MPI PARALLEL I/O

Termin 2

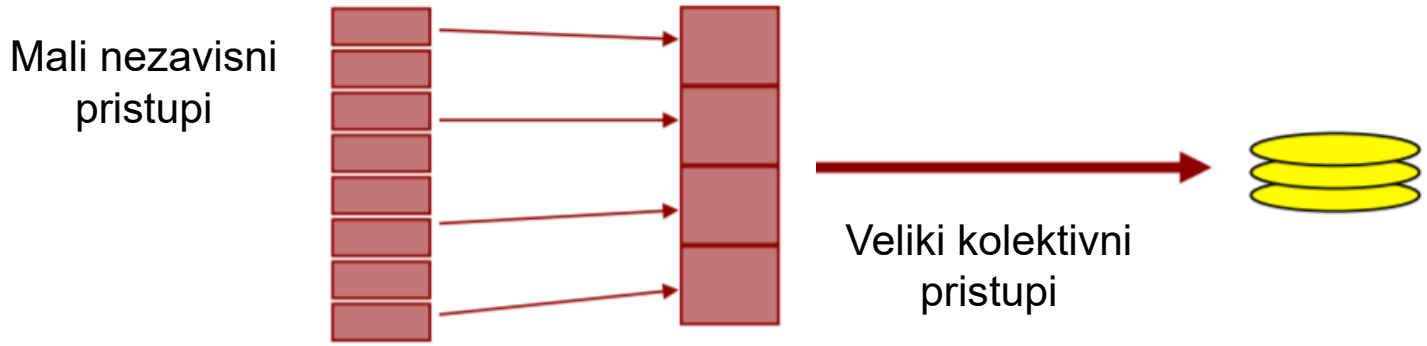
Mast. inž. Nađa Gavrilović
Prof. dr Natalija Stojanović

Grupne I/O operacije

- Operacije koje smo do sada koristili obezbeđuju **nezavisan pristup fajlu** od strane procesa – svaki proces pristupa fajlu nezavisno u odnosu na ostale procese
- Poziv **grupne I/O operacije** podrazumeva da svi procesi u grupi **u isto vreme vrše čitanje/upis podataka** i da „čekaju jedni na druge“
- Sva standardna pravila za rad sa grupnim operacijama važe i sada!
  - Grupne operacije su operacije koje se primenjuju simultano nad svim članovima jedne grupe.
  - Operacija se izvršava kada svi procesi pozovu odgovarajuću operaciju sa svojim parametrima.
  - Svaki proces mora da pozove grupnu operaciju da bi se ona obavila!

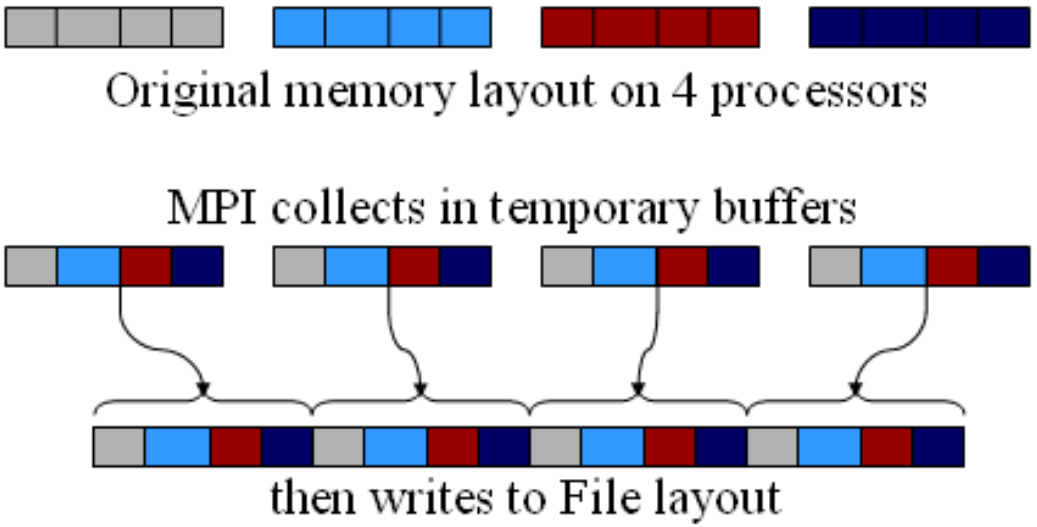
Grupne I/O operacije - Nastavak

- **Prednost grupnog I/O** – Poboljšanje performansi pristupa
  - MPI implementacija vrši optimizaciju velikog broja read/write zahteva svih procesa, **spajanjem više zahteva** u cilju veće efikasnosti
  - Efikasno u slučaju kada su pristupi različitih procesa nekontinualni i preklapaju se



Grupne I/O operacije - Nastavak

- Pozivanjem grupnih I/O funkcija omogućava se da MPI implementacija u pozadini optimizuje zahtev na osnovu kombinovanih zahteva svih procesa.
- Implementacija može objediniti zahteve različitih procesa i opslužiti objedinjeni zahtev efikasno.
- Primer – Slučaj kada su pristupi različitih procesa nekontinualni i isprepleteni:



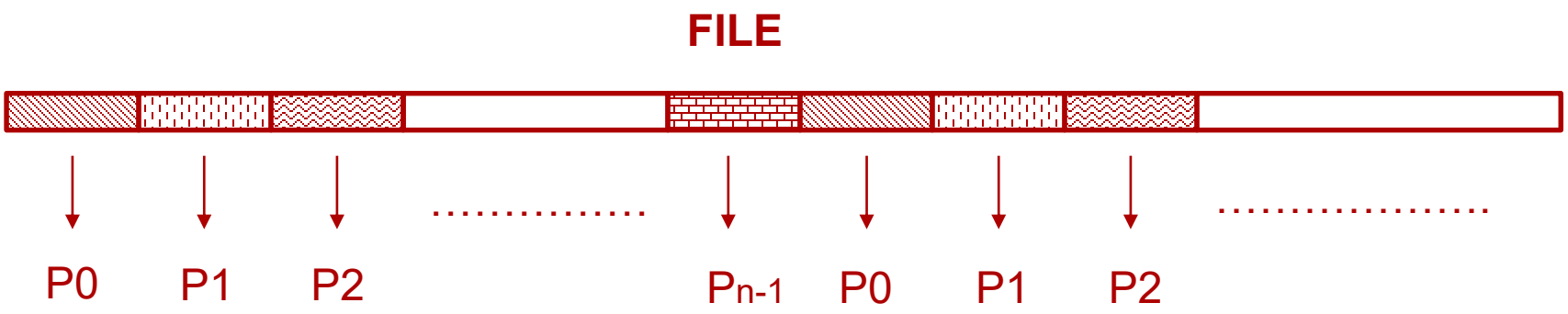
Grupne I/O operacije - Nastavak

- Grupne operacije za upis i čitanje podataka:
  - int **MPI_File_read_all**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
  - int **MPI_File_write_all**(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

- Parametri funkcija su isti kao u slučaju osnovnih **MPI_File_read** i **MPI_File_write** funkcija

Grupne I/O operacije - Primer

- Ukupno N procesa
- Svaki proces čita manje delove fajla po round-robin principu



- Efikasno rešenje:
  - Nekontinualni pristup podacima korišćenjem **pogleda podataka** i dodatno, upotreba **grupnih operacija** za simultano čitanje/upis podataka od strane svih procesa u grupi

Grupne I/O operacije – Primer – Rešenje

- 1. korak – Otvoriti fajl za čitanje
  - **MPI_File_open** (MPI_COMM_WORLD , "/pfs/datafile" , MPI_MODE_RDONLY, MPI_INFO_NULL , &fh);
- 2. korak – Kreirati izvedeni tip podatka koji odgovara zahtevima za pristup podacima od strane svakog procesa

```
bufsize = FILESIZE/nprocs;
buf = (int *) malloc(bufsize);
nints = bufsize/sizeof(int);
```

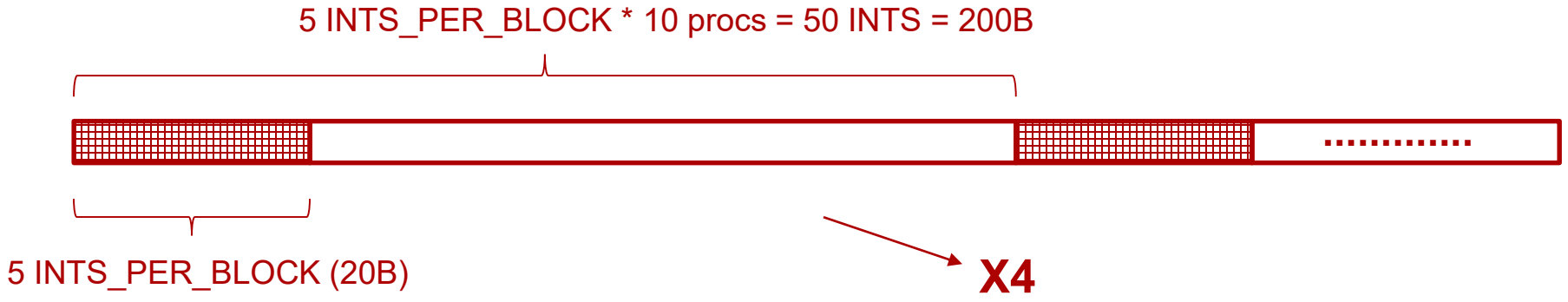
  - **MPI_Type_vector** (nints / INTS_PER_BLK, INTS_PER_BLK, INTS_PER_BLK * nprocs, MPI_INT, &filetype);
  - **MPI_Type_commit** (&filetype);

MPI_Type_vector primer

- Primer:
  - FILESIZE = 800(B)
  - nprocs = 10
  - bufsize = 80
  - nints = 80/4 = 20
  - INTS_PER_BLK = 5 – broj integer-a u svakom pojedinačnom bloku

```
bufsize = FILESIZE/nprocs;
buf = (int *) malloc(bufsize);
nints = bufsize/sizeof(int);
```

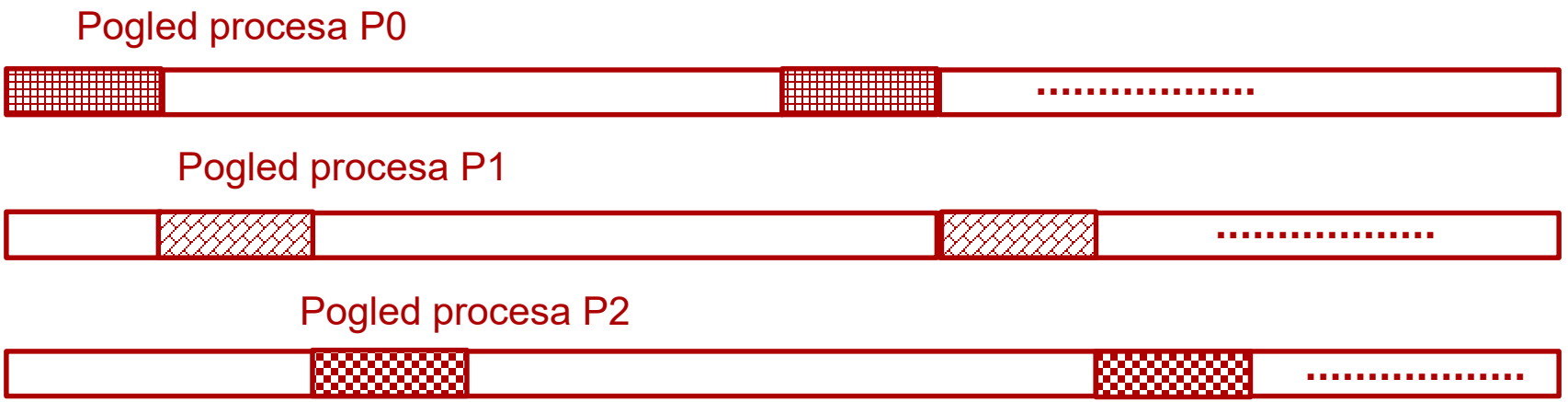
- **MPI_Type_vector** (nints / INTS_PER_BLK, INTS_PER_BLK, INTS_PER_BLK * nprocs, MPI_INT , &filetype);



Grupne I/O operacije – Primer – Rešenje - Nastavak

- 3. korak – Kreirati pogled na osnovu izvedenog tipa podatka
  - **MPI_File_set_view** (fh, INTS_PER_BLK * sizeof(int) * rank, MPI_INT, filetype, "native", MPI_INFO_NULL);

- Pogled iz ugla svakog procesa:



Grupne I/O operacije – Primer – Rešenje - Nastavak

- 4. korak – Iskoristiti grupnu operaciju za čitanje odgovarajućih podataka od strane svakog procesa
  - **MPI_File_read_all** (fh, buf, **nints**, MPI_INT, MPI_STATUS_IGNORE);
  - Funkcija se mora pozvati od strane svakog procesa u komunikatoru koji je prosleđen tokom otvaranja fajla (funkcijom MPI_File_open)!

- 5. korak – Zatvaranje fajla
  - **MPI_File_close**(&fh);

Grupne I/O operacije – Primer – Celo rešenje

```
#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);

  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;
}
```

Pristup nizovima u datotekama

Pristup nizovima u datotekama

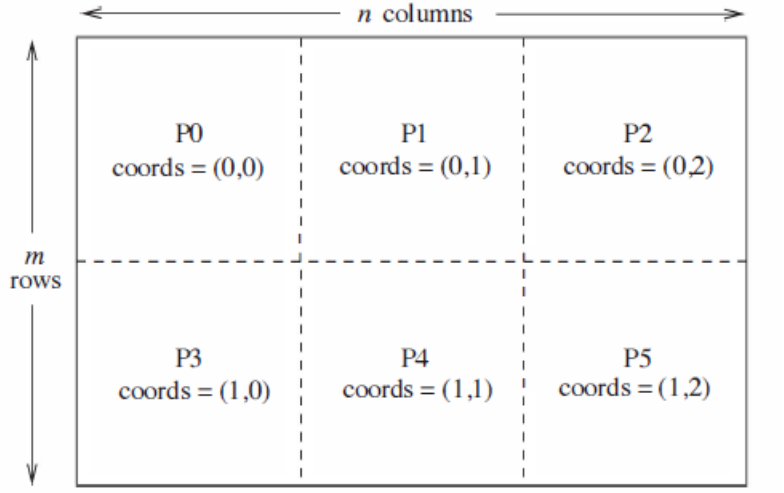
- Čest I/O zahtev u paralelnim programima - pristup podnizovima i dustribuiranim nizovima koji su sačuvani u fajlovima
- MPI omogućava jednostavan pristup multidimenzionalnim nizovima, koji su na neki način distribuirani procesima u grupi
  - int **MPI_Type_create_darray**(int size, int rank, int ndims, 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 **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)

Funkcija MPI_Type_create_darray

- Funkcija kreira izvedeni tip podatka opisom N-dimenzionalnog podpolja unutar N-dimenzionalnog polja
- int **MPI_Type_create_darray**(int size, int rank, int ndims, int gsizes[], int distribs[], int dargs[], int psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - Size – Broj procesa među kojima je niz distribuiran
  - Ndims – Broj dimenzija polja i podpolja (N)
  - gsizes - Broj elemenata starog tipa (oldtype) u svakoj dimenziji polja (niz pozitivnih celih brojeva)
  - distribs – Način distribucije - MPI_DISTRIBUTE_BLOCK
  - dargs – Parametar distribucije za svaku dimenziju - MPI_DISTRIBUTE_DFLT_DARG
  - psizes – Broj **procesa** u svakoj dimenziji među kojima je polje distribuirano. Uzima se da grid procesa ima isti broj dimenzija kao polje i podpolje. Ako niz po nekoj dimenziji nije distribuiran, broj procesa u toj dimenziji je 1!
  - Order - Način predstavljanja polja u memoriji, ili MPI_ORDER_C ili MPI_ORDER_FORTRAN
  - Tip svakog elementa polja
- Treba voditi računa o deljivosti N i broja procesa. Podrazumevano uzima se $\lceil N/p \rceil$

Primer 1

- 2D polje, dimenzija m x n
- 6 procesa, raspoređenih 2 x 3




```
gsizes[0] = m;
gsizes[1] = n;

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;
psizes[1] = 3;

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, "filename", 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, buf_local_array, local_array_size, MPI_FLOAT, &status);

MPI_File_close(&fh);
```

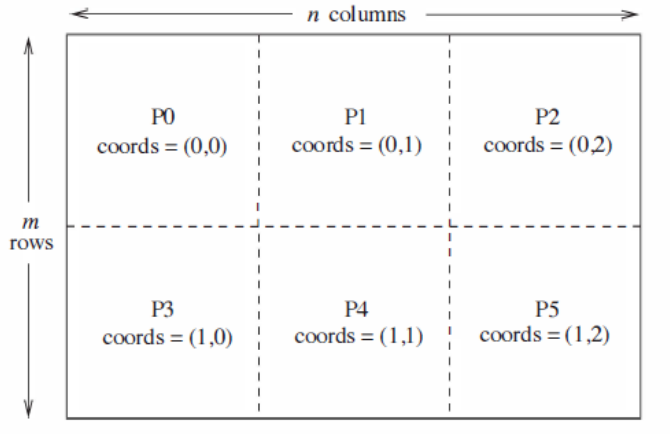
MPI_Type_create_subarray – Podsetnik

- Funkcija kreira izvedeni tip podatka definisanjem početka i dimenzija podpolja
- int **MPI_Type_create_subarray**(int ndims, int *sizes, int *subsizes, int *offsets, int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
    - ndims - broj dimenzija polja (N)(pozitivan broj)
    - sizes - broj elemenata starog tipa (oldtype) u svakoj dimenziji polja (niz pozitivnih celih brojeva)
    - subsizes - broj elemenata starog tipa (oldtype) u svakoj dimenziji podpolja(niz pozitivnih celih brojeva)
    - offsets - početne koordinate podpolja u svakoj dimenziji(niz nenegativnih brojeva)
    - order - način predstavljanja polja u memoriji, ili MPI_ORDER_C ili MPI_ORDER_FORTRAN

Primer 1 – pomoću *subarray* f-je

```
gsizes[0] = m; // dimenzije polja
gsizes[1] = n;

psizes[0] = 2; // broj procesa vertikalno
psizes[1] = 3; // broj procesa horizontalno

lsizes[0] = m/psizes[0]; // dimenzije podpolja
lsizes[1] = m/psizes[1];

dims[0] = 2; // dimenzije Cartesian virtuelne topologije
dims[1] = 3;
periods[0] = periods[1] = 1; // obe dimenzije su periodične

MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Comm rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);

start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];
```

*** Napomena: Kreiramo virtuelnu Cartesian topologiju kako bismo jednostavnije odredili koordinate svakog procesa, a zatim njih upotrebili za određivanje početnog indeksa podpolja u svakom procesu




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

MPI_File_open(MPI_COMM_WORLD, "filename", 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, buf_local_array, local_array_size, MPI_FLOAT, &status);

MPI_File_close(&fh);
```

Neblokirajući I/O i „*Split Collective*“ I/O

Neblokirajuće I/O funkcije

- Podsetnik - Motivacija za komunikaciju bez blokiranja:
  - Izbegavanje deadlock-a
  - Izbegavanje nezaposlenih procesa
  - Izbegavanje bespotrebne sinhronizacije
  - Preklapanje komunikacije i izačunavanja (korisnog posla), tj. skrivanje ”troškova komunikacije”
- MPI podržava neblokirajuće verzije funkcija svih nezavisnih I/O funkcija
- Naziv: MPI_File_ixxx –
  - int **MPI_File_iread**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, **MPI_Request* request**)
  - int **MPI_File_iwrite**(MPI_File fh, ROMIO_CONST void *buf, int count, MPI_Datatype datatype, **MPI_Request* request**)
  - int **MPI_File_iwrite_at**(MPI_File fh, MPI_Offset offset, const void *buf, int count,MPI_Datatype datatype, **MPI_Request* request**);
  - int **MPI_File_iread_at**(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, **MPI_Request* request**)

Podsetnik - Način rada neblokirajućih f-ja

- Mehanizam rada isti kao kod neblokirajućih funkcija za komunikaciju - MPI_Isend, MPI_Irecv
  - Funkcije vraćaju MPI_Request objekat
  - Na osnovu request-a se proverava (testira) status inicirane operacije ili kompletira njeno izvršenje
    - int **MPI_Wait(MPI_Request *request, MPI_Status *status);** - iz ove funkcije se proces vraća onda kada se operacija identifikovana sa request završi. Ova operacija je blokirajuća.
    - int **MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);** - funkcija vraća informaciju o trenutnom stanju operacije koja je identifikovana argumentom request. Argument flag se posatavlja na true ukoliko je operacija završena, u suprotnom na false. Argument status sadrži dodatne statusne informacije. Ova operacija nije blokirajuća.

Primer

- Primer preklapanja komunikacije i izačunavanja (korisnog posla), tj. skrivanja ”troškova komunikacije”:

```
MPI_Request request;
MPI_File_iwrite_at(fh, offset, buf, count, datatype, &request);
for (i=0; i<1000; i++)
        //neko izračunavanje
}
MPI_Wait(&request, &status);
```

Split collective I/O

- **Grupne** MPI I/O operacije obezbeđuju specifičnu formu **neblokirajućih funkcija** - *Split collective* I/O
- Kako bi se koristile grupne neblokirajuće I/O operacije, korisnik mora definisati **početak** i **kraj** operacije:
  - int **MPI_File_write_all_begin**(MPI_File fh, const void *buf, int count, MPI_Datatype datatype);
  - int **MPI_File_write_all_end**(MPI_File fh, const void *buf, MPI_Status *status);
- Ograničenje – U jednom trenutku može biti aktivna **samo jedna** split collective I/O operacija nad jednim fajlom!
- Primer:

```
MPI_File_write_all_begin(fh, buf, count, datatype);
for (i=0; i<1000; i++)
    //neko izračunavanje
}
MPI_File_write_all_end(fh, buf, &status);
```

Deljeni pokazivač na fajlove

- Do sada smo koristili individualne file pointere i pointere sa eksplicitnim pomerajem
- Dodatno, pokazivač na fajl se može deliti između procesa u okviru komunikatora koji se prosleđuje prilikom otvaranja fajla
- Funkcije za čitanje/upis sa trenutne pozicije deljenog pokazivača:
  - int **MPI_File_write_shared**(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status);
  - int **MPI_File_read_shared**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status * status);
  - int **MPI_File_seek_shared**(MPI_File fh, MPI_Offset offset, int whence);

Deljeni pokazivač na fajlove - Nastavak

- Nakon svakog poziva ovih funkcija, pozicija pokazivača se ažurira za količinu upisanih/pročitanih podataka
- Naredni poziv funkcije **bilo kog procesa** iz grupe čita/upisuje podatke na **novu poziciju** pokazivača
- Svi procesi moraju imati isti pogled na fajl (File_view)
- **MPI_File_seek_shared** se može koristiti za eksplicitno pomeranje pozicije pokazivača

Deljeni pokazivač na fajlove - Primer

- Primer – Svi procesi upisuju log podatke u log fajl, redosled upisa procesa nije važan

```
#include "mpi.h"
int main(int argc, char *argv[])
{
    int buf[1000];
    MPI_File fh;
    MPI_Init(&argc, &argv);
    MPI_File_open(MPI_COMM_WORLD, "filename", MPI_MODE_CREATE |
    MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
    MPI_File_write_shared(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
    MPI_File_close(&fh);
    MPI_Finalize();
}
```

Deljeni pokazivač na fajlove - Nastavak

- Neblokirajuća verzija funkcija sa deljenim pokazivačem
  - int **MPI_File_iwrite_shared**(MPI_File fh, ROMIO_CONST void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
  - int **MPI_File_iread_shared**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
- Grupne operacije sa deljenim pokazivačem
  - int **MPI_File_read_ordered**(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
  - int **MPI_File_write_ordered**(MPI_File fh, ROMIO_CONST void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
  - Upis i čitanje se u ovom slučaju vrše redom, po ranku procesa!

Konzistentnost

- U dva slučaja konzistetnost ne predstavlja problem:
  - Svi procesi vrše **samo čitanje** iz fajla
  - Svaki proces pristupa **posebnom fajlu**
- U slučaju da više procesa pristupa **istom fajlu,** i da ujedno **bar jedan** proces vrši **upis u fajl** – mora se voditi računa o konzistentnosti
- Prvi primer:

Process 0	Process 1
MPI_File_open(MPI_COMM_WORLD, "file", ... , &fh1) MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ...) MPI_File_read_at(fh1, 0, buf, 100, MPI_BYTE, ...)	MPI_File_open(MPI_COMM_WORLD, "file", ... , &fh2 MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ...) MPI_File_read_at(fh2, 100, buf, 100, MPI_BYTE, ...)

- Procesi pristupaju različitim delovima fajla, bez preklapanja
- MPI garantuje da proces može pročitati upisane podatke, bez dodatne sinhronizacije

Konzistentnost - Nastavak

- Drugi primer:
  - Svaki proces mora da pročita podatke upravo upisane od strane drugog procesa! Tj. pristupi dva procesa se preklapaju.
  - U ovakvom slučaju, MPI ne garantuje konzistentnost.
- Korisnik mora primeniti dodatne korake u cilju potpune konzistentnosti. Tri su opcije:

1. Postavljanje atomičnosti

Process 0	Process 1
MPI_File_open(MPI_COMM_WORLD, "file", ... , &fh1)	MPI_File_open(MPI_COMM_WORLD, "file", ... , &fh2
MPI_File_set_atomicity(fh1, 1)	MPI_File_set_atomicity(fh2, 1)
MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ...)	MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ...)
MPI_Barrier(MPI_COMM_WORLD)	MPI_Barrier(MPI_COMM_WORLD)
MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ...)	MPI_File_read_at(fh2, 0, buf, 100, MPI_BYTE, ...)

Poziv **MPI_File_set_atomicity**(MPI_File fh, int flag) pre upisa garantuje da se čitanje može obaviti odmah nakon upisa. **MPI_Barrier**(MPI_Comm comm) osigurava da će svaki proces završiti upis pre čitanja od strane drugog procesa. Posledica atomičnosti može biti pad performansi!

Konzistentnost - Nastavak

2. Zatvoriti, a zatim otvoriti fajl

Process 0	Process 1
MPI_File_open(MPI_COMM_WORLD, "file", ... , &fh1)	MPI_File_open(MPI_COMM_WORLD, "file", ... , &fh2
MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ...)	MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ...)
MPI_File_close(&fh1)	MPI_File_close(&fh2)
MPI_Barrier(MPI_COMM_WORLD)	MPI_Barrier(MPI_COMM_WORLD)
MPI_File_open(MPI_COMM_WORLD, "file", ... , &fh1)	MPI_File_open(MPI_COMM_WORLD, "file", ... , &fh2)
MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ...)	MPI_File_read_at(fh2, 0, buf, 100, MPI_BYTE, ...)

Na ovaj način, čitanje se obavlja nad drugim *file handle* objektima. Barijera se koristi iz istog razloga kao u primeru 1.

3. Obezbediti da nijedna sekvenca upisa nije konkurentna sa sekvencom upisa/čitanja drugog procesa

*Sekvencom se smatra grupa operacija između dva poziva MPI_File_open, MPI_File_close ili MPI_File_sync funkcija

- MPI garantuje da podaci upisani od strane jednog procesa mogu biti pročitani od strane drugog, ako **sekvenca upisa** jednog procesa nije konkurentna sa **bilo kojom sekvencom** drugog procesa!

3. opcija konzistentnosti - primer

Process 0	Process 1
MPI_File_open(MPI_COMM_WORLD, "file", ... , &fh1)	MPI_File_open(MPI_COMM_WORLD, "file", ... , &fh2
MPI_File_write_at(fh1, 0, buf, 100, MPI_BYTE, ...)	
MPI_File_sync(fh1)	MPI_File_sync(fh2) *(needed for collective operation)*
MPI_Barrier(MPI_COMM_WORLD)	MPI_Barrier(MPI_COMM_WORLD)
MPI_File_sync(fh1) *(needed for collective operation)*	MPI_File_sync(fh2)
	MPI_File_write_at(fh2, 100, buf, 100, MPI_BYTE, ...)
MPI_File_sync(fh1) *(needed for collective operation)*	MPI_File_sync(fh2)
MPI_Barrier(MPI_COMM_WORLD)	MPI_Barrier(MPI_COMM_WORLD)
MPI_File_sync(fh1)	MPI_File_sync(fh2) *(needed for collective operation)*
MPI_File_read_at(fh1, 100, buf, 100, MPI_BYTE, ...)	MPI_File_read_at(fh2, 0, buf, 100, MPI_BYTE, ...)
MPI_File_close(&fh1)	MPI_File_close(&fh2)

- **MPI_File_sync**(MPI_File fh) je grupna operacija, mora se pozvati iz oba procesa
- Ovakav pristup nije moguć kod grupnih operacija!

Postizanje visokih MPI I/O performansi

Kako uticati na MPI I/O performanse?

- Jednako raspoređenim podacima u fajlovima se može pristupiti na više načina
- Performanse I/O pristupa zavise od načina pristupa!
- Možemo definisati 4 „nivoa“ pristupa (*levels of access*)
- Primer:
  - 2D polje
  - 16 procesa
  - Polje je distribuirano na 16 procesa
  - Svaki kvadrat predstavlja podpolje u memoriji zasebnog procesa
  - Procesi pristupaju nekontinualnim delovima podataka

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Primer – 4 nivoa pristupa

- **Nivo 0**
  - Procesi pristupaju „svojim“ podacima u stilu Unix pristupa – po jedna nezavisna operacija čitanja za svaki red u lokalnom polju
- **Nivo 1**
  - Isti način pristupa kao kod nivoa 0, ali uz upotrebu grupnih operacija
- **Nivo 2**
  - Svaki proces kreira izvedeni tip podatka koji opisuje „šablon“ pristupa podacima, pogled na fajl (file view), a zatim poziva nezavisne I/O operacije
- **Nivo 3**
  - Isti način pristupa kao kod nivoa 2, ali uz upotrebu grupnih operacija

Primer – 4 nivoa pristupa – Nastavak

```
MPI_File_open(….,“filename“,….,&fh)
for(i = 0; i < n_local_rows; i++)
{
        MPI_File_seek(fh,…)
        MPI_File_read(fh, row[i]…)
}
MPI_File_close(&fh)
```

LEVEL 0

```
MPI_File_open(MPI_COMM_WORLD,
“filename“,….,&fh)
for(i = 0; i < n_local_rows; i++)
{
        MPI_File_seek(fh,…)
        MPI_File_read_all(fh, row[i]…)
}
MPI_File_close(&fh)
```

LEVEL 1

```
MPI_Type_create_subarray(…,&subarray,…)
MPI_Type_commit(&subarray)
MPI_File_open(….,“filename“,….,&fh)
MPI_File_set_view(fh,…,subarray,…)
MPI_File_read(fh, local_array,…)
MPI_File_close(&fh)
```

LEVEL 2

```
MPI_Type_create_subarray(…,&subarray,…)
MPI_Type_commit(&subarray)
MPI_File_open(MPI_COMM_WORLD,“filename“,
….,&fh)
MPI_File_set_view(fh,…,subarray,…)
MPI_File_read_all(fh, local_array,…)
MPI_File_close(&fh)
```

LEVEL 3

Kako postići visoke performanse

- Od 0. do 3. nivoa količina podataka **po zahtevu** raste
- Sa povećanjem količine podataka po zahtevu, **raste nivo performansi pristupa**!
- Od implementacije zavisi kako će operacije biti izvršene
- Uvek treba težiti 3. nivou pristupa, umesto 0. ili 1. nivoa!

Zaključna razmatranja

- Ključni izazovi koje MPI I/O mora adresirati:
  - Visoko kašnjenje
    - Neblokirajuće I/O operacije
    - Kooperativni pristup fajlu od strane procesa
  - Neefikasni I/O u slučaju malih i čestih pristupa fajl sistemu
    - Grupne operacije
    - Izvedeni tipovi podataka
    - Pogled na fajl (File view)