МЕТОДИ И ТЕХНИКИ ЗА ПАРАЛЕЛЕН ВХОД/ИЗХОД ЧАСТ 2

ПРОФ. ПЛАМЕНКА БОРОВСКА

©BOROVSKA

MPI-IO

- MPI-IO е разработен през 1994 г. в IBM Watson Laboratory с цел поддържане на паралелен В/И за MPI.
- NASA използва MPI-IO за изследователските си проекти през 1996 г. и през същата година MPI Forum решава да заложи MPI-IO в MPI-2.
- Стандартът MPI-2 е създаден от MPI форум през 1997 г.
- За разлика от MPI-1, стандартът MPI-2 осигурява паралелен В/И, както и нови възможности за преносима паралелна обработка като операции за отдалечен достъп до паметта и управление на динамични процеси.
- По отношение на В/И, *целта на MPI-2 е да се осигури преносим паралелен В/И*.
- ROMIO е високопроизводителна преносима имплементация на MPI-2 и е оптимиран за достъп до несъседни области от данни, както и съдържа оптимирана имплементация на колективен В/И.

©BOROVSKA

MPI-2 IO

- MPI-IO разширява MPI с производни типове данни и колективни комуникации.
- *Записът на MPI файлове е подобно на изпращането на MPI съобщения, а четенето на MPI файлове е подобно да получаването на MPI съобщения*.
- *MPI-IO файловете са споделени*, т.е., множество процеси, изпълнявани на различни процесори, могат да осъществяват достъп едновременно до един MPI-IO файл.
- Файлът може да е разпределен върху няколко диска, принадлежащи на различни процесори, или върху паралелна дискова система, до която процесорите имат достъп през паралелните комуникационни канали.
- Освен това MPI-IO се възползва в пълна степен от всестранността и гъвкавостта на типовете данни в MPI – и след това доразработва тази концепция чрез формулирането на т.нар. MPI файлови изгледи.
- *„Изгледът на файла” за даден процес* (*file view*) се описва в отделелно колективно извикване с типа на данните във файла и отместването.
- *Изгледът на файла определя достъпните области във файла за даден процес*.

©BOROVSKA

Колективен вход/изход

- Много популярен подход за В/И, базиран на библиотеката MPI-IO, е колективният В/И, който дефинира набор от подпрограми, които осъществяват трансфер на данни към/от външната памет.
- Производните типове данни за паметта и за файловете на колективния В/И интерфейс позволяват съвместни В/И оптимизации при процесите като моделите за достъп на високо ниво се запазват непроменени.
- MPI-IO осигурява възможост за четене и запис в нормален режим, т.е. режим с блокиране, и впоследствие – в асинхронен режим без блокиране, така че, паралелно с обработката на процесите, файловете могат да бъдат четени/записвани във фонов режим.

©BOROVSKA

MPI-2

Поддържат са следните режими за достъп:

- *MPI_MODE_RDONLY — само четене (read only)*
- *MPI_MODE_RDWR — четене и запис,*
- *MPI_MODE_WRONLY —само запис (write only),*
- *MPI_MODE_CREATE — създаване на нов файл,*
- *MPI_MODE_EXCL — грешка, ако се създава файл, който вече съществува,*
- *MPI_MODE_DELETE_ON _CLOSE — изтриване на файла след затваряне,*
- *MPI_MODE_UNIQUE_OPEN — уникално отваряне на файл (не се отваря паралелно от друг процес),*
- *MPI_MODE_SEQUENTIAL — достъпът до файла е само последователен,*
- *MPI_MODE_APPEND — указателите на файловете се позиционират в неговия край.*

MPI-2: дефиниции при В/И

- MPI файлът представлява подредена колекция от типизирани данни.
- MPI поддържа произволен или последователен достъп до всяко интегрално множество данни.
- Даден файл може да бъде отворен колективно от група процеси.
- Всички извиквания за колективен В/И за файл са валидни и колективни за групата процеси.

©BOROVSKA

MPI-2: дефиниции при В/И

- *Отместване във файла (file displacement) - представлява абсолютната позиция на байта относно началото на файла.*
- Отместването определя позицията, от която започва “изгледа на файла”.
- *etype (elementary type) – участъкът или единицата (unit) за достъп до данните и позиционирането*
- *Достъпът до данните се осъществява по etype елементарни участъци*
- *Отместванията се представят в брой елементарни участъци (etype)*

©BOROVSKA

MPI-2: дефиниции за В/И

- Тип на файла (*filetype*) – представлява базата за разпределяне на файла между процесите и дефинира шаблон за достъп до файла
- *filetype* представлява единичен *etype* или *производен MPI datatype ,* който се конструира от множество инстанции на един елементарен тип *etype*
- *Изглед на файла (view) –* дефинира текущото множество от данни, видимо и достъпно в отворен файл, като наредено множество от елементарни типове *(etype)*
- *Всеки процес разполага със свой собствен изглед на файла, дефиниран от 3 параметъра: displacement, etype, filetype*

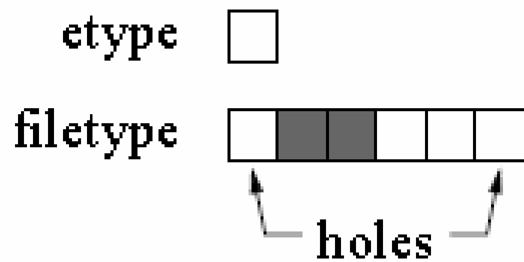
©BOROVSKA

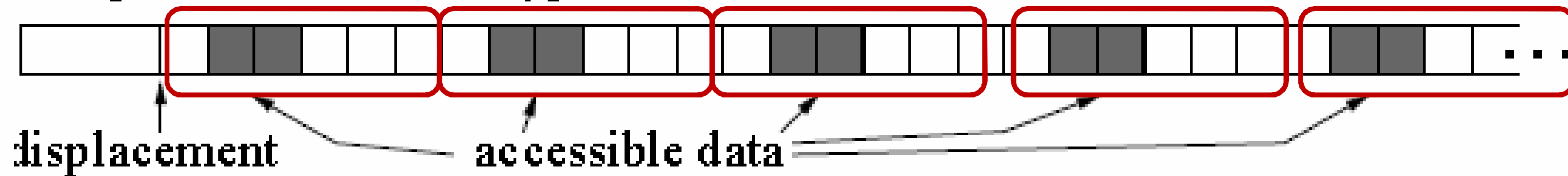
MPI-2: дефиниции за В/И

- За определяне на изгледа се използва шаблона, описан във filetype, който се прилага многократно, започвайки от отместването
- *Default view* – линеен поток от байтове, отместването е 0, *etype* и *filetype* са идентични с *MPI_BYTE*

©BOROVSKA

File Tiling





©BOROVSKA

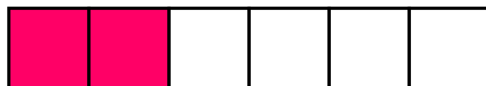
file view

- Файловите типове са изградени от *etypes*, които сами по себе си могат да бъдат получени от типовете данни.
- Моделът за достъп до файловия тип имплицитно се повтаря напред, започвайки от disp
- Файловият изглед задава достъпните области на даден файл чрез използването на типа на данните във файла като шаблон и неговото повтаряне, започвайки от байтовото изместване

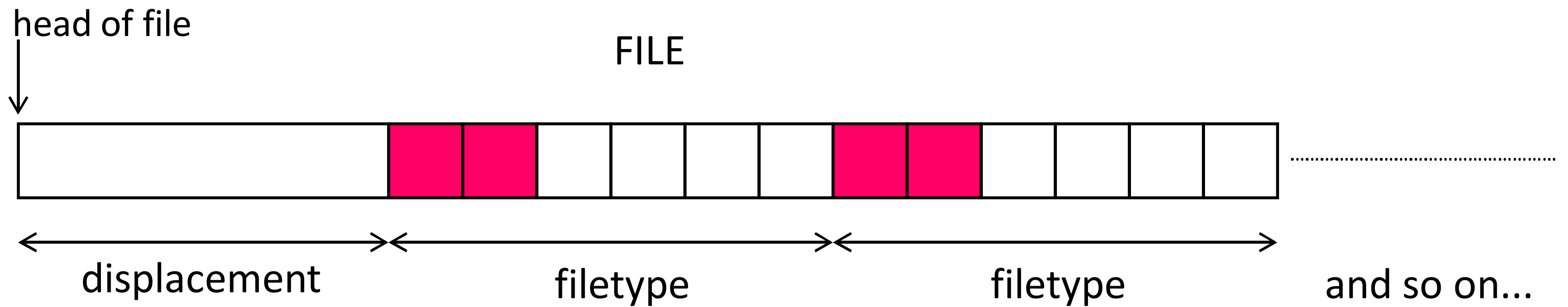
©BOROVSKA

file view

etype = MPI_INT



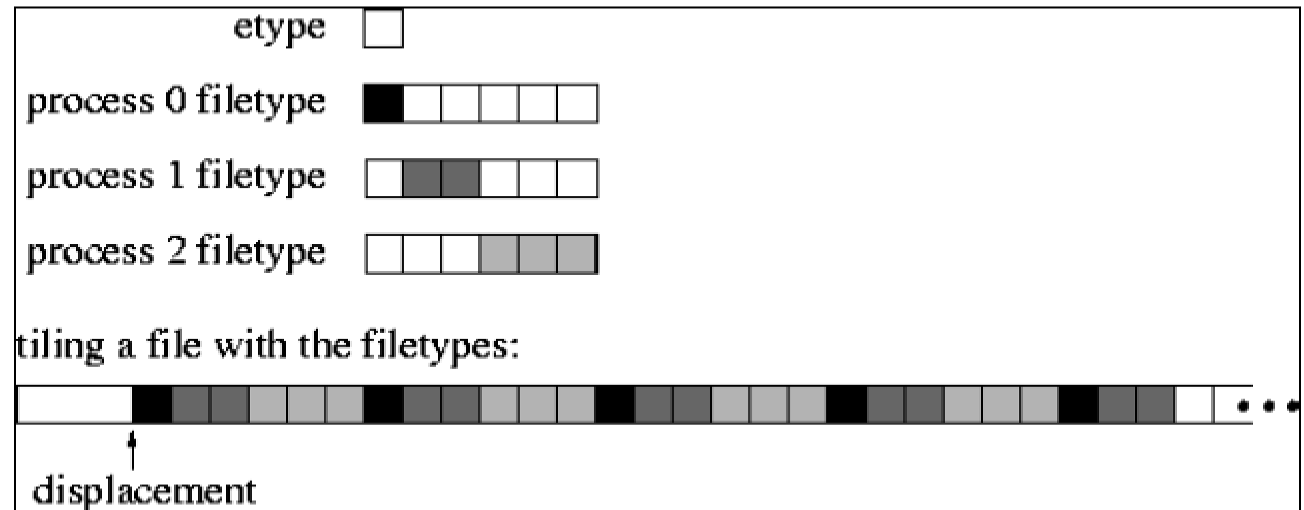
filetype = two MPI_INTs followed by
a gap of four MPI_INTs



©BOROVSKA

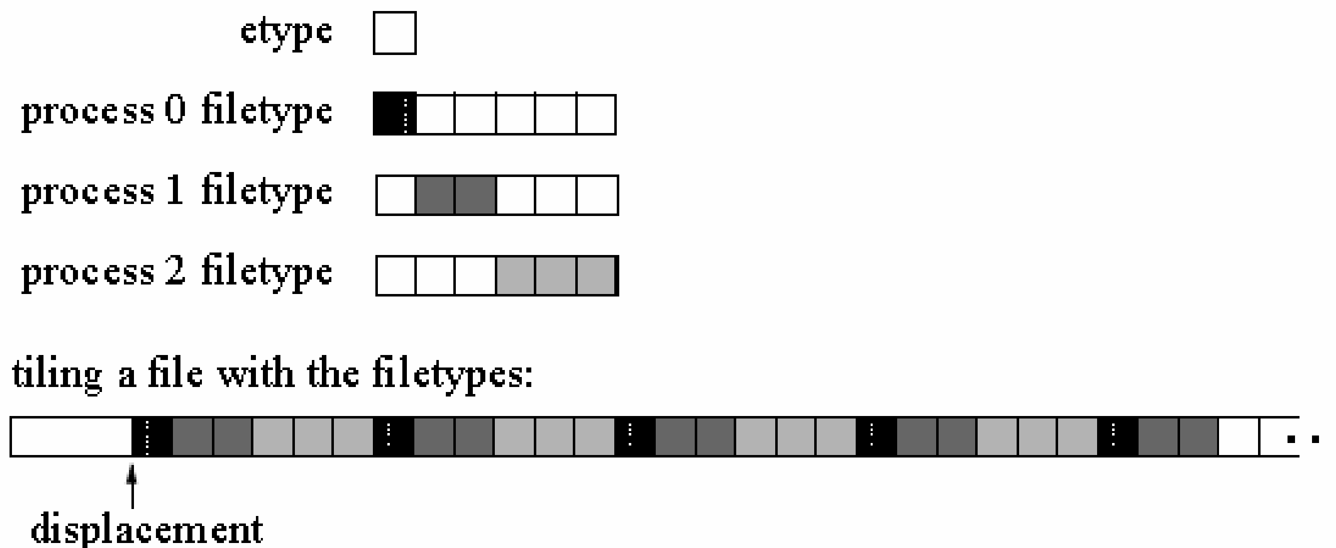
file view

File View
Part of the file which is visible to a process.



Комплементарни изгледи

- Група от процеси могат да използват комплементарни изгледи за глобално разпределение на данните, като напр., при gather/scatter



©BOROVSKA

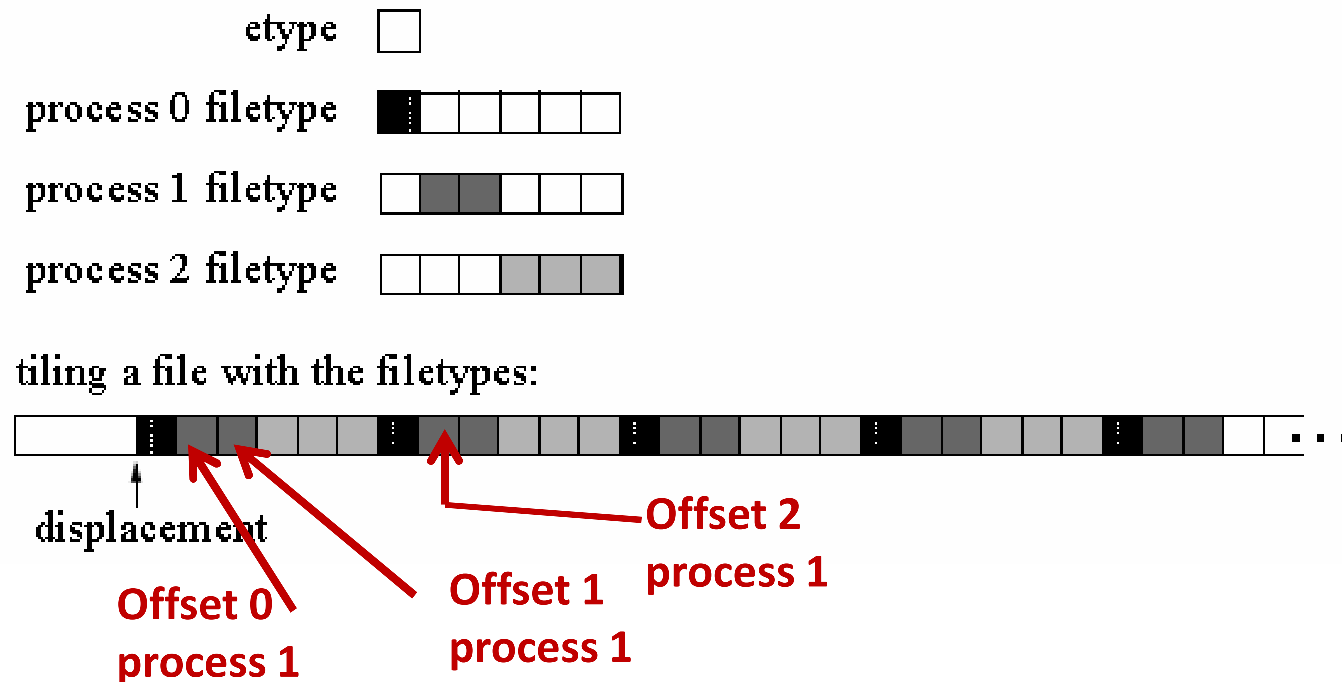
офсет

- *Указва на процеса стартовата позиция за достъп до файла*
- Offset – указва позиция във файла спрямо текущия изглед, изразен в брой елементарни типове (etype)
- Празните позиции (дупките) в fyletype на изгледа се пропускат при изчисляването на офсета
- Офсет 0 е позицията на първия etype, видим в изгледа

©BOROVSKA

offset

- За примера - офсет 2 за процес 1 е позицията на 8-мия елементарен тип във файла след отместването



©BOROVSKA

File size и end of file

- File size – размерът на MPI файловете се измерва в брой байтове от началото на файла
- Дължината на новосъздаден файл е 0
- Когато се третира като изместване, размерът на файла указва позицията след последния байт на файла
- За всеки изглед *end of file* е офсета на първия etype, достъпен в текущия изглед, след последния байт на файла

©BOROVSKA

Указател на файл (File pointer)

- Представлява явен (експлицитен) офсет, поддържан от MPI
- Индивидуалните указатели на файлове са локални за всеки процес, отворил файла
- Споделен указател на файл “shared file pointer” – споделя се от групата процеси, отворили файла
- File handle (файлов манипулатор) – прозрачен обект, създаден от MPI_FILE_OPEN и освободен от MPI_FILE_CLOSE
- Всички операции над отворен файл реферират файла посредством file handle

©BOROVSKA

Отваряне на файл

MPI_FILE_OPEN(comm, filename, amode, info, fh)

IN	comm	communicator (handle)
IN	filename	name of file to open (string)
IN	amode	file access mode (integer)
IN	info	info object (handle)
OUT	fh	new file handle (handle)

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info,
              MPI_File *fh)
```

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER COMM, AMODE, INFO, FH, IERROR
```

```
static MPI::File MPI::File::Open(const MPI::Intracomm& comm,
              const char* filename, int amode, const MPI::Info& info)
```


©BOROVSKA

MPI_FILE_OPEN

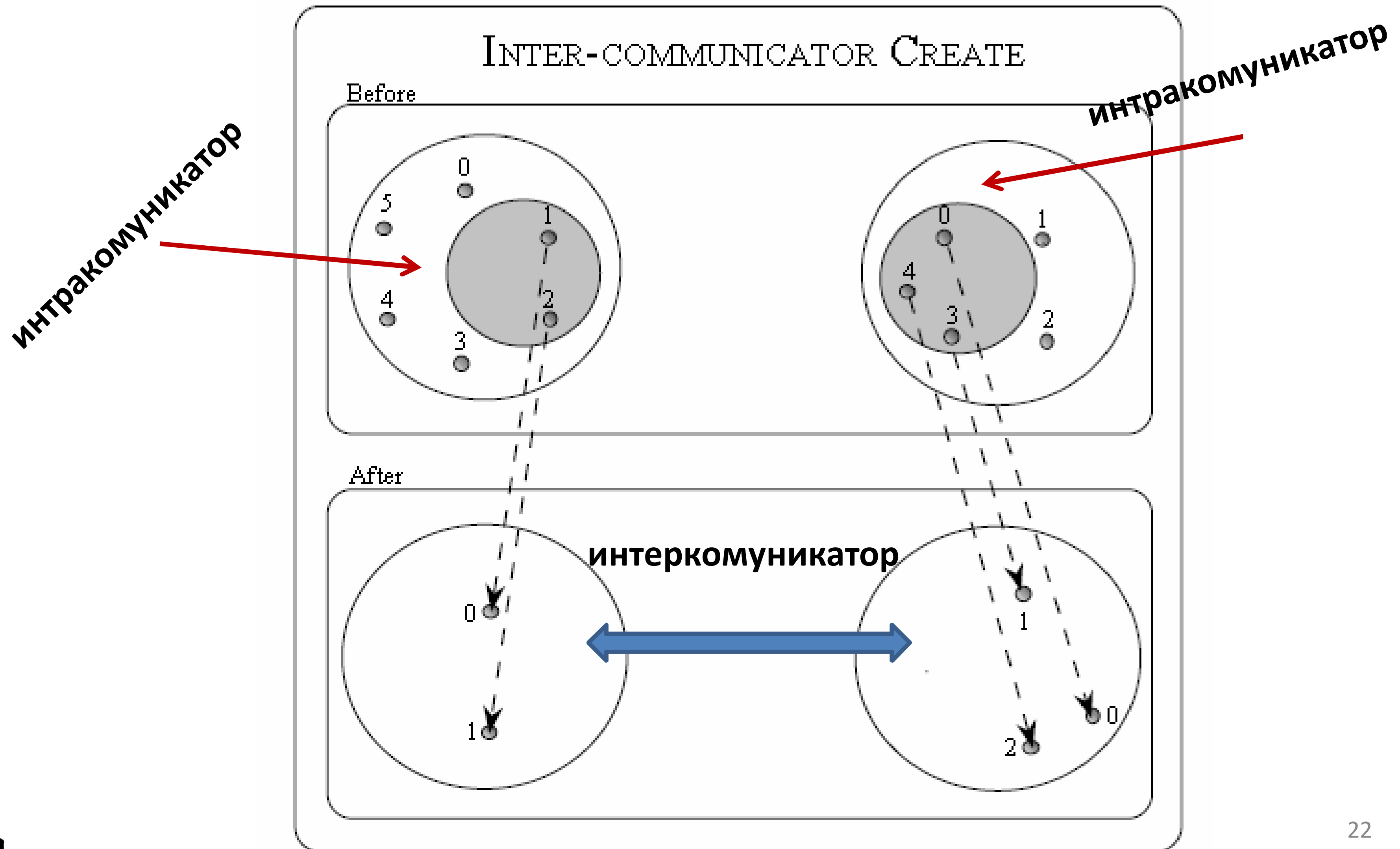
- MPI_FILE_OPEN отваря файла за всички процеси в комуникатора
- comm. трябва да бъде интракомуникатор
- Всички процеси трябва да изпълнят операцията, като реферират един и същи файл
- Отделен процес може да отвори самостоятелно файл с комуникатора MPI_COMM_SELF
- Преди да се извика MPI_FINALIZE програмистът трябва да затвори всички отворени файлове с MPI_FILE_CLOSE

©BOROVSKA

Интракомуникатори

- Интракомуникаторите (Intracommunicators) са най-често използваната форма на комуникатори при MPI.
- Всеки интракомуникатор съдържа множество процеси, всеки от които се идентифицира посредством своя "ранг" в рамките на комуникатора.
- Ранговете на процесите са от 0 до Size-1.
- Всеки процес в комуникатора може да изпрати съобщение до всеки друг процес в рамките на комуникатора или да получи съобщение всеки друг процес в рамките на комуникатора.
- Интракомуникаторите също така поддържат широк спектър от колективни операции, които касаят всички процеси в комуникатора.
- Мнозинството комуникации при MPI се осъществяват в рамките на интракомуникатори, докато малък брой MPI програми изискват интеркомуникатори.

Конструктори за интеркомуникатори – създават интеркомуникатор от интракомуникатори



©BOROVSKA

Интеркомуникатори

- MPI_INTERCOMM_CREATE, създава intercommunicator от два intracommunicators,
- MPI_COMM_DUP, създава дупликат на съществуващ intercommunicator (or intracommunicator).
- Двете групи в интеркомуникатора се наричат *лява* и дясна *група*.
- Даден процес в интеркомуникатора е член на една от двете групи – лявата или дясната
- От гледна точка на даден процес, групата, към която този процес принадлежи, се нарича “локална група”
- Другата група, спрямо разглеждания процес, е отдалечената група (*remote* group).
- Етикетите лява и дясна група дават възможност да се описват двете групи в интеркомуникатора без да св взема предвид гледната точка на процесите

Конструктори за интеркомуникатори

MPI_COMM_CREATE(comm_in, group, comm_out)

IN comm_in	original communicator (handle)
IN group	group of processes to be in new communicator (handle)
OUT comm_out	new communicator (handle)

Затваряне на файл

```
MPI_FILE_CLOSE(fh)
  INOUT    fh                                   file handle (handle)

int MPI_File_close(MPI_File *fh)

MPI_FILE_CLOSE(FH, IERROR)
    INTEGER FH, IERROR
```

- Затварянето на файл е колективна операция
- При затварянето му файлът се изтрива, ако е бил отворен в режим *MPI_MODE_DELETE_ON _CLOSE*
- *Програмистът трябва да осигури финализирането на всички неблокиращи заявки и колективни операции, асоциирани с fh , създаден от процес, преди затварянето на файла*

©BOROVSKA

Изтриване на файл

MPI_FILE_DELETE(filename, info)

IN	filename	name of file to delete (string)
IN	info	info object (handle)

```
int MPI_File_delete(char *filename, MPI_Info info)

MPI_FILE_DELETE(FILENAME, INFO, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER INFO, IERROR

static void MPI::File::Delete(const char* filename, const MPI::Info& info)
```


©BOROVSKA

MPI_FILE_SET_VIEW

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)

INOUT	fh	file handle (handle)
IN	disp	displacement (integer)
IN	etype	elementary datatype (handle)
IN	filetype	filetype (handle)
IN	datarep	data representation (string)
IN	info	info object (handle)

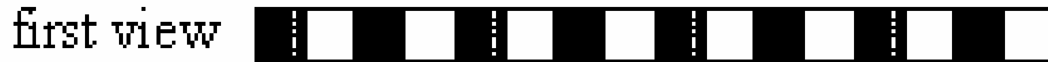
```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
             MPI_Datatype filetype, char *datarep, MPI_Info info)
```

```
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
    INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
    CHARACTER*(*) DATAREP
    INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

```
void MPI::File::Set_view(MPI::Offset disp, const MPI::Datatype& etype,
             const MPI::Datatype& filetype, const char* datarep,
             const MPI::Info& info)
```


©BOROVSKA

MPI_FILE_SET_VIEW



file structure:

- Препоръчително е, MPI_FILE_SET_VIEW да се използва непосредствено след MPI_FILE_OPEN
- Използва се, когато различни участъци от данните се обработват по различен модел

©BOROVSKA

MPI_FILE_GET_VIEW

MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)

IN	fh	file handle (handle)
OUT	disp	displacement (integer)
OUT	etype	elementary datatype (handle)
OUT	filetype	filetype (handle)
OUT	datarep	data representation (string)

```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
              MPI_Datatype *filetype, char *datarep)
```

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
    INTEGER FH, ETYPE, FILETYPE, IERROR
    CHARACTER*(*) DATAREP, INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

```
void MPI::File::Get_view(MPI::Offset& disp, MPI::Datatype& etype,
              MPI::Datatype& filetype, char* datarep) const
```


©BOROVSKA

MPI_FILE_SEEK

MPI_FILE_SEEK(fh, offset, whence)

INOUT	fh	file handle (handle)
IN	offset	file offset (integer)
IN	whence	update mode (state)

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

MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

void MPI::File::Seek(MPI::Offset offset, int whence)
```

©BOROVSKA

MPI_FILE_SEEK

- Актуализира индивидуалния указател на файла, съобразно “whence”, като whence може да има следните стойности:

1. MPI_SEEK_SET – указателя се установява на offset
2. MPI_SEEK_CUR - указателя се установява на текущата позиция на указателя + offset
3. MPI_SEEK_END - указателя се установява на end of file + offset

- Offset може да има отрицателна стойност, с което се прави търсене назад

©BOROVSKA

DATA ACCESS

3 ортогонални аспекта

1. *Позициониране* – явни офсети или неявни указатели на файлове
2. *Синхронизация* – с блокиране или без блокиране и колективно с разделяне (split collective)
3. *Координация* – колективна или неколективна

©BOROVSKA

DATA ACCESS

positioning	synchronism	coordination	
		noncollective	*collective*
explicit offsets	*blocking*	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	nonblocking & split collective	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
individual file pointers	*blocking*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	nonblocking & split collective	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
shared file pointer	*blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	nonblocking & split collective	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

©BOROVSKA

Представяне на данните при MPI

- MPI поддържа няколко вида представяне на данните – native, internal, external32
- MPI поддържа и представяне на данните, дефинирано от потребителя
- Видовете “native” и “internal” зависят от имплементацията, докато видът “external32” е общ за всички MPI имплементации и улеснява оперативната съвместимост
- Видът на представяне на данните се специфицира в аргумента *datarep* в MPI_FILE_SET_VIEW

©BOROVSKA

Представяне на данните при MPI

- “native” – при това представяне данните се съхраняват във файла така, както са съхранени в паметта – използва се само в хомогенна среда
- “internal” – (32-bit big-endian IEEE format) - използва се за В/И както в хомогенна, така и в хетерогенна среда; данните могат да се съхранят на диска в произволен формат, но задължително се документира изпълнителната среда на имплементацията, за да може да се използват отново данните

©BOROVSKA

Представяне на данните при MPI

- "external32"- при всички операции четене/запис данните се конвертират от/във вида "external32"
- Данните на диска са винаги в каноничното представяне "external32", докато данните в паметта са представени във вида "native"
- Използва се в хетерогенна среда – при четене от файла, данните автоматично се конвертират във вида "native"
- Оперативна съвместимост
- Допълнителни системни разходи за конвертиране на данните

©BOROVSKA

Консистентност на файла

- File consistency semantics
- Consistency – съгласуваност, кохерентност
- Семантиката на консистентността дефинира резултата и последствията от множествения достъп до един и същ файл
- Последователната консистентност означава, че резултатът от множество операции ще бъде еквивалентен на резултата от изпълнението на операциите последователно, в съответствие с реда на изпълнението им в програмата
- MPI осигурява 3 нива на консистентност:

1. Последователна консистентност на процесите, използвайки един и същ манипулатор на файла (file handle)
2. Последователна консистентност на достъпите, при което манипулаторите на файловете са създадени при едно колективно отваряне на файла в режим “atomic”
3. Консистентност на достъпите, наложена от програмиста, посредством извиквания на MPI_FILE_SYNC

©BOROVSKA

Пример 1

- Файлът е отворен с комуникатор MPI_COMM_WORLD.
- Всеки процес прави запис в отделен участък на файла и впоследствие чете само това, което е записал.
- В този случай MPI гарантира коректно четене на данните

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)
```

Пример 2

- Аналогично на пример 1, с тази разлика, че всеки процес след записа чете данните, записани от другите процеси (припокриващи се достъпи - overlapping accesses)
- В случая MPI не гарантира, че данните автоматично ще бъдат прочетени правилно

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)
```

Допълнителни мерки за консистентност за пример 2

1. Активиране на “atomicity to true”
2. Затваряне на файла и повторно отваряне
3. Осигуряване, че записите на кой да е процес не са едновременни с четене или запис на друг процес

Пример 2, вариант 1
Set atomicity to true

Process 0	Process 1
MPI_File_open(MPI_COMM_WORLD,…)	MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_set_atomicity(fh1,1)	MPI_File_set_atomicity(fh2,1)
MPI_File_write_at(off=0,cnt=100)	MPI_File_write_at(off=100,cnt=100)
MPI_Barrier	MPI_Barrier
MPI_File_read_at(off=100,cnt=100)	MPI_File_read_at(off=0,cnt=100)

Пример 2, вариант 2
Close and reopen file

Process 0	Process 1
MPI_File_open(MPI_COMM_WORLD,…)	MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_write_at(off=0,cnt=100)	MPI_File_write_at(off=100,cnt=100)
MPI_File_close	MPI_File_close
MPI_Barrier	MPI_Barrier
MPI_File_open(MPI_COMM_WORLD,…)	MPI_File_open(MPI_COMM_WORLD,…)
MPI_File_read_at(off=100,cnt=100)	MPI_File_read_at(off=0,cnt=100)

Пример 2, Вариант 3

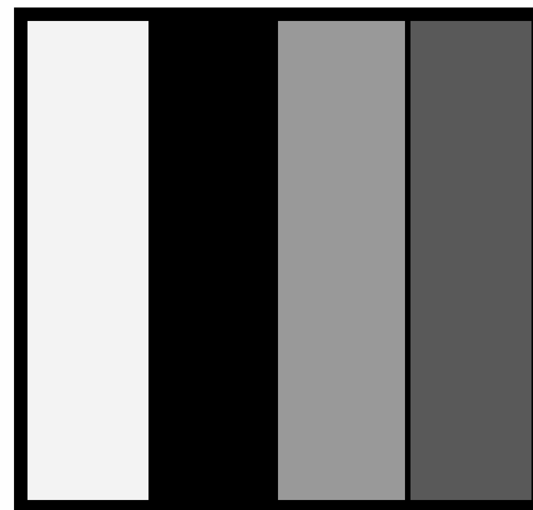
- *Да се осигури, че никоя последователност на записа на някой процес не е едновременна с коя да е последователност (read или write) на друг процес*
- Последователност е множество операции между всяка двойка функции open, close, или file_sync
- Последователност на записа е последователност, при която коя да е от функциите е запис

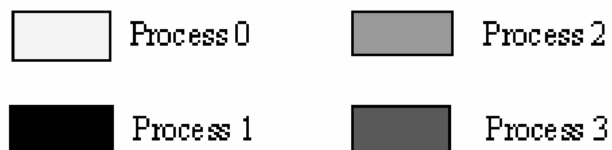
Пример 2, Вариант 3

Process 0	Process 1
`MPI_File_open(MPI_COMM_WORLD,…)`	`MPI_File_open(MPI_COMM_WORLD,…)`
`MPI_File_write_at(off=0,cnt=100)`	
`MPI_File_sync`	`MPI_File_sync /*collective*/`
`MPI_Barrier`	`MPI_Barrier`
`MPI_File_sync /*collective*/`	`MPI_File_sync`
	`MPI_File_write_at(off=100,cnt=100)`
`MPI_File_sync /*collective*/`	`MPI_File_sync`
`MPI_Barrier`	`MPI_Barrier`
`MPI_File_sync`	`MPI_File_sync /*collective*/`
`MPI_File_read_at(off=100,cnt=100)`	`MPI_File_read_at(off=0,cnt=100)`
`MPI_File_close`	`MPI_File_close`

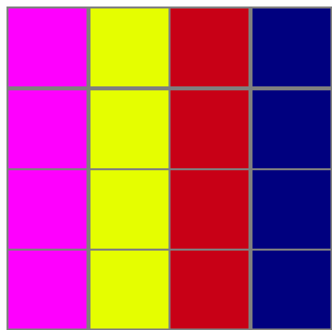
Конструктор Filetype за подмасив

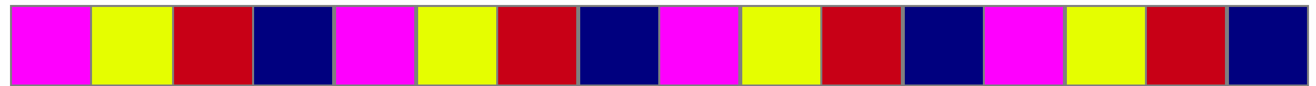
- Проектираме паралелна програма, при която 4 процеса обработват двумерен масив 100x100 от числа с плаваща точка с двойна точност
- Всеки процес обработва блок от 25 колони
- Процес 0 обработва блока с колони 0-24
- Процес 1 обработва блока с колони 25-49 и т.н.





Примерен двумерен масив





ДВУМЕРЕН МАСИВ, ЗАПИСАН ВЪВ ФАЙЛА ПО РЕДОВЕ

ДВУМЕРЕН МАСИВ,
ОБРАБОТВАН ПО КОЛОНИ

Създаване на filetype за всеки процес

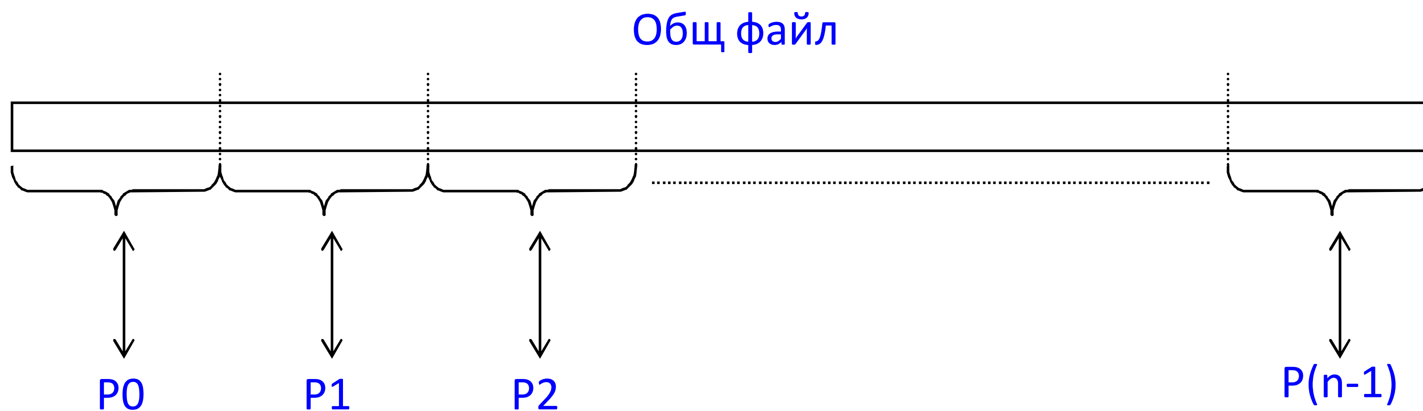
```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;

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);
```

Паралелен В/И

- Множество процеси на паралелната програма четат или записват данни от общ файл
- Висока производителност
- Общият файл с резултатите може да се използва от други инструменти, като например за визуализация



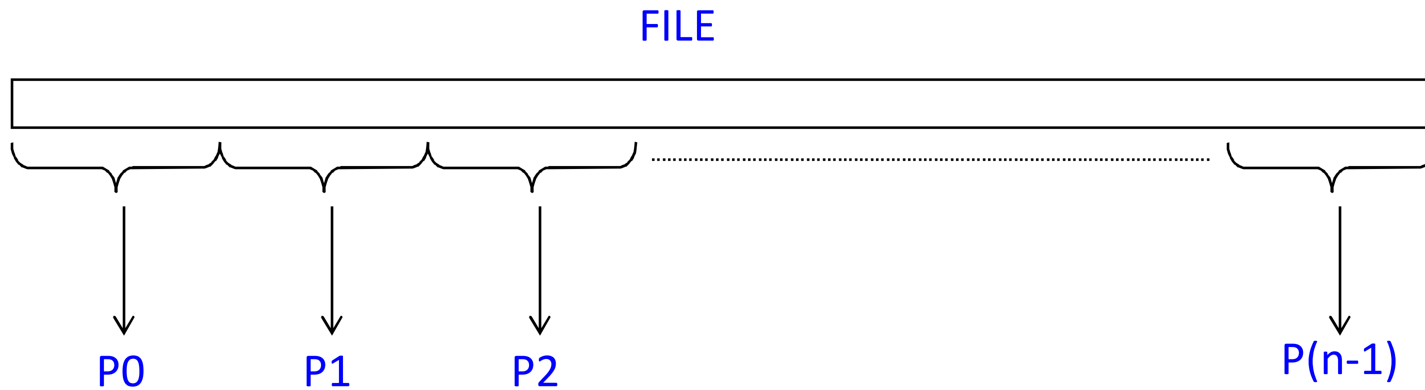
©BOROVSKA

Предимства на паралелния В/И

- Записът на файл е аналогичен на изпращането на съобщение, а четенето – на получаване на съобщение
- Механизми на паралелната В/И система
  - За дефиниране на колективни операции (*MPI комуникатори*)
  - Дефиниране (оформяне) на разпръснати области с данни в паметта и във файла (*MPI datatypes*)
  - Финализиране на неблокиращи операции с тестване (*MPI request objects*)

©BOROVSKA

Прост MPI В/И



Всеки процес трябва да прочете част от данните от общ файл

©BOROVSKA

Използване на индивидуални указатели (File Pointers)

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

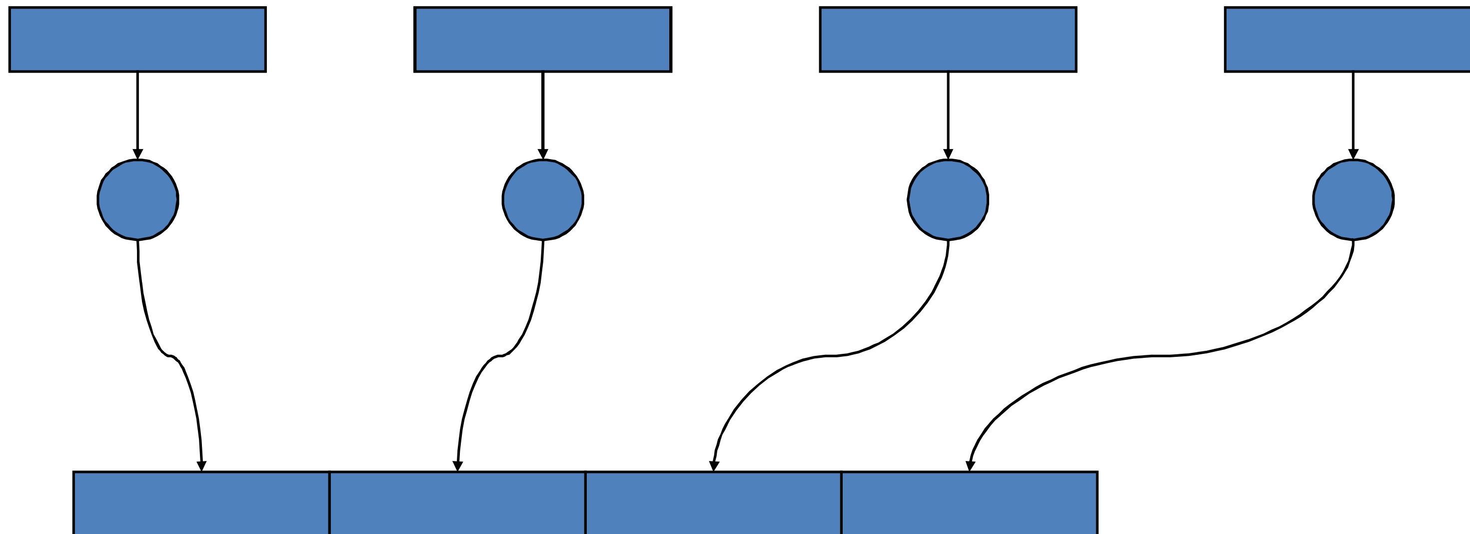
©BOROVSKA

Запис на файл

- Използваме ***MPI_File_write*** или ***MPI_File_write_at***
- Използваме ***MPI_MODE_WRONLY*** или ***MPI_MODE_RDWR*** като флагове на ***MPI_File_open***
- Ако файлът е новосъздаден, то флагът ***MPI_MODE_CREATE*** също трябва да бъде изплзван при ***MPI_File_open***

©BOROVSKA

Използване на изгледи на файла при запис



- Процесите осъществяват запис в споделен файл
- ***MPI_File_set_view*** разпределя участъци от файла на паралелните процеси

©BOROVSKA

Използване на изгледи на файла при запис

- Изгледат на файла се задава от тройката изместване (disp), etype и filetype, която се залага в **`MPI_File_set_view`**
- *displacement* = брой байтове, които се пропускат от началото на файла
- *etype* = базова едница за достъп до данните (може да бъде базов или производен datatype)
- *filetype* = специфицира кои части от файла ще бъде видими за процеса

©BOROVSKA

Пример: запис на файл с изгледи

```
MPI_File thefile;

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);
```


©BOROVSKA

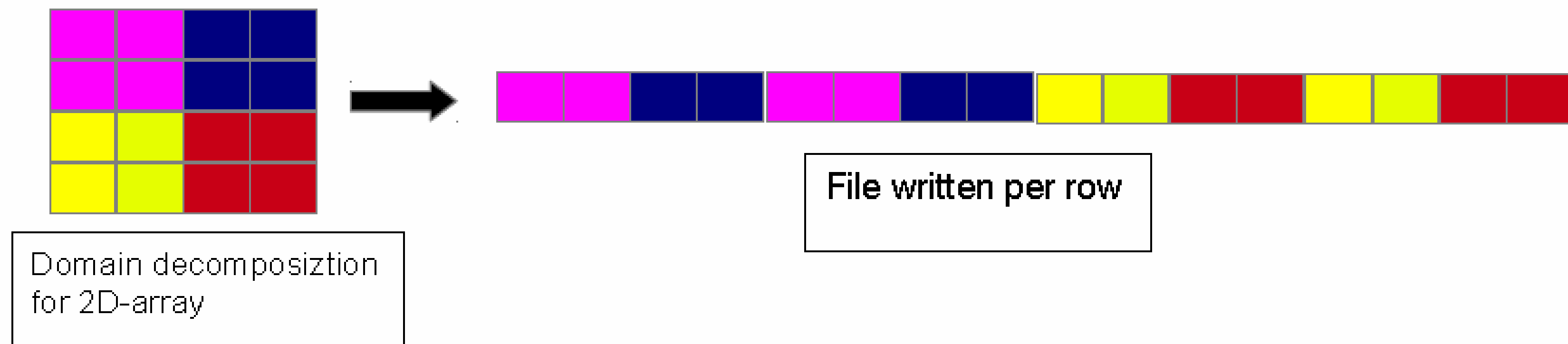
COLLECTIVE BLOCKING I/O

```
int MPI_File_write_all(MPI_File fh, void *buf, int count,
   MPI_Datatype datatype, MPI_Status *status)

int MPI_File_read_all( MPI_File mpi_fh, void *buf, int
   count, MPI_Datatype datatype, MPI_Status *status )
```

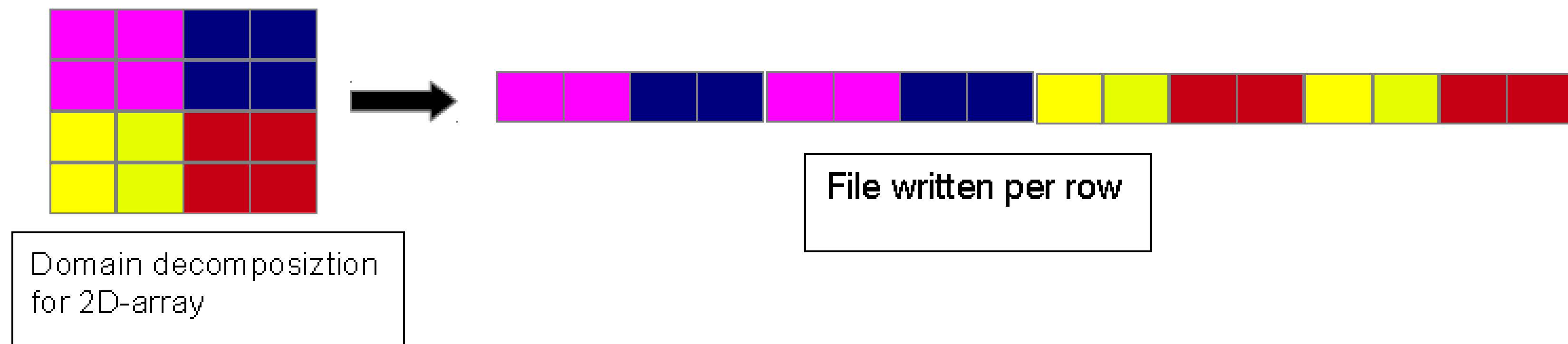
- При колективния В/И всички процеси в комуникатора изпълняват В/И операции
- Възможност за оптимизация на процедурата за четене/запис
- Особено ефективен при неатомични операции

Запис на многомерни масиви




```
...
INTEGER :: sizes = (/4, 4/)
INTEGER :: subsizes = (/2, 2/)
INTEGER, DIMENSION(2,2) :: buf
...
MPI_CART_COORDS(MPI_COMM_WORLD, myid, 2, starts, err)
CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, MPI_INTEGER, &
    MPI_ORDER_C, filetype, err)
CALL MPI_TYPE_COMMIT(filetype)
CALL MPI_FILE_SET_VIEW(file, 0, MPI_INTEGER, filetype, "native", &
MPI_INFO_NULL, err)
CALL MPI FILE WRITE(file, buf, count, MPI INTEGER, status, err)
```

Запис на многомерни масиви – колективен В/И




```
INTEGER :: sizes = (/4, 4/)
INTEGER :: subsizes = (/2, 2/)
INTEGER, DIMENSION(2,2) :: buf
...
CALL MPI_CART_COORDS(MPI_COMM_WORLD, myid, 2, starts, err)
CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, MPI_INTEGER, &
    MPI_ORDER_C, filetype, err)
CALL MPI_TYPE_COMMIT(filetype)
CALL MPI_FILE_SET_VIEW(file, 0, MPI_INTEGER, filetype,&
        'native', MPI_INFO_NULL, err)
CALL MPI_FILE_WRITE_ALL(file, buf, count, MPI_INTEGER, status, err)
```

Darray и колективен В/И

```
/* int MPI_Type_create_darray (int size, int rank, int ndims, int
   array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int
   array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype
   *newtype) */
```

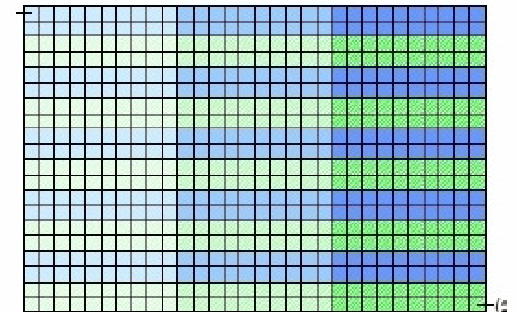
```
int gsizes[2], distribs[2], dargs[2], psizes[2];

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 */
```



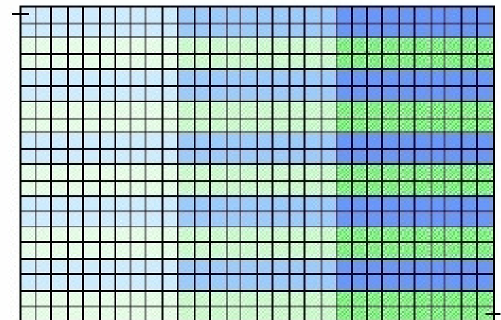
Darray и колективен В/И

```
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);
```



Колективен неблокиращ В/И

- При колективния В/И само ограничена форма на неблокиращ В/И се поддържа, нар. Split Collective
- Колективните операции могат да се разделят на 2 части
- Само една split или колективна операция за файлов манипулатор се допуска в даден момент
- Трябва да се използват еднакви аргументи BUF при in-begin и _end извикванията

```
MPI_File_read_all_begin( MPI_File mpi_fh, void *buf, int count,
   MPI_Datatype datatype )

      …computation…

MPI_File_read_all_end( MPI_File mpi_fh, void *buf, MPI_Status
   *status );
```

КРАЙ

ЧАСТ 2

©BOROVSKA