

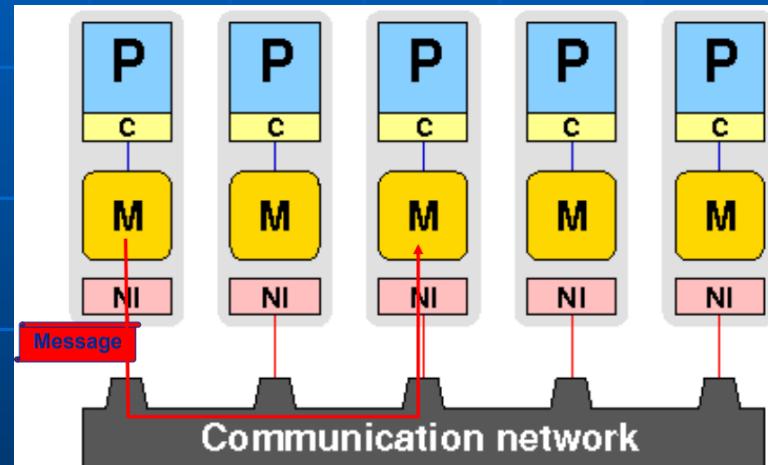
# Paralelni sistemi: MPI-Uvod i Point-to- Point komunikacija



Prof. dr Natalija Stojanović

# Message Passing paradigm

Arhitektura sa distribuiranom memorijom: Svaki proces pristupa svom adresnom prostoru. Nema deljivog adresnog prostora. Razmena podataka i komunikacija između procesa se izvodi eksplicitnim slanjem poruka kroz komunikacionu mrežu.



Message passing biblioteka:

- Treba biti fleksibilna, efikasna i portabilna
- Treba da skriva razliku u hardveru koja postoji između računara i kao i komunikaciju niskog nivoa od aplikacionog programera

# MPI-Message Passing Interface

MPI (Message Passing System) je standardizovan, prenosiv sistem za razmenu poruka.

Namenjen je za podršku komunikacijama u paralelnim sistemima (klasterima, mreži računara, multiračunarima, multiprocesorima) u svrhu poboljšanja performansi izvršenja aplikacije na ovakvim arhitekturama.

Pre pojave ovog standarda, svaki proizvođač je implementirao svoju varijantu sistema za ostvarivanje komunikacije između procesa razmenom poruka, pa je ovakav **koncept** komunikacije strogoo zavisio od arhitekture računarskog sistema.

# MPI standard

Iz potrebe za definisanjem standarda nezavisnog od hardverske platforme nastao je MPI.

MPI nije programski jezik već definiše sintaksu i semantiku bibliotečkih funkcija korisnih za pisanje prenosivih programa za razmenu poruka.

MPI je biblioteka funkcija koje se mogu pozivati iz konvencionalnih programske jezike, kao što su Fortran, C, C++. Danas, MPI 3 standard podržava preko 440 funkcija

Program je pisan korišćenjem sekvencijalnog programskog jezika  
Razmena podataka između procesa: Slanjem/primanjem poruka  
korišćenjem funkcija iz MPI biblioteke  
Sve promenljive su lokalne za proces!  
Isti program se izvršava na svakoj mašini!

# MPI middleware

Glavni cilj MPI standarda je bio postizanje prenosivosti sistema na različite mašine.

Potrebno je samo da MPI biblioteka bude dostupna. MPI poseduje visok stepen fleksibilnosti, čemu govori u prilog činjenica da se može izvršavati u mreži radnih stanica, kao i na jednom računaru.

U prilog fleksibilnosti ovog sistema govori činjenica da se MPI sistem može izvršavati na mreži heterogenih računara, odnosno skupu procesora različitih arhitektura.

Korisnik ne vodi računa o tome da li se poruke razmenjuju između procesora istih ili različitih arhitektura.

MPI automatski obavlja sve neophodne konverzije podataka i obezbeđuje odgovarajući protokol za komunikaciju.

## MPI prednosti

MPI omogućava delimično preklapanje procesa komunikacije i procesa izračunavanja.

Koristi prednosti inteligentnih sistema za komunikaciju, i prikriva latencije pri komunikaciji.

To se postiže pozivom procesa da komuniciraju bez blokiranja, čime je iniciranje komunikacije odvojeno od njenog kompletiranja.

Tvorac MPI-a je MPI forum. MPI forum čini grupa istraživača koja definiše, održava i razvija MPI standard.

Najnoviji standard pored gore pomenutih karakteristika obezbeđuje podršku za kreiranje dinamičkih procesa, jednostranu komunikaciju, paralelni U/I, proširene grupne operacije itd.

# MPI procesi

MPI program podrazumeva skup procesa, gde svaki obavlja neki zadatak nad svojim lokalnim podacima.

Zadatak koji obavlja svaki proces je definisan kodom programa.

Razmena podataka između procesa obavlja se korišćenjem funkcija koje se nalaze u MPI biblioteci.

Procesi mogu pripadati imenovanim grupama.

Grupe omogućavaju da se MPI operacije mogu ograničiti isključivo na procese koji pripadaju jednoj grupi.

# MPI procesi i grupe procesa

■ MPI podrazumeva da se komunikacija obavlja u okviru poznate grupe procesa.

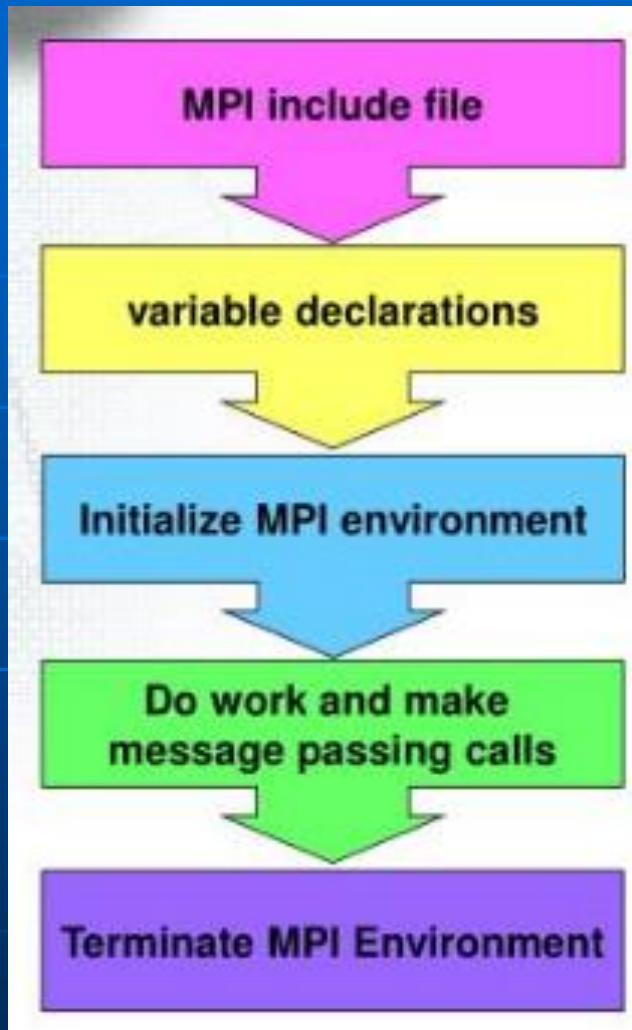
- Svakoj grupi se dodeljuje identifikator
- Svakom procesu unutar grupe se takođe dodeljuje identifikator.
  - Par (identifikator\_grupe, identifikator\_procesa) na jedinstveni način identificuje izvor ili odredište poruke
- U sistemu može postojati više grupa procesa (koje se mogu i preklapati) koje obavljaju izračunavanje i koje se izvršavaju u isto vreme.

# MPI program

MPI program se sastoji iz više instanci sekvensijalnog programa koji komuniciraju pozivima odgovarajućih funkcija MPI biblioteke. Ove funkcije se mogu svrstati u četiri grupe:

- funkcije koje vrše inicijalizaciju i okončavanje komunikacija
- funkcije koje služe za komunikaciju između para procesa
- funkcije koje izvode operacije nad grupom procesa
- funkcije za kreiranje proizvoljnih (izvedenih) tipova podataka
- funkcije za kreiranje novih komunikatora (grupa procesa koji komuniciraju)

# Struktura MPI programa



Svi MPI programi imaju sledeću opštu strukturu

- ...uključivanje MPI header file-a (#include <mpi.h>)
- ...deklaracija promenljivih
- ...inicijalizacija MPI okruženja
- ...odgovarajuća izračunavanja i pozivi MPI funkcija
- ...zatvaranje svih MPI komunikacija

# Struktura MPI programa

MPI header file sadrži prototipove MPI funkcija kao i definicije makroa, specijalnih konstanti i tipova podataka korišćenih od strane MPI. Odgovarajuća #include direktiva mora postojati u bilo kom programu koji koristi MPI funkcije ili konstante.

#include <mpi.h>

Inicijalizacija MPI okruženja obavlja se pozivom funkcije **`MPI_Init(&argc, &argv)`**.

Ova funkcija obavlja specijalna podešavanja tako da može da se koristi MPI biblioteka.

U slučaju da postoji neki problem ova funkcija vraća grešku.

Zatvaranje svih MPI komunikacija obavlja se pozivom funkcije **`MPI_Finalize()`** koja briše sve MPI strukture podataka, završava sve operacije koje nisu kompletirane, itd.

Posle poziva ove funkcije nijedan poziv bilo koje MPI funkcije ne može biti izveden. Ako svi procesi ne izvrše ovu funkciju, program se blokira.

```
#include <mpi.h>
void main (int argc, char *argv[])
{
    int np, rank;
    MPI_Init(&argc, &argv); //1
    MPI_Comm_rank(MPI_COMM_WORLD,&rank); //2
    MPI_Comm_size(MPI_COMM_WORLD,&np); //2
    /*  Do Some Works      */
    MPI_Finalize();
}
```

# MPI konvencije

Imena svih MPI funkcija, konstanti, tipova itd. počinje sa MPI\_.  
Imena MPI funkcija imaju sledeći izgled:

`MPI_Xxxx(....)`

Primer. `MPI_Init(&argc, &argv).`

Imena MPI konstanti i osnovnih MPI tipova podataka se sastoje od svih velikih slova, npr.

`MPI_COMM_WORLD, MPI_FLOAT`

Imena specijalnih MPI tipova poštuju istu konvenciju kao za funkcije, npr.

`MPI_Comm, MPI_Datatype`

# MPI konvencije-nastavak

Sve MPI funkcije u C-u vraćaju INT vrednost koja ukazuje na moguću grešku.

U slučaju da je ta vrednost jednaka predefinisanoj celobrojnoj konstanti MPI\_SUCCESS nije došlo do greške prilikom poziva funkcije, u suprotnom jeste.

```
int ierr;  
...  
ierr = MPI_Init(&argc, &argv);  
...  
if (ierr == MPI_SUCCESS) {  
    ...routine ran correctly...  
}
```

# MPI komunikacija

- Komunikacija između 2 procesa
  - Slanje/primanje MPI poruka
- MPI poruka
  - Niz elemenata konkretnog MPI tipa



- MPI tip
  - Osnovni tip
  - Izvedeni tip

# MPI osnovni tipovi podataka

MPI obezbeđuje osnovne tipove podataka koji odgovaraju različitim osnovnim tipovima podataka u C-u

MPI Datatype	C Type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int

# MPI osnovni tipovi podataka

MPI obezbeđuje osnovne tipove podataka koji odgovaraju različitim osnovnim tipovima podataka u C-u

MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	(none)
MPI_PACKED	(none)

## MPI osnovni tipovi podataka-nastavak

Generalno pravilo je da MPI tip podataka koji se nalazi u pozivu funkcije `MPI_Recv` (funkcija za primanje podataka) mora poklopiti sa MPI tipom podatka u pozivu `MPI_Send` (funkcija za slanje podataka).

MPI omogućava definisanje proizvoljnog tipa podatka izvedenog iz osnovnog. U tu svrhu MPI obezbeđuje posebne funkcije koje to omogućavaju.

# Komunikatori

MPI vodi računa o jedinstvenoj identifikaciji procesa prilikom komunikacije sa drugim procesima.

Korišćenjem identifikatora sve funkcije koje se koriste u komunikaciji su nezavisne od fizičke lokacije učesnika

Dovoljno je znati samo identifikator procesa da bi se sa njim obavila komunikacija.

Komunikatori predstavljaju grupe procesa koji mogu međusobno komunicirati.

Ime komunikatora je argument svake MPI funkcije preko koje se obavlja komunikacija među procesima.

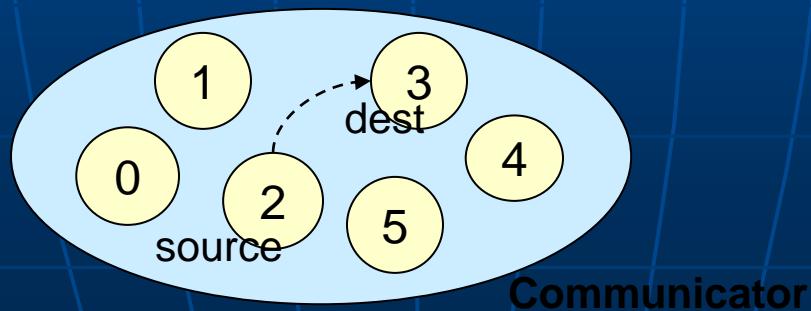
Procesi mogu komunicirati samo ako dele komunikator.

# Komunikatori-nastavak

U okviru svakog komunikatora identifikatori procesa uzimaju uzastopne vrednosti (od 0 do broj\_procesa-1). Ova vrednost naziva se rang (eng. rank) procesa.

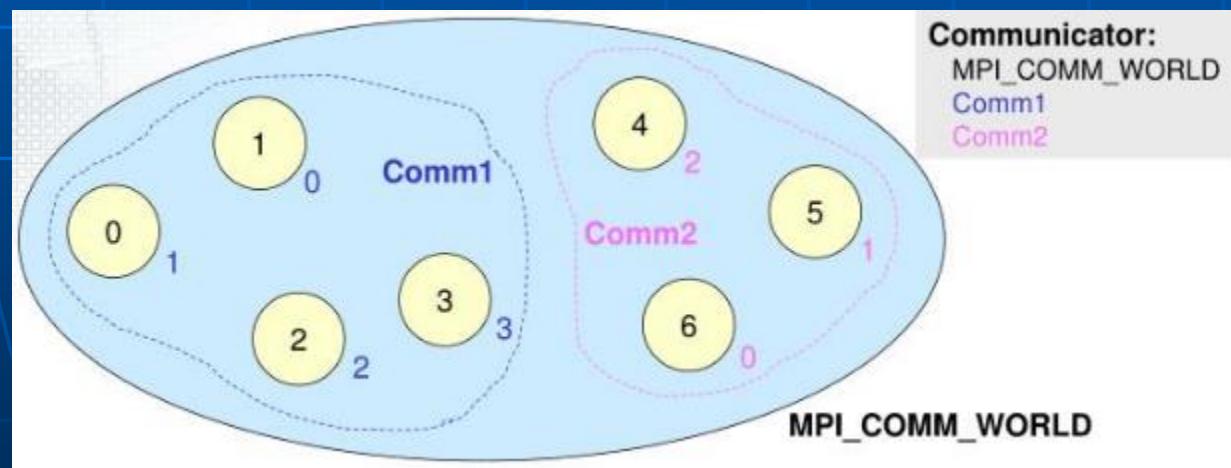
MPI automatski obezbeđuje osnovni (globalni) komunikator **MPI\_COMM\_WORLD**.

Korišćenjem ovog komunikatora svaki proces može komunicirati sa svim ostalim procesima u okviru njega. Po definiciji, svi procesi su deo komunikatora **MPI\_COMM\_WORLD**, tj. njegovi članovi.



# Komunikatori-nastavak

U programu može biti definisano više komunikatora i jedan proces može biti član više njih. U okviru svakog komunikatora procesi za identifikatore (rank) uzimaju vrednosti od 0 do ukupan\_broj\_procesa-1



# Komunikatori-nastavak

- Proces određuje svoj svoj rang u komunikatoru korišćenjem funkcije:

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

Isti proces može imati različite vrednosti za rang (rank) u različitim komunikatorima.

Proces može odrediti veličinu komunikatora (tj. broj procesa u njemu) kome pripada korišćenjem funkcije

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

# Komunikatori-nastavak

Pr.

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]) {
    int myrank, size; // svaki proces ima svoje promenljivu myrank i size
    MPI_Init(&argc, &argv); // Inicijalizacija MPI okruženja
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); // svaki proces izvršava ovu funkciju i na taj način postavlja svoju vrednost za myrank
    MPI_Comm_size(MPI_COMM_WORLD, &size); // svaki proces izvršava ovu funkciju i na taj način postavlja svoju vrednost size
```

```
printf("Processor %d of %d: Hello World!\n", myrank, size);
MPI_Finalize(); /* Terminate MPI */
}
```

Izlaz programa koji se poziva za 4 procesa na jednom računaru sa mpiexec -n 4 imefajla.exe je:

	PO	P1	P2	P3
myrank	0	1	2	3
size	4	4	4	4

**Processor 2 of 4: Hello World!**  
**Processor 1 of 4: Hello World!**  
**Processor 3 of 4: Hello World!**  
**Processor 0 of 4: Hello World!**

# Point-to-point komunikacija

PtP komunikacija podrazumeva komunikaciju tipa: jedan proces šalje poruku a drugi prima tu poruku. MPI podržava ovu komunikaciju odgovarajućim funkcijama. Komunikacija između dva procesa uključuje sledeće komponente: pošiljaoca (*sender*), primaoca (*receiver*), podatke same poruke, oznaku poruke (*message tag*) i komunikator, koji obezbeđuje kontekst komunikacije. Funkcija koja implementira slanje sa blokiranjem ima sledeći oblik

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,  
int dest, int tag, MPI_Comm comm);
```

gde **buf** ukazuje na mesto u memoriji odakle počinje slanje **count** podataka tipa **dtype**. Broj podataka u receive pozivu (MPI\_Recv) treba da bude veći ili jednak broju **count** u MPI\_Send.

# Point-to-point komunikacija-nastavak

`dest` je rang procesa kome se šalje poruka, `tag` je proizvoljan broj koji služi za prepoznavanje odgovarajuće poruke na prijemu(taj broj mora da bude isti u **MPI\_Send** i **MPI\_Recv** koji razmenjuju istu poruku), `comm`-komunikator u okviru koga se odvija komunikacija.

Npr.

```
MPI_Send(a,10,MPI_INT,0,10,MPI_COMM_WORLD);  
MPI_Send(&b,1,MPI_DOUBLE,2,19,Comm1);
```

Funkcija koja implementira prijem sa blokiranjem je

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype,  
int source, int tag, MPI_Comm comm, MPI_Status  
*status);
```

# Point-to-point komunikacija-nastavak

`buf` ukazuje na mesto u memoriji od koje počinje poruka koja je primljena. Maksimalan broj podataka tipa `dtype` koji se prima određen je drugim i trećim argumentom `count` i `dtype`. Argument `source` je rang izvora poruke, `tag` oznaka poruke a `comm` komunikator u kojem moraju da budu oba procesa.

Pr.

```
MPI_Recv(c,10,MPI_INT,1,10,MPI_COMM_WORLD,&stat);  
MPI_Recv(&d,1,MPI_DOUBLE,0,19,Comm1,&stat);
```

Moguće je korišćenje globalnog znaka (wildcard) za source (`MPI_ANY_SOURCE`, tj. prima poruku od bilo kog procesa) i tag (`MPI_ANY_TAG`, prima poruku sa bilo kojom oznakom) u ovoj funkciji. Tada argument `status` daje informaciju o izvoru i oznaci poruke(u slučaju korišćenja wildcard), kao i broju podataka koji su primljeni.

# Point-to-point komunikacija-nastavak

Makimalno count (može i manje!!) podataka može biti primljeno u suprotnom poruka o grešci. Pošiljalac i primaoc treba da se slažu po tipu podataka, u suprotnom rezultat je nedefinisan.

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

- status - status of receive operation
- datatype - datatype of each receive buffer element
- count - number of received elements

Tag of received message – status.MPI\_TAG

Rank of the sender – status.MPI\_SOURCE

# Point-to-point komunikacija-nastavak

## Kako funkcioniše MPI\_Send?

Postoje dva scenarija izvršenja zavisno od veličine poruke koja se šalje.

Ako je veličina poruke toliko da može da stane u sistemski bafer na strani primaoca, poruka se kopira upravo u taj bafer. Proces koji je pozvao MPI\_Send se vraća i nastavlja sa daljim izvršenjem. Kada se u procesu primaocu javi odgovarajući poziv **MPI\_Recv** poruka se kopira u user bafer tj. u odgovarajuću promenljivu buf. U slučaju da je veličina poruke veća od veličine sistemskog bafera tada se komunikacija odvija bez baferovanja tj. MPI\_Send čeka da se poruka isporuči direktno u user bafer primaoca tj. MPI\_Send se blokira dok proces primaoc ne dođe do **MPI\_Recv** u svom izvršenju.

# Point-to-point komunikacija-deadlock

## Kako funkcioniše MPI\_Recv?

MPI\_Recv se vraća u proces tek kad poruka koja se očekuje da bude primljena bude u user baferu.

Situacija kada sigurno nastaje deadlock

P0	P1
MPI_Recv (from P1)	MPI_Recv (from P0)
MPI_Send (to P1)	MPI_Send (to P0)

Situacija kada zavisno od veličine sis.bafera nastaje deadlock

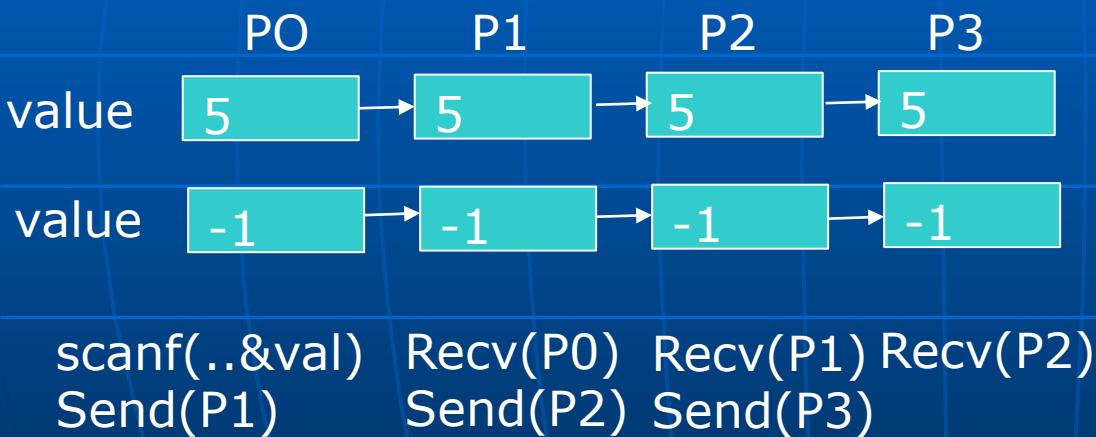
P0	P1
MPI_Send (to P1)	MPI_Send (to P0)
MPI_Recv (from P1)	MPI_Recv (from P0)

# Point-to-point komunikacija-bez deadlocka

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
    int myrank;
    MPI_Status status;
    int x,y;
    MPI_Init(&argc, &argv); /* Initialize MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get rank */
    if( myrank == 0 ) {
        x=3;
        MPI_Recv( &y, 1, MPI_INT, 1, 19, MPI_COMM_WORLD, &status
    );
        MPI_Send( &x, 1, MPI_INT, 1, 17, MPI_COMM_WORLD );
    }
    else if( myrank == 1 ) {
        x=5;
        MPI_Send(&x, 1, MPI_INT, 0, 19, MPI_COMM_WORLD );
        MPI_Recv(&y, 1, MPI_INT, 0, 17, MPI_COMM_WORLD, &status );
    }
    printf("Proc %d y= %d", myrank, y);
    MPI_Finalize();
}
```

# P-t-p komunikacija-zadaci

**zadatak.** Napisati program koji uzima podatke od nultog procesa i šalje ih svim drugim procesima tako što proces  $i$  treba da primi podatke i pošalje ih procesu  $i+1$ , sve dok se ne stigne do poslednjeg procesa. Unos podataka se završava nakon što se prenese negativna vrednost podatka.



# P-t-p komunikacija-zadaci

**zadatak.** Napisati program koji uzima podatke od nultog procesa i šalje ih svim drugim procesima tako što proces  $i$  treba da primi podatke i pošalje ih procesu  $i+1$ , sve dok se ne stigne do poslednjeg procesa. Unos podataka se završava nakon što se prenese negativna vrednost podatka.

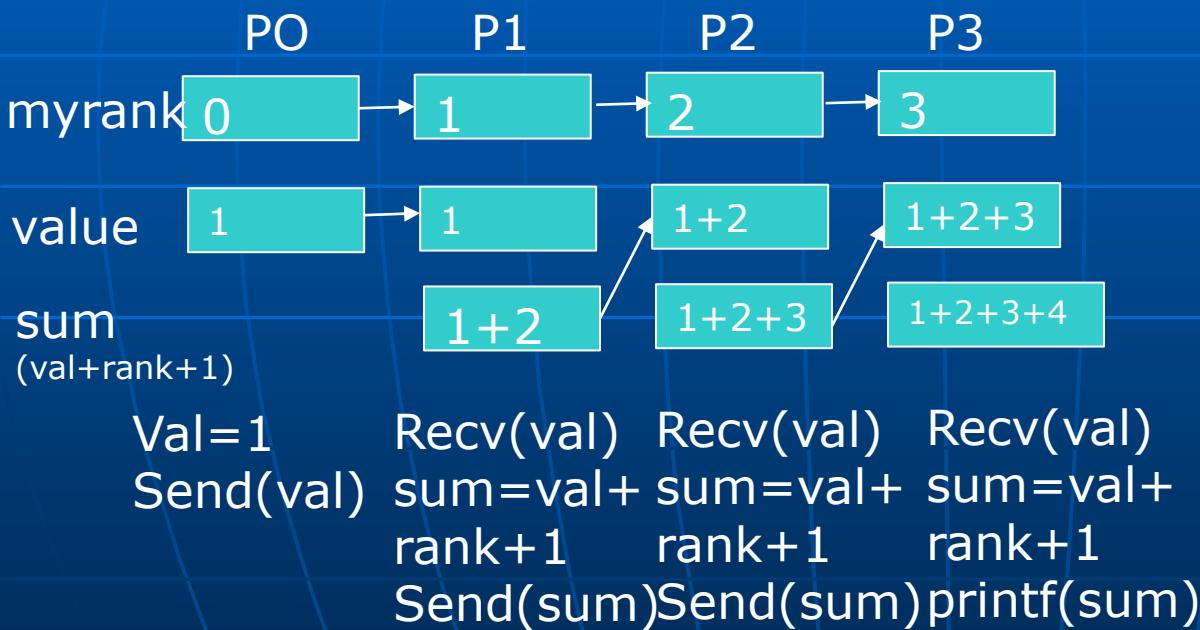
```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv )
{
    int rank, value, size;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    do {
        if (rank == 0) {
            scanf( "%d", &value );
            MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_
WORLD );
        }
    } while (value >= 0);
}
```

# P-t-p komunikacija-zadaci

```
else {
    MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
              &status );
    if (rank < size - 1)
        MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
    printf( "Process %d got %d\n", rank, value );
} while (value >= 0);
MPI_Finalize( );
return 0;
}
```

# P-t-p komunikacija-zadaci

**zadatak.** Napisati program koji nalazi sumu n celih brojeva korišćenjem ptp komunikacije tako da svaki proces učestvuje u sumiranju.



# P-t-p komunikacija-zadaci

```
#include <stdio.h>
#include "mpi.h"
int main(int argc,char *argv[])
{
    int MyRank, Numprocs;
    int value, sum = 0;
    int Root = 0;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&Numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&MyRank);
    if (MyRank == Root) {
        value=1;
        MPI_Send(&value, 1, MPI_INT, MyRank + 1, 0,
                 MPI_COMM_WORLD);
    }
}
```

# P-t-p komunikacija-zadaci

```
else{
    if(MyRank<Numprocs-1){
        MPI_Recv(&value, 1, MPI_INT, MyRank-1, 0,
                 MPI_COMM_WORLD, &status);
        sum = MyRank +1+ value;
        MPI_Send(&sum, 1, MPI_INT, MyRank+1, 0,
                 MPI_COMM_WORLD);
    }
    else{
        MPI_Recv(&value, 1, MPI_INT, MyRank-1, 0,
                 MPI_COMM_WORLD, &status);
        sum = MyRank +1+ value;
        printf("MyRank %d Final SUM %d\n", MyRank, sum);
    }
}
MPI_Finalize();
}
```

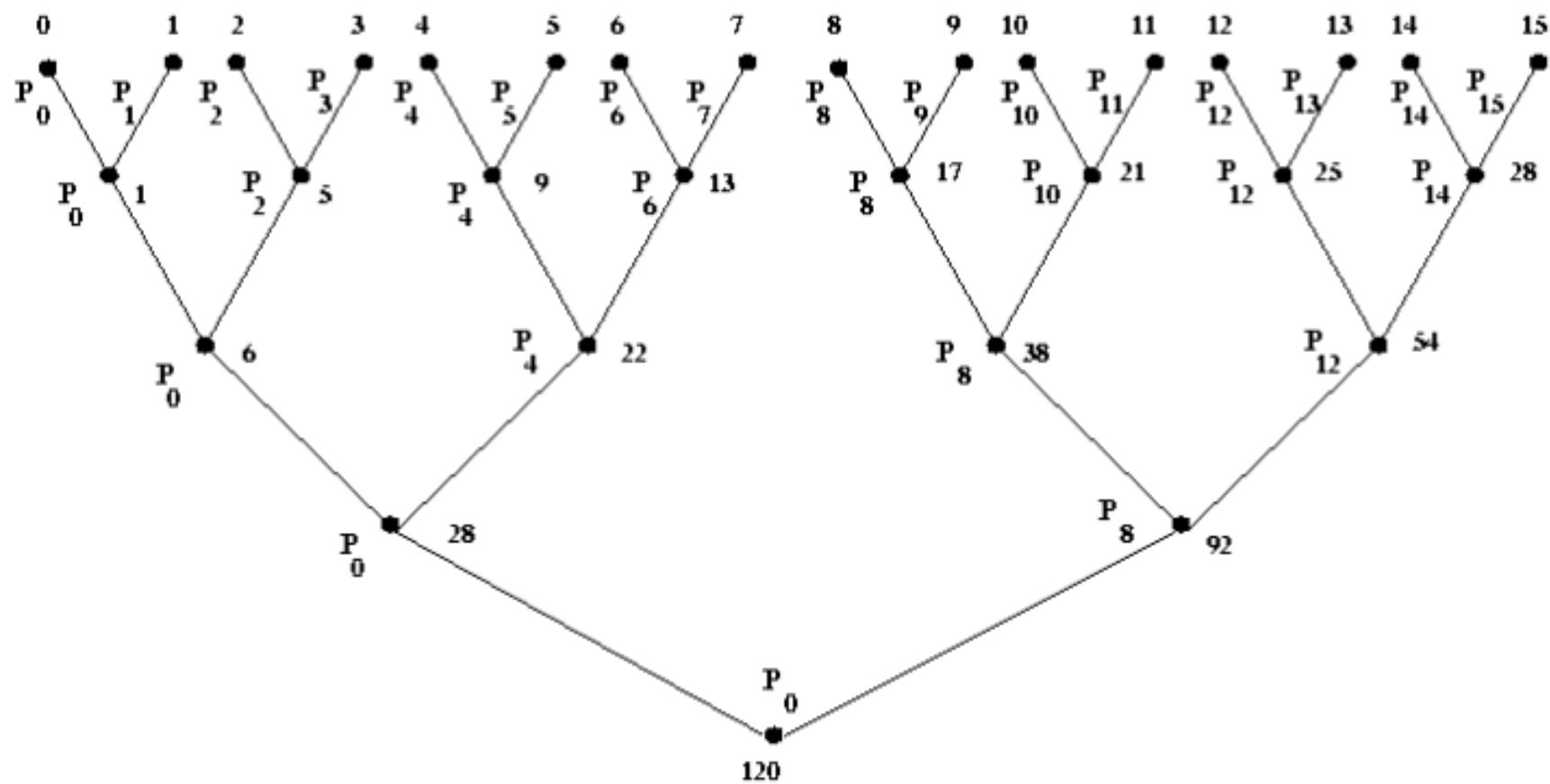
# P-t-p komunikacija-zadatak za vežbu

Napisati MPI program koji izračunava sumu N celih brojeva (N je stepen 2) korišćenjem Point-to-point komunikacije.

U prvom koraku, procesi se grupišu u parove ( $P_0, P_1$ ), ( $P_2, P_3$ ), ..., ( $P_{p-2}, P_{p-1}$ ). Zatim se izračunavaju parcijalne sume u svim parovima korišćenjem P-to-P komunikacije i akumuliraju u procesima ( $P_0, P_2, \dots, P_{p-2}$ ). Npr. process  $P_i$  (i-parno) izračunava parcijalne sume za par procesa ( $P_i, P_{i+1}$ ).

U sledećem koraku razmatraju se parovi procesa ( $P_0, P_2$ ), ( $P_4, P_6$ ), ..., ( $P_{p-4}, P_{p-2}$ ) pronađe parcijalne sume i akumuliraju u ( $P_0, P_4, \dots, P_{p-4}$ ). Postupak se ponavlja dok ne ostanu 2 procesa i rezultat se akumulira u  $P_0$ .

# P-t-p komunikacija-zadatak za vežbu

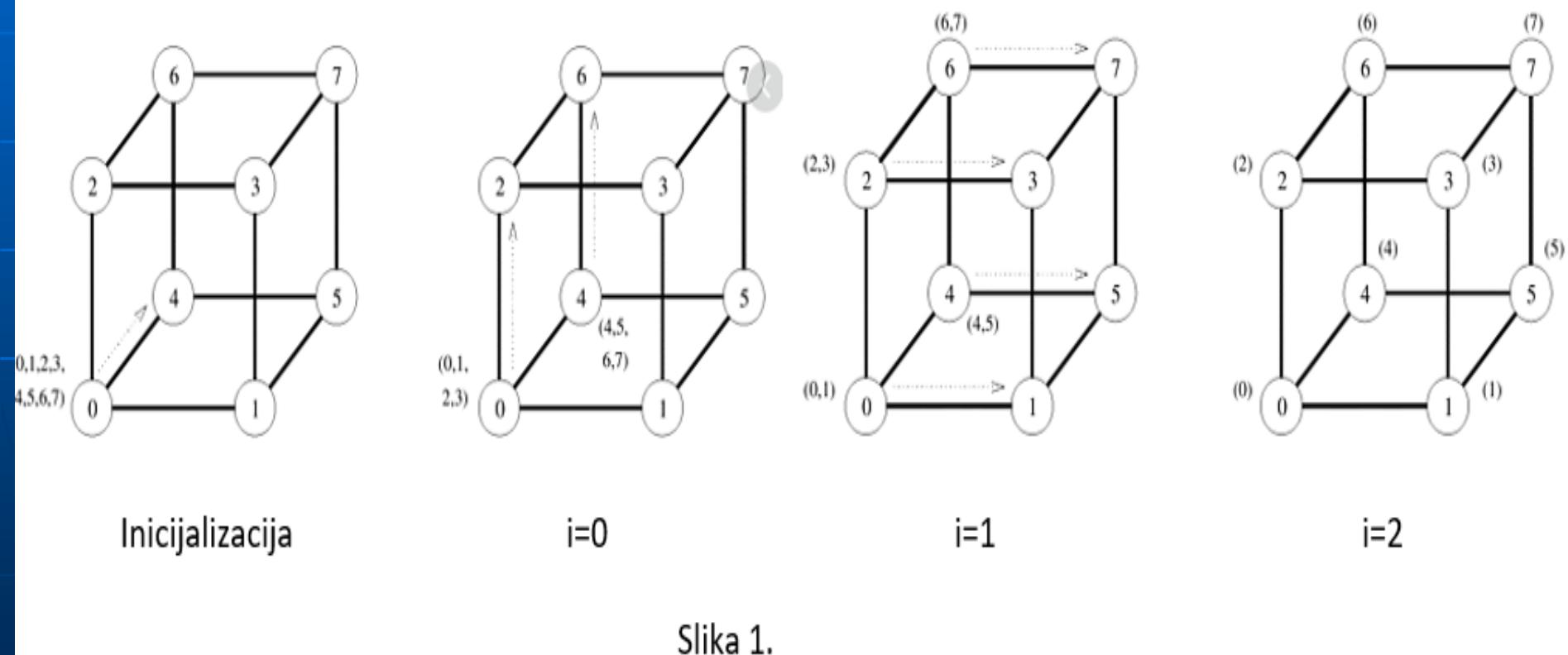


```

sum = MyRank;
Source_tag = 0;
Destination_tag = 0; NoofLevels=(int)(log(double)p)/log(2.0);
for(ilevel = 0 ; ilevel < NoofLevels ; ilevel++) {
    Level = (int)(pow((double)2, (double)ilevel));
    if((MyRank % Level) == 0){
        NextLevel = (int)(pow((double)2, (double)(ilevel+1)));
        if((MyRank % NextLevel) == 0){
            Source = MyRank + Level;
            MPI_Recv(&value, 1, MPI_INT, Source, Source_tag,
                    MPI_COMM_WORLD, &status);
            sum = sum + value;
        }
        else{
            Destination = MyRank - Level;
            MPI_Send(&sum, 1, MPI_INT, Destination, Destination_
tag, MPI_COMM_WORLD);
        }
    }
}
if(MyRank == Root)
    printf("%d Final SUM %d\n", MyRank, sum);

```

Napisati MPI program koji realizuje grupnu operaciju MPI\_Scatter, korišćenjem Point-to-Point komunikacije u kojoj učetvuje  $p$  procesa uređenih u hiperkub kao na Slici 1. Program se odvija u  $\log_2 p$  koraka. Nakon izvršenja programa svaki proces  $P_k$  ( $k=0, p-1$ ) treba da prikaže svoje podatke. Proses razmene podataka između procesa, prikazan je po koracima na Slici 1. U () zagradama se nalaze vrednosti koje se nalaze u svakom procesu, u odgovarajućem koraku.



Slika 1.

```
void main(int argc, char* argv[])
{
    int a[n];
    MPI_Status status;
    int rank,p,lev,nl,i,k,b,sum;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    if (rank==0)
        for (i = 0; i < n; i++) {
            a[i] = i;

    }
```

```

k=n;
for(i =(int)(log(double)p)/log(2.0)-1; i >= 0; i--)
{
    lev=pow(2.0,i);k=k/2;
    if (rank % lev ==0)
    {
        nl=pow(2.0,i+1);
        if (rank % nl ==0)
            MPI_Send(a+k,k,MPI_INT,rank+lev,0,MPI_COMM_WORLD);
        else
            MPI_Recv(a,k,MPI_INT,rank-lev,0,MPI_COMM_WORLD,&status);
    }
}

for (i = 0; i< k; i++)
{
    printf("Proces %d ima a[%d]=%d ",rank,i,a[i]);
    printf("\n");
}

```