Параллельные вычисления (часть 3)
Стандарт MPI

Михаил Георгиевич Курносов

Email: mkurnosov@gmail.com

WWW: http://www.mkurnosov.net

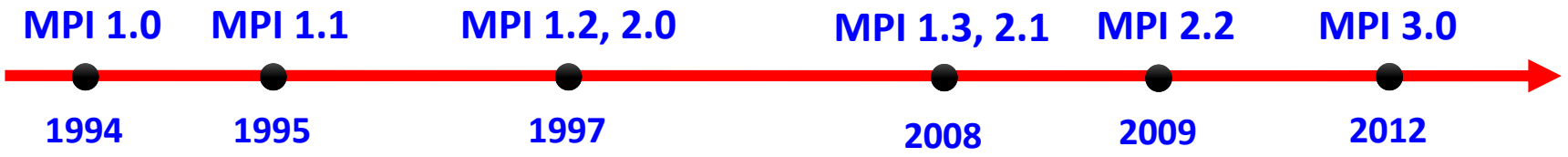
Курс «Параллельные и распределенные вычисления»

Школа анализа данных Яндекс (Новосибирск)

Весенний семестр, 2015

Стандарт MPI

- **Message Passing Interface (MPI)** – это стандарт на программный интерфейс коммуникационных библиотек для создания параллельных программ в модели передачи сообщений (message passing)
- Стандарт определяет интерфейс для языков программирования C и Fortran
- Стандарт де-факто для систем с распределенной памятью



http://www.mpi-forum.org

http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

Стандарт MPI

- **Переносимость** программ на уровне исходного кода между разными вычислительными системами (Cray, IBM, NEC, Fujitsu, SiCortex)
- **Высокая производительность**
  (MPI – это “ассемблер” в области параллельных вычислений)
- **Масштабируемость** (миллионы процессорных ядер)

Стандарт MPI

MPI: A Message-Passing Interface Standard
Version 3.0

Message Passing Interface Forum

September 21, 2012

3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

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

MPI_Send(buf, count, datatype, dest, tag, comm, ierror) BIND(C)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
```

Реализации MPI

- **MPICH** (Open source, Argone NL, http://www.mcs.anl.gov/research/projects/mpich2)
- Производные от MPICH2:
  MVAPICH2 (MPICH2 for InfiniBand), IBM MPI, Cray MPI, Intel MPI, HP MPI, Microsoft MPI

- **Open MPI** (Open source, BSD License, http://www.open-mpi.org)
- Производные от Open MPI: Oracle MPI

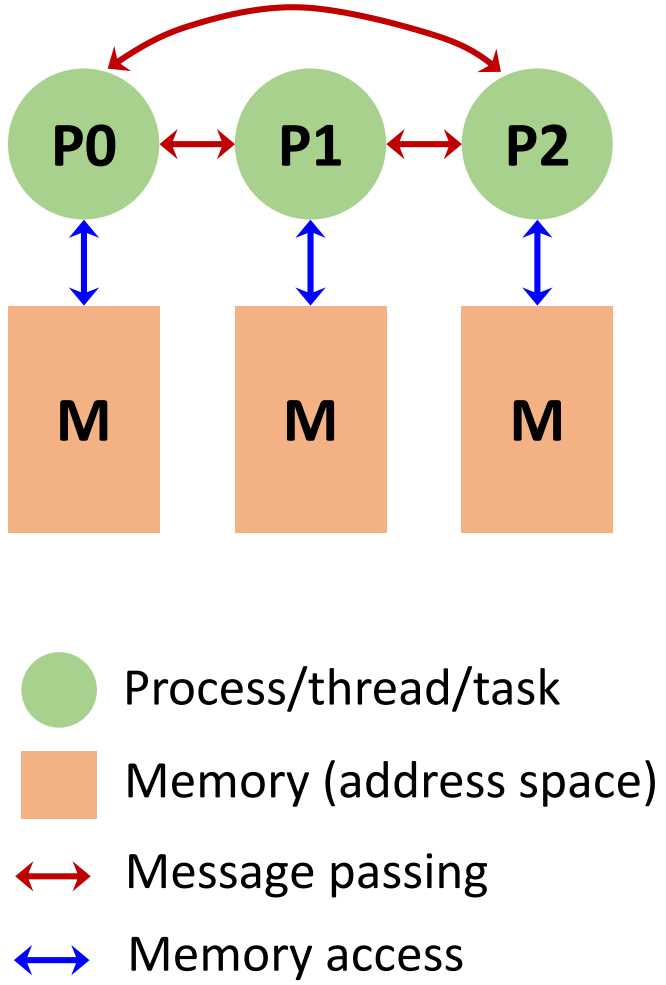
- Высокоуровневые интерфейсы
  - C++: Boost.MPI
  - Java: Open MPI Java Interface, MPI Java, MPJ Express, ParJava
  - C#: MPI.NET, MS-MPI
  - Python: mpi4py, pyMPI

Отличия в реализациях MPI

- **Спектр поддерживаемых архитектур процессоров:**
  Intel, IBM, ARM, Fujitsu, NVIDIA, AMD
- **Типы поддерживаемых коммуникационных технологий/сетей:** InfiniBand, 10 Gigabit Ethernet, Cray Gemeni, IBM PERCS/5D torus, Fujitsu Tofu, Myrinet, SCI, …
- **Протоколы дифференцированных обменов двусторонних обменов (Point-to-point):**
  хранение списка процессов, подтверждение передачи (ACK), буферизация сообщений, …
- **Коллективные операции обменов информацией:** коммуникационная сложность алгоритмов, учет структуры вычислительной системы (torus, fat tree, …), неблокирующие коллективные обмены (MPI 3.0, методы хранение collective schedule)
- **Алгоритмы вложения графов программ в структуры вычислительных систем**
  (MPI topology mapping)
- **Возможность выполнения MPI-функций в многопоточной среде и поддержка ускорителей**
  (GPU NVIDIA/AMD, Intel Xeon Phi)

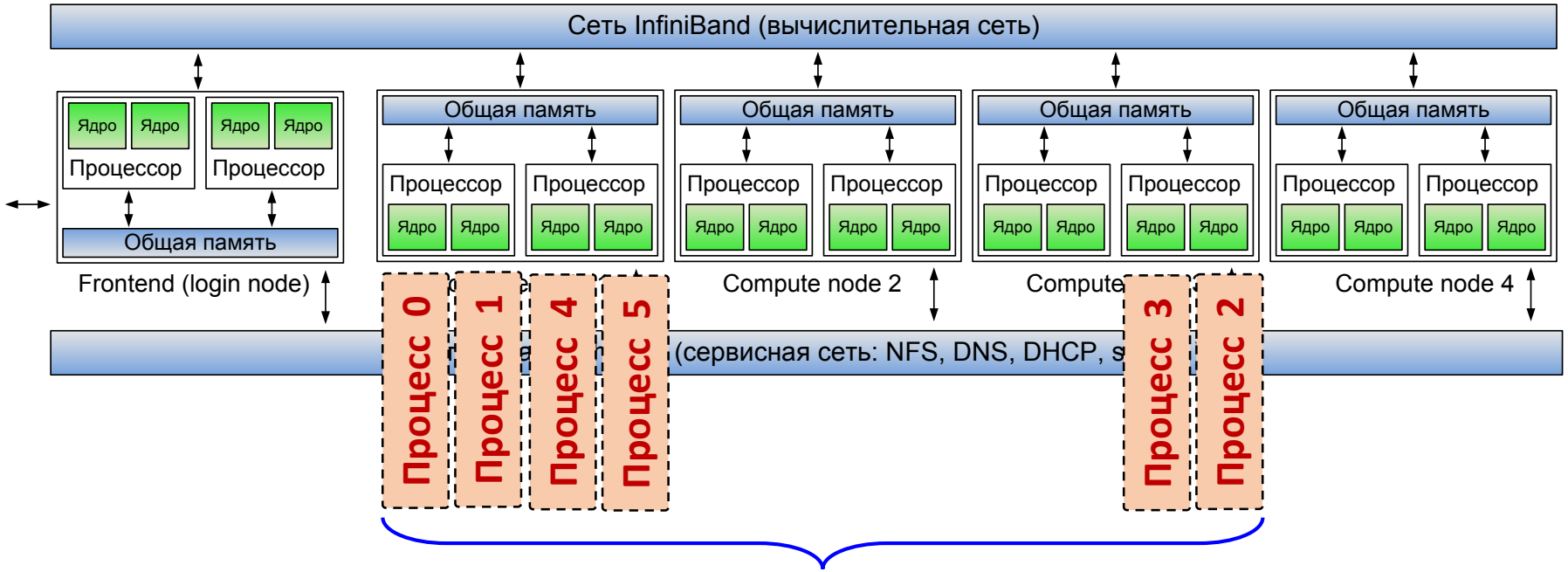
Модель программирования

- 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



Понятие коммуникатора (Communicator)

- **Коммуникатор (communicator)** – группа процессов, образующая логическую область для выполнения коллективных операций между процессами
- В рамках коммуникатора процессы имеют номера: 0, 1, …, $N - 1$
- Все MPI-сообщения должны быть связаны с определенным коммуникатором



Коммуникатор **MPI_COMM_WORLD** включает все процессы

Функции MPI

- Заголовочный файл mpi.h
- Функции, типы данных и именованные константы имеют префикс MPI_
- Функции возвращают MPI_SUCCESS или код ошибки
- Результаты возвращаются через аргументы функций

Hello, MPI World!

```
#include <mpi.h>

int main(int argc, char *argv[])
{
    int commsize, rank, len;
    char procname[MPI_MAX_PROCESSOR_NAME];

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

    printf("Hello, MPI World! Process %d of %d on node %s.\n",
           rank, commsize, procname);

    MPI_Finalize();
    return 0;
}
```

Компиляция MPI-программ

```
# Программа на C
$ mpicc -Wall -o hello ./hello.c

# Программа на C++
$ mpicxx -Wall -o hello ./hello.cpp

# Программа на Fortran
$ mpif90 -o hello ./hello.f90
```

Запуск MPI-программ на кластере (TORQUE)

```
# Формируем паспорт задачи (job-файл)
$ cat task.job
#PBS -N MyTask
#PBS -l nodes=1:ppn=8
#PBS -j oe

cd $PBS_O_WORKDIR

mpiexec ./hello

# Ставим задачу в очередь
$ qsub ./task.job
882
```

Запуск MPI-программ на кластере (TORQUE)

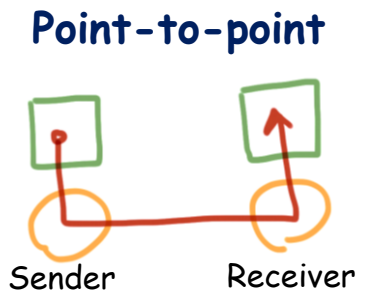
```
# Проверяем состояние задачи в очереди
$ qstat
Job ID          Name             User             Time Use S Queue
--------------- ---------------- ---------------- -------- - -----
876             stream-ep        foobar                 0 Q release
877             xdtask           pdcuser99              0 R debug
882             MyTask           mkurnosov              0 C debug

# Проверяем результат
$ cat ./MyTask.o882
Hello, MPI World! Process 0 of 8 on node cn15.
Hello, MPI World! Process 4 of 8 on node cn15.
Hello, MPI World! Process 1 of 8 on node cn15.
Hello, MPI World! Process 2 of 8 on node cn15.
Hello, MPI World! Process 3 of 8 on node cn15.
Hello, MPI World! Process 5 of 8 on node cn15.
Hello, MPI World! Process 6 of 8 on node cn15.
Hello, MPI World! Process 7 of 8 on node cn15.
```

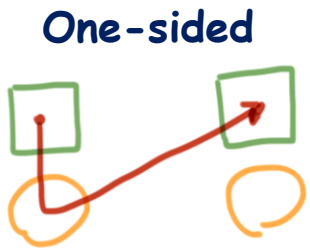
Модель передачи сообщений MPI

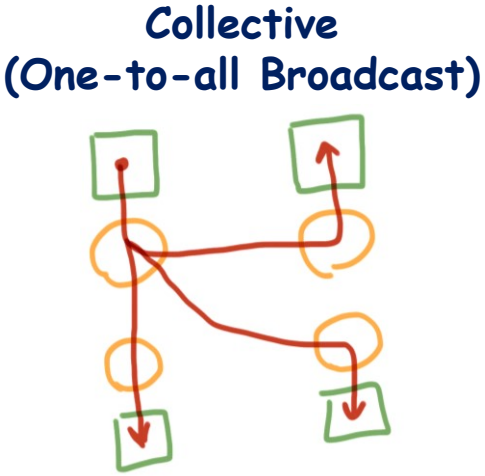
- **Двусторонние обмены (Point-to-point communication)**
  - Один процесс инициирует передачу сообщения (Send), другой его принимает (Receive)
  - Изменение памяти принимающего процесса происходит при его явном участии
  - Обмен совмещен с синхронизацией процессов
- **Односторонние обмены (One-sided 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, Remote memory access) –** 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 (непрерывный участок памяти)
  - Число элементов в буфере
  - Тип данных элементов в буфере
- **Заголовок (envelope)**
  - Идентификаторы отправителя и получателя
  - Тег сообщения (Tag)
  - Коммуникатор (Communicator)

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 сообщении
- 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	

Hello, MPI World (2)!

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

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

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

Hello, MPI World (2)!

```
Hello, World. I am master (0 of 16) on cn15
Message from 1: 'Hello, master. I am 1 of 16 on cn15'
Message from 2: 'Hello, master. I am 2 of 16 on cn15'
Message from 3: 'Hello, master. I am 3 of 16 on cn15'
Message from 4: 'Hello, master. I am 4 of 16 on cn15'
Message from 5: 'Hello, master. I am 5 of 16 on cn15'
Message from 6: 'Hello, master. I am 6 of 16 on cn15'
Message from 7: 'Hello, master. I am 7 of 16 on cn15'
Message from 14: 'Hello, master. I am 14 of 16 on cn16'
Message from 15: 'Hello, master. I am 15 of 16 on cn16'
Message from 8: 'Hello, master. I am 8 of 16 on cn16'
Message from 9: 'Hello, master. I am 9 of 16 on cn16'
Message from 12: 'Hello, master. I am 12 of 16 on cn16'
Message from 11: 'Hello, master. I am 11 of 16 on cn16'
Message from 10: 'Hello, master. I am 10 of 16 on cn16'
Message from 13: 'Hello, master. I am 13 of 16 on cn16'
```

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 (хотим получить меньше, чем нам отправили)

```
int main(int argc, char **argv) {
    float buf[100];
    // ...

    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) {
        // Пытаемся получить меньше, чем нам отправили
        MPI_Status status;
        MPI_Recv(buf, 10, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
    }

    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 (хотим получить больше, чем нам отправили)

```
int main(int argc, char **argv) {
    float buf[100];
    // ...
    if (rank == 0) {
        for (int i = 0; i < NELEMS(buf); i++)
            buf[i] = (float)i;
        MPI_Send(buf, 10, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status status; // Пытаемся получить больше, чем нам отправили
        MPI_Recv(buf, 100, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Master received: ");
        int count;
        MPI_Get_count(&status, MPI_FLOAT, &count);  // count = 10
            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 (MPI_Recv) элементов типа datatype

```
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

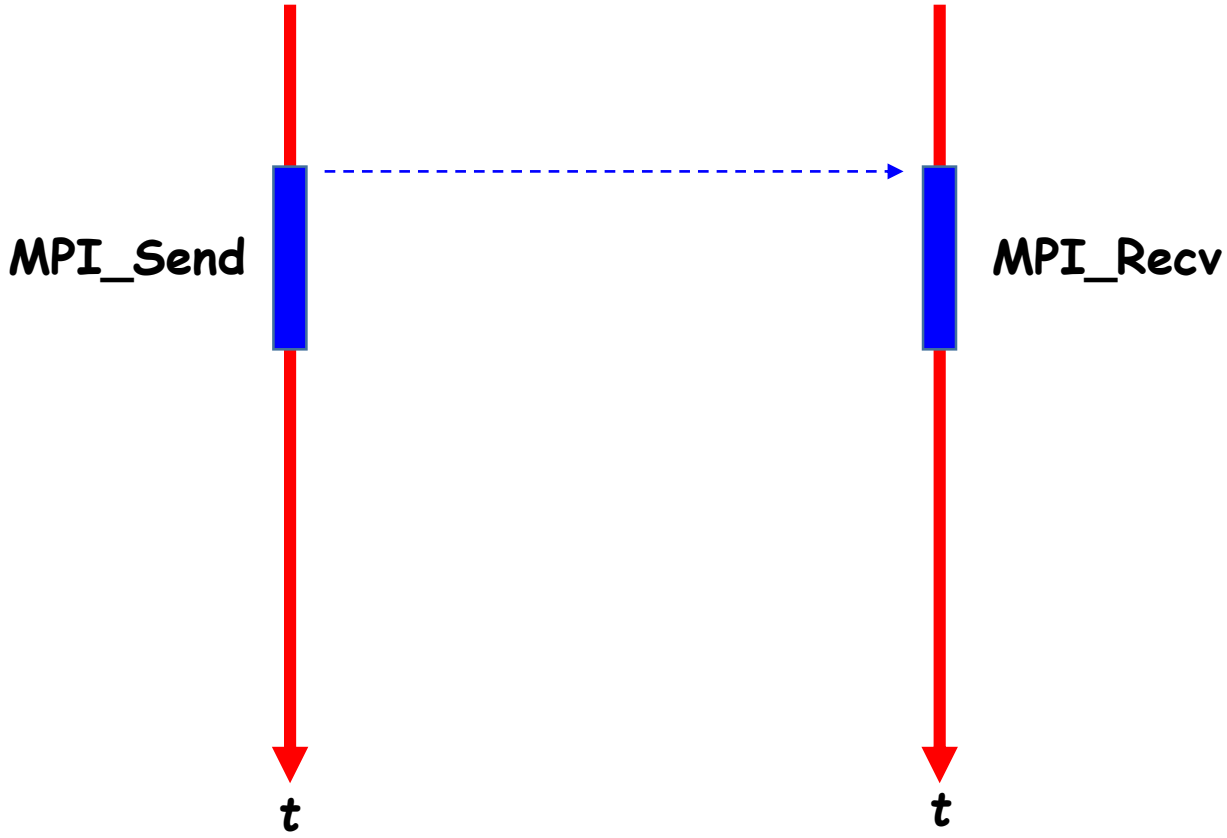
Пример MPI_Probe

```
int main(int argc, char **argv) {

    if (rank == 0) {
        float buf[100];
        MPI_Send(buf, 10, MPI_FLOAT, 2, 0, comm);    // Отправили массив float[10]
    } else if (rank == 1) {
        int buf[32];
        MPI_Send(buf, 6, MPI_INT, 2, 1, comm);       // Отправили массив int[6]
    } else if (rank == 2) {
        MPI_Status status;
        for (int m = 0; m < 2; m++) {
            MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);  // Ждем любого сообщения
            if (status.MPI_TAG == 0) {                              // Определяем тип сообщения
                MPI_Get_count(&status, MPI_FLOAT, &count); // Сколько пришло MPI_FLOAT?
                float *buf = malloc(sizeof(*buf) * count);
                MPI_Recv(buf, count, MPI_FLOAT, status.MPI_SOURCE, status.MPI_TAG, comm, &status);
            } else if (status.MPI_TAG == 1) {
                MPI_Get_count(&status, MPI_INT, &count);   // Сколько пришло MPI_INT?
                int *buf = malloc(sizeof(*buf) * count);
                MPI_Recv(buf, count, MPI_INT, status.MPI_SOURCE, status.MPI_TAG, comm, &status);
            }
        }
    }
```

Состояние процесса после завершения MPI_Send

- Буфер можно повторно использовать, не опасаясь испортить передаваемое сообщение?
- Сообщение покинуло узел процесса-отправителя?
- Сообщение принято процессом-получателем?

Реализация Send/Recv – идеальная ситуация

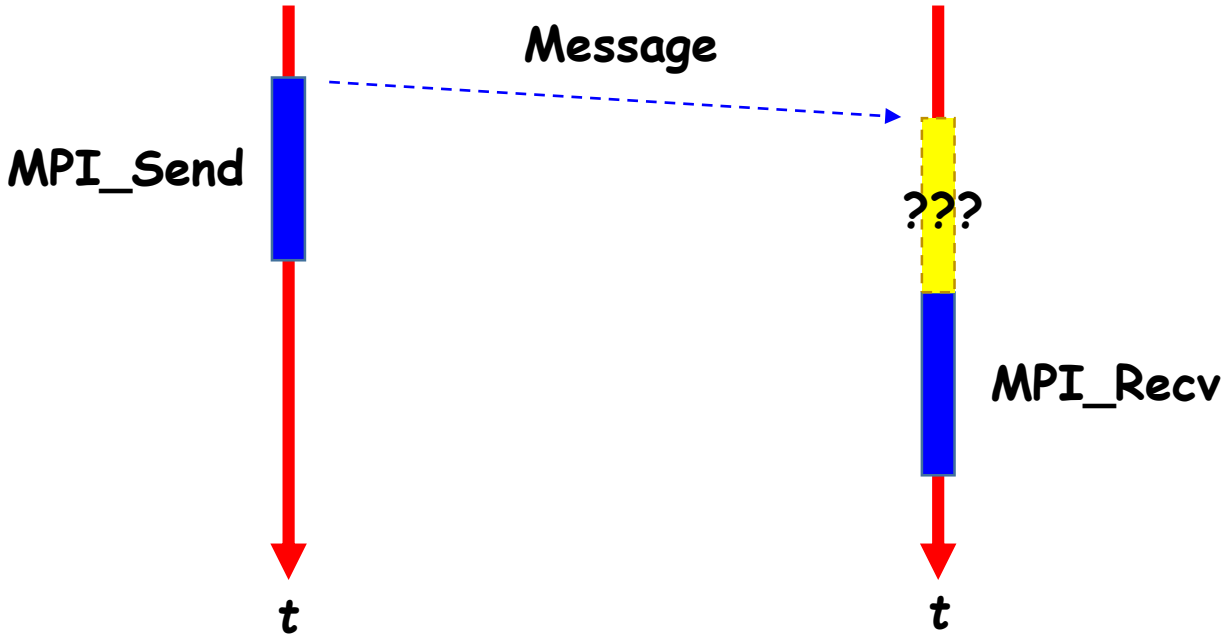


В идеальной ситуации вызов Send должен быть синхронизирован с вызовом Recv (функции должны вызываться в один момент времени)

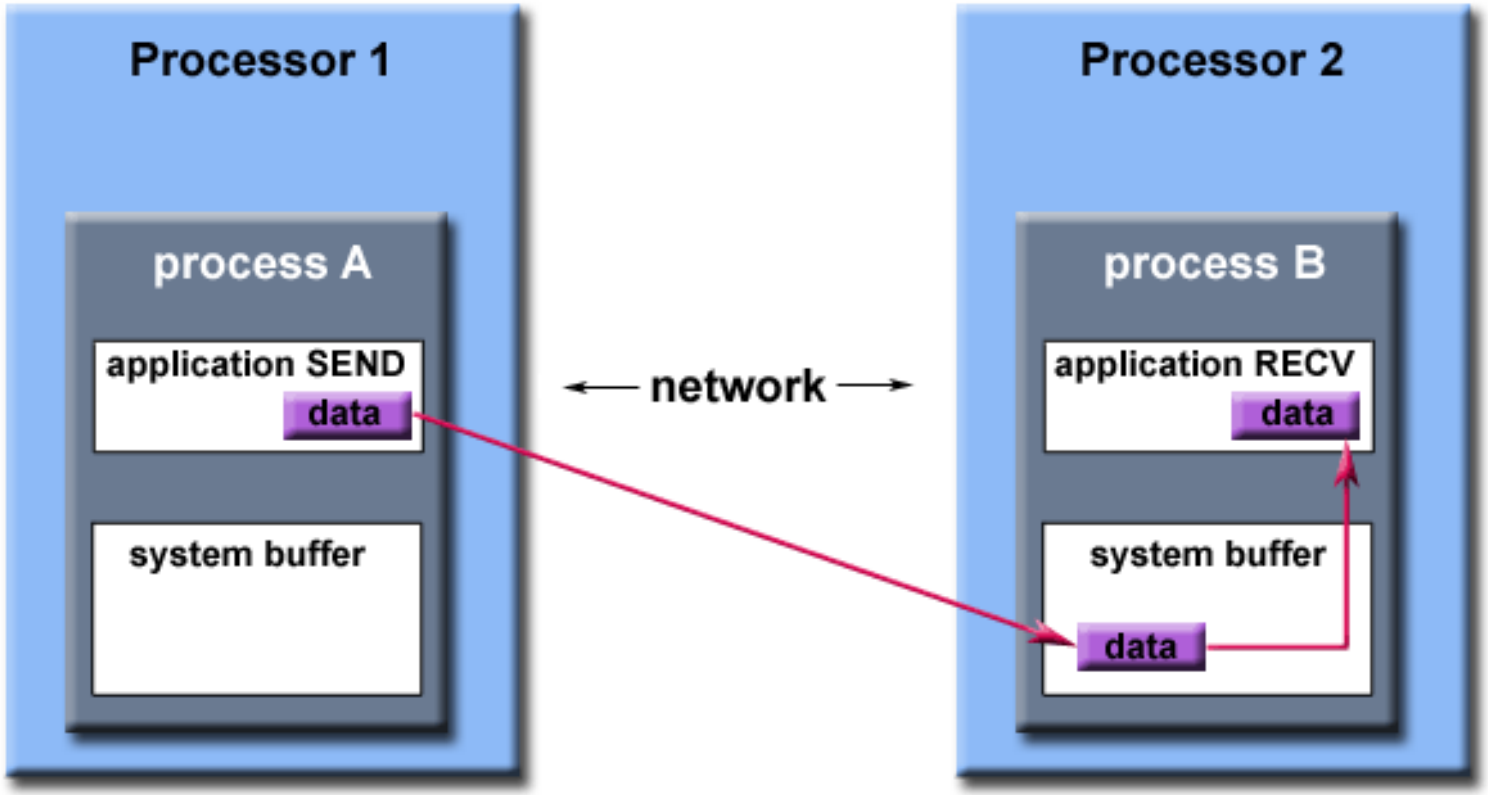
Реализация Send/Recv – реальная ситуация

- Что будет, если Send вызван за 5 секунд до вызова Recv?
  Где будет храниться исходящее сообщение?

- Что будет если несколько процессов одновременно отправляют сообщения процессу-получателю? Как он должен их хранить?



Реализация Send/Recv



Path of a message buffered at the receiving process

https://computing.llnl.gov/tutorials/mpi/

Коммуникационные режимы блокирующих обменов

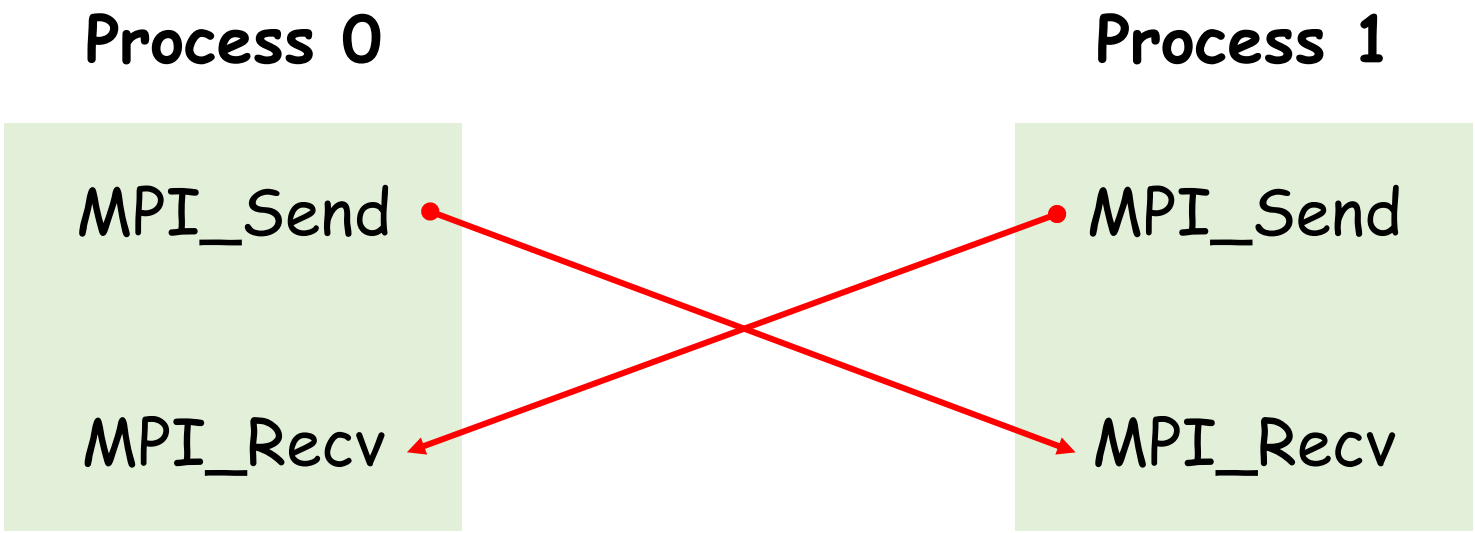
- **Стандартный режим** (Standard communication mode, local/non-local) – реализация определяет будет ли исходящее сообщение буферизовано:
  a) сообщение помещается в буфер, вызов MPI_Send завершается до вызова советующего MPI_Recv
  b) буфер недоступен, вызов MPI_Send не завершится пока не будет вызван соответствующий MPI_Recv (non-local)

- **Режим с буферизацией** (Buffered mode, local) – завершение MPI_Bsend не зависит от того, вызван ли соответствующий MPI_Recv; исходящее сообщение помещается в буфер, вызов MPI_Bsend завершается

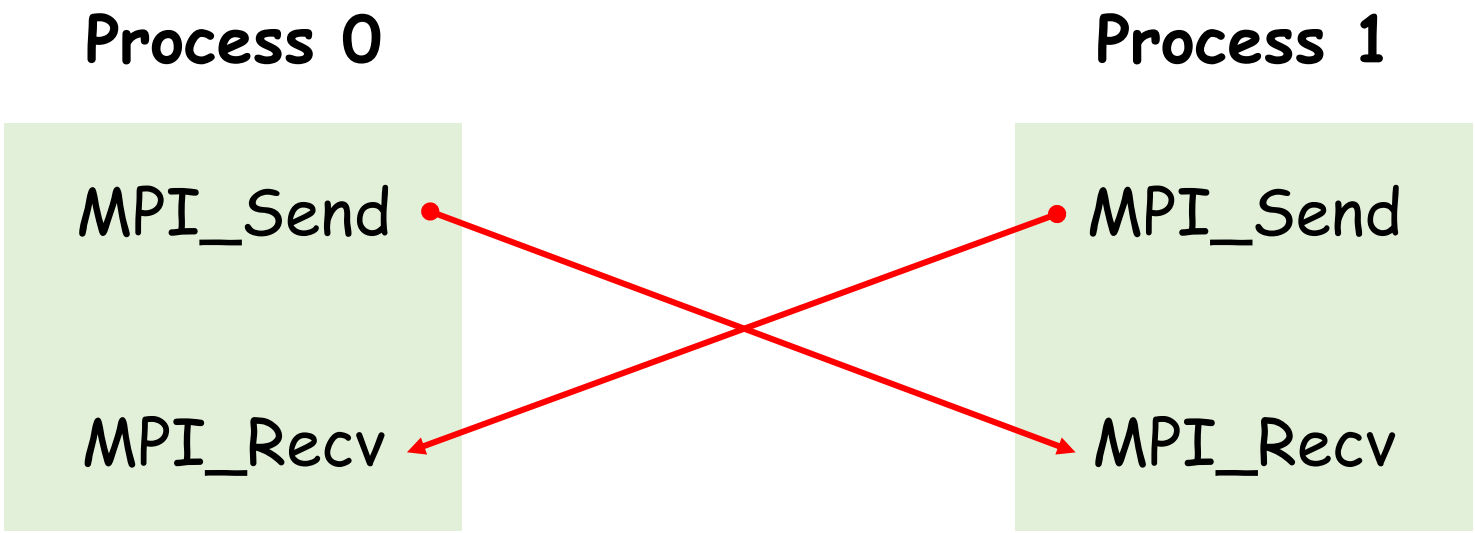
- **Синхронный режим** (Synchronous mode, non-local + synchronization) – вызов MPI_Ssend завершается если соответствующий вызова MPI_Recv начал прием сообщения

- **Режим с передачей по готовности** (Ready communication mode) – вызов MPI_Rsend может начать передачу сообщения если соответствующий MPI_Recv уже вызван (позволяет избежать процедуры “рукопожатия” для сокращения времени обмена)

Взаимная блокировка процессов



Взаимная блокировка процессов



- Поменять порядок операций Send/Recv
- Использовать неблокирующие операции
- Использовать функцию совмещенного обмена 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!

Неблокирующие функции Send/Recv (Non-blocking)

- Возврат из функции происходит сразу после инициализации процесса передачи/приема
  - Буфер использовать нельзя до завершения операции
- Передача
  - `MPI_Isend(..., MPI_Request *request)`
  - `MPI_Ibsend(..., MPI_Request *request)`
  - `MPI_Issend(..., MPI_Request *request)`
  - `MPI_Irsend(..., MPI_Request *request)`
- Прием
  - `MPI_Irecv(..., MPI_Request *request)`

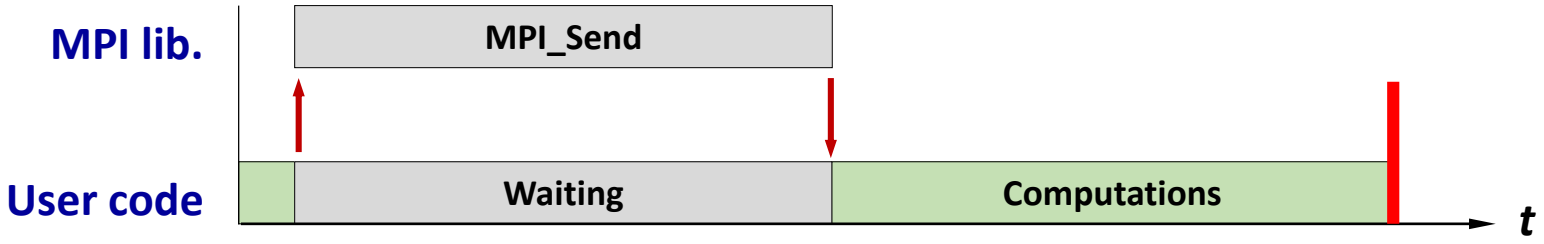
Ожидание завершения неблокирующей операции

- **Блокирующее ожидание завершения операции**
- `int MPI_Wait(MPI_Request *request, MPI_Status *status)`
- `int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status)`
- `int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])`

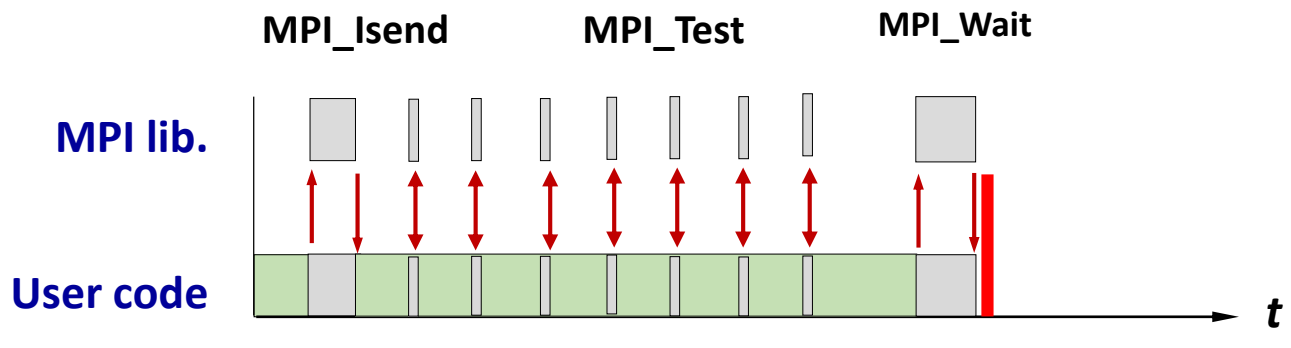
- **Блокирующая проверка состояния операции**
- `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
- `int MPI_Testany(int count, MPI_Request array_of_requests[], int *index, int *flag, MPI_Status *status)`
- `int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, MPI_Status array_of_statuses[])`

Совмещение обменов и вычислений (Overlaping)

Использование блокирующих функций



Использование неблокирующих функций




```
MPI_Isend(buf, count, MPI_INT, 1, 0,
          MPI_COMM_WORLD, &req);

do {
    //
    // Вычисления (не использовать buf)

    MPI_Test(&req, &flag, &status);
} while (!flag)
```

Hello, MPI World (3)!

```
int main(int argc, char **argv) {
    // ...
    char inbuf_prev[50], inbuf_next[50], outbuf_prev[50], outbuf_next[50];
    MPI_Request reqs[4];
    MPI_Status stats[4];

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

    snprintf(outbuf_prev, NELEMS(outbuf_prev), "Hello, prev. I am %d of %d on %s",
             rank, commsize, host);
    snprintf(outbuf_next, NELEMS(outbuf_next), "Hello, next. I am %d of %d on %s",
             rank, commsize, host);
    MPI_Isend(outbuf_prev, NELEMS(outbuf_prev), MPI_CHAR, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
    MPI_Isend(outbuf_next, NELEMS(outbuf_next), MPI_CHAR, next, tag2, MPI_COMM_WORLD, &reqs[1]);
    MPI_Irecv(inbuf_prev, NELEMS(inbuf_prev), MPI_CHAR, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
    MPI_Irecv(inbuf_next, NELEMS(inbuf_next), MPI_CHAR, next, tag1, MPI_COMM_WORLD, &reqs[3]);

    MPI_Waitall(4, reqs, stats);
    printf("[%d] Msg from %d (prev): '%s'\n", rank, stats[2].MPI_SOURCE, inbuf_prev);
    printf("[%d] Msg from %d (next): '%s'\n", rank, stats[3].MPI_SOURCE, inbuf_next);
    // ...
}
```

Неблокирующая проверка сообщений

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

- В параметре flag возвращает значение 1, если сообщение с подходящими атрибутами уже может быть принято и 0 в противном случае
- В параметре status возвращает информацию об обнаруженном сообщении (если flag == 1)

Постоянные запросы (persistent)

- Постоянные функции **привязывают** аргументы к дескриптору запроса (persistent request), дальнейшие вызовы операции осуществляется по дескриптору запроса
- Позволяет сократить время выполнения запроса

- `int MPI_Send_init(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- `int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`

- **Запуск операции (например, в цикле)**
- `int MPI_Start(MPI_Request *request)`
- `int MPI_Startall(int count, MPI_Request array_of_requests[])`

Коллективные обмены (Collective communications)

Трансляционный обмен (One-to-all)
- MPI_Bcast
- MPI_Scatter
- MPI_Scatterv

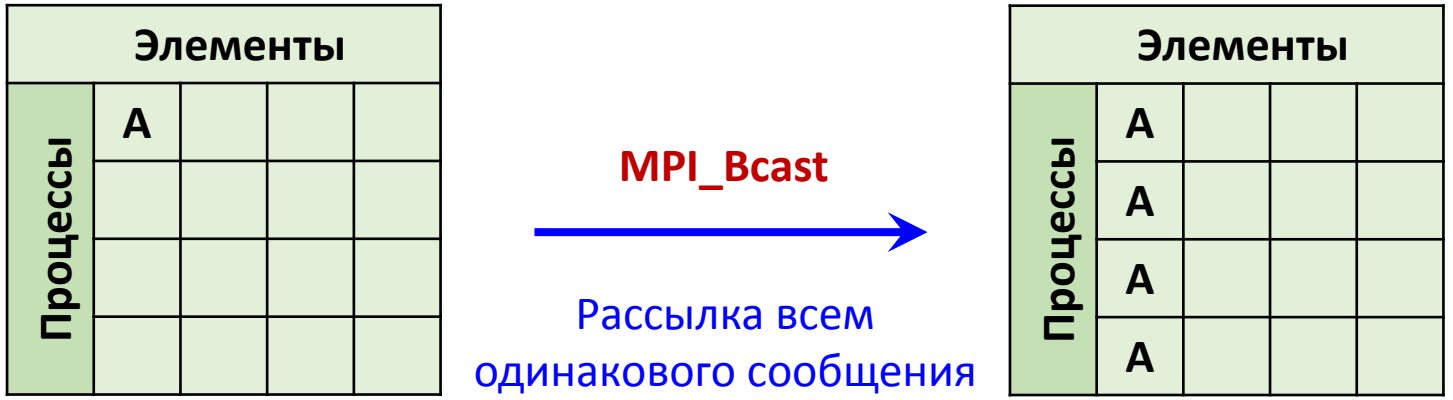
Коллекторный обмен (All-to-one)
- MPI_Gather
- MPI_Gatherv
- MPI_Reduce

Трансляционно-циклический обмен (All-to-all)
- MPI_Allgather
- MPI_Allgatherv
- MPI_Alltoall
- MPI_Alltoallv
- MPI_Allreduce
- MPI_Reduce_scatter

- Участвуют все процессы коммуникатора
- Коллективная функция должна быть вызвана каждым процессом коммуникатора
- Коллективные и двусторонние обмены в рамках одного коммуникатора используют различные контексты

MPI_Bcast

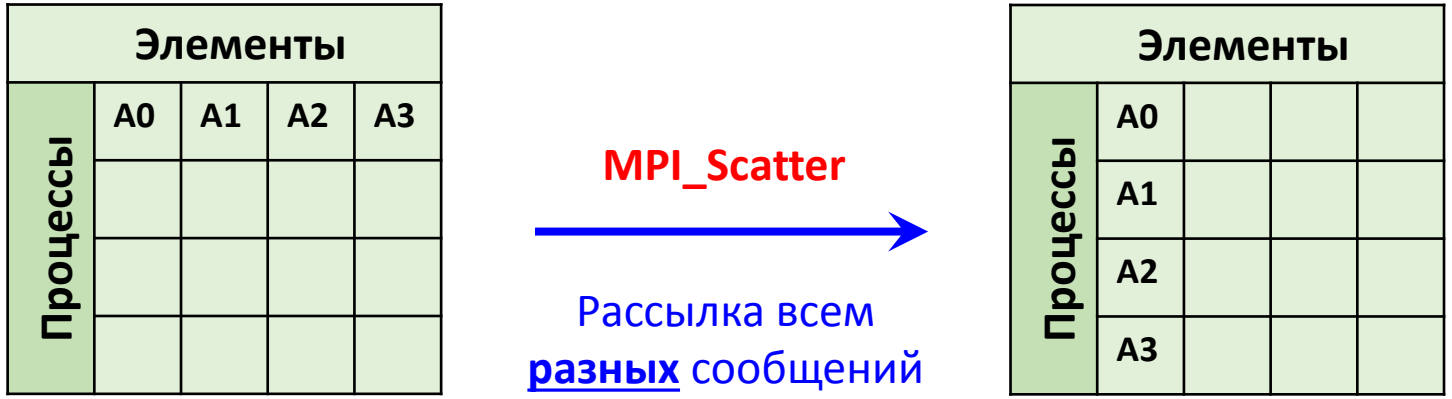
```
int MPI_Bcast(void *buf, int count,
              MPI_Datatype datatype,
              int root, MPI_Comm comm)
```



- **MPI_Bcast** – рассылка всем процессам сообщения buf
- Если номер процесса совпадает с root, то он отправитель, иначе – приемник

MPI_Scatter

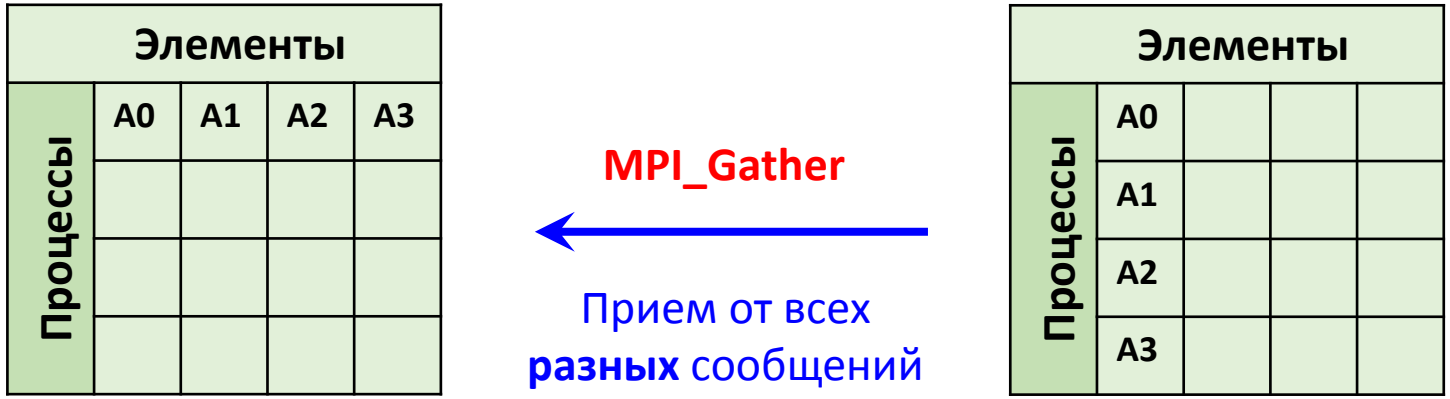
```
int MPI_Scatter(void *sendbuf, int sendcnt,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcnt,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```



- Размер **sendbuf =** sizeof(sendtype) * sendcnt * commsize
- Размер **recvbuf =** sizeof(sendtype) * recvcnt

MPI_Gather

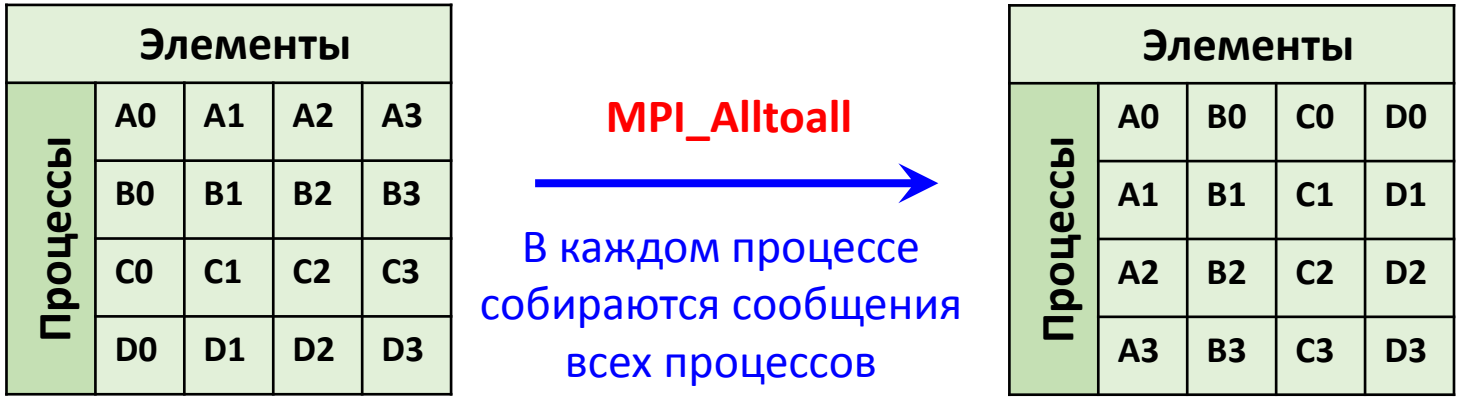
```
int MPI_Gather(void *sendbuf, int sendcnt,
               MPI_Datatype sendtype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```



- Размер **sendbuf**: sizeof(sendtype) * sendcnt
- Размер **recvbuf**: sizeof(sendtype) * sendcnt * commsize

MPI_Alltoall

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                 MPI_Comm comm)
```



- Размер sendbuf: sizeof(sendtype) * sendcount * commsize
- Размер recvbuf: sizeof(recvtype) * recvcount * commsize

All-to-all

```
int MPI_Allgather(void *sendbuf, int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm comm)

int MPI_Allgatherv(void *sendbuf, int sendcount,
                   MPI_Datatype sendtype,
                   void *recvbuf, int *recvcounts,
                   int *displs,
                   MPI_Datatype recvtype,
                   MPI_Comm comm)

int MPI_Allreduce(void *sendbuf, void *recvbuf,
                  int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```

MPI_Reduce

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)
```

- Размер sendbuf: sizeof(datatype) * count
- Размер recvbuf: sizeof(datatype) * count

Операции MPI_Reduce

- MPI_MAX
- MPI_MIN
- MPI_MAXLOC
- MPI_MINLOC
- MPI_SUM
- MPI_PROD
- MPI_LAND
- MPI_LOR
- MPI_LXOR
- MPI_BAND
- MPI_BOR
- MPI_BXOR

```
int MPI_Op_create(MPI_User_function *function,
                  int commute, MPI_Op *op)
```

- Операция пользователя должна быть ассоциативной
  A * (B * C) = (A * B) * C
- Если commute = 1, то операция коммутативная
  A * B = B * A

Барьерная синхронизация

```
int MPI_Barrier(MPI_Comm comm)
```

- Блокирует работу процессов коммуникатора, вызвавших данную функцию, до тех пор, пока все процессы не выполнят эту процедуру

Неблокирующие коллективные операции (MPI 3.0)

MPI 3.0

- **Неблокирующий коллективный обмен (Non-blocking collective communication)** – коллективная операция, выход из которой осуществляется не дожидаясь завершения операций обменов
- Пользователю возвращается дескриптор запроса (request), который он может использовать для проверки состояния операции
- Цель – обеспечить возможность совмещения вычислений и обменов информацией

```
int MPI_Ibcast(void *buf, int count,
               MPI_Datatype datatype,
               int root, MPI_Comm comm,
               MPI_Request *request)
```

Неблокирующие коллективные операции (MPI 3.0)

```
MPI_Request req;

MPI_Ibcast(buf, count, MPI_INT, 0, MPI_COMM_WORLD, &req);
while (!flag) {

    // Вычисления...

    // Проверяем состояние операции
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
}
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Вычисление числа π

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

    int n = 1000000000;
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    double h = 1.0 / (double)n;
    double sum = 0.0;
    for (int i = rank + 1; i <= n; i += commsize) {
        double x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x * x);
    }
    double pi_local = h * sum;

    double pi = 0.0;
    MPI_Reduce(&pi_local, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) printf("PI is approximately %.16f\n", pi);
    MPI_Finalize();
    return 0;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

$$\pi \approx h \sum_{i=1}^{n} \frac{4}{1+(h(i-0.5))^2} \qquad h = \frac{1}{n}$$

Итерации циклически (round-robin) распределены межу процессами

Обработка изображения (contrast)

```
npixels = width * height;
npixels_per_process = npixels / commsize;
```

Обработка изображения (contrast)

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

    int width, height, npixels, npixels_per_process;
    uint8_t *pixels = NULL;
    if (rank == 0) {
        width = 15360;  // 15360 x 8640: 16K Digital Cinema (UHDTV) ~ 127 MiB
        height = 8640;
        npixels = width * height;
        pixels = xmalloc(sizeof(*pixels) * npixels);
        for (int i = 0; i < npixels; i++)
            pixels[i] = rand() % 255;
    }

    MPI_Bcast(&npixels, 1, MPI_INT, 0, MPI_COMM_WORLD);  // Send size of image
    npixels_per_process = npixels / commsize;
    uint8_t *rbuf = xmalloc(sizeof(*rbuf) * npixels_per_process);
    // Send a part of image to each process
    MPI_Scatter(pixels, npixels_per_process, MPI_UINT8_T, rbuf, npixels_per_process, MPI_UINT8_T,
                0, MPI_COMM_WORLD);
```

Обработка изображения (contrast, 2)

```
int sum_local = 0;
for (int i = 0; i < npixels_per_process; i++)
    sum_local += rbuf[i] * rbuf[i];

/* Calculate global sum of the squares */
int sum = 0;
// MPI_Reduce(&sum_local, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Allreduce(&sum_local, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

double rms;
// if (rank == 0)
rms = sqrt((double)sum / (double)npixels);

//MPI_Bcast(&rms, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Обработка изображения (contrast, 3)

```
    /* Contrast operation on subimage */
    for (int i = 0; i < npixels_per_process; i++) {
        int pixel = 2 * rbuf[i] - rms;
        if (pixel < 0)
            rbuf[i] = 0;
        else if (pixel > 255)
            rbuf[i] = 255;
        else
            rbuf[i] = pixel;
    }
    MPI_Gather(rbuf, npixels_per_process, MPI_UINT8_T, pixels,
               npixels_per_process, MPI_UINT8_T, 0, MPI_COMM_WORLD);
    if (rank == 0)
        // Save image...

    free(rbuf);
    if (rank == 0)
        free(pixels);
    MPI_Finalize();
}
```

Пользовательские типы данных (MPI Derived Data Types)

- Как передать структуру C/C++ в другой процесс?
- Как передать другом процессу столбец матрицы?
  (в C/C++ массивы хранятся в памяти строка за строкой – row-major order,
  в Fortran столбец за столбцом – column-major order)
- Как реализовать прием сообщений различных размеров
  (заголовок сообщения содержит его тип, размер)?

Пользовательские типы данных (MPI Derived Data Types)

```
typedef struct {
    double x;
    double y;
    double z;
    double f;
    int data[8];
} particle_t;

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

    int nparticles = 1000;
    particle_t *particles = malloc(sizeof(*particles) * nparticles);
```

Как передать массив частиц другому процессу?

Пользовательские типы данных (MPI Derived Data Types, 2)

```
/* Create data type for message of type particle_t */
MPI_Datatype types[5] = {MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE,
                         MPI_DOUBLE, MPI_INT};
int blocklens[5] = {1, 1, 1, 1, 8};

MPI_Aint displs[0];
displs[0] = offsetof(particle_t, x);
displs[1] = offsetof(particle_t, y);
displs[2] = offsetof(particle_t, z);
displs[3] = offsetof(particle_t, f);
displs[4] = offsetof(particle_t, data);

MPI_Datatype parttype;
MPI_Type_create_struct(5, blocklens, displs, types, &parttype);
MPI_Type_commit(&parttype);
```

Пользовательские типы данных (MPI Derived Data Types, 3)

```
    /* Init particles */
    if (rank == 0) {
        // Random positions in simulation box
        for (int i = 0; i < nparticles; i++) {
            particles[i].x = rand() % 10000;
            particles[i].y = rand() % 10000;
            particles[i].z = rand() % 10000;
            particles[i].f = 0.0;
        }
    }
    MPI_Bcast(particles, nparticles, parttype, 0, MPI_COMM_WORLD);

    MPI_Type_free(&parttype);
    free(particles);
    MPI_Finalize( );
    return 0;
}
```

Упаковка данных (MPI_Pack)

```
int main(int argc, char **argv)
{
    int rank, packsize, position;
    int a;
    double b;
    uint8_t packbuf[100];

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

    if (rank == 0) {
        a = 15;
        b = 3.14;
```

Как передать int a и double b одним сообщением ?

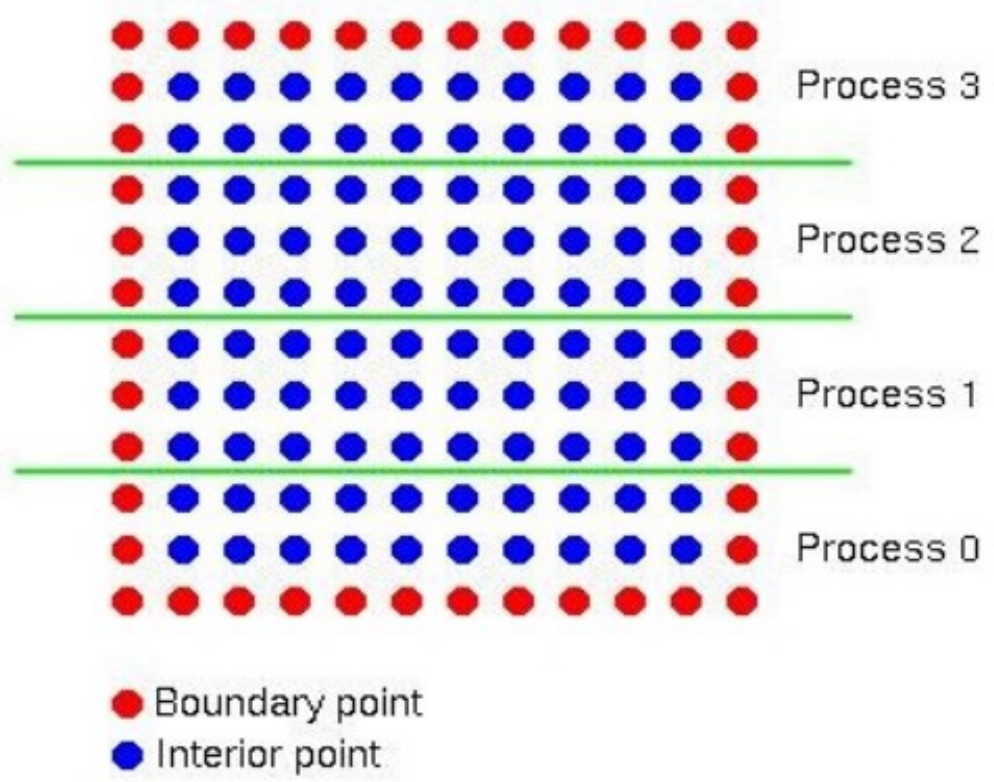
Упаковка данных (MPI_Pack, 2)

```
        packsize = 0; /* Pack data into the buffer */
        MPI_Pack(&a, 1, MPI_INT, packbuf, 100, &packsize, MPI_COMM_WORLD);
        MPI_Pack(&b, 1, MPI_DOUBLE, packbuf, 100, &packsize, MPI_COMM_WORLD);
    }

    MPI_Bcast(&packsize, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(packbuf, packsize, MPI_PACKED, 0, MPI_COMM_WORLD);
    if (rank != 0) {
        position = 0; /* Unpack data */
        MPI_Unpack(packbuf, packsize, &position, &a, 1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(packbuf, packsize, &position, &b, 1, MPI_DOUBLE,
                   MPI_COMM_WORLD);
    }
    printf("Process %d unpacked %d and %lf\n", rank, a, b);
    MPI_Finalize( );
    return 0;
}
```

Виртуальные топологии (Virtual topologies)

- Позволяют задать удобную схему адресации процессов, соответствующую структуре алгоритма
- Не связаны с физической топологией сети
- Могут использоваться для оптимизации распределения процессов по процессорам (Task mapping, Pinning, Task allocation)
- Поддерживаемые типы топологий
  - Декартова топология (*k*-мерная решетка)
  - Произвольный граф

Численное решение уравнения Лапласа методом Якоби




```
while (not converged) {
    for (i, j)
        xnew[i][j] =(x[i+1][j] + x[i-1][j] +
                     x[i][j+1] + x[i][j-1])/4;
    for (i,j)
        x[i][j] = xnew[i][j];
}
```

Схема пульсации

- Отправить свои границы соседям
- Получить границы от соседей
- Вычислить значения внутри своей полосы

1D decomposition

[*] Эндрюс Г. **Основы многопоточного, параллельного и распределенного программирования**. - М.: Вильямс, 2003.

Jacobi (1)

```
/* This example handles a 12 x 12 mesh, on 4 processors only */
#define maxn 12

int main(int argc, char *argv[])
{
    int rank, value, size, errcnt, toterr, i, j, itcnt;
    int i_first, i_last;
    MPI_Status status;
    double diffnorm, gdiffnorm;
    double xlocal[(12 / 4) + 2][12];
    double xnew[(12 / 4) + 2][12];
    double x[maxn][maxn];

    MPI_Init(&argc, &argv);

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

    if (size != 4)
        MPI_Abort(MPI_COMM_WORLD, 1);
```

Jacobi (2)

```
/* xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper */
/* Read the data from the named file */
if (rank == 0) {
    FILE *fp;
    fp = fopen("in.dat", "r");
    if (!fp)
        MPI_Abort(MPI_COMM_WORLD, 1);
    /* This includes the top and bottom edge */
    for (i = maxn - 1; i >= 0; i--) {
        for (j = 0; j < maxn; j++) {
            fscanf(fp, "%lf", &x[i][j]);
        }
        fscanf(fp, "\n");
    }
}

// Scatters chunks of data - 1D decomposition
MPI_Scatter(x[0], maxn * (maxn / size), MPI_DOUBLE,
            xlocal[1], maxn * (maxn / size), MPI_DOUBLE,
            0, MPI_COMM_WORLD);
```

Jacobi (3)

```
/* Note that top and bottom processes have one less row of interior points */
i_first = 1;
i_last = maxn / size;
if (rank == 0) i_first++;
if (rank == size - 1) i_last--;

itcnt = 0;
do {
    /* Send up unless I'm at the top, then receive from below */
    if (rank > 0)
        MPI_Send(xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
                 MPI_COMM_WORLD);
    if (rank < size - 1)
        MPI_Recv(xlocal[maxn / size + 1], maxn, MPI_DOUBLE, rank + 1,
                 1, MPI_COMM_WORLD, &status);
    /* Send down unless I'm at the bottom */
    if (rank < size - 1)
        MPI_Send(xlocal[maxn / size], maxn, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
    if (rank > 0)
        MPI_Recv(xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, &status);
```

Jacobi (4)

```
        itcnt++; /* Compute new values (but not on boundary) */
        diffnorm = 0.0;
        for (i = i_first; i <= i_last; i++) {
            for (j = 1; j < maxn - 1; j++) {
                xnew[i][j] = (xlocal[i][j + 1] + xlocal[i][j - 1] +
                              xlocal[i + 1][j] + xlocal[i - 1][j]) / 4.0;
                diffnorm += (xnew[i][j] - xlocal[i][j]) *
                            (xnew[i][j] - xlocal[i][j]);
            }
        }

        /* Only transfer the interior points */
        for (i = i_first; i <= i_last; i++)
            for (j = 1; j < maxn - 1; j++)
                xlocal[i][j] = xnew[i][j];

        MPI_Allreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        gdiffnorm = sqrt(gdiffnorm);
        if (rank == 0) printf("At iteration %d, diff is %e\n", itcnt, gdiffnorm);
    } while (gdiffnorm > 1.0e-2 && itcnt < 100);
```

Jacobi (5)

```
    /* Collect the data into x and print it */
    MPI_Gather(xlocal[1], maxn * (maxn / size), MPI_DOUBLE,
               x, maxn * (maxn / size), MPI_DOUBLE, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("Final solution is\n");
        for (i = maxn - 1; i >= 0; i--) {
            for (j = 0; j < maxn; j++)
                printf("%f ", x[i][j]);
            printf("\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

Оптимизации

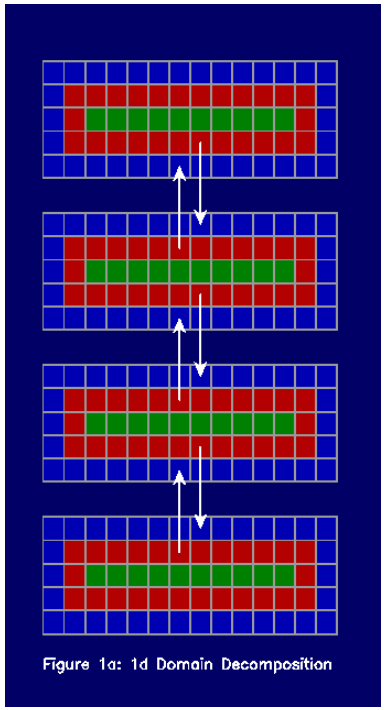
- Сокращение обменов
  - ❑ Обмениваться границами через итерацию
- 2D-декомпозиция?
- Совмещение вычислений и обмена данными?

Влияние способа декомпозиции

- **Время передачи данных между процессами (модель Дж. Хокни)**

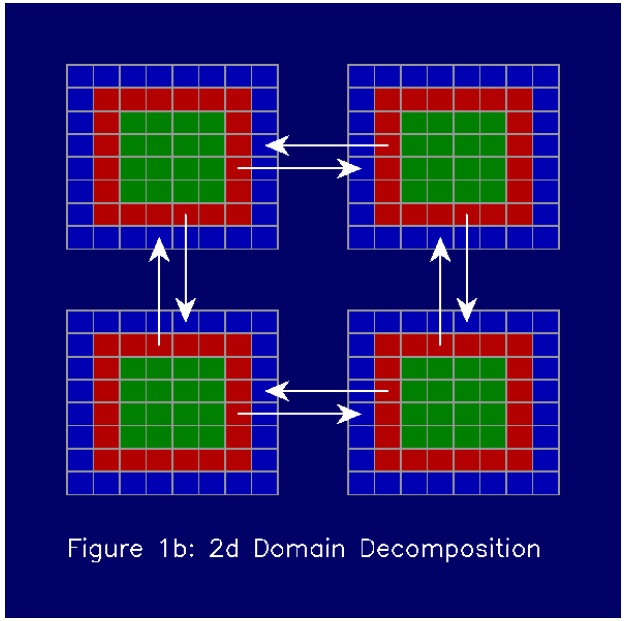
$$T = L + \frac{n}{B}$$

1D decomposition



$$\boldsymbol{T = 2(L + \frac{n}{B})}$$

2D decomposition



$$\boldsymbol{T = 4(L + \frac{n}{\sqrt{pB}})}$$

http://rcsg-gsir.imsb-dsgi.nrc-cnrc.gc.ca/benchmarks/parallel/poisson/poisson.htm

Совмещение обменов и вычислений

- Обновить края своей полосы
- Приготовиться к приему краев соседей (MPI_Irecv)
- Начать отправку своих краев соседям (MPI_Isend)
- Обновить внутренние клетки своей полосы
- Дождаться завершения приема граничных элементов от соседей (MPI_Waitall)

Домашнее чтение

- Pavan Balaji, Torsten Hoefler. **Advanced Parallel Programming with MPI-1, MPI-2, and MPI-3** // ACM Symposium on Principles and Practice of Parallel Programming, 2013 http://htor.inf.ethz.ch/teaching/mpi_tutorials/ppopp13/2013-02-24-ppopp-mpi-advanced.pdf

- Rolf Rabenseifner, Georg Hager, Gabriele Jost. **Hybrid MPI and OpenMP Parallel Programming** // Day-long tutorial on Hybrid MPI and OpenMP Parallel Programming from SC13, 2013 http://openmp.org/wp/sc13-tutorial-hybrid-mpi-and-openmp-parallel-programming/