Семинар 9
Стандарт MPI (часть 2)

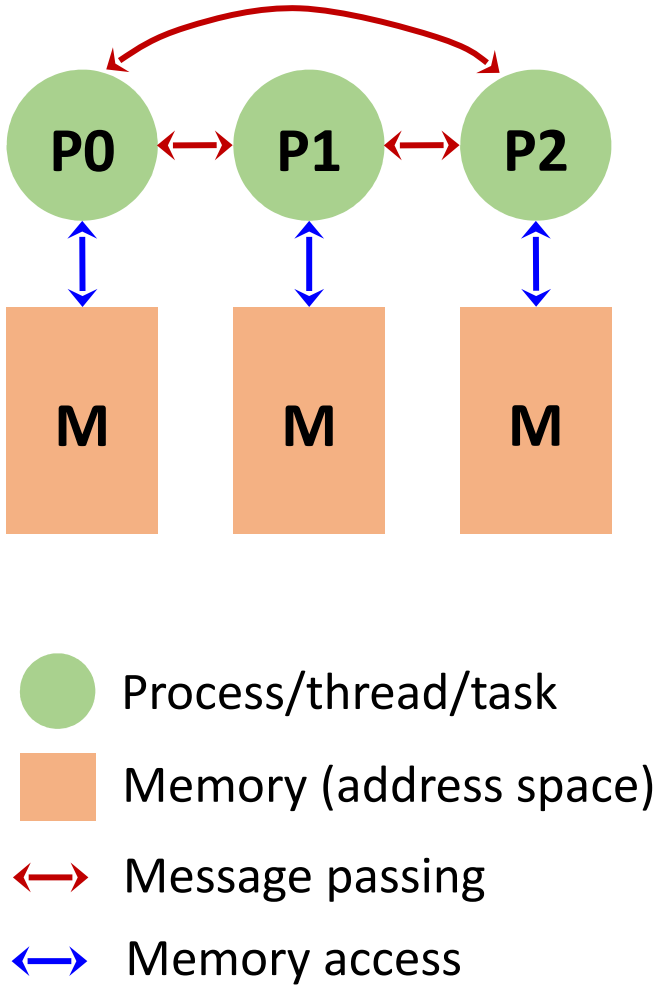
Михаил Курносов

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Цикл семинаров «Основы параллельного программирования»
Институт физики полупроводников им. А. В. Ржанова СО РАН
Новосибирск, 2015

Модель программирования

- Программа состоит из N параллельных процессов, которые порождаются при запуске программы (MPI 1) или могут быть динамически созданы во время выполнения (MPI 2)
- Каждый процесс имеет уникальный идентификатор [0, N – 1] и изолированное адресное пространство (SPMD)
- Процессы взаимодействуют путем передачи сообщений (message passing)
- Процессы могут образовывать группы для реализации коллективных операций обмена информацией

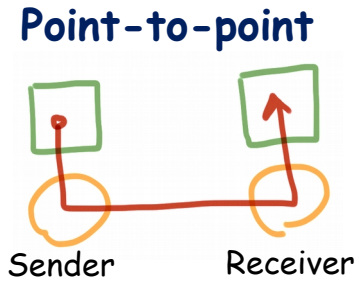


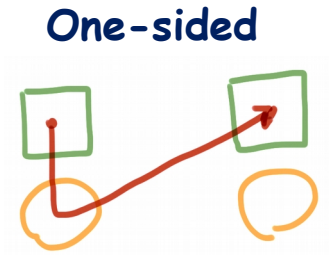
Виды обменов сообщениями в MPI

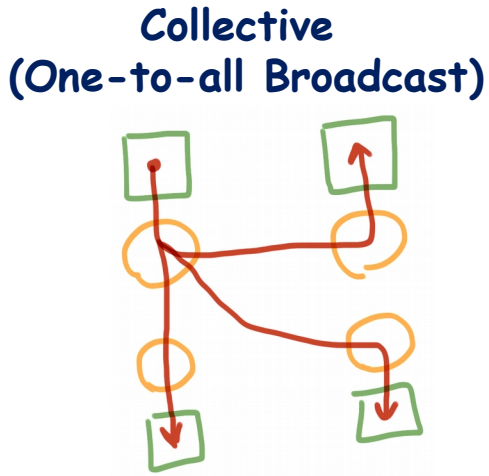
- **Двусторонние обмены (point-to-point communication)**
  - Один процесс инициирует передачу сообщения (send), другой его принимает (receive)
  - Изменение памяти принимающего процесса происходит при его явном участии
  - Обмен совмещен с синхронизацией процессов
- **Односторонние обмены (one-sided communication, remote memory access)**
  - Только один процесс явно инициирует передачу/прием сообщения из памяти удаленного процесса
  - Синхронизация процессов отсутствует

Виды обменов сообщениями в MPI

- **Двусторонние обмены (point-to-point communication)** – участвуют два процесса коммуникатора (send, recv)
- **Односторонние обмены (one-sided communication)** – участвуют два процесса коммуникатора (без синхронизации процессов, put, get)
- **Коллективные обмены (collective communication)** – участвуют все процессы коммуникатора (one-to-all broadcast, all-to-one gather, all-to-all broadcast)







Структура сообщения (point-to-point)

- **Данные**
  - Адрес буфера в памяти
  - Число элементов в буфере (count)
  - Тип данных элементов в буфере (datatype)
- **Заголовок (envelope)**
  - Идентификаторы отправителя и получателя (source, destination)
  - Тег сообщения (tag)
  - Коммуникатор (communicator)

```
int MPI_Send(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)
```

Двусторонние обмены (point-to-point)

Блокирующие (Blocking)

- MPI_Bsend
- MPI_Recv
- MPI_Rsend
- MPI_Send
- MPI_Sendrecv
- MPI_Sendrecv_replace
- MPI_Ssend
- ...

Неблокирующие (Non-blocking)

- MPI_Ibsend
- MPI_Irecv
- MPI_Irsend
- MPI_Isend
- MPI_Issend
- ...

Проверки состояния запросов (Completion/Testing)

- MPI_Iprobe
- MPI_Probe
- MPI_Test{, all, any, some}
- MPI_Wait{, all, any, some}
- ...

Постоянные (Persistent)

- MPI_Bsend_init
- MPI_Recv_init
- MPI_Send_init
- ...
- MPI_Start
- MPI_Startall

Блокирующие функции Send/Recv

```
int MPI_Send(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)
```

- buf — адрес буфера
- count — число элементов в буфере
- datatype — тип данных элементов в буфере
- dest — номер процесса-получателя
- source — номер процесса-отправителя или MPI_ANY_SOURCE
- tag — тег сообщения или MPI_ANY_TAG
- comm — идентификатор коммуникатора или MPI_COMM_WORLD
- status — параметры принятого сообщения (содержит поля source, tag)

Соответствие типов данных MPI типам языка C

MPI datatype	C datatype
MPI_CHAR	`char` (treated as printable character)
MPI_SHORT	`signed short int`
MPI_INT	`signed int`
MPI_LONG	`signed long int`
MPI_LONG_LONG_INT	`signed long long int`
MPI_LONG_LONG (as a synonym)	`signed long long int`
MPI_SIGNED_CHAR	`signed char` (treated as integral value)
MPI_UNSIGNED_CHAR	`unsigned char` (treated as integral value)
MPI_UNSIGNED_SHORT	`unsigned short int`
MPI_UNSIGNED	`unsigned int`
MPI_UNSIGNED_LONG	`unsigned long int`
MPI_UNSIGNED_LONG_LONG	`unsigned long long int`
MPI_FLOAT	`float`
MPI_DOUBLE	`double`
MPI_LONG_DOUBLE	`long double`
MPI_WCHAR	`wchar_t` (defined in `<stddef.h>`) (treated as printable character)
MPI_C_BOOL	`_Bool`
MPI_INT8_T	`int8_t`
MPI_INT16_T	`int16_t`
MPI_INT32_T	`int32_t`
MPI_INT64_T	`int64_t`
MPI_UINT8_T	`uint8_t`
MPI_UINT16_T	`uint16_t`
MPI_UINT32_T	`uint32_t`
MPI_UINT64_T	`uint64_t`
MPI_C_COMPLEX	`float _Complex`
MPI_C_FLOAT_COMPLEX (as a synonym)	`float _Complex`
MPI_C_DOUBLE_COMPLEX	`double _Complex`
MPI_C_LONG_DOUBLE_COMPLEX	`long double _Complex`
MPI_BYTE	
MPI_PACKED	

Определение размера принятого сообщения

```c
#define NELEMS(x) (sizeof(x) / sizeof((x)[0]))

int main(int argc, char **argv)
{
    int rank, commsize, len, count, tag = 1;
    char host[MPI_MAX_PROCESSOR_NAME], msg[128 + MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Get_processor_name(host, &len);

    if (rank > 0) {
        count = snprintf(msg, NELEMS(msg), "Hello, I am %d of %d on %s", rank, commsize, host) + 1;
        MPI_Send(msg, count, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
    } else {
        MPI_Status status;
        for (int i = 1; i < commsize; i++) {
            MPI_Recv(msg, NELEMS(msg), MPI_CHAR, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_CHAR, &count);
            printf("Message from %d (tag %d, count %d): '%s'\n", status.MPI_SOURCE, status.MPI_TAG, count, msg);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Каждый процесс
1, 2, ..., p — 1
отправляет корневому
процессу сообщение
переменной длины

Определение размера принятого сообщения

```
Message from 1 (tag 1, count 27): 'Hello, I am 1 of 16 on cn3'
Message from 2 (tag 1, count 27): 'Hello, I am 2 of 16 on cn3'
Message from 3 (tag 1, count 27): 'Hello, I am 3 of 16 on cn3'
Message from 4 (tag 1, count 27): 'Hello, I am 4 of 16 on cn3'
Message from 5 (tag 1, count 27): 'Hello, I am 5 of 16 on cn3'
Message from 6 (tag 1, count 27): 'Hello, I am 6 of 16 on cn3'
Message from 12 (tag 1, count 28): 'Hello, I am 12 of 16 on cn4'
Message from 7 (tag 1, count 27): 'Hello, I am 7 of 16 on cn3'
Message from 14 (tag 1, count 28): 'Hello, I am 14 of 16 on cn4'
Message from 13 (tag 1, count 28): 'Hello, I am 13 of 16 on cn4'
Message from 15 (tag 1, count 28): 'Hello, I am 15 of 16 on cn4'
Message from 10 (tag 1, count 28): 'Hello, I am 10 of 16 on cn4'
Message from 11 (tag 1, count 28): 'Hello, I am 11 of 16 on cn4'
Message from 8 (tag 1, count 27): 'Hello, I am 8 of 16 on cn4'
Message from 9 (tag 1, count 27): 'Hello, I am 9 of 16 on cn4'
```

Семантика двусторонних обменов (point-to-point)

- Гарантируется сохранение порядка сообщений от каждого процесса-отправителя
- Не гарантируется “справедливость” доставки сообщений от нескольких отправителей

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

Сообщение, отправленное первым send,
должно быть получено первым recv

Пример Send/Recv (попытка получить меньше, чем нам отправили)

```
int main(int argc, char **argv)
{
    int rank, commsize;
    float buf[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);

    if (rank == 0) {
        for (int i = 0; i < NELEMS(buf); i++)
            buf[i] = (float)i;
        MPI_Send(buf, NELEMS(buf), MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status status;
        MPI_Recv(buf, 10, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Master received: ");
        int count;
        MPI_Get_count(&status, MPI_FLOAT, &count);
        for (int i = 0; i < count; i++)
            printf("%f ", buf[i]);
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Fatal error in MPI_Recv:
Message truncated, error stack:
MPIDI_CH3U_Receive_data_found(284): Message from rank 0 and tag 0 truncated; 400 bytes received but buffer size is 40

Пример Send/Recv (попытка получить больше, чем нам отправили)

```
int main(int argc, char **argv)
{
    int rank, commsize;
    float buf[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);

    if (rank == 0) {
        for (int i = 0; i < NELEMS(buf); i++)
            buf[i] = (float)i;
        MPI_Send(buf, NELEMS(buf), MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status status;
        MPI_Recv(buf, 1000, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Master received: ");
        int count;
        MPI_Get_count(&status, MPI_FLOAT, &count);
        for (int i = 0; i < count; i++)
            printf("%f ", buf[i]);
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Master received: 0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 ...

Информация о принятом сообщении

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

- Записывает в count число принятых элементов типа datatype (информация о них хранится в переменной status)

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

- Блокирует выполнение процесса, пока не поступит сообщение (source, tag, comm)
- Информация о сообщении возвращается через параметр status

- Далее, пользователь может создать буфер нужного размера и извлечь сообщение функцией MPI_Recv

Информация о принятом сообщении

```
int main(int argc, char **argv)
{
    int rank, commsize, count;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);

    if (rank == 0) {
        float buf[10];
        for (int i = 0; i < 10; i++)
            buf[i] = i * 10.0;
        MPI_Send(buf, 10, MPI_FLOAT, 2, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        int buf[6];
        for (int i = 0; i < 6; i++)
            buf[i] = i * 2 + 1;
        MPI_Send(buf, 6, MPI_INT, 2, 1, MPI_COMM_WORLD);
    }
```

Процесс 0
отправляет
10 элементов типа float

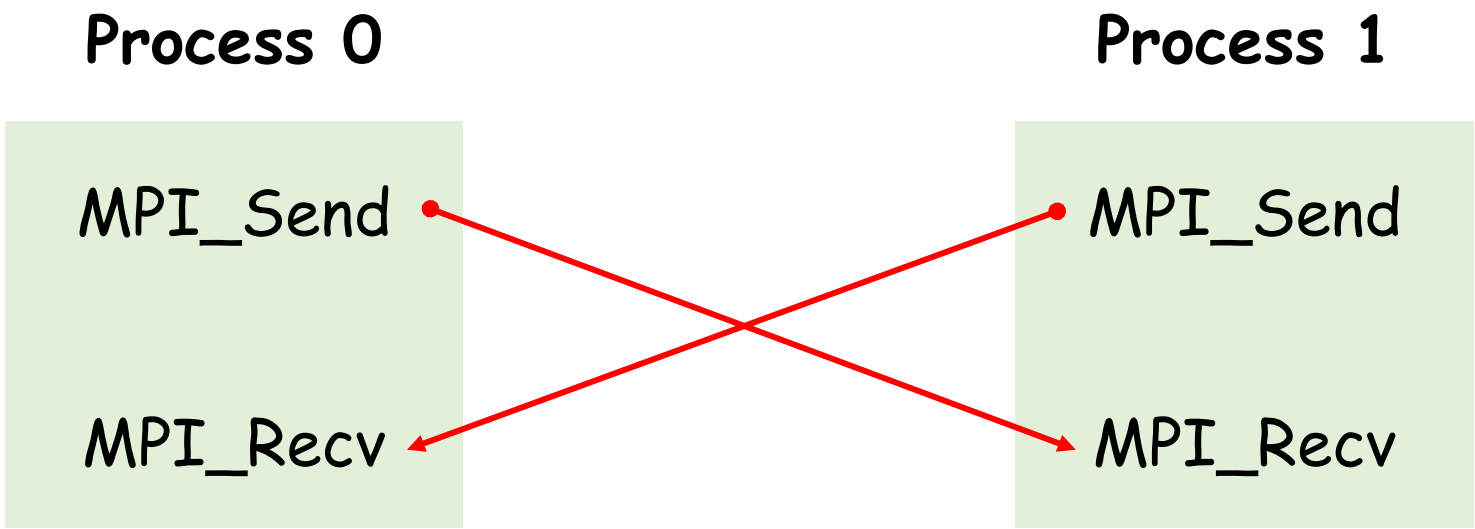
Процесс 1
отправляет
6 элементов типа int

Информация о принятом сообщении (2)

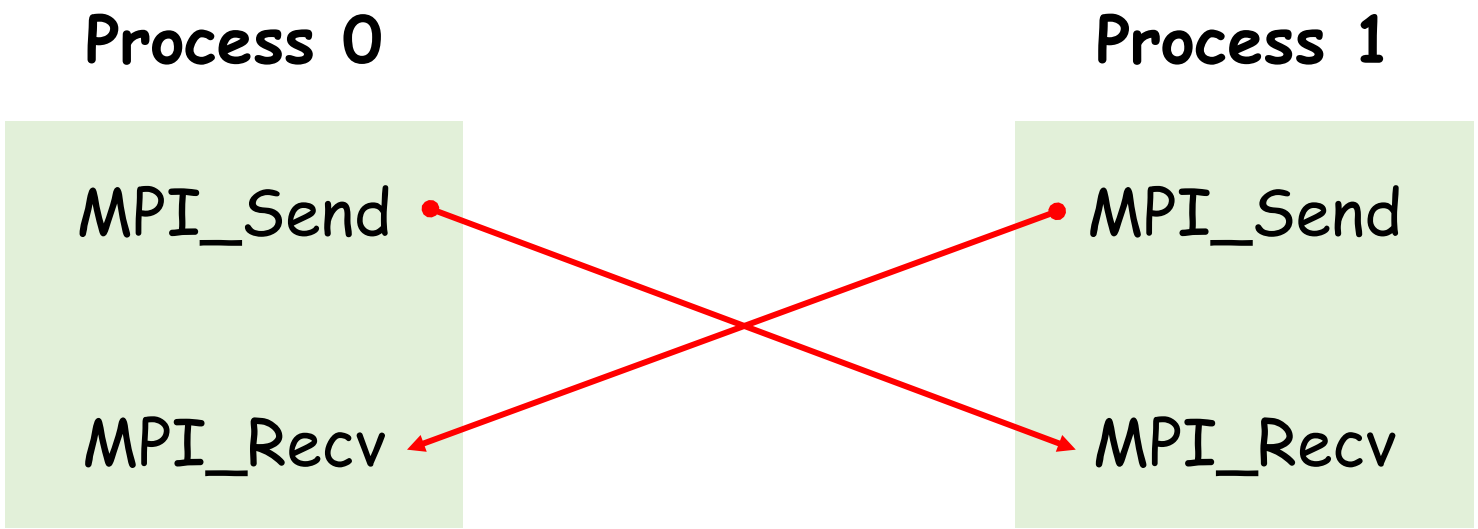
```
    else if (rank == 2) {
        MPI_Status status;
        for (int m = 0; m < 2; m++) {
            MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            if (status.MPI_TAG == 0) {
                MPI_Get_count(&status, MPI_FLOAT, &count);
                float *buf = malloc(sizeof(*buf) * count);
                MPI_Recv(buf, count, MPI_FLOAT, status.MPI_SOURCE, status.MPI_TAG, MPI_COMM_WORLD, &status);
                printf("Master received: ");
                for (int i = 0; i < count; i++)
                    printf("%.2f\n", buf[i]);
                free(buf);
            } else if (status.MPI_TAG == 1) {
                MPI_Get_count(&status, MPI_INT, &count);
                int *buf = malloc(sizeof(*buf) * count);
                MPI_Recv(buf, count, MPI_INT, status.MPI_SOURCE, status.MPI_TAG, MPI_COMM_WORLD, &status);
                printf("Master received: ");
                for (int i = 0; i < count; i++)
                    printf("%d\n", buf[i]);
                free(buf);
            }
        }
    }
    MPI_Finalize(); return 0;
}
```

Процесс 2 определяет тип принимаемых сообщений по тегу

Взаимная блокировка процессов (deadlock)



Взаимная блокировка процессов (deadlock)



- Поменять порядок операций Send/Recv
- Использовать неблокирующие операции Isend/Irecv/Wait
- Использовать функцию MPI_Sendrecv

Совмещение передачи и приема

```
int 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)
```

- Предотвращает возникновение взаимной блокировки при вызове Send/Recv
- Не гарантирует защиту от любых взаимных блокировок!

Передача сообщения по кольцу

```
enum {
    BUFSIZE = 100
};

char rbuf[BUFSIZE], sbuf[BUFSIZE];

int main(int argc, char **argv)
{
    int rank, commsize;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);

    int prev = (rank - 1 + commsize) % commsize;
    int next = (rank + 1) % commsize;

    MPI_Send(&sbuf, BUFSIZE, MPI_CHAR, next, 0, MPI_COMM_WORLD);
    MPI_Recv(&rbuf, BUFSIZE, MPI_CHAR, prev, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received from %d\n", rank, prev);

    MPI_Finalize();
    return 0;
}
```

```
Process 1 received from 0
Process 2 received from 1
Process 3 received from 2
Process 0 received from 3
```

Передача сообщения по кольцу (2)

```
enum {
    BUFSIZE = 1000000
};

char rbuf[BUFSIZE], sbuf[BUFSIZE];

int main(int argc, char **argv)
{
    int rank, commsize;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);

    int prev = (rank - 1 + commsize) % commsize;
    int next = (rank + 1) % commsize;

    MPI_Send(&sbuf, BUFSIZE, MPI_CHAR, next, 0, MPI_COMM_WORLD);
    MPI_Recv(&rbuf, BUFSIZE, MPI_CHAR, prev, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received from %d\n", rank, prev);

    MPI_Finalize();
    return 0;
}
```

Программа зависла (deadlock)

- Процесс 0 ждет когда процесс 1 примет сообщение
- Процесс 1 ждет 2
- Процесс 2 ждет 3
- Процесс три ждет 1

Цикл в графе зависимости

Для небольших сообщений MPI_Send не блокирует выполнение — сообщение сохраняется во внутреннем буфере

Передача сообщения по кольцу (3)

```
enum {
    BUFSIZE = 1000000
};

char rbuf[BUFSIZE], sbuf[BUFSIZE];

int main(int argc, char **argv)
{
    int rank, commsize;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);

    int prev = (rank - 1 + commsize) % commsize;
    int next = (rank + 1) % commsize;

    MPI_Sendrecv(&sbuf, BUFSIZE, MPI_CHAR, next, 0, &rbuf, BUFSIZE, MPI_CHAR, prev, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received from %d\n", rank, prev);

    MPI_Finalize();
    return 0;
}
```

Задание

Программа Ring

- Написать программу передачи сообщения по кольцу:
  процесс 0 передает сообщение процессу N — 1 и принимает сообщение от процесса 1, процесс 1 передает сообщение процессу 0 и принимает от 2 и т.д.

Программа PingPong

- Написать программу, в которой процесс 0 передает сообщение процессу 1, процесс 1 принимает сообщение и возвращает его обратно процессу 0
- В процессе 0 измерить суммарное время выполнения Send + Recv:

  1) когда оба процесс запущена на одном узле (обмены через общую память)

  2) когда процессы на разных узлах (обмены через Infiniband QDR)