# Семинар 10
# Стандарт MPI (часть 3)

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

# 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)

# 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)`

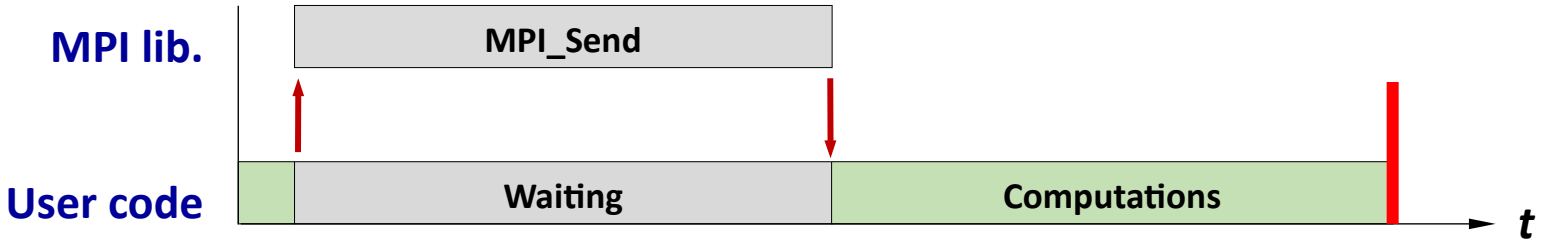
# Ожидание завершения неблокирующей операции

- **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[])`

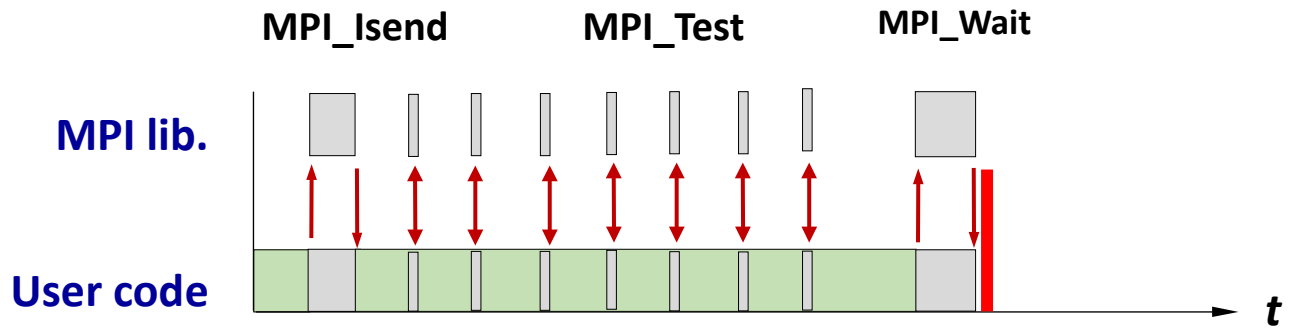
- **Блокирующая проверка состояния операции**
- `int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
- `int MPI_Testany(int count, MPI_Request array_of_requests[], int *index, int *flag, MPI_Status *status)`
- `int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, MPI_Status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

# Ring example

- **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

# 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 прочитано 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

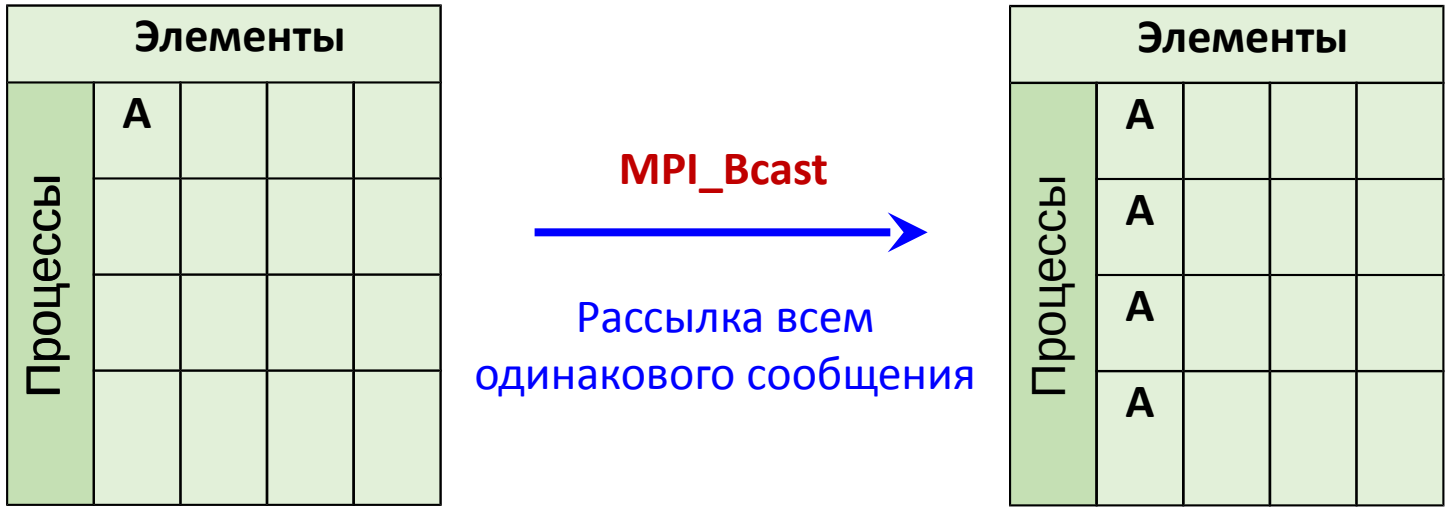
**Коллекторный обмен (All-to-one)**
- MPI_Gather
- MPI_Gatherv
- MPI_Reduce

**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

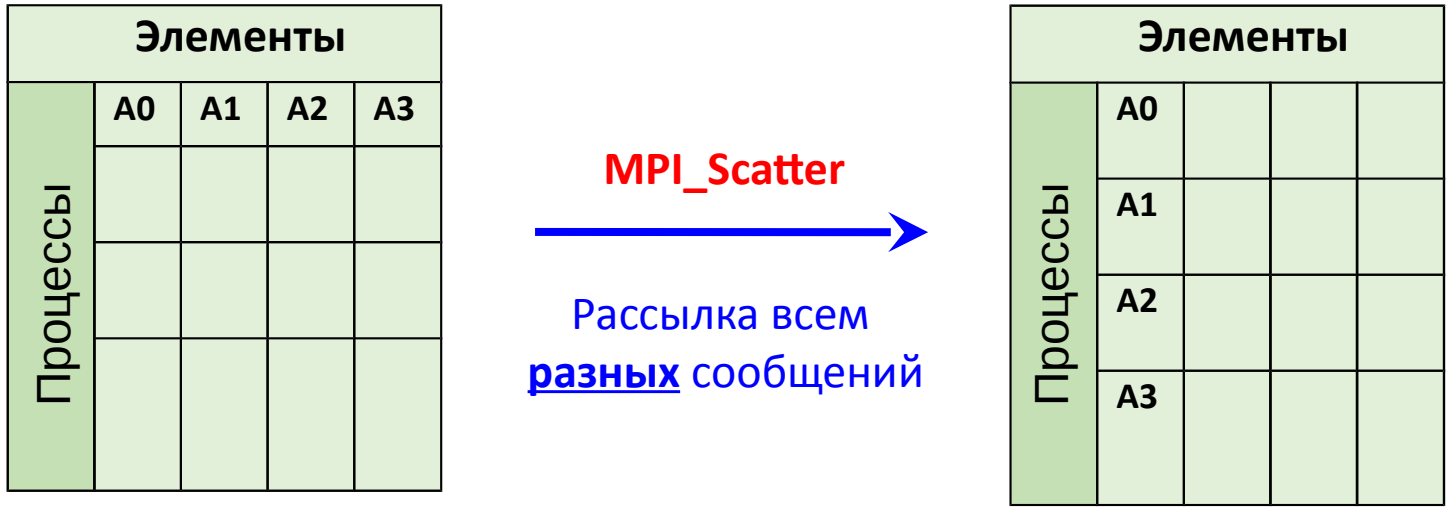
```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```




- **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

# Задание

- Разработать на MPI параллельную программу численного интегрирования методом прямоугольников (семинар 2, пример `1_integrate`)
- Провести анализ эффективности параллельной программы