

Параллельное программирование для высокопроизводительных вычислительных систем

сентябрь – декабрь 2021 г.

Лектор доцент Н.Н.Попова

Лекция 13

29 ноября 2021 г.

9 декабря 2021 г. (продолжение)

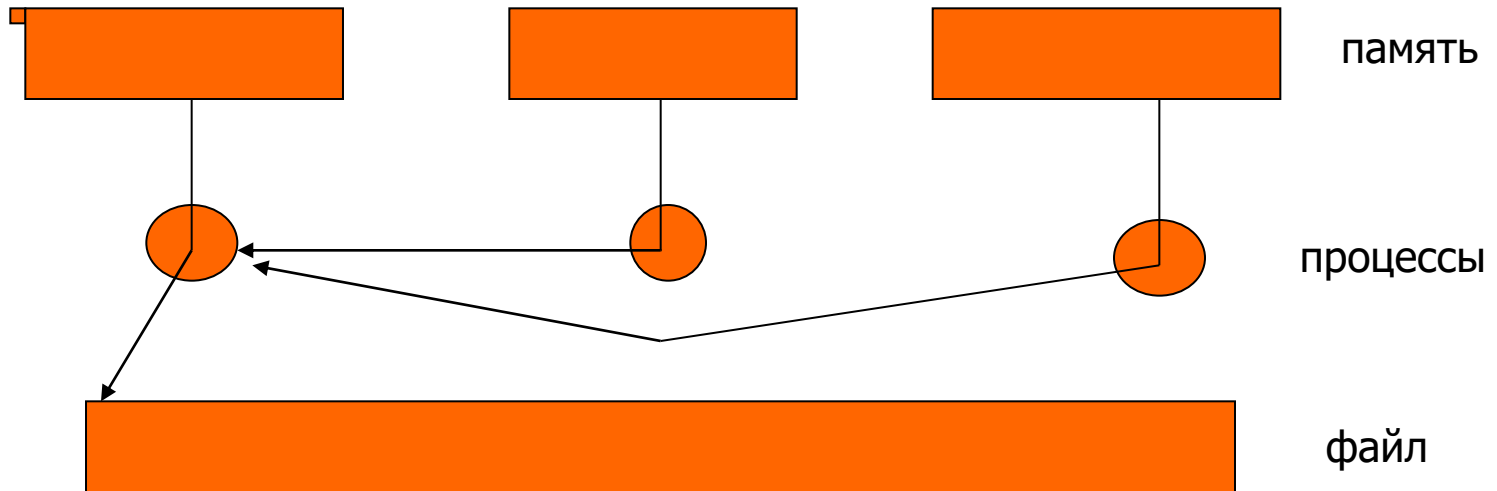
Тема

- Параллельная работа с файлами

Основные возможности

- Произвольный доступ к файлам
- Коллективные операции ввода/вывода
- Индивидуальные и разделяемые файловые указатели
- Неблокирующий I/O
- Переносимое представление данных

MPI-2 I/O: непараллельный I/O



Непараллельный I/O из MPI прогр. (2)

```
if (myrank != 0)
    MPI_Send(buf, BUFSIZE, MPI_INT, 0, 99, MPI_COMM_WORLD);
else {
    myfile = fopen("testfile", "w");
    fwrite(buf, sizeof(int), BUFSIZE, myfile);
    for (i=1; i<numprocs; i++) {
        MPI_Recv(buf, BUFSIZE, MPI_INT, i, 99, MPI_COMM_WORLD,
                &status);
        fwrite(buf, sizeof(int), BUFSIZE, myfile);
    }
    fclose(myfile);
}
MPI_Finalize();
return 0;
}
```

Непараллельный I/O из MPI прогр. (3)

— *Когда имеет смысл:*

- I/O поддерживается только на определенном узле многопроцессорной системы
- В программе используется высокоуровневая библиотека, не поддерживающая параллельный ввод-вывод
- Результирующий файл должен обрабатываться последовательным ПО
- Возможное повышение эффективности за счет буферизации данных

Почему надо использовать параллельный ввод-вывод:

- Масштабируемость, эффективность при увеличении числа процессоров

MPI-2 I/O: не MPI параллельный I/O

/* не MPI параллельная запись в разные файлы */

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[]) {
    int i, myrank, buf[BUFSIZE];
    char filename[128];
    FILE *myfile;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    for (i=0; i<BUFSIZE; i++)
        buf[i] = myrank * BUFSIZE + i;
    sprintf(filename, "testfile.%d", myrank);
    myfile = fopen(filename, "w");
    fwrite(buf, sizeof(int), BUFSIZE, myfile);
    fclose(myfile);
    MPI_Finalize();
    return 0; }
```

Преимущества:

-параллельный доступ

Недостатки:

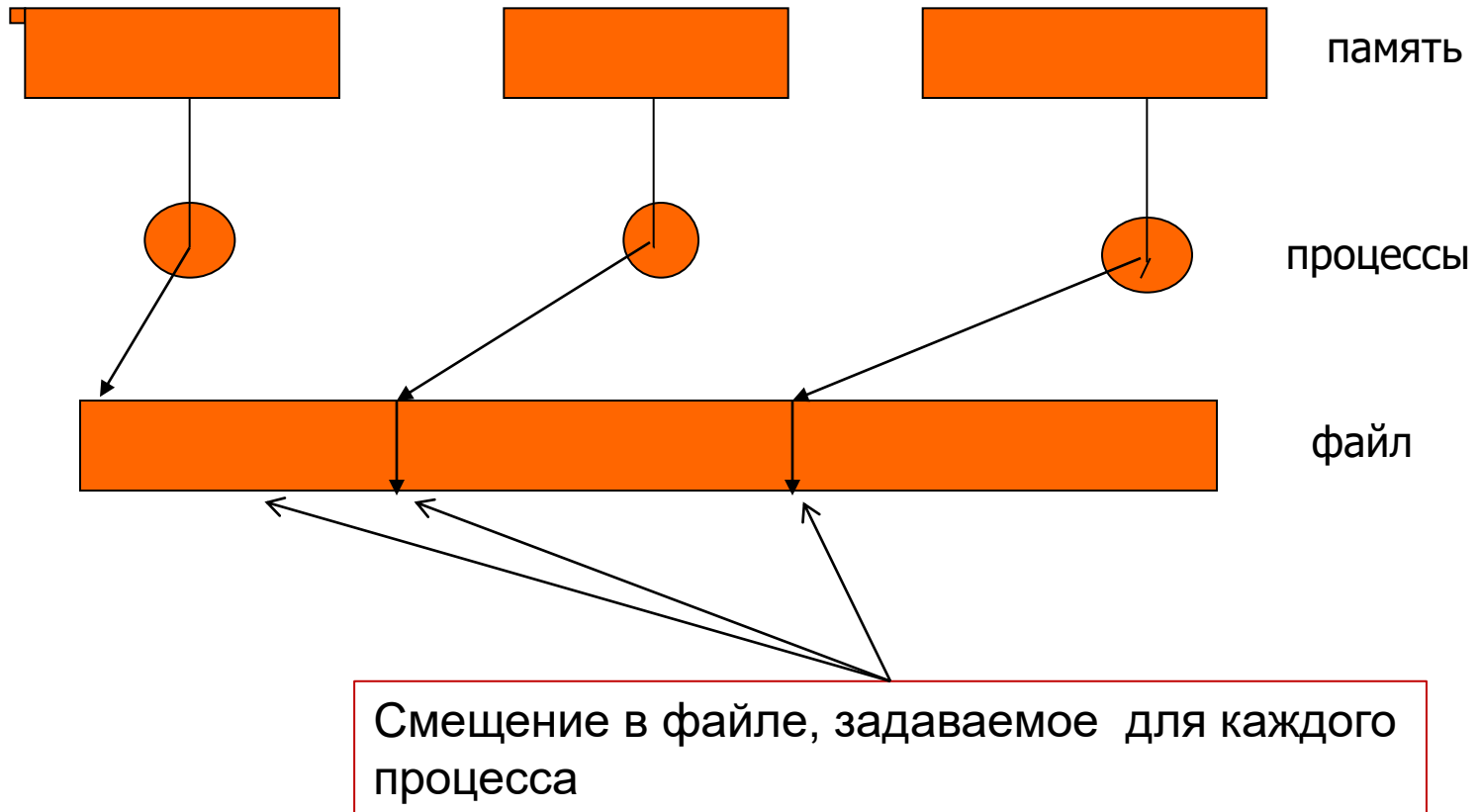
-много файлов

-использование
файлов при повторном
запуске (то же число
процессов)

MPI-2 I/O: MPI I/O в разные файлы

```
/* Параллельная MPI-запись в разные файлы */
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[]){
    int i, myrank, buf[BUFSIZE];
    char filename[128];
    MPI_File myfile;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    for (i=0; i<BUFSIZE; i++)
        buf[i] = myrank * BUFSIZE + i;
    sprintf(filename, "testfile.%d", myrank);
    MPI_File_open(MPI_COMM_SELF, filename,
                MPI_MODE_WRONLY | MPI_MODE_CREATE,
                MPI_INFO_NULL, &myfile);
    MPI_File_write(myfile, buf, BUFSIZE, MPI_INT,
                  MPI_STATUS_IGNORE);
    MPI_File_close(&myfile);
    MPI_Finalize(); return 0; }
```


Параллельный I/O в один файл



Параллельный MPI I/O: запись в один файл

/* параллельный MPI вывод в файл */

```
#include "mpi.h"
```

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

```
#define BUFSIZE 100
```

```
int main(int argc, char *argv){  
    int i, myrank, buf[BUFSIZE];  
    MPI_File thefile;
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
    for (i=0; i<BUFSIZE; i++)
```

```
        buf[i] = myrank * BUFSIZE + i;
```

```
    MPI_File_open(MPI_COMM_WORLD, "testfile",  
                  MPI_MODE_CREATE | MPI_MODE_WRONLY,  
                  MPI_INFO_NULL, &thefile);
```

```
    MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),  
                      MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
```

```
    MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,  
                   MPI_STATUS_IGNORE);
```

```
    MPI_File_close(&thefile);
```

```
    MPI_Finalize();
```

```
    return 0;
```

```
}
```

Data representation

Displacement

Тип: MPI_Offset

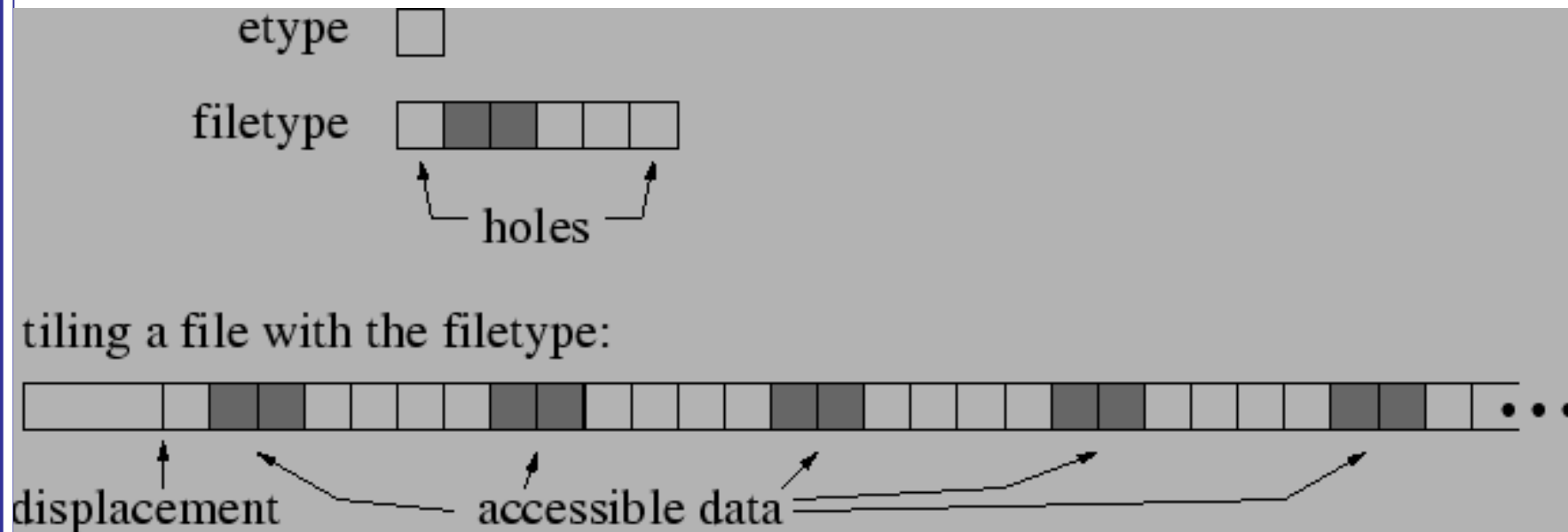
ftype

etype

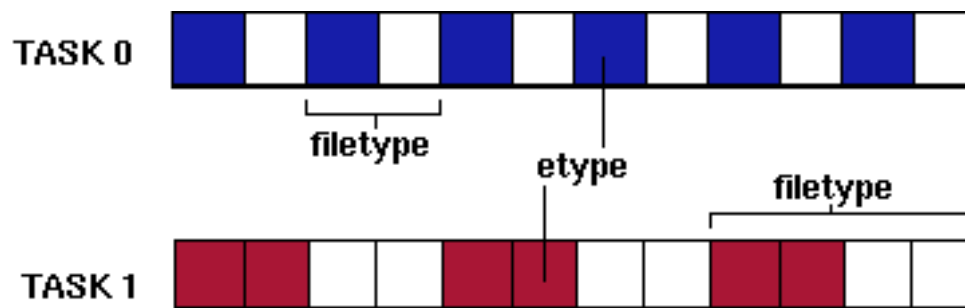
MPI-2 I/O: Терминология

- **Е-тип (элементарный тип данных)** - единица доступа к данным и позиционирования. Это может быть любой определенный в *MPI* базовый тип или производный тип данных.
- **Файловый тип** – базис для разбиения файла в среде процессов, определяет шаблон доступа к файлу.
Обычный е-тип или производный тип данных *MPI*, состоящий из нескольких элементов одного и того же е-типа.
Ограничения: смещения в производном типе должны быть упорядочены по возрастанию! Например, производный тип со смещениями: {2, 6, 5, 7, 4} нельзя использовать для построения файлового типа

Е-типы и Файловые типы



Файловые виды и структура файла



File structure



MPI-2 I/O: Терминология

- **View** (file view)- набор данных, видимый и доступный из открытого файла как упорядоченный набор e-типов. Каждый процесс имеет свой вид файла, определенный тремя параметрами: смещением, e-типом и файловым типом. Шаблон, описанный в файловом типе, повторяется, начиная со смещения.
- **Смещение** - это позиция в файле относительно текущего вида, представленная как число e-типов. ``Дыры" в файловом типе вида пропускаются при подсчете номера этой позиции. Нулевое смещение - это позиция первого видимого e-типа в виде (после пропуска смещения и начальных ``дыр" в виде).

MPI-2 I/O: Терминология

- **Размер MPI файла** измеряется в байтах от начала файла
- **Конец файла** - это смещение первого e-типа, доступного в данном виде, начинающегося после последнего байта в файле
- **Индивидуальные файловые указатели** - файловые указатели, локальные для каждого процесса, открытого файл.
- **Общие файловые указатели** - файловые указатели, которые используются одновременно группой процессов, открывающих файл.
- **Дескриптор файла** - это закрытый объект, создаваемый MPI_FILE_OPEN и уничтожаемый MPI_FILE_CLOSE. Все операции над открытым файлом работают с файлом через его дескриптор

MPI-2 I/O: Базовый алгоритм работы

- Определение необходимых переменных и типов данных
- Открытие файла (MPI_File_open)
- Установка вида файла (MPI_File_set_view)
- Запись/чтение (MPI_File_write, MPI_File_read)
- Для неблокирующих операций, ожидание их завершения (напр., MPI_Wait)
- Закрывание файла (MPI_File_close)

Базовый алгоритм для работы с файлами

```
MPI_File fh;
MPI_Datatype filetype;
MPI_Status status;
MPI_Offset offset;
int mode;
float data[100];
/* other code */
/* set offset and filetype */
mode = MPI_MODE_CREATE | MPI_MODE_RDWR;
MPI_File_open(MPI_COMM_WORLD, "myfile", mode, MPI_INFO_NULL,
    &fh);
MPI_File_set_view(fh, offset, MPI_FLOAT, filetype, "native",
    MPI_INFO_NULL);
MPI_File_write(fh, data, 100, MPI_FLOAT, &status);
MPI_File_close(&fh);
```

Открытие файла

- `int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)`
 - *comm* коммуникатор (дескриптор)
 - *filename* имя открываемого файла (строка)
 - *amode* тип доступа к файлу (целое)
 - *info* информационный объект (дескриптор)
 - *fh* новый дескриптор файла (дескриптор)
- открывает файл с именем *filename* для всех процессов из группы коммуникатора *comm*
- все процессы должны обеспечивать одинаковое значение *amode* и имена файлов, указывающие на один и тот же файл
- *info* используется как «подсказка» (шаблоны доступа)

Типы доступа

- `MPI_MODE_RDONLY` -- только чтение,
- `MPI_MODE_RDWR` -- чтение и запись,
- `MPI_MODE_WRONLY` -- только запись,
- `MPI_MODE_CREATE` -- создавать файл, если он не существует,
- `MPI_MODE_EXCL` -- ошибка, если создаваемый файл уже существует,
- `MPI_MODE_DELETE_ON_CLOSE` -- удалять файл при закрытии,
- `MPI_MODE_UNIQUE_OPEN` -- файл не будет параллельно открыт где-либо еще,
- `MPI_MODE_SEQUENTIAL` -- файл будет доступен лишь последовательно,
- `MPI_MODE_APPEND` -- установить начальную позицию всех файловых указателей на конец файла.

Заккрытие файла

- `int MPI_File_close(MPI_File *fh)`
 - *fh* дескриптор файла (дескриптор)
- сначала синхронизирует состояние файла затем закрывает файл, ассоциированный с *fh*
- пользователь должен обеспечить условие, чтобы все ожидающие обработки неблокирующие запросы и разделенные коллективные операции над *fh*, производимые процессом, были выполнены до вызова `MPI_FILE_CLOSE`

Установка индивидуального указателя файла

`int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)`

- *fh* дескриптор файла (дескриптор)
- *offset* – смещение о начала файла (всегда в БАЙТАХ!)
- *whence*:
 - MPI_SEEK_SET – указатель файла устанавливается на *offset*
 - MPI_SEEK_CUR: указатель файла на текущую позицию плюс *offset*
 - MPI_SEEK_END: указатель файла на конец файла плюс *offset*

По умолчанию при открытии файла:

offset=0, всегда измеряется в *etype*,

etype и *filetype* = MPI_BYTE

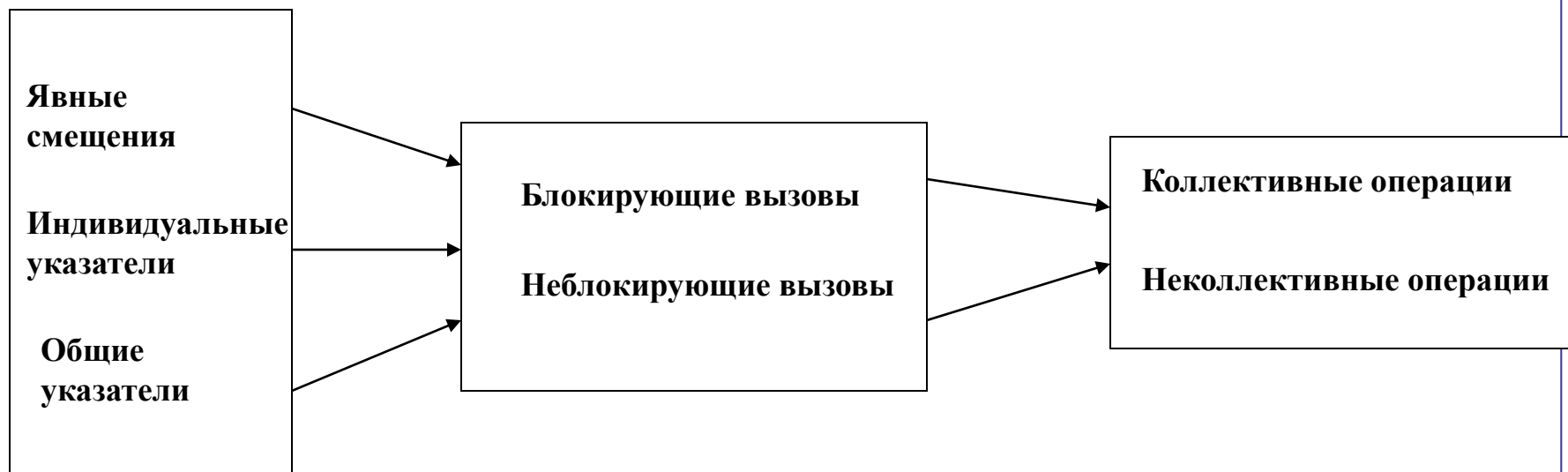
Файловые виды (1)

- `int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info)`
 - *fh* дескриптор файла (дескриптор)
 - *disp* смещение (целое)
 - *etype* элементарный тип данных (дескриптор)
 - *filetype* тип файла (дескриптор)
 - *datarep* представление данных (строка)
 - *info* информационный объект (дескриптор)

Файловые виды (2)

- `int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep)`
 - *fh* дескриптор файла (дескриптор)
 - *disp* смещение (целое)
 - *etype* элементарный тип данных (дескриптор)
 - *filetype* тип файла (дескриптор)
 - *datarep* представление данных (строка)

Доступ к данным



Позиционирование	Синхронизация	Координация	
		неколлективные	коллективные
Явные смещения	блокирующие	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	неблокирующие и расщепленные коллективные	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
Индивидуальные указатели	блокирующие	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	неблокирующие и расщепленные коллективные	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
Общие указатели	блокирующие	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	неблокирующие и расщепленные коллективные	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

Доступ к данным

- `int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
 - *fh* - дескриптор файла (дескриптор)
 - *buf* - начальный адрес буфера (выбор)
 - *count* - количество элементов в буфере (целое)
 - *datatype* - тип данных каждого элемента буфера (дескриптор)
 - *status* - объект состояния (Status)
- `int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
 - *fh* - дескриптор файла (дескриптор)
 - *buf* - начальный адрес буфера (выбор)
 - *count* - количество элементов в буфере (целое)
 - *datatype* - тип данных каждого элемента буфера (дескриптор)
 - *status* - объект состояния (Status)

Доступ к данным

- `int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
 - INOUT *fh* - дескриптор файла (дескриптор)
 - OUT *buf* - начальный адрес буфера (выбор)
 - IN *count* - количество элементов в буфере (целое)
 - IN *datatype* - тип данных каждого элемента буфера (дескриптор)
 - OUT *status* - объект состояния (Status)

- `int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
 - INOUT *fh* - дескриптор файла (дескриптор)
 - IN *buf* - начальный адрес буфера (выбор)
 - IN *count* - количество элементов в буфере (целое)
 - IN *datatype* - тип данных каждого элемента буфера (дескриптор)
 - OUT *status* - объект состояния (Status)

Коллективные операции I/O

- Используя коллективные функции I/O, пользователь полагается на оптимизацию их выполнения
- Реализация коллективных функций I/O может быть выполнена с условием объединения
- Рекомендуется использовать в случае, когда доступ к файлу из разных процессов производится в произвольном порядке и может пересекаться по времени

Коллективные операции I/O MPI-2

- `MPI_File_read_all`, `MPI_File_read_at_all`, etc
- `_all` означает, что все процессы, входящие в группу, заданную коммуникатором при открытии файла, должны вызвать эту функцию
- Каждый процесс определяет свою собственную информацию для выполнения этой функции -- список параметров такой же, как и для неколлективных операций

Пример: коллективные операции

```
/* noncontiguous access with a single collective I/O function */
```

```
#include "mpi.h"
```

```
#define FILESIZE 1048576
```

```
#define INTS_PER_BLK 16
```

```
int main(int argc, char **argv)
```

```
{
```

```
    int *buf, rank, nprocs, nints, bufsize;
```

```
    MPI_File fh;
```

```
    MPI_Datatype filetype;
```

```
    MPI_Init(&argc,&argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

```
    bufsize = FILESIZE/nprocs;
```

```
    buf = (int *) malloc(bufsize);
```

```
    nints = bufsize/sizeof(int);
```

```
    MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY,  
                  MPI_INFO_NULL, &fh);
```

Пример: коллективные операции

```
MPI_Type_vector(nints/INTS_PER_BLK, INTS_PER_BLK,
                INTS_PER_BLK*nprocs, MPI_INT, &filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh, INTS_PER_BLK*sizeof(int)*rank, MPI_INT,
                  filetype, "native", MPI_INFO_NULL);

MPI_File_read_all(fh, buf, nints, MPI_INT, MPI_STATUS_IGNORE);
MPI_File_close(&fh);

MPI_Type_free(&filetype);
free(buf);
MPI_Finalize();
return 0;
}
```

Неблокирующие коллективные операции I/O

- Ограниченная форма неблокирующих коллективных операций I/O
- Только **ОДНА** активная коллективная неблокирующая операция может выполняться в данный момент времени над заданным файловым указателем
- Не требуется request

```
MPI_File_write_all_begin(fh, buf, count, datatype);
```

```
for (i=0; i<1000; i++) {  
    /* perform computation */  
}
```

```
MPI_File_write_all_end(fh, buf, &status);
```


Разделяемый файловый указатель

- Функции для работы с разделяемым указателем:
 - MPI_File_read_shared
 - MPI_File_write_shared
 - MPI_File_seek_shared
 - MPI_File_iread_shared
 - MPI_File_iwrite_shared

Операция начинается над текущим указателем в файле, значение которого меняется после выполнения каждой операции

Процесс может сдвинуть файловый указатель -
MPI_File_seek_shared

Все процессы должны определить одинаковый file view.

Разделяемые коллективные упорядоченные операции

- Коллективные (блокирующие) упорядоченные операции:

- MPI_File_read_ordered
- MPI_File_write_ordered

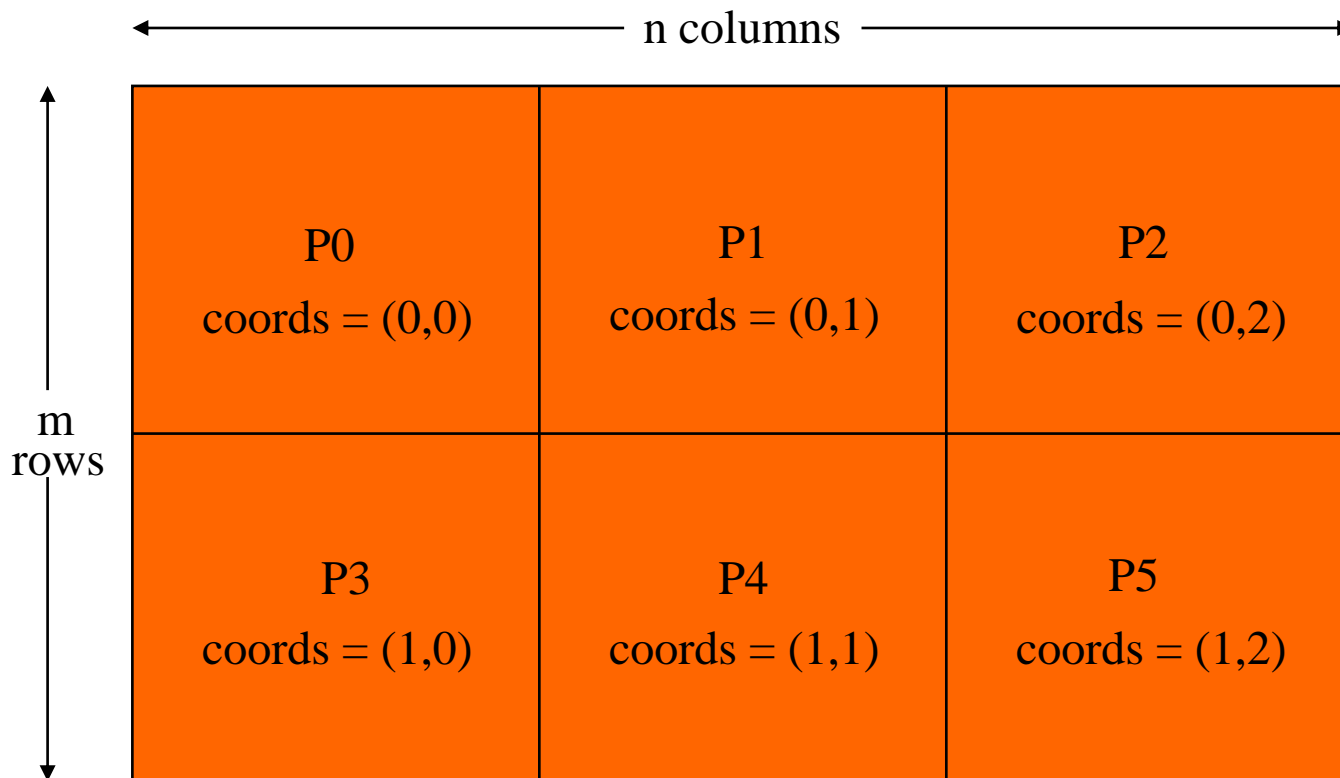
.....

- Разделяемые коллективные неблокирующие упорядоченные операции:

- MPI_File_read_ordered_begin
- MPI_File_read_ordered_end

.....

Доступ к массивам, хранящимся в файлах



$nproc(1) = 2, nproc(2) = 3$

Использование типа данных Subarray

```
gsizes[0] = m; /* no. of rows in global array */
gsizes[1] = n; /* no. of columns in global array*/
psizes[0] = 2; /* no. of procs. in vertical dimension */
psizes[1] = 3; /* no. of procs. in horizontal dimension */

lsizes[0] = m/psizes[0]; /* no. of rows in local array */
lsizes[1] = n/psizes[1]; /* no. of columns in local array */
dims[0] = 2; dims[1] = 3;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);
```

Subarray (продолжение)

```
/* global indices of first element of local array */
```

```
start_indices[0] = coords[0] * lsizes[0];
```

```
start_indices[1] = coords[1] * lsizes[1];
```

```
MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,  
    MPI_ORDER_C, MPI_FLOAT, &filetype);
```

```
MPI_Type_commit(&filetype);
```

```
MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",  
    MPI_MODE_CREATE | MPI_MODE_WRONLY,  
    MPI_INFO_NULL, &fh);
```

```
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",  
    MPI_INFO_NULL);
```

```
local_array_size = lsizes[0] * lsizes[1];
```

```
MPI_File_write_all(fh, local_array, local_array_size,  
    MPI_FLOAT, &status);
```

Обработка ошибок MPI-функций

Определяется константой ***MPI_SUCCESS***

Для обработки ошибок необходимо выполнить:

MPI_Comm_set_errhandler (MPI_COMM_WORLD, MPI_ERRORS_RETURN)

MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_FATAL)

```
int error;
```

```
.....
```

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
```

```
error = MPI_Init(&argc, &argv);
```

```
If (error != MPI_SUCCESS)
```

```
{
```

```
    fprintf (stderr, " MPI_Init error \n");
```

```
    return 1; /* exit(1); */
```

```
}
```

Обработка ошибок IO

- Возвращаемое значение всех функций `MPI_SUCCESS` в случае нормального завершения
- В случае ошибки возвращается код (целое число), зависящий от реализации
- Возвращаемое значение может быть приведено к «стандартному» значению функцией `MPI_Error_class`
- `MPI_Error_string` может быть использована для получения текстового сообщения

MPI_Error_class

- `int MPI_Error_class(int errorcode, int *errorclass);`

MPI_Error_string

`int MPI_Error_string(int errorcode, char *string, int *resultlen)`

- `errorcode` - код ошибки, возвращаемой функцией MPI
- `string` - текстовая строка, соответствующая коду ошибки `errorcode`
- `resultlen` - длина строки `string`

Память для **string** должна быть не меньше MPI_MAX_ERROR_STRING.
Реальная длина - **resultlen**.

Обработка ошибок IO: пример

```
errcode = MPI_File_open(MPI_COMM_WORLD,
    "/gpfs/data/user_login/file.txt", MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
if (errcode != MPI_SUCCESS) {
    MPI_Error_class(errcode, &errclass);
    if (errclass == MPI_ERR_NO_SUCH_FILE)
        printf ("File does not exist \n");
    else{
        MPI_Error_string(errcode, str, &len);
        printf ( "%s \n", str);
    }
}
```

Error classes MPI

MPI_ERR_FILE	Invalid file handle
MPI_ERR_NOT_SAME	Collective argument not identical on all processes, or collective routines called in a different order by different processes
MPI_ERR_AMODE	Error related to the amode passed to MPI_FILE_OPEN
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to MPI_FILE_SET_VIEW
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on a file which supports sequential access only
MPI_ERR_NO_SUCH_FILE	File does not exist
MPI_ERR_FILE_EXISTS	File exists
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
MPI_ERR_ACCESS	Permission denied
MPI_ERR_NO_SPACE	Not enough space
MPI_ERR_QUOTA	Quota exceeded
MPI_ERR_READ_ONLY	Read-only file or file system
MPI_ERR_FILE_IN_USE	File operation could not be completed, as the file is currently open by some process
MPI_ERR_DUP_DATAREP	Conversion functions could not be registered because a data representation identifier that was already defined was passed to MPI_REGISTER_DATAREP
MPI_ERR_CONVERSION	An error occurred in a user supplied data conversion function
MPI_ERR_IO	Other I/O error

Поддержка согласованности доступа к файлу

Семантика согласованности операций ввода/вывода определяет, каким должен быть результат в случае доступа нескольких процессов к файлу.

Простые случаи:

- доступ к файлу проходит только для чтения данных;
- каждый процесс имеет доступ к своему файлу. MPI гарантирует, что данные записанные процессом в файл, могут быть считаны процессом любое число раз.

Более сложные случаи:

- несколько процессов имеют доступ к общему файлу и, по крайней мере, один процесс записывает данные.

MPI гарантирует более сильную семантику для коммуникатора `MPI_COMM_WORLD` и более слабую для подкоммуникаторов.

Функции MPI для поддержки согласованности доступа к файлу

- `MPI_FILE_SET_ATOMICITY(fh, flag)`
- `MPI_FILE_GET_ATOMICITY(fh, flag)`
- `MPI_FILE_SYNC(fh)`

Поддержка согласованности MPI_File_sync

- `int MPI_File_Sync (MPI_File fh)`
 - `fh` Дескриптор файла (дескриптор)

Сброс всех буферов в файл.

Операция коллективная! Должна выполняться **ВСЕМИ** процессами, входящими в коммуникатор, указанный при открытии файла.

Поддержка согласованности. Атомарность.

- `int MPI_File_set_atomicity(MPI_File fh, int flag)` - все записи в файл немедленно записываются на диск; **коллективная операция**
 - `fh` дескриптор файла
 - `flag` - true для установки атомарного режима, false для отмены атомарного режима.
- `int MPI_File_get_atomicity(MPI_File fh, int *flag)` - возвращает текущее значение семантики согласованности для операций доступа к данным
 - `fh` дескриптор файла
 - `flag` - true при атомарном режиме, false при неатомарном режиме

Операция должна выполняться во **ВСЕХ** процессах коммутатора, указанного при открытии файла, параметр `flag` должен быть **ОДИНАКОВЫМ** у всех процессов.

Доступ к общему файлу, открытому с MPI_COMM_WORLD

- Самый простой случай: каждый процесс имеет доступ только к своей порции файла. В этом случае MPI гарантирует согласованность

Пример 1

- Файл открыт с использованием `MPI_COMM_WORLD`. Каждый процесс пишет в свою область в файле и читает то, что сам записал.

Process 0

```
MPI_File_open(MPI_COMM_WORLD,...)
MPI_File_write_at(off=0,cnt=100)
MPI_File_read_at(off=0,cnt=100)
```

Process 1

```
MPI_File_open(MPI_COMM_WORLD,...)
MPI_File_write_at(off=100,cnt=100)
MPI_File_read_at(off=100,cnt=100)
```

- MPI гарантирует, что данные будут считаны корректно.

Пример 2

- Аналогично примеру 1, за исключением того, что каждый процесс читает то, что записал другой (overlapping accesses).
- В этом случае MPI не гарантирует, что данные автоматически будут считаны верно.

Process 0

```
/* incorrect program */  
MPI_File_open(MPI_COMM_WORLD,...)  
MPI_File_write_at(off=0,cnt=100)  
MPI_Barrier  
MPI_File_read_at(off=100,cnt=100)
```

Process 1

```
/* incorrect program */  
MPI_File_open(MPI_COMM_WORLD,...)  
MPI_File_write_at(off=100,cnt=100)  
MPI_Barrier  
MPI_File_read_at(off=0,cnt=100)
```

Пример 2 (продолж.)

- Пользователь должен предпринять специальные действия для того, чтобы обеспечить корректность.
- 3 возможных варианта действий:
 - установка `atomicity` в `true`. При открытии файла `atomicity = false`.
 - Закрывать и снова открывать файл
 - Убедиться, что никакая записывающая последовательность в каком-либо процессе не пересекается во времени ни с какой –либо другой последовательностью (чтения или записи) в других последовательностях

Пример 2, вариант1

Установка atomicity в true

Process 0

```
MPI_File_open(MPI_COMM_WORLD,...)
MPI_File_set_atomicity(fh1,1)
MPI_File_write_at(off=0,cnt=100)
MPI_Barrier
MPI_File_read_at(off=100,cnt=100)
```

Process 1

```
MPI_File_open(MPI_COMM_WORLD,...)
MPI_File_set_atomicity(fh2,1)
MPI_File_write_at(off=100,cnt=100)
MPI_Barrier
MPI_File_read_at(off=0,cnt=100)
```

Пример 2, вариант 2. Закрывать и снова открывать файл.

Process 0

```
MPI_File_open(MPI_COMM_WORLD,...)
MPI_File_write_at(off=0,cnt=100)
MPI_File_close
MPI_Barrier
MPI_File_open(MPI_COMM_WORLD,...)
MPI_File_read_at(off=100,cnt=100)
```

Process 1

```
MPI_File_open(MPI_COMM_WORLD,...)
MPI_File_write_at(off=100,cnt=100)
MPI_File_close
MPI_Barrier
MPI_File_open(MPI_COMM_WORLD,...)
MPI_File_read_at(off=0,cnt=100)
```

Поддержка согласованности.

- Критерий: последовательность операций записи в любом процессе не пересекается с любой последовательностью операций чтения в других процессах.
- *Последовательность* – множество операций IO, заключенная в пару операций MPI_File_sync, MPI_File_open, MPI_File_close. Последовательность записи – если в последовательности есть операция записи.
- Пример: sync-write-read-sync, open-write-close, sync-read-read-sync.
- MPI гарантирует, что данные, записанные одним процессом могут быть считаны другим процессом, если последовательность записи в одном процессе не пересекается ни с одной другой последовательностью любого другого процесса.

Пример 2, вариант 3

Process 0

```
MPI_File_open(MPI_COMM_WORLD,...)
MPI_File_write_at(off=0,cnt=100)
MPI_File_sync

MPI_Barrier

MPI_File_sync /*collective*/

MPI_File_sync /*collective*/

MPI_Barrier

MPI_File_sync
MPI_File_read_at(off=100,cnt=100)
MPI_File_close
```

Process 1

```
MPI_File_open(MPI_COMM_WORLD,...)

MPI_File_sync /*collective*/

MPI_Barrier

MPI_File_sync
MPI_File_write_at(off=100,cnt=100)
MPI_File_sync

MPI_Barrier

MPI_File_sync /*collective*/
MPI_File_read_at(off=0,cnt=100)
MPI_File_close
```

Пример 3

- Аналогично примеру 2, за исключением того, что каждый процесс использует `MPI_COMM_SELF` при открытии общего файла
- Единственный путь – убедиться, что ни одна записывающая последовательность в любом из процессов не пересекается с читающей последовательностью в любом другом процессе.

Пример 3

Process 0

```
MPI_File_open(MPI_COMM_SELF,...)
MPI_File_write_at(off=0,cnt=100)
MPI_File_sync
```

```
MPI_Barrier
```

```
MPI_Barrier
```

```
MPI_File_sync
```

```
MPI_File_read_at(off=100,cnt=100)
```

```
MPI_File_close
```

Process 1

```
MPI_File_open(MPI_COMM_SELF,...)
```

```
MPI_Barrier
```

```
MPI_File_sync
```

```
MPI_File_write_at(off=100,cnt=100)
```

```
MPI_File_sync
```

```
MPI_Barrier
```

```
MPI_File_read_at(off=0,cnt=100)
```

```
MPI_File_close
```

Переносимость (interoperability)

Означает:

- MPI-файл может использоваться обычной файловой системой
- MPI-файл может переноситься с одной вычислительной системы на другую
- MPI-файл, записанный на одной системе, может быть прочитан на другой, используя различные способы представления данных

Переносимость

Обеспечивается:

- Использование `datatype` параметра функции `MPI_File_set_view`:
 - `native` – данные записываются в файл, как в памяти,
 - `internal` – форма представления зависит от реализации,
 - `external32` – 32-bit big-endian IEEE формат
- Использование производных MPI-типов

Дополнительные слайды

- Тема: Параллельная работа с файлами

Дополнительные типы данных MPI, используемые для работы с файлами

- Distributed Array (аналогично High Performance Fortran)
- Subarray
- INDEXED_BLOCK

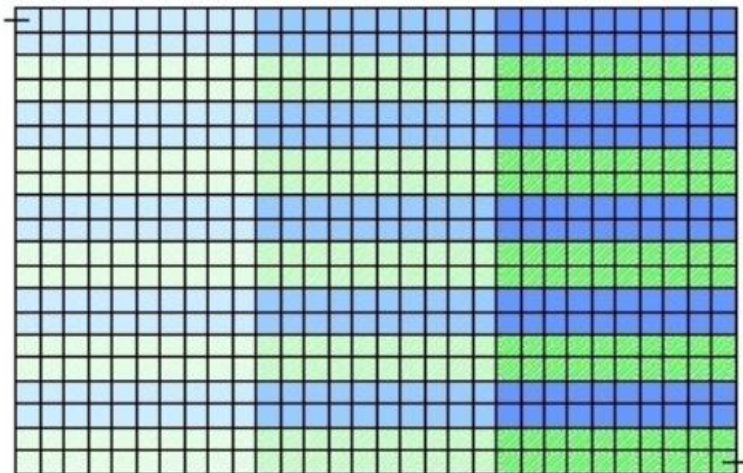
Пример задания параметров

Пример распределения данных на процессной решетке:

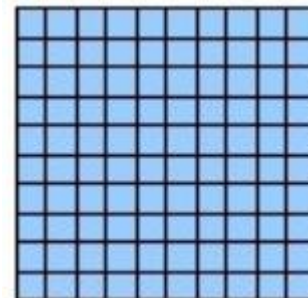
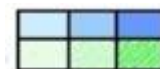
-CYCLIC(2), BLOCK:

- распределение с шагом 2 по первому измерению,
- блочное распределение

Глобальный массив



процессная решетка



“Distributed Array” (Darray) Datatype

```
int gsizes[2], distribs[2], dargs[2], psizes[2];
gsizes[0] = m; /* no. of rows in global array */
gsizes[1] = n; /* no. of columns in global array */
distribs[0] = MPI_DISTRIBUTE_BLOCK;
distribs[1] = MPI_DISTRIBUTE_BLOCK;
dargs[0] = MPI_DISTRIBUTE_DFLT_DARG;
dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;

psizes[0] = 2; /* no. of processes in vertical dimension
                of process grid */
psizes[1] = 3; /* no. of processes in horizontal dimension
                of process grid */
```

Darray (2)

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
MPI_Type_create_darray(6, rank, 2, gsizes, distribs, dargs,  
    psizes, MPI_ORDER_C, MPI_FLOAT, &filetype);  
MPI_Type_commit(&filetype);
```

```
MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",  
    MPI_MODE_CREATE | MPI_MODE_WRONLY,  
    MPI_INFO_NULL, &fh);  
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",  
    MPI_INFO_NULL);
```

```
local_array_size = num_local_rows * num_local_cols;  
MPI_File_write_all(fh, local_array, local_array_size,  
    MPI_FLOAT, &status);  
MPI_File_close(&fh);
```


MPI_Type_create_darray

```
int MPI_Type_create_darray(  
    int size, /* число процессов */  
    int rank, /* номер процесса */  
    int ndims, /* размерность массива: и глоб, и локального */  
    int array_of_gsizes[], /* размеры глоб. массива */  
    int array_of_distribs[], /* способ распределения элементов глоб.  
        массива. Например, MPI_DISTRIBUTE_BLOCK */  
    int array_of_dargs[], /* размер чанка, по умолч.  
        MPI_DISTRIBUTE_DFLT_DARGS */  
    int array_of_psize[], /* число элементов в процессорной  
        решетке по каждому измерению */  
    int order, /* способ представления лок. Массива:  
        MPI_ORDER_C */  
    MPI_Datatype oldtype,  
    MPI_Datatype *newtype)
```

Замечания по типу Darray

- Тип darray предполагает конкретную схему распределения данных – как в HPF
- Например, если размер массива не делится нацело на число процессов, то размер блока определяется округлением до ближайшего целого «сверху» ($20 / 6 = 4$)
- darray предполагает строчное распределение процессов в процессорной решетке (так, как это определяется для декартовых топологий в MPI-1)
- Для других схем распределения данных используется тип subarray .