Zadnja izmjena: 17. prosinca 2015. 12:56

Osnove MPI-a

- MPI (*Message-Passing Interface*) je **specifikacija sučelja biblioteke za razmjenu poruka** između procesa
- idejno, imamo određen broj procesa (radnika) koji rješavaju problem ⇒ cilj MPI-a je definirati kako bi trebali izgledati pozivi funkcija za razmjenu poruka
- MPI je specifikacija, ne implementacija ⇒ MPI deklarira procedure (funkcije u C-u/subrutine u Fortran-u), ali ih ne implementira ⇒ postoje brojne implementacije (OpenMPI, MPICH, MSMPI, implementacije velikih proizvođača paralelnih računala HP, IBM, Intel, ...)
- osim "obične" razmjene poruka, MPI pruža i podršku za razne kolektivne operacije (redukcija, scan i sl.), dinamičko stvaranje procesa, paralelni I/O, posebne tipove podataka, itd.
- MPI programi se sastoje od autonomnih procesa ⇒ svaki izvršava svoj kod, koji može biti isti (**S**ingle **P**rogram **M**ultiple **D**ata) ili pak različit (**M**ultiple **P**rogram **M**ultiple **D**ata) ⇒ za naše potrebe dovoljan je samo prvi model
- `proces` se sastoji od barem jedne dretve, te su mu dodijeljeni programsko brojilo, adresni prostor, registri i varijable
- svaki proces izvršava svoj kod i ima vlastite varijable
- primjer rudimentarnog MPI programa je sljedeći:

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

int main(int argc, char **argv)
{
  int    size;
  int    rank;

  MPI_Init(&argc, &argv);

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

  printf("Hello world from process %d/%d\n", rank, size);

  MPI_Finalize();
  return 0;
}
```

`hello.c`

- potrebno koristiti `#include <mpi.h>` ili `#include "mpi.h"`
- pozivi svih MPI funkcija moraju se nalaziti između poziva funkcija

  `MPI_Init(char *argc, char ***argv)`

  i

  `MPI_Finalize(void)`

  - njihova svrha je alokacija i dealokacija memorije potrebne MPI-u, te inicijalizacija osnovnih MPI objekata
  - novije verzije MPI-a dozvoljavaju da se `MPI_Init` poziva s oba argumenta `NULL`
  - jedine iznimke za gornje pravilo su funkcije

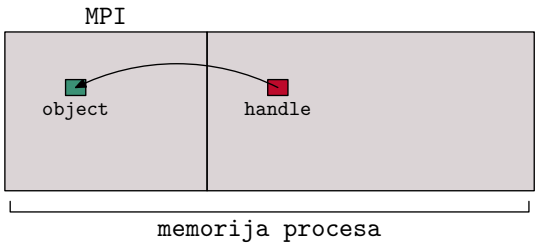
    `MPI_Get_version(int *version, int *subversion` – vraća verziju i podverziju korištenog MPI standarda

`MPI_Get_library_version(char *version, int *resultlen)` – vraća podatke o implementaciji MPI-a, pri čemu je duljina `version` najviše `MPI_MAX_LIBRARY_VERSION_STRING` − 1

`MPI_Initialized(int *flag)` – vraća istinu u `flag` ako je MPI inicijaliziran

`MPI_Finalized(int *flag)` – vraća istinu u `flag` ako je MPI finaliziran

- kod poziva funkcije `MPI_Init` MPI alocira komad memorije za vlastite strukture podataka ⇒ standard ne definira ni što su te strukture ni kako izgledaju, već samo operacije koje možemo raditi s njima, pa se takvi objekti zovu `opaque objekti` (netransparentni objekti) ⇒ na programeru je da alocira memoriju za `handle`-ove preko kojih se pristupa takvim objektima (na njih možete gledati kao na pointere, iako sam standard ne definira jesu li to pointeri, indeksi u polje ili nešto treće)



- sve MPI funkcije vraćaju odgovarajuću poruku o grešci kodiranu kao `int` ⇒ ako je posao funkcije uspješno izvršen, funkcija vraća `MPI_SUCCESS`, dok se u suprotnom program "skrši" (sve greške su, barem inicijalno, fatalne)
- funkcije

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

i

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

sastavni su dio gotovo svakog MPI programa

  - kao prvi argument primaju komunikator `comm` tipa `MPI_Comm` ⇒ to je `handle` na komunikatorski objekt, a sami komunikatori su, ukratko rečeno, objekti koji definiraju tko s kim može komunicirati ⇒ za sada je dovoljno koristiti predefinirani komunikator `MPI_COMM_WORLD` u kojem svi procesi mogu međusobno komunicirati
  - `MPI_Comm_size` vraća u `size` broj svih procesa unutar `MPI_COMM_WORLD`
  - `MPI_Comm_rank` vraća u `rank` redni broj procesa unutar `MPI_COMM_WORLD` ⇒ svaki proces ima jedinstveni `rank` $\in \{0, \ldots, \texttt{size} - 1\}$

- obično se C/C++ kompajler za MPI programe zove `mpicc`/`mpic++` ⇒ ostatak opcija je isti kao i kod `gcc`/`g++`
- primjer kompajliranja gornjeg programa je:

```
mpicc hello.c -o hello
```

- programe pokrećemo naredbom `mpirun` – ukoliko želimo pokrenuti program `hello` s 4 procesa, koristimo

```
mpirun -np 4 hello
```

- mogući ispis gornjeg programa je

```
Hello from process 2/4
Hello from process 3/4
Hello from process 1/4
Hello from process 0/4
```

- poredak ispisivanja, kao i poredak izvršavanja procedura izvan dosega MPI-a, nije definiran standardom
- svi procesi mogu pisati na `stdout` i `stderr`, dok podatke sa `stdin` prima samo proces s rangom 0 (kod ostalih procesa je preusmjeren na `/dev/null`)

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

int main(int argc, char **argv)
{
  int    rank;
  int    size;
  int    x;
  int    t;

  MPI_Init(&argc, &argv);

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

  x = 0;
  t = scanf("%d", &x);

  printf("%d: t = %d, x = %d\n", rank, t, x);


  MPI_Finalize();
  return 0;
}
```

input.c

Point-to-point komunikacija

- osnovne komunikacijske operacije su slanje (`send`) i primanje (`receive`) poruke
- za slanje poruke koristimo funkciju

```
MPI_Send(const void *sendbuf, int sendcount, MPI_Datatype datatype, int dest, int
                                tag, MPI_Comm comm)
```

  pri čemu prva tri argumenta funkcije predstavljaju pointer `sendbuf` na polje od barem `sendcount` $\geq$ 0 elemenata tipa `datatype`

- `MPI_Datatype` je također `handle` na MPI objekt koji reprezentira tip podataka $\Rightarrow$ neki od osnovnih MPI tipova i njihovi C-ovski ekvivalenti su

MPI tip	C tip
MPI_CHAR	char
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_C_DOUBLE_COMPLEX	double _Complex

- posljednja tri argumenta funkcije `MPI_Send` definiraju `message envelope`, koji sadrži polja:
  - `source` (implicitno određen)
  - `destination`
  - `tag`
  - `komunikator`

- `dest` je rang procesa kojem šaljemo poruku
- `tag` $\in \{0, \ldots, \texttt{UB}\}$, `UB` ≥ 32767, možemo koristi za razlikovanje tipova poruka – npr. jedan tip poruka mogu biti kontrolne, a drugi podatkovne poruke
- posljednji argument je komunikator $\Rightarrow$ za sada je dovoljno koristiti `MPI_COMM_WORLD`
- poruke primamo pomoću funkcije

```
MPI_Recv(void *recvbuf, int recvcount, MPI_Datatype datatype, int src, int tag,
                        MPI_Comm comm, MPI_Status *status)
```

- prva tri argumenta `recv` operacije imaju isto značenje kao i kod `send`-a
  - argument `recvcount` je duljina buffer-a `recvbuf` elemenata tipa `datatype`
  - potrebno osigurati da je `sendcount` $\leq$ `recvcount` (u suprotnom dolazi ili do kraćenja poruke ili do segmentacijske greške)
  - ako je poslana poruka duljine `sendcount` $<$ `recvcount`, zadnjih `recvcount` $-$ `sendcount` elemenata u `recvbuf` se ne mijenja
- druga tri argumenta također definiraju `message envelope`, no sada je `destination` implicitno određen
- `recv` može primiti poruku poslanu `send` operacijom ukoliko im se podudaraju `message envelope`-i
- kod `recv` operacije možemo koristiti i `MPI_ANY_SOURCE` za `src` i/ili `MPI_ANY_TAG` za `tag`
  - u tom slučaju `src` i `tag` određujemo iz argumenta `status` – `MPI_Status` je struktura (nije handle!) s elementima `MPI_SOURCE`, `MPI_TAG` i `MPI_ERROR`
  - `MPI_ERROR` se koristi samo kod funkcija koje vraćaju više statusa, u suprotnom se greška vraća kroz povratnu vrijednost funkcije (nema smisla vraćati istu informaciju više puta)
  - `MPI_Status` sadrži i informaciju o duljini primljene poruke $\Rightarrow$ do tog podatka dolazimo pomoću funkcije

    ```
    MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count)
    ```

  - ako nam nije potreban `status`, možemo koristiti `MPI_STATUS_IGNORE` (ili `MPI_STATUSES_IGNORE` kod funkcija koje vraćaju niz statusa)
- dozvoljeno koristiti `src` $=$ `dest`, no može doći do deadlock-a
- za svaku `send` operaciju mora postojati odgovarajuća `recv` operacija i obratno

```c
#include  <stdio.h>
#include  <string.h>
#include  <mpi.h>

#define   MAXLEN     100

int main(int argc, char **argv)
{
  int          rank;
  int          size;

  char         msg[MAXLEN];
  int          count;

  MPI_Status   status;


  MPI_Init(&argc, &argv);

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

  if (rank == 0) {
    for (int i = 1; i < size; ++i) {
      MPI_Recv(msg, MAXLEN, MPI_CHAR, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
      MPI_Get_count(&status, MPI_CHAR, &count);

      printf("Message from %d: \"%s\" (len = %d)\n", status.MPI_SOURCE, msg, count);
    }
  } else {
    sprintf(msg, "Hello world from %d", rank);
    MPI_Send(msg, strlen(msg) + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}
```

hello_comm.c

- u gornjem primjeru svaki proces izuzev procesa 0 generira svoju poruku i šalje ju procesu 0, dok proces 0 prima poruke u proizvoljnom poretku (zbog `MPI_ANY_SOURCE`), te ih ispisuje
- mogući ispis gornjeg programa pokrenutog s 6 procesa je

```
Message from 1: "Hello world from 1" (len = 19)
Message from 3: "Hello world from 3" (len = 19)
Message from 4: "Hello world from 4" (len = 19)
Message from 2: "Hello world from 2" (len = 19)
Message from 5: "Hello world from 5" (len = 19)
```

Tipovi komunikacije

- ukoliko pokrenemo sljedeći program

```c
#include  <stdio.h>
#include  <unistd.h>
#include  <mpi.h>

#define   MAXLEN    100

int main(int argc, char **argv)
{
  int   rank;
  int   size;

  int   x[MAXLEN];

  MPI_Init(&argc, &argv);

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

  if (size != 2)
    goto die;

  if (rank == 0) {
    printf("%d: send started\n", rank);
    MPI_Send(x, MAXLEN, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("%d: send finished\n", rank);
  } else {
    sleep(5);
    printf("%d: recv started\n", rank);
    MPI_Recv(x, MAXLEN, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("%d: recv finished\n", rank);
  }

die:
  MPI_Finalize();
  return 0;
}
```

send.c

s dva procesa dobit ćemo ispis:

```
0: send started
0: send finished
1: recv started
1: recv finished
```

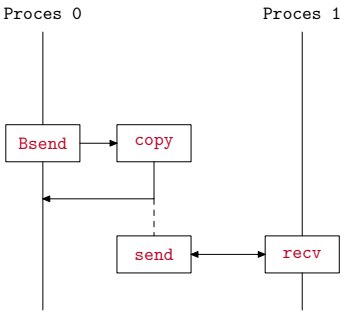
pri čemu će između ispisa procesa 0 i procesa 1 nastupiti pauza od 5s

- u ovom slučaju su sadržaj niza `x`, te `message envelope` kopirani u MPI-ev interni buffer kako proces 0 ne bi trošio vrijeme na čekanje
- kada bismo `MAXLEN` promijenili na veći broj (npr. 10000), ishod bi bio drugačiji ⇒ prva linija bi se ispisala, potom bi nastupila pauza od 5s i na kraju se ispisale preostale tri linije teksta ⇒ u ovom slučaju je sadržaj koji šaljemo prevelik za MPI-ev buffer, pa je potrebno čekati dok se ne pojavi odgovarajući poziv `MPI_Recv`
- u oba slučaja koristi se `blokirajući send` ⇒ poziv funkcije `MPI_Send` je gotov tek kada ponovo možemo koristiti podatke u nizu `x`
- ako želimo eksplicitno koristiti kopiranje u MPI-evu internu memoriju koristimo `buffered send`

```
MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
          MPI_Comm comm)
```

  - ukoliko nam je potrebno više memorije nego što to MPI dopušta, možemo MPI-u dati unaprijed alocirani komad memorije na korištenje
  - nakon što smo alocirali memoriju za `buffer`, dodijeljujemo ga MPI-u na sljedeći način

```
MPI_Buffer_attach(void *buffer, int size)
```



pri čemu je `buffer` pointer na niz od `size` byte-a

- moguće je koristiti samo jedan buffer po procesu u nekom trenutku
- na kraju je potrebno "odvojiti" MPI od buffera pozivom funkcije

```
MPI_Detach_buffer(void *buffer_addr, int *size)
```

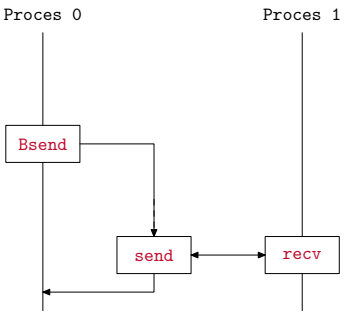
pri čemu funkcija u `buffer_addr` vraća pointer na prethodno dodijeljenji buffer, a u `size` njegovu veličinu

- s jedne strane, bufferiranje je dobro jer poziv `send` funkcije završava čim je sadržaj poruke kopiran, te pozivajući proces može nastaviti s radom; s druge strane, dolazi do potencijalno dugotrajnog kopiranja podataka iz `sendbuf` u buffer
- ovakvo slanje je `lokalna operacija` ⇒ izvršavanje funkcije `MPI_Send` završava čim se svi potrebni podaci kopiraju u buffer, tj. neovisno o tome kada proces kojem šaljemo poruku pozove `MPI_Recv`

- u slučaju da ne želimo koristiti nikakvo bufferiranje, već započeti transfer podataka tek kada je pozvan odgovarajući `MPI_Recv`, koristimo `synchronous send`

```
MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
          MPI_Comm comm)
```

- osim slanja poruke daje nam i informaciju da je proces kojem šaljemo poruku došao do određene točke u izvršavanju (jer je pozvao `MPI_Recv`)
- ovakvo slanje `nije lokalna operacija` ⇒ završavanje ovisi o drugim procesima



- običan `MPI_Send` je tzv. `standard send` i on je kombinacija navedenih načina slanja ⇒ na implementaciji leži odluka koji će način slanja koristiti u konkretnom slučaju
  - ova operacija `nije lokalna` ⇒ generalno može, ali i ne mora ovisiti o drugim procesima
- postoji i treći način slanja poruke: `ready send` ⇒ koristi se samo u slučajevima kada znamo da je odgovarajući `MPI_Recv` pozvan

```
MPI_Rsend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
                                                    MPI_Comm comm)
```

  - pogrešno ga je pozvati ukoliko odgovarajući `MPI_Recv` nije pozvan
  - brži je od ostalih načina slanja, jer izbjegava *hand-shake* operacije
  - u većini implementacija je implementiran kao standardni send ⇒ nema poboljšanja performansi
  - operacije `nije lokalna`

- postoji samo jedna `recv` operacija ⇒ do izlaska iz poziva funkcije dolazi tek kada je primljena cjelokupna poruka

Neblokirajuća komunikacija

- moguće je poboljšati performanse MPI aplikacije ukoliko se računanje i komunikacija odvijaju istovremeno ⇒ s jedne strane možemo koristiti dretve, dok nam s druge strane MPI pruža jednostavniji mehanizam `neblokirajućih procedura`
- poziv neblokirajućih funkcija započinje samu operaciju, ali ju ne završava ⇒ potrebni su pozivi drugih funkcija kako bismo odredili je li započeta operacija dovršena
- nakon poziva neblokirajuće funkcije, MPI u pozadini obavlja operaciju
- neblokirajući analogoni `MPI_Send` i `MPI_Recv` su (I stoji za "immediate", jer do povratka iz funkcije dolazi gotovo pa trenutno)

```
MPI_Isend(const void *sendbuf, int sendcount, MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm, MPI_Request *request)
```

  i

```
MPI_Irecv(const void *recvbuf, int recvcount, MPI_Datatype datatype, int src, int
                    tag, MPI_Comm comm, MPI_Request *request)
```

  - značenje svih argumenata je isto kao i kod blokirajućih procedura, osim posljednjeg pointera na `handle` na netransparentni `request objekt` ⇒ sve neblokirajuće procedure alociraju prostor za `request objekt` i asociraju ga s handle-om `request` ⇒ koristimo ga kod provjere je li operacija završena

- započinjanje neblokirajućeg slanja indicira da MPI može kopirati podatke iz `sendbuf`, dok poziv neblokirajućem primanju indicira da MPI moće početi pisati u `recvbuf`
- sam poziv neblokirajuće funkcije pokreće zahtjev za odgovarajućom operacijom, a postoje tri vrste završetka:

  `završetak operacije` – možemo ponovo koristiti resurse i svi izlazni bufferi sadrže nove vrijednosti

  `završetak zahtjeva` – događa se kod poziva neke od procedura za čekanje ili testiranje završetka operacije

  `završetak komunikacije` – sve operacije vezane uz slanje i primanje podataka su dovršene

- za vrijeme trajanja neblokirajućeg slanja zabranjeno je mijenjati sadržaj `sendbuf`, dok je za vrijeme neblokirajućeg primanja zabranjeno pristupati `recvbuf`
- i ovdje imamo 4 modova slanja: `standard`, `synchronous`, `ready` i `buffered send` ⇒ svi su `lokalni` pozivi, a jedina razlika spram blokirajućih rutina je posljedni argument, pointer na handle tipa `MPI_Request`

```
MPI_Irsend(const void* sendbuf, int sendcount, MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm, MPI_Request *request)
MPI_Ibsend(const void* sendbuf, int sendcount, MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm, MPI_Request *request)
MPI_Issend(const void* sendbuf, int sendcount, MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm, MPI_Request *request)
```

- dvije osnovne funkcije za provjeru završetka neblokirajućih operacija su `MPI_Wait` i `MPI_Test`

```
MPI_Wait(MPI_Request *request, MPI_Status *status)
```

  - čeka dok operacije vezana uz `request` nije završena ⇒ pri izlasku iz rutine, `request objekt` na kojeg pokazuje handle `request` je dealociran, a vrijednost `request` postavljena na null-handle `MPI_REQUEST_NULL`
  - u `status` vraća status završene operacije ⇒ korisno kod `recv` operacije, kod `send` operacije vrijednost generalno nije definirana
  - nije lokalna operacija
  - dozvoljeno koristiti `request` postavljen `MPI_REQUEST_NULL` ⇒ u tom slučaju je `status` postavljen na vrijednost `praznog statusa`
    - prazan status ima polja `tag = MPI_ANY_TAG`, `source = MPI_ANY_SOURCE`, `error = MPI_SUCCESS` i konfiguriran je tako da svaki poziv funkciji `MPI_Get_count` vraća `count = 0`

```
MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

  - vraća `flag = true` ukoliko je operacije vezana uz `request` završena i postavlja odgovarajući status, te pritom dealocira `request objekt` i postavlje `request` na `null handle`
  - u suprotnom vraća `flag = false`, a vrijednost `status` nije definirana

- `request objekt` se može dealocirati bez čekanja na završetak operacije pomoću funkcije

```
MPI_Request_free(MPI_Request *request)
```

  - `request` se pritom postavlja na `MPI_REQUEST_NULL`
  - sam `request objekt` se dealocira kada se dovrši operacija
  - generalno se koristi kod operacija vezanih uz slanje ⇒ kod `recv` operacija ne bismo znali kada smo primili poruku

- ponekad je korisno čekati ili provjeriti završetak jedne, nekoliko ili svih operacija u nizu

```
MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status
                                    *status)
```

  - procedura blokira izvršavanje programa dok jedna od `count` operacija asociranih sa zahtjevima u `array_of_requests` nije završena
  - ako je više od jedne operacije završeno, jedna od njih je proizvoljno izabrana
  - vraća u `index` indeks zahtjeva koji je završen, a u `status` njegov status
  - pri izlasku je `request objekt` asociran s `array_of_requests[index]` dealociran, a sam handle postavljen na vrijednost `MPI_REQUEST_NULL`
  - ukoliko su svi elementi niza `array_of_requests` null handle-i, funkcije vraća `MPI_UNDEFINED` u `index`

```
MPI_Testany(int count, MPI_Request array_of_requests[], int *index, int *flag,
                                MPI_Status *status)
```

- provjerava je li ikoji od `count` zahtjeva u `array_of_requests` završio
- ako je ijedan završio, vraća `flag = true` i njegov indeks u `index`, te status u `status`
- završeni `request objekt` je dealociran, a njegov handle postavljen na `MPI_REQUEST_NULL`
- ako ni jedan zahtjev nije dovršen, vraća `flag = false` i `MPI_UNDEFINED` u `index`

```
MPI_Waitsome(int incount, MPI_Request array_of_requests[], int *outcount, int
             array_of_indices[], MPI_Status array_of_statuses[])
```

- blokira izvršavanje programa dok barem jedan od `count` zahtjeva u `array_of_requests` nije završen
- vraća u prvih `outcount` elemenata niza `array_of_indices` indekse svih završenih zahtjeva, a u `statuses` njihove statuse, pri čemu su `request objekti` dealocirani, a odgovarajući handle-i postavljeni na `MPI_REQUEST_NULL`
- ukoliko niti jedan od zahtjeva nije aktivan, vraća `outcount = MPI_UNDEFINED`

```
MPI_Testsome(int incout, MPI_Request array_of_requests[], int *outcount, int
                              array_of_indices[])
     MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status
                              array_of_statuses[])
 MPI_Testall(int count, MPI_Request array_of_requests, int *flag, MPI_Status
                              array_of_statuses[])
```

- pravila su slična kao i kod `MPI_Waitsome`, samo što `MPI_Waitall` i `MPI_Testall` provjeravaju jesu li svi zahtjevi dovršeni

- na sve blokirajuće `send` operacije moguće je odgovoriti neblokirajućim `recv` operacijama i obratno (dakle, imamo sveukupno 16 mogućih `send-recv` kombinacija)

Semantika point-to-point komunikacije

- MPI garantira neka općenita svojstva point-to-point komunikacije

  UREĐAJ

  - ako pošiljatelj pošalje dvije poruke na istu destinaciju i obje odgovaraju istoj `recv` operaciji, operacija ne može primit drugu poruku ako prva još nije primljena
  - ako primatelj prima dvije poruke od istog pošiljatelja i obje odgovaraju poslanoj poruci, tada druga `recv` operacija ne može biti ispunjena ako prva još uvijek čeka
  - ovakav pristup garantira da je slanje i primanje poruka deterministično, ukoliko se koristi jedna dretva po procesu i ako se ne koristi `MPI_ANY_SOURCE`
  - ukoliko koristimo više dretvi (objašnjeno u kasnijim poglavljima), ne postoji uređaj između poziva funkcija $\Rightarrow$ ti pozivi su konkurentni, iako možda fizički jedan prethodi drugome

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

int main(int argc, char **argv)
{
  int    rank;
  int    size;

  int    buf1;
  int    buf2;

  MPI_Init(&argc, &argv);

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

  if (size != 2)
    goto die;
```

```
19
20    if (rank == 0) {
21      buf1 = 5;
22      buf2 = 8;
23
24      MPI_Send(&buf1, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
25      MPI_Send(&buf2, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
26    } else {
27      MPI_Recv(&buf1, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
28      MPI_Recv(&buf2, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
29
30      printf("buf1: %d\n", buf1);
31      printf("buf2: %d\n", buf2);
32    }
33
34  die:
35    MPI_Finalize();
36    return 0;
37  }
```

order.c

- u gornjem primjeru je poruka poslana s prvim `MPI_Send` primljena s prvim `MPI_Recv`, a poruka poslana s drugim `MPI_Send` s drugim `MPI_Recv`
- neblokirajuće operacije su uređene s obzirom na poredak poziva procedura koje iniciraju komunikaciju

PROGRES

- ako je pozvan odgovarajući `send`-`recv` par na dva procesa, tada će se barem jedna od te dvije operacije završiti
- `send` operacija će završiti, osim ako `recv` nije zadovoljen nekom drugom porukom, a stoga i završena
- `recv` operacija će završiti, osim ako poslana poruka nije primljena drugom `recv` operacijom pozvanom na istom procesu

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

int main(int argc, char **argv)
{
  int    rank;
  int    size;

  int    buf1;
  int    buf2;

  MPI_Init(&argc, &argv);

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

  if (size != 2)
    goto die;

  if (rank == 0) {
    buf1 = 5;
    buf2 = 8;

    MPI_Bsend(&buf1, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Ssend(&buf2, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
  } else {
    MPI_Recv(&buf1, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&buf2, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    printf("buf1: %d\n", buf1);
    printf("buf2: %d\n", buf2);
  }

die:
  MPI_Finalize();
  return 0;
}
```

progressc

- u gornjem primjeru proces 0 prvo poziva `bufferirani send`, pri čemu će se sadržaj `buf1` biti kopiran u buffer, a potom završen neovisno o stanju procesa 1 (ukoliko je buffer premalen, doći će do greške)
- potom će proces 0 pozvati drugu `send` funkciju koja odgovara prvoj `recv` operaciji procesa 1 (obratite pozornost na tag-ove), te će obje biti završene
- na kraju proces 1 poziva drugu `recv` operaciju koja može primiti bufferiranu poruku od procesa 0
- primijetite da su poruke primljene u suprotnom poretku od onog u kojem su poslane

PRAVEDNOST

- MPI ne garantira pravednost u komunikaciji
- ako je pozvana `send` operacija, moguće je da se na `dest` procesu stalno poziva odgovarajuća `recv` operacija, no da poruka ni u jednom slučaju nije primljena, jer je svaki put primljena neka druga odgovarajuća poruka
- ako je pozvana `recv` operacija u višedretvenom procesu, moguće je da proces stalno prima odgovarajuće poruke, no `recv` operacija nikad nije zadovoljena

LIMITIRANOST RESURSA

- svaka komunikacijska operacija koja čeka na odgovor nekog drugog procesa troši sistemske resurse koji su ograničeni $\Rightarrow$ može doći do greške ukoliko nedostatak resursa sprječava izvršavanje operacije
- program je `siguran` ako nikakvo bufferiranje poruka nije potrebno kako bi se program izvršio

Tipovi podataka

- želimo slati poruke koje sadrže vrijednosti različitog tipa, i/ili vrijednosti koje nisu uzastopno smještene u memoriju
- moguće rješenje je kopirati sve podatke u uzastopne buffere, poslati ih, te razmjestiti na odgovarajuće lokacije nakon slanja ⇒ zahtjeva dodatna kopiranja između memorijskih lokacija na obje strane
- MPI pruža mehanizam za stvaranje općenitih tipova podataka koji su sastavljeni od različitih osnovnih tipova podataka na ne nužno uzastopnim lokacijama ⇒ na implementaciji je da odluči hoće li ih kopirati u buffere ili će podaci biti sakupljeni sa svojih originalnih memorijskih lokacija pri samom slanju
- tipovi koji nisu osnovni (`basic datatype`) zovu se izvedeni tipovi (`derived datatype`)
- općeniti tip podataka (`general datatype`) je netransparentan objekt koji specificira:
  1. niz osnovnih tipova podataka
  2. niz cjelobrojnih pomaka (u byteovima)
- pomaci ne moraju biti pozitivni, različiti ni u rastućem poretku
- takav niz parova zovemo `type map`

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

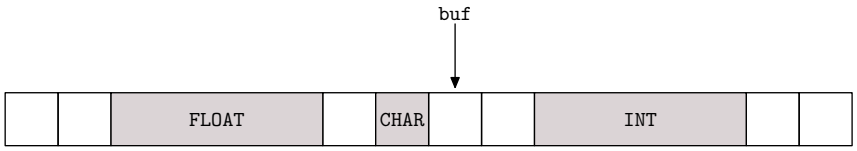
- niz osnovnih tipova (bez pomaka) zovemo `type signature` tipa podataka

$$Typesig = \{type_0, \ldots, type_{n-1}\}$$

- `type map` zajedno s (`void`) pointerom `buf` specificira komunikacijski buffer koji se sastoji od n elemenata, pri čemu se i-ti element nalazi na adresi $buf + disp_i$ i ima tip $type_i$
- poruka sastavljena iz takvog komunikacijskog buffera sastoji se od n vrijednosti tipova definiranih s $Typesig$ ⇒ npr. ako imamo tip s

$$Typemap = \{(\texttt{char}, -1), (\texttt{int}, 2), (\texttt{float}, -6)\}$$

poziv funkcije `MPI_Send(buf, 1, tip, ...)` poslat će podatke prikazane na slici i to u poretku



danom s $Typesig$

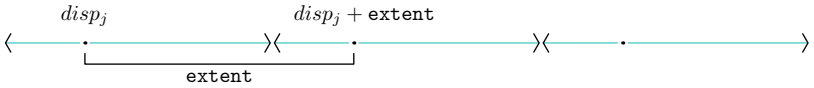
- osnovni tipovi su specijalni slučajevi općenitih tipova podataka i predefinirani su – npr. `MPI_INT` je predefinirani handle za tip podataka s `type map`-om $\{(\texttt{int}, 0)\}$
- raspon (`extent`) tipa je raspon između prvog i zadnjeg byte-a kojeg okupiraju dijelovi tipa zaokružen kako bi se zadovoljili zahtjevi za alignement podataka u memoriji
  - osnovni tipovi podataka većinom moraju biti smješteni na adrese koje su djeljive njihovom veličinom
  - npr. `double` veličine 8 byte-a mora biti smješten na adrese djeljive s 8

– zbog istog razloga će struktura koja se sastoji od jednog elementa tipa `double` i jednog elementa tipa `char` imati veličinu 16 bytea, dok u stvarnosti zauzima samo 9 bytea ⇒ u suprotnom bi došlo do krivog smještaja elemenata kod nizova

– za gore definirani $Typemap$ vrijedi

$$\begin{aligned} \text{lb}(Typemap) &= \min_j disp_j \\ \text{ub}(Typemap) &= \max_j(disp_j + sizeof(type_j)) + \varepsilon \\ \text{extent}(Typemap) &= \text{ub}(Typemap) - \text{lb}(Typemap) \end{aligned}$$

– ako $type_j$ mora biti smješten na adresu koja je višekratnik od k_j, tada `buf` + $disp_j$ i `buf` + $disp_j$ + `extent` moraju biti djeljivi s k_j, iz čega slijedi da `extent` mora biti djeljiv sa svim k_j ⇒ kako su k_j potencije od 2, dovoljno je zahtjevati da je `extent` djeljiv s $\max_j k_j$ ⇒ zato nam je potreban ε



- handle-ovi na MPI-eve tipove podataka su tipa `MPI_Datatype`

Konstruktori tipova podataka

- `MPI_Type_commit(MPI_Datatype *datatype)`
  – nakon što konstruiramo općeniti tip podataka koristeći neku od niže navednih rutina, a prije nego ga koristimo u komunikaciji, potrebno je MPI "obavijestiti" da taj tip planiramo koristiti u komunikaciji ⇒ za to služi `commit` operacija
  – moguće ju je pozvati više puta za isti tip ⇒ u svim naknadnim pozivima, funkcija ne radi ništa
  – osnovne tipove nije potrebno commitati ⇒ oni su precommitani
- `MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)`
  – duplicira postojeći tip `oldtype` u `newtype` sa svim svojstvima
  – ukoliko je `oldtype` bio commitan, automatski će biti i `newtype`
- `MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)`
  – stvara novi tip `newtype` repliciranjem tipa `oldtype` na `count` uzastopnih lokacija
  – ako je `oldtype` imao raspon ex i typemap $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$, tada `newtype` ima typemap s `count` $\cdot\, n$ elemenata

$$\begin{aligned} \{&(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1}), (type_0, disp_0 + 1 \cdot ex), \\ &\ldots, (type_{n-1}, disp_{n-1} + 1 \cdot ex), \ldots, (type_0, disp_0 + (\texttt{count} - 1) \cdot ex)\} \end{aligned}$$

- `MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`
  – vraća u `newtype` novi tip podataka građen od `count` blokova duljine `blocklength` tipa `oldtype`, pri čemu je `stride` $\cdot$ extent(`oldtype`) razmak između početaka blokova
  – `stride` može biti i negativan
  – ako `oldtype` ima raspon ex i typemap $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$, te neka je `bl` `blocklength`, tada `newtype` ima `count` $\cdot$ `bl` $\cdot\, n$ elemenata:

- `MPI_Type_indexed(int count, const int array_of_blocklengths[], const int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)`
  - vraća u `newtype` novi tip podataka građen od `count` blokova elemenata `oldtype`, pri čemu blok `i` sadrži `array_of_blocklengths[i]` elemenata i počinje na `array_of_displacements[i]` · extent(*oldtype*)
- `MPI_Type_indexed_block(int count, int blocklength, const int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)`
  - ima istu funkcionalnost kao i `MPI_Type_indexed`, samo su svi blokovi konstantne duljine `blocklength`
- `MPI_Type_create_struct(int count, const int array_of_blocklengths[], const MPI_Aint array_of_displacements[], const MPI_Datatype array_of_types[], MPI_Datatype *newtype)`
  - vraća u `newtype` novi tip podataka koji se sastoji od `count` blokova, pri čemu se blok `i` sastoji od `array_of_blocklengths[i]` elemenata tipa `array_of_types[i]`, počevši od (relativne) adrese `array_of_displacements[i]`
  - `MPI_Aint` je predefinirani tip dovoljno velik da može sadržavati sve adrese (nije handle na MPI-ev tip podataka)
  - moguće je zadati `array_of_displacements` s apsolutnim adresama ⇒ tada kod slanja/primanja umjesto `buf` koristimo predefiniranu konstantu `MPI_BOTTOM` ⇒ kako `&` operator u C-u vraća pointer koji ne mora biti apsolutna adresa (iako najčešće jest), adresu `addr` lokacije `loc` u memoriji dobivamo pomoću funkcije `MPI_Get_address(const void *loc, MPI_Aint *addr)`

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

int main(int argc, char **argv)
{
  int           rank;
  int           size;
  MPI_Datatype  type;

  int           x;
  double        y;
  char          z;

  int           sizes[3] = {1, 1, 1};
  MPI_Datatype  types[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
  MPI_Aint      displ[3];

  MPI_Init(&argc, &argv);

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

  if (size != 2)
    goto die;

  MPI_Get_address(&x, &displ[0]);
  MPI_Get_address(&y, &displ[1]);
  MPI_Get_address(&z, &displ[2]);

  MPI_Type_create_struct(3, sizes, displ, types, &type);
  MPI_Type_commit(&type);

  if (rank == 0) {
    x = 5;
    y = 7.8;
    z = 'g';

    MPI_Send(MPI_BOTTOM, 1, type, 1, 0, MPI_COMM_WORLD);
  } else {
    MPI_Recv(MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    printf("x = %d\n", x);
    printf("y = %lg\n", y);
    printf("z = %c\n", z);
  }

  MPI_Type_free(&type);

die:
  MPI_Finalize();
  return 0;
}
```

address.c

Destruktori tipova

- `MPI_Type_free(MPI_Datatype *datatype)`
  - označava netransparentan objekt vezan uz `datatype` za dealokaciju i postavlja `datatype` na `MPI_DATATYPE_NULL`
  - sva komunikacija koja koristi `datatype` normalno završava
  - dealokacija ne utječe na druge tipove konstruirane iz `datatype`
  - greška je pozvati funkciju s `datatype` koji nije valjani tip

Raspon i granice tipova

- ponekad je potrebno eksplicitno definirati donju granicu lb i gornju granicu ub type map-a ⇒ omogućuje tipove s rupama na početku i kraju ili pak tipove čiji elementi leže izvan granica
- za tu svrhu uvodimo dva konceptualna tipa podataka, `lb_marker` i `ub_marker`, koji ne zauzimaju nikakav prostor (extent(`lb_marker`) = extent(`ub_marker`) = 0) i ne utječu na veličinu tipa poda-

taka niti sadržaj poruke kreirane s tim tipom, ali utječu na definiciju raspona tipa, a s tim i na njegovu replikaciju u konstruktorima tipova

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

$$\text{lb}(Typemap) = \begin{cases} \min_j disp_j & type_j \neq \texttt{lb\_marker}, \forall j \\ \min_j \{disp_j \mid type_j = \texttt{lb\_marker}\} & \text{inače} \end{cases}$$

i

$$\text{ub}(Typemap) = \begin{cases} \max_j (disp_j + sizeof(type_j)) + \varepsilon & type_j \neq \texttt{ub\_marker}, \forall j \\ \max_j \{disp_j \mid type_j = \texttt{lb\_marker}\} & \text{inače} \end{cases}$$

pa je tada

$$\text{extent}(Typemap) = \text{ub}(Typemap) - \text{lb}(Typemap)$$

- "lažni" raspon se koristi kod komunikacijskih operacija s `count` > 1 i konstrukcije tipova
- `MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)`
  - ako je type map tipa `datatype` $Typemap$, vraća u `lb` $\text{lb}(Typemap)$, a u `extent` $\text{extent}(Typemap)$
- `MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype)`
  - vraća u `newtype` handle na novi tip podataka koji je identičan `oldtype`, samo što mu je donja granica postavljena na `lb`, a gornja granica na `lb + extent`
  - svi postojeći `lb_marker` i `ub_marker` su izbrisani i novi odgovarajući par je postavljen
- postoji i "pravi" raspon (`true extent`) tipova $\Rightarrow$ ako je $Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$, definiramo

$$\text{true\_lb}(Typemap) = \min_j \{disp_j \mid type_j \neq \texttt{lb\_marker}\}$$

i

$$\text{true\_ub}(Typemap) = \max_j (disp_j + sizeof(type_j))$$

i tada je

$$\text{true\_extent}(Typemap) = \text{true\_ub}(Typemap) - \text{true\_lb}(Typemap)$$

- "pravi" raspon je minimalan broj byteova u memoriji potrebnih kako bi se pohranio tip podataka (nekompresirano)
- `MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint *true_extent)`
  - ako je type map tipa `datatype` $Typemap$, vraća u `true_lb` $\text{true\_lb}(Typemap)$, a u `true_extent` $\text{true\_extent}(Typemap)$

Tipovi podataka u komunikaciji

- transfer podataka s jednog procesa na drugi možemo podijeliti na tri faze:
  1. podaci su uzeti iz `sendbuf` i poruka je sastavljena
  2. poruka je prenesena od pošiljatelja do primatelja
  3. podaci iz dolazne poruke su smješteni u `recvbuf`
- podudaranje osnovnih tipova moramo promatrati u sve tri faze $\Rightarrow$ imamo podudaranje između tipova u jeziku u kojem pišemo aplikaciju s onima koji se koriste u komunikacijskim operacijama, te podudaranje tipova između pošiljatelja i primatelja poruke
- u prvoj i trećoj fazi, tip varijabli jezika se podudara s tipom varijabli u komunikaciji ukoliko ime drugoga odgovara osnovnom tipu u jeziku $\Rightarrow$ npr. `MPI_INTEGER` odgovara C-ovskom `int`-u

- iznimka je `MPI_BYTE` koji odgovara bilo kojem byte-u memorije, neovisno o tipu kojem taj byte pripada
- u drugoj fazi, tipovi `send`-a i `recv`-a se podudaraju ukoliko imaju identična imena ⇒ `MPI_INTEGER` odgovara `MPI_INTEGER`
- kako bi MPI trebao podržavati heterogene arhitekture, kod slanja/primanja podatak bi trebalo doći do konverzija:

  `konverzija tipa` – mijenja tip vrijednosti, npr. zaokružuje `float` u `int`

  `konverzija reprezentacije` – mijenja binarnu reprezentaciju vrijednosti, npr. ASCII u EBCDIC za `char`
- gornja pravila podudaranja tipova osiguravaju da nikada ne dolazi do konverzije tipa, dok s druge strane MPI zahtjeva da se konverzija reprezentacije provede ukoliko je potrebno (npr. kod `MPI_BYTE` nikada ne dolazi do konverzije)
- konverzija podataka se primijenjuje i na `message envelope` ⇒ `source`, `dest` i `tag` su tipa `int`

- općenite tipove također možemo koristiti u komunikciji
- poziv `MPI_Send(buf, count, datatype, ...` s `count` > 1 se interpretira kao da je pozivu dan novi tip koji je konkatenacija `count` kopija tipa `datatype`, tj. kao

```
MPI_Type_contiguous(count,datatype,&newtype);
MPI_Type_commit(&newtype);
MPI_Send(buf,1,newtype,...);
MPI_Type_free(&newtype);
```

- neka je `datatype` tip s $Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$ i rasponom $extent$, te bez eksplicitno navedenih markera donje i gornje granice
- poziv funkcije `MPI_Send(sendbuf, count, datatype, ...)` šalje $n \cdot$ `count` elemenata, pri čemu se element $i \cdot n + j$ nalazi na lokaciji $addr_{i,j} = sendbuf + extent \cdot i + disp_j$, $i = 0, \ldots,$ `count` $- 1$, $j = 0, \ldots, n-1$ ⇒ sami elementi ne moraju biti niti uzastopni niti različiti
- varijabla na adresi $addr_{i,j}$ u pozivajućem programu mora biti tipa $type_j$ u skladu sa zahtjevima za podudaranje tipova
- poziv funkcije `MPI_Recv(recvbuf, count, datatype, ...)` prima najviše $n \cdot$ `count` elemenata, pri čemu se element $i \cdot n + j$ nalazi na lokaciji $addr_{i,j} = recvbuf + extent \cdot i + disp_j$, $i = 0, \ldots,$ `count` $- 1$, $j = 0, \ldots, n-1$ ⇒ ako je primljeno k elemenata, mora vrijedi $k \leq n \cdot$ `count`, a element $i \cdot n + j$ mora imati tip $type_j$
- k ne mora biti višekratnik od n ⇒ broj primljenih elemenata možemo saznati iz statusa
- `MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype, int *count)`
  - vraća u `count` broj primljenih elemenata `recv` operacijom pozvanom s tipom `datatype` i statusom `status`
  - prije definirana funkcija `MPI_Get_count` vraća broj primljenih `datatype` elemenata ⇒ ako je primljeno sveukupno $k \cdot n$ osnovnih elemenata, vraća k, u suprotnom `MPI_UNDEFINED`
- podudaranje općenitih tipova definira se s obzirom na podudaranje nizova osnovnih tipova ⇒ ne ovisi o pomacima ili korištenim intermedijarnim tipovima
- tip podataka može sadržavati elemente koji se preklapaju ⇒ korištenje takvih tipova u `recv` operaciji je pogrešno, čak i onda kada je poruka prekratka da bi pisala po već korištenim memorijskim lokacijama

Grupe, konteksti i komunikatori

- jedan od osnovnih ciljeva MPI-a je razvoj paralelnih biblioteka, koji zahtjeva:
  - siguran komunikacijski prostor bez konflikata van biblioteke
  - domenu grupe za kolektivne operacije ⇒ izbjegavanje nepotrebne sinkronizacije s procesima koji ne sudjeluju u operacijama
  - abstraktno označavanje procesa ⇒ biblioteke mogu koristiti vlastitu komunikaciju s obzirom na svoje strukture podataka i algoritme
- odgovarajući koncepti koje nam pruža MPI su:
  - `komunikacijski konteksti`
  - `procesne grupe`
  - `virtualne topologije`
  - `komunikatori`
- `komunikatori` enkapsuliraju sve navedene ideje kako bi pružali odgovarajući doseg za sve komunikacijske operacije u MPI-u

Procesne grupe

- sve MPI programe pokrećemo s određeni brojem procesa
  - svi mogu međusobno komunicirati preko `MPI_COMM_WORLD` ⇒ definira jednu grupu
  - svaki proces je dobio jedinstveni `rank` $\in \{0, \ldots, n-1\}$, pri čemu je `n` broj procesa s kojim smo pozvali `mpirun`
- `procesna grupa` je uređen skup procesnih identifikatora
- koristi se unutar komunikatora kako bi se opisali sudionici u komunikacijskom "univerzumu", te rangirali svi sudionici ⇒ ukoliko nisu pridružene komunikatoru nemaju nikakav utjecaja na komunikaciju
- mogu se manipulirati neovisno o komunikatorima, ali samo komunikatori mogu sudjelovati u komunikaciji
- grupa daje svakom procesu jedinstven identifikator – `rang` ⇒ koristi se za točkovnu komunikaciju, ali grupa definira i raspon kolektivne komunikacije
- neovisno o MPI-u svaki proces ima svoj `procesni ID (PID)` ⇒ jednostavnije je i portabilnije je koristiti linearne rangove
- grupe se mogu implementirati kao translacijske tablice iz rangova u PID-ove
- reprezentiratni netransparentnim `group objektom` ⇒ ne mogu se direktno prenijeti s jednog procesa na drugi
- predefinirana grupa `MPI_GROUP_EMPTY`
  - nema članova
  - može se koristiti kao valjan argument
- predefinirana konstanta `MPI_GROUP_NULL`
  - koristi se kao neispravna grupa
  - povratna vrijednost kod oslobađanja grupe

Akcesori grupa

- `MPI_Group_size(MPI_Group group, int *size)`
  - vraća u `size` broj procesa u grupi `group`
- `MPI_Group_rank(MPI_Group group, int *rank)`
  - vraća u `rank` rang pozivajućeg procesa u grupi `group` ili `MPI_UNDEFINED` ukoliko proces nije član grupe
- `MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[], MPI_Group group2, int ranks2[])`
  - vraća u `ranks2` `n` rangova procesa u grupi `group2` čiji su rangovi u `group1` navedeni u `ranks1`
  - vraća `MPI_UNDEFINED` ako se rang ne pojavljuje u grupi
  - `MPI_PROC_NULL` je validan rang u `ranks1`
- `MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)`
  - vraća u `result` `MPI_IDENT` ako su članovi grupa i njihov poredak jednaki u obje grupe, `MPI_SIMILAR` ako su članovi grupa jednaki, ali im je poredak različit, u suprotnom `MPI_UNEQUAL`

Konstruktori grupa

- koriste se za stvaranje podskupova i nadskupova postojećih grupa
- lokalne operacije $\Rightarrow$ mogu se definirati različite grupe na različitim procesima
- procesi mogu definirati grupe koje ih ne sadrže
- potrebne su konzistente definicije kada se grupe koriste u konstruktorima komunikatora
- mogu se graditi samo iz već postojećih grupa
- osnovna grupa asocirana je s `MPI_COMM_WORLD`
- duplikacija grupa nije potrebna – jednom stvorena, može imate više referenci kopiranjem handle-a

- `MPI_Comm_group(MPI_Comm comm, MPI_Group *group)`
  - vraća handle na grupu `group` pridruženu komunikatoru `comm`
- `MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)`
  - stvara grupu `newgroup` u kojoj su svi elementi `group1` praćeni svim elementima grupe `group2` koji nisu u `group1`
- `MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)`
  - stvara grupu `newgroup` u kojoj su svi elementi `group1` koji su i u `group2`, poredani kao u grupi `group1`
- `MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)`
  - stvara grupu `newgroup` u kojoj su svi elementi `group1` koji nisu u `group2`, poredani kao u grupi `group1`
- skupovne operacije nisu komutativne, ali su asocijativne
- `MPI_Group_incl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)`

  - stvara novu grupu `newgroup` koja se sastoji od `n` procesa grupe `group`, pri čemu će proces ranga `rank[i]` u `group` biti ranga `i` u grupi `newgroup`
  - svaki od elemenata mora biti pravilan rang i svi elementi moraju biti različiti – u suprotnom je program pogrešan
  - $n = 0 \Rightarrow$ vraća se `MPI_GROUP_EMPTY`

- `MPI_Group_excl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)`
  - stvara novu grupu `newgroup` brisanjem iz `group` procesa s rangovima `rank[0]`, ..., `rank[n-1]`
  - poredak isti kao i u `group`
  - svaki od elemenata mora biti pravilan rang i svi elementi moraju biti različiti – u suprotnom je program pogrešan
  - $n = 0 \Rightarrow$ `newgroup` je identična `group`

- `MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)`
  - ako se `ranges` sastoji od trojki

$$(first_1, last_1, stride_1), \ldots (first_n, last_n, stride_n),$$

    tada se `newgroup` sastoji od niza procesa u `group` s rangovima

$$first_1, first_1 + stride_1, \ldots, first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor \cdot stride_1, \ldots,$$
$$first_n, first_n + stride_n, \ldots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor \cdot stride_n$$

  - svaki izračunati rang mora biti valjani rang u `group` i svi izračunati rangovi moraju biti različiti – u suprotnom je program pogrešan
  - može biti $first_i > last_i$ i $stride_i < 0$, ali ne i $stride_i = 0$

- `MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)`
  - svaki izračunati rang mora biti valjani rang u `group` i svi izračunati rangovi moraju biti različiti – u suprotnom je program pogrešan
  - funkcionalnost ekvivalentna kao da se `ranges` raspiše u niz rangova, a rezultirajući niz koristi u pozivu `MPI_Group_excl`

- `range` operacije ne numeriraju rangove eksplicitno $\Rightarrow$ skalabilnije su ako se implementiraju korektno

Destruktori grupa

- `MPI_Group_free(MPI_Group *group)`
  - označava `group objekt` za dealokaciju
  - handle `group` se postavlja na `MPI_GROUP_NULL`
  - sve operacije vezane uz grupu se dovrše prije stvarne dealokacije

Komunikacijski konteksti

- `konteksti` su svojstvo komunikatora koji dozvoljavaju particionaranje komunikacijskog prostora
- poruka poslana u jednom kontekstu ne može biti primljena u drugom
- nisu eksplicitni MPI objekti $\Rightarrow$ dolaze kao dio komunikatora
- različiti komunikatori imaju različite kontekste

  - esencijalno je sistemski dodijeljen tag (ili tagovi) potrebni za sigurnu točkovnu i kolektivnu komunikaciju ⇒ točkovne i kolektivne operacije ne interferiraju ni unutar jednog ni među različitim komunikatorima

- kolektivne operacije su neovisne o tekućim point-to-point komunikacijama

Komunikatori

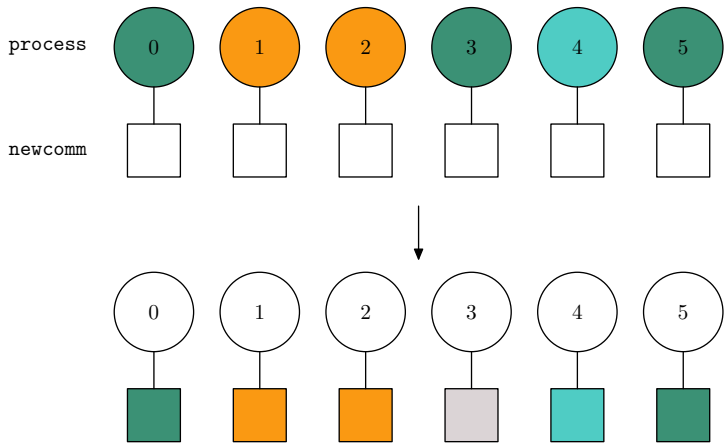
- dijele se na:
  - `intrakomunikatore` – služe za komunikaciju unutar jedne grupe,
  - `interkomunikatore` – služe za komunikaciju između dvije grupe (njima se nećemo baviti)
- spajaju koncepte grupe i konteksta
- svaki sadrži grupu ispravnih sudionika koja uvijek sadrži i lokalan proces
  - procesi `src` i `dest` pri slanju i primanju poruke identificirani su rangom unutar te grupe
  - specificira sve procese koji sudjeluju u kolektivnoj operaciji (i njihov poredak gdje je potrebno)
- restringiraju komunikacijski prostor i pružaju adresiranje procesa neovisno o njihovim fizičkim identitetima (PID-ovima)
- reprezentirani netransparentnim `intrakomunikatorskim objektom` `MPI_Comm` ⇒ ne mogu se prenositi s jednog procesa na drugi
- predefinirani intrakomunikatori `MPI_COMM_WORLD` koji sadrži sve procese i `MPI_COMM_SELF` u kojem se nalazi samo lokalni proces ⇒ dostupni nakon poziva `MPI_Init`
- veličina `MPI_COMM_WORLD`-a se ne mijenja tokom izvođenja programa kod procesa koji su inicijalno pokrenuti s `mpirun`
- postoji predefinirana konstana `MPI_COMM_NULL` – služi za invalidirane handle-ove

Akcesori komunikatora

- `MPI_Comm_size(MPI_Comm comm, int *size)`
  - ekvivalentno nizu poziva:
    ```
    MPI_Comm_group(comm, &group)
    MPI_Group_size(group, size)
    MPI_Group_free(&group)
    ```
- `MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - ekvivalentno nizu poziva:
    ```
    MPI_Comm_group(comm, &group)
    MPI_Group_rank(group, rank)
    MPI_Group_free(&group)
    ```
- `MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)`
  - vraća u `result`:
    - `MPI_INDENT` – `comm1` i `comm2` su handle-ovi na isti objekt
    - `MPI_CONGRUENT` – odgovarajuće grupe identične, različiti konteksti
    - `MPI_SIMILAR` – članovi grupe isti, ali različit poredak rangova
    - `MPI_UNEQUAL` – različite grupe i različiti konteksti

Konstruktori komunikatora

- svi osim `MPI_Comm_create_group` su kolektivne operacije koje se pozivaju na svim procesima grupe asocirane s komunikatorom `comm` (`MPI_Comm_create_group` zovu samo procesi u grupi komunikatora koji se konstruira)
- osnovni komunikator je `MPI_COMM_WORLD`
- `MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)`
  - vraća u `newcomm` novi komunikator s istom grupom, topologijom, ali novim kontekstom
  - poziv je valjan neovisno o tekućim točkovnim komunikacijama
- `MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)`
  - vraća novi intrakomunikator `newcomm` s komunikacijskom grupom `group`
  - svaki proces u `comm` mora pozvati `MPI_Comm_create` s argumentom `group` koja je podgrupa grupe asocirane s `comm` (moguće i `MPI_GROUP_EMPTY` ukoliko proces neće biti ni u jednom od nastalih komunikatora)
  - procesi mogu specificirati različite vrijednosti za `group`
  - ako proces pozove funkciju s `group` $\neq$ `MPI_GROUP_EMPTY`, svi procesi u toj grupi moraju pozvati funkciju s istim argumentom $\Rightarrow$ implicira da argumenti `group` moraju biti disjunktne grupe!
  - ako proces koristi grupu čiji nije element, funkcija vraća `MPI_COMM_NULL` kao `newcomm`
  - funkcija je kolektivna i moraju je pozvati svi procesi u grupi komunikatora `comm`
  - svaki proces dobije handle na komunikator čiji je član



  - pretpostavimo da su nam na raspolaganju 6 procesa i tri grupe: `zelena` $= \{0, 5\}$, `plava` $= \{4\}$ i `žuta` $= \{1, 2\}$, te da svaki proces poziva `MPI_Comm_create` s `group` koji odgovara njegovoj boji na gornjoj slici
    - procesi 1 i 2 su u žutoj grupi, pa će njihov `newcomm` obuhvaćati točno ta dva procesa (označen žuto); analogna priča vrijedi i za procese 0 i 5 u zelenoj grupi (`newcomm` označen zeleno)
    - proces 4 je jedini koji pripada plavoj grupi, pa će grupa njegovog komunikatora `newcomm` (označen plavo) sadržavati samo njega
    - proces 3 poziva funkciju s grupom čiji nije član $\Rightarrow$ `newcomm` $=$ `MPI_COMM_NULL` (označen sivo)
- `MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcoom)`

  - sličan `MPI_Comm_create`, samo ga moraju pozvati svi procesi u grupi `group` umjesto svih procesa u komunikatorskoj grupi
  - ako svi procesi imaju `group` = `MPI_GROUP_EMPTY`
  - `tag` argument je neovisan o komunikacijskom tag atributima i ne smije biti `MPI_ANY_TAG` – služi kod višedretvenih procesa kako bi se razlikovali različiti pozivi funkcije
  - može imati veći overhead od `MPI_Comm_create` jer ne može iskoristiti kolektivnu komunikaciju

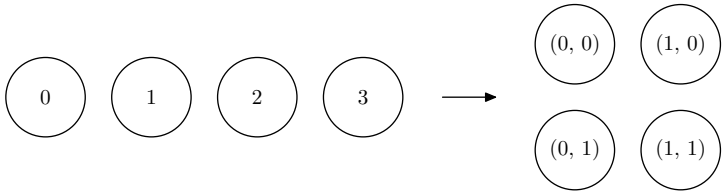
- `MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)`
  - particionira grupu asociranu s `comm` u disjunktne pogrupe, po jednu za svaki `color`
  - procesi su rangirani unutar podgrupe po vrijednosti `key`, pri čemu se neriješeni slučajevi riješavaju s obzirom na rang u staroj grupi
  - novi komunikator nastaje za svaku podgrupu
  - proces može staviti `color` = `MPI_UNDEFINED` – `newcomm` je `MPI_COMM_NULL`
  - kolektivan poziv
  - `color` mora biti nenegativan ili `MPI_UNDEFINED`
  - koristan kada procesi nemaju potpunu informaciju o drugim članovima svoje grupe ⇒ MPI ih otkriva komunikacijom
  - ako procesi imaju informaciju o svojoj grupi, može se izbjeći nepotreban overhead korištenjem drugih funkcija
  - ukoliko svi procesi imaju isti `key`, imat će isti relativan poredak kao i u nadgrupi

Destruktori komunikatora

- `MPI_Comm_free(MPI_Comm *comm)`
  - handle se postavlja na `MPI_COMM_NULL`
  - sve započete operacije će se završiti prije dealokacije

Virtualne topologije

- `topologija` je dodatan, opcionalan atribut koji se može pridružiti intrakomunikatoru
- pruža jednostavnije označavanje procesa unutar grupe komunikatora i može pomoći pri mapiranju procesa na fizičke čvorove
- postoji razlika između virtualne i fizičke topologije – virtualna topologija se može iskoristiti kod pridruživanja procesa fizičkim procesorima ako to poboljšava performanse – to mapiranje je izvan dosega MPI-a
- linearno rangiranje procesa ne reflektira adekvatno komunikacijske uzorke procesa – općenito je logički poredak procesa opisan grafom ⇒ čvorovi su procesi, bridovi povezuju čvorove koji međusobno komuniciraju
- nepostojeći brid između dva čvora u specificiranom grafu ne sprječava razmjenu poruka odgovarajućih procesa – topologija samo ne daje smislen način za označavanje te komunikacije
- specificiranje topologije grafom je prekomplicirano i neefikasno u većini slučajeva, jer su grafovi regularni
- u praksi najčešći prsteni, 2D- i višedimenzionalni gridovi i torusi – mapiranje jednostavnije nego za općenite grafove



Kartezijeva topologija

- ukoliko procese organiziramo u Kartezijevu topologiju, oni i dalje imaju linearne rangove, no svaki proces ujedno ima i dodatnu oznaku (koordinate)
- koristi se za sljedeće topologije:
  - `mesh`
  - `torus`
  - `hiperkocka`
  - `hipercilindar`
- `MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[], const int periods[], int reorder, MPI_Comm *comm_cart)`
  - vraća handle na novi komunikator `comm_cart` s priloženom topološkom informacijom
  - svi procesi moraju imati iste argumente
  - ako je `reorder` = `false`, rang svakog procesa u novoj grupi je identičan rangu u staroj grupi; u suprotnom funkcija može renumerirati rangove procesa kako bi odabrala dobro ulaganje virtualne topologije na fizikalne mašine
  - ako je veličina Kartezijanskog grida manja od veličine grupe `comm_old`, neki procesi dobivaju `MPI_COMM_NULL`
  - `ndims` može biti 0 ⇒ nul-dimenzionalna Kartezijeva topologija je stvorena
  - greška je pozvati funkciju koja specificira grid veći od veličine grupe ili s negativnim `ndims`

  - `periods` specificira koje su dimenzije periodične
    - `mesh` – ni jedna dimenzija nije periodična
    - `torus` – sve dimenzije periodične
    - `hiperkocka` – n-dimenzionalni torus s 2 procesa po smjeru
    - `hipercilindar` – neke dimenzije periodične

- `MPI_Dims_create(int nnode, int ndims, int dims[])`
  - vraća balansiranu distribuciju procesa po koordinati $\Rightarrow$ dimenzije su odabrane tako da budu što bliži brojevi
  - dimenzije se fiksiraju u polju `dims` $\Rightarrow$ funkcija mijenja samo elemente s vrijednošću 0
  - negativne dimenzije su pogrešne
  - dolazi do greške ako `nnodes` nije višekratnik $\prod_{i,\texttt{dims}[i]\neq 0} \texttt{dims}[i]$
  - za `dims[i]` postavljene pozivom funkcije, vrijednosti su poredane u ne-rastućem poretku
  - poziv je lokalan

- `MPI_Cartdim_get(MPI_Comm comm, int *ndims)`
  - vraća broj dimenzija `ndims` u Kartezijevoj topologiji

- `MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[], int coords[])`
  - prima `maxdims` kojeg vraća `MPI_Cartdim_get`
  - u `dims[i]` vraća broj procesa u `i`-toj dimenziji, a u `periods[i]` vraća periodičnost `i`-te dimenzije
  - u `coords` vraća Kartezijske koordinate pozivajućeg procesa

- `MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)`
  - za ulazne koordinate `coords` (niz veličine `ndims`) vraća rang procesa koji se koristi u točkovnoj komunikaciji
  - ako je `i`-ta dimenzija periodična i ako `coords[i]` $\notin \{0, \ldots, \texttt{dims[i]} - 1\}$, `cords[i]` je automatski pomaknut u odgovarajuć interval
  - ako je Kartezijeva topologija nul-dimenzionalna, `coords` je irelevantan i vraća se 0 kao rang

- `MPI_Cart_coords(MPI_Comm comm, int rank, int ndims, int coords[])`
  - inverzno mapiranje funkcije `MPI_Cart_rank`
  - ako je topologija nul-dimenzionalna, `coords` je nepromijenjen

- `MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`
  - istovremeni blokirajući `send` i `recv`
  - `sendbuf` i `recvbuf` moraju biti disjunktni
  - semantika kao kad bi se napravile dvije dretve, jednu za slanje, drugu za primanje podataka, te ih se potom `join`-alo

- `MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`
  - analogno `MPI_Sendrecv`
  - isti buffer se koristi za primanje i slanje podataka

- `MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)`

  - `direction` $\in \{0, \ldots, \texttt{ndims} - 1\}$ je koordinatna dimenzija pomaka
  - podržava cirkularne i *end-off* pomake $\Rightarrow$ ako se šalje preko ruba, moguća je vrijednost `MPI_PROC_NULL` za `rank_source` ili `rank_dest`
  - pogreška je pozvati funkciju s `direction` $\notin \{0, \ldots, \texttt{ndims} - 1\}$
  - `disp` je *upwards shift* ako je > 0 ili *downwards shift* ako je < 0

- `MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)`
  - particionira grupu komunikatora kreiranog s `MPI_Cart_create` u podgrupe koje tvore niždimenzionalne Kartezijeve podgridove
  - `i`-ta vrijednost u `remain_dims` specificira koristi li se `i`-ta dimenzija u podgridu

Kolektivna komunikacija

- komunikacija koja uključuje grupu ili grupe procesa
- jedan od ključnih argumenata kod poziva kolektivnih rutina je komunikator koji definira grupu koja sudjeluje u procesu i daje kontekst za operaciju
- neke imaju jedinstveni proces koji šalje i prima poruke ⇒ korijen (`root`) ⇒ neki argumenti kolektivnih operacija bitni su samo kada funkciju poziva korijen, dok ih svi ostali sudionici komunikacije ignoriraju
- sintaksa i semantika kolektivnih operacija je konzistenta sa sintaksom i semantikom za točkovnu komunikaciju ⇒ posebice se odnosi na tipove podataka
- pravila za podudaranje tipova su striktnija nego kod točkovne komunikacije ⇒ količina poslanih i primljenih podataka mora biti ista, dok su različiti type map-ovi kod primatelja i pošiljatelja i dalje dozvoljeni
- kolektivne operacije mogu završiti čim pozivatelj više ne mora sudjelovati u komunikaciji
- blokirajuća operacija je gotova čim je poziv funkcije gotov
- kod neblokirajuće komunikacije su potrebni dodatni pozivi funkcija za provjeru završetka
- završetak kolektivne operacije povlači da pozivatelj može ponovo koristiti buffer ⇒ ne implicira da su drugi procesi završili ili uopće pozvali funkciju (osim `MPI_Barrier`)
- pozivi mogu koristiti iste komunikatore kao i točkovna komunikacija ⇒ MPI garantira da se poruke poslane kolektivnom komunikacijom neće miješati s poruka poslanima kolektivnom komunikacijom
- kolektivne operacije ne koriste `tag`

- `MPI_Barrier(MPI_Comm comm)`
  - blokira pozivatelja dok svi članovi grupe komunikatora `comm` ne pozovu funkciju
- `MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
  - šalje `count` elemenata tipa `datatype` s adrese `buffer` procesa `root` svim ostalim procesima u grupi komunikatora `comm`
- `MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  - svaki proces (uključujući `root`) šalje sadržaj `sendbuf` koji se sastoji od `sendcount` elemenata tipa `sendtype` procesu `root`
  - proces `root` prima od procesa `i` poruku od `recvcount` elemenata tipa `recvtype`, koju pohranjuje na lokaciju `recvbuf` + `i` $\cdot$ *recvcount* $\cdot$ `extent(recvtype)`, pri čemu je `extent` dobiven pozivom `MPI_Type_get_extent`
  - `recvbuf`, `recvcount` i `recvtype` su nebitni na svim procesima osim `root`
  - možemo koristiti `MPI_IN_PLACE` umjesto `sendbuf` na procesu `root` ⇒ `sendcount` i `sendtype` su ignorirani i pretpostavlja se da su podaci koje šalje `root` već na mjestu
  - specifikacija `sendcount`, `recvcount`, `sendtype` i `recvtype` treba osigurati da na istu lokaciju ne pišemo više puta ⇒ takav poziv je pogrešan
- `MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm)`

  - proširuje funkcionalnost `MPI_Gather` dozvoljavajući varijabilan broj podataka od svakog procesa s obzirom da je `recvcounts` polje, te daje fleksibilnost kod pisanja podataka u `recvbuf` procesa `root` pomoću polja `displs`
  - proces `root` prima od procesa `j` `recvcounts[j]` elemenata tipa `recvtype` koje pohranjuje na lokaciju `recvbuf` + `displs[j]` · `extent(recvtype)`
  - na procesu `root` su svi argumenti bitni, dok su na ostalim procesima bitni samo argumenti `sendbuf`, `sendcount`, `sendtype`, `root` i `comm`
  - argumenti `root` i `comm` moraju imati istu vrijednost na svim procesima
  - ostatak pravila je isti kao i kod `MPI_Gather`

- `MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  - inverzna operacija operaciji gather
  - proces `root` šalje procesu `j` `sendcount` elemenata tipa `sendtype` počevši od lokacije `sendbuf` + `i` · `sendcount` · `extent`(`sendtype`)
  - send buffer je ignoriran na svim procesima osim `root`
  - signatura tipa asociranog s `sendcount`, `sendtype` na procesu `root` mora biti ista signaturi tipa asociranog s `recvcount`, `recvtype` na svim procesima
  - `root` može koristiti `MPI_IN_PLACE` za `recvbuf`

- `MPI_Scatterv(const void *sendbuf, const int sendcounts[], const int displs[], MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  - proširuje funkcionalnost `MPI_Scatter`
  - proces `root` šalje procesu `i` `sendcount[i]` elemenata počevši od lokacije `sendbuf` + `displs`$[i]$ · `extent` · (`sendtype`)
  - ostala pravila su analogna onima za `MPI_Scatter` i `MPI_Gatherv`

- `MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
  - ponaša se kao `MPI_Gather`, samo svi procesi dobiju rezultat
  - svi argumenti su bitni na svim procesima

- `MPI_Allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, MPI_Comm comm)`
  - ponaša se kao `MPI_Gatherv`, samo svi procesi dobiju rezultat

- `MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
  - još se zove i potpuna razmjena (complete exchange)
  - proširenje funkcije `MPI_Allgather` na slučaj kada svi procesi šalju različite podatke svim drugim procesima
  - proces `i` šalje procesu `j` `sendcount` elemenata tipa `sendtype` s lokacije `sendbuf` + `j` · `sendcount` · `extent`(`sendtype`), dok proces `j` prima od procesa `i` `recvcount` elemenata tipa `recvtype` na lokaciju `recvbuf` + `i` · `recvcount` · `extent`(`recvtype`)
  - svi argumenti su bitni na svim procesima
  - može se koristiti opcija `MPI_IN_PLACE` umjesto `sendbuf` na svim procesima

- `MPI_Alltoallv(const void *sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm)`

- proces `i` šalje procesu `j` `sendcounts[j]` elemenata tipa `sendtype` s lokacije `sendbuf`+`sdispls[j]`· `sendcount` · `extent`(`sendtype`), dok proces `j` prima od procesa `i` `recvcount` elemenata tipa `recvtype` na lokaciju `recvbuf` + `rdispls[i]` · `recvcount` · `extent`(`recvtype`)
- može se koristiti opcija `MPI_IN_PLACE` umjesto `sendbuf` na svim procesima

- `MPI_Alltoallw(const void *sendbuf, const int sendcounts[], const int sdispls[], const MPI_Datatype sendtypes[], void *recvbuf, const int recvcounts[], const int rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm)`
  - najopćenitiji tip potpune razmjene
  - proces `i` šalje procesu `j` `sendcounts[j]` elemenata tipa `sendtypes[j]` s lokacije `sendbuf` + `sdispls[j]`, dok proces j prima `recvcounts[i]` elemenata tipa `recvtypes[i]` s lokacije `recvbuf` + `rdispls[i]`
  - vrijednosti u `sdispls` i `rdispls` dane su u byteovima
  - može se koristiti opcija `MPI_IN_PLACE` umjesto `sendbuf` na svim procesima

- `MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
  - za 0 $\leq$ `i` $\leq$ `count - 1`, vraća u `recvbuf` + `i` · `extent`(`datatype`) rezultat redukcije niza `recvbuf` + `i` · `extent`(`datatype`) korištenjem operacije `op`
  - pretpostavka je da je operacija `op` uvijek asocijativna $\Rightarrow$ sve predefinirane operacije su i komutativne
  - `datatype` mora biti kompatibilan s `op`
  - predefinirane operacije rade samo s osnovnim tipovima podataka
  - moguće je definirati operacije koje će funkcionirati s općenitim tipovima podataka
  - tipovi podataka u `sendbuf` se mogu preklapati, dok je to greška kod `recvbuf`
  - `MPI_IN_PLACE` opcija se može koristiti pod `sendbuf` na `root` procesu
  - proces `root` koristi `MPI_ROOT` za `root`, dok ostali procesi koriste njegov rang
  - predefinirane operacije dane su u sljedećoj tablici

operacija	značenje
MPI_MAX	maksimum
MPI_MIN	minimum
MPI_SUM	suma
MPI_PROD	produkt
MPI_LAND	logički AND
MPI_BAND	bitwise AND
MPI_LOR	logički OR
MPI_BOR	bitwise OR
MPI_LXOR	logički XOR
MPI_BXOR	bitwise XOR
MPI_MAXLOC	maksimum i lokacija
MPI_MINLOC	minimum i lokacija

  - operatacija `maxloc` se koristi za traženje globalnog maksimuma i njemu pridruženog indeksa $\Rightarrow$ definirana je s

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix},$$

pri čemu je $w = \max(u, v)$ i

$$k = \begin{cases} i & u > v \\ \min(i, j) & u = v \\ j & u < v \end{cases}$$

  - operacija `minloc` se definira analogno
  - s obzirom da operacija koristi dva argumenta, postoje specijalni tipovi koje možemo koristiti: `MPI_FLOAT_INT`, `MPI_DOUBLE_INT`, `MPI_LONG_INT`, `MPI_2INT`, `MPI_SHORT_INT`, `MPI_LONG_DOUBLE_INT`

- `MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)`
  - veže korisnički definiranu funkciju `user_fn` za netransparentan operator objekt, pri čemu vraća handle na taj objekt u `op`
  - ako je `commute` istina, operacija je komutativna, u suprotnom je poredak operanada fiksan i definiran u rastućem poretku rangova procesa
  - ISO C prototip korisnički definirane funkcije je `typedef void MPI_User_function(void *invec, void *inoutvec, int *len, MPI_Datatype *datatype)`
  - `datatype` je handle na tip koji se koristi u pozivu `MPI_Reduce`
  - funkcija radi `inoutvec[i]` $=$ `invec[i]` $\circ$ `inoutvec[i]`, $0 \leq i \leq$ `len`
  - jedina MPI funkcija koja se može pozvati unutar `user_fn` je `MPI_Abort`

- `MPI_Op_free(MPI_Op *op)`
  - označava korisnički definirani operator `op` za dealokaciju i postavlja `op` na `MPI_OP_NULL`

- `MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
  - ponaša se kao i `MPI_Reduce`, samo se rezultat vraća svim procesima
  - `MPI_IN_PLACE` opcija se može koristiti pod `sendbuf` na svim procesima

- `MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf, int recvcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
  - funkcija prvo vrši globalnu redukciju po elementima vektora od `count` $=$ `n` $\cdot$ `recvcount` elemenata tipa `datatype` danih u send bufferu (`sendbuf`, `count`, `datatype`) koristeći operaciju `op`, pri čemu je `n` broj procesa $\Rightarrow$ rezultirajući vektor se tretira kao `n` uzastopnih blokova od `recvcount` elemenata na kojem se vrši `scatter` svim procesima grupe, pri čemu je `i`-ti blok poslan procesu `i` i pohranjen u recv bufferu (`recvbuf`, `recvcount`, `datatype`)
  - ekvivalentno pozivu `MPI_Reduce` kod kojeg je `count` jednak `n` $\cdot$ `recvcount`, nakon kojeg je pozvan `MPI_Scatter` sa `sendcount` jednak `recvcount`
  - `MPI_IN_PLACE` se može koristiti kao argument za `sendbuf` na svim procesima

- `MPI_Reduce_scatter(const void *sendbuf, void *recvbuf, const int recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
  - proširuje funkcionalnost `MPI_Reduce_scatter_block` dozvoljavanjem varijabilnih veličina blokova
  - operacija prvo vrši redukciju po elementima vektora od `count` $= \sum_{i=0}^{n-1}$ `recvcounts[i]` elemenata iz send buffera (`sendbuf`, `count`, `datatype`) i koristeći operaciju `op` $\Rightarrow$ rezultirajući vektor se tretira kao `n` uzastopnih blokova pri čemu blok `i` ima `recvcounts[i]` elemenata nad kojima se vrši `scatter` svim procesima grupe
  - `MPI_IN_PLACE` se može koristiti kao argument za `sendbuf` na svim procesima

- `MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm`
  - radi `prefix scan` na distribuiranim podacim grupe komunikatora `comm`
  - operacija vraća u `recvbuf` rezultat `count` redukcija elemenata tipa `datatype` proces s rangovima $0, \ldots,$ `i` (inkluzivno)
  - podržane operacije su iste kao i za `MPI_Reduce`

  - MPI_IN_PLACE se može koristiti za sendbuf

- MPI_Exscan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  - radi ekskluzivan scan
  - vrijednost u recvbuf procesa 1 jednaka je vrijednosti sendbuf procesa 0
  - ostatak svojstava je isti kao i kod MPI_Scan

Paralelni I/O

- iako POSIX pruže model vrlo portabilnog file system-a, nije dovoljan za portabilnost i optimizaciju potrebnu za paralelni I/O ⇒ značajna optimizacija moguća je samo ako paralelni I/O sustav pruža sučelje koje podržava particioniranje podataka u datoteci među procesima i kolektivno sučelje koje dozvoljava potpun transfer globalnih struktura podataka između memorije procesa i datoteka
- dodatne optimizacije moguće su ukoliko se omogući asinkroni I/O, pristup datotekama s proizvoljnih offset-a i kontrola nad fizičkim rasporedom datoteke na disku
- umjesto definiranja načina pristupa datotekama pomoću klasičnih uzoraka (broadcast, redukcija, scatter, gather), MPI pristupa I/O pomoću izvedenih tipova podataka ⇒ fleksibilnost
- `file` je uređena kolekcija tipiranih jedinica
  - MPI pruća sekvencijalan pristup bilo kojem cjelobrojnom skupu tih jedinica
  - grupa procesa kolektivno otvara `file` i svi kolektivni I/O pozivi na tom `file`-u su kolektivni nad tom grupom
- `displacement` je apsolutna pozicija u byteovima od početka datoteke
  - definira lokaciju gdje počinje `view`
  - različit od typemap displacement-a
- `etype` (elementarni tip) je jedinica pristupanja i pozicioniranja podataka
  - može biti bilo koji MPI tip podataka
  - mogu se konstruirati bilo kojim konstruktorima tipova podataka pod uvjetom da su svi displacement-i nenegativni i monotono nepadajući
  - pristup podacima se vrši u etype jedinicima, a pisanje i čitanje moguće je samo za cijele etype-ove
  - `offset`-i se izražavaju u broju etypeova
  - file pointeri pokazuju na početke etypeova
  - ovisno o kontekstu, izraz etype može označavati partikularni MPI tip, podatkovnu jedinicu unutar tipa ili raspon tog tipa
- `filetype` je osnova za particioniranje datoteke među procesima i definira uzorak za pristupanje datoteci
  - ili je jedna etype ili MPI tip konstruiran iz više instanci istog etypea
  - raspon svake "rupe" u filetypu mora biti višekratnik raspona etypea
  - displacement typemapa filetype ne moraju biti različiti, ali moraju biti nenegativni i monotono nepadajući
- `view` definira trenutan skup podataka vidljivih i dostupnih iz otvorene datoteke kao uređen skup etypeova
  - svaki proces ima vlasiti view datoteke, definiran s tri vrijedonsti: displacement, etype i filetype
  - uzorak definiran s filetype se ponavlja, počevši od displacement, sve do kraja filea
- `offset` je pozicija u datoteci relativno na trenutni view, izražena kao broj etypeova
  - rupe u filetypeu su preskočene kada se računa pozicija
  - offset 0 je lokacija prvog etypea vidljivog u view-u (nakon preskakanja displacementa i incijalnih rupa u viewu)
- `file size` MPI datoteke se mjeri u byteovima od počekta datoteke

  - novostvorena datoteka ima veličinu nula
  - za svaki view, `end of file` je offset od prvog etypea dostupnog u tom viewu počevši od zadnjeg bytea u fileu
- `file pointer` je implicitni offset kojeg koristi MPI
  - individualni file pointeri su lokalni svakom procesu koji je otvorio file
  - zajednički file pointer dijeli grupa procesa koji su otvorili file
- `file handle` je netransparentan objekt stvoren s `MPI_File_open` i oslobođen s `MPI_File_close`

Manipulacija datotekama

- `MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info info, MPI_File *fh)`
  - otvara file identificiran s `filename` na svim procesima grupe komunikatora `comm`
  - kolektivan rutina ⇒ svi procesi ju moraju pozvati s istim vrijednostima `amode` i svi moraju koristiti `filename` koji referencira isti file
  - proces može otvoriti datoteku neovisno o drugim procesima koristeći komunikatori `MPI_COMM_SELF`
  - potrebno je zatvoriti sve otvorene fileove prije poziva `MPI_Finalize`
  - format za `filename` ovisi o implementaciji i mora biti dokumentiran
  - inicijalno svi procesi gledaju file kao linearan niz byteova i svaki proces gleda podatke u svojoj reprezentaciji (nema konverzije)
  - `amode` dobivamo bitwise OR operacijom nad nekim od sljedećih vrijednosti:
    * `MPI_MODE_RDONLY` – samo čitanje
    * `MPI_MODE_RDWR` – čitanje i pisanje
    * `MPI_MODE_WRONLY` – samo pisanje
    * `MPI_MODE_CREATE` – stvara file ukoliko ne postoji
    * `MPI_MODE_EXCL` – greška ukoliko stvaramo file koji već postoji
    * `MPI_MODE_DELETE_ON_CLOSE` – briše file pri zatvaranju
    * `MPI_MODE_UNIQUE_OPEN` – file neće biti istovremeno otvoren drugdje (eliminira file locking)
    * `MPI_MODE_SEQUENTIAL` – fileu će se pristupati sekvencijalno
    * `MPI_MODE_APPEND` – inicijalna pozicija svih file pointera postavljena je na `end of file`
  - `info` argument pruža dodatne argumente o pristupima datotekama i file sistemu ⇒ može se koristiti `MPI_INFO_NULL`
- `MPI_File_close(MPI_File *fh)`
  - prvo sinkronizira stanje filea, a potom zatvara file asociran s `fh`
  - kolektivna rutina
  - potrebno osigurati da su svi neblokirajući zahtjevi i split kolektivne rutine nad fileom dovršene
  - dealocira objekt pridružen `fh` i postavlja `fh` na `MPI_FILE_NULL`
- `MPI_File_delete(const char *filename, MPI_Info info)`
  - briše file `filename`
  - ako proces ima file trenuto otvoren, rezultat pristupanja fileu ovisi o implementaciji

File viewovi

- `MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, const char *datarep, MPI_Info info)`
  - postavlja procesov view na file počevši od `disp` s tipom podataka `etype`, pri čemu je distribucija podataka procesa postavljena na `filetype`, a reprezentacija na `datarep`
  - resetira individualne file pointere i zajedničke file pointere na nulu
  - kolektivna operacija ⇒ vrijednosti `datarep` i rasponi `etype` u reprezentaciji podataka moraju biti identični na svim procesima grupe, dok vrijednosti za `disp`, `filetype` i `info` mogu varirati
  - tipovi `etype` i `filetype` moraju biti commitani
  - `disp` je dan u apsolutnom offsetu u byteovima od početka datoteke ⇒ koristi se kako bi se preskočili mogući headeri ili kada datoteke sadrže nizove segmenata podataka kojima se pristupa na različite načine
  - ako je file otvoren za pisanje, ni etype ni filetype ne smiju sadržavati preklapajuća područja (tipovi korišteni na različitim procesima se mogu preklapati)
  - ako `filetype` ima rupe, one nisu dostupne pozivajućem procesu, no `disp`, `etype` i `filetype` se mogu promijeniti za buduće pozive
  - pogrešno je koristiti apsolutne adrese u konstruktorima etypeova i filetypeova
  - za `datarep` je dovoljno koristiti `"native"` (podaci u fileu su spremljeni točno onako kako su spremljeni i u memoriji)
  - sve neblokirajući zahtjevi i split kolektivne rutine moraju završiti prije poziva funkcije
- `MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep)`
  - vraća odgovarajuće podatke o viewu filea `fh`
  - `etype` i `filetype` su novi tipovi podataka s typemapovima jednakim onima trenutnog etypea i filetypea
  - korisnik je odgovoran da je `datarep` dovoljno velik ⇒ veličina je limitirana s `MPI_MAX_DATAREP_STRING`

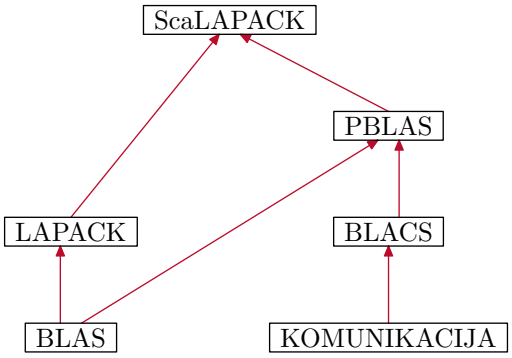
Pristup podacima

- podaci se transferiraju između datoteka i procesa korištenjem poziva za čitanje i pisanje
- postoje tri ortogonalna aspekta pristupu podataka
  - pozicioniranje (eksplicitan offset – implicitan file pointer)
  - sinkronizam (blokiranje – neblokiranje i split kolektivne operacije)
  - kordinacija (nekolektivno – kolektivno)
- klasični `read`/`fread` i `write`/`fwrite` su blokirajuće, nekolektivne operacije koje koriste individualne pointere
- implementacija rutina za pristup podacima može bufferirati podatke kako bi poboljšala performanse ⇒ ne utječe na čitanje, ali utječe na pisanje ⇒ potrebno je pozvati `MPI_File_sync` kako bi bili sigurni da su podaci zapisani na disk
- lokacija podataka u datoteci dana je offsetom u trenutni view ⇒ specificirana je trojkom (`buf`, `count`, `datatype`), pri čemu je pristup podacima definiran kao i kod komunikacijskih procedura
- kod čitanja možemo odrediti da smo došli do EOF ukoliko smo pročitali manje podataka nego što je to traženo
- pisanje iznad EOF povećava veličinu datoteke

- `MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
  - čita iz `fh` `count` elemenata tipa `datatype` počevši od adrese `offset` i vraćajući rezultat u `buf`

ScaLAPACK

- Scalable Linear Algebra PACKage
- paralelna implementacija LAPACK-a
- namijenjena računalima s distribuiranom memorijom koja rade u MIMD modelu ili pak mrežama radnih stanica
- koristi PVM ili MPI za razmjenu poruka
- ciljevi:
  - `efikasnost` ⇒ rutine moraju biti što je moguće brže
  - `skalabilnost` ⇒ porastom veličine problema i broja procesa performanse trebaju ostati konstantne
  - `pouzdanost` ⇒ uključuje i granice za greške
  - `portabilnost` ⇒ mora funkcionirati na svim bitnim paralelnim strojevima
  - `fleksibilnost` ⇒ korisnici mogu jednostavno konstruirati nove rutine iz već postojećih
  - `jednostavno korištenja` ⇒ sučelja LAPACK-a i ScaLAPACK-a trebaju biti što je moguće sličnija
- svi ciljevi ostvareni su korištenjem dvije osnovne biblioteke
  - BLAS – Basic Linear Algebra Subroutines
  - BLACS – Basic Linear Algebra Communication Subprograms
- pisana u Fortranu 77 izuzev nekoliko rutina za simetrični svojstveni problem i pomoćnih rutina koji eksploatiraju IEEE aritmetiku
- koristi eksplicitnu razmjenu poruka za međuprocesorsku komunikaciju
- rutine su bazirane na blokiranim algoritmima kako bi se minimizirala učestalost transfera podataka između različitih nivoa memorijske hijerarhije
- osnovni gradivni blokovi su distribuirane verzije BLAS rutina obuhvaćene u biblioteci PBLAS, te komunikacijske rutine koje se često pojavljuju u paralelnim problemi linearne algebre obuhvaćene u biblioteci BLACS
- kao i LAPACK, sastoji se od
  - driver rutina – rješavaju standardne tipove problema
  - computational rutina – vrše različite zadatke
  - auxiliary rutina – vrše određen podprobleme ili učestala računanja
- podrška za guste i vrpčaste matrice, ali ne i za rijetke matrice
- slična funkcionalnost je pružana za realna i kompleksne matrice
- ScaLAPACK koristi LAPACK rutine što je to više moguće ⇒ LAPACK je paraleliziran samo ako je dostupna paralelna implementacija BLAS-a
- BLACS ⇒ biblioteka za razmjenu poruka specifično dizajnirana za linearnu algebru



  - računski model se sastoji od 1D ili 2D procesnog grida, pri čemu svaki proces sadrži dio matrica ili vektora
  - sadrži rutine za sinkrono slanje/primanje matrica s jednog procesa na drugi, broadcast podmatrica više procesa ili za globalne redukcije, ili za dohvat informacija o gridu
  - potrebni su različiti konteksti za izvršavanje rutina

- korištenje ScaLAPACK-a možemo podijeliti u 4 koraka:

  1. inicijalizicija procesnog grid-a
  2. distribuiranje matrice na procesni grid
  3. poziv ScaLAPACK funkcija
  4. oslobađanje procesnog grida

BLACS

- `blacs_get_(int *icontxt, int *what, int *val)`
  - vraća u `val` vrijednost svojstva `what` konteksta `icontxt`
  - ukoliko je `what` nevezan za kontekst, vrijednost argumenta `icontxt` nije bitna
  - moguće vrijednosti argumenta `what` su
    - `0` – osnovni sistemski kontekst
    - `1` – raspon ID-eva poruka
    - `2` – razine debuggiranja s kojom je biblioteka kompajlirana
    - `10` – handle na kontekst koji je korišten kako bi se definirao kontekst `icontxt`
    - `11` – broj prstena koji trenutna višeprstenasta topologija koristi
    - `12` – broj grana koje topologija stabla koristi