Параллельное программирование для высокопроизводительных вычислительных систем

сентябрь – декабрь 2018 г.

Лектор доцент Н.Н.Попова

Лекция 9
19 ноября 2018 г.

Тема

- Параллельный алгоритм матричного умножения SUMMA
- 3D блочный параллельный алгоритм матричного умножения DNS
- 2,5D алгоритм матричного умножения

Ограничения алгоритмов Фокса и Кеннона

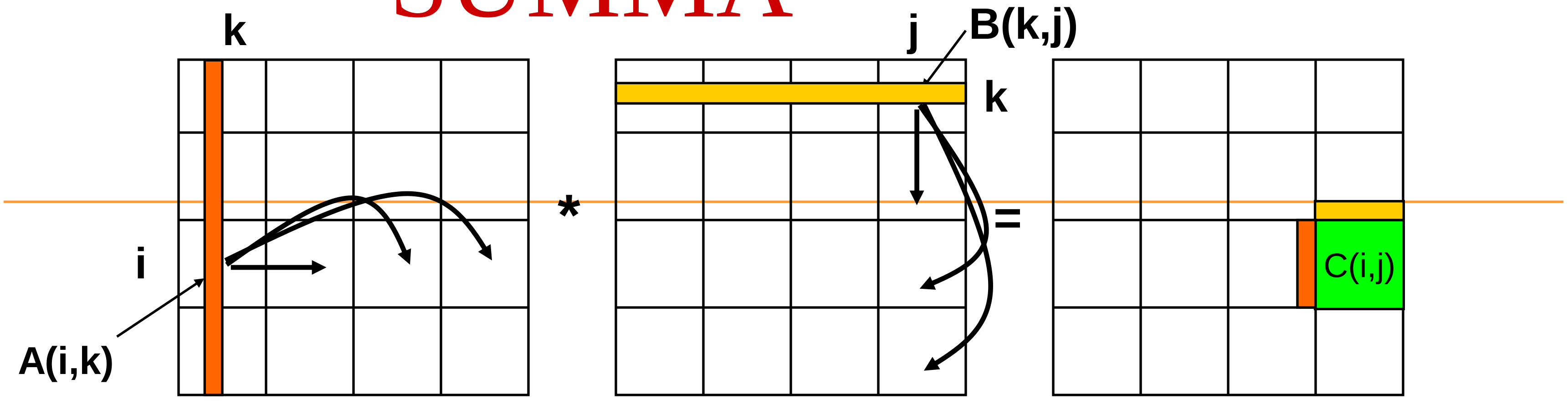
- Трудно обобщаются для случаев:
  - p не полный квадрат
  - A и B не квадратные
  - Размерности A, B не делятся нацело на s=sqrt(p)
- Требуется дополнительная память для хранения копий блоков

Алгоритм SUMMA

- **SUMMA** = Scalable Universal Matrix Multiply*
- Менее эффективный, чем алгоритм Кеннона, но проще и легче обобщается на случай разных способов распределения данных
- Требует меньше дополнительной памяти, но в то же время и больше пересылок (в log p раз больше, чем в методе Кеннона)
- Используется на практике в PBLAS = Parallel BLAS

*   R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. Concurrency: Pract. Ex. , 9(4):255–274, 1997

SUMMA

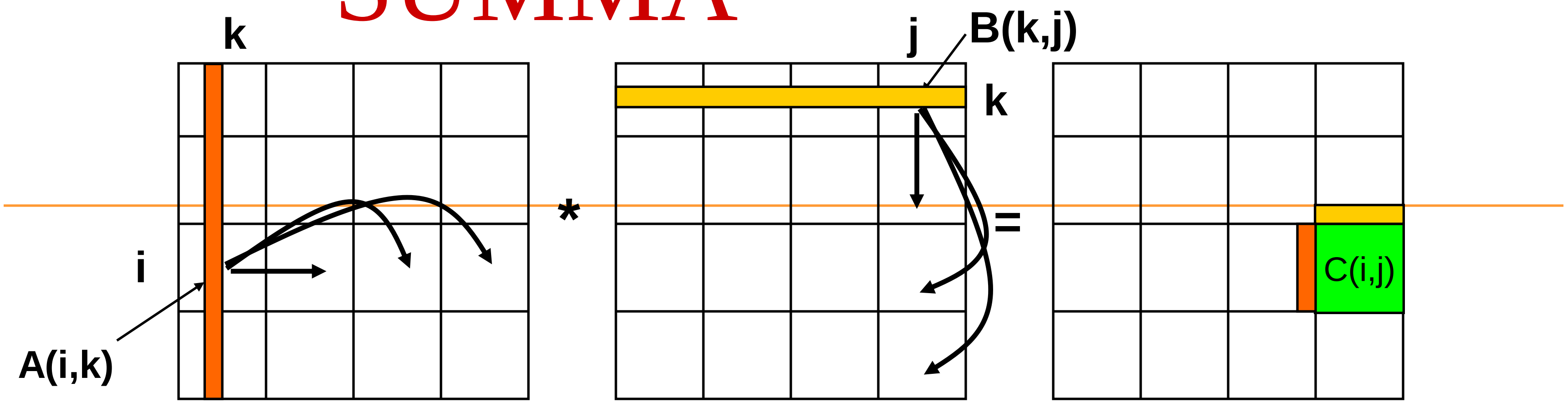


- Процессорная решетка не обязательно должна быть квадратной: $P = pr * pc$
- $b << N/\max(px,py)$ – размер блока
- k – блок с $b \geq 1$ строками или столбцами

  $C(i,j) = \Sigma_k A(i,k)*B(k,j)$

5

SUMMA



for k=0 to n-1 … или n/b-1 где b – размер блока
… = # cols in A(i,k) and # rows in B(k,j)

for all i = 1 to p_r … in parallel

owner of A(i,k) broadcasts it to whole processor row

for all j = 1 to p_c … in parallel

owner of B(k,j) broadcasts it to whole processor column

Receive A(i,k) into Acol

Receive B(k,j) into Brow

C_myproc = C_myproc + Acol * Brow

Оценка времени выполнения алгоритма SUMMA

- Для упрощения преположим, что s = sqrt(p)

```
for k=0 to n/b-1
    for all i = 1 to s   …  s = sqrt(p)
        owner of A(i,k) broadcasts it to whole processor row
          … time = log s *( α + β * b*n/s), используя дерево
    for all j = 1 to  s
        owner of B(k,j) broadcasts it to whole processor column
          … time = log s *( α + β * b*n/s), используя дерево
    Receive A(i,k) into Acol
    Receive B(k,j) into Brow
    C_myproc = C_myproc + Acol * Brow
          … time = 2*(n/s)^2*b
```

2D параллельные алгоритмы матричного умножения

- **2D**
  - **Cannon**
    - Эффективность = 1/(1+O($\alpha * (\text{sqrt}(p) / n)^3 + \beta * \text{sqrt}(p) / n$)) – оптимальная
    - Трудно обобщать на случай произвольного p, n, блочно-циклического распределения данных
  - **SUMMA**
    - Эффективность = 1/(1 + O($\alpha * \log p * p / (b*n^2) + \beta * \log p * \text{sqrt}(p) / n$))
    - Легко обобщается
    - b маленькое => меньше памяти, меньше эффективность
    - b большое => больше памяти, выше эффективность
    - Используется на практике (PBLAS)

Matrix Multiply: DNS алгоритм
Dekel, Nassimi, Sahni

Оценка времени выполнения: O(log N), используя O(N^3/log N) процессов.

Предположим::

A, B, C: размера N x N

$$C_{rs} = \sum_{t=1}^{K} A_{rt} B_{ts}$$

P = K^3 число процессоров, организованных в K x K x K 3D решетку,

A, B, C - K x K блочные матрицы, каждый блок (N/K) x (N/K)

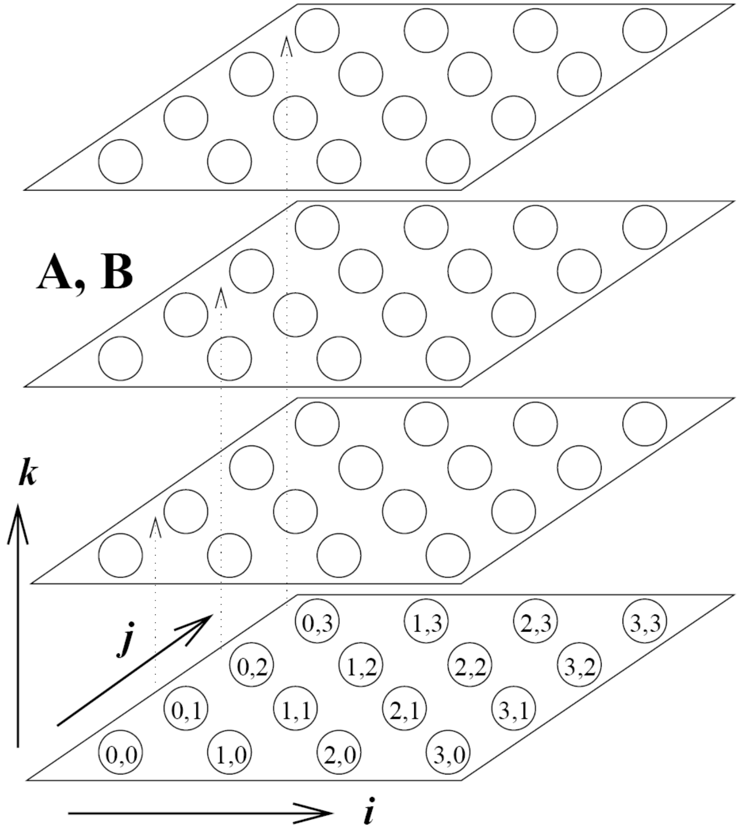
Общее число K*K*K блочных матричных умножений

Идея: каждый блок назначается на отдельный процессор

Процессор (i,j,k) вычисляет C_{ij}=A_{ik}*B_{kj}

Вычисляется редукционная сумма (i,j,k), k=0,…,K-1

Matrix Multiply: DNS алгоритм



(a) Initial distribution of *A* and *B*

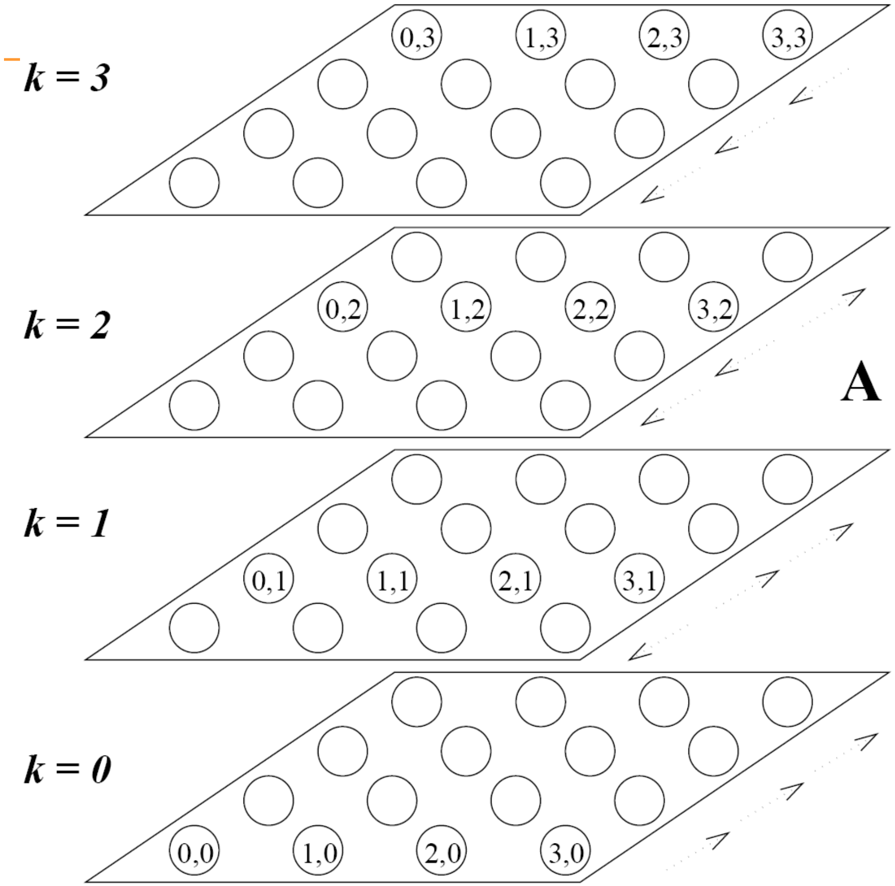
Начальное распределение данных: A_{ij} и B_{ij} на процессор (i,j,0)

Передать A_{ik} (i,k=0,…,K-1) на процессор (i,j,k) for all j=0,1,…,K-1

Два шага:

- Переслать A_{ik} с процессора (i,k,0) на (i,k,k);
- Broadcast A_{ik}c процессора (i,k,k) на процессоры (i,j,k);

Matrix Multiply

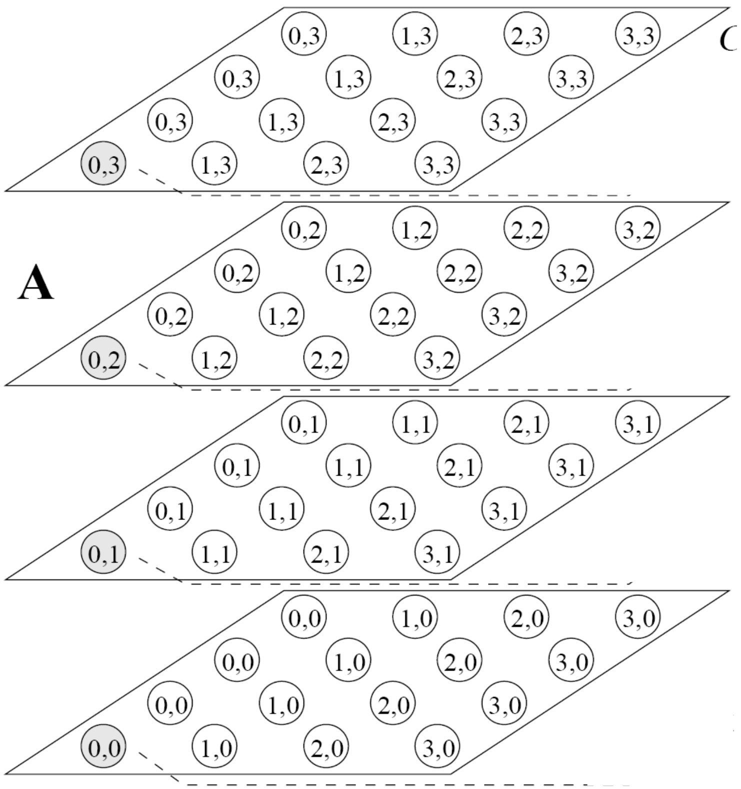


(b) After moving $A[i,j]$ from $P_{i,j,0}$ to $P_{i,j,j}$

Переслать A_{ik} с (i,k,0) на (i,k,k)

Broadcast A_{ik} с (i,k,k) на (i,j,k)

Matrix Multiply

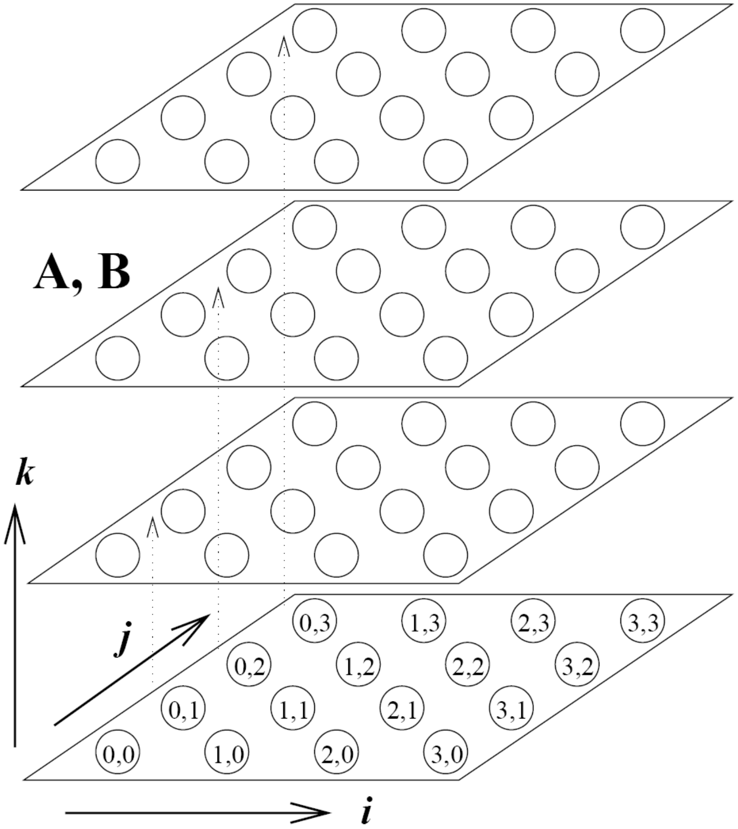


(c) After broadcasting *A[i,j]* along *j* axis

Финальное распределение блоков матрицы A

A может быть рассмотрена как распределенная по (i,k) плоскости с broadcast вдоль j-оси.

Распределение элементов матрицы B

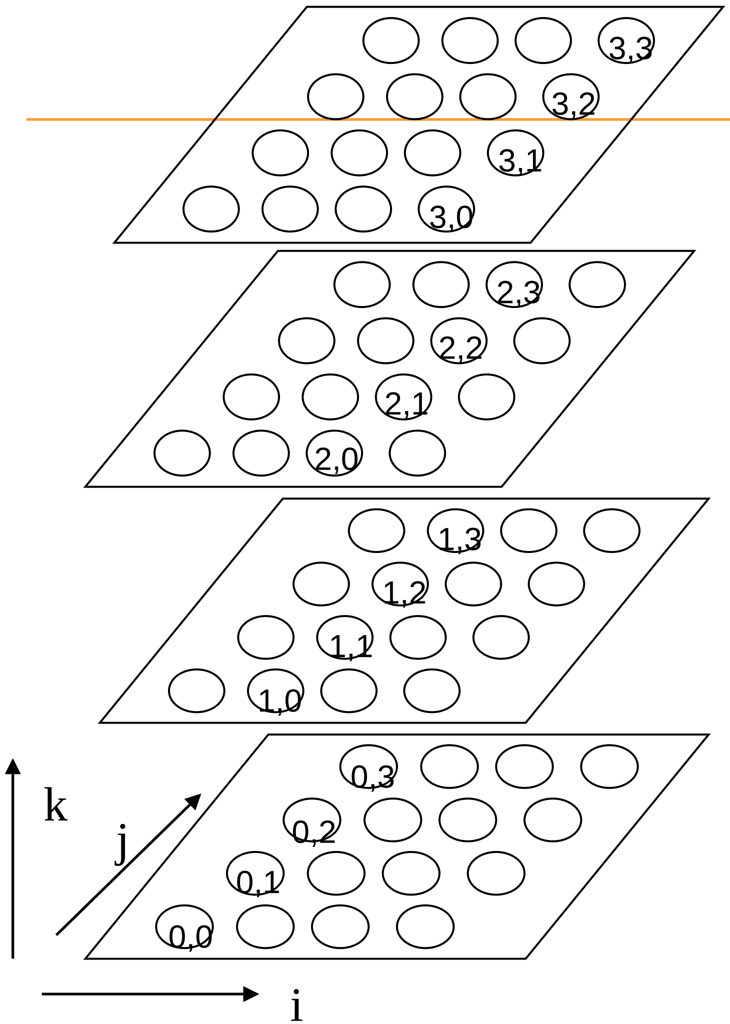


(a) Initial distribution of A and B

B распределение:
1. B_{kj} на (k,j,0);
Требуется передать на процессоры (i,j,k) for all i=0,1,…,K-1

Два шага:
-Переслать B_{kj} с (k,j,0) на (k,j,k)
-Broadcast B_{kj} с (k,j,k) на (i,j,k) for all i=0,…,K-1, т.е. вдоль i-направления

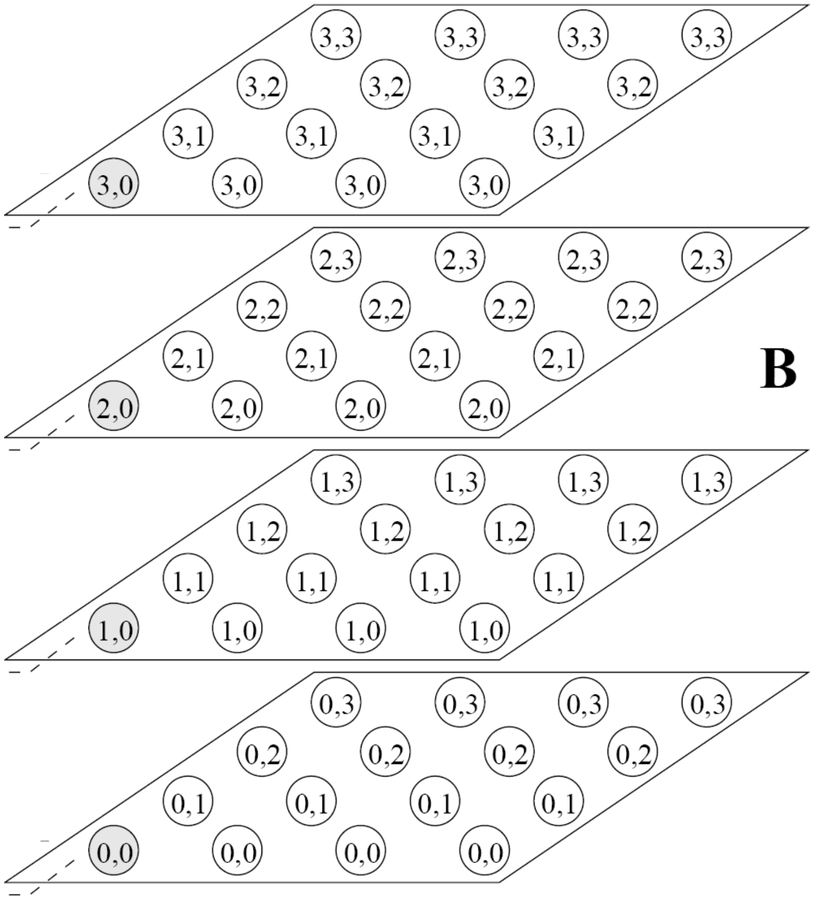
Распределение матрицы B



Переслать B_{kj} с (k,j,0) на (k,j,k)

Broadcast (k,j,k) вдоль i направления

Matrix Multiply

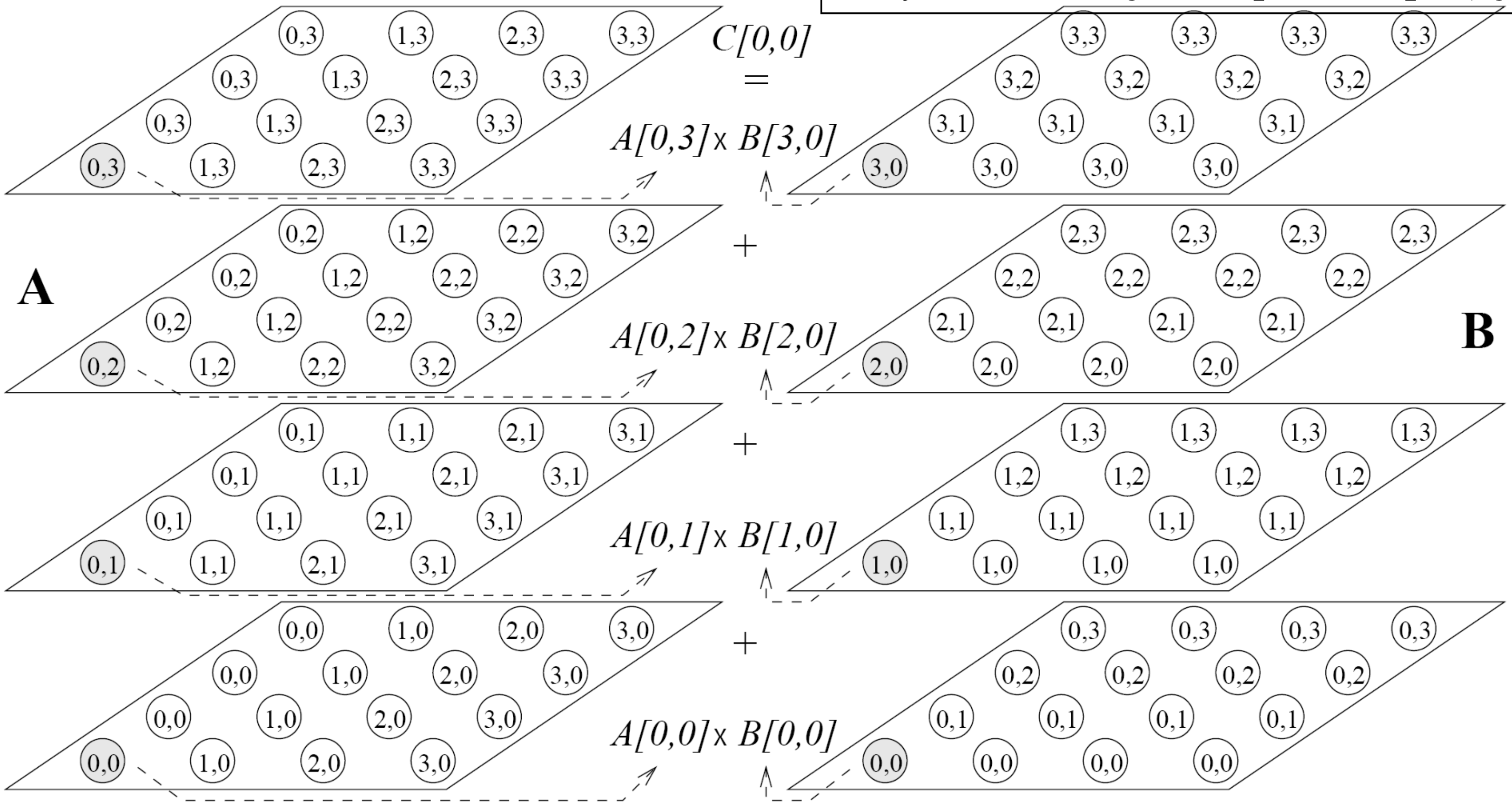


(d) Corresponding distribution of *B*

Финальное распределение B

Matrix Multiply

A_{ik} и B_{kj} на процессорах (i,j,k)
Вычисляем C_{ij} локально
Reduce (sum) C_{ij} вдоль k-направления
Результат: C_{ij} на процессоре (i,j,0)



(c) After broadcasting $A[i,j]$ along j axis

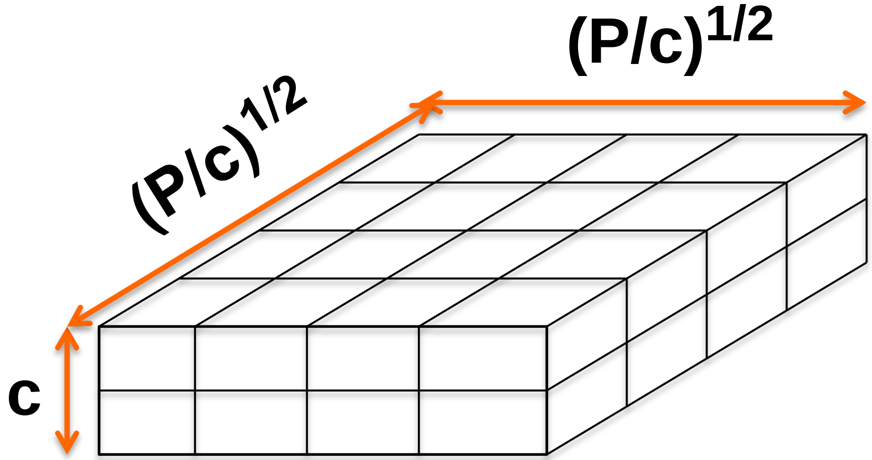
(d) Corresponding distribution of B

2.5D матричное умножение
Communication avoiding algorithm

Edgar Solomonik and James Demmel.
Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In Proceedings of the 17th international conference on Parallel processing - Volume Part II , Euro-Par’11, pages 90–109, Berlin, Heidelberg, 2011. Springer-Verlag.

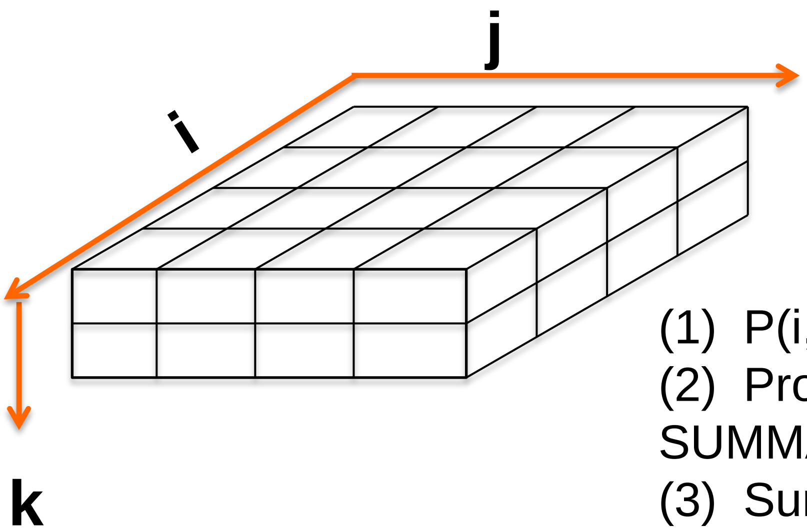
2.5D матричное умножение

- Предположим, что мы можем разместить cn^2/P данных на процессор, $c>1$
- Процессоры формируют $(P/c)^{1/2}$ x $(P/c)^{1/2}$ x c сетку



Пример: P = 32, c = 2

2.5D Matrix Multiplication



Начальное распределение данных:
на P(i,j,0) находятся A(i,j) and B(i,j)
каждый размером $n(c/P)^{1/2}$ x $n(c/P)^{1/2}$

(1) P(i,j,0) broadcasts A(i,j) and B(i,j) to P(i,j,k)
(2) Processors at level k perform 1/c-th of SUMMA, i.e. 1/c-th of Σ_m A(i,m)*B(m,j)
(3) Sum-reduce partial sums Σ_m A(i,m)*B(m,j) along k-axis so P(i,j,0) owns C(i,j)

2.5D Matrix Multiplication

Algorithm 2: $[C] = $ 2.5D-matrix-multiply(A,B,n,p,c)

Input: square n-by-n matrices A, B distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ blocks A_{ij} and B_{ij} for each i, j

Output: square n-by-n matrix $C = A \cdot B$ distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ block C_{ij} for each i, j

```
/* do in parallel with all processors                                    */
forall i, j ∈ {0, 1, ..., √(p/c) − 1}, k ∈ {0, 1, ..., c − 1} do
    P_ij0 broadcasts A_ij and B_ij to all P_ijk          /* replicate input matrices */
    s := mod (j − i + k√(p/c³), √(p/c))                  /* initial circular shift on A */
    P_ijk sends A_ij to A_local on P_isk
    s' := mod (i − j + k√(p/c³), √(p/c))                 /* initial circular shift on B */
    P_ijk sends B_ij to B_local on P_s'jk
    C_ijk := A_local · B_local
    s := mod (j + 1, √(p/c))
    s' := mod (i + 1, √(p/c))
    for t = 1 to √(p/c³) − 1 do
        P_ijk sends A_local to P_isk                     /* rightwards circular shift on A */
        P_ijk sends B_local to P_s'jk                    /* downwards circular shift on B */
        C_ijk := C_ijk + A_local · B_local
    end
    P_ijk contributes C_ijk to a sum-reduction to P_ij0
end
```

2.5D Matrix Multiplication

Algorithm 2: $[C]$ – 2.5D-matrix-multiply(A,B,n,p,c)

Input: square n-by-n matrices A, B distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ blocks A_{ij} and B_{ij} for each i, j

Output: square n-by-n matrix $C = A \cdot B$ distributed so that P_{ij0} owns $\frac{n}{\sqrt{p/c}}$-by-$\frac{n}{\sqrt{p/c}}$ block C_{ij} for each i, j

/ do in parallel with all processors */*

forall $i, j \in \{0, 1, ..., \sqrt{p/c} - 1\}$, $k \in \{0, 1, ..., c - 1\}$ **do**

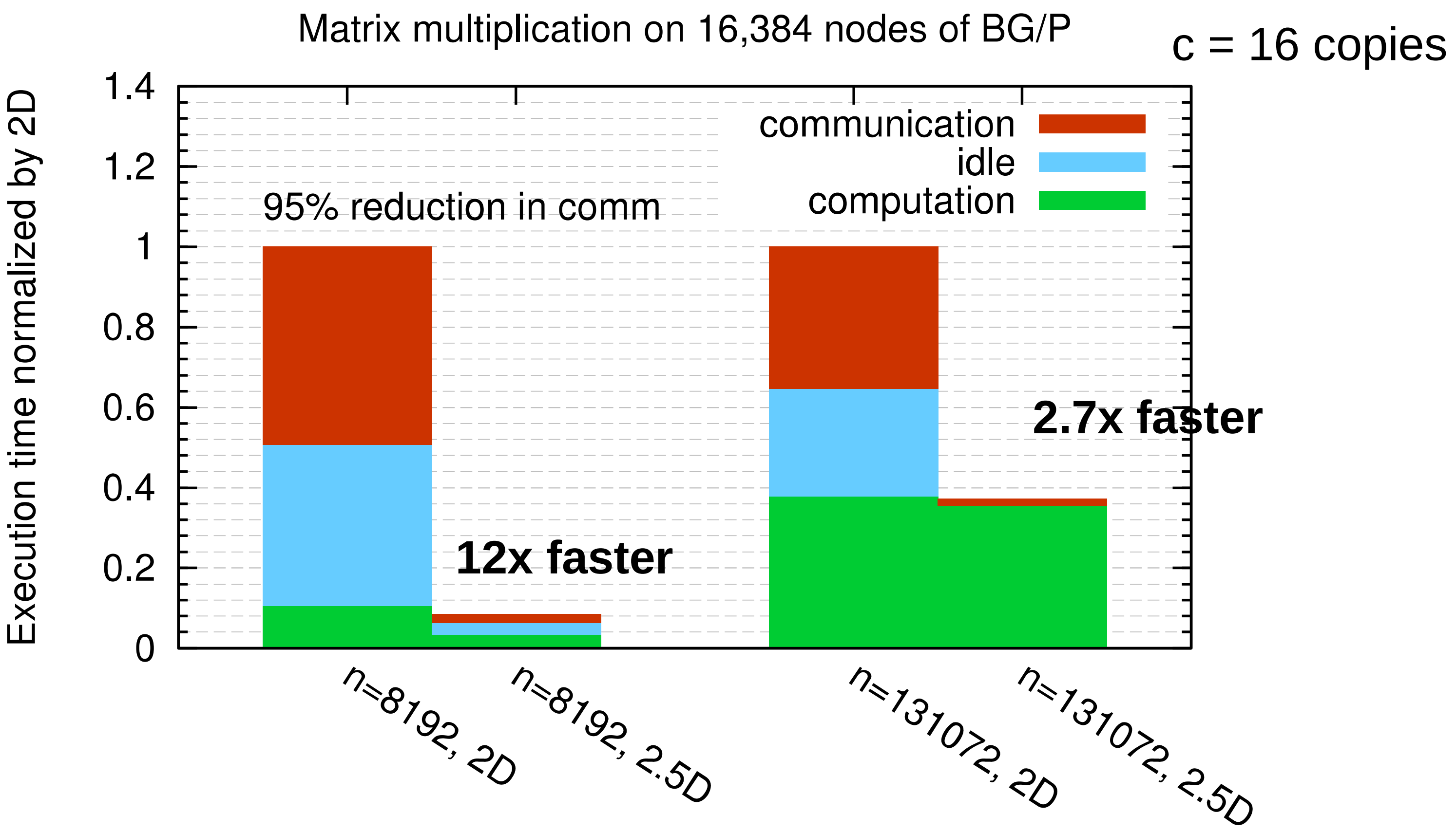
- P_{ij0} broadcasts A_{ij} and B_{ij} to all P_{ijk} — */* replicate input matrices */*
- $s :=$ mod $(j - i + k\sqrt{p/c^3}, \sqrt{p/c})$ — */* initial circular shift on A */*
- P_{ijk} sends A_{ij} to A_{local} on P_{isk}
- $s' :=$ mod $(i - j + k\sqrt{p/c^3}, \sqrt{p/c})$ — */* initial circular shift on B */*
- P_{ijk} sends B_{ij} to B_{local} on $P_{s'jk}$
- $C_{ijk} := A_{\text{local}} \cdot B_{\text{local}}$
- $s :=$ mod $(j + 1, \sqrt{p/c})$
- $s' :=$ mod $(i + 1, \sqrt{p/c})$
- **for** $t = 1$ **to** $\sqrt{p/c^3} - 1$ **do**
  - P_{ijk} sends A_{local} to P_{isk} — */* rightwards circular shift on A */*
  - P_{ijk} sends B_{local} to $P_{s'jk}$ — */* downwards circular shift on B */*
  - $C_{ijk} := C_{ijk} + A_{\text{local}} \cdot B_{\text{local}}$
- **end**
- P_{ijk} contributes C_{ijk} to a sum-reduction to P_{ij0}

end

Эквивалентно алгоритму Кеннона

Эквивалентно алгоритму Кеннона

2.5D Matmul, BG/P, 16K процессоров, 64K ядер



Умножение разреженных матриц

- Встречаются разные формулировки:
  - Разреженной называют матрицу, имеющую малый процент ненулевых элементов.
  - Матрица размера N x N называется разреженной, если количество ее ненулевых элементов есть O(N).
  - Известны и другие определения.

Приведенные варианты являются не вполне точными в математическом смысле.

На практике классификация матрицы зависит не только от количества ненулевых элементов

Способы хранения разреженных матриц

– Джордж А., Лю Дж. Численное решение больших разреженных систем уравнений. – М.: Мир, 1984. – Писсанецки С. Технология разреженных матриц. — М.: Мир,1988. – Тьюарсон Р. Разреженные матрицы. – М.: Мир, 1977.

Координатный формат

- Элементы матрицы и ее структура хранятся в трех массивах, содержащих значения, их X и Y координаты.
- Оценим объем необходимой памяти (M):
  - Плотное представление: M = 8 N^ 2 байт.
  - В координатном формате: M = 8 NZ + 4 NZ + 4 NZ = 16 NZ байт
  - Ясно, что 16 NZ << 8 N^2, если NZ << N^2.

Формат CRS (CSR)

- Формат хранения CSR (Compressed Sparse Rows) или CRS (Compressed Row Storage),
- Используются три массива:

– первый массив хранит значения элементов построчно,

– второй – номера столбцов для каждого элемента,

– третий заменяет номера строк, используемые в координатном формате, на индекс начала каждой строки.

Пример

A

1				2	
		3	4		
			8		5
	7	1			6

Структура хранения:

1	2	3	4	8	5	7	1	6

Value

0	4	2	3	3	5	1	2	5

Col

0	2	4	4	6	6	9

RowIndex