Analiza efektywności mnożenia macierzy w systemach z pamięcią współdzieloną

MATERIAŁY POMOCNICZE DO LABORATORIUM Z PRZETWARZANIA RÓWNOLEGŁEGO KWIECIEŃ 2018

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

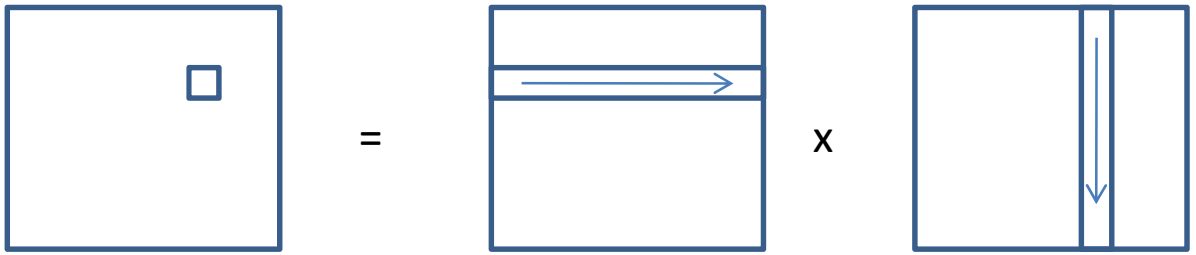
A,B,C są tablicami nxn

```
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] ;
```

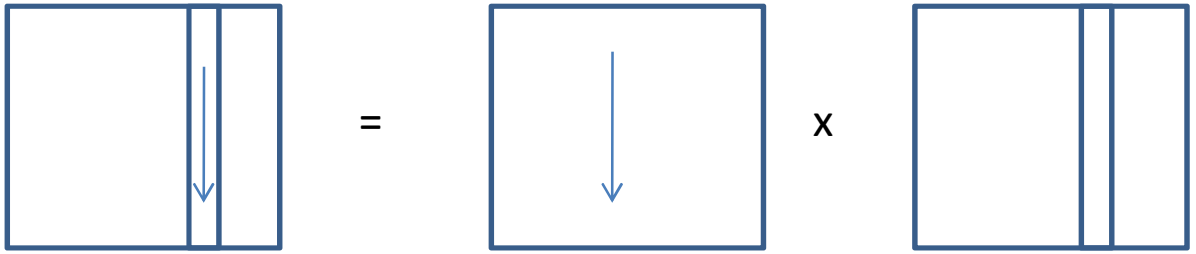
A[i][*] lokalność przestrzenna danych – rózne elementy z linii pp wykorzystane w kolejnych iteracjach

B[*][j] możliwy brak wielokrotnego odczytu raz pobranej linii (kiedy?)

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



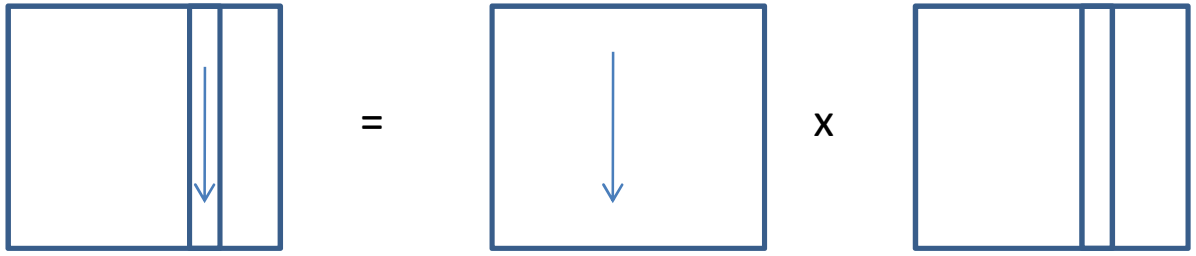
Ilość danych wykorzystywanych w pętli wewnetrznej



Ilość danych wykorzystywanych w 2 pętlach wewnętrznych A współbieżnie ?

C A B

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 odwołań do danych – pamięć podręczna mieści tablicę A i linie pp z kolumnami tablicy C i B.

Zarządzanie pamięcią
Pamięć wirtualna

Pamięć wirtualna

- przydzielana w blokach o wielkości „strony pamięci wirtualnej”,
- udostępniana procesom po zapisaniu strony do obszaru pamięci operacyjnej - rzeczywistej – do tzw. „ramki pamięci”,
- umożliwia przydział procesom większej ilości pamięci niż jest dostępna fizycznie w systemie,
- aby proces mógł pobrać dane konieczne jest odwzorowanie adresu wirtualnego na aktualny adres fizyczny, pod którym dane aktualnie się znajdują,
- konieczna jest zatem translacja adresów wirtualnych na adresy fizyczne.

Zarządzanie pamięcią
Bufor translacji adresu (TLB)

- Każdy dostęp procesora do pamięci powoduje konieczność określenia fizycznego adresu pod którym znajduje się wartość spod określonego w kodzie adresu wirtualnego.
- Odwzorowanie (translacja) jest realizowane przez bufor translacji adresu TLB (ang. translation lookaside buffer), który zawiera adresy fizyczne ostatnio translowanych adresów wirtualnych.
- Pamięć TLB może posiadać strukturę wielopoziomową i może być oddzielna dla danych i kodu. W przypadku braku adresu wirtualnego w TLB system operacyjny korzystając z katalogu i tablic stron określa brakujący adres i wpisuje go do TBL wykonując procedurę obsługi przerwania (znaczy koszt czasowy).

Zarządzanie pamięcią II

Brak informacji w TLB jest nazywany brakiem trafienia do TLB. Niski stosunek trafień do TLB jest spowodowany niską przestrzenną lokalnością kodu. Np. w programie (język C) odczyt różnych elementów kolumny tablicy z długimi wierszami.

Gdy znany jest adres fizyczny operandu wtedy można określić:

- czy pobierana z pp L1 (współbieżność działań) wartość jest poprawna,
- skąd należy wartość pobrać, czy jest w strukturze pp czy trzeba pobrać z pamięci operacyjnej .

W przypadku braku linii z żądanymi danymi w pp poziomu1 (cache L1) brakująca linia (cache miss) jest pobierana z pamięci niższego poziomu pamięci podręcznej (L2, L3) lub pamięci operacyjnej.

W przypadku braku strony z żądanymi danymi w pamięci operacyjnej (page fault) żądana strona odczytywana jest z dysku.

Zależności PP DTBL

Komputer laboratoryjny

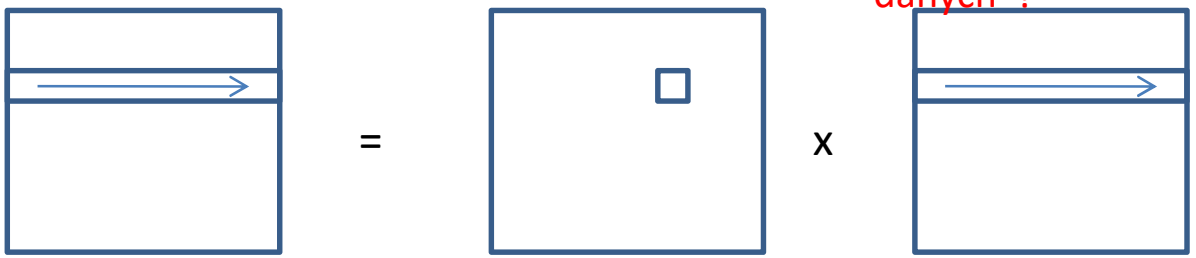
- DTLB ok. 500 par wpisów adresów
- PP L3 pomieści dane 500 stron wirtualnych o wielkości 4 kB – 2MB
- 4 rdzenie z własnymi DTLB
- Niektóre dane, których adres jest znany mogą nie być w PP L3 - 8MB > 6MB pp L3
- Nietrafienie DTLB, trafienie pp
- Trafienie DTLB, nietrafienie pp

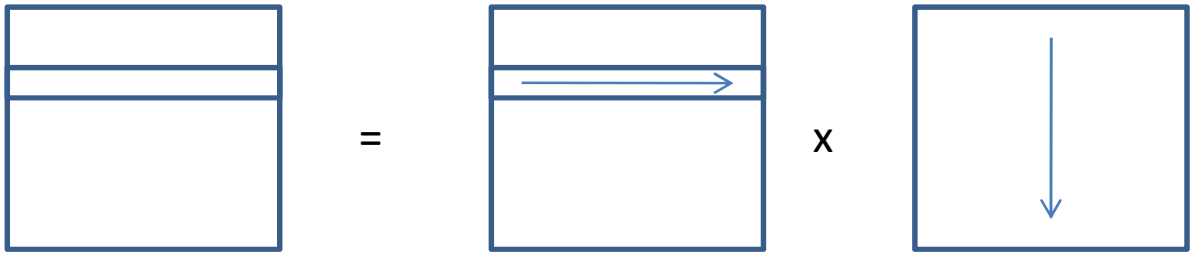
Mnożenie macierzy – pamięć podręczna[C, i,k,j][1]

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 ?

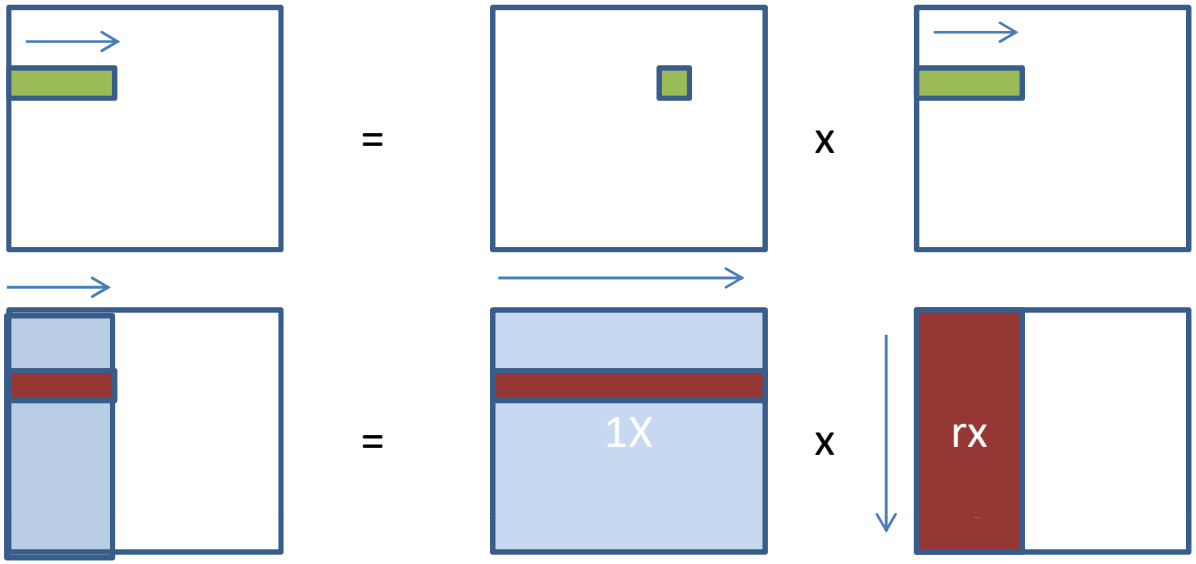




Najbardziej zagnieżdżona pętla j zapewnia lokalność przestrzenną dostępu do danych, odpowiednia wielkość pamięci podręcznej zapewnia lokalność czasową dostępu do danych.

Mnożenie macierzy – pamięć podręczna

[C, i,k,j*] zmniejszenie zakresu pętli wewnętrznej

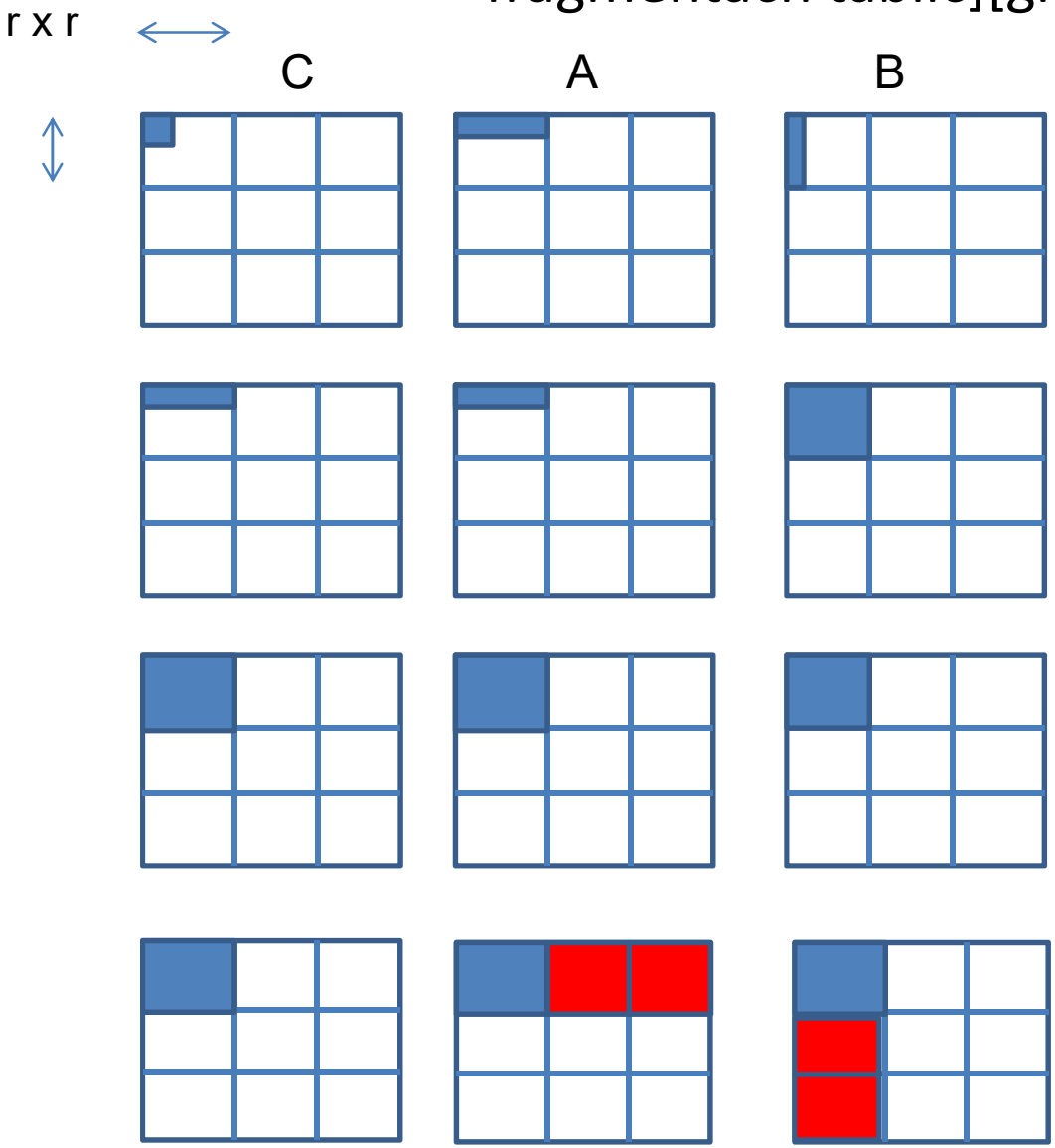


Obliczamy wpierw fragmenty wiersza macierzy wynikowej nie cały wiersz

```
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 ?

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]

```
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
       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 dostępów do danych 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 .

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.

Metoda 6 pętlowa
analiza lokalności czasowej

W ramach 4 wewnętrznych pętli:

- korzysta się z wielu tablic o wielkości r*r. Jedna tablica (część wyniku) jest potrzebna na każdym z etapów, tablice wejściowe (części tablicy A i B) jednocześnie są potrzebne w 2 egzemplarzach po jednej dla A i B w kolejnych etapach nowe części po r*r elementów.
- Jednocześnie potrzebne są dla każdego z wątków przetwarzania 3 tablice o rozmiarze r*r. W pamięci podręcznej o rozmiarze M używanej przez jeden wątek zmieszczą się 3 tablice o rozmiarze
- $r <= (M/ts/3)^{1/2}$
- (ts rozmiar typu zmiennej, 3 - liczba tablic używanych przez jeden wątek).

- Dla przyjętej kolejności zagnieżdżenia tablica B jest czytana wielokrotnie i musi być podobnie jak tablica C cały czas dostępna, tablica A jest czytana tylko raz (wierszami) i faktycznie mogłaby (w aktualnie potrzebnym zakresie – bez potrzeby ponownego pobierania do pamięci podręcznej) zajmować jedną linię pamięci podręcznej przy efektywnym zarządzaniu pamięcią. Jednakże dla zapewnienia ciągłej obecności w pamięci wielokrotnie używanych fragmentów tablic C i B bezpieczniej również dla tablicy A zarezerwować obszar w pamięci podręcznej równy r*r (wg wzoru powyżej).
- Iteracje zewnętrznych pętli to realizacja obliczeń powyższego typu dla innych wyników w oparciu o te same lub inne dane (tablice A i B). W całości przetwarzania każdy blok tablic A,B,C o rozmiarze r*r jest używany wielokrotnie w n/r etapach i tyle razy pobierany do pamięci podręcznej w najgorszym razie przy spełnieniu powyższej zależności na r.

Metoda 6 pętlowa równoległa - analiza lokalności czasowej

W sytuacji, gdy przetwarzanie jest **realizowane równolegle** w zależności od sposobu podziału pracy mamy różne zależności:

- Wersja A - podział pracy przed pierwszą pętlą
  - Każdy z wątków wykonuje prace na podzbiorze bloków położonych obok siebie w pasach – liczba zadań do podziału wynosi N/r i powinna być dobrana dla zapewnienia zrównoważenia pracy systemu (drugi warunek na r (dla równoległości) przy zmieniającym się warunku pierwszym ze względu na liczbę wątków korzystających z tej samej pamięci podręcznej – potencjalnie wzrasta liczba równocześnie używanych tablic)
- Wersja B - podział pracy przed pętlą czwartą
  - Każdy z wątków dzieli pracę w ramach wyliczania sum częściowych każdego wynikowego bloku c [r,r], każdy wątek liczy inną cześć wyników w oparciu o wszystkie dane wejściowe niezbędne dla tego celu. Liczba zadań do podziału wynosi r. Podział pracy wprowadza synchronizację wątków.
- Wersja C - podział pracy przed pętlą trzecią
  - może powodować wyścig w dostępie do danych – możliwe jednoczesne uaktualnienia tych samych elementów zmiennych doprowadzają do błędnych wyników ze względu na brak atomowości uaktualnienia zmiennej. Zapewnienie atomowości uaktualnienia wprowadza dalszą synchronizację wątków.

Metoda 6 pętlowa równoległa - analiza lokalności przestrzennej

Lokalność przestrzenna dostępu do danych wynika:

- z kolejności najbardziej wewnętrznych pętli programu,
- odległości w pamięci elementów tablicy położonych w sąsiednich wierszach (wielkość wiersza),
- liczby wierszy przetwarzanych na danym etapie obliczeń (wielokrotnie) wielkość parametru r.

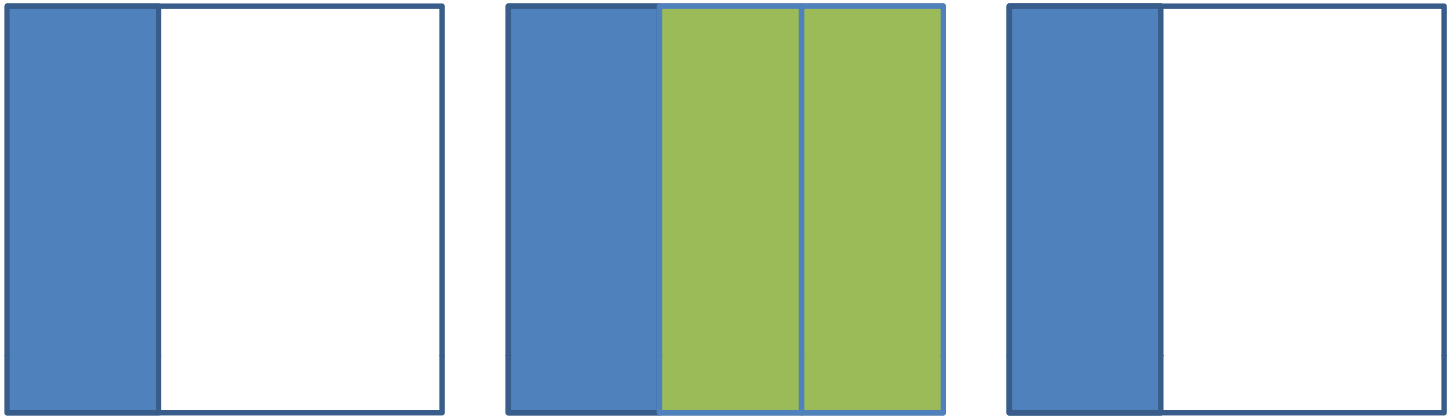
Mnożenie macierzy w systemach z pamięcią rozproszoną

WYKŁAD Z PRZETWARZANIA RÓWNOLEGŁEGO KWIECIEŃ 2018

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.

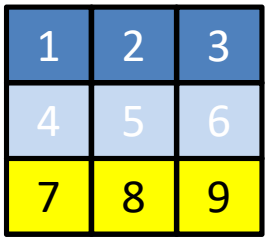
Mnożenie macierzy – systemy wieloprocesorowe z pamięcią rozproszoną (dystrybucja kolumnowa)[2]

```
//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] ;
```

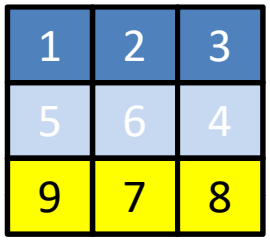
Każdy procesor przetwarza $(1+2/P)n^2$ elementów, każdy procesor rozsyła n^2/P elementów do P procesorów.

a_send, a_recv oznaczają operacje asynchroniczne odpowiednio nadawania i odbioru.

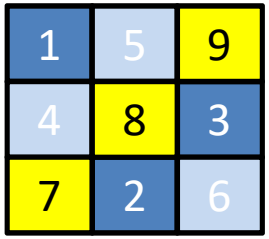
Mnożenie macierzy (dystrybucja szachownicowa)



Struktura systemu i przydzielone części tablicy wyniku UWAGA: właściciel oblicza



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



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:

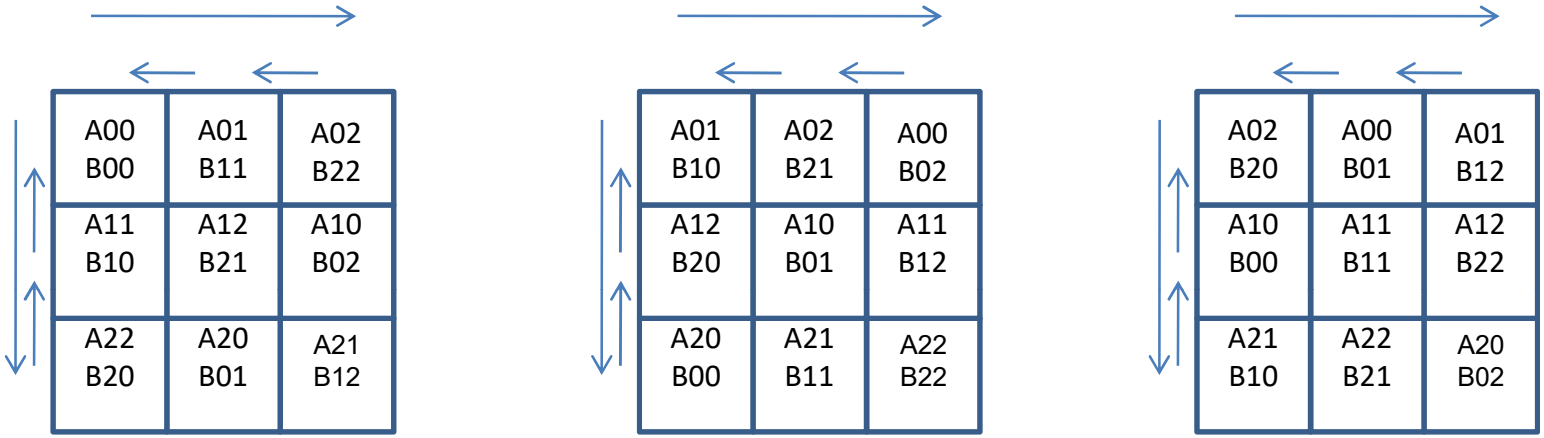
- od prawego sąsiada (cyklicznie) wartości jego aktualnej części tablicy A
- od dolnego sąsiada (cyklicznie) wartości jego aktualnej części tablicy B

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)



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;
}

}
```