Porazdeljeni sistemi

9.
MPI, 2. del

Predavatelja: izr. prof. Uroš Lotrič
Asistent: Davor Sluga

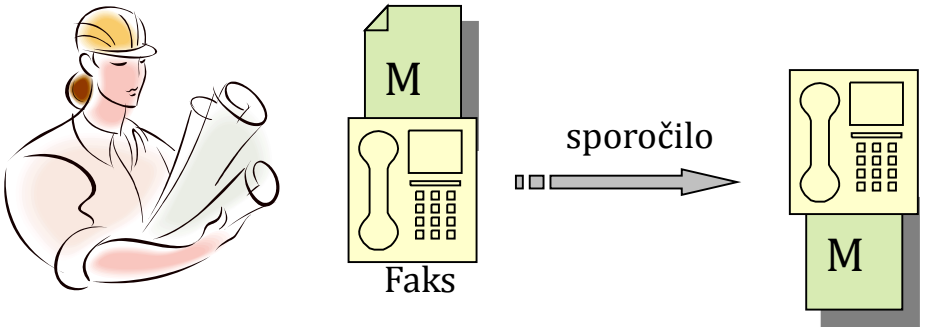
MPI: komunikacija točka – točka

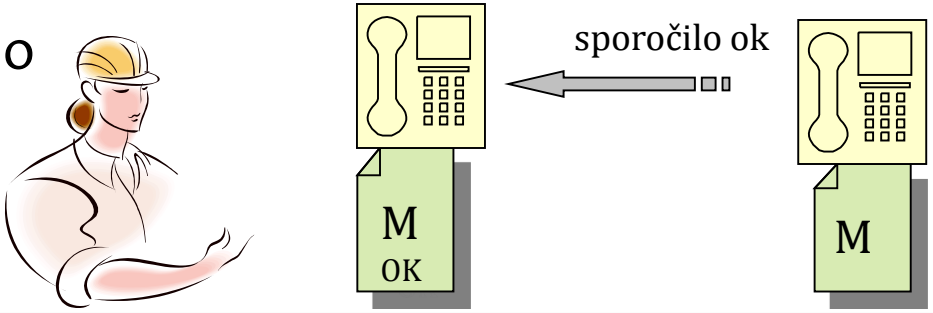
- Obstaja več različnih ukazov send/receive
- Delimo jih na:
  - sinhronizirane
  - medpomnilniške (buffered)
  - takojšnje (ready)
- Hkrati so rutine lahko
  - blokirajoče ali neblokirajoče (takojšnje, immediate)

- Med seboj lahko povezujemo en tip ukaza send z drugim tipom ukaza receive

MPI: komunikacija točka – točka

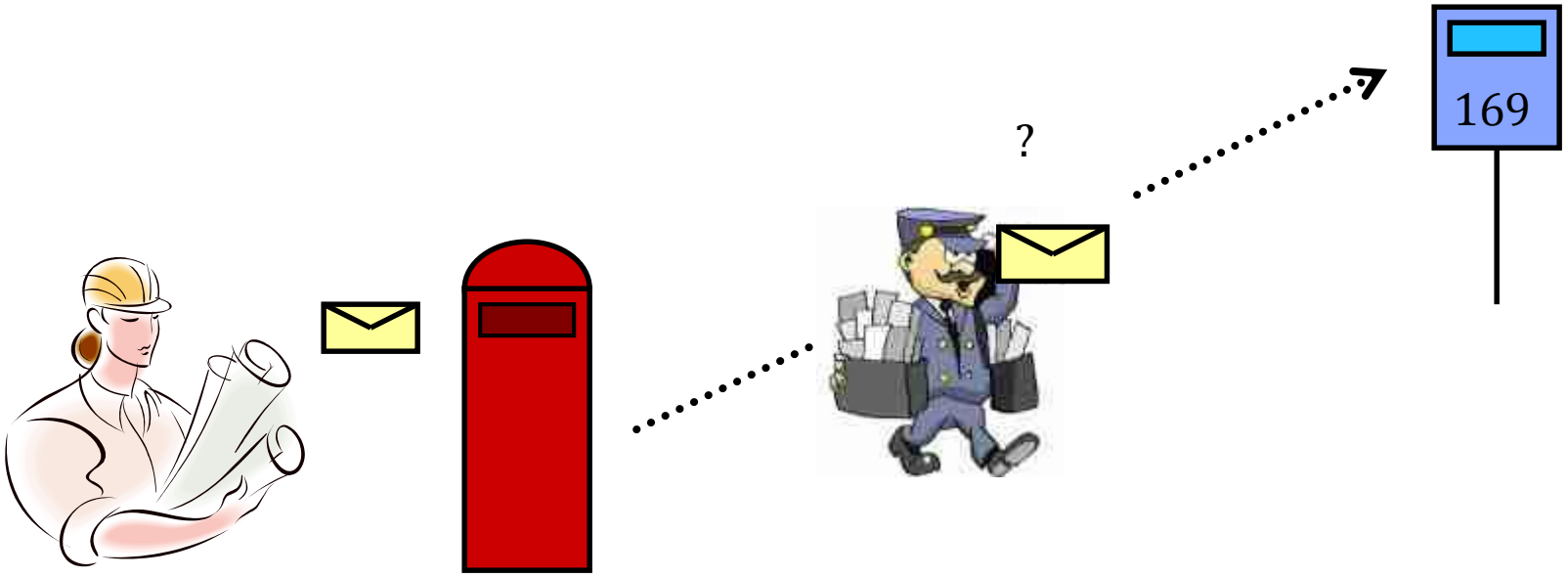
- Sinhroni send
  - vrne informacijo o uspešnosti pošiljanja
    - OK: podatki so bili uspešno poslani
    - `MPI_Ssend()` je sinhrono pošiljanje z blokado.
    - Pošiljanje sporočila se začne šele, ko sprejemnik potrdi, da ga lahko sprejme (protokol randezvous) – v medpomnilniku mora biti dovolj prostora za sprejem podatkov
    - Klic se zaključi, če je sporočilo dostavljeno ali je prišlo do napake





MPI: komunikacija točka – točka

- Send z uporabo medpomnilnika (buffered)
  - Klic se zaključi ne glede na to, kaj se je zgodilo s sporočilom
    - Podatki, ki jih pošiljamo, se prepišejo v medpomnilnik
    - `MPI_Bsend()` se zaključi takoj, ko se podatki prepišejo v medpomnilnik
    - Ne vemo ali je sporočilo prispelo do prejemnika ali ne

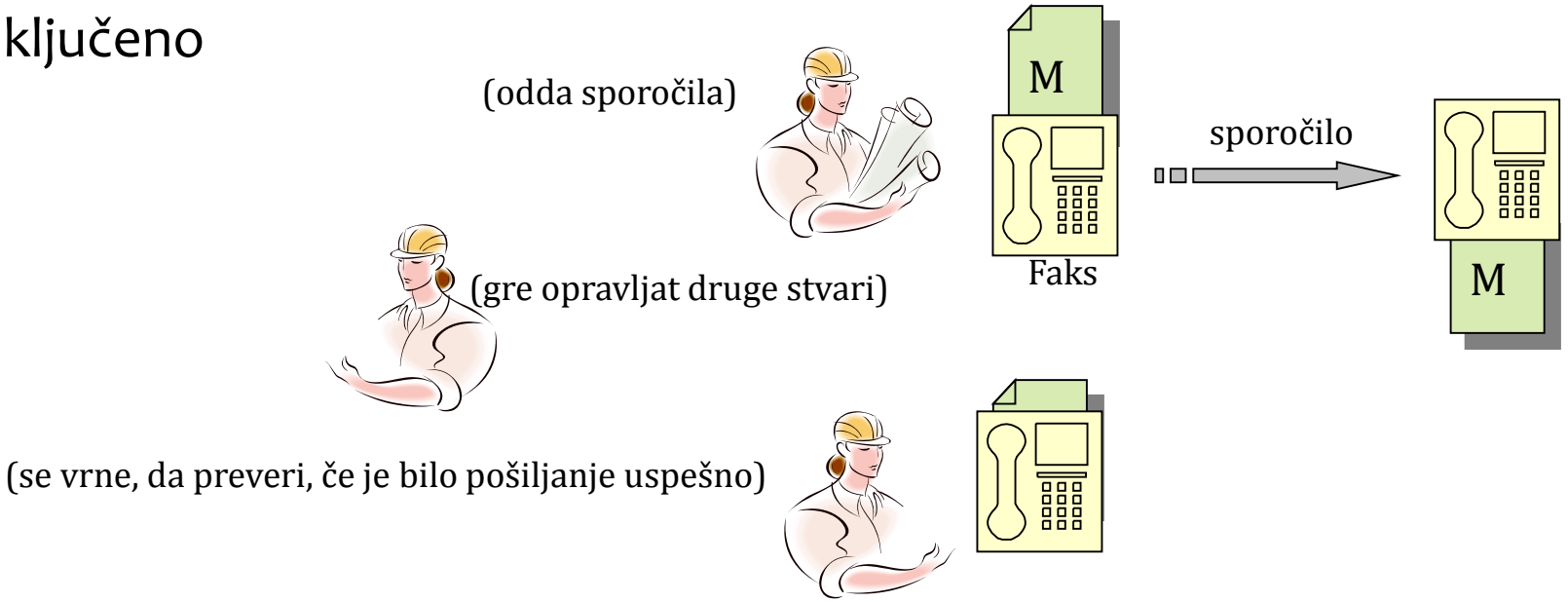


MPI: komunikacija točka – točka

- `MPI_Send()`
  - Standardni klic za pošiljanje sporočil
  - Lahko je sinhroni ali preko medpomnilnika, odvisno od implementacije MPI
    - krajša sporočila (< 16 kB) – `MPI_Bsend()`
    - daljša sporočila (> 16 kB) – `MPI_Ssend()`
  - V vsakem primeru gre za blokirajoči klic
  - Funkcija zaključi takoj, ko je varno prepisovati spremenljivke, ki smo jih pošiljali

MPI: komunikacija točka – točka

- Neblokirajoče (takojšnje) pošiljanje
  - Izvajanje funkcije se konča hitro po njenem klicu, program nadaljuje z izvajanjem naslednjih ukazov
  - Namen takojšnjega pošiljanja je prepletanje računanja in komunikacije in s tem prikrivanje latence
  - Kasneje lahko preverimo ali je pošiljanje uspelo
  - Novo pošiljanje lahko začnemo šele potem, ko je prejšnje zaključeno



MPI: komunikacija točka – točka

- Neblokirajoče pošiljanje
  - `MPI_Isend`
  - Vrne rokovalnik (handler), z uporabo katerega lahko kadarkoli preverimo uspešnost pošiljanja

```
MPI_Request request;
...
MPI_Isend(&buf, count, datatype, dest, tag, comm, request);
...
while(!done)
{
  do_stuff();
  ...
  MPI_Test(&request, &done, MPI_STATUS_IGNORE);
}
```

MPI: komunikacija točka – točka (povzetek)

	Funkcija	Opis
Blokirajoča	MPI_Send	Blokirajoče pošiljanje, lahko je sinhrono ali preko medpomnilnika. Klic funkcije se zaključi, ko je sporočilo varno na poti.
	MPI_Recv	Blokirajoče sprejemanje. Preprečuje nadaljnje izvajanje programa, dokler ne sprejme sporočila.
	MPI_Ssend	Sinhrono pošiljanje. Klic se zaključi, ko je sporočilo dostavljeno sprejemniku.
	MPI_Bsend	Pošiljanje preko medpomnilnika za pošiljanje. Klic se zaključi, ko je sporočilo skopirano v medpomnilnik.
Neblokirajoča (takojšnja)	MPI_Isend	Osnovno takojšnje pošiljanje. S klicem testne funkcije (MPI_Test()/MPI_Wait() izvemo ali je sporočilo varno na poti.
	MPI_Irecv	Takojšnje sprejemanje. Začne s sprejemanjem sporočila. S klicem testne funkcije MPI_Test()/MPI_Wait() izvemo, kdaj je bilo sporočilo sprejeto.
	MPI_Issend	Sinhrono takojšnje pošiljanje. S klicem testne funkcije MPI_Test()/MPI_Wait() izvemo, kdaj se sporočilo preneseno.
	MPI_Ibsend	Takojšnje pošiljanje preko medpomnilnika. S klicem funkcije MPI_Test()/MPI_Wait() izvemo, ali je sporočilo uspešno preneseno v medpomnilnik.
	MPI_Test	Preveri ali je katera od takojšnjih funkcij zaključena.
	MPI_Wait	Čaka, dokler se katera od takojšnjih funkcij ne zaključi.

MPI: komunikacija točka – točka

- Primer
  - Vsak proces prejme sporočilo od predhodnega procesa in odda svoje sporočilo naslednjemu procesu

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

#define buffsize 1000000

int main(int argc,char** argv)
{
    int             taskid, ntasks;
    int             i;
    int             *sendbuff, *recvbuff;
    double          inittime, totaltime;
    MPI_Request sendrequest, recvrequest;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

    sendbuff = malloc(sizeof(int)*buffsize);
    recvbuff = malloc(sizeof(int)*buffsize);

    for(i=0; i<buffsize; i++)
        sendbuff[i] = i;
```

MPI: komunikacija točka – točka

- Primer
  - `MPI_Send` in `MPI_Recv`
    - Pošiljanje začne proces 0

```
    MPI_Barrier(MPI_COMM_WORLD);
    inittime = MPI_Wtime();

    if(taskid == 0)
    {
        MPI_Send(sendbuff, buffsize, MPI_INT, taskid+1,
                 0, MPI_COMM_WORLD);
        MPI_Recv(recvbuff, buffsize, MPI_INT, ntasks-1,
                 MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    else
    {
        MPI_Recv(recvbuff, buffsize, MPI_INT, (taskid-1)%ntasks,
                 MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(sendbuff, buffsize, MPI_INT, (taskid+1)%ntasks,
                 0, MPI_COMM_WORLD);
    }

    MPI_Barrier(MPI_COMM_WORLD);
    totaltime = MPI_Wtime() - inittime;

    if(taskid == 0)
        printf("Cas prenosa: %f s\n", totaltime);

    free(recvbuff);
    free(sendbuff);

    MPI_Finalize();
}
```

MPI: komunikacija točka – točka

- Primer
  - `MPI_Isend` in `MPI_Irecv`
    - Pošiljati začnejo vsi hkrati, nato čakajo na sprejem
    - Koda je bolj enostavna
    - Hitrejše izvajanje

```
MPI_Barrier(MPI_COMM_WORLD);
inittime = MPI_Wtime();

MPI_Isend(sendbuff, buffsize, MPI_INT, (taskid+1+ntasks)%ntasks,
          0, MPI_COMM_WORLD, &sendrequest);
MPI_Irecv(recvbuff, buffsize, MPI_INT, (taskid-1+ntasks)%ntasks,
          MPI_ANY_TAG, MPI_COMM_WORLD, &recvrequest);

MPI_Wait(&sendrequest, MPI_STATUS_IGNORE);
MPI_Wait(&recvrequest, MPI_STATUS_IGNORE);

MPI_Barrier(MPI_COMM_WORLD);
totaltime = MPI_Wtime() - inittime;

if(taskid == 0)
    printf("Cas prenosa: %f s\n", totaltime);

free(recvbuff);
free(sendbuff);

MPI_Finalize();
}
```

MPI: komunikacija točka – točka

- Primer
  - `MPI_Sendrecv`
  - Če uporabimo sestavljeni ukaz, nam ni potrebno paziti na vrstni red!!!

```
    MPI_Barrier(MPI_COMM_WORLD);
    inittime = MPI_Wtime();

    MPI_Sendrecv(sendbuff, buffsize, MPI_INT, (taskid+1+ntasks)%ntasks, 0,
                 recvbuff, buffsize, MPI_INT, (taskid-1+ntasks)%ntasks, MPI_ANY_TAG,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    MPI_Barrier(MPI_COMM_WORLD);
    totaltime = MPI_Wtime() - inittime;

    if(taskid == 0)
        printf("Cas prenosa: %f s\n", totaltime);

    free(recvbuff);
    free(sendbuff);

    MPI_Finalize();
}
```

MPI: merjenje časa

- Za merjenje časa je priporočljivo uporabljati funkcije:
  - `double MPI_Wtime()`
    - Vrne čas v sekundah, merjen od neznanega trenutka v preteklosti
    - Z merjenjem razlike lahko zelo natančno določimo čas. Funkcija je mnogo bolj natančna kot vgrajene funkcije!
    - Razlika v času med dvema procesoma ni smiselna!!!
  - `double MPI_Wtick()`
    - Vrne natančnost ure v sekundah
  - `int MPI_Barrier(MPI_comm comm)`
    - Prepreka, s katero zahtevamo, da se vsi procesi počakajo (glej naprej)

MPI: skupinska komunikacija

- Obstaja več načinov
  - Sinhronizacija
    - Procesi čakajo, da vsi člani komunikatorja pridejo do določene točke v programu
  - Prenos podatkov
    - Oddajanje (broadcast), raztros in zbiranje, vsi vsem, …
  - Skupinsko računanje
    - Krčenje ali redukcija:
      en član skupine zbira podatke iz vseh ostalih procesov in hkrati izvaja računsko operacijo na teh podatkih

MPI: skupinska komunikacija

- Glavne lastnosti
  - Vse skupinske operacije so blokirajoče
  - Pri skupinskih operacijah ni zaznamkov
  - Uporabljamo jih lahko samo na podatkih podanih z osnovnimi podatkovnimi tipi
  - Za izvajanje skupinskih operacij na manjši skupini procesov moramo najprej definirati ustrezen komunikator, nato pa operacije izvesti na procesih v tem komunikatorju

MPI: skupinska komunikacija

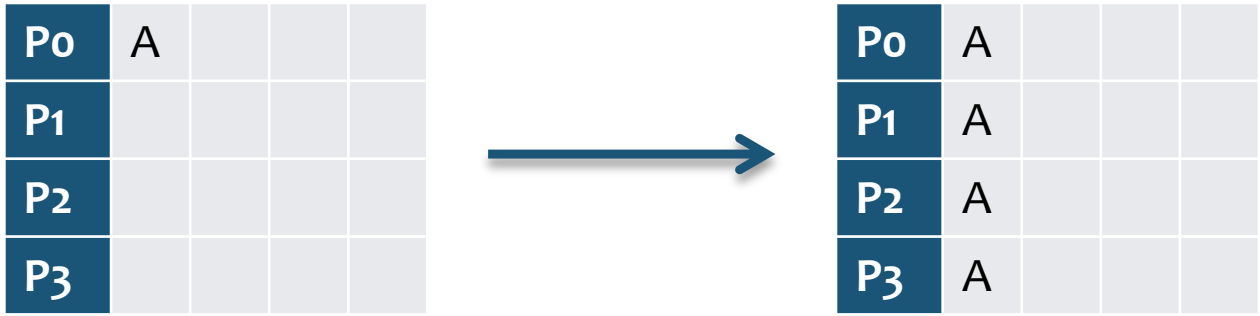
- Sinhronizacija
  - `int MPI_Barrier(MPI_comm comm)`
    - Prepreka, s katero zahtevamo, da se vsi procesi počakajo
    - Izvajanje programa za `MPI_Barrier` se nadaljuje šele potem, ko `MPI_Barrier` dosežejo vsi procesi
    - Klic funkcije `MPI_Barrier` morajo izvesti vsi procesi v komunikatorju

MPI: skupinska komunikacija

- Oddajanje (broadcast)
  - `int MPI_Bcast( void* buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
  - Pošiljanje sporočila s procesa z rankom `root` vsem ostalim procesom v komunikatorju
  - `root` označuje korenski proces
  - Za korenski proces je polje `buf` vhodni parameter, pri vseh ostalih procesih pa izhodni parameter
  - Postopek oddajanja izvajajo vsi procesi, ne samo korenski!!!
  - Vsi ostali procesi morajo tudi klicati `MPI_Bcast`, pri čemer je z argumentom `root` naveden korenski proces!!!

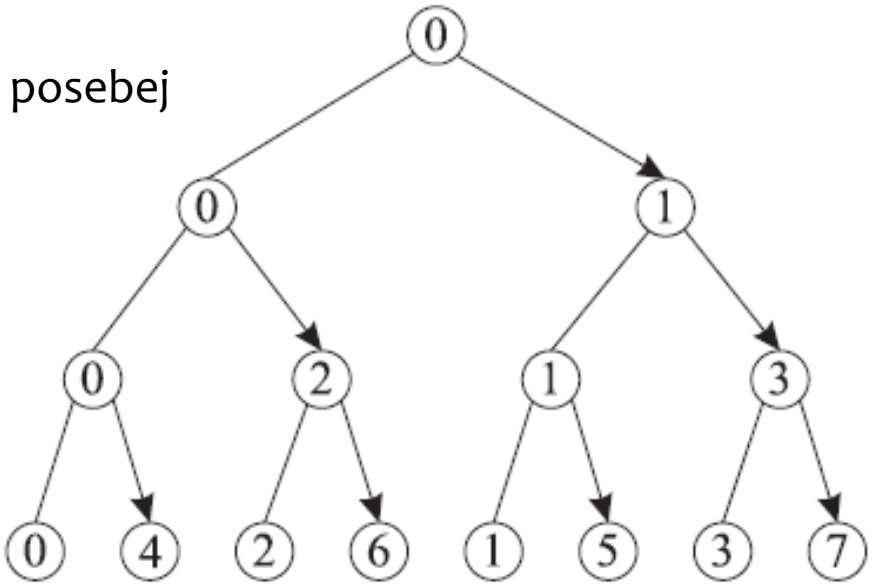
MPI: skupinska komunikacija

- Oddajanje (broadcast)
  - Shematično



MPI: skupinska komunikacija

- Oddajanje (broadcast)
  - Izvedba ukaza je odvisna od strojne opreme.
    - Drevesna struktura
      - Veliko bolj učinkovito kot vsakemu posebej
        - 0 → 1
        - 0 → 2, 1 → 3
        - 0 → 4, 2 → 6, 1 → 5, 3 → 7
        - …
    - Myrinet, Infiniband: posebej prilagojeni sistemi, ki znajo take ukaze učinkovito izvajati
    - Ethernet: knjižnica lahko izkorišča obstoječe mehanizme za oddajanje



MPI: skupinska komunikacija

- Oddajanje (broadcast)
  - Primer
    - Proces 0 zahteva vnos vrednosti
    - Vrednost se z `MPI_Bcast` prenese na vse ostale procese

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

int main(int argc, char* argv[])
{
    int  id, value;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);

    if (id == 0)
    {
        printf("Vnesi vrednost: ");
        fflush(stdout);             // zahteva takojšen izpis
        scanf("%d", &value);
    }

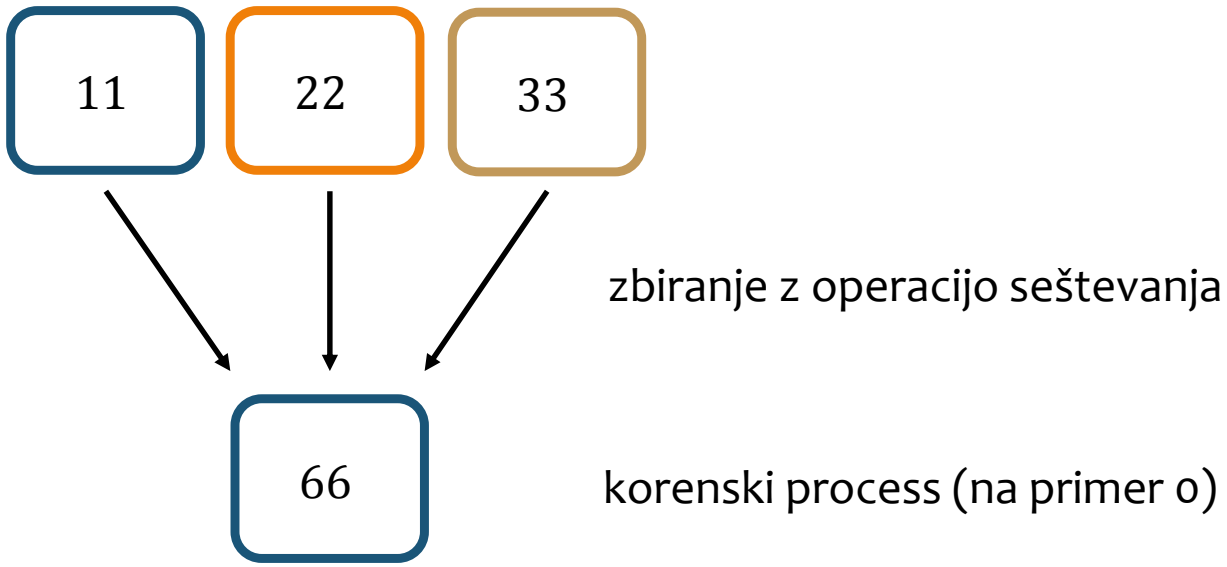
    // MPI_Bcast kličejo vsi procesi na enak način !!!
    MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Proces %d: value = %d\n", id, value);

    MPI_Finalize();
}
```

MPI: skupinska komunikacija

- Krčenje (redukcija)
  - `int MPI_Reduce(void* sendbuf, void* recvbuf,`
    `int count, MPI_Datatype datatype,`
    `MPI_Op op, int root, MPI_Comm comm)`
  - Podatki z vseh procesov se zbirajo na izbranem procesu z upoštevanjem izbrane računske operacije



MPI: skupinska komunikacija

- Krčenje (redukcija)
  - Bolj smiselno kot da podatke sprejema korenski proces od vseh ostalih je, da se uporabi obratna drevesna struktura – začnemo na listih in nadaljujemo do korena
  - Podprte operacije:

Operacija	Pomen
MPI_MAX	Maksimum
MPI_MIN	Minimum
MPI_SUM	Vsota
MPI_PROD	Produkt
MPI_LAND	Logični in
MPI_BAND	In na bitih
MPI_LOR	Logični ali
MPI_BOR	Ali na bitih
MPI_LXOR	Logični ekskluzivni ali
MPI_BXOR	Ekskluzivni ali na bitih
MPI_MAXLOC	Maksimum in lokacija
MPI_MINLOC	Maksimum in lokacija

MPI: skupinska komunikacija

- Krčenje (redukcija)
  - Primer
    - seštevanje vrednosti spremenljivke `valuesend` na vseh procesih

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

int main(int argc, char* argv[])
{
    int     id;
    int     valuesend, valuerecv;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);

    valuesend = id;

    printf("Proces %d: value = %d\n", id, valuesend);

    // MPI_Reduce kličejo vsi procesi na enak način !!!
    MPI_Reduce(&valuesend, &valuerecv, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

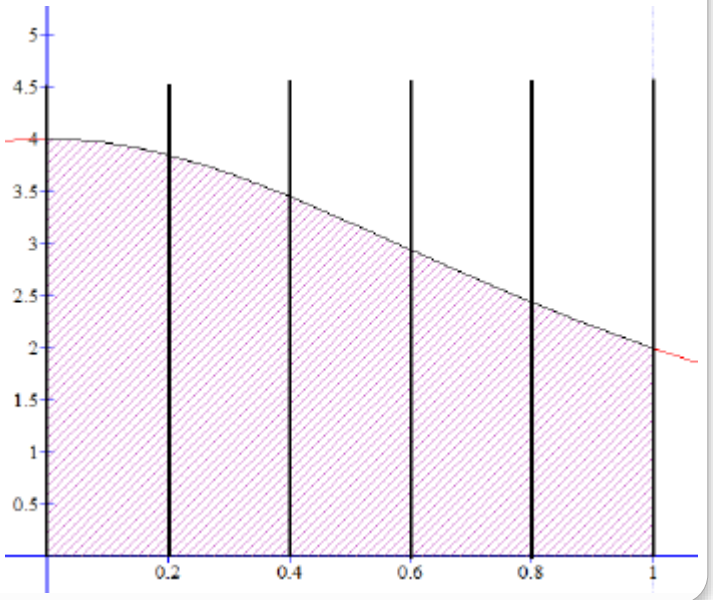
    if (id == 0)
        printf("Vsota vrednosti z vseh procesov je %d\n", valuerecv);

    MPI_Finalize();
}
```

MPI: skupinska komunikacija

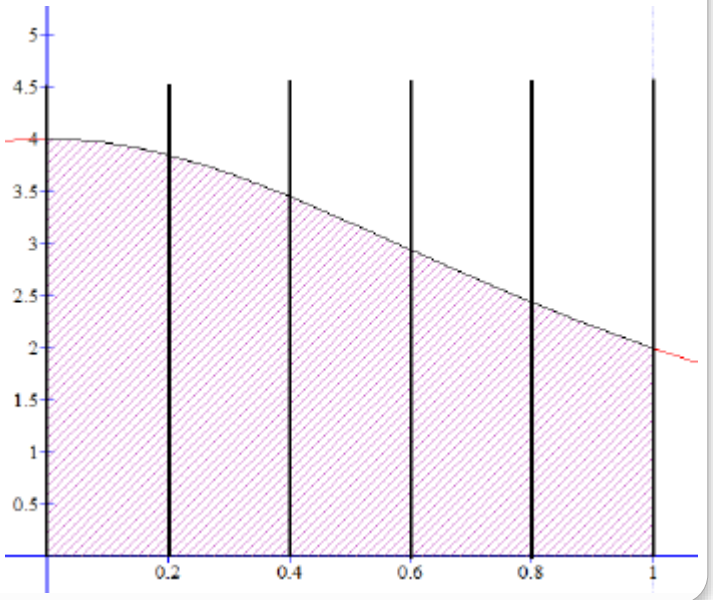
- Primer: računanje števila π
  - Računamo določeni integral $\int_0^1 \frac{4}{1+x^2} dx = \pi$
  - Uporabimo metodo pravokotnikov
  - Ideja za paralelizacijo:
    celotno območje integracije enakomerno razdelimo na toliko manjših območij, kolikor je procesov

razdelitev pri 5 procesih →



MPI: skupinska komunikacija

- Primer: računanje števila π
  - Postopek
    - V osnovnem procesu vpišemo število pravokotnikov
    - Število vseh pravokotnikov razpošljemo vsem procesom
    - Vsak proces izračuna svoj del integrala
    - Na koncu se vsi prispevki seštejejo na korenskem procesu



MPI: skupinska komunikacija

- Primer: računanje števila π
  - Program

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

int main(int argc, char* argv[])
{
    int myid, procs;
    int n;
    double h, mystart, mystop;
    double x, mysum, sum;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &procs);

    if(myid == 0)
    {
        printf("Stevilo pravokotnikov = ");
        fflush(stdout);
        scanf("%d", &n);
        h = 1.0/n;
    }

    MPI_Bcast(&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    mystart = myid/(double)procs;
    mystop = (myid+1)/(double)procs;

    mysum = 0.0;
    x = mystart;
    while (x < mystop)
    {
        mysum += 4.0/(1+x*x)*h;
        x += h;
    }

    MPI_Reduce(&mysum, &sum, 1, MPI_DOUBLE,
               MPI_SUM, 0, MPI_COMM_WORLD);

    if( myid == 0)
        printf("Pi = %lf\n", sum);

    MPI_Finalize();
}
```

MPI: skupinska komunikacija

- Raztros in zbiranje
  - Shematično

  - Z raztrosom lahko enostavno porazdelimo podatke na več procesov
  - Z zbiranjem lahko podatke iz več procesov zberemo na enem samem

  - Pri klicu funkcij za raztros in zbiranje so določeni parametri veljavni samo na strani pošiljatelja, določeni pa samo na strani sprejemnika

MPI: skupinska komunikacija

- Raztros in zbiranje
  - ```
    int MPI_Scatter(void *sendbuf, int sendcount,
                    MPI_Datatype sendtype,
                    void *recvbuf, int recvcount,
                    MPI_Datatype recvtype,
                    int root, MPI_Comm communicator )
    ```
  - ```
    int MPI_Gather( void *sendbuf, int sendcount,
                    MPI_Datatype sendtype,
                    void *recvbuf, int recvcount,
                    MPI_Datatype recvtype,
                    int root, MPI_Comm communicator )
    ```
  - Del podatkov ob raztrosu dobi tudi korenski proces
  - Pri zbiranju je potrebno upoštevati, da je del podatkov tudi na korenskem procesu

MPI: skupinska komunikacija

- Primer: skalarni produkt
  - Inicializacija okolja MPI
  - pripravljanje vektorjev A in B

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv)
{
    int     procs, myid;
    int     scatterSize;
    int     i;
    double  *vectorA, *vectorB;
    double  *myvectorA, *myvectorB;
    double  mydotProduct, dotProduct, *dotProducts;
    int     vectorSize = 100;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &procs);

    vectorA = NULL;
    vectorB = NULL;
    if(myid == 0)
    {
        vectorA  = (double *) malloc(vectorSize*sizeof(double));
        vectorB  = (double *) malloc(vectorSize*sizeof(double));
        for(i=0; i<vectorSize; i++)
        {
            vectorA[i] = i/2.0;
            vectorB[i] = i;
        }
    }
```

MPI: skupinska komunikacija

- Primer: skalarni produkt
  - pošiljanje podatkov
  - računanje skalarnega produkta na lastnem delu vektorja

```
MPI_Bcast(&vectorSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

if((vectorSize < procs) || (vectorSize % procs != 0))
{
    MPI_Finalize();
    if(myid == 0)
        printf("Število elementov ni večkratnih procesov...\n");
    exit(0);
}

scatterSize = vectorSize / procs;
myvectorA = (double *)malloc(scatterSize * sizeof(double));
myvectorB = (double *)malloc(scatterSize * sizeof(double));

MPI_Scatter( vectorA, scatterSize, MPI_DOUBLE,
             myvectorA, scatterSize, MPI_DOUBLE,
             0, MPI_COMM_WORLD);
MPI_Scatter( vectorB, scatterSize, MPI_DOUBLE,
             myvectorB, scatterSize, MPI_DOUBLE,
             0, MPI_COMM_WORLD);

mydotProduct = 0.0;
for(i=0; i<scatterSize; i++)
    mydotProduct += (myvectorA[i] * myvectorB[i]);
```

MPI: skupinska komunikacija

- Primer: skalarni produkt
  - Končni izračun z zbiranjem (gather) in seštevanjem

```
    dotProducts = NULL;
    if( myid == 0)
        dotProducts = (double *)malloc(procs*sizeof(double));

    MPI_Gather( &mydotProduct, 1, MPI_DOUBLE,
                dotProducts, 1, MPI_DOUBLE,
                0, MPI_COMM_WORLD);
    if(myid == 0)
    {
        dotProduct = 0.0;
        for( i=0; i<procs; i++)
            dotProduct += dotProducts[i];
        printf("Skalarni produkt: %f\n", dotProduct);
    }

    free(vectorA);
    free(vectorB);
    free(myvectorA);
    free(myvectorB);

    MPI_Finalize();
}
```

MPI: skupinska komunikacija

- Primer: skalarni produkt
  - Končni izračun s krčenjem (reduce)

```
    MPI_Reduce( &mydotProduct, &dotProduct, 1, MPI_DOUBLE,
                MPI_SUM, 0, MPI_COMM_WORLD);

    if(myid == 0)
        printf("Skalrni produkt: %f\n", dotProduct);

    free(vectorA);
    free(vectorB);
    free(myvectorA);
    free(myvectorB);

    MPI_Finalize();
}
```

MPI: skupinska komunikacija

- Raztros in zbiranje
  - Pri `MPI_Scatter()` in `MPI_Gather()` imajo kosi, raztrošeni po procesih, enako število elementov
  - Večkrat nam to zaradi narave podatkov ne ustreza. V tem primeru uporabimo funkciji `MPI_Scatterv()` in `MPI_Gatherv()`
  - Število elementov na posameznem procesu je različno (`MPI_...v` kot variable)

MPI: skupinska komunikacija

- Raztros in zbiranje bolj splošno
  - ```
    int MPI_Scatterv( void *sendbuffer,
                      int *sendcount,
                      int *senddisplacement,
                      MPI_Datatype sendtype,
                      void *receivebuffer,
                      int receivecount,
                      MPI_Datatype receivetype,
                      int root, MPI_Comm comm)
    ```
  - Dodatno na procesu `root`
    - `*sendcount` – polje, v katerem je za vsak proces navedeno število dodeljenih elementov
    - `*senddisplacement` – polje, v katerem je za vsak proces naveden položaj prvega dodeljenega elementa glede na elemente v `sendbuffer`
    ```

MPI: skupinska komunikacija

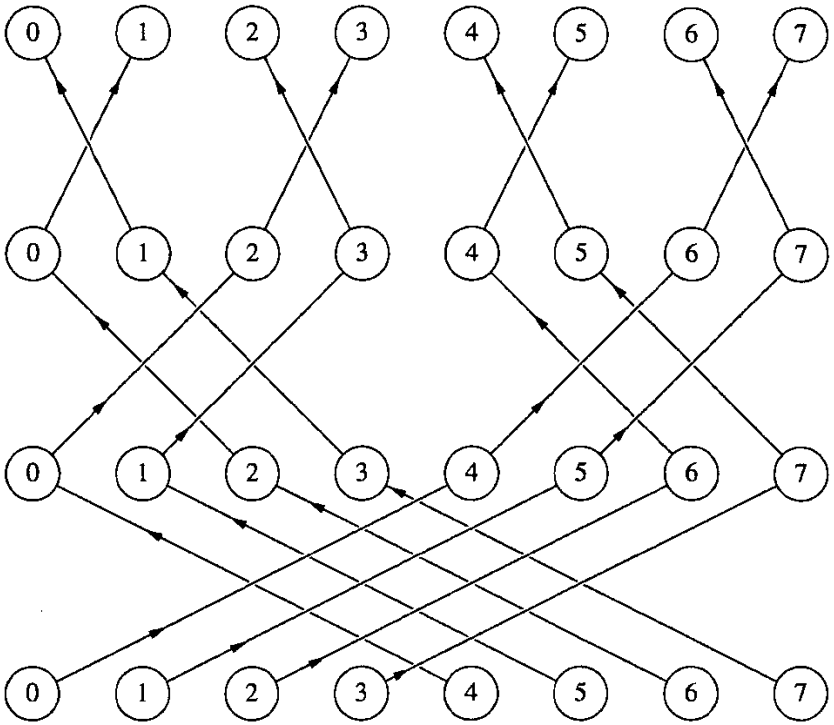
- Raztros in zbiranje bolj splošno
  - `int MPI_Gatherv( void *sendbuffer,`
    `int sendcount,`
    `MPI_Datatype sendtype,`
    `void *receivebuffer,`
    `int *receivecount,`
    `int *recivedisplacement,`
    `MPI_Datatype receivetype,`
    `int root, MPI_Comm comm)`
  - Dodatno na procesu `root`
    - `*receivecount` – polje, v katerem je za vsak proces navedeno koliko elementov bo poslal
    - `*receivedisplacement` – polje, v katerem je za vsak proces naveden položaj prvega elementa v `receivebuffer`, kamor se morajo prenesti podatki

MPI: skupinska komunikacija

- Še enkrat krčenje
  - Kaj, če bi rezultat krčenja za nadaljnje računanje potrebovali vsi procesi?
  - Rešitev
    - `MPI_Reduce(&mysum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)`
    - `MPI_Bcast(&sum, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD)`
    - Dvakrat komuniciramo. V najslabšem primeru dvakrat uporabimo drevesno topologijo. Je potrebno?

MPI: skupinska komunikacija

- Še enkrat krčenje
  - Kaj, če bi rezultat krčenja za nadaljnje računanje potrebovali vsi procesi?



  - Boljša rešitev
    - `MPI_Allreduce(&mysum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)`
  - Podobna ideja: krčenje in raztros
    - `MPI_Reduce_scatter(...)`

MPI: skupinska komunikacija

- `MPI_All...`
  - `MPI_Allgather()`, `MPI_Allgatherv()`

  - `MPI_Alltoall()`, `MPI_Alltoallv()`

MPI: medpomnjenje, sinhronizacija, ...

- MPI na Ethernetu (TCP/IP)
  - Servis MPI teče v aplikacijski plasti nad sejno, transportno in mrežno plastjo TCP/IP
  - Naše aplikacije so še eno plast višje, saj kličejo servis MPI v aplikacijski plasti
  - Prenos
    - Ob MPI_Init() se vzpostavi dvosmerni kanal TCP/IP med procesoma A in B v obliki para socket-ov
    - Ko proces A kliče MPI_Send() bo MPI pisal na sklad TCP/IP
    - Sklad TCP/IP na A bo prenesel podatke na sklad TCP/IP na B.
    - Sklad TCP/IP na B bo prenesel podatke na MPI procesa B

MPI: medpomnjenje, sinhronizacija, ...

- MPI na Ethernetu (TCP/IP)
  - Prenos (podrobnosti)
    - S stališča TCP/IP je vsa komunikacija med dvema procesoma zgleda kot eno samo dolgo sporočilo
      - Če proces A 5-krat pokliče MPI_Send, bo MPI 5-krat zapisal na sklad TCP/IP, ki pa tega ne bo razumel kot 5 ločenih sporočil ampak samo kot del zelo dolgega sporočila
    - Proces B neprestano bere dolgo sporočilo in ga reže na sporočila MPI.
    - Sporočila MPI so na procesu B potem pripravljena, da jih program prebere z MPI_Recv()
      - Vrstni red sporočil je lahko drugačen kot ga zahteva proces B!!!
      - Če sporočilo že obstaja, se MPI_Recv() zaključi
      - **Če sporočila še ni, MPI_Recv() čaka, da se pojavi in s tem blokira nadaljnje izvajanje programa!!!**

MPI: medpomnjenje, sinhronizacija, ...

- Infiniband
  - Kopiranje v medpomnilnik (sklad TCP/IP) upočasni komunikacijo
  - Boljše rešitve se temu izogibajo, običajno z oddaljenim neposrednim dostopom do pomnilnika (remote direct memory access)
    - Proces A lahko piše neposredno v pomnilnik procesa B
    - Če uporabljamo sinhrono komunikacijo, bo v tem primeru trajalo še dlje

MPI: medpomnjenje, sinhronizacija, ...

- Varnost
  - Smrtni objem
    - A želi poslati sporočilo B, B želi poslati sporočilo A
    - A čaka, da B sprejme sporočilo in obratno
    - A želi poslati B najprej sporočilo U, potem še sporočilo V
    - B želi najprej sprejeti sporočilo V, potem še sporočilo U
  - **Uporaba takojšnjih rutin je s stališča MPI nevarna**
    - Program lahko pravilno teče na mnogih sistemih
    - Na določenih sistemih lahko pride do težav pri oddajanju/sprejemanju sporočil zaradi velikosti medpomnilnika
    - Ker je delovanje hitrejše je večkrat vredno sprejeti to tveganje

Dobro je vedeti

- Opis funkcij MPI s primeri

  http://mpi.deino.net/mpi_functions/index.htm