OpenMPI — део 1

Рачунарски системи високих перформанси

Петар Трифуновић Вељко Петровић

Факултет техничких наука
Универзитет у Новом Саду

Рачунарске вежбе, Зимски семестар 2022/2023.

Message Passing Interface (MPI)

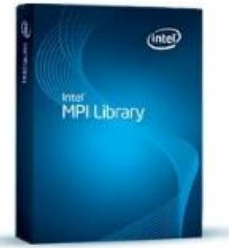
- Стандард који прописује комуникацију разменом порука на различитим паралелним архитектурама.
  - **MPI 1.x** (1994.)
  - **MPI 2.x** (1997.)
  - **MPI 3.x** (2012.)
  - **MPI 4.x** (2018.)
- Подршка за C (C++), Fortran.
- Постоје различите имплементације *MPI* стандарда (комерцијалне и отвореног кода).

MPI имплементације

OPEN MPI

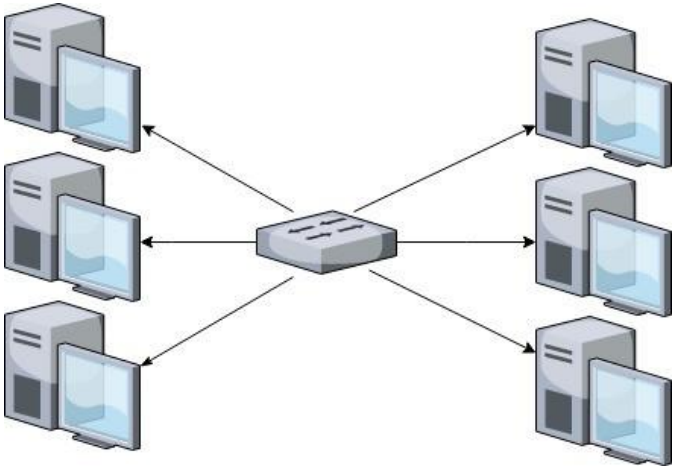
MVAPICH

MPICH-V
MPI Implementation for Volatile resources



IBM Spectrum MPI

MPI циљна архитектура



- Рачунари (процеси) повезани мрежом преко које комуницирају ради извршавања посла у паралели.
- Посао се по рачунарима (процесима) расподељује разменом порука.

MPI и OpenMP — разлике

OpenMP	MPI
Скуп компајлерских директива, библиотечких рутина и променљивих окружења	Стандард који има различите имплементације
Паралелизација на нивоу нити	Паралелизација на нивоу процеса
Намењен за системе са дељеном меморијом	Намењен за системе са дистрибуираном меморијом
Комуникација преко дељене меморије	Комуникација разменом порука преко мреже (мада новије имплементације могу користити и дељену меморију)

OpenMPI

- Имплементација *MPI* стандарда.
- Отвореног кода.
- Имплементација за *C/C++* и *Fortran*.
- https://github.com/open-mpi/ompi

Формат OpenMPI програма

```
#include <mpi.h>
int main(int argc, char *argv[]) {

    MPI_Init(&argc, &argv);

    // MPI code

    MPI_Finalize();

    return 0;

}
```

Компајлирање MPI програма

- Позиционирати се у директоријум у којем се налази изворни код *MPI* програма и унети:

```
mpicc <izvorna_datoteka>
```

- `mpicc` је омотачка скрипта за `gcc`, па се при компајлирању могу навести опције `gcc` компајлера.

- Покретање:

```
mpiexec [-np <N>] [--bind-to core|hwthread] <izvrsna_datoteka>
```

- **-np <N>** — опција за задавање броја процеса који ће бити креирани при покретању програма.
- **--bind-to** — опција којом се број процеса који ће бити креирани при покретању програма везује или за број физичких (`core`) језгара, или за број логичких (`hwthread`) језгара процесора.

Компајлирање MPI програма

```
mpiexec [-np <N>] [--bind-to core|hwthread] <izvrsna_datoteka>
```

- комбинација **-np <N>** и **--bind-to** опција:
  - **--bind-to core** — ако је **<N>** мање или једнако броју **физичких** језгара процесора, биће покренуто **<N>** процеса; ако је **<N>** веће, доћи ће до грешке
  - **--bind-to hwthreads** — исто као **core** опција, само се **<N>** упоређује са бројем **логичких** језгара
  - **-np <N>** без **--bind-to** даће грешку уколико **<N>** буде веће од броја **физичких** језгара
- **--oversubscribe** — у комбинацији са **-np <N>** уклања ограничења за вредност параметра **<N>**

Пример 1: Hello World!

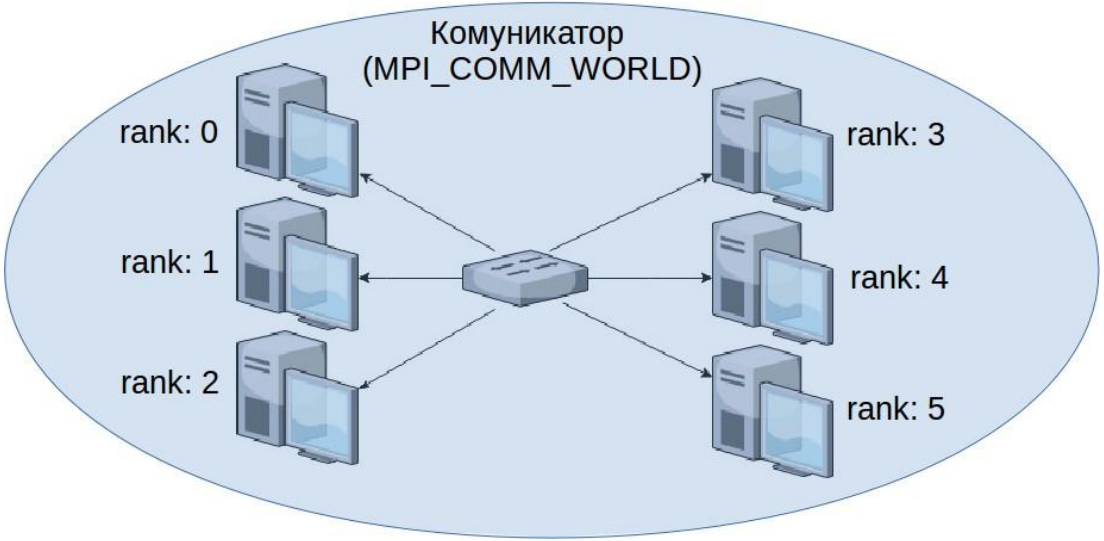
```c
#include <stdio.h>
#include <mpi.h>

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

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank;

    printf("Hello World iz %d/%d.\n", rank, size);
    MPI_Finalize();

    return 0;
}
```

MPI основни концепти

- Комуникатор (енг. *communicator*)
- Point to Point комуникација (енг. *Point-to-Point communication*)
- Колективна комуникација (енг. *Collective communication*)

Комуникатори

- Логички гледано, **комуникатор** представља групу процеса унутар које сваки процес има свој **ранк** (односно идентификатор).



- **MPI_COMM_WORLD** — комуникатор у ком се налазе сви покренути процеси
- **MPI_COMM_SELF** — сваки процес је једини процес у свом приватном SELF комуникатору
- **MPI_COMM_NULL** — комуникатор у коме се конкретан процес не налази

Комуникатори

- Током извршавања *MPI* истовремено може да постоји више комуникатора.
- `MPI_Comm` тип података.
- Функције за прављење комуникатора:
  - *креирање новог комуникатора дељењем постојећег*

    ```
    MPI_Comm_split(MPI_Comm comm, int color, int key,
        MPI_Comm* newcomm);
    ```
  - *нови комуникатор је копија постојећег*

    ```
    int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
    ```
  - *и друге*

Комуникатори — *MPI_Comm_split*

- Дели прослеђени комуникатор на онолико нових колико има различитих вредности параметра *color* по процесима.
- Параметар *color* — сви процеси који позову *MPI_Comm_split* са истом вредношћу овог параметра наћи ће се у истом резултујућем комуникатору.
- Параметар *key* — процеси ће у новом комуникатору бити рангирани на основу вредности овог параметра; ако два процеса исте боје (*color*) имају исту вредност параметра *key*, ранк ће се одредити на основу ранка из комуникатора над којим се врши подела.

Задатак 1: Комуникатори

- Направити OpenMPI C програм који коришћењем функције `MPI_Comm_split` на основу подразумеваног прави два нова комуникатора. Процесе поделити у два комуникатора на основу парности ранка унутар `MPI_COMM_WORLD` комуникатора. Притом релативни поредак процеса унутар комуникатора треба да буде исти као и у подразумеваном комуникатору. Сваки процес на стандардни излаз треба да испише свој ранк унутар `MPI_COMM_WORLD` и новоформираног комуникатора.

- **Пример исписа за један процес:**

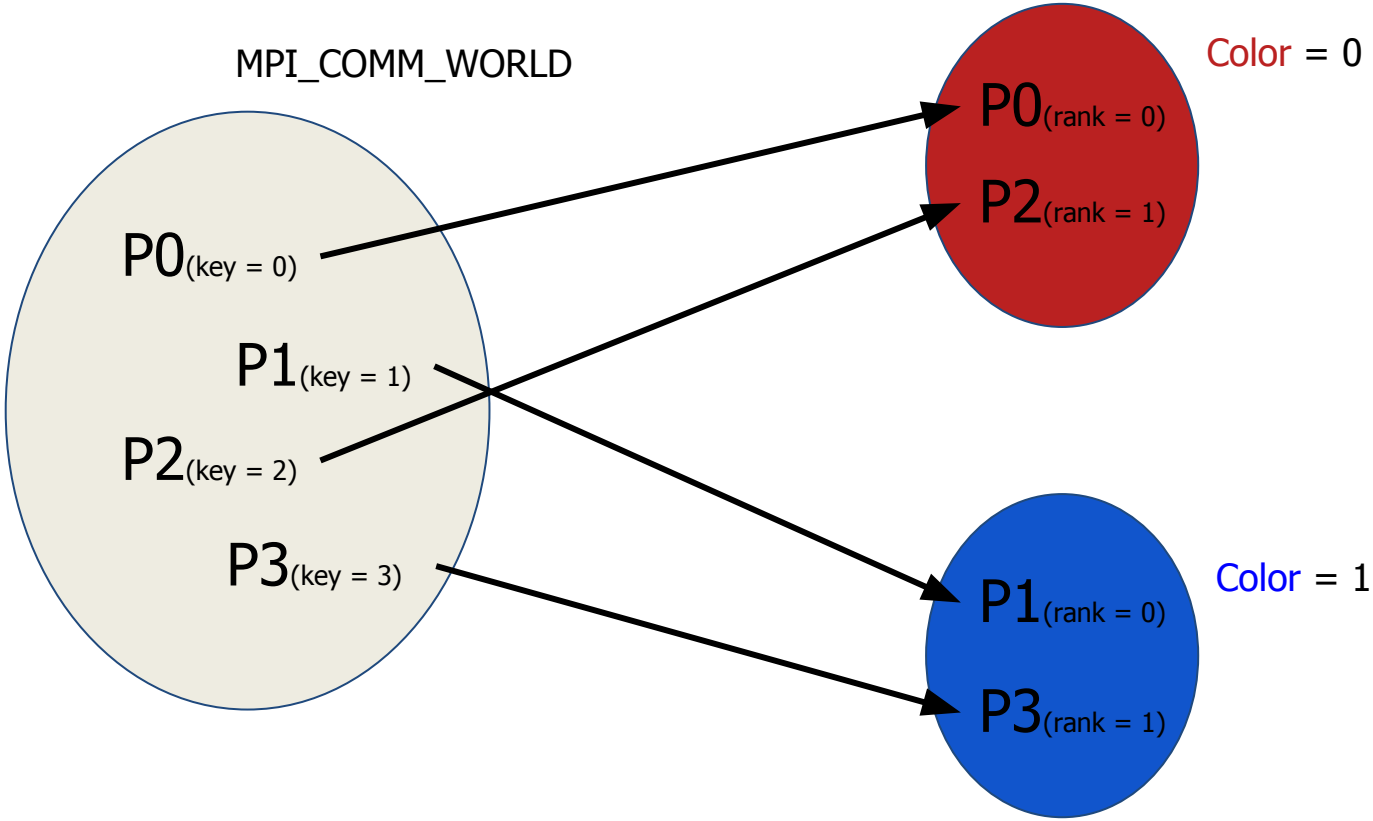
```
MPI_COMM_WORLD rank: 0/4 - ncomm rank: 0/2
```

- **Решење:** датотека `01_communicators.c`, директоријум `resenja`.

Задатак 1: Комуникатори — појашњење решења

- Параметар *color* има две могуће вредности, и то 0 за процесе са парним, 1 за процесе са непарним ранком — дакле, креираће се два нова комуникатора.
- Ако има 4 процеса, процеси 0 и 2 иду у један, процеси 1 и 3 у други комуникатор.
- Као кључ прослеђује се ранк процеса из почетног комуникатора.
- У новом комуникатору ће рангирање процеса пратити релативно уређење кључева — процес 0 имаће ранк 0 у новом комуникатору, процес 2 имаће ранк 1; процес 1 имаће ранк 0 у новом комуникатору, процес 3 имаће ранк 1.

Задатак 1: Комуникатори — илустрација решења



MPI типови података

- Зарад портабилности MPI стандард дефинише типове података.

MPI тип податка	**C тип податка**
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_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

[1] Комплетна листа MPI типова података по MPI2 стандарду.

Point to Point комуникација

- Комуникација два процеса. Један процес **шаље** поруку, други процес **прима** поруку.
- Функције за размену порука:

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

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

[1] OpenMPI 2.0 MPI_Recv docs
[2] OpenMPI 2.0 MPI_Send docs

Пример 2: Point to Point комуникација

```
//...
if (rank == 0) {
     int message = 1;
     printf("Proces %d salje poruku procesu %d.\n", rank, 1);
     MPI_Send(&message, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
    int message;
     printf("Proces %d treba da primi poruku od procesa %d.\n",  rank, 0);
     MPI_Recv(&message, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, NULL)
     printf("Proces %d primio poruku %d od procesa %d.\n",  rank,
              message, 0);
}
//...
```

Режими Point-to-Point комуникације

- P2P комуникациони режими слања поруке:
  - **Synchronous** (`MPI_Ssend`) — логички, пошиљалац шаље захтев за слање поруке, након што прималац одговори порука се шаље (*handshake* протокол); у реалности, `MPI_Ssend` ће се окончати само ако постоји `MPI_Recv` са којим се може упарити
  - **Buffered** (`MPI_Bsend`) — по иницирању слања порука се шаље у бафер одакле је прималац може преузети; ако нема места у баферу, доћи ће до грешке
  - **Standard** (`MPI_Send`) — може имати карактеристике `buffered` или `synchronous` режима; зависи од величине поруке, доступног простора у баферу и количине оптимизације
  - **Ready** (`MPI_Rsend`) — претпоставља се да процес прималац већ чека на поруку у тренутку иницирања слања; **Synchronous** режим блокира извршење и чека на позив `MPI_Recv` са стране примаоца, **Ready** не чека већ одмах долази до грешке ако `MPI_Recv` позив не постоји
- За пријем поруке постоји само један режим и порука се сматра примљеном када је преузета и може даље да се користи.

Напомена: у претходном тексту, *бафер* представља системски бафер у који се пре слања (евентуално) копирају подаци прослеђени некој од наведених `Send` метода

Point-to-Point комуникација

- Да би **процес2** примио поруку коју му шаље **процес1** мора да важи:
  - Да `comm` параметар оба процеса има исту вредност,
  - Да параметар `dest` процеса 1 буде једнак ранку процеса 2, а да параметар `source` процеса 2 буде једнак ранку процеса 1 или да буде постављен на `MPI_ANY_SOURCE`,
  - Да параметар `tag` има исту вредност за оба процеса или да је вредност параметра `tag` `MPI_ANY_TAG`

Типови Point-to-Point комуникације

- Слање и пријем поруке могу бити:
  - **блокирајући** - Ако је `MPI_Send` блокирајућа, контрола тока се неће вратити позиваоцу функције све док услов слања не буде испуњен. Након изласка из функције бафер поруке може бити безбедно преписан. Ако је `MPI_Recv` блокирајућа контрола се не враћа позиваоцу функције све док порука не буде преузета **(подразумевано)**.
  - **неблокирајући** - Из `MPI_Send`, тј. `MPI_Recv` се излази након иницијације слања, тј. примања поруке. Када се појави потреба за коришћењем изворног односно одредишног бафера, потребно је претходно проверити да ли је податак послат тј. да ли је стигао.

```
MPI_{I}[S, B, R]Send(...), MPI_{I}Recv(...)
```

Напомена: у претходном тексту, *бафер* представља први параметар `MPI_Send` и `MPI_Recv` метода

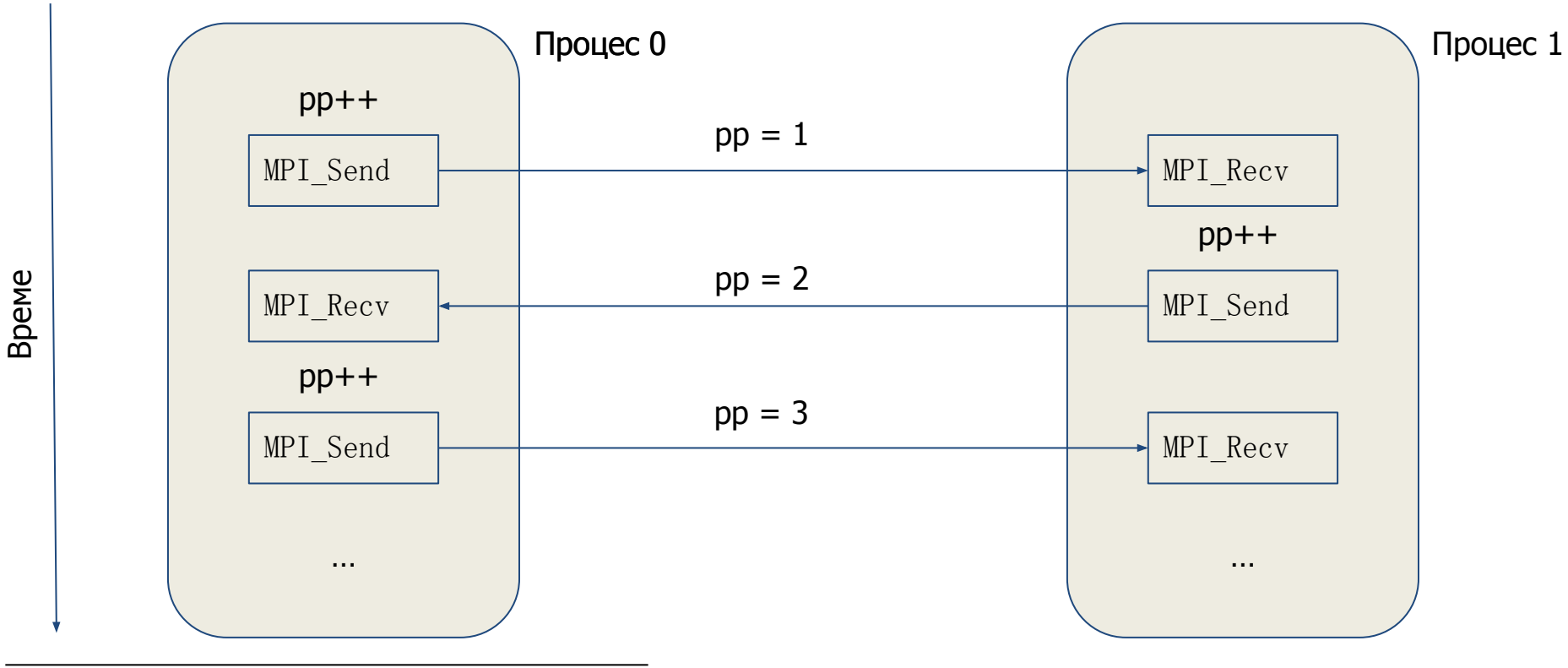
Задатак 2: Пинг понг

- Направити *OpenMPI* програм имплементиран у *C* програмском језику који симулира играње пинг понга између два процеса. Лоптицу симулирати променљивом типа `int`. Увећати ову променљиву сваки пут када неки од процеса удари лоптицу, односно, пре него што неки од процеса пошаље променљиву другом процесу и исписати одговарајућу поруку.
- **Формат очекиваног исписа:**

```
p0 sent ping_pong_count to p1 and incremented it to 1.
p1 received ping_pong_count 1 from p0.
p1 sent ping_pong_count to p0 and incremented it to 2.
```

- **Напомене:**
  - Претпоставка је да ће програм бити позван опцијом `-np 2` и од овога се не мора штитити.
  - Не значи да програм није исправан уколико испис на екрану није у очекиваном редоследу.
- **Решење:** датотека `02_ping_pong.c`, директоријум `resenja`.

Задатак 2: Пинг понг — илустрација решења



pp — Променљива којом се симулира лоптица.

Задатак 3: Пинг понг — секвенцијални испис

- Модификовати основни пинг понг задатак тако да се испис на стандардни излаз одвија у редоследу у којем процеси ударају пинг понг лоптицу. Програм покренути са три процеса — процеси 0 и 1 играју пинг понг, док трећи процес исписује поруке на стандардни излаз. Сваки пут када неки од процеса играча удари лоптицу, он процесу штампачу шаље поруку коју треба исписати на стандардни излаз. Поруке за испис процесу штампачу стижу у произвољном редоследу, али он треба да их испише у редоследу који одговара секвенцијалном извршавању програма. Све поруке су исте дужине. Пинг понг се игра до 9.
- **Пример извршавања:**

```
p0 sent ping_pong_count to p1 and incremented it to 1.
p1 sent ping_pong_count to p0 and incremented it to 2.
p0 sent ping_pong_count to p1 and incremented it to 3.
```

- **Решење:** датотека `03_ping_pong_seq.c`, директоријум `resenja`.

Задатак 3: Пинг понг — секвенцијални испис, појашњење решења

- Поље *tag* ће обезбедити да процес који исписује поруке (**процес2**) увек чека на пријем поруке која следећа треба бити исписана, без обзира на редослед њиховог слања.
- Постоји шанса да ће ово успорити рад процеса који играју пинг-понг (процеси **процес0** и **процес1)** — ако се у позадини `MPI_Send` функције користи синхрони режим, процес који шаље поруку неће наставити са радом све док **процес2** не изврши позив `MPI_Recv` функције са одговарајућим тагом; овог проблема неће бити ако се у позадини користи баферовани режим.
- Због свега овога, у овом задатку може бити паметније експлицитно користити `MPI_Bsend`.

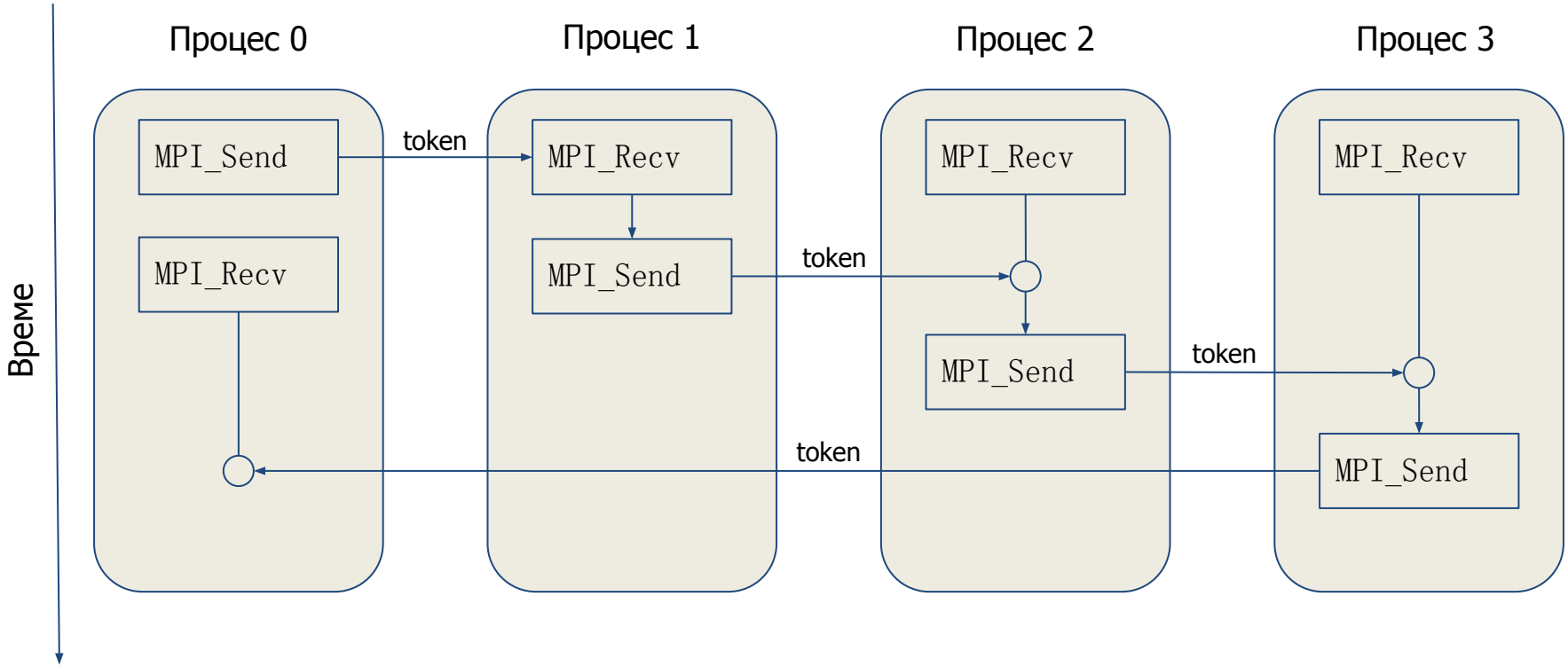
Задатак 4: Прстен

- Написати *OpenMPI C* програм који прослеђује жетон између процеса по принципу прстена. Жетон је представљен бројем -1 и поседује га процес ранга 0. Сви процеси осим последњег шаљу жетон процесу са рангом за један већим од свог. Последњи процес (процес са највећим рангом у комуникатору) жетон прослеђује назад процесу ранга 0.
- Након што процес ранга 0 прими жетон, програм се завршава. Исписати поруку на стандардни излаз сваки пут када неки од процеса прими жетон.
- **Формат очекиваног исписа:**

```
Process 1 received token -1 from process 0
Process 2 received token -1 from process 1
Process 3 received token -1 from process 2
Process 0 received token -1 from process 3
```

- **Решење:** датотека `04_ring.c`, директоријум `resenja`.

Задатак 4: Прстен — илустрација решења



Задатак 4: Прстен — појашњење решења

- На почетку сви позивају `MPI_Recv` осим процеса са ранком нула.
- Када би и **процес0** позвао функцију за пријем поруке, дошло би до *deadlock*-а, јер би сви процеси чекали на поруку коју нико још увек није послао.

П2П: Неблокирајућа комуникација

- `MPI_Isend` (и други режими), `MPI_Irecv`
- Процес иницира слање или примање поруке, али **не чека на завршетак** операције.

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

- Тестира стање захтева и поставља променљиву `flag` на `true` уколико је захтев извршен, односно на `false` уколико није.

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

- Тестира да ли је захтев извршен и завршава се када се захтев изврши. Функција је блокирајућа.

[1] MPI_Test docs
[1] MPI_Wait docs

Пример 3: 03_send_recv_nonblocking.c

```
if (rank == 0) {
      MPI_Request send_request;
      char *message = "Zdravo!";

      MPI_Issend(message, 8, MPI_CHAR, 1, 0, MPI_COMM_WORLD,  &send_request);
      printf("Proces %d inicirao slanje poruke.\n", rank);
      printf("Proces radi nesto drugo dok se poruka salje.");

      int flag = 0;

      MPI_Test(&send_request, &flag, MPI_STATUS_IGNORE);
      if (flag != 0)
          /* poslato */
      else
          /* nije */
} else /*... */
```

Пример 3: 03_send_recv_nonblocking.c

```
if (rank == 0) /*...*/
} else {
    MPI_Request receive_request;
    char message[8];

    MPI_Irecv(message, 8, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
              &receive_request);
    printf("Proces %d inicirao primanje poruke.\n", rank);

    printf("Proces radi nesto drugo dok se poruka prima...");

    MPI_Wait(&receive_request, MPI_STATUS_IGNORE);
    printf("Proces %d primio poruku: \"%s\"\n", rank, message)
}
```

Задатак 5: Пинг понг — неблокирајући секвенцијални испис

- Модификовати задатак 4 тако да слање порука процесу штампачу буде неблокирајуће. Притом обезбедити да програм ради коректно, односно да се не деси да порука која није послата буде преписана новом поруком пре него што се стара пошаље.
- **Решење**: датотека `05_ping_pong_printf_async.c`

Задатак 5: Пинг понг — неблокирајући секвенцијални испис

- Да у решењу нема *if(has_sent)* провере, позив `MPI_Wait` морао би да следи одмах након позива `MPI_Isend` да се променљива *send_str* не би променила пре него што заправо буде послата.
- `MPI_Isend` позив праћен позивом `MPI_Wait` без било каквог извршења између ове две функције имао би исти ефекат као да је позвана блокирајућа варијанта функције за слање поруке.
- Из овог разлога уведена је *has_sent* променљива која омогућава да се пре `MPI_Wait` изврши наредна итерација и стигне до позива `MPI_Recv`.

П2П: динамичка комуникација

- Некада поруке које процеси размењују нису фиксне дужине. Тада је прво потребно очитати дужину поруке, алоцирати бафер за поруку, па тек онда започети њено примање.

```
MPI_Probe(
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Status* status)
```

- Симулира примање поруке. Попуњава `status` поље.

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

- Очитава број примљених података типа `datatype` на основу `status` поља.

[1] MPI_Probe docs
[1] MPI_Get_count docs

Пример 4: 04_dynamic_communication.c

```
if (rank == 0) {
    int size = rand() % 10 + 1;
    char *message = (char *) calloc(size + 1, sizeof(char));

    for (int i = 0; i < size; i++) {
        message[i] = 'a';
    }

    MPI_Send(message, size + 1, MPI_CHAR, 1, 0,
             MPI_COMM_WORLD);
    free(message);
} else {
    /*...*/
}
```

Пример 4: dynamic_communication.c

```
if (rank == 0) {
    /*...*/
} else {
    MPI_Status status; int size;
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_CHAR, &size);

    char *message = (char *) malloc(size * sizeof(char));
    MPI_Recv(message, size, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);

    printf("Primljena poruka: %s.\n", message);
}
```

Задатак 6: Пинг понг - порука варијабилне дужине

- Модификовати задатак 3 тако да се процесу штампачу шаљу поруке променљиве дужине. Користити функције `MPI_Probe` и `MPI_Get_count`. Процеси могу да изврше максимално 999 размена лоптицом.
- **Решење**: датотека `06_ping_pong_printf_variablelen.c`