Средства и системы параллельного программирования

кафедра СКИ
сентябрь – декабрь 2021 г.

Лектор доцент Н.Н.Попова

Лекция 11
22 ноября 2021 г.

Тема

- Параллельный алгоритм матричного умножения SUMMA
- 3D блочный параллельный алгоритм матричного умножения DNS
- Производные типы данных

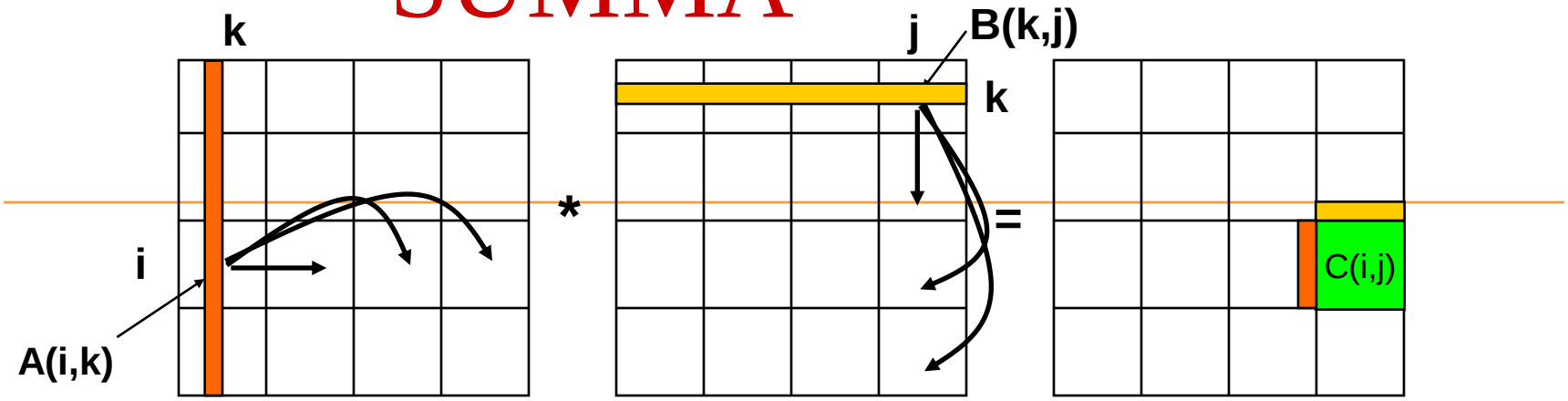
Алгоритм SUMMA

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

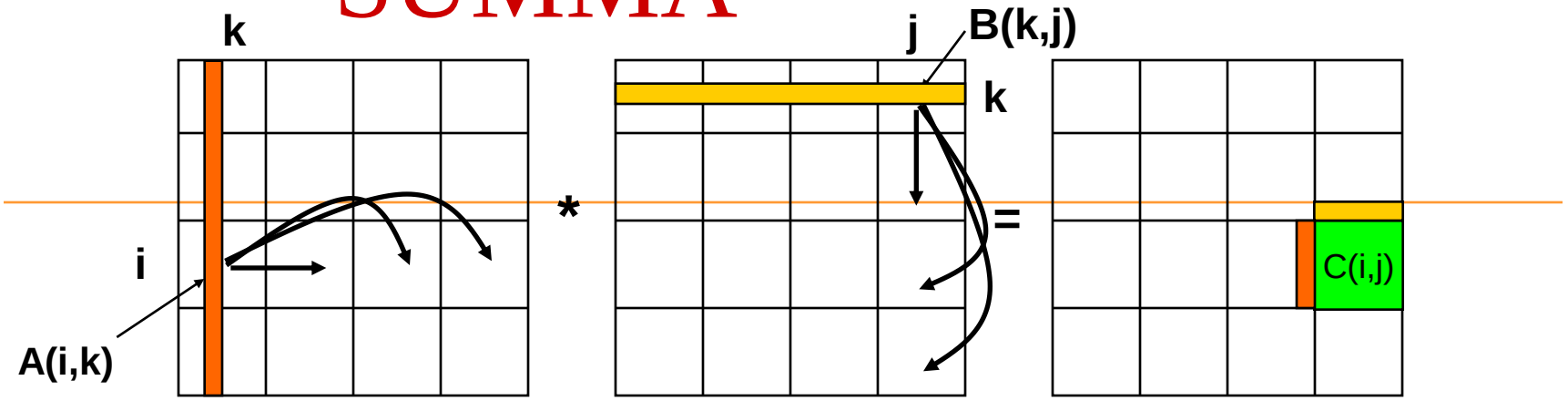
SUMMA



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

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

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

Matrix Multiply: DNS алгоритм

Dekel, **N**assimi, **S**ahni

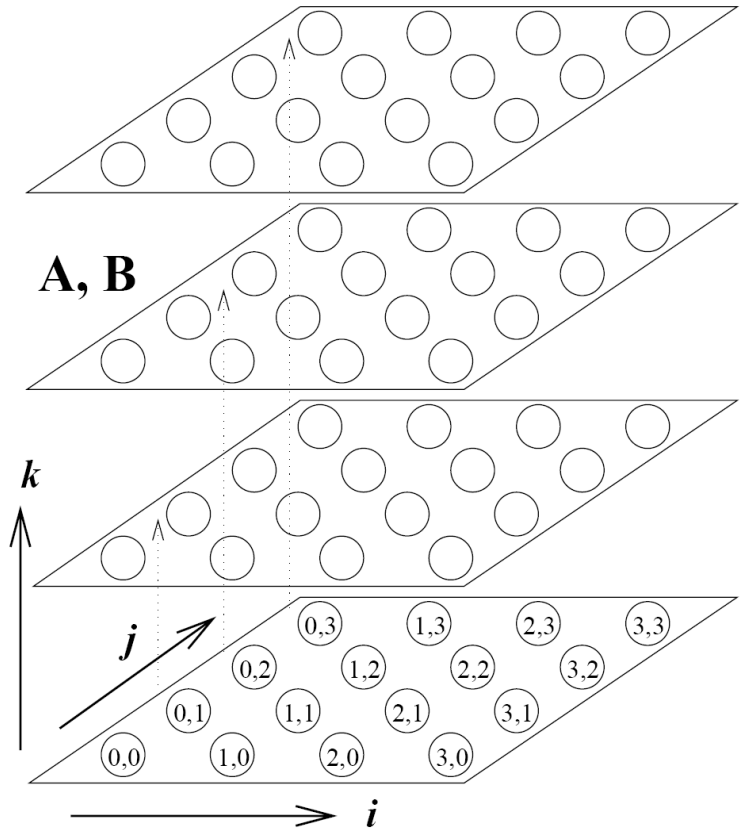
Предположим::
A, B, C: размера N x N
P = K^3 число процессоров, организованных в K x K x K 3D решетку,

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

Eliezer Dekel, David Nassimi, and Sartaj Sahni Parallel Matrix and Graph Algorithms SIAM J. Comput., 10(4), 657–675

Matrix Multiply: DNS алгоритм



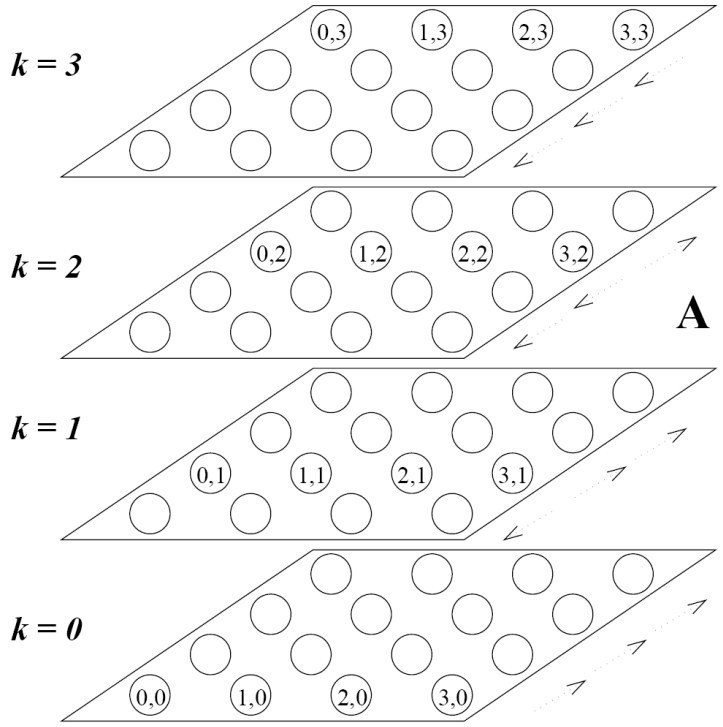
(a) Initial distribution of A and B

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

Переслать 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}с процессора (i,k,k) на процессоры (i,j,k);

Matrix Multiply

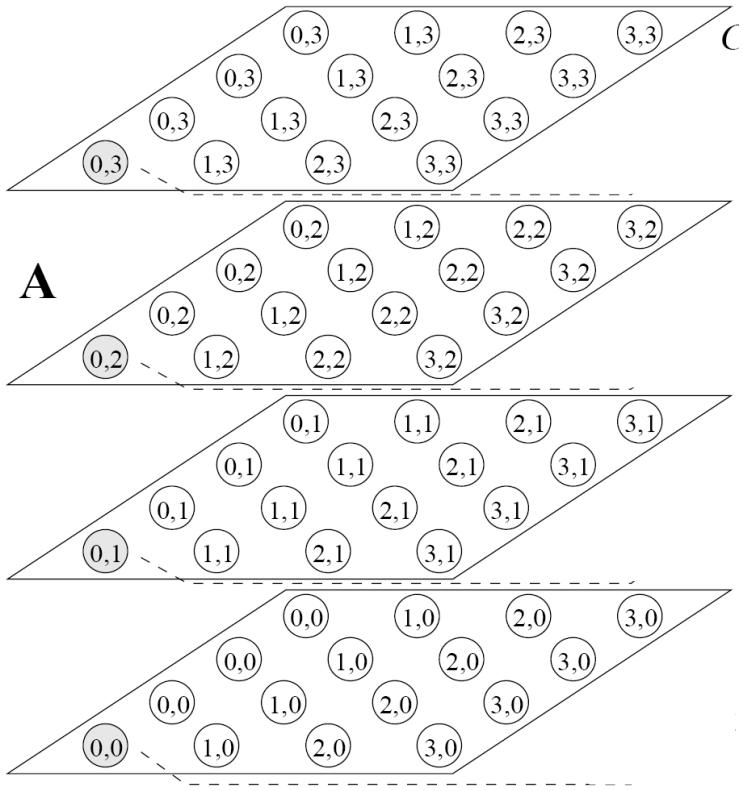


(b) After moving $A[i,j]$ from $P_{i,j,0}$ to $P_{i,j,j}$

Переслать A_{ik} с (i,k,0) на (i,k,k)

Broadcast A_{ik} с (i,k,k) на (i,j,k)

Matrix Multiply

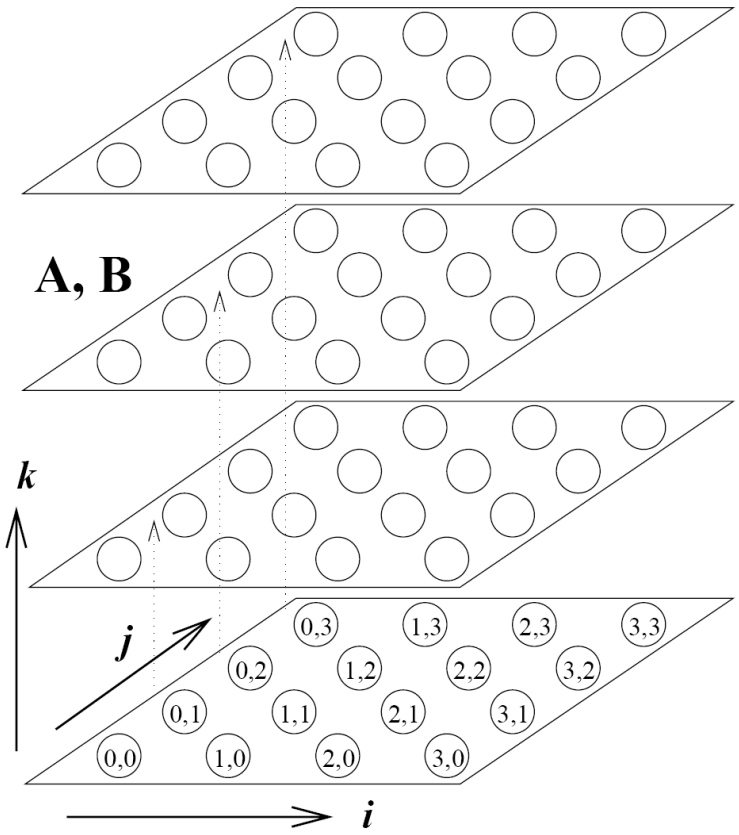


(c) After broadcasting *A[i,j]* along *j* axis

Финальное распределение блоков матрицы A

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

Распределение элементов матрицы B

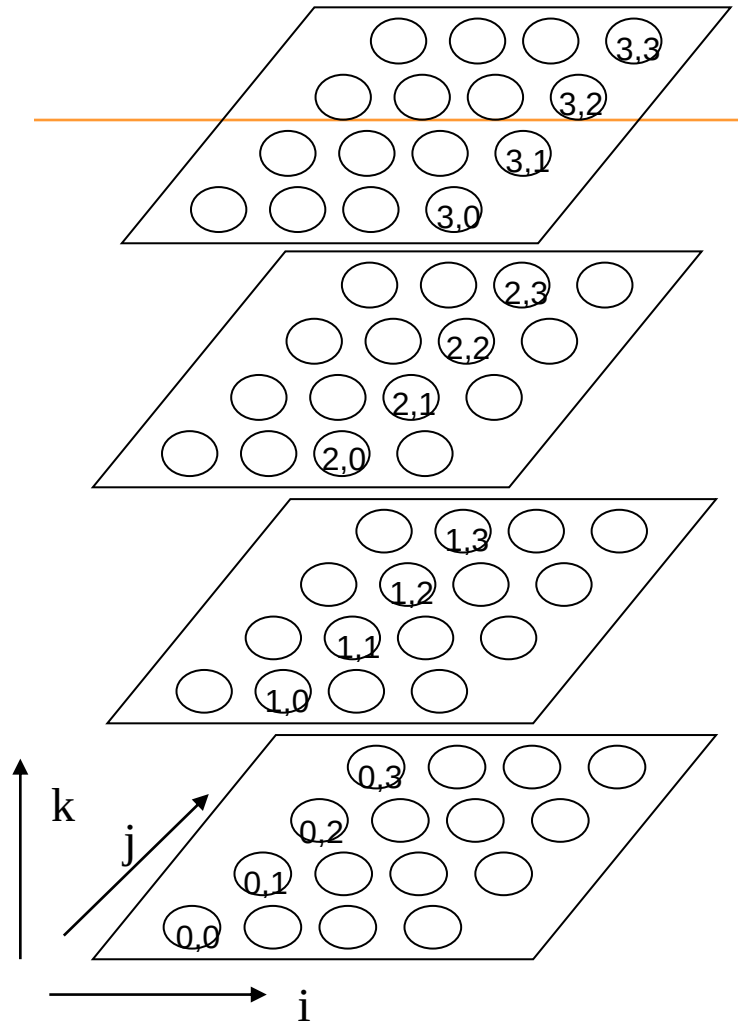


(a) Initial distribution of *A* and *B*

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

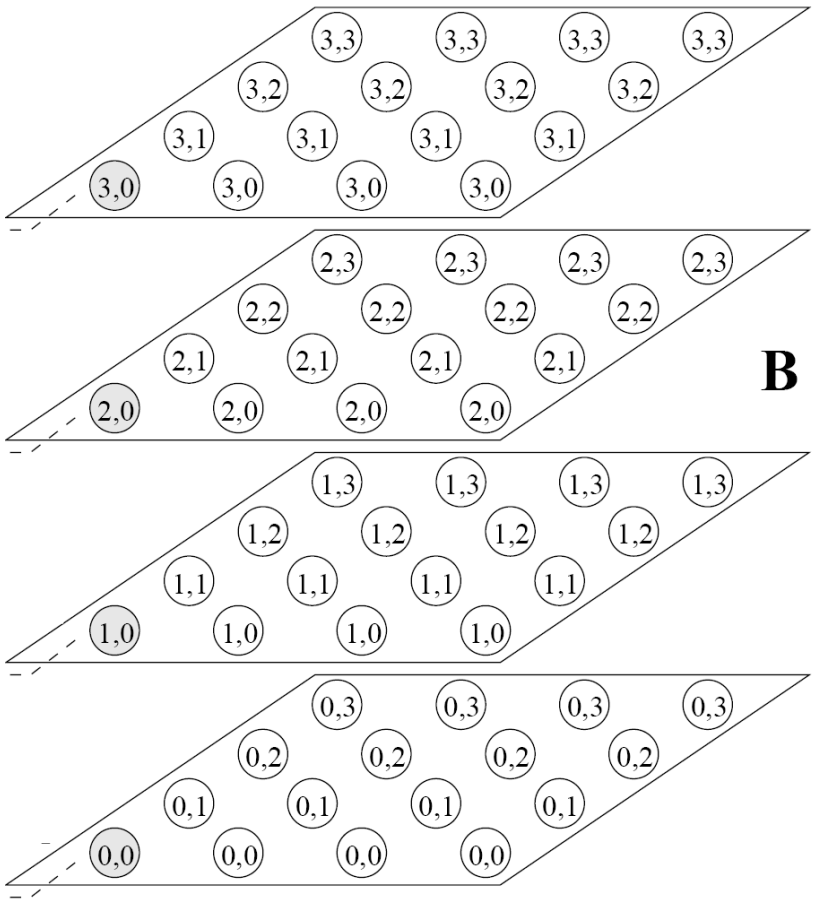
Распределение матрицы B



Переслать B_{kj} с (k,j,0) на (k,j,k)

Broadcast (k,j,k) вдоль i направления

Matrix Multiply

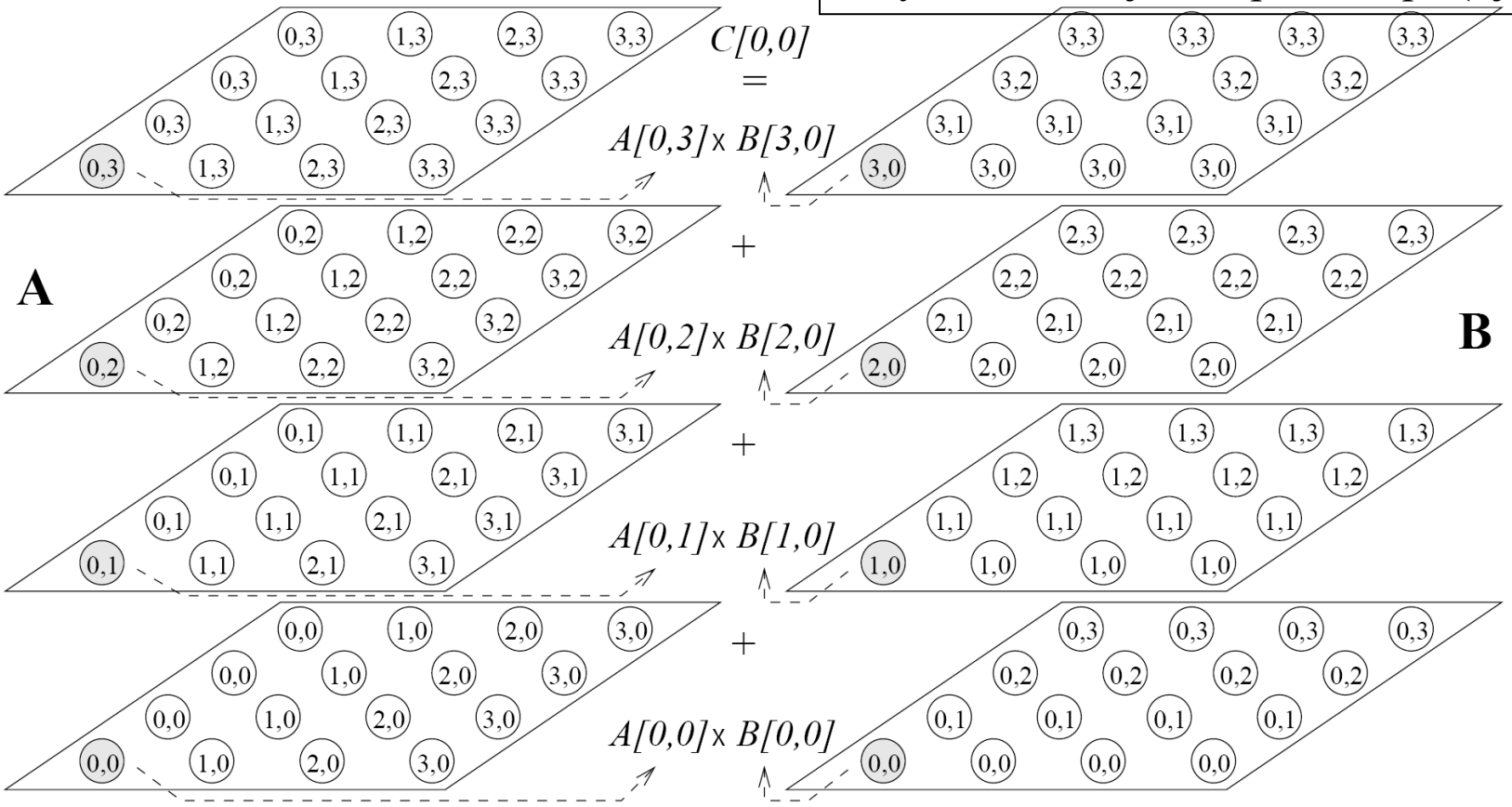


(d) Corresponding distribution of *B*

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

Эффективность алгоритма DNS

Шаг 1:
- Переслать A_{ik} с процессора (i,k,0) на (i,k,k);
-Broadcast A_{ik}с процессора (i,k,k) на процессоры (i,j,k) for all j=0,1,…,K-1;
Шаг 2:
-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
Шаг 3:
-Выполнить локальное блочное умножение матриц

Шаг 4:

- Редукционная сумма C_{I,j } вдоль k

Тема

- Производные типы данных MPI

Производные типы данных MPI

Назначение:

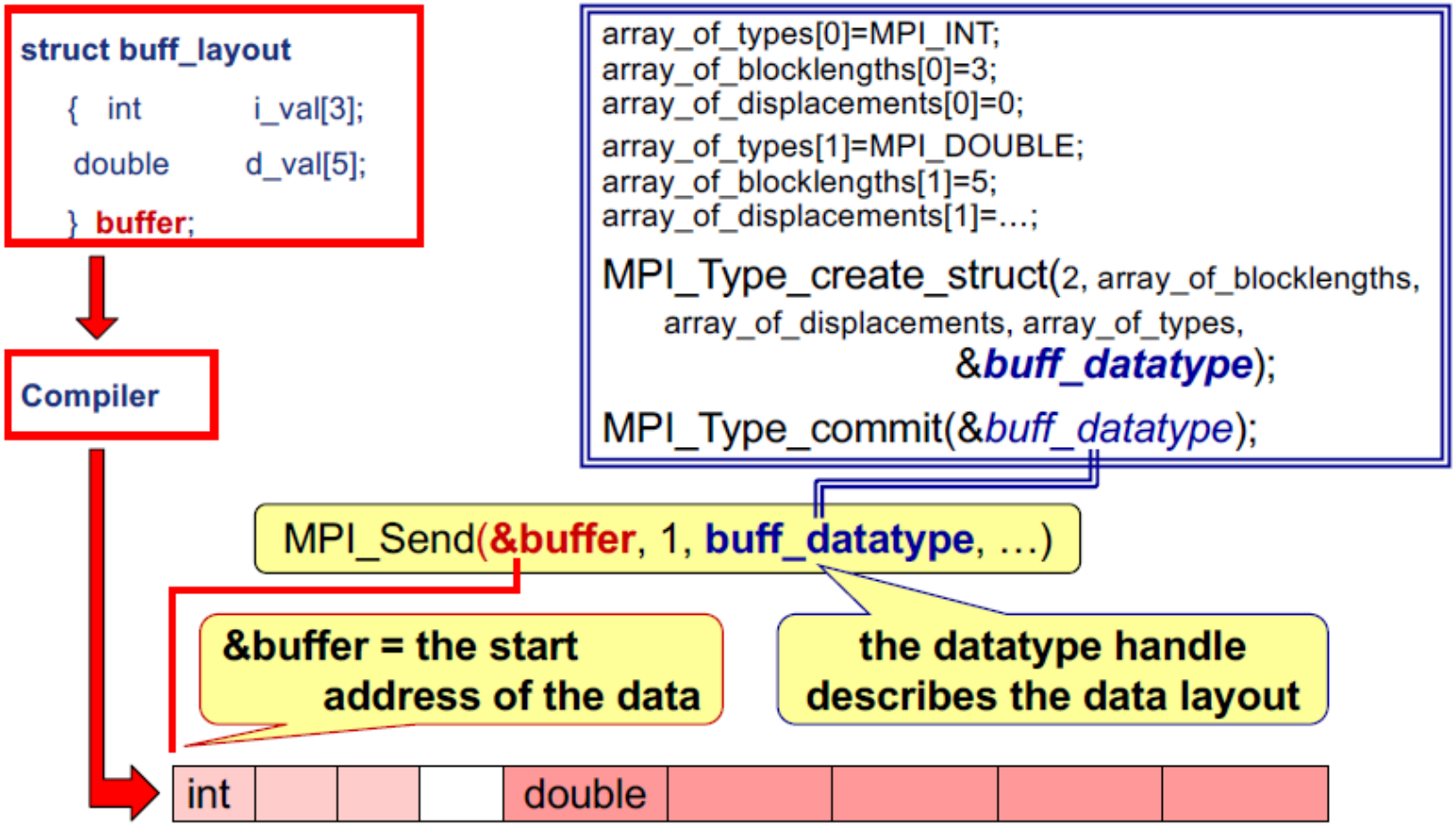
- пересылка данных, расположенных в несмежных областях памяти в одном сообщении
- пересылка разнотипных данных в одном сообщении
- облегчение понимания программы

Производные типы данных

- не могут использоваться ни в каких операциях, кроме коммуникационных
- производные ТД следует понимать как описатели расположения в памяти элементов базовых типов
- производный ТД - скрытый объект
- отображение типа:

Typemap = $\{(type_0, disp_0), \ldots , (type_{n-1}, disp_{n-1})\}$

Пример использования производных типов данных MPI



Типы данных MPI

- Элементарные типы - базовые типы, определяемые языком (например, MPI_INT и т.д., в том числе тип MPI_PACKED)
- Векторные (vector) – однотипные данные, расположенные в памяти с заданным шагом. Единица отсчета – тип данных элементов вектора
- Непрерывные типы (contiguous) – вектор с шагом 1
- Hvector – вектор с шагом, задаваемым в байтах
- Indexed – вектор, расположение элементов которого задается массивом. Единица отсчета - тип данных
- Hindexed – аналогично indexed, но шаги задаются в байтах
- Структурный тип - наиболее общий тип, соответствует типу в С

СЦЕНАРИЙ РАБОТЫ С ПРОИЗВОДНЫМИ ТИПАМИ

- Создание типа с помощью конструктора.
- Регистрация.
- Использование.
- Освобождение памяти.

Стандартный сценарий создания и использования производного ТД

- Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов:

  MPI_Type_contiguous,

  MPI_Type_vector, MPI_Type_create_hvector,

  MPI_Type_indexed, MPI_Type_create_hindexed,

  MPI_Type_create_struct

  MPI_Type_create_subarray

  MPI_Type_create_darray
- Новый производный тип регистрируется вызовом функции MPI_Type_commit
- Когда производный тип становится ненужным, он уничтожается функцией MPI_Type_free

Производные типы MPI. Карта типа.

Typemap = {(type$_0$, disp$_0$), … , (type$_{n-1}$, disp$_{n-1}$)}

- отображение типа вместе с базовым адресом начала расположения данных buf определяет коммуникационный буфер обмена (состоит из n элементов)
- i-й элемент имеет адрес buf+disp$_i$ и базовый тип type$_i$
- Базовый тип можно представить как производный тип:

  Например, *MPI_INT={(int,0)}*

Графическая иллюстрация карты памяти

Пример:
Typemap = {(int,0), (double,8), (char,16)}

Характеристики производного типа

- Дополнительные определения
  - *lower_bound(Typemap)* = min $disp_j$, j = 0, …, n-1
  - *upper_bound(Typemap)* = max($disp_j$ + sizeof($type_j$)) + ε
- Протяженность (кол-во байт, которое переменная данного типа занимает в памяти)
  - Функция MPI_Type_extent

    extent(Typemap) = upper_bound(Typemap) - lower_bound(Typemap)
- Размер (кол-во реально передаваемых байт в коммуникационных операциях)
  - Функция MPI_Type_size

MPI_Type_get_extent

int MPI_Type_get_extent (MPI_Datatype datatype,
MPI_Aint *lb, MPI_Aint *extent)

datatype - тип данных

extent - протяженность элемента заданного типа

MPY_Type_size

int MPI_Type_size (MPI_Datatype datatype, int *size)

datatype - тип данных

size - размер элемента заданного типа

Пример использования MPI_Get_address

```
……
int buf[10];
        MPI_Aint a1, a2;
        MPI_Init ( &argc, &argv );

        MPI_Get_address( &buf[0], &a1 );
        MPI_Get_address( &buf[1], &a2 );
        if ((int)(a2-a1) != sizeof(int)) {
            errs++;
            printf( "Get address of two address did not return values the
    correct distance apart\n" );fflush(stdout);
        }
```

MPI_Type_commit

- Каждый конструктор возвращает незарегистрированный (*uncommited)* тип. Процесс commit можно представить себе как процесс компиляции типа во внутреннее представление.
- Для регистрации типа должен вызываться *MPI_Type_commit* (&datatype).
- После регистрации тип может повторно использоваться.
- Повторный вызов commit не имеет эффекта.

MPI_Type_commit

int MPI_Type_commit(MPI_Datatype *datatype)

datatype - новый производный тип данных

Конструктор непрерывного типа MPI_Type_contiguous

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
MPI_Datatype *newtype)

count - число элементов базового типа

oldtype - базовый тип данных

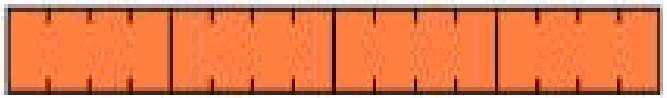
newtype - новый производный тип данных

Графическая иллюстрация типа MPI_Type_contiguous

oldtype = MPI_REAL

count = 4

newtype



MPI_Type_contiguous. Пример (1)

```
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

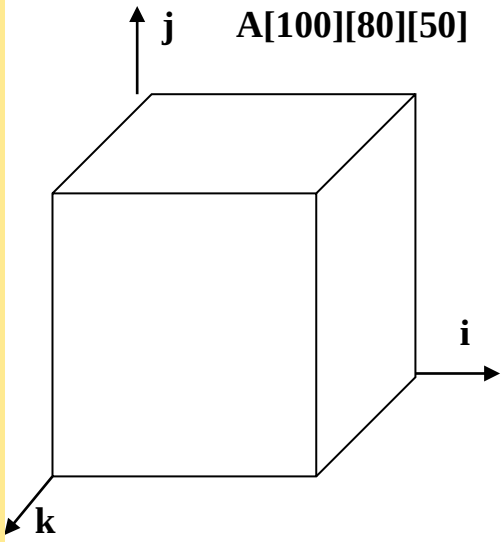
a[4][4]

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

MPI_Type_contiguous. Пример (2)

Surface: A[0][:][:]

```
MPI_Datatype face_jk;
MPI_Type_contiguous(80*50, MPI_DOUBLE,
&face_jk);
MPI_Type_commit(&face_jk);
MPI_Send(&A[0][0][0],1,face_jk,rank,tag,comm);
//
MPI_Send(&A[0][0][0],80*50,MPI_DOUBLE,rank,tag,
comm);
MPI_Send(&A[99][0][0],1,face_jk,rank,tag,comm);
...
MPI_Type_free(&face_jk);
```



Конструктор векторного типа MPI_Type_vector

oldtype = MPI_REAL 

count = 3, blocklenght = 2, stride = 3

newtype

MPI_Type_vector

```
int MPI_Type_vector(int count, int blocklength, int stride,
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

count **-** число блоков

blocklength **-** число элементов базового типа в каждом блоке

stride - шаг между началами соседних блоков, измеренный числом элементов базового типа

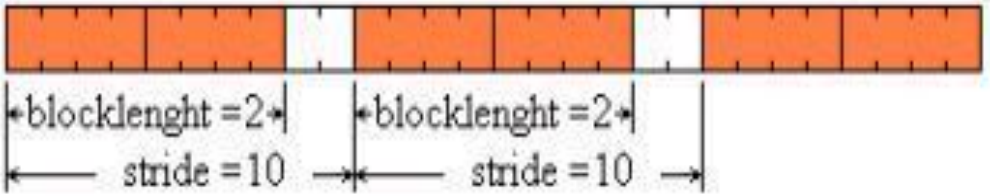
oldtype **-** базовый тип данных

newtype **-** новый производный тип данных

MPI_Type_hvector

oldtype = MPI_REAL 

count = 3, blocklenght = 2, stride = 10

newtype 

MPI_Type_create_hvector

int MPI_Type_create_hvector (int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

count - число блоков
blocklength - число элементов базового типа в каждом блоке
stride - шаг между началами соседних блоков в байтах
oldtype - базовый тип данных
newtype - новый производный тип данных.

Отличие от MPI_Type_vector: stride определяется в байтах, не в элементах (‘h’ – heterogeneous)

MPI_Type_create_hvector. Пример 1

```
double A[4][4];
MPI_Datatype column;

MPI_Type_create_hvector(4,1,4*sizeof(double),
                MPI_DOUBLE, &column);
MPI_Type_commit(&column);
MPI_Send(&A[0][1],1,column,rank,tag,comm);
...
```

Число блоков

Размер блока

Шаг

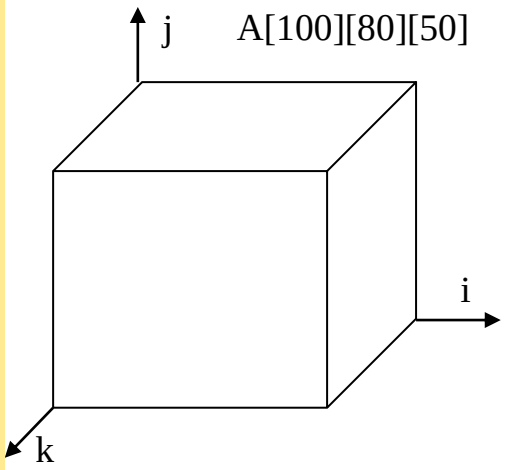
1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

MPI_Type_hvector. Пример 2

Плоскость ij: A[:][:][49]

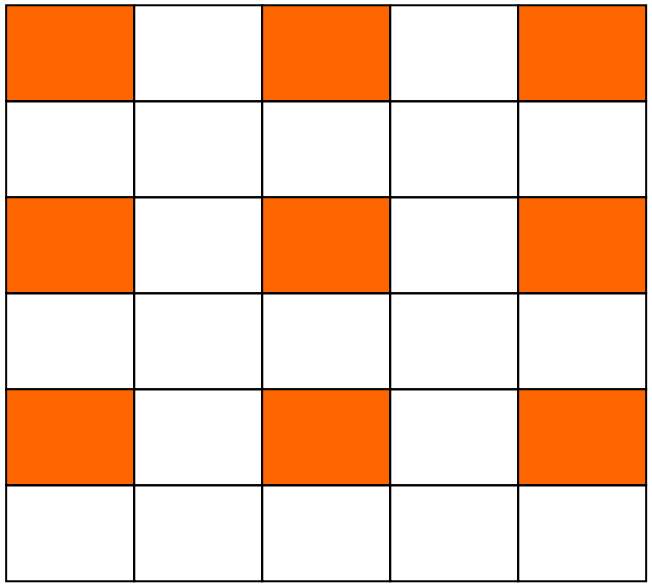
```
double A[100][80][50];
MPI_Datatype face_ij, line_j;

MPI_Type_vector(80,1,50,MPI_DOUBLE,&line_j);
MPI_Type_create_hvector(100,1,80*50*sizeof(dou
ble),
                line_j, &face_ij);
MPI_Type_commit(&face_ij);
MPI_Send(&A[0][0][49],1,face_ij,rank,tag,comm)
;
...
```



MPI_Type_create_hvector. Пример 3 – рассылка клеточной структуры

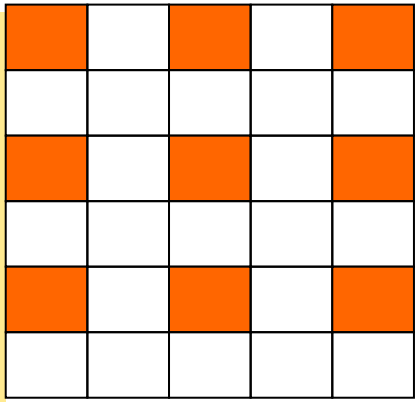
Используются вместе MPI_type_vector и MPI_Type_create_hvector для рассылки отмеченных клеток:



Пример 3, продолжение

```
double a[6][5], e[3][3];
MPI_Datatype oneslice, twoslice
MPI_Aint  lb, sz_dbl
int myrank, ierr

MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
MPI_Type_get_extent (MPI_DOUBLE, &lb, &sz_dbl);
MPI_Type_vector (3,1,2,MPI_DOUBLE, &oneslice);
MPI_Type_create_hvector (3,1,12*sz_dbl, oneslice,
&twoslice);
MPI_Type_commit (&twoslice);
```



count

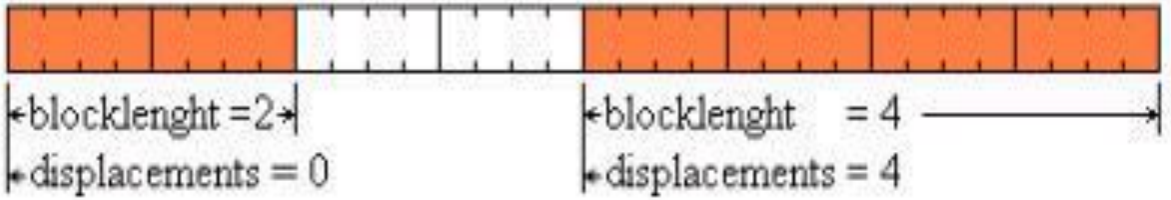
blocklength

stride

MPI_Type_indexed

oldtype = MPI_REAL 

count = 2 blocklenght = (2, 4) displacements = (0,4)

newtype 

MPI_Type_indexed

int MPI_Type_indexed(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

***count* -** число блоков

***array_of_blocklengths* -** массив, содержащий число элементов базового типа в каждом блоке

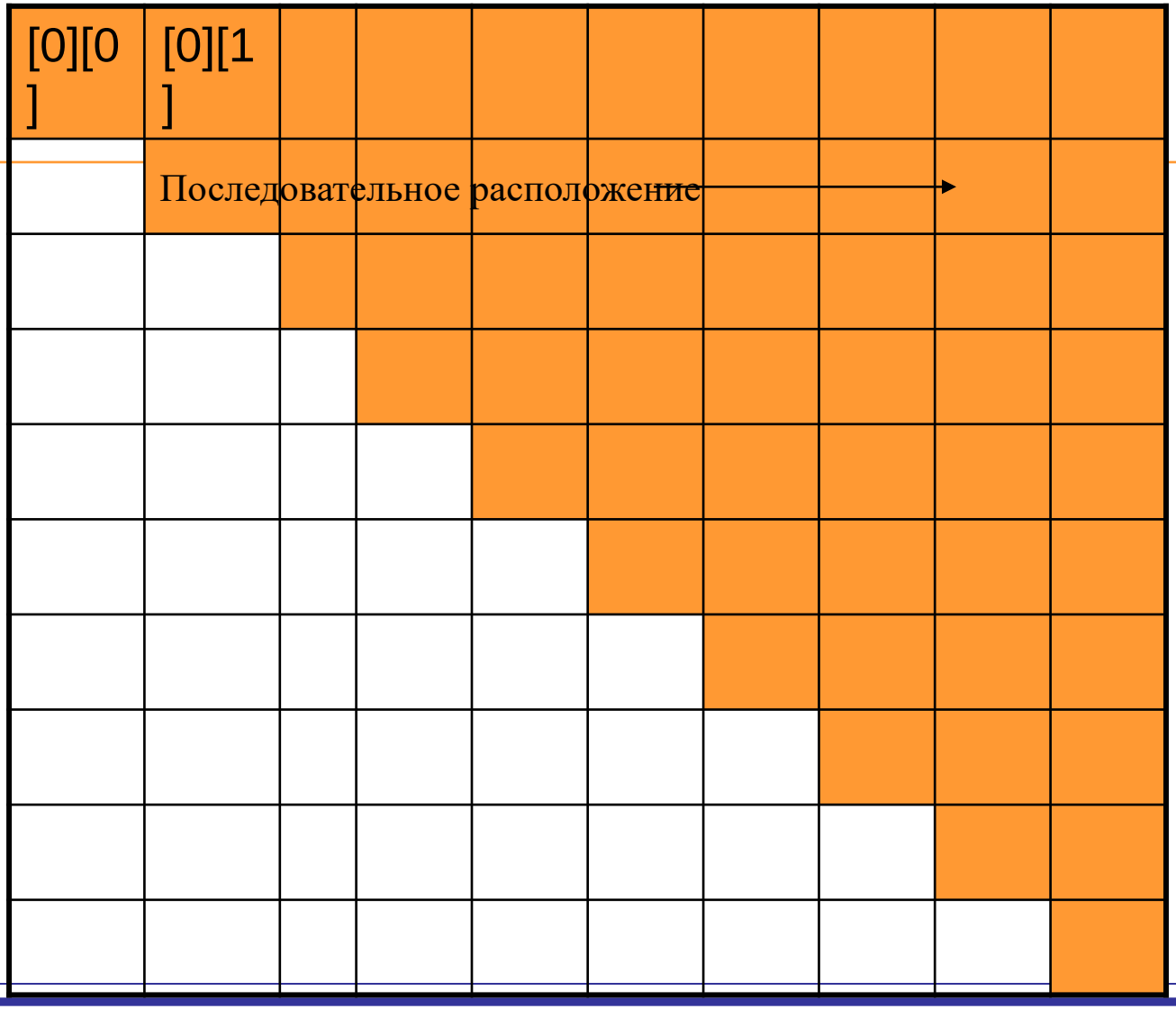
array_of_displacements - массив смещений каждого блока от начала размещения элемента нового типа, смещения измеряются числом элементов базового типа

oldtype - базовый тип данных

newtype - новый производный тип данных

Смещения между последовательными блоками не обязательно должны совпадать. Это позволяет выполнять пересылку данных одним вызовом.

Пример: верхнетреугольная матрица



Пересылка верхнетреугольной матрицы

```
double a[100][100];
int disp[100], blocklen[100], i, dest, tag;
MPI_Datatype upper;
/* compute start and size of each row */
for (i = 0; i < 100; ++i){
  disp[i] = 100*i + i;
  blocklen[i] = 100 – i;
}
MPI_Type_indexed(100, blocklen, disp, MPI_DOUBLE, &upper);
MPI_Type_commit(&upper);
MPI_Send(a, 1, upper, dest, tag, MPI_COMM_WORLD);
MPI_Type_free (&upper);
```

MPI_Type_indexed_block

int MPI_Type_create_indexed_block (int count, int blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

count **-** число блоков

blocklengths **-** число элементов базового типа в каждом блоке

array_of_displacements - массив смещений каждого блока от начала размещения элемента нового типа, смещения измеряются числом элементов базового типа

oldtype - базовый тип данных

newtype - новый производный тип данных

Отличие от MPI_Type_indexed – все блоки одинаковой длины

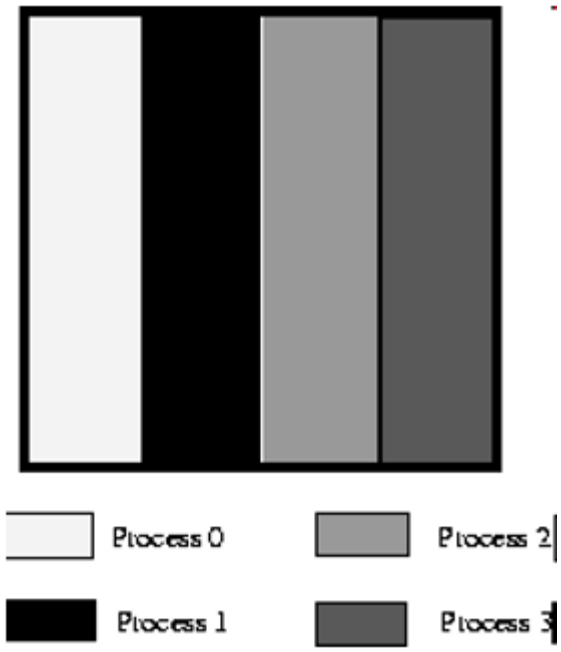
MPI_Type_create_subarray

int MPI_Type_create_subarray (int ndims, int *array_of_sizes,
int *array_of_subsizes, int *array_of_starts, int order,
MPI_Datatype oldtype, MPI_Datatype *newtype)

ndims – число измерений массива
array_of_sizes – количество элементов по измерениям глобального массива
array_of_subsizes – количество элементов пл измерениям подмассива
array_of_starts – координаты начала подмассива в глобальном массиве
order – порядок расположения элементов в массиве (C или Fortran)
MPI_Datatype oldtype, MPI_Datatype *newtype

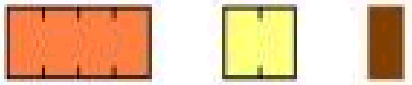
MPI_Type_create_subarray. Пример

```
double subarray[100][25];
MPI_Datatype filetype;
 int sizes[2], subsizes[2], starts[2]; int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];
MPI_Type_create_subarray(2, sizes, subsizes, starts)
MPI_ORDER_C, MPI_DOUBLE, &filetype);
MPI_Type_commit(&filetype);
```



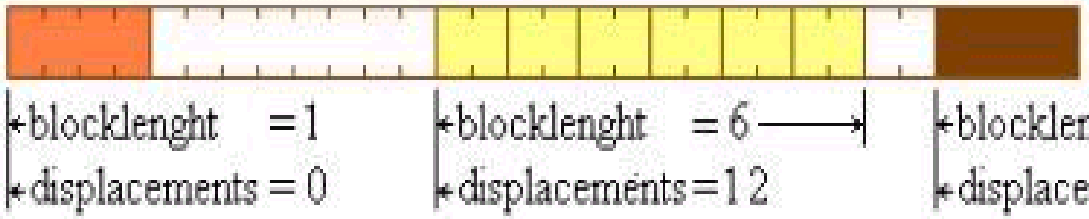
MPI_Type_create_struct

oldtypes = (MPI_INT, MPI_SHORT, MPI_CHAR)



count = 3 blocklenght = (1, 6, 4) displacements = (0, 12, 26)

newtype



MPI_Type_create_struct

int MPI_Type_create_struct (int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)

count **-** число блоков;

array_of_blocklength **-** массив, содержащий число элементов одного из базовых типов в каждом блоке;

array_of_displacements **-** массив смещений каждого блока от начала размещения структуры, смещения измеряются в байтах;

array_of_type **-** массив, содержащий тип элементов в каждом блоке;

newtype **-** новый производный тип данных**.**

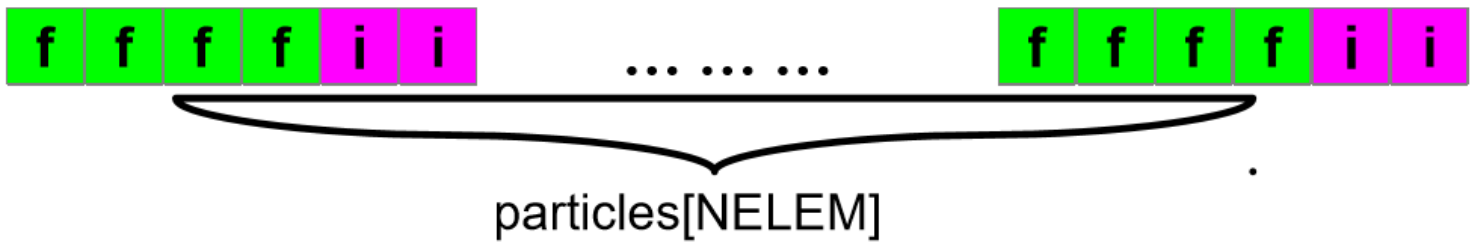
Наиболее общий конструктор.

```
struct {
   float x, y, z, velocity;
   int n, type;
} Particle;

Particle particles[NELEM];
```

```
MPI_Type_extent(MPI_FLOAT, &extent);

count = 2;
blockcounts[0] = 4;          blockcount[1] = 2;
oldtypes[0]= MPI_FLOAT;      oldtypes[1] = MPI_INT;
displ[0] = 0;                displ[1] = 4*extent;
```



MPI_Type_create_struct (count, blockcounts, displ, oldtypes, &particletype);
MPI_Type_commit(&particletype);

struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];

int count, blockcounts[2]; MPI_Aint displ[2]; MPI_Datatype particletype, oldtypes[2];

count = 2; blockcounts[0] = 4; blockcount[1] = 2; oldtypes[0]= MPI_FLOAT; oldtypes[1] = MPI_INT;

MPI_Type_extent (MPI_FLOAT, &extent); displ[0] = 0; displ[1] = 4*extent;

MPI_Type_create_struct (count, blockcounts, displ, oldtypes, &particletype);

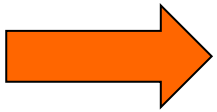
MPI_Type_commit(&particletype);

MPI_Send (particles, NELEM, particletype, dest, tag, MPI_COMM_WORLD);

MPI_Free(&particletype);

Проблемы выравнивания элементов структур

```
struct mystruct
{ char a;
  int b;
   char c;
} x
```



```
struct mystruct
 { char a;
   char gap_0[3];
   int b; char c;
    char gap_1[3]; } x
```

```
struct PartStruct {
char class;
double d[6];
 int b[7]; } particle[100];
```

```
MPI_Datatype ParticleType; int count = 3;
MPI_Datatype type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_INT};
int blocklen[3] = {1, 6, 7};
MPI_Aint disp[3];
MPI_Get_address(&particle[0].class, &disp[0]);
MPI_Get_address(&particle[0].d, &disp[1]);
MPI_Get_address(&particle[0].b, &disp[2]);
 /* Make displacements relative */
 disp[2] -= disp[0]; disp[1] -= disp[0]; disp[0] = 0;
MPI_Type_create_struct (count, blocklen, disp, type, &ParticleType);
    MPI_Type_commit (&ParticleType);
MPI_Send(particle,100,ParticleType,dest,tag,comm); MPI_Type_free
    (&ParticleType);
```

int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype * newtype)

Возможные проблемы при выравнивании. Последний член структуры

int MPI_Type_create_resized (MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype * newtype)

Создается новый тип, идентичный старому за исключением того, что новый lb = lb + extent

Пример использования MPI_Type_create_resized

```
/* Sending an array of structs portably */
struct PartStruct particle[100];
MPI_Datatype ParticleType;
 …
/* check that the extent is correct */
 MPI_Type_get_extent(ParticleType, &lb, &extent);
if ( extent != sizeof(particle[0]) )
 {
    MPI_Datatype old = ParticleType;
    MPI_Type_create_resized ( old, 0, sizeof(particle[0]), &ParticleType);
    MPI_Type_free(&old);
  }
 MPI_Type_commit ( &ParticleType);
```

Упаковка данных

- Упаковка / распаковка:
  - MPI_Pack
  - MPI_Unpack
- Определение размера буфера для упаковки:
  - MPI_Pack_size

MPI_Pack

int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)

inbuf - адрес начала области памяти с элементами, которые требуется упаковать;
incount - число упаковываемых элементов;
datatype - тип упаковываемых элементов;
outbuf - адрес начала выходного буфера для упакованных данных;

outsize - размер выходного буфера в байтах;
position - текущая позиция в выходном буфере в байтах;
comm - коммуникатор.

MPI_Unpack

int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
int outcount, MPI_Datatype datatype, MPI_Comm comm)

inbuf - адрес начала входного буфера с упакованными данными;
insize - размер входного буфера в байтах;
position - текущая позиция во входном буфере в байтах;
outbuf - адрес начала области памяти для размещения распакованных элементов;
outcount - число извлекаемых элементов;
datatype - тип извлекаемых элементов;
comm - коммуникатор.

Пересылка элементов разного типа

- элементы нужно предварительно запаковать в один массив, последовательно обращаясь к функции упаковки MPI_Pack
- при первом вызове функции упаковки параметр position, как правило, устанавливается в 0, чтобы упакованное представление размещалось с начала буфера
- для непрерывного заполнения буфера необходимо в каждом последующем вызове использовать значение параметра position, полученное из предыдущего вызова
- распаковывать - аналогично

MPI_Pack_size

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)

incount - число элементов, подлежащих упаковке

datatype - тип элементов, подлежащих упаковке

comm - коммуникатор

size - размер сообщения в байтах после его упаковки

Пример рассылки разнотипных данных с использованием функций MPI_Pack и MPI_Unpack

```
char buff[100];
double x, y;
int position, a[2];
{
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{  /* Упаковка данных*/
position = 0;
MPI_Pack(&x, 1, MPI_DOUBLE, buff, 100, &position, MPI_COMM_WORLD);
MPI_Pack(&y, 1, MPI_DOUBLE, buff, 100, &position, MPI_COMM_WORLD);
MPI_Pack(a, 2, MPI_INT, buff, 100, &position, MPI_COMM_WORLD);
}
   /* Рассылка упакованного сообщения */
MPI_Bcast(buff, position, MPI_PACKED, 0, MPI_COMM_WORLD);
   /* Распаковка сообщения во всех процессах */
if (myrank != 0)
position = 0;
MPI_Unpack(buff, 100, &position, &x, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buff, 100, &position, &y, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buff, 100, &position, a,  2, MPI_INT, MPI_COMM_WORLD);
}
```