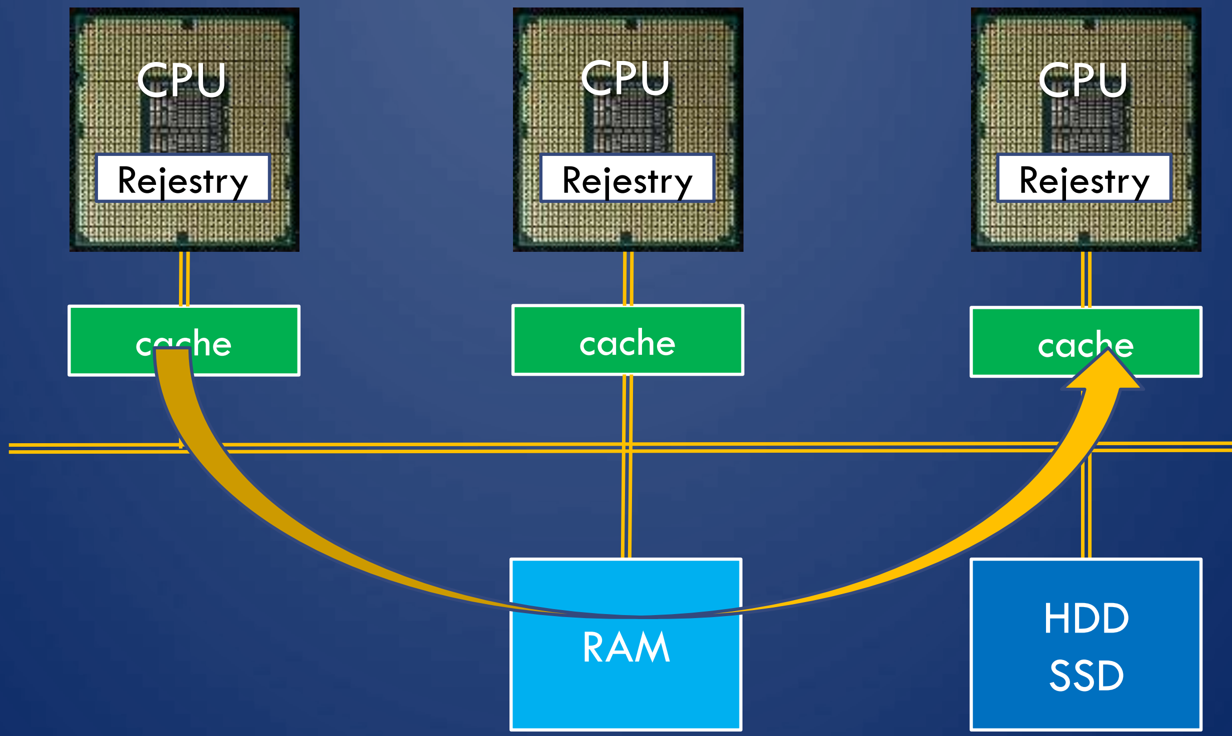
bardzo mało, bardzo szybkie, bardzo drogie)

Pamięć buforowa (*cache*)
(mała, szybka, droga)

Pamięć operacyjna (RAM)
(pojemna, wolna, tania)

Pamięć masowa (HDD, SSD)
(bardzo pojemna, bardzo wolna, bardzo tania)

ARCHITEKTURA SMP A HIERARCHIA PAMIĘCI



Fizyczna synchronizacja danych

WYKORZYSTANIE PAMIĘCI WSPÓŁDZIELONEJ

- Optymalne użycie CPU: obliczenia na rejestrach
- Podstawowy zasięg programu: 3 poziomy pamięci (rejestry, cache i RAM) - wirtualnie program działa tylko w rejestrach i RAM
- Czas przenoszenia danych między poziomami pamięci (zwłaszcza w przypadku skrajnych transferów) przekracza czas wykonania operacji na danych

Oznaczenie:

$$q = \frac{liczba\ operacji\ zmiennoprzecinkowych}{liczba\ odwołań\ do\ pamięci}$$

Dla danego algorytmu współczynnik q wskazuje na intensywność obliczeń względem operacji dostępu do pamięci – „jakość” algorytmu.

Większe q – algorytm bardziej przydatny do zastosowania.

IMPLEMENTACJA MNOŻENIA MACIERZY Z UŻYCIEM PAMIĘCI WSPÓŁDZIELONEJ

Założenia:

- Liczba poziomów hierarchii pamięci = 2
  (pamięć szybka – mała, pamięć wolna – duża)
- Pamięć szybka – pojemność E elementów macierzy, gdzie

$$N < E << N^2$$

- Pamięć wolna – wystarczająca dla umieszczenia elementów wszystkich macierzy
- Optymalna gospodarka danymi (minimalizacja czasu dostępu do danych ponownie wykorzystywanych)

IMPLEMENTACJA MNOŻENIA MACIERZY Z UŻYCIEM PAMIĘCI WSPÓŁDZIELONEJ (2)

Podstawowy algorytm (oznaczenia jak poprzednio)

```
for (i=1; i<=N; i++)
   for (k=1; k<=K; k++)
      for (t=1; t<=M; t++)
         c[i][k] = c[i][k] + a[i][t] * b[t][k];
```

Załóżmy, że:

1. Macierz A pobieramy z pamięci wierszami i przechowujemy je w szybkiej pamięci, dopóki są potrzebne (odczyt),
2. Elementy macierzy B pobieramy pojedynczo (odczyt),
3. Obliczane elementy macierzy C przechowujemy w szybkiej pamięci przez cały czas potrzebny do obliczenia wartości (odczyt i zapis)

IMPLEMENTACJA MNOŻENIA MACIERZY Z UŻYCIEM PAMIĘCI WSPÓŁDZIELONEJ (3)

Analiza złożoności algorytmu:

A. Odwołania do pamięci
   1. $N*N = N^2$ odczytów elementów macierzy A
   2. $N*N*N = N^3$ odczytów elementów macierzy B
   3. $(N+N)*N = 2*N^2$ dostępów do elementów macierzy C

   Łączna liczba odwołań do pamięci $A = N^3 + 3*N^2$.

   Dla dużych N, można przyjąć $A \approx N^3$.

B. Liczba operacji zmiennoprzecinkowych $F = 2*N^3$

Zatem $q = \frac{F}{A} \approx \frac{2*N^3}{N^3} \approx 2$.

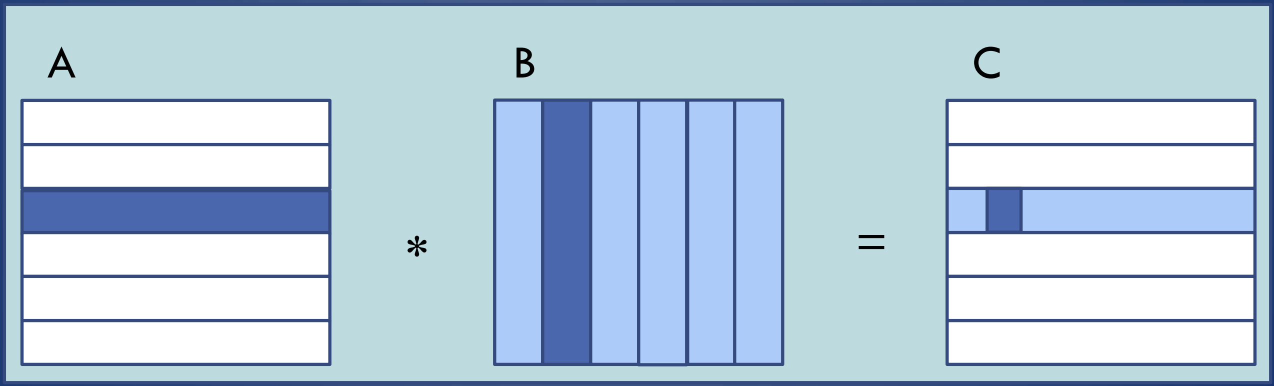
$q \approx 2$ nie jest dobrym wynikiem.

IMPLEMENTACJA MNOŻENIA MACIERZY Z UŻYCIEM PAMIĘCI WSPÓŁDZIELONEJ (4)

Jeżeli nie jest możliwe przechowanie całego wiersza macierzy A w pamięci szybkiej, to liczba dostępów do jej elementów zwiększa się wtedy do N^3.

Zatem $q = \frac{F}{A} \approx \frac{2 * N^3}{2 * N^3} \approx 1$.

IMPLEMENTACJA MNOŻENIA MACIERZY Z PODZIAŁEM NA BLOKI 1D



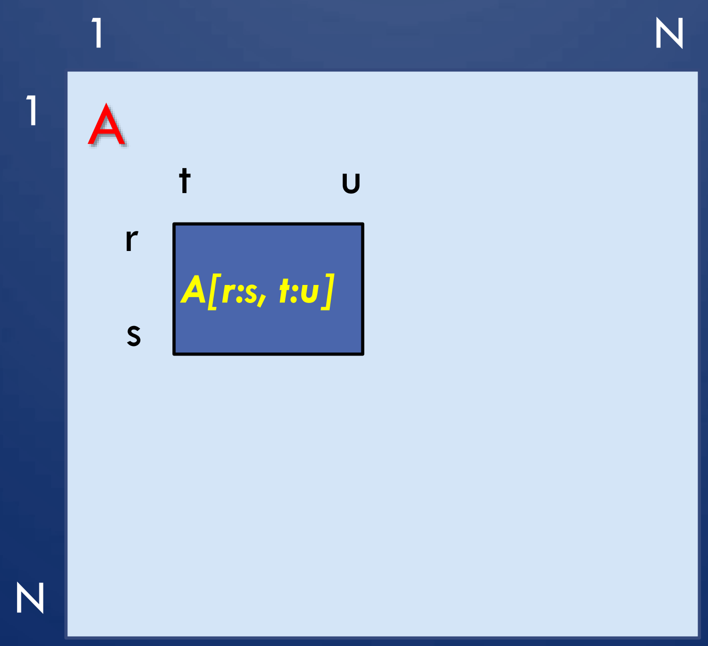
Bloki oznaczone kolorem granatowym biorą udział w tym samym podzadaniu obliczania elementów macierzy C.

Wyznaczenie całego bloku macierzy C wymaga dostępu do wszystkich kolumn bloku macierzy B.

IMPLEMENTACJA MNOŻENIA MACIERZY Z PODZIAŁEM NA BLOKI 1D (2)

Oznaczenie:

A[r:s, t:u] – podmacierz macierzy A zawierająca elementy należące do wierszy $r \ldots s$ i kolumn $t \ldots u$.



IMPLEMENTACJA MNOŻENIA MACIERZY Z PODZIAŁEM NA BLOKI 1D (3)

- W macierzy A wyodrębniamy wiersze;
  k-ty wiersz to $A[k{:}0, k{:}(n-1)]$;
- Macierz B dzielimy na bloki kolumn $B=[B^0, B^1,\ldots, B^{b-1}]$;
  liczba bloków wynosi b;
- Macierz C dzielimy na bloki wierszy $C=[C^0, C^1,\ldots, C^{b-1}]$

Algorytm przyjmuje postać:

```
for (j=0, j<b; j++)
   for (k=0; k<n; k++)
      Cj = Cj + A[k:0, k:(n-1)] * Bj;
```

OSZACOWANIE ZŁOŻONOŚCI

Szybka pamięć powinna pomieścić równocześnie:

- 1 wiersz macierzy A (n elementów)
- 1 blok kolumn macierzy B (n*n/b elementów)
- 1 blok wierszy macierzy C (n*n/b elementów)

Łączne zapotrzebowanie na pamięć szybką: $M \geq 2\frac{n^2}{b}+n$

Liczba kontaktów z pamięcią szybką:

- N-krotny odczyt macierzy A - $b*n^2$ dostępów
- Jednokrotne odczytanie każdego bloku B - $n*n/b*b=n^2$
- Jednokrotne odczytanie i zapisanie bloku C - $2n^2$

Łączna liczba dostępów do pamięci: $(b+3)n^2$.

OBLICZENIE WSPÓŁCZYNNIKA „JAKOŚCI”

Przypomnienie: $q = \frac{liczba\ operacji\ zmiennoprzecinkowych}{liczba\ odwołań\ do\ pamięci}$

Z poprzednich obliczeń mamy:

- Liczba operacji zmp = $2n^3$
- Liczba odwołań do pamięci: $(b+3)n^2$; b=liczba bloków

Po podstawieniu, mamy $q = \frac{2n^3}{(b+3)n^2}$.

Z zapotrzebowania na pamięć mamy

$$b \geq \frac{2n^2}{M-n} \approx \frac{2n^2}{M}, \text{ dla } M \gg n .$$

Zatem $q \approx \frac{M}{n}$, tj. M powinno rosnąć tak szybko, jak n lub szybciej.

INNE ALGORYTMY MNOŻENIA MACIERZY

- Pamięć współdzielona
  - Mnożenie macierzy z podziałem na bloki 2D
- Pamięć rozproszona
  - Algorytmy blokowe (różne wersje, różne topologie połączeń, w tym 2D i 3D)

SORTOWANIE BĄBELKOWE

ZADANIE SORTOWANIA

Dane: ciąg liczb $a_1, a_2, \ldots , a_n; n>1;$

Cel: zmiana kolejności elementów ciągu a_i tak, aby uzyskać ciąg $a_{i1}, a_{i2}, \ldots , a_{in}$, w którym

$a_{ij} \leq a_{i(j+1)}$ dla każdego j = 1, 2, …, n.

UWAGA: indeksy ij należy interpretować jako i_j .

IDEA SORTOWANIA BĄBELKOWEGO

Zadany ciąg liczb (docelowe uporządkowanie: niemalejące)

7	2	32	9	18	3	1	2	9	0	12	5

Krok 1: porównanie elementów i ewentualna zamiana miejscami

7	2	32	9	18	3	1	2	9	0	12	5

Wynik

2	7	32	9	18	3	1	2	9	0	12	5

Krok 2: porównanie elementów i ewentualna zamiana miejscami

2	7	32	9	18	3	1	2	9	0	12	5

Wynik (bez zmian)

itd. aż do ostatniego elementu

2	7	9	18	3	1	2	9	0	12	15	32

Koniec przejścia #1

2	7	9	18	3	1	2	9	0	12	15	32

Element na swoim miejscu

Następne przejścia dostarczają uporządkowania:

2 7 9 3 1 2 9 0 12 5 18 32

2 7 3 1 2 9 0 9 5 12 18 32

2 3 1 2 7 0 9 5 9 12 18 32

2 1 2 3 0 7 5 9 9 12 18 32

1 2 2 0 3 5 7 9 9 12 18 32

1 2 0 2 3 5 7 9 9 12 18 32

1 0 2 2 3 5 7 9 9 12 18 32

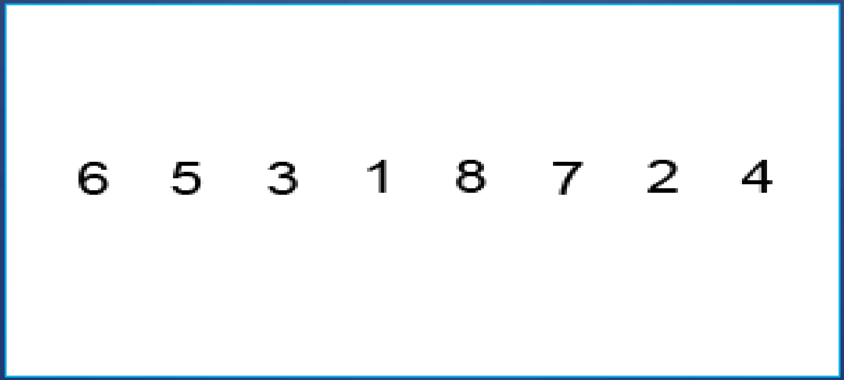
0 1 2 2 3 5 7 9 9 12 18 32

0 1 2 2 3 5 7 9 9 12 18 32

0 1 2 2 3 5 7 9 9 12 18 32

0 1 2 2 3 5 7 9 9 12 18 32 ← koniec procedury

Animacja przedstawiającą ideę sortowania bąbelkowego



https://pl.wikipedia.org/wiki/Plik:Bubble-sort-example-300px.gif

Taniec węgierski na temat sortowania bąbelkowego

https://www.youtube.com/watch?v=lyZQPjUT5B4

PODSTAWOWY ALGORYTM

```
for (i=0; i<n-1; i++)
{ for (j=0; j<n-i-1; j++)
    { if (array[j] > array[j+1])
        { swap = array[j];
          array[j] = array[j+1];
          array[j+1] = swap;
        }
    }
}
```

[na podstawie: http://www.programmingsimplified.com/c/source-code/c-program-bubble-sort]

W publikacjach – warianty różniące się kolejnością działań.

PODSTAWOWY ALGORYTM (2)

Liczba wykonywanych porównań:

- Przejście #1: n-1
- Przejście #2: n-2
- …
- Przejście #n: 0

Łącznie: $n(n-1)/2 \approx n^2/2$ porównań dla dużych n

ULEPSZENIE PODSTAWOWEGO ALGORYTMU

Dodatkowy wskaźnik informujący o tym, czy w czasie danego przejścia wystąpiła zamiana elementów.

- Przed rozpoczęciem przejścia – zerowanie wskaźnika.
- W razie wykonania zamiany elementów – ustawianie.
- Po zakończeniu przejścia – badanie wskaźnika: nieustawiony oznacza zakończenie procesu sortowania.

Efekt: eliminacja zbędnych przejść (skrócenie czasu sortowania) dla ciągów częściowo uporządkowanych.

ULEPSZENIE (2)

```
void bubblesort( int a[], int n )
{ int i, j, tmp, change;
  for (i=0; i<n-1; ++i)
  { change=0;
    for (j=0; j<n-1-i; j++)
    { if (a[j+1] < a[j]) //porównanie sąsiadów
      { tmp = a[j];
        a[j] = a[j+1];
        a[j+1] = tmp;     //wypchanie bąbelka
        change=1;
      }
    }
    if(!change)
      break; // nie dokonano zmian - koniec!
  }
}
```

[na podstawie: https://pl.wikibooks.org/wiki/Kody_źródłowe/Sortowanie_bąbelkowe]

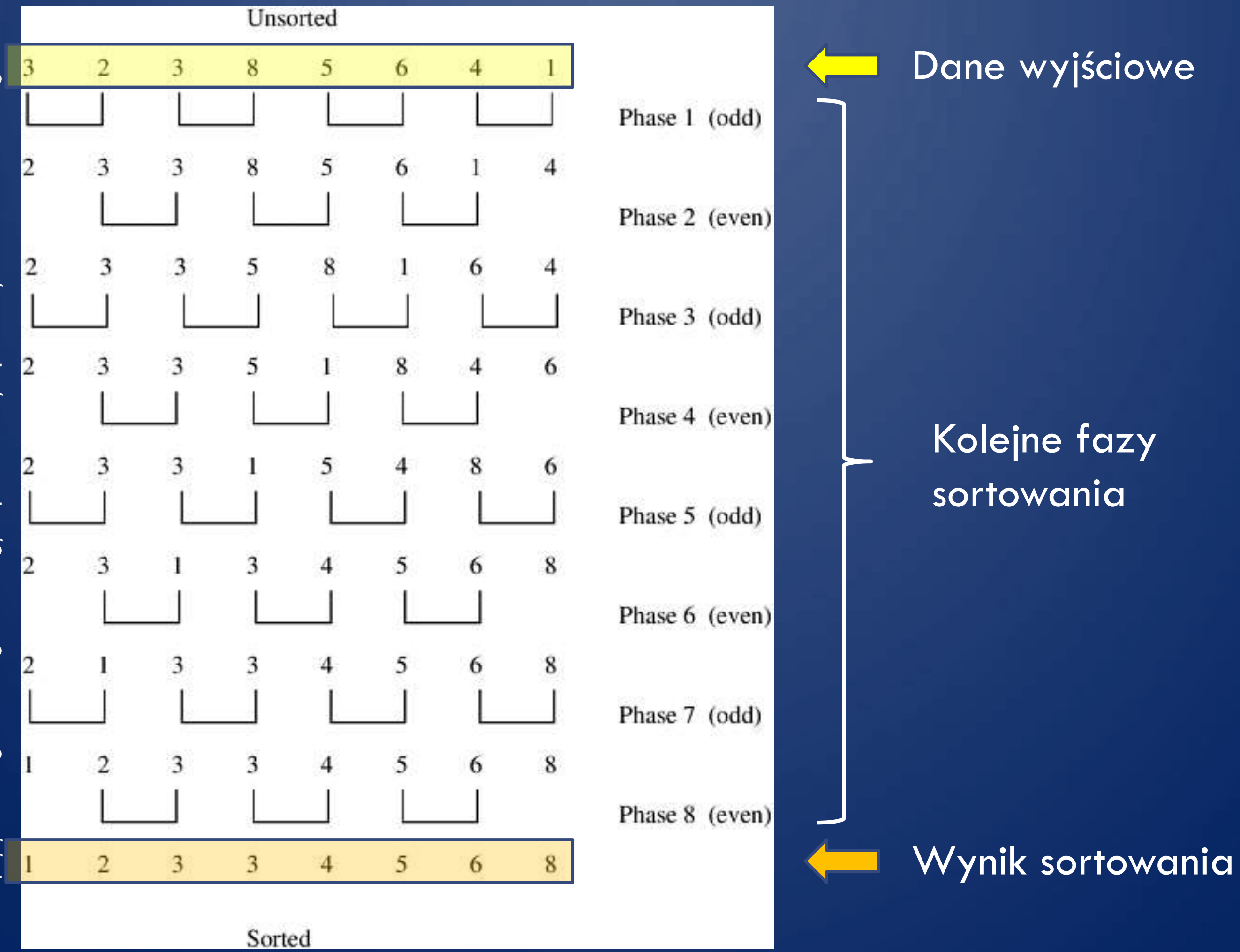
ODMIANA SORTOWANIA BĄBELKOWEGO: ODD-EVEN SORT

Modyfikacja polega na podziale na 2 fazy: nieparzystą (ang. *odd*) i parzystą (ang. *even*). Algorym kończy działanie gdy ciąg zostanie posortowany i w każdym przejściu wystąpi faza nieparzysta i parzysta.

```
{
  for (i=0; i < n; i--)
  {
    if nieparzyste(i)
      for (j=0; j <=(n/2-1); j++)
        porównaj_zamien(a2j+1, a2j+2);
    if parzyste(i)
      for (j=1; (j <= (n/2-1)); j++)
        porównaj_zamien(a2j, a2j+1);
  };
};
```

[Lucjan Stapp, *Programowanie równoległe i rozproszone, Wykład 12*, http://www.mini.pw.edu.pl/~lucjan/PRiR/wyklad12.pdf]

ODMIANA SORTOWANIA BĄBELKOWEGO: ODD-EVEN SORT (2)



http://www.geeksforgeeks.org/wp-content/uploads/Even-Odd-Sort.gif

ANALIZA ZŁOŻONOŚCI ALGORYTMU
Odd-Even Sort

1 krok → $\frac{N}{2}$ lub $\frac{N}{2} - 1$ porównań

N kroków → rząd $N * \frac{N}{2} \approx \frac{N^2}{2}$ porównań, dla dużych N

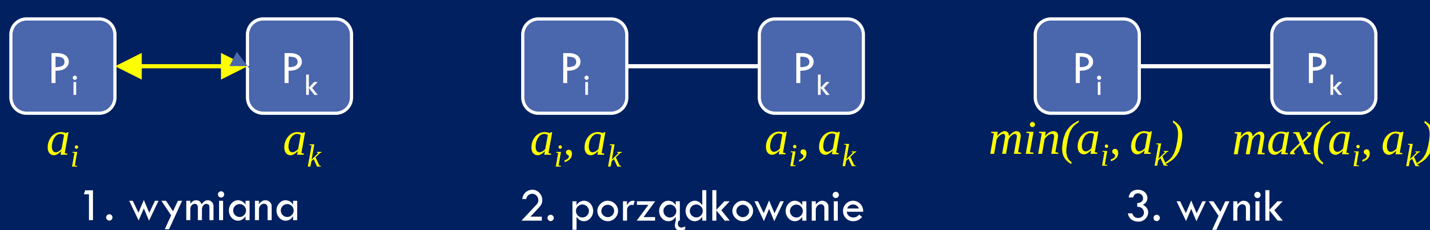
Złożoność analogiczna jak dla wersji podstawowej.

ALGORYTM ODD-EVEN SORT W WERSJI RÓWNOLEGŁEJ

Założenia:

- Sortowanie odbywa się z wykorzystaniem p procesorów,
- Każdy procesor przechowuje 1 element ciągu,
- Każdy procesor komunikuje się jedynie ze swoimi sąsiadami (lewym lub prawym), o ile istnieją.

- Przebieg operacji *porównanie_wymiana*:



- **Efekt wykonania operacji**: przechowywane wartości zachowują porządek numeracji procesorów, tj. dla $i<k$ jest $a_i <= a_k$

RÓWNOLEGŁY BLOKOWY ALGORYTM ODD-EVEN SORT

Założenia:

- Sortowanie odbywa się z wykorzystaniem p procesorów,
- Każdy procesor przechowuje blok N/p uporządkowanych elementów podciągu (N podzielne przez p)
- Każdy procesor komunikuje się jedynie ze swoimi sąsiadami (lewym lub prawym), o ile istnieją.
- Wykonuje się operacje *porównanie_sklejanie* analogiczne jak w przypadku przechowywania pojedynczego elementu ciągu
    1. Wymiana prowadzi do chwilowego podwojenia liczby elementów podciągu w procesorach,
    2. Porządkowanie ustala dwa identyczne podciągi w danej parze procesorów
    3. Wynik: procesor o niższym numerze zachowuje N/p początkowych elementów podciągu, a procesor o wyższym – pozostałe N/p elementów.
- Efekt wykonania operacji: przechowywane wartości zachowują porządek numeracji swoich procesorów.

RÓWNOLEGŁY BLOKOWY ALGORYTM ODD-EVEN SORT (2)

Wymiana	P_i ⟷	P_{i+1}
	[2, 5, 7, 8]	[1, 4, 6, 9]
Porządkowanie	P_i ⟷	P_{i+1}
	[2, 5, 7, 8] [1, 4, 6, 9]	[1, 4, 6, 9] [2, 5, 7, 8]
Wynik	P_i ⟷	P_{i+1}
	[1, 2, 4, 5]	[6, 7, 8, 9]

ŹRÓDŁA

- Jarosław Pytliński, *Mnożenie macierzy – algorytmy równoległe,* http://www-users.mat.uni.torun.pl/~bala/sem_mgr_2000/matrix1.html
- Lucjan Stapp, *Programowanie równoległe i rozproszone, Wykład 9*, http://www.mini.pw.edu.pl/~lucjan/PRiR/wyklad9.pdf
- Lucjan Stapp, *Programowanie równoległe i rozproszone, Wykład 12*, http://www.mini.pw.edu.pl/~lucjan/PRiR/wyklad12.pdf
- *Odd-Even Sort/Brick Sort*, http://www.geeksforgeeks.org/odd-even-sort-brick-sort/
- *Kody źródłowe/Sortowanie bąbelkowe,* https://pl.wikibooks.org/wiki/Kody_źródłowe/Sortowanie_bąbelkowe