Системы и средства параллельного программирования

кафедра СКИ
сентябрь – декабрь 2021 г.

Лектор доцент Н.Н.Попова

Лекция 10
8 ноября 2021 г.

Тема

- Параллельные алгоритмы умножения матрицы на вектор.
- Параллельные алгоритмы матричного умножения. Алгоритм Кеннона.

Умножение матрицы на вектор: c=Ab

$$A = \begin{bmatrix} a_{0,0} & a_{0,1}, & a_{0,2} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1}, & a_{1,2} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1}, & a_{m-1,2} & \cdots & a_{m-1,n-1} \end{bmatrix} \qquad b = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

$$c = Ab = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} \qquad c_i = a_{i,0}b_0 + a_{i,1}b_1 + a_{i,2}b_2 + \ldots + a_{i,n-1}b_{n-1}$$

Декомпозиция данных: матрица A

3 стратегии декомпозиции:

- Строчно-блочная: непрерывные группы (ленты) по [m/p] строк матрицы A

- - Столбцово-блочная: непрерывные группы (ленты) по [n/p] стобцов матрицы A

-

- - Шахматная (блочная) для матрицы A

Декомпозиция данных: вектора b и c

2 способа:

- Копирование векторов по всем процессам
- Блочное распределение векторов

Копирование векторов по процессам:

-память для A: mn значений

-память для b :n значений

- память для c: m значений

Три стратегии распараллеливания

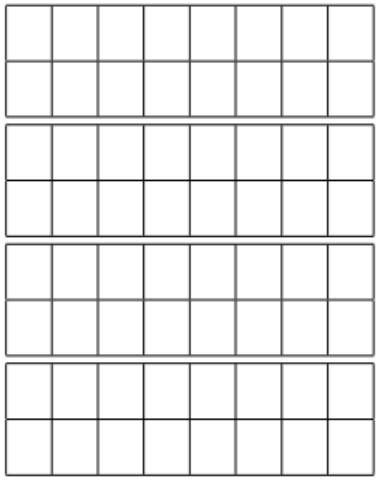
1. Строчно-ленточное для матрицы A, копирование векторов b и c

2. Ленточное по столбцам матрицы A, блочное распределение векторов

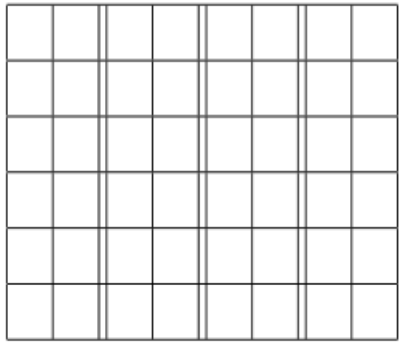
3. Шахматно-блочное распределение матрицы A, блочное распределение векторов

Стратегии распределения матрицы A

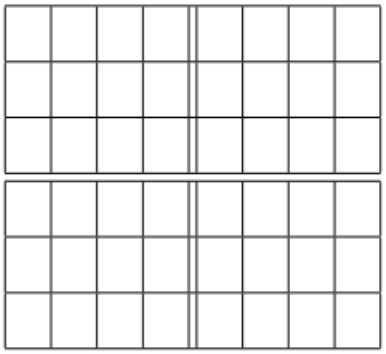
Rowwise block-striped



Columnwise block-striped



Checkerboard block



Параллельный алгоритм: строчно-блочное распределение

- Каждая строка матрицы A – элементарная задача
- Вектора b и c копируются по элементарным задачам

- Задача i определена для строки i и копии вектора b:

  - вычисление скалярного произведения между строкой i и вектроором b.
- Чтобы собрать весь вектор c для каждой элементарной задачи требуется сбор и копирование данных:

  - шаг all-gather

- Агломерация: объединяем группу строк в один процесс

Оценка сложности

- Пусть m=n, сложность последовательного алгоритма: O(*n*^2)
- Если используем *p* процессов, каждый процесс обрабатывает [*n/p*] строк A
- Вычислительная сложность (без коммуникаций) для каждого процесса: O(*n*^2/*p*)
- Эффективная реализация all-gather предполагает выполнение каждым процессом [log2p] пересылок

  - общее число пересылаемых данных равно n(p−1)/p

  - коммуникационная сложность: O(*n log2p*)
- Итоговая оценка: O(*n*^2/*p* +*n* $log_2 p$)

Параллельный алгоритм: столбцово-блочное распределение

- Столбец матрицы A – элементарная задача
- Каждая элементарная задача получает один элемент вектора b
- Задача i : умножение столбца i на i-ый элемент вектора b

  - результат – вектор частичных результатов A[i, *] x b[i]
- Рассылка частичных результатов по процессам: A[i,j] x b[i] отсылается j-ому процессу

  - MPI_All2all пересылка
- Каждая элементарная задача суммирует поступившие частичные результаты

Оценка сложности

- Агломерация: группируем несколько столбцов
  - каждый процесс получает по ⌈n/p⌉ столбцов A
  - каждый процесс получает блок ⌈n/p⌉ значений вектора b
  - каждый процесс вычисляет блок ⌈n/p⌉ значений вектора c
- Пусть m=n
- Сложность локального умножения: *O(n^2/p)*
- Сложность финального суммирования: *O(n)*
- Сложность all-to-all коммуникаций: ⌈*log2p*⌉ шагов
  - на каждом шаге процесс посылает n/p значений и получает n/p значений
  - сложность: *O(n/p* log2p*)
- Итоговая оценка: O(*n^2/p+n+n/p*log2p*)

Параллельный алгоритм: шахматно-блочное распределение

- Элементарная задача – элемент матрицы A.
- Действия элементарной задачи: умножение d_i,j = a_i,j * b_j
- Каждый элемент вектора c может быть вычислен как

$$c_i = \sum_{j=0}^{n-1} d_{i,j}$$

•Агломерация – объединяем несколько задач в блок

Три принципиальных шага

- Создание 2D процессной решетки: каждому процессу назначается блок Ai,j
- Предположим, что вектор b первоначально распределен на процесс с первым столбцом процессной решетки
- Шаг 1: перераспределить b таким образом, чтобы каждый процесс получил соответствующий блок вектора b_j
- Шаг 2: умножение Ai,j на b_j в каждом процессе
- Шаг 3: каждая строка процессной строки выполняет редукционное суммирование

Распределение вектора b

- Предположим, что размер 2D процессной решетки k×l
- Первоначально вектор b равномерно распределен по k процессам первого столбца процессной решетки
- Необходимо выделить коммуникаторы, соответствующие процессным строкам/столбцам
- Надо: распределить вектор b по процессной строке

  - если k=l, send b_i, в нулевую строку, broadcast b_i по i-ому столбцу
  - если k ≠l, gather, scatter, broadcast

Оценка сложности

- Предположим m=n и p – полный квадрат
- В каждом процессе хранится блок матрицы A размером $[n/\sqrt{p}]\times[n/\sqrt{p}]$

  - вычислительная сложность локального матричного умножения O(n^2/p)
- Сложность распределения вектора b

  - каждый процесс первого столбца посылает свой блок вектора b процессу в первой строке ⇒ сложность :$O(n/\sqrt{p})$

  - каждый процесс первой строки процессной матрицы broadcasts свой блок по процессному столбцу ⇒ сложность: $O(n\log_2 p/\sqrt{p})$
- Сложность финальной стадии (редукционная сумма): $O(n\log_2 p/\sqrt{p})$

```
/* parallel_mat_vect.c -- computes a parallel matrix-vector product.  Matrix
 *     is distributed by block rows.  Vectors are distributed by blocks.
 * Input:
 *     m, n: order of matrix
 *     A, x: the matrix and the vector to be multiplied
 *
 * Output:
 *     y: the product vector
 *
 * Notes:
 *     1.  Local storage for A, x, and y is statically allocated.
 *     2.  Number of processes (p) should evenly divide both m and n.
 *
 */
```

```
#include <stdio.h>
#include "mpi.h"

#define MAX_ORDER 100

typedef float LOCAL_MATRIX_T [MAX_ORDER][MAX_ORDER];

main(int argc, char* argv[]) {
    int             my_rank;
    int             p;
    LOCAL_MATRIX_T  local_A;
    float           global_x[MAX_ORDER];
    float           local_x[MAX_ORDER];
    float           local_y[MAX_ORDER];
    int             m, n;
    int             local_m, local_n;
```

```
void Read_matrix(char* prompt, LOCAL_MATRIX_T local_A, int
local_m, int n,   int my_rank, int p);
void Read_vector(char* prompt, float local_x[], int local_n, int my_rank,
        int p);
void Parallel_matrix_vector_prod( LOCAL_MATRIX_T local_A, int m,
        int n, float local_x[], float global_x[], float local_y[],
        int local_m, int local_n);
void Print_matrix(char* title, LOCAL_MATRIX_T local_A, int local_m,
        int n, int my_rank, int p);
void Print_vector(char* title, float local_y[], int local_m, int my_rank,
        int p);
```

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank == 0) {
    printf("Enter the order of the matrix (m x n)\n");
    scanf("%d %d", &m, &n);
}
MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

local_m = m/p;
local_n = n/p;
```

```
    Read_matrix("Enter the matrix", local_A, local_m, n, my_rank, p);
    Print_matrix("We read", local_A, local_m, n, my_rank, p);

   Read_vector("Enter the vector", local_x, local_n, my_rank, p);
    Print_vector("We read", local_x, local_n, my_rank, p);

    Parallel_matrix_vector_prod(local_A, m, n, local_x, global_x,
        local_y, local_m, local_n);
    Print_vector("The product is", local_y, local_m, my_rank, p);

    MPI_Finalize();

} /* main */
```

```
/* All arrays are allocated in calling program. Note that argument m is unused   */
void Parallel_matrix_vector_prod(
        LOCAL_MATRIX_T  local_A     /* in  */,
        int             m           /* in  */,
        int             n           /* in  */,
        float           local_x[]   /* in  */,
        float           global_x[]  /* in  */,
        float           local_y[]   /* out */,
        int             local_m     /* in  */,
        int             local_n     /* in  */) {
    /* local_m = m/p, local_n = n/p */
    int i, j;
    MPI_Allgather(local_x, local_n, MPI_FLOAT, global_x, local_n, MPI_FLOAT,
                  MPI_COMM_WORLD);
    for (i = 0; i < local_m; i++) {
        local_y[i] = 0.0;
        for (j = 0; j < n; j++)
            local_y[i] = local_y[i] + local_A[i][j]*global_x[j];
    }  }  /* Parallel_matrix_vector_prod */
```

```
void Print_matrix(
      char*          title      /* in */,
      LOCAL_MATRIX_T  local_A    /* in */,
      int            local_m    /* in */,
      int            n          /* in */,
      int            my_rank    /* in */,
      int            p          /* in */) {
   int   i, j;
   float temp[MAX_ORDER][MAX_ORDER];
   MPI_Gather(local_A, local_m*MAX_ORDER, MPI_FLOAT, temp,
      local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);
   if (my_rank == 0) {
      printf("%s\n", title);
      for (i = 0; i < p*local_m; i++) {
         for (j = 0; j < n; j++)
            printf("%4.1f ", temp[i][j]);
         printf("\n");
      }
   } }  /* Print_matrix */
```

Параллельные алгоритмы матричного умножения.

Блочный алгоритм Кеннона.

Матричное умножение

Существует множество вариантов матричного умножения на многопроцессорных системах.

Алгоритм решения существенным образом зависит от распределения матриц по процессам (аналогично рассмотренным вариантам умножения матрицы на вектор)

Умножение матриц

Умножение матриц

C= A * **B**, где

A - *n* x *l матрица* and B *l* x *m матрица*

ci,j (0 <= *i* < *n*, 0 <= *j* < *m*) вычисляются :

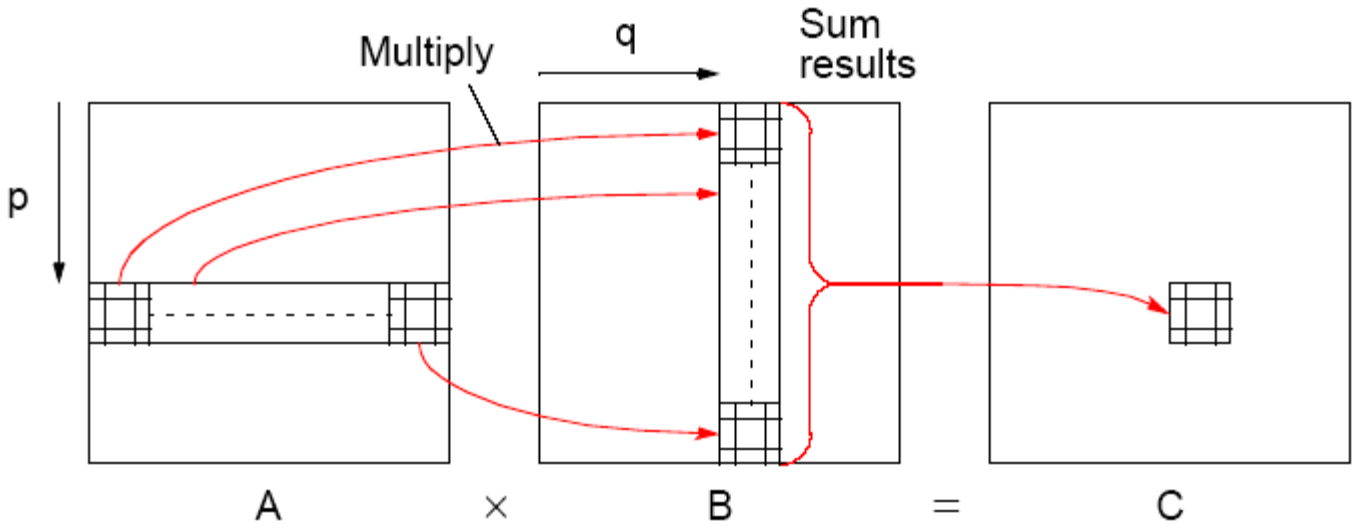
$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

Последовательный алгоритм

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
     c[i][j] = 0;
      for (k = 0; k < n; k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

- *n*3 операций умножения и *n*3 операций сложения
- Сложность алгоритма O(*n*3).

Блочное умножение матриц



Блочный алгоритм

Делим матрицы на подматрицы

Пусть матрицы разделены на *s*^2 подматриц.
Каждый блок содержит *n*/*s* x *n*/*s элементов*.
Обозначим Ap,q - блок матрицы A

```
for (p = 0; p < s; p++)
  for (q = 0; q < s; q++) {
    Cp,q = 0;                          /* обнуление блоков */
    for (r = 0; r < m; r++)            /*блочное умножение  */
      Cp,q = Cp,q + Ap,r * Br,q;       /*и сложение блоков*/
}
```

Строка

Cp,q = Cp,q + Ap,r * Br,q;

означает умножение блоков Ap,r и Br,q, используя матричное умножение

Распределение матриц по процессам

Каждая из трех матриц (A,B и C) может быть распределена одним из способов:

- копии матриц находятся в каждом процессе;
- матрицы распределены по столбцам;
- матрицы распределены по строкам;
- матрицы распределены блочно на двумерную или трехмерную процессную сетку.

Могут использоваться различные комбинации. Все зависит от решаемой задачи.

Пример распределения матриц по процессам

- Матрицы A и C – распределены по процессам «ленточно» - равным количеством строк.

  Это 1D распределение (ленточное, по строкам)

- Матрица B – 1D ленточно, по столбцам.

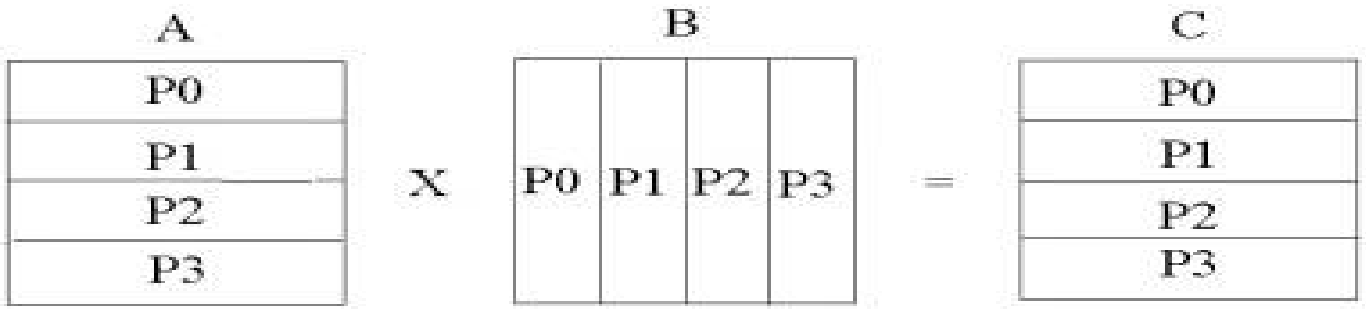


Схема параллельного алгоритма.

Распределение матриц: A,C – построчно, B – по столбцам

- Пусть размер матриц n x n, число процессов – p, n % p =0
- Определим L = n/p – количество строк или столбцов для каждого процесса. Блок = «лента» матрицы
- У каждого процесса хранятся ленты A, B, C
- Создадим одномерную процессную решетку с L строками матриц A и C и L столбцами матрицы B в каждом процессе

Схема параллельного алгоритма.

Распределение матриц: A,C – построчно, B – по столбцам

Алгоритм:

1. /*Инициализация лент матриц A, B и C: Alocal, Blocal, Clocal */
2. Определение rank каждого процесса в одномерной решетке
3. for (k=0; k< p k++) {
   Clocal[k+rank] = Alocal x Blocal;
   // Циклически сдвинуть ленту B_local на 1 влево
   }

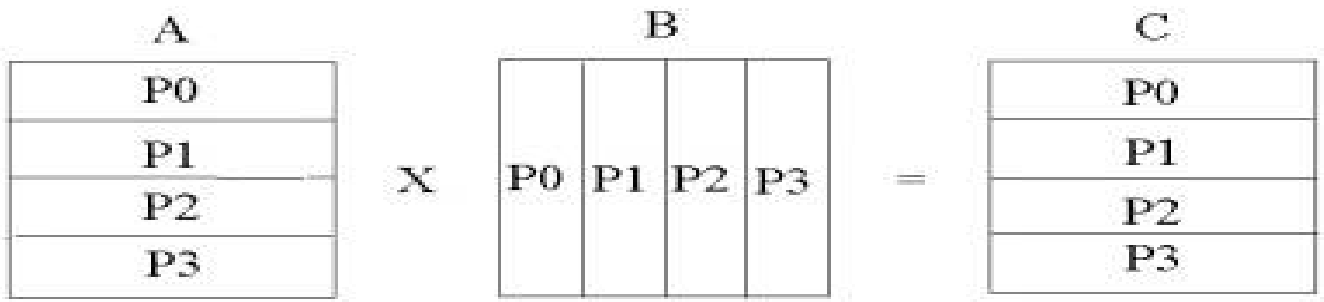


Схема параллельного алгоритма.
Распределение матриц: A,B, C – построчно

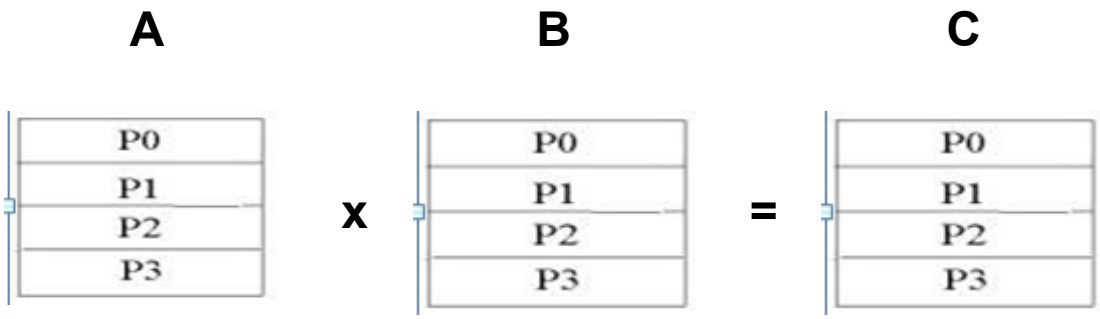


Схема параллельного алгоритма.
Распределение матриц: A,B,C – построчно

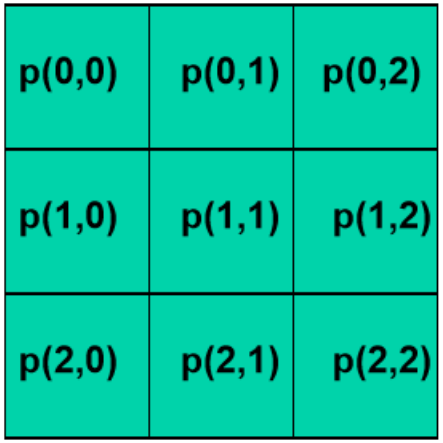
- У каждого процесса хранится лента A, B, C

Алгоритм:

1. Инициализация лент матриц Alocal, Blocal и Clocal
2. Транспонирование матрицы B, используя в каждом процессе дополнительный буфер: Btr-local – лента-столбец матрицы B
3. Далее см. предыдущий алгоритм

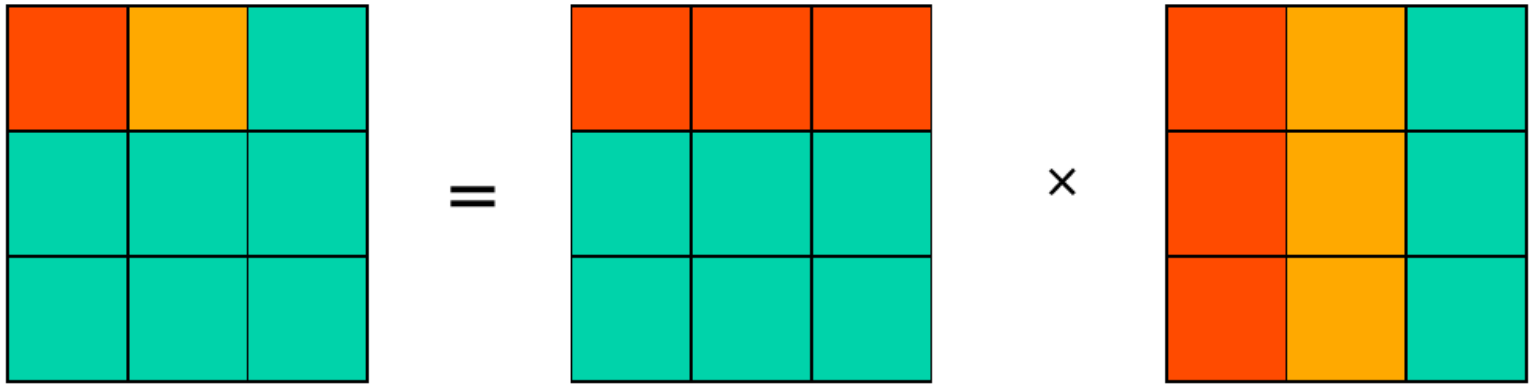
Блочный параллельный алгоритм матричного умножения

-Организуем виртуальную топологию
- Координаты процесса – пара (I,j)
- Предположим, что p – полный квадрат
- Каждый процесс содержит подматрицу размером
  $n/\sqrt{p} \times n/\sqrt{p}$
- Предположим, что имеется эффективное последовательное матричное умножение (dgemm, sgemm)



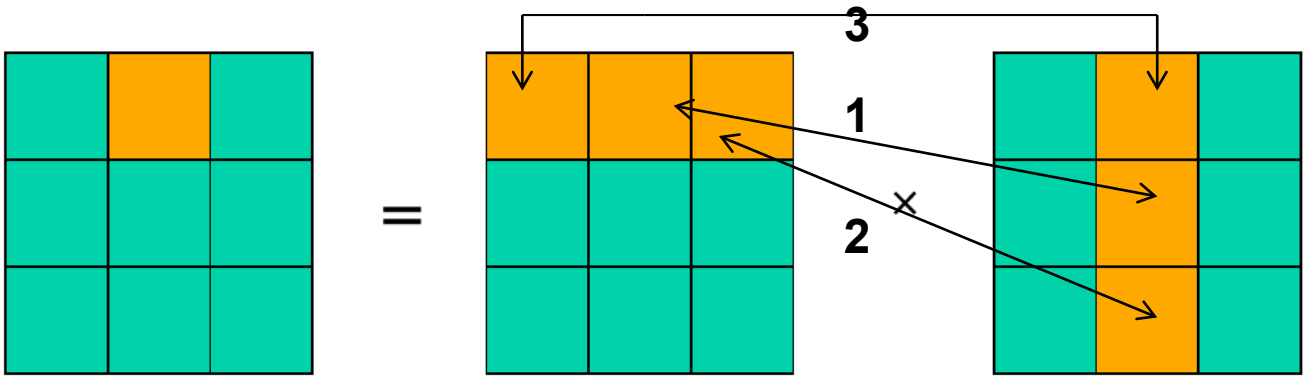
Блочный параллельный алгоритм

- Каждый элемент A[i,j] будем трактовать как блок
- A[i,k] * B[k,j] – матричное умножение
- C[i, j] += A[i, k] * B[k, j]



Блочный параллельный алгоритм

- Используем преимущества организации 2-мерной процессной решетки
- В каждом процессе храним только блоки матриц
- Пересылаем блоки между процессами процессной решетки
- $s=\sqrt{p}$ шагов алгоритма



Алгоритм Кеннона

- Основан на описанном алгоритме
- Рассмотрим, например, итерацию i=1, j=2:

C[1,2] = A[1,0]*B[0,2] + A[1,1]*B[1,2] + A[1,2]*B[2,2]