Analiza efektywności mnożenia macierzy w systemach z pamięcią współdzieloną

MATERIAŁY POMOCNICZE DO LABORATORIUM Z PRZETWARZANIA RÓWNOLEGŁEGO LISTOPAD 2016

Mnożenie macierzy – dostęp do pamięci podręcznej [język C, kolejność - j,i,k][1]

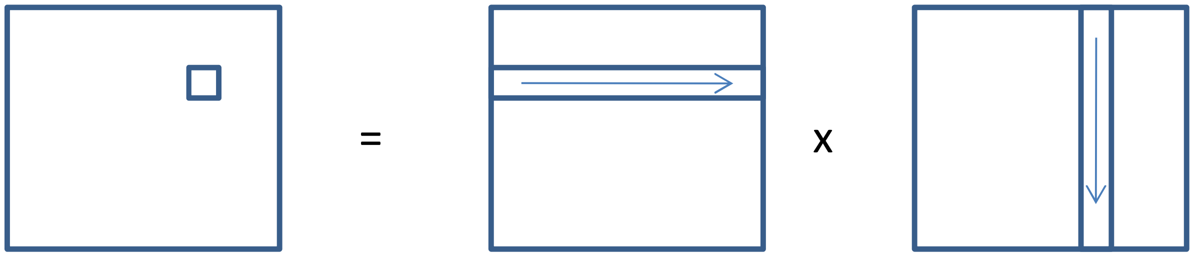
A,B,C są tablicami nxn

```
for (int j = 0 ; j < n ; j++)
     for ( int i = 0 ; i < n ; i++)
            for (int k = 0 ; k < n ; k++)
                    C[i][j] + = A[i][k] * B[k][j] ;
```

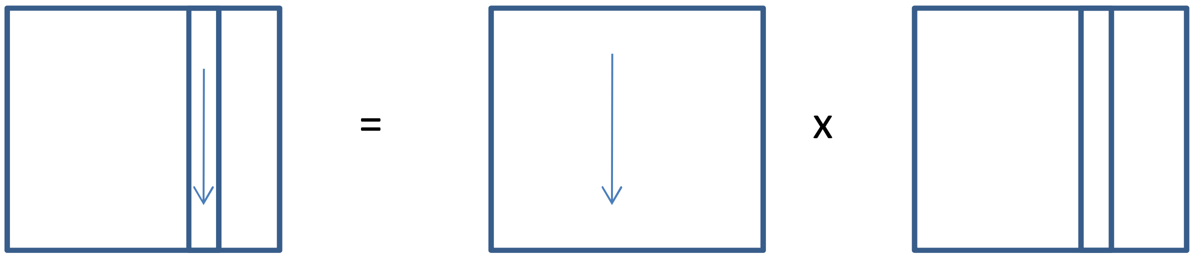
A[i][*] lokalność przestrzenna danych – rózne elementy z linii pp wykorzystane w kolejnych iteracjach

B[*][j] możliwy brak wielokrotnego odczytu raz pobranej linii

C[i][j] lokalność odwołań – ten sam element dla każdej iteracji pętli wewnętrznej



Ilość danych wykorzystywanych w pętli wewnetrznej

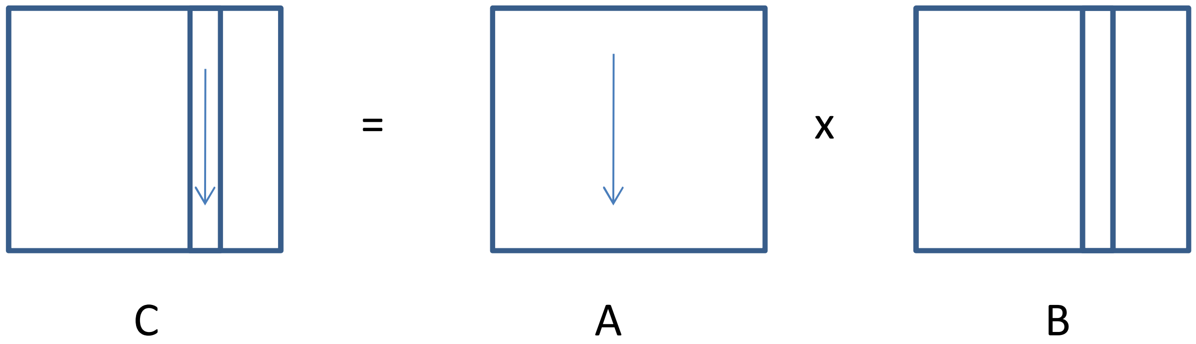


C A B

Ilość danych wykorzystywanych w 2 pętlach wewnętrznych

A parallel ?

Mnożenie macierzy – pamięć podręczna [C, j,i,k][2]



C[*][j] brak lokalności przestrzennej odwołań

A[*][*] warunek na lokalność czasową odwołań

B[*][j] brak lokalności przestrzennej,

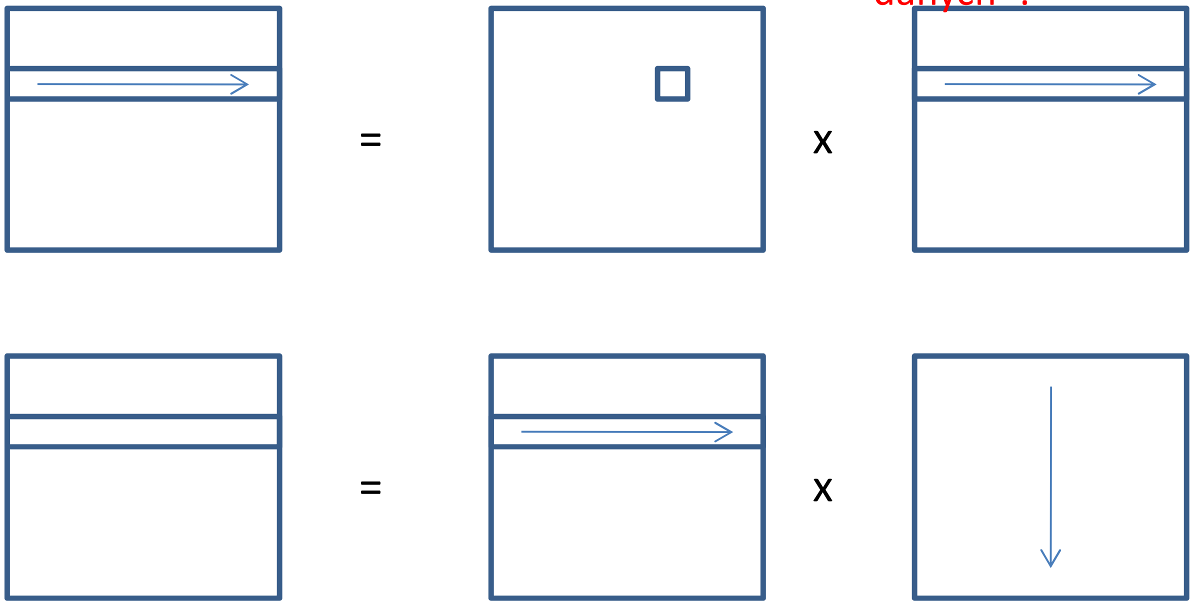
Warunek na lokalność czasową odwołańdo danych – pamięć podręczna mieści tablicę B i linie pp z kolumnami tablicy C.

Mnożenie macierzy – pamięć podręczna[C, i,k,j][1]

A,B,C są tablicami nxn

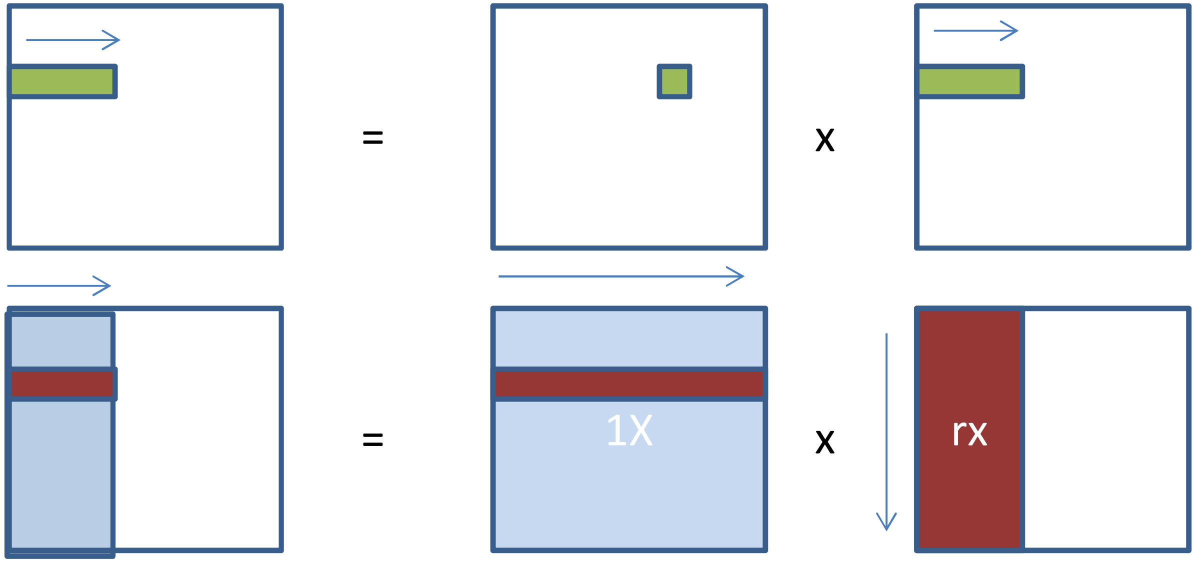
```
for ( int i = 0 ; i < n ; i++)
    for (int k = 0 ; k < n ; k++)
        for (int j = 0 ; j < n ; j++)
            C[i][j] + = A[i][k] * B[k][j] ;
```

A[i][k] – lokalność czasowa odwołań
B[k][*], C[i][*] – lokalność przestrzenna odwołań
A[i][*] – lokalność przestrzenna odwołań
B[*][*] – brak lokalności czasowej odwołań jeżeli suma rozmiaru(wiersz A, wiersz C i tablica B) większe od rozmiaru pamięci podręcznej
Warunek na czasową lokalność dostępów do danych ?



Najbardziej zagnieżdżona pętla j zapewnia lokalność przestrzenną dostępu do danych, odpowiednia wielkość pamięci podręcznej zapewnia lokalność czasową dostępu do danych.

Mnożenie macierzy – pamięć podręczna
[C, i,k,j*] zmniejszenie zakresu pętli wewnętrznej

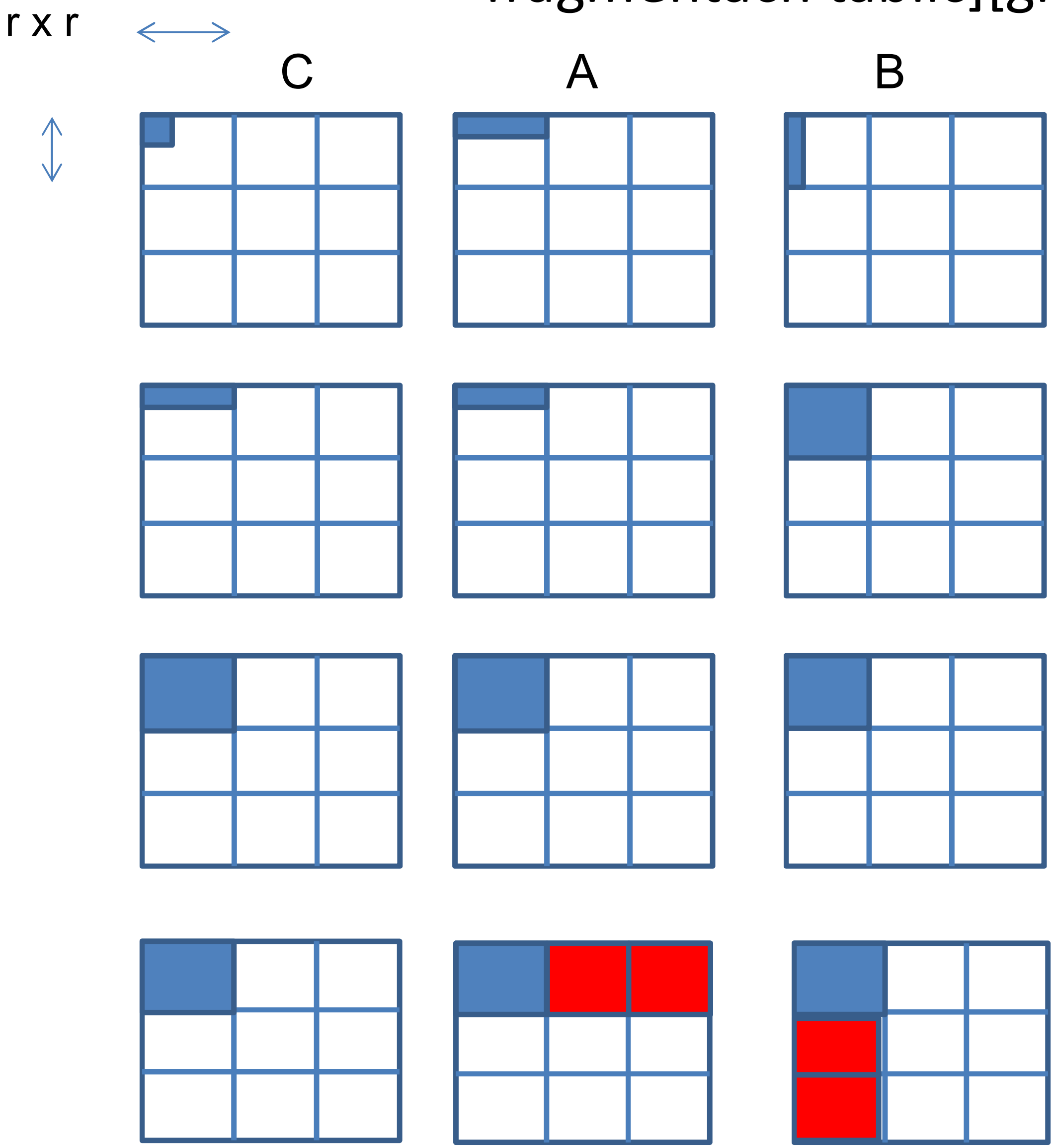


Obliczamy wpierw fragmenty wiersza macierzy wynikowej nie cały wiersz

```
for ( int j = 0 ; j < n ; j+=r) // cała macierz wynikowa
   for ( int i = 0 ; i < n ; i++)  // wyznaczenie niebieskiej części wyniku
         for (int k = 0 ; k < n ; k++)  // wyznaczenie brązowej części wyniku
            for (int jj = j ; jj < j+r-1 ; jj++)
                  C[i][jj] + = A[i][k] * B[k][jj] ;
```

Przy odpowiedniej wielkości r możliwa lokalność czasowa odwołań do B[*][jj:jj+r-1]
Zmniejszenie wielkości fragmentów tablic, na podstawie których realizowane są obliczenia (w jednej fazie przetwarzania) prowadzi do większej lokalności odwołań. Konieczne ponowne pobrania macierzy A – ile razy ?
(a efektywność translacji adresów ? – rozwiązanie na kolejnym slajdzie).

Mnożenie macierzy – pamięć podręczna [operacje na fragmentach tablic][graficznie]




```
for (int kk = k ; kk < k+r ; kk++)
     C[ii][jj] + = A[ii][kk] * B[kk][jj];
```

```
for (int jj = j ; jj < j+r ; jj++)
     for (int kk = k ; kk < k+r ; kk++)
            C[ii][jj] + = A[ii][kk] * B[kk][jj];
```

```
for ( int ii = i ; ii < i+r; ii++)
   for (int jj = j ; jj < j+r ; jj++)
      for (int kk = k ; kk < k+r ; kk++)
            C[ii][jj] + = A[ii][kk] * B[kk][jj];
```

```
for (int k = 0 ; k < n ; k+=r)
   for ( int ii = i ; ii < i+r; ii++)
      for (int jj = j ; jj < j+r ; jj++)
         for (int kk = k ; kk < k+r ; kk++)
            C[ii][jj] + = A[ii][kk] * B[kk][jj];
```

Mnożenie macierzy – pamięć podręczna [operacje na fragmentach tablic][kod]

```
#pragma omp parallel for
for ( int i = 0 ; i < n ; i+=r)
   for ( int j = 0 ; j < n ; j+=r)
       for (int k = 0 ; k < n ; k+=r) // kolejne fragmenty  we
        for ( int ii = i ; ii < i+r; ii++)//fragment wyniku
              for (int jj = j ; jj < j+r ; jj++)
                     for (int kk = k ; kk < k+r ; kk++)
                            C[ii][jj] + = A[ii][kk] * B[kk][jj];
```

Dla C[ii][jj], A[ii][kk], B[kk][jj] lokalność czasowa i przestrzenna dostępów przy założeniu, że wszystkie podmacierze A,B i C (A[i:i+r-1][k:k+r-1],B[k:k+r-1][j:j+r-1], C[i:i+r-1][j:j+r-1]) mieszczą się w pamięci podręcznej

Zakładając, że rozmiar pp równy M można wyznaczyć wymagane $r <= (M/3)^{1/2}$ (dla przetwarzania sekwencyjnego – 1 procesor – 3 tablice)

1. 3 pętle wewnętrzne służą do wyznaczenia wyniku częściowego dla fragmentu tablicy wynikowej (sum iloczynów elementów wierszy i kolumn fragmentów macierzy wejściowych),
2. czwarta pętla (po k) służy do uzupełnienia wyniku o pozostałe iloczyny wynikające z uwzględnienia kolejnych (branych po r) elementów wierszy i kolumn fragmentów macierzy wejściowych,
3. pętle piąta i szósta służą do wyznaczenia kolejnych kwadratowych (r) obszarów macierzy wynikowej.

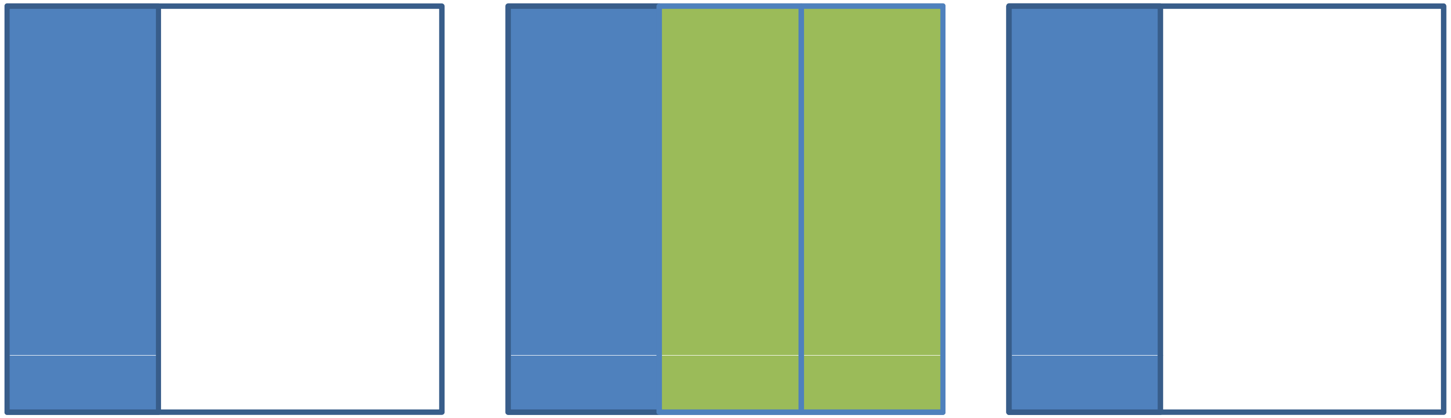
Mnożenie macierzy w systemach z pamięcią rozproszoną

WYKŁAD Z PRZETWARZANIA RÓWNOLEGŁEGO LISTOPAD 2016

Kolumnowa dystrybucja tablic

- A,B, C sa macierzami nxn
- P – liczba procesorów
- A,B,C są rozdystrybuowane w pamięciach procesorów w sposób blokowy (całymi kolumnami)
- Obliczenia C[i][j] + = A[i][k] * B[k][j] ; są realizowane zgodnie z zasadą właściciel wyniku przetwarza.
- Zgodnie z zasadą 3 pętlowego algorytmu mnożenia tablic (kolejnośc ijk - znany kod) kolumny tablic C i B sa przetwarzane lokalnie (tylko w różnych procesorach). Ze względu na to, że A jest używane w wszystkich procesorach konieczne jest rozesłanie (all-to-all broadcast) tablicy A.

Kolumnowa dystrybucja tablic



- Kolejno narysowane tablice C,A,B – C=A*B
- 3 komputery uczestniczą w przetwarzaniu.
- Dystrybucja kolumnowa, kolor niebieski – dane dla procesora 1
- Kolor zielony – wartości procesora 1 otrzymane w wyniku rozgłaszania od pozostałych 2 procesorów.

Mnożenie macierzy – systemy wieloprocesorowe z pamięcią rozproszoną (dystrybucja kolumnowa)[2]

```
//P procesorów – każdy posiada podzbiór kolumn tablic
float a[n][n/P], b[n][n/P], c[n][n/P],tmp[n][n];
……
 for (int proc = 0 ; j < P ; j++) //rozsyłanie do wszystkich odbiorców
      a_send(proc, a)
for (int proc = 0 ; j < P ; j++)
      a_recv(proc, tmp[1:n][proc*n/P: proc*(n/P+1)-1]; // co oznacza ten zapis ?
// oczekiwanie na zakończenie asynchronicznej komunikacji (dlaczego asynchroniczna ?)
for ( int i = 0 ; i < n ; i++)
        for (int k = 0 ; k < n ; k++)
                for (int j = 0 ; j < n/P ; j++)
                        c[i][j] + = tmp[i][k] * b[k][j] ;
```

Każdy procesor przetwarza $(1+2/P)n^2$ elementów, każdy procesor rozsyła n^2/P elementów do P procesorów.

a_send, a_recv oznaczają operacje asynchroniczne odpowiednio nadawania i odbioru.

Mnożenie macierzy (dystrybucja szachownicowa)

1	2	3
4	5	6
7	8	9

Struktura systemu i przydzielone części tablicy wyniku UWAGA: właściciel oblicza

1	2	3
5	6	4
9	7	8

Dystrybucja wśród procesorów początkowej tablicy A w 9 procesorach (liczby oznaczają część tablicy) – podtablice danych A w kolejnych wierszach tablicy procesorów przesunięte o 1 w stosunku do dystrybucji wyniku

1	5	9
4	8	3
7	2	6

Dystrybucja początkowa tablicy B (liczby oznaczają część tablicy) – podtablice danych B w kolejnych kolumnach tablicy procesorów przesunięte o 1 w stosunku do dystrybucji wyniku.

A,B,C sa macierzami nxn
P = 9 – liczba procesorów
A,B są rozdystrybuowane w pamięciach procesorów w układzie szachownicy (z przesunięciem - rysunek)
Obliczenia C[i][j] + = A[i][k] * B[k][j] są realizowane zgodnie z zasadą właściciel wyniku przetwarza.
Dla wyznaczenia wyniku każdy procesor w kolejnym kroku musi otrzymać:

- od prawego sąsiada (cyklicznie) wartości jego aktualnej części tablicy A
- od dolnego sąsiada (cyklicznie) wartości jego aktualnej części tablicy B

Zajętość pamięci:

- na procesor na poziomie $3n^2/P$, sumarycznie jest równa zajętości dla przetwarzania sekwencyjnego,
- Każdy procesor przesyła n^2/P elementów do 2(sqrt(P)-1) procesorów

Mnożenie macierzy
(algorytm Cannona – minimalizcja zajętości pamięci)

A00 B00	A01 B11	A02 B22
A11 B10	A12 B21	A10 B02
A22 B20	A20 B01	A21 B12

A01 B10	A02 B21	A00 B02
A12 B20	A10 B01	A11 B12
A20 B00	A21 B11	A22 B22

A02 B20	A00 B01	A01 B12
A10 B00	A11 B11	A12 B22
A21 B10	A22 B21	A20 B02

Wymiana części tablic wejściowych odbywa się synchronicznie (cyklicznie, kolumnami dla B i wierszami dla A) z obliczeniami wykorzystującymi te części tablic, każda tablica jest przechowywana w jednej kopii . Symbole A?? B?? oznaczają podtablice, których odpowiednie wiersze i kolumny pomnożone przez siebie stanowią wynik częściowy generowany w danym etapie przetwarzania

Mnożenie macierzy
(algorytm Cannona – minimalizcja zajętości pamięci)

c=axb – a,b,c są macierzami nxn

c jest dystrybuowane w bloki szachownicowo,

a i b są dystrybuowane w bloki szachownicowo (z odpowiednim przesunięciem)

Obliczenia odbywają się z cykliczną rotacją w kolejnych krokach

algorytm dla P^2 procesorów komunikacja w strukturze tablicy proc[p,p]

Komunikacja asynchroniczna z buforowaniem

```
float a[n/P][n/P], b[n/P][n/P], c[n/P][n/P];
……
for ( int kk = 0 ; i < P ; i++)
    {
    for ( int i = 0 ; i < n/P ; i++)
            for (int j = 0 ; j < n/P ; j++)
                    for (int k = 0 ; k < n/P ; k++)
                            c[i][j] + = a[i][k] * b[k][j] ;
    send(proc[pi,pj-1], a);
    send(proc[pi-1,pj], b);
    recv(proc[pi,pj+1], a);
    recv(proc[pi+1,pj], b);
    }
```

Mnożenie macierzy – algorytm Cannona
koszt przetwarzania

- Koszt komunikacji:
  – każdy z procesorów przesyła do sąsiada
    - w wierszu macierzy procesorów - tablicę **a** o rozmiarze $n/\sqrt{p}$
    - w kolumnie macierzy procesorów - tablicę **b** o rozmiarze $n/\sqrt{p}$
  – koszt wysyłania n^2/p elementów jednej tablicy realizowany $(\sqrt{p}-1)$ razy w kolejnych etapach to
    - $(t_s + t_w n^2/p)(\sqrt{p}-1)$
- Koszt obliczeń na jednym komputerze (realizowane równolegle na p komputerach) :
  – mnożenie tablic o rozmiarze $n/\sqrt{p}$ - liczba operacji mnożenia i dodawania to $(n/\sqrt{p})^3$
  – mnożenie tablic jest powtarzane $\sqrt{p}$ razy co daje n^3/p operacji dodawania i mnożenia
  – Czas obliczeń $n^3/p *(t_d+t_m)$
- Całkowity koszt obliczeń (złożoność) wynosi $n^3/p *(t_d+t_m) + 2(t_s + t_w n^2/p)(\sqrt{p}-1)$
  – 2 powyżej oznacza wynika z zastosowania sekwencyjnego (synchronicznego) przesyłania obu tablic **a i b** w kazdym z etapów

```
row = my_rank / 5; col = my_rank % 5; //25 procesorów
for (int i = 0; i < n / PP; i++)
for (int j = 0; j < n / PP; j++)
{
a[i][j] = float(my_rank); b[i][j] = float(my_rank); c[i][j] = 0;
}

pra = aa; prb = bb; psa = a; psb = b;


for (int kk = 0; kk < PP; kk++)
{
		for (int i = 0; i < n / PP; i++)
		for (int j = 0; j < n / PP; j++)
		for (int k = 0; k < n / PP; k++)
				c[i][j] += psa[i][k] * psb[k][j];
MPI_Irecv(pra, n*n / PP / PP, MPI_FLOAT, (5 * row + (5 + col - 1) % 5), tag, MPI_COMM_WORLD, reqRecv);
MPI_Irecv(prb, n*n / PP / PP, MPI_FLOAT, (5 * ((row + 1) % 5) + col), tag, MPI_COMM_WORLD, &reqRecv[1]);
MPI_Isend(psa, n*n / PP / PP, MPI_FLOAT, (5 * row + (col + 1) % 5), tag, MPI_COMM_WORLD, reqSend);
MPI_Isend(psb, n*n / PP / PP, MPI_FLOAT, (5 * ((5 + row - 1) % 5) + col), tag, MPI_COMM_WORLD, &reqSend[1]);
//MPI_Wait(reqRecv, statRecv);
//MPI_Wait(&reqRecv[1], &statRecv[1]);

MPI_Barrier(MPI_COMM_WORLD);
if (mod = ((mod + 1) % 2)) { pra = a; prb = b; psa = aa; psb = bb; }

else {

pra = aa; prb = bb; psa = a; psb = b;
}

}
```