Лекция 10:
Стандарт MPI
(MPI – Message Passing Interface)

Курносов Михаил Георгиевич

к.т.н. доцент Кафедры вычислительных систем
Сибирский государственный университет
телекоммуникаций и информатики

http://www.mkurnosov.net

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

http://www.mpi-forum.org

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

Реализации стандарт MPI (библиотеки)

- **MPICH2** (Open source, Argone NL)

  http://www.mcs.anl.gov/research/projects/mpich2

  - ❑ MVAPICH2
  - ❑ IBM MPI
  - ❑ Cray MPI
  - ❑ Intel MPI
  - ❑ HP MPI
  - ❑ SiCortex MPI

- **Open MPI** (Open source, BSD License)

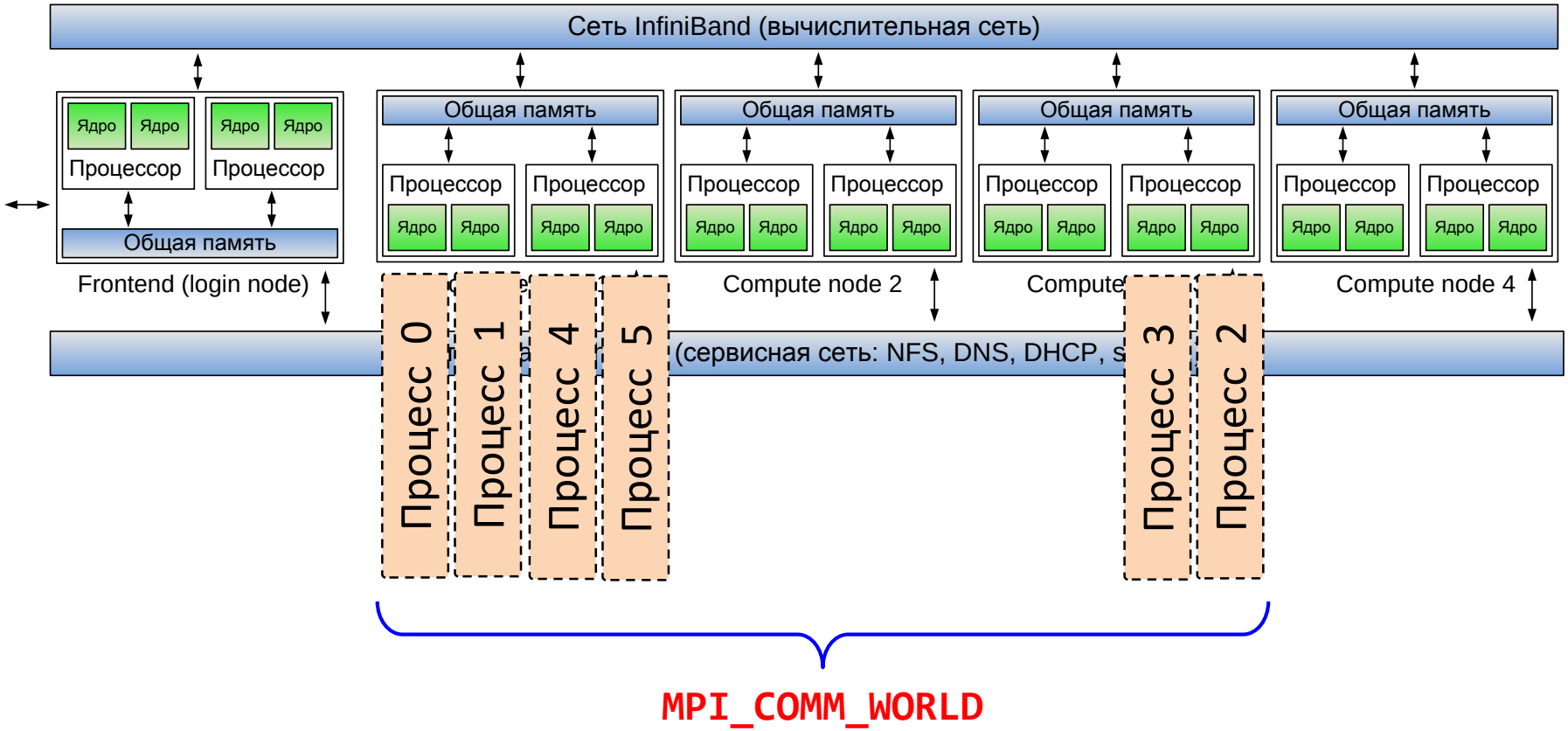
  http://www.open-mpi.org

  - ❑ Oracle MPI

Различия в реализациях стандарта 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
- **Реализованные протоколы дифференцированных 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)

Стандарт MPI



- **Коммуникатор (communicator)** – множество процессов, образующих логическую область для выполнения коллективных операций (обменов информацией и др.)
- В рамках коммуникатора ветви имеют номера: 0, 1, …, $n - 1$

MPI: Hello World

```
#include <mpi.h>

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);

    printf("Hello, World: process %d of %d\n",
           rank, commsize);

    MPI_Finalize();
    return 0;
}
```

Компиляция MPI-программ

Компиляция C-программы:

```
$ mpicc -o prog ./prog.c
```

Компиляция C++-программы:

```
$ mpicxx -o prog ./prog.cpp
```

Компиляция Fortran-программы:

```
$ mpif90 -o prog ./prog.f
```

Компиляция C-программы компилятором Intel C/C++:

```
$ mpicc -cc=icc -o prog ./prog.c
```

Запуск MPI-программ на кластере Jet

1. Формируем job-файл с ресурсным запросом (task.job):

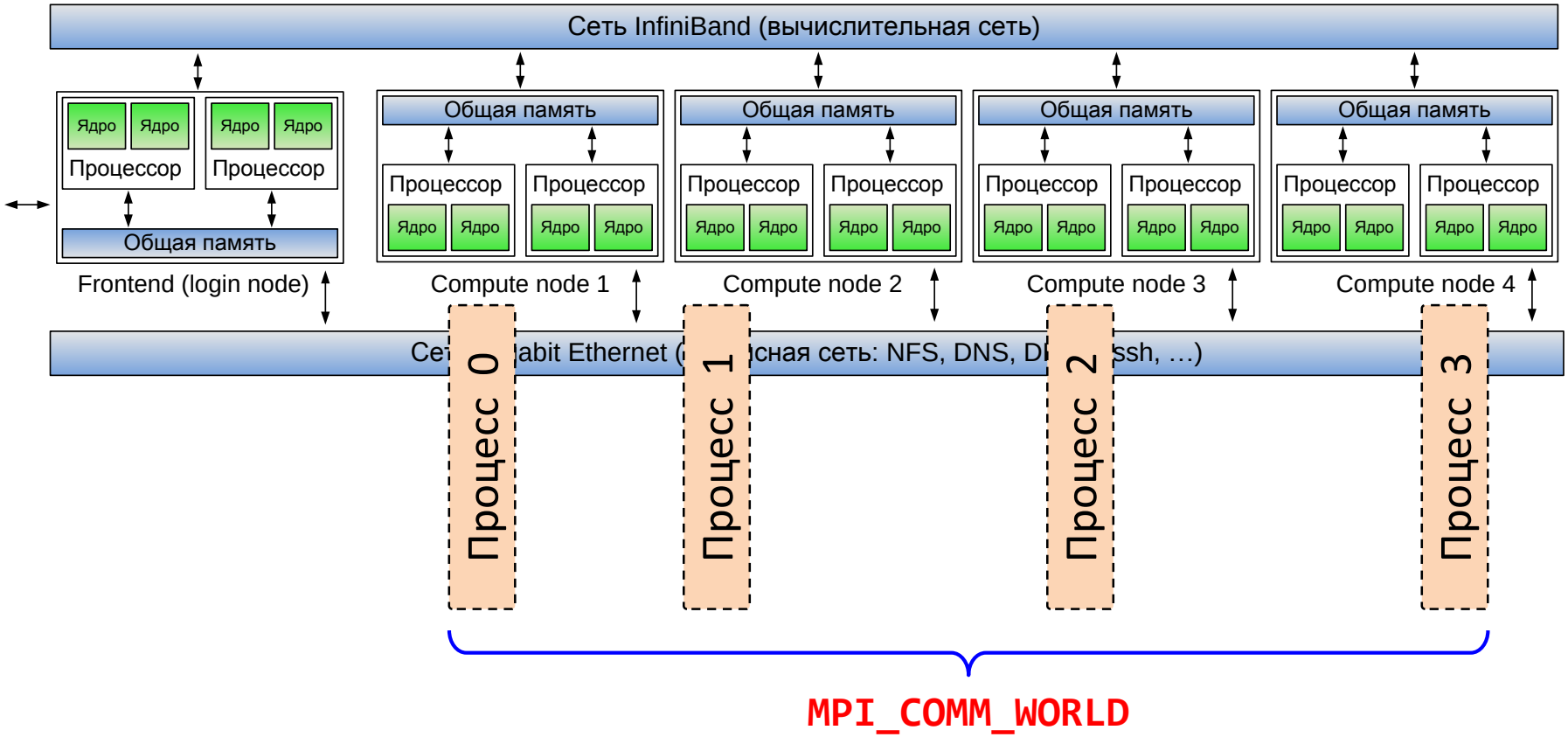
```
#PBS –N MyTask
#PBS -l nodes=4:ppn=1          # 4 узла, 1 процесс на узел
#PBS -j oe


cd $PBS_O_WORKDIR
mpiexec ./prog
```

2. Ставим задачу в очередь системы пакетной обработки заданий TORQUE

```
$ qsub task.job
7719.jet
```

Job ID

Выполнение MPI-программ



- Система TORQUE запустит задачу, когда для нее будут доступны ресурсы
- В данном примере на каждом узле могут также присутствовать 3 процесса других задач (пользователей)

Состояние заданий в очереди TORQUE

```
$ qstat
Job id            Name             User            Time Use S Queue
----------------- ---------------- --------------- -------- - -----
7732.jet          simmc            mkurnosov       00:00:00 C debug
7733.jet          Heat2D           mkurnosov              0 R debug
7734.jet          simmc            mkurnosov       00:00:00 C debug
7735.jet          mpitask          mkurnosov       00:00:00 C debug
7736.jet          simmc            mkurnosov       00:00:00 C debug
7737.jet          mpitask          mkurnosov       00:00:00 C debug
7738.jet          Heat2D           mkurnosov              0 R debug
```

Job state

- **C** – completed
- **R** – running
- **Q** – waiting in queue

Формирование ресурсных запросов (Job)

- 8 ветвей – все ветви на одном узле

```
#PBS -l nodes=1:ppn=8
```

- 6 ветвей – две ветви на каждом узле

```
#PBS -l nodes=3:ppn=2
```

- Команды управления задачами в очереди
  - **qdel** – удаление задачи из очереди
  - **qstat** – состояние очереди
  - **pbsnodes** – состояние вычислительных узлов

Стандарт MPI

- Двусторонние обмены (Point-to-point communication)
- Составные типы данных (Derived datatypes)
- Коллективные обмены (Collective communication)
- Группы процессов, коммуникаторы (Groups, communicators)
- Виртуальные топологии процессов
- Управление средой выполнения (Environmental management)
- Управление процессами (Process management)
- Односторонние обмены (One-sided communication)
- Файловый ввод-вывод (I/O)

Двусторонние обмены (Point-to-point)

- **Двусторонний обмен (Point-to-point communication)** – это обмен сообщением между двумя процессами: один инициирует передачу (send), а другой – получение сообщения (receive)

- Каждое сообщение снабжается конвертом (envelope):

  - номер процесса отправителя (**source**)
  - номер процесса получателя (**destination**)
  - 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

MPI_Send/MPI_Recv

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

- **buf** – указатель на буфер с информацией
- **count** – количество передаваемых элементов типа datatype
- **dest** – номер процесса получателя сообщения
- **tag** – тег сообщения

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

Пример MPI_Send/MPI_Recv

```
int main(int argc, char **argv)
{
    int rank;
    MPI_Status status;

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

    if (rank == 0)
        MPI_Send(buf, BUFSIZE, MPI_DOUBLE, 1, 1,
                 MPI_COMM_WORLD);
    else
        MPI_Recv(buf, BUFSIZE, MPI_DOUBLE, 0, 1,
                 MPI_COMM_WORLD, &status);

    MPI_Finalize();
    return 0;
}
```

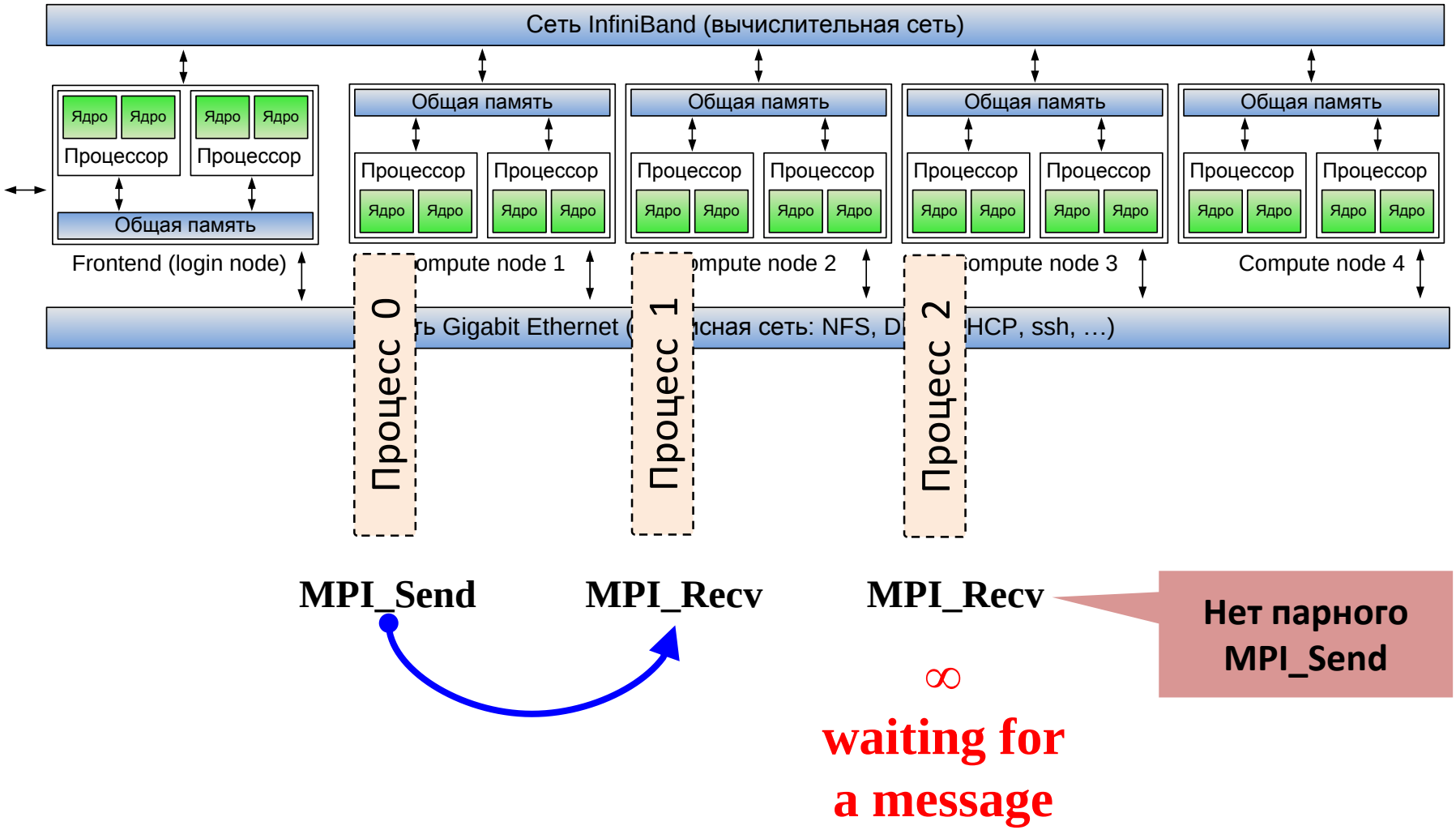
Пример MPI_Send/MPI_Recv

```
int main(int argc, char **argv)
{
    int rank;
    MPI_Status status;

    MPI_In
    MPI_Comm

    if
        MP

    else
        MPI_Re      ,     ZE,       OUBLE,    1,
                MPI_      _WORLD,    tatus);

    MPI_Finalize();
    return 0;
}
```

Infinite waiting

(if commsize > 2)

Бесконечное ожидание (commsize = 3)



Пример MPI_Send/MPI_Recv

```
int main(int argc, char **argv)
{
    int rank;
    MPI_Status status;

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

    if (rank == 0)
        MPI_Send(buf, BUFSIZE, MPI_DOUBLE, 1, 1,
                 MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(buf, BUFSIZE, MPI_DOUBLE, 0, 1,
                 MPI_COMM_WORLD, &status);

    MPI_Finalize();
    return 0;
}
```

Режимы передачи сообщений

- **Standard** (MPI_Send) – библиотека сама принимает решение буферизовать сообщение и сразу выходить из функции или дожидаться окончания передачи данных получателю (например, если нет свободного буфера)
- **Buffered** (MPI_Bsend) – отправляемое сообщение помещается в буфер и функция завершает свою работу. Сообщение будет отправлено библиотекой.
- **Synchronous** (MPI_Ssend) – функция завершает своё выполнение когда процесс-получатель вызвал MPI_Recv и начал копировать сообщение. После вызова функции можно безопасно использовать буфер.
- **Ready** (MPI_Rsend) – функция успешно выполняется только если процесс-получатель уже вызвал функцию MPI_Recv

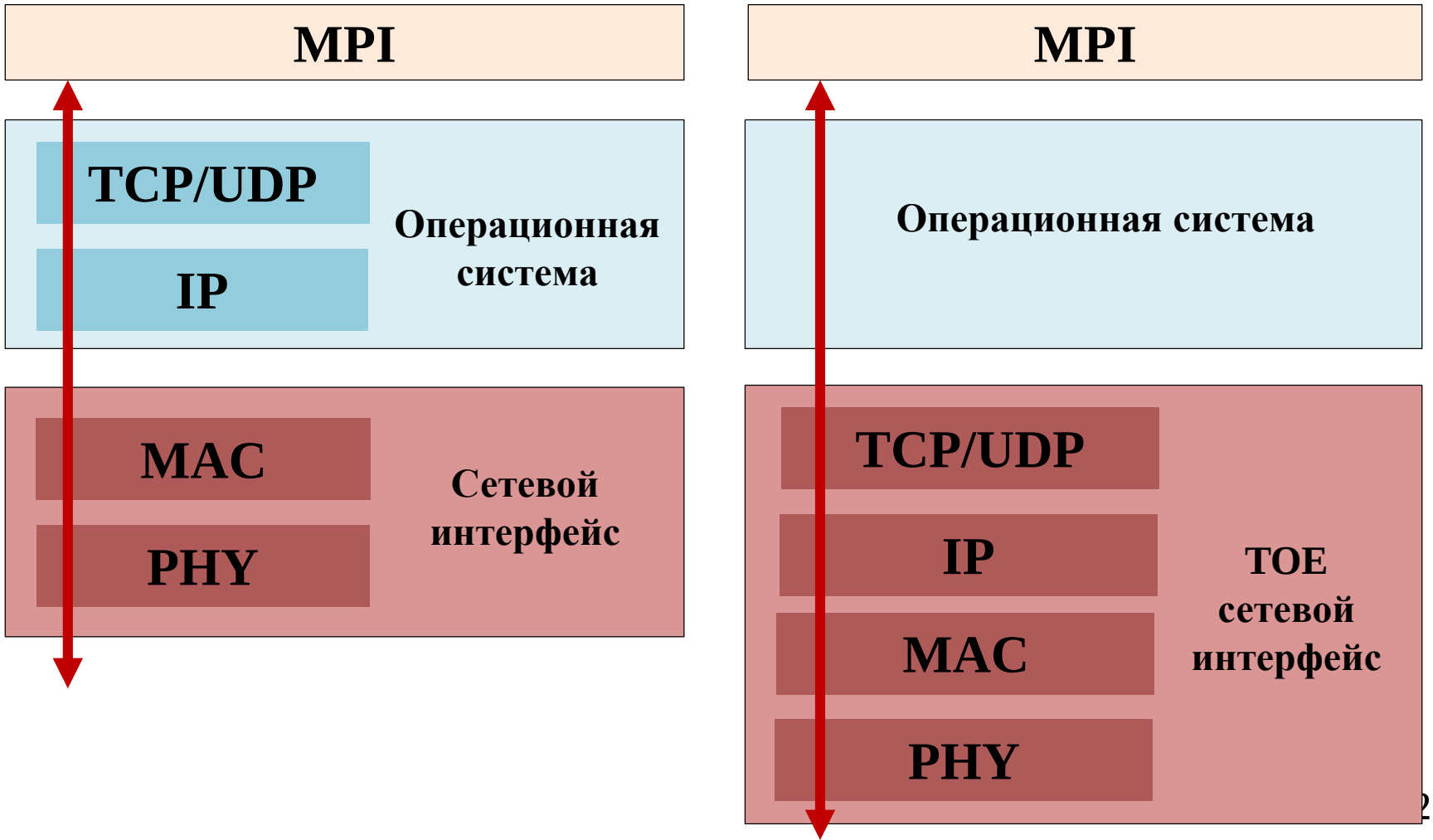
Неблокирующие двусторонние обмены

- **Неблокирующие передача/прием сообщений (Nonblocking point-to-point communication)** – операция, выход из которой осуществляется не дожидаясь завершения передачи информации
- Пользователю возвращается дескриптор запроса (request), который он может использовать для проверки состояния операции
- Цель – обеспечить возможность **совмещения вычислений и обменов информацией** (Overlap computations & communications)

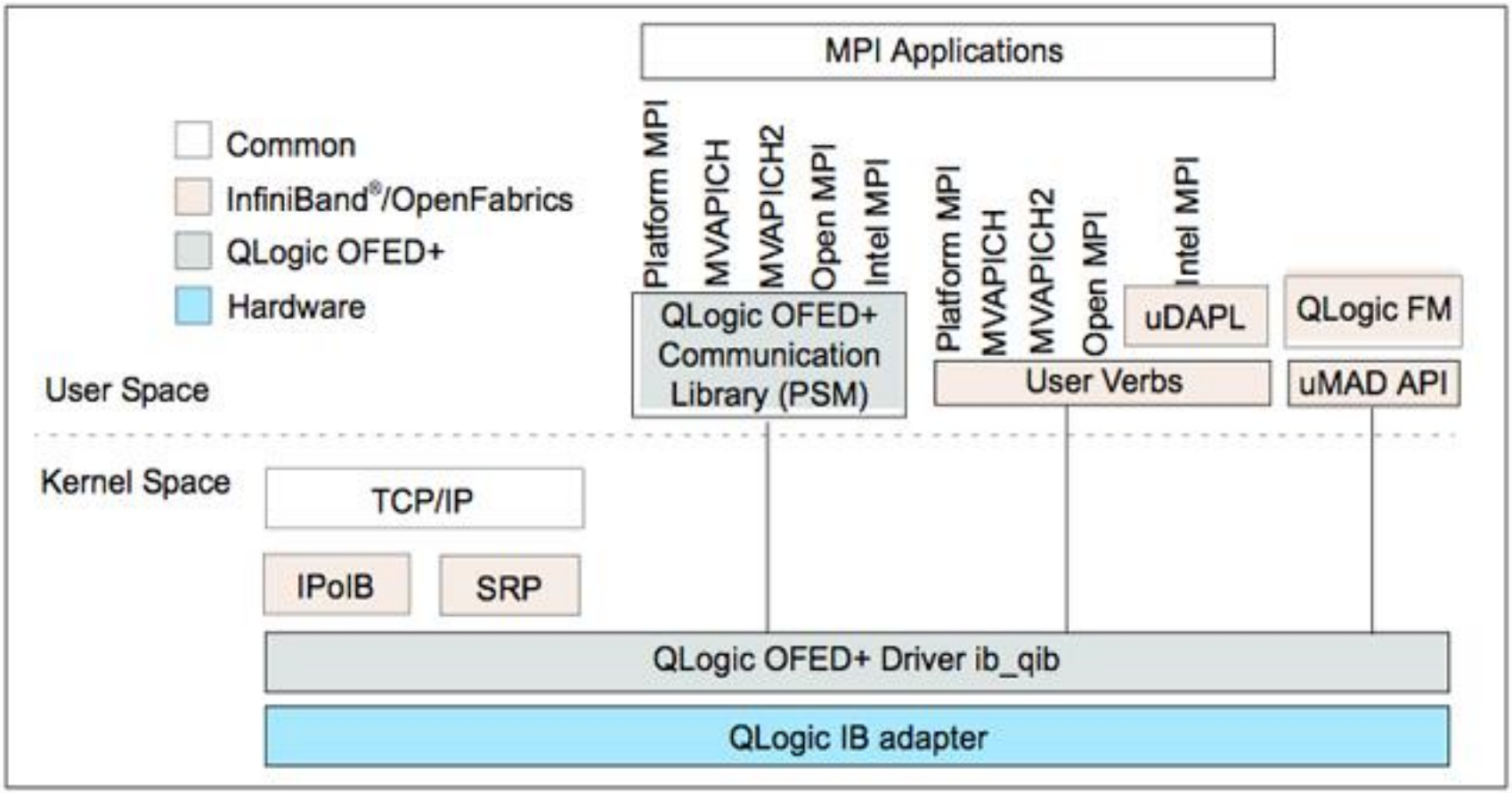
```
int MPI_Isend(const void *buf, int count,
              MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm,
              MPI_Request *request)
```

Network offload

Высокопроизводительные сетевые контроллеры аппаратурно реализуют часть функций сетевого стека => низкая латентность передачи сообщений, совмещение обменов и вычислений



InfiniBand aware MPI



http://www.mn.uio.no/astro/english/services/it/help/programming/mpi.html

Неблокирующие двусторонние обмены

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

  Блокирует выполнение процесса пока не закончится операция, ассоциированная с запросом request

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

  Возвращает flag = 1 если операция, ассоциированная с запросом request, завершена

Совмещение обменов и вычислений

```
MPI_Isend(buf, count, MPI_INT, 1, 0,
          MPI_COMM_WORLD, &req);
do {
    //
    // Вычисления ... (не использовать buf)
    //

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

Неблокирующие двусторонние обмены

- ```
  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[])
  ```

- ```
  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[])
  ```

Типы данных MPI (Datatypes)

- MPI_PACKED 1
- MPI_BYTE 1
- MPI_CHAR 1
- MPI_UNSIGNED_CHAR 1
- MPI_SIGNED_CHAR 1
- MPI_WCHAR 2
- MPI_SHORT 2
- MPI_UNSIGNED_SHORT 2
- MPI_INT 4
- MPI_UNSIGNED 4
- MPI_LONG 4
- MPI_UNSIGNED_LONG 4
- MPI_FLOAT 4
- MPI_DOUBLE 8
- MPI_LONG_DOUBLE 16
- MPI_CHARACTER 1
- MPI_LOGICAL 4
- MPI_INTEGER 4
- MPI_REAL 4
- MPI_DOUBLE_PRECISION 8
- MPI_COMPLEX 2*4
- MPI_DOUBLE_COMPLEX 2*8

В MPI имеется возможность создавать пользовательские типы данных (Derived Datatypes, структуры, части массивов)

Коллективные обмены (блокирующие)

Трансляционный обмен (One-to-all)

- MPI_Bcast
- MPI_Scatter
- MPI_Scatterv

Коллекторный обмен (All-to-one)

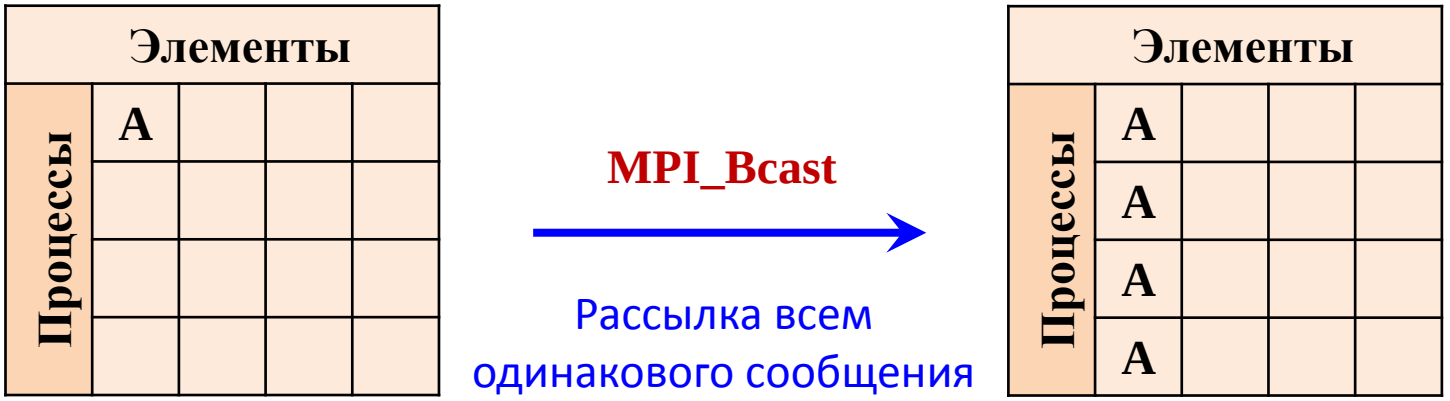
- MPI_Gather
- MPI_Gatherv
- MPI_Reduce

Трансляционно-циклический обмен (All-to-all)

- MPI_Allgather
- MPI_Allgatherv
- MPI_Alltoall
- MPI_Alltoallv
- MPI_Allreduce
- MPI_Reduce_scatter

MPI_Bcast (One-to-all broadcast)

```
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

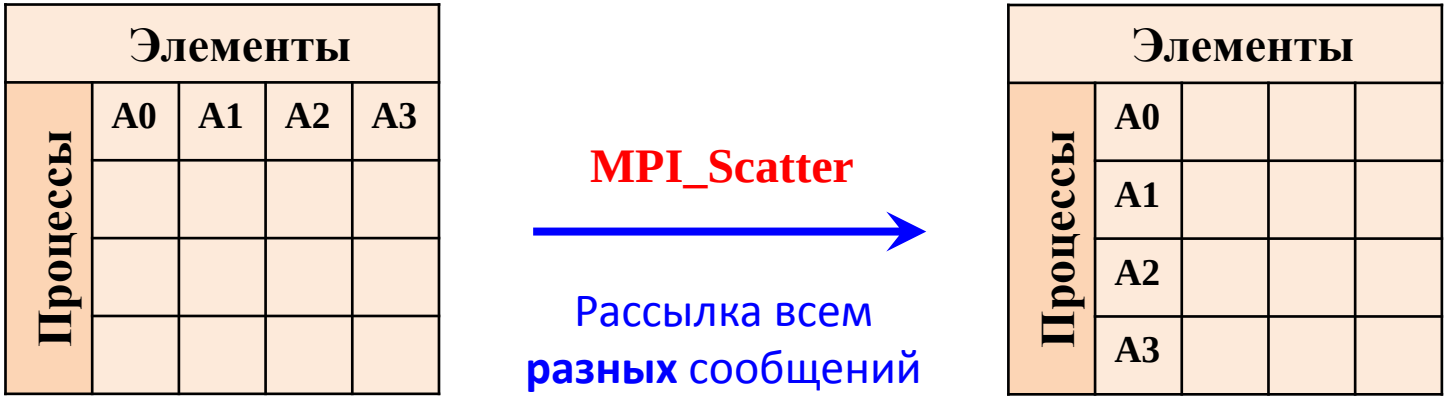
MPI_Bcast (One-to-all broadcast)

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

```
int main(int argc, char *argv[])
{
    double buf[n];
    MPI_Init(&argc,&argv);

    /* Code */

    // Передать массив buf всем процессам (и себе)
    MPI_Bcast(&buf, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```

MPI_Scatter (One-to-all broadcast)

```
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

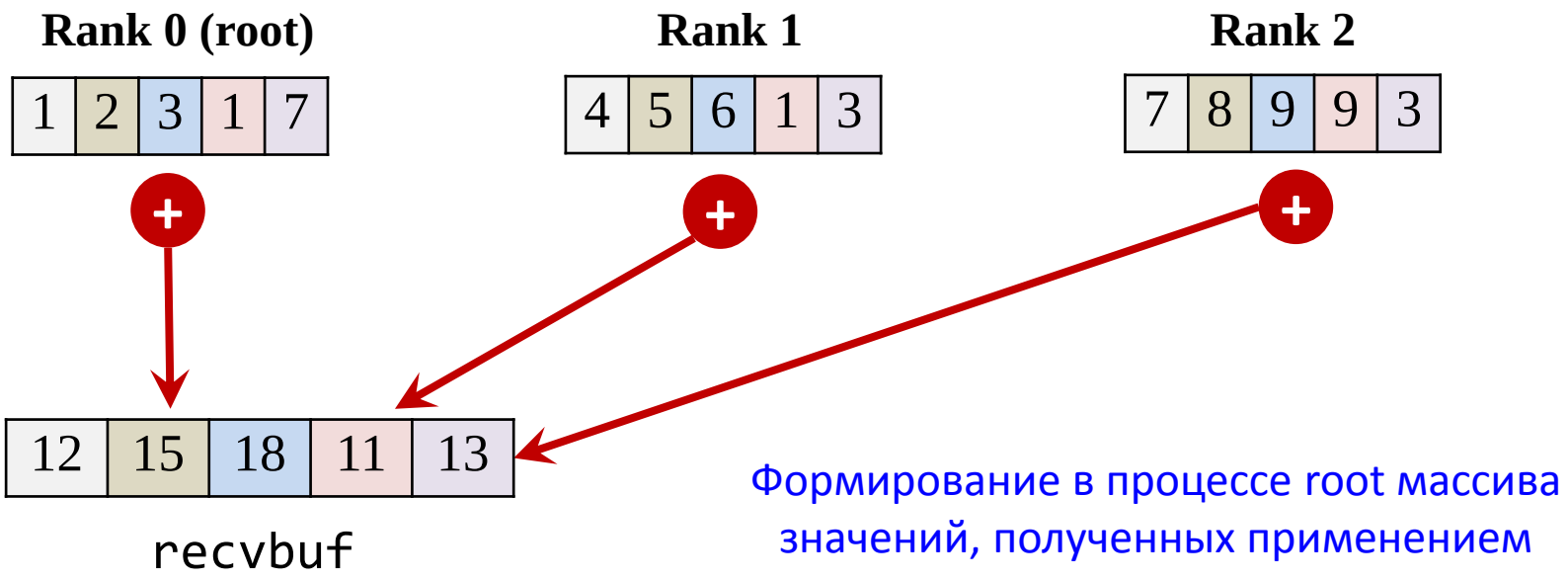
MPI_Gather (All-to-one)

```
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_Reduce (All-to-one)

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



Формирование в процессе root массива значений, полученных применением операции op

- 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

Пользовательская операция MPI_Reduce

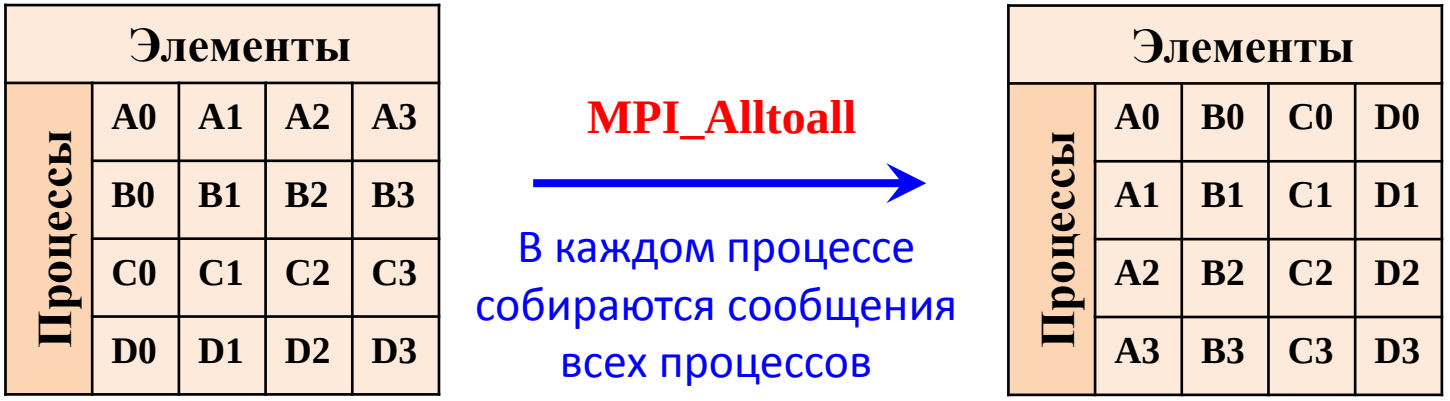
```
typedef struct {
    double real, imag;
} Complex;

Complex sbuf[100], rbuf[100];
MPI_Op complexmulop;
MPI_Datatype ctype;

MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
MPI_Type_commit(&ctype);

// Умножение комплексных чисел
MPI_Op_create(complex_mul, 1, &complexmulop);

MPI_Reduce(sbuf, rbuf, 100, ctype, complexmulop,
           root, comm);

MPI_Op_free(&complexmulop);
MPI_Type_free(&ctype);
```

Пользовательская операция MPI_Reduce

```
// Умножение массивов комплексных чисел
void complex_mul(void *inv, void *inoutv, int *len,
                 MPI_Datatype *datatype)
{
    int i;
    Complex c;
    Complex *in = (Complex *)inv;
            *inout = (Complex *)inoutv;

    for (i = 0; i < *len; i++) {
        c.real = inout->real * in->real -
                 inout->imag * in->imag;
        c.imag = inout->real * in->imag +
                 inout->imag * in->real;
        *inout = c;
        in++;
        inout++;
    }
}
```

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

MPI_Barrier

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

- Барьерная синхронизация процессов – ни один процесс не завершит выполнение этой функции пока все не войдут в неё

```
int main(int argc, char *argv[])
{
    /* Code */

    MPI_Barrier(MPI_COMM_WORLD);

    /* Code */

    MPI_Finalize();
    return 0;
}
```

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)
```

Реализация коллективных операций

- **Аппаратная реализация** коллективных операций – специализированные коммуникационные сети: IBM BlueGene/P Tree network, barrier network

- **Программная реализация** – коллективная операция реализуется на основе операций Send/Recv

- Программная реализация коллективных операций имеет наибольшее распространение (MPICH2, Open MPI)

Алгоритмы коллективных операций

One-to-all Broadcast (MPI_Bcast, MPI_Scatter)

- Алгоритм биномиального дерева (Binomial tree)
- Scatter + Allgather

All-to-one Broadcast (MPI_Reduce, MPI_Gather):

- Алгоритм биномиального дерева
- Reduce_Scatter + Gather

All-to-all Broadcast
(MPI_Allgather, MPI_Allreduce)

- Алгоритм рекурсивного сдваивания (Recursive doubling)
- Алгоритм Дж. Брука (J. Bruck, 1997)

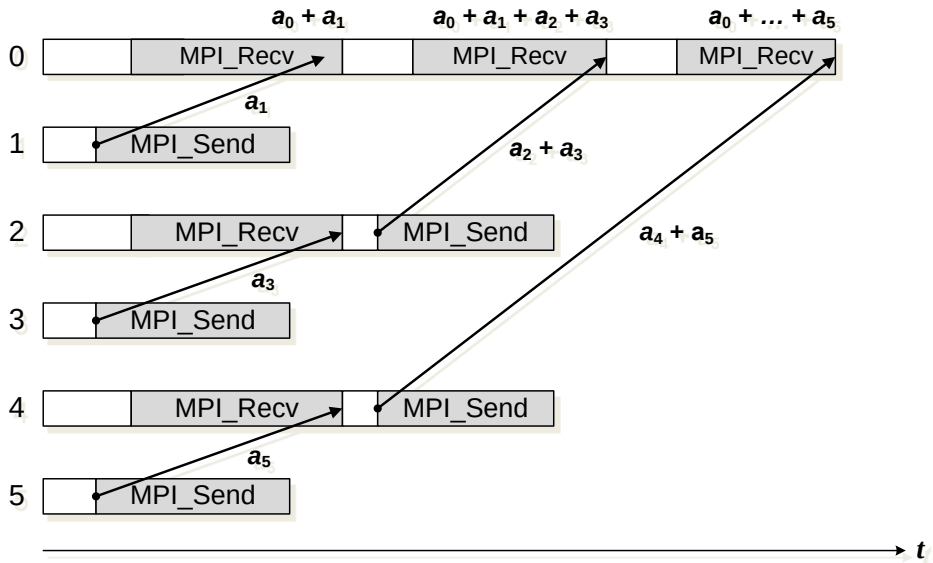


Диаграмма выполнения операции MPI_Reduce алгоритмом биномиального дерева (суммирование элементов 6 ветвей)

Реализация коллективных операций

- Как правило, для каждой коллективной операции имеется несколько алгоритмов ее реализации – выбор алгоритма основан на эвристике

mpich2-3.0.4

```
int MPIR_Allgather_intra(...)
{
    // ...
    tot_bytes = (MPI_Aint)recvcount * comm_size * type_size;
    if ((tot_bytes < MPIR_PARAM_ALLGATHER_LONG_MSG_SIZE) &&
        !(comm_size & (comm_size - 1))) {
        /* Short or medium size message and power-of-two no.
           of processes. Use recursive doubling algorithm */
        // ...
    else if (tot_bytes < MPIR_PARAM_ALLGATHER_SHORT_MSG_SIZE) {
        /* Short message and non-power-of-two no. of processes.
           Use Bruck algorithm (see description above). */
        // ...
    else {
        /* Long message or medium-size message and non-power-of-two
           no. of processes. Use ring algorithm. */
        // ...
    }
}
```

Выбор алгоритма коллективной операции

- **Как динамически (во время вызова функции) выбирать алгоритм реализации коллективной операции?**

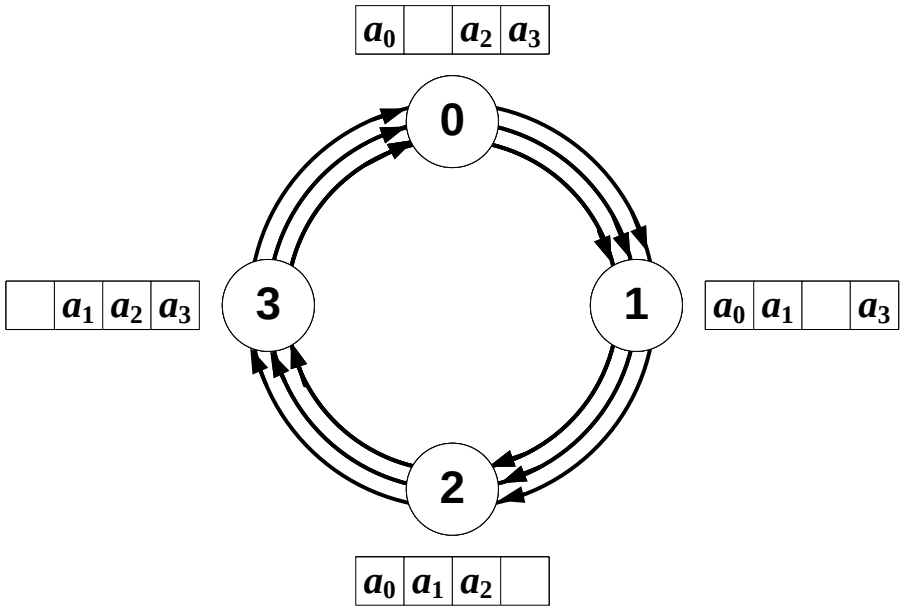
- **Учитывать только размер сообщения?**
  Такой выбор будет оптимальным только для той системы, на которой провели предварительные замеры времени выполнения операций (MPICH2, Open MPI)

- **Использовать данные предварительных замеров времени выполнения алгоритмов** (при установке библиотеки на вычислительную систему)?

- **Оценивать время выполнения алгоритма на модели (LogP, LogGP, PLogP, …)?**

Алгоритмы реализации MPI_Allgather

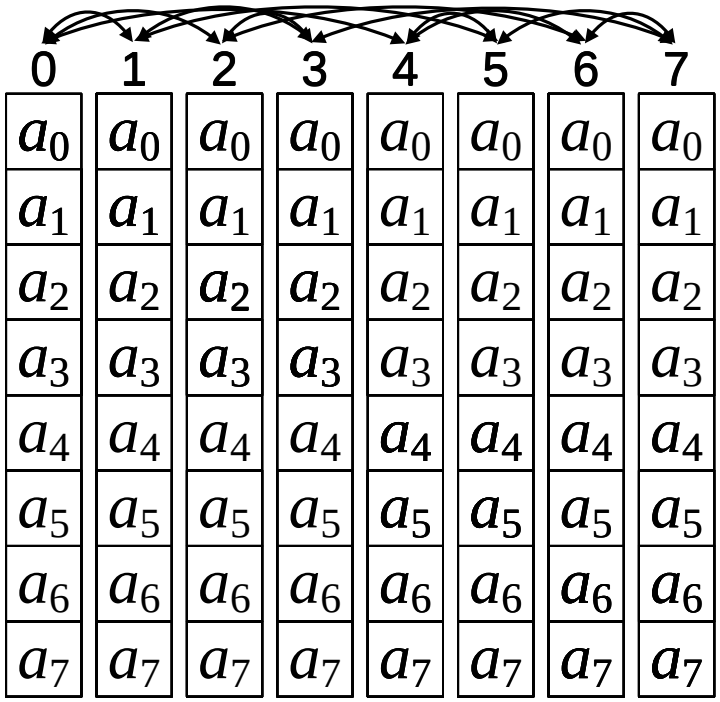
Кольцевой алгоритм (ring)



Каждая ветвь выполняет 2(*n* – 1) обменов (Send/Recv)

Алгоритмы реализации MPI_Allgather

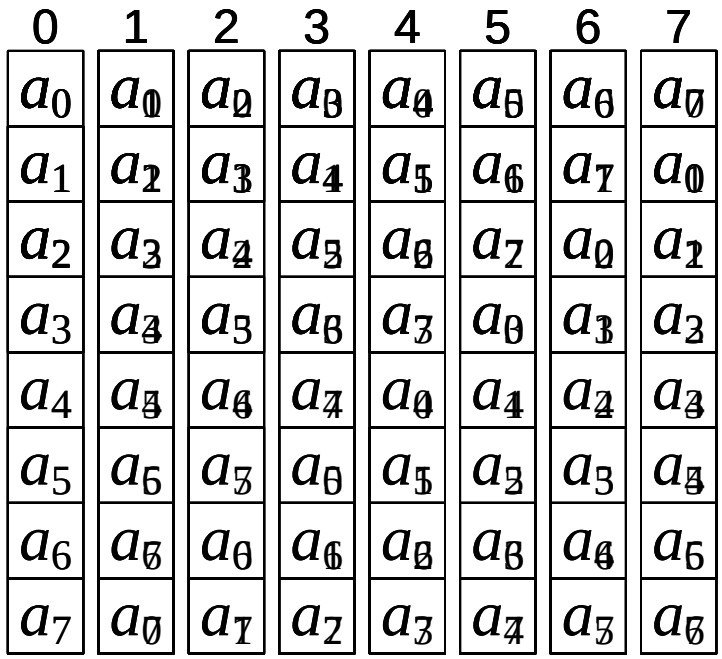
Алгоритм рекурсивного сдваивания (recursive doubling)



Количество обменов: $2\log_2 n$

Только для n равного степени двойки

Алгоритм Дж. Брука (J. Bruck et al., 1997)



Количество обменов: $2\lceil \log_2 n \rceil$

На шаге k ветвь i взаимодействует с ветвями $(i - 2^k + n) \bmod n$ и $(i + 2^k) \bmod n$

На каждом шаге размер передаваемого блока удваивается: m, $2m$, $4m$

Таймер

```
double MPI_Wtime()
```

```
int main(int argc, char **argv)
{
    double t = 0.0

    t -= MPI_Wtime();
    /* Code */
    t += MPI_Wtime();

    printf("Elapsed time: %.6f seconds\n", t);
}
```

Неблокирующие коллективные операции

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)
```

MPI_Ibcast

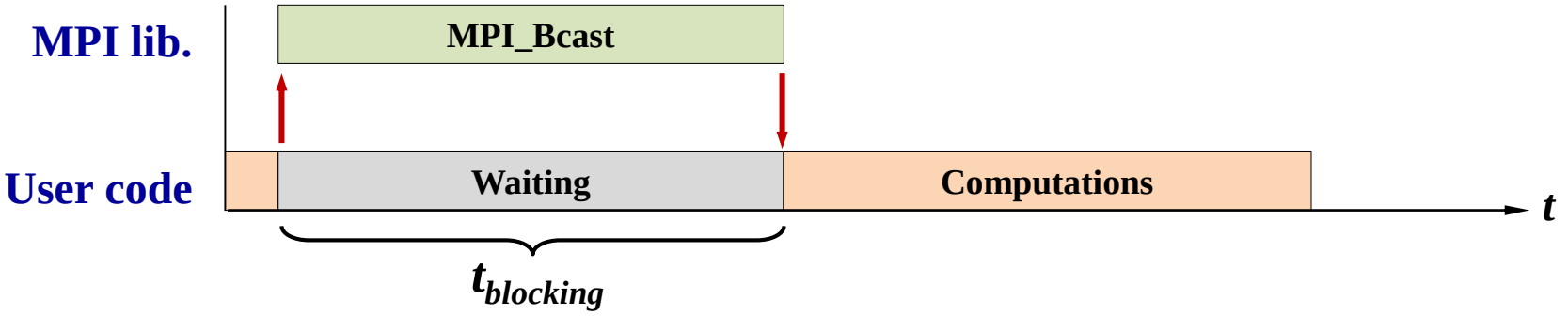
MPI 3.0

```
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);
```

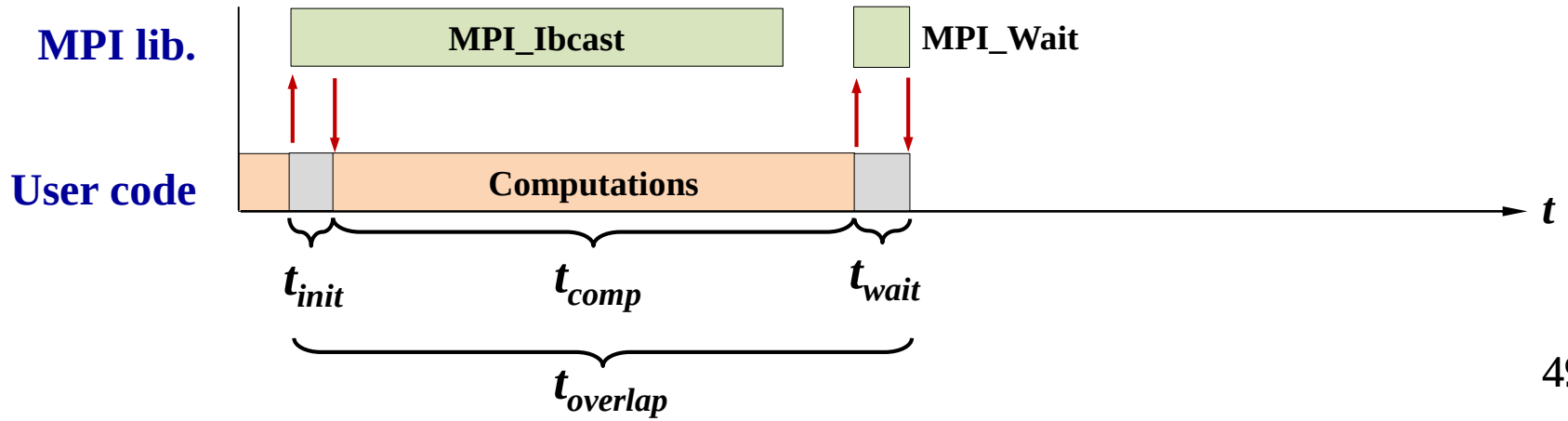
Non-blocking MPI Collectives

MPI 3.0

Блокирующие коллективные операции (Blocking MPI Collectives)



Неблокирующие коллективные операции (Non-blocking collectives)



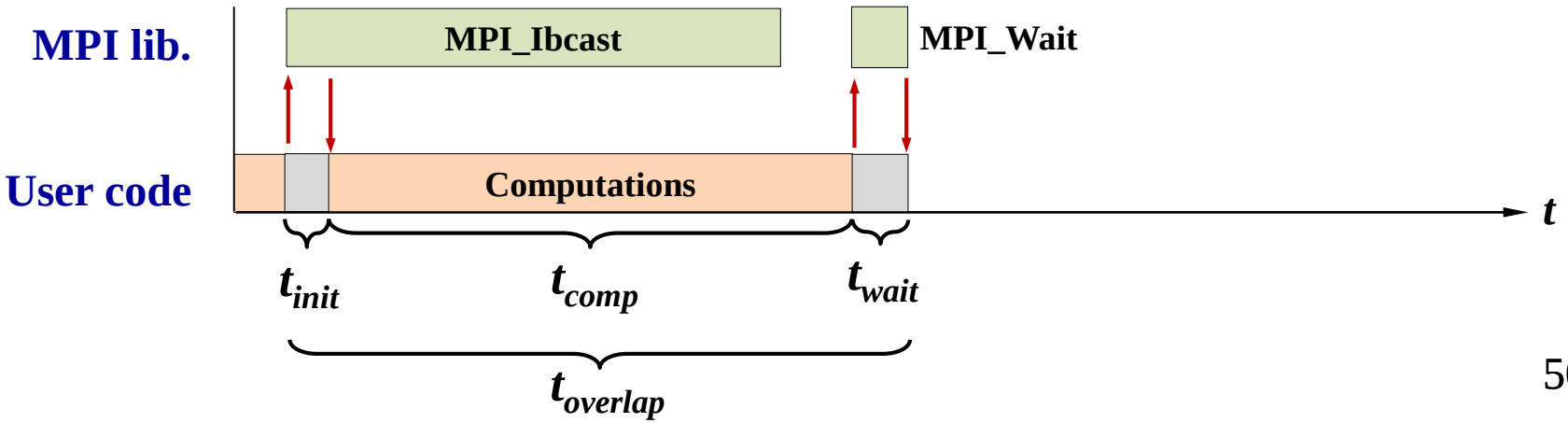
Non-blocking MPI Collectives

MPI 3.0

Коэффициент α совмещения вычисления и обменов информацией

$$\alpha = 1 - \frac{t_{overlap} - t_{comp}}{t_{blocking}}$$

Неблокирующие коллективные операции (Non-blocking collectives)



Неблокирующие коллективные операции

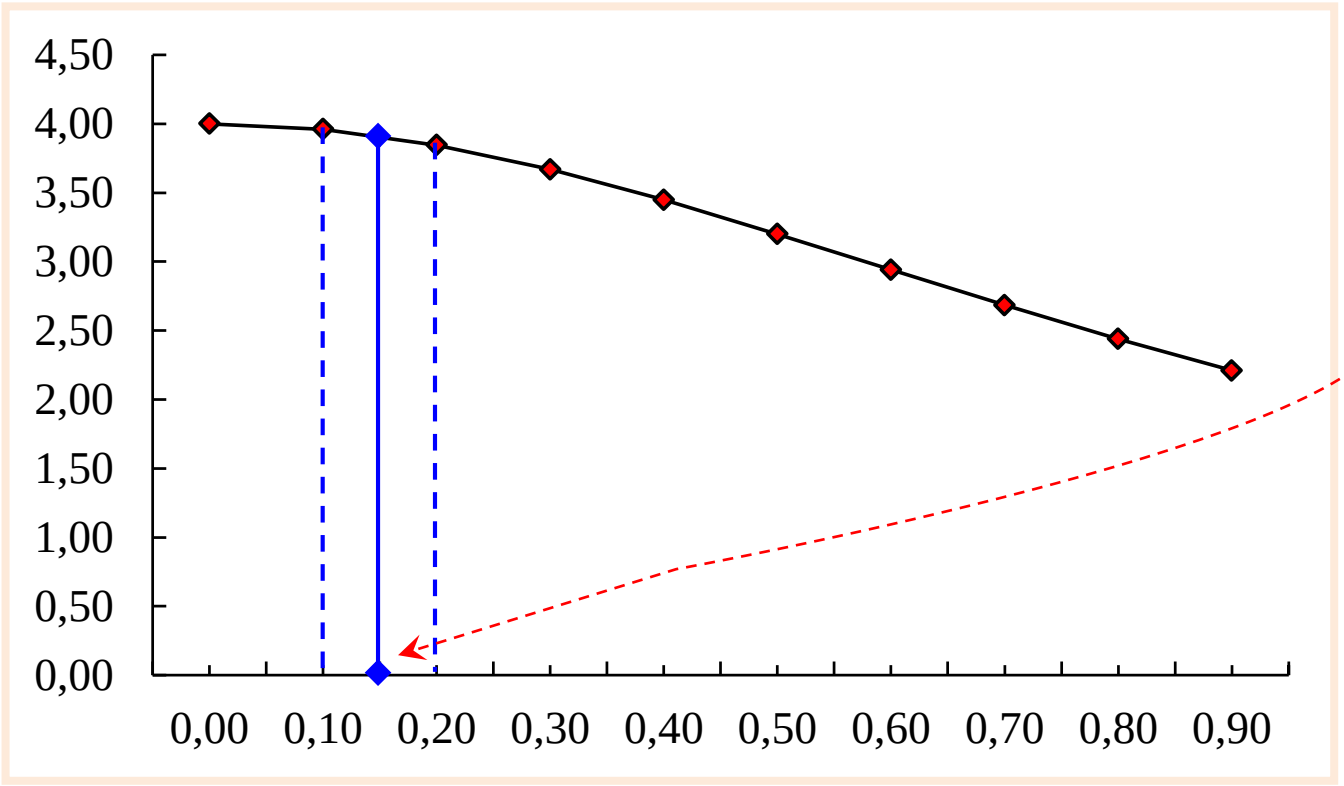
- При вызове неблокирующей коллективной операции создается **расписание выполнения обменов** (collective schedule)
- **Progress engine** – механизм, который в фоновом режиме реализует обмены по созданному расписанию – как правило обмены выполняются при вызове MPI_Test (в противном случае необходим дополнительный поток)

```
MPI_Request req;
MPI_Ibcast(buf, count, MPI_INT, 0, MPI_COMM_WORLD, &req);
while (!flag) {
    // Вычисления...
    // Проверяем состояние и продвигаем обмены по расписанию
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
}
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Вычисление числа π

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

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



$$h = \frac{1}{n}$$

Вычисление числа π (sequential version)

```
int main()
{
    int i, nsteps = 1000000;
    double step = 1.0 / (double)nsteps;
    double pi, x, sum = 0.0;

    for (i = 1; i <= nsteps; i++) {
        x = step * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x * x);
    }
    pi = step * sum;

    printf("PI = %.16f\n", pi);
    return 0;
}
```

Вычисление числа π (parallel version)

```
int main(int argc, char *argv[])
{
    int i, rank, commsize, nsteps = 1000000;
    double step, local_pi, pi, x, sum = 0.0;

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

    // Передаем все количество шагов интегрирования
    MPI_Bcast(&nsteps, 1, MPI_INT, 0,
              MPI_COMM_WORLD);

    step = 1.0 / (double)nsteps;
```

Вычисление числа π (parallel code)

```
    // Процесс rank получает точки x:
    // rank + 1, rank + 1 + p, rank + 1 + 2p, …
    for (i = rank + 1; i <= nsteps; i += commsize) {
         x = step * ((double)i - 0.5);
         sum += 4.0 / (1.0 + x * x);
     }
     local_pi = step * sum;

     // Формируем результирующую сумму
     MPI_Reduce(&local_pi, &pi, 1, MPI_DOUBLE,
                MPI_SUM, 0, MPI_COMM_WORLD);

     if (rank == 0)
         printf("PI = %.16f\n", pi);

     MPI_Finalize();
     return 0;
}
```

Эффективность параллельных программ

- **Коэффициент *S* ускорения параллельной программы (Speedup)**

$$S = \frac{T_1}{T_n},$$

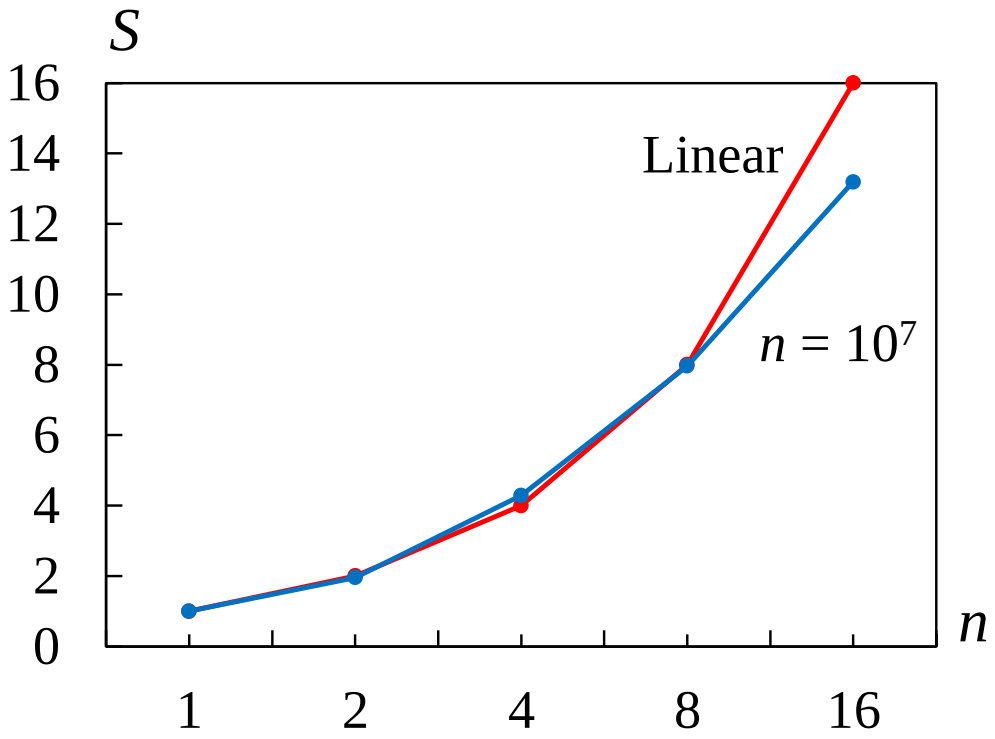
где:

- T_1 – время выполнения последовательной программы,
- T_n – время выполнения параллельной программы на n процессорных ядрах (n процессов)

- **Коэффициент *E* эффективности параллельной программы**

$$E = \frac{S}{n} = \frac{T_1}{nT_n}$$

Эффективность программы π

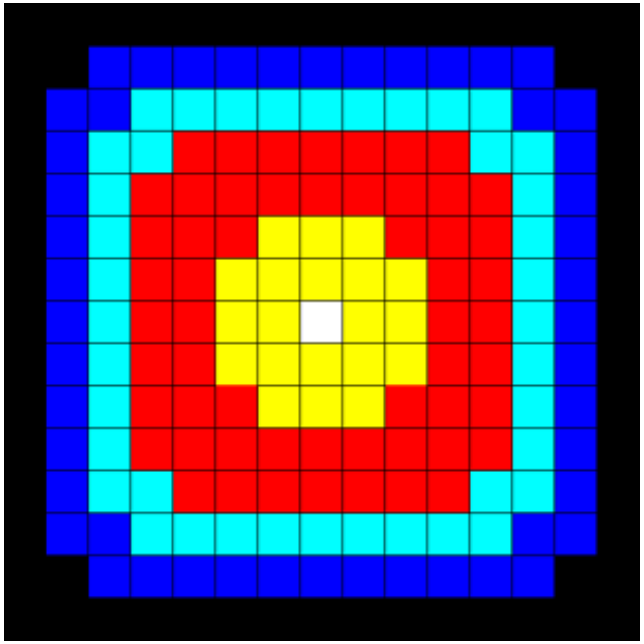


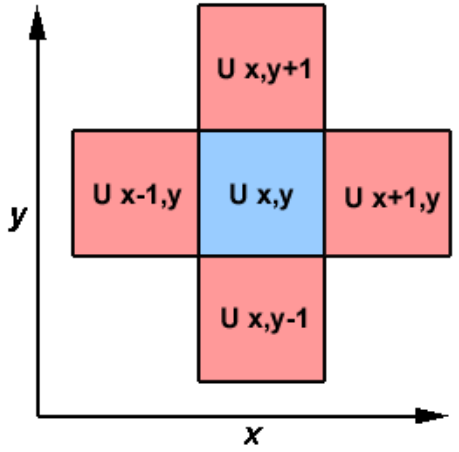
Зависимость коэффициента ускорения *S*
от числа *n* процессорных ядер

(вычислительные узлы: 2 x Intel Xeon E5420, коммуникационная сеть Gigabit Ethernet; MPICH2 1.4.1p1, GCC 4.4.6;
на каждом вычислительном узле 1 процесс)

Heat 2D

- Уравнение теплопроводности описывает изменение температуры в заданной области с течением времени
- Приближенное решение можно найти методом конечных разностей
- Область покрывается сеткой
- Производные аппроксимируются конечными разностями
- Известна температура на границе области в начальный момент времени



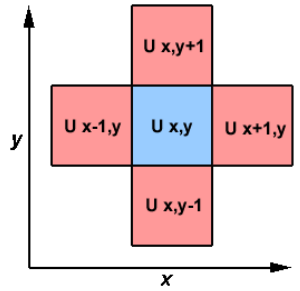


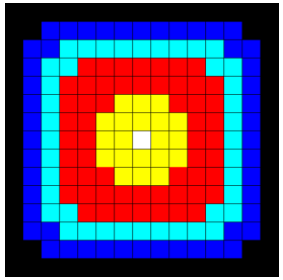
[*] Blaise Barney. Introduction to Parallel Computing (LLNL)

Heat 2D (serial code)

- Температура хранится в двумерном массиве – расчетная область

```
do iy = 2, ny - 1
  do ix = 2, nx - 1
    u2(ix, iy) = u1(ix, iy) +
                 cx * (u1(ix + 1, iy) +
                 u1(ix - 1, iy) – 2.0 * u1(ix, iy)) +
                 cy * (u1(ix, iy + 1) +
                 u1(ix, iy - 1) – 2.0 * u1(ix, iy))
  end do
end do
```

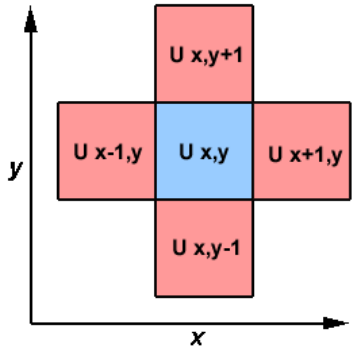


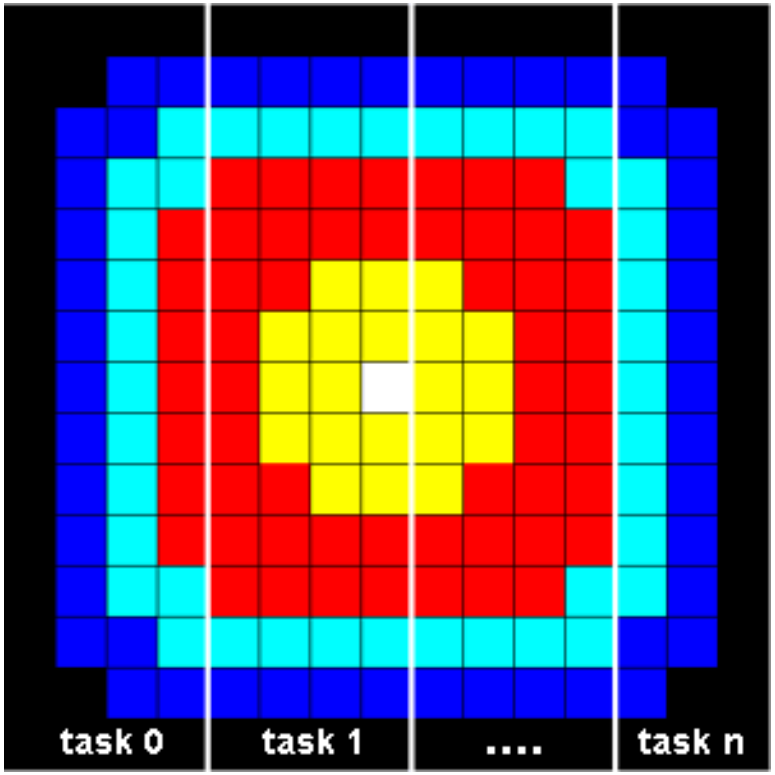


Heat 2D (parallel version)

- Каждый процесс будет обрабатывать свою часть области

```
for t = 1 to nsteps
  1. Update time
  2. Send neighbors my border
  3. Receive from neighbors
  4. Update my cells
end for
```

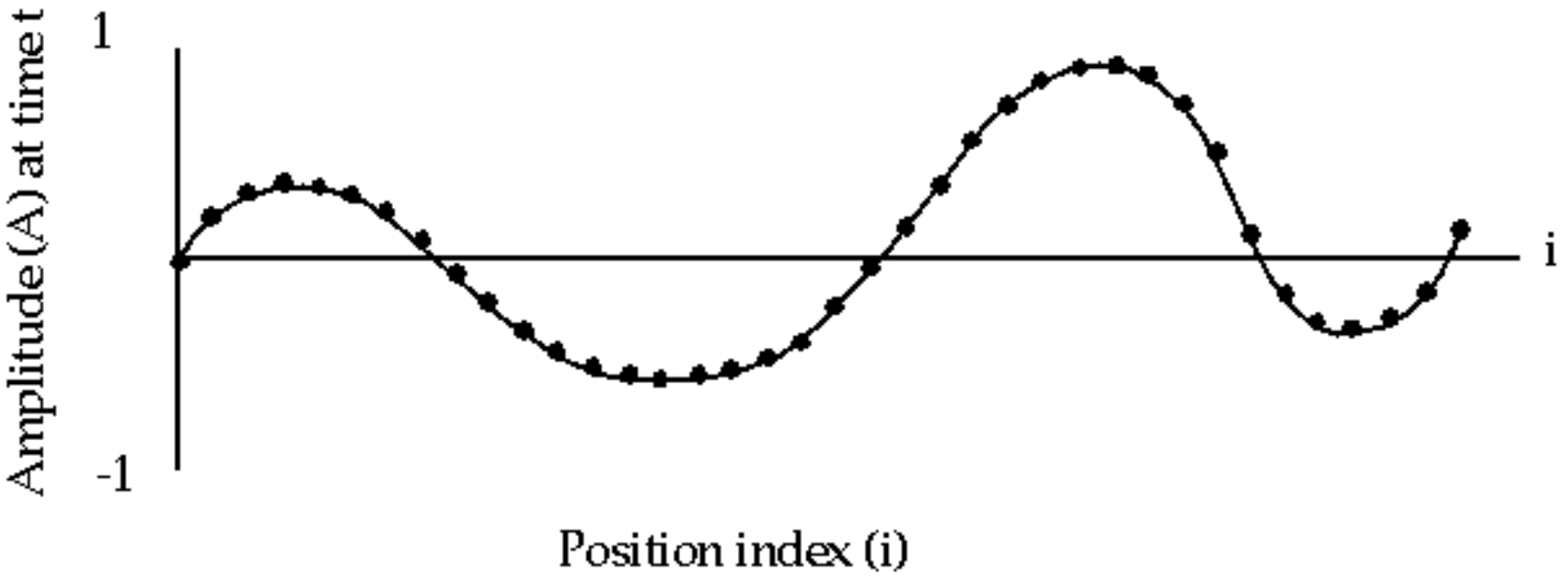




Расчетная область разбита на вертикальные полосы – массив распределен

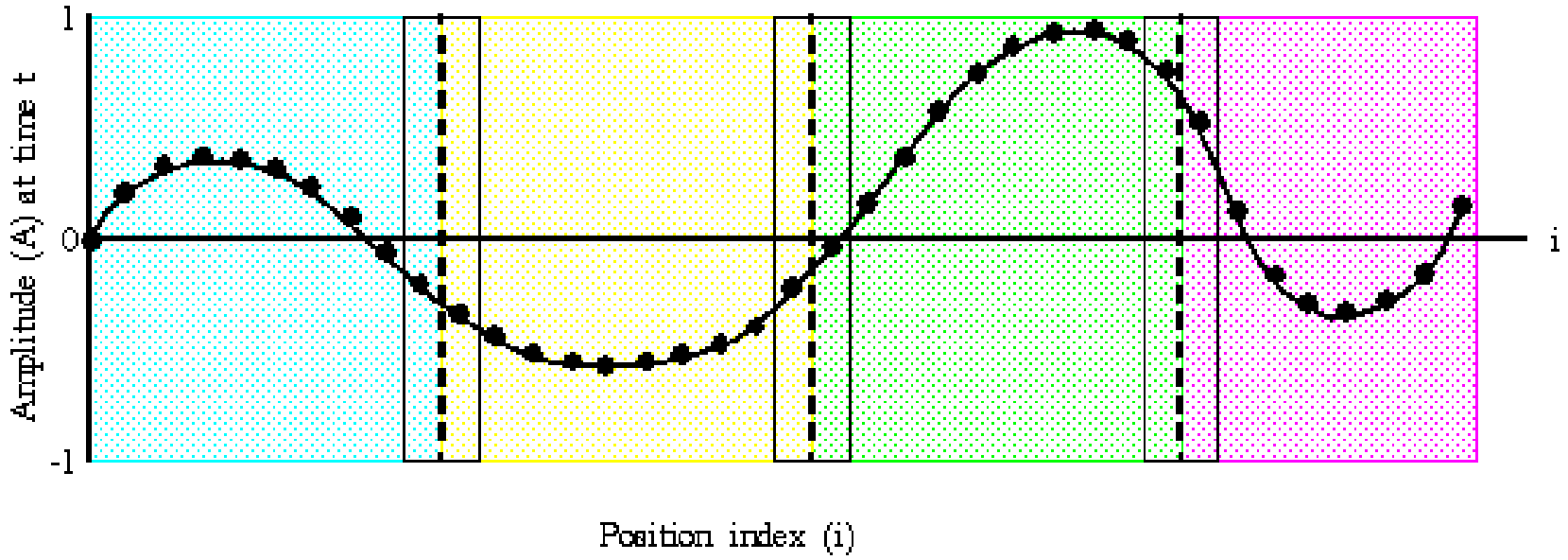
[*] Blaise Barney. Introduction to Parallel Computing (LLNL)

Wave Equation




```
A(i, t + 1) = 2.0 * A(i, t) - A(i, t - 1) +
              c * (A(i - 1, t) - (2.0 * A(i, t) +
              A(i + 1, t))
```

Wave Equation (data decomposition)

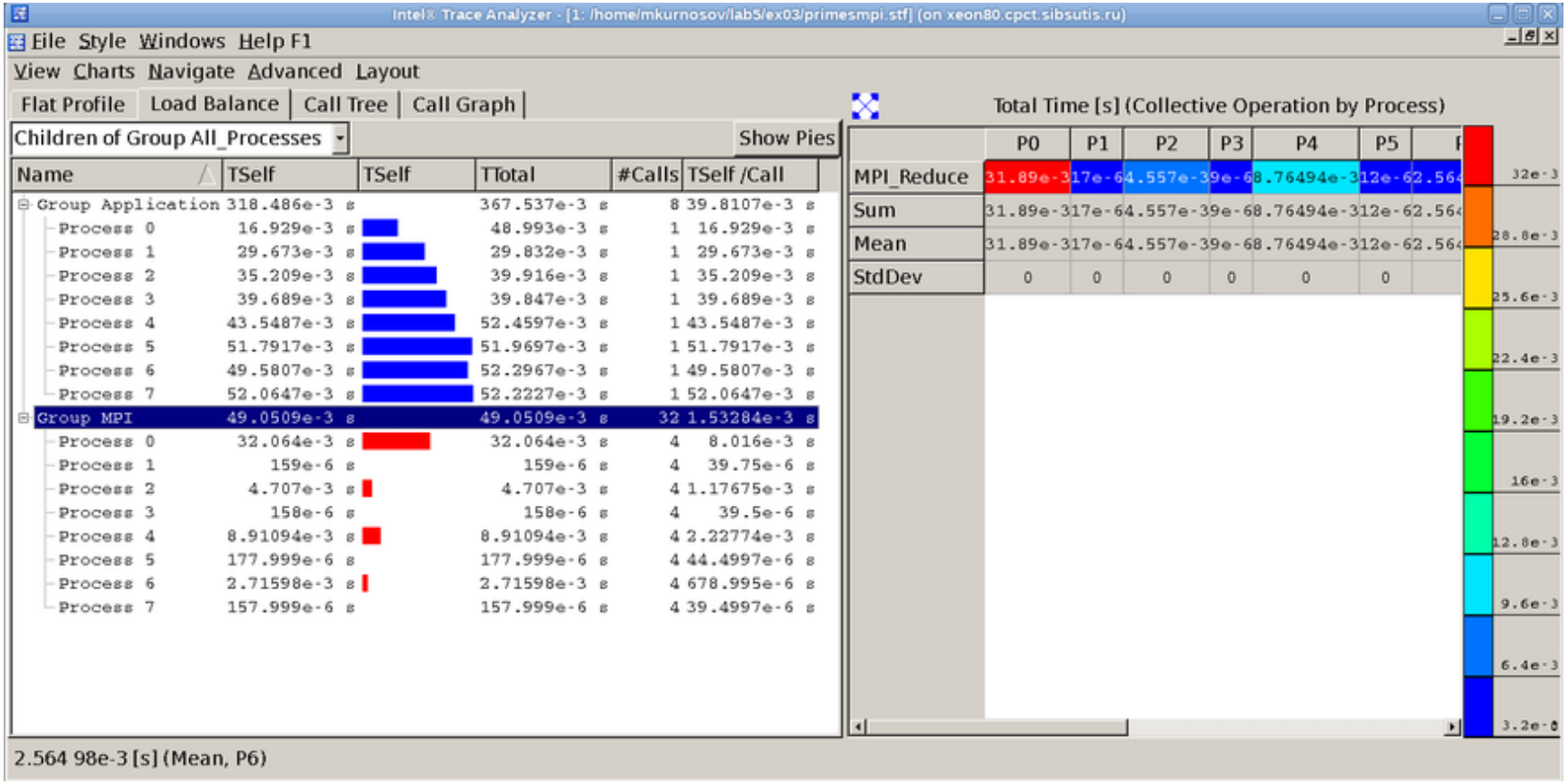



```
A(i, t + 1) = 2.0 * A(i, t) - A(i, t - 1) +
              c * (A(i - 1, t) - (2.0 * A(i, t) +
              A(i + 1, t))
```

Расчетная область разбита на вертикальные полосы –
массив распределен (одномерное разбиение)

[*] Blaise Barney. Introduction to Parallel Computing (LLNL)

Intel Trace Analyzer and Collector



- **Intel Trace Analyzer & Collector** – профилировщик для MPI-программ, показывает timeline с обменами между процессами, статистику по переданным сообщениям; позволяет находить дисбаланс загрузки процессов

Профилировщики MPI

- Intel Trace Analyzer & Collector
- mpiP
- VamirTrace + Vampir
- Tau
- Allinea MAP
- MPICH2 MPE (Jumpshot)
- Oracle Solaris Studio (collect)

Интерфейс профилирования PMPI

- Каждая функция MPI имеет 2 версии:
  - MPI_Function
  - PMPI_Function
- Функции с именами MPI_Function можно перехватывать – подменять функциями врапперами (wrapper) с такими же именами
- Во врапперах осуществляется регистрация времени выполнения, собирается статистика по размерам сообщения и пр.
- Из враппера вызывается оригинальная MPI-функция PMPI_Function

sendprof.c

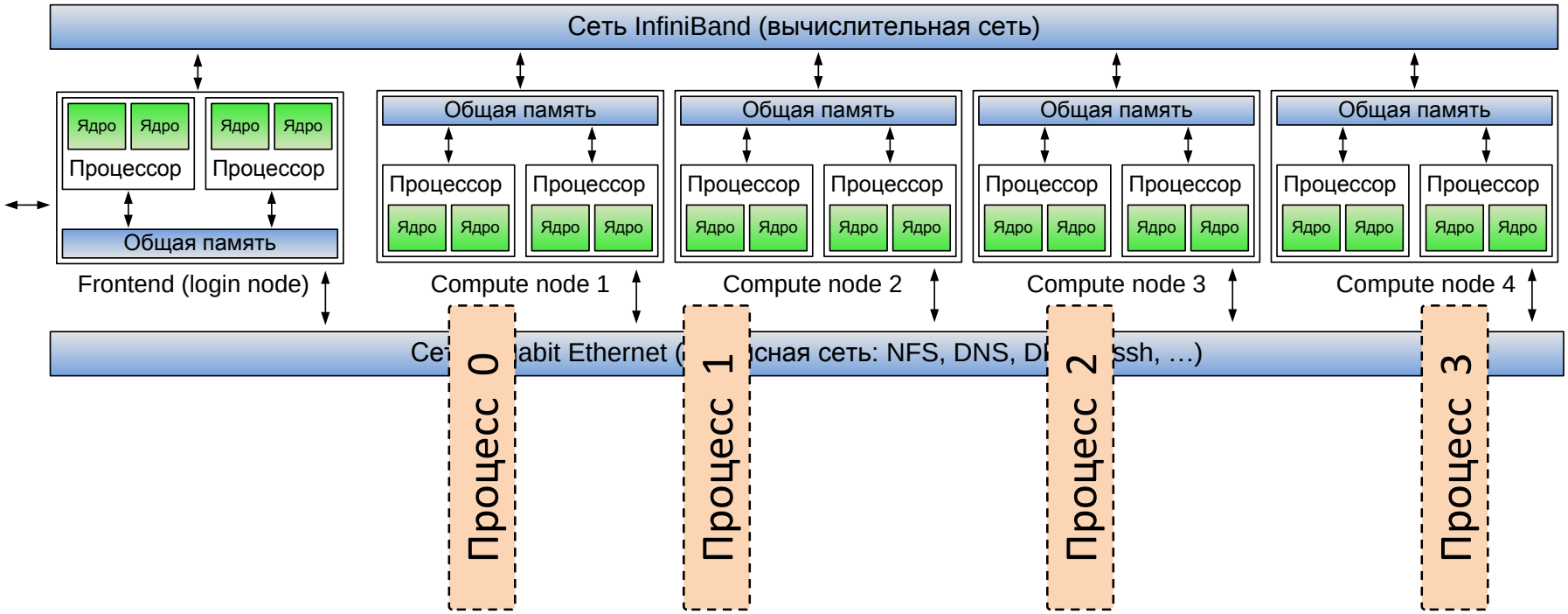
```
static int totalBytes = 0;
static double totalTime = 0.0;

// Wrapper for Send
int MPI_Send(const void* buffer, int count,
             MPI_Datatype datatype, int dest, int tag,
             MPI_Comm comm)
{
    double tstart = MPI_Wtime();
    int extent;
    // Call original (real) send
    int rc = PMPI_Send(buffer, count, datatype, dest,
                       tag, comm);
    totalTime += MPI_Wtime() - tstart;
    MPI_Type_size(datatype, &extent);       /* Message size */
    totalBytes += count * extent;
    return rc;
}
```

Использование sendprof.c

```
$ mpicc –c –o sendprof.o ./sendprof.c
$ ar rcs libsendprof.a sendprof.o
$ mpicc –o mpiprog ./mpiprog.c \
        –L. -lsendprof
```

- **Какие функции перехватывать?**
- **MPI_Initialize(), MPI_Init_threads()** – инициализируем профилировщик
- **MPI_Finalize()** – вывод отчета
- **MPI_Pcontrol()** – связь профилировщика с пользовательской программой
- Если необходимо профилировать Fortran-программы, то имена врапперов должны соответствовать правилам Fortran (нет указателей, преобразование констант Fortran -> C, …)

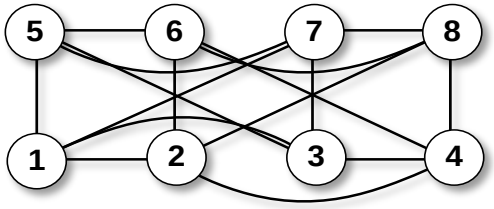
MPI Process mapping



- Время выполнения обменов зависит от того как процессы размещены по процессорным ядрами (process mapping/pining, task mapping, task allocation, task assignment)
- Распределение процессов задается при старте программы (mpiexec) или при формировании виртуальной топологии (MPI Virtual topolgy)

Пример вложения программы в ВС

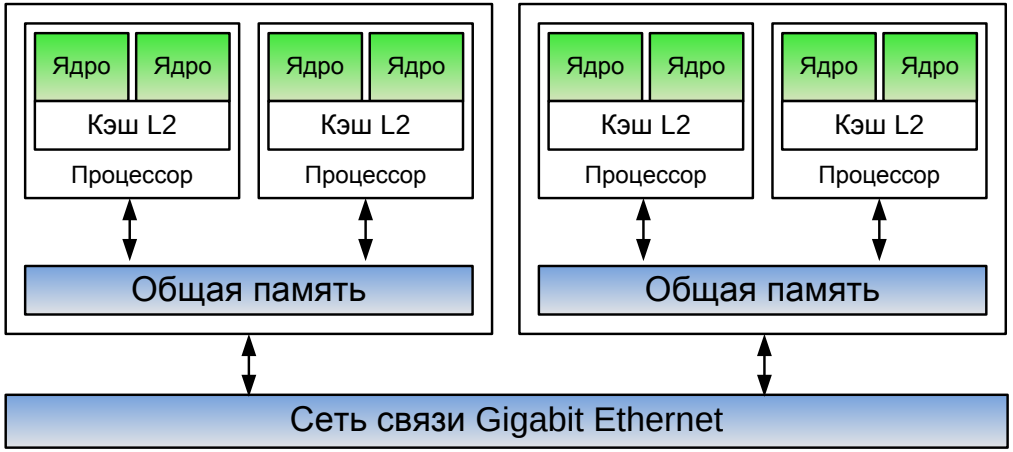
Вложение теста High Performance Linpack в подсистему:



Граф программы

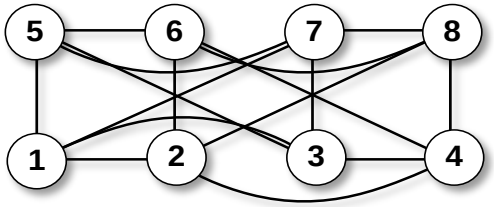
High Performance Linpack (HPL)

Вычислительный кластер с иерархической структурой (2 узла по 2 Intel Xeon 5150)



Пример вложения программы в ВС

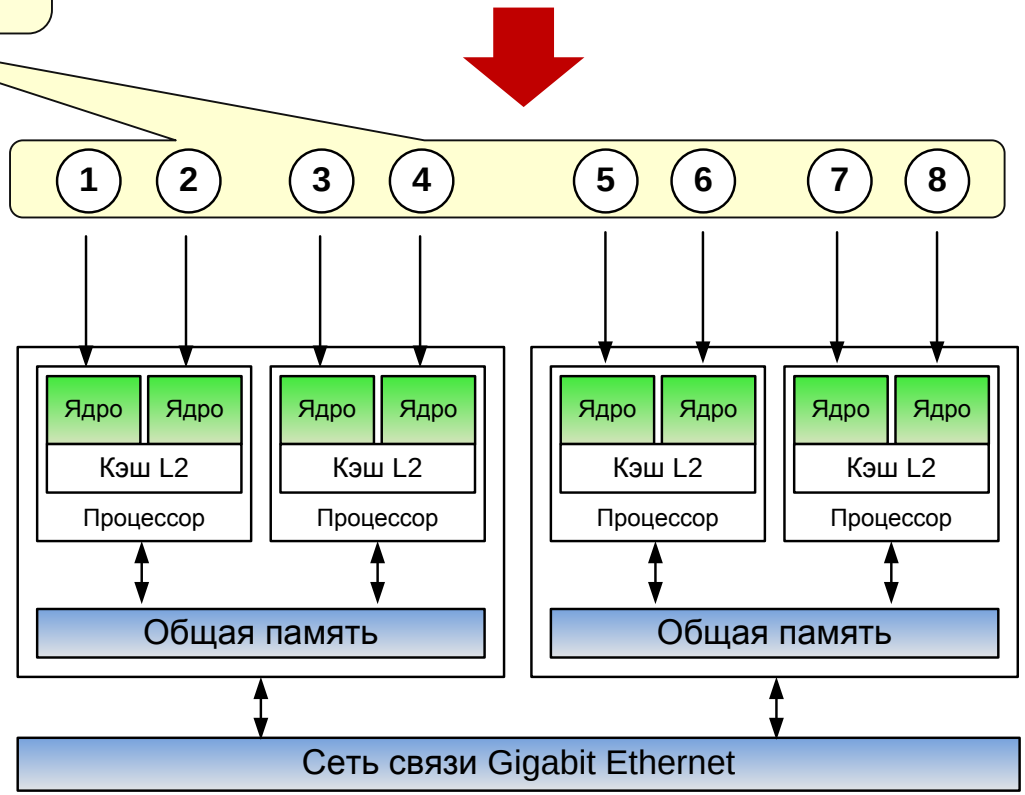
Вложение теста High Performance Linpack в подсистему:



Граф программы

High Performance Linpack (HPL)

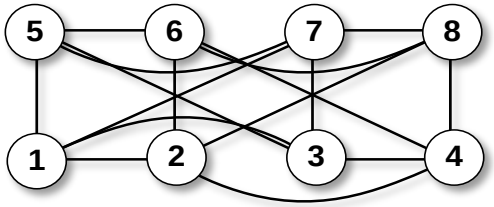
стандартными MPI-утилитами (mpiexec) – время выполнения **118 сек.** (44 GFLOPS)



Вычислительный кластер с иерархической структурой (2 узла по 2 Intel Xeon 5150)

Пример вложения программы в ВС

Вложение теста High Performance Linpack в подсистему:



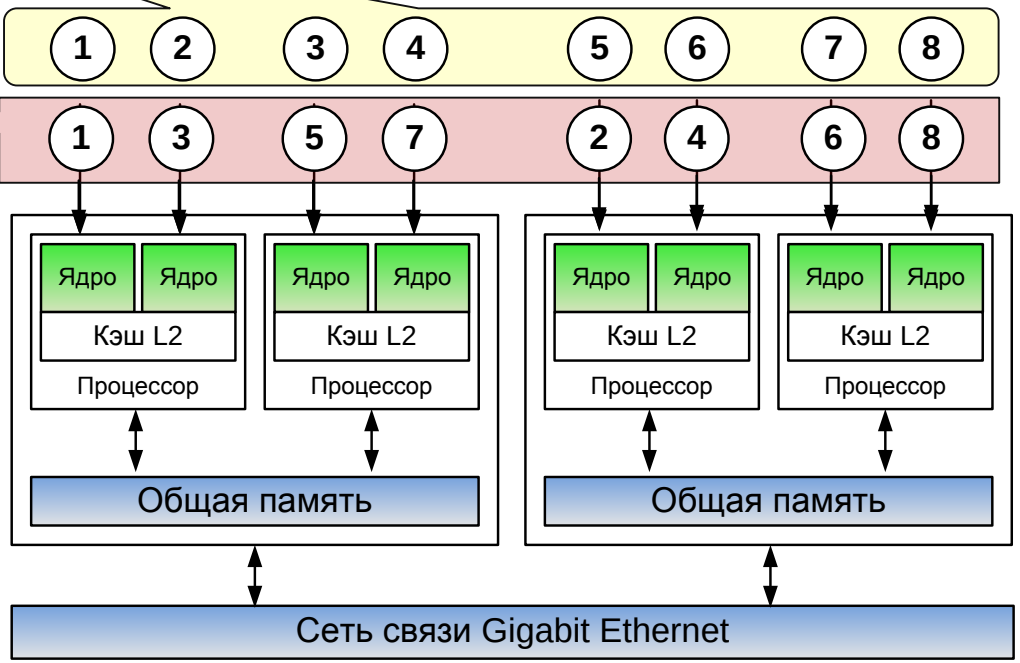
Граф программы

High Performance Linpack (HPL)

стандартными MPI-утилитами (mpiexec) – время выполнения **118 сек.** (44 GFLOPS)

с учетом структуры ВС – время выполнения **100 сек.** (53 GFLOPS)

Вычислительный кластер с иерархической структурой (2 узла по 2 Intel Xeon 5150)

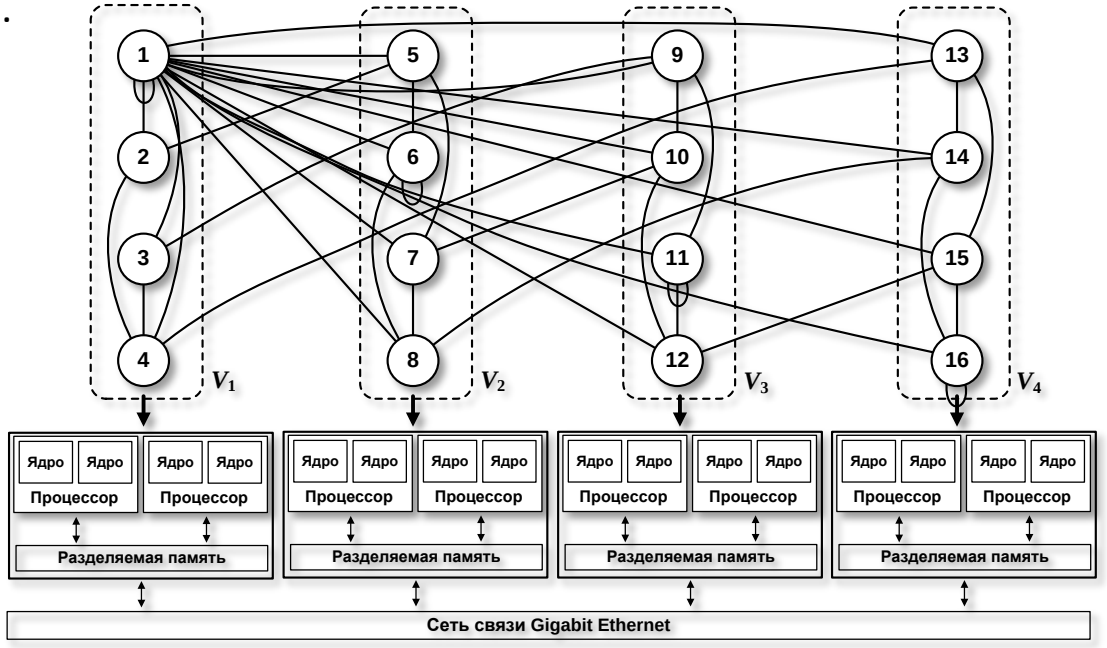


Иерархический метод вложения программ

Метод вложения на основе многоуровневых (multilevel) алгоритмов разбиения графов *G* = (*V*, *E*) параллельных программ

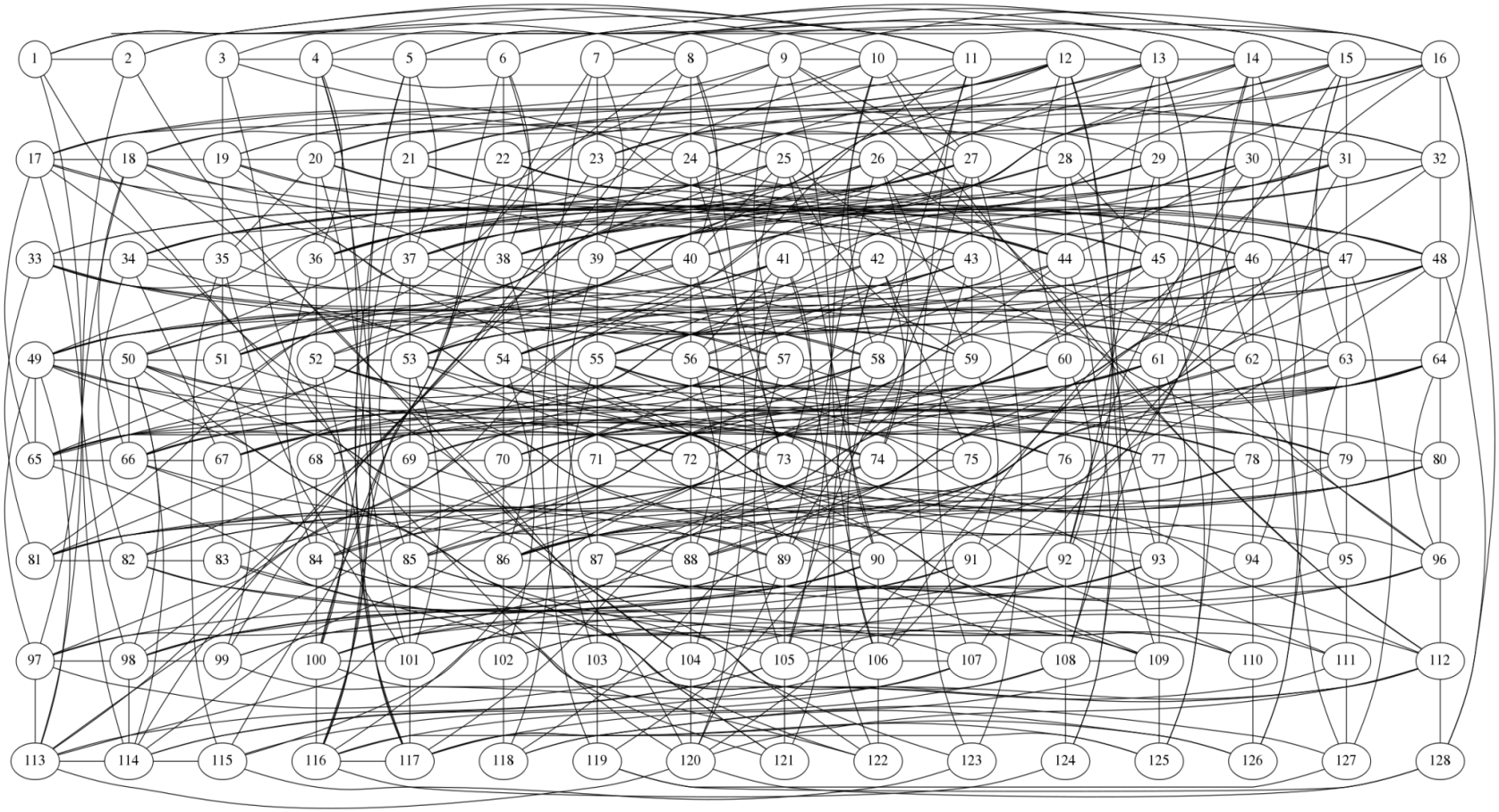
1. Граф *G* разбивается на *k* подмножеств; *k* – количество узлов, составляющих ВС. В каждое из подмножеств включаются ветви, интенсивно обменивающиеся данными.

2. Параллельные ветви из *i*-го подмножества распределяются по ядрам *i*-го вычислительного узла, $i \in \{1, 2, .., k\}$.



Вложение MPI-программы Conjugate Gradient (CG) из пакета NAS Parallel Benchmarks, реализующей решение системы линейных алгебраических уравнений методом сопряженных градиентов в вычислительный кластер

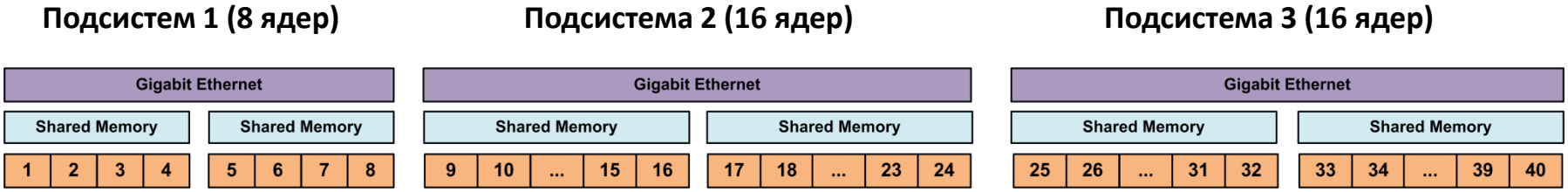
Иерархический метод вложения программ

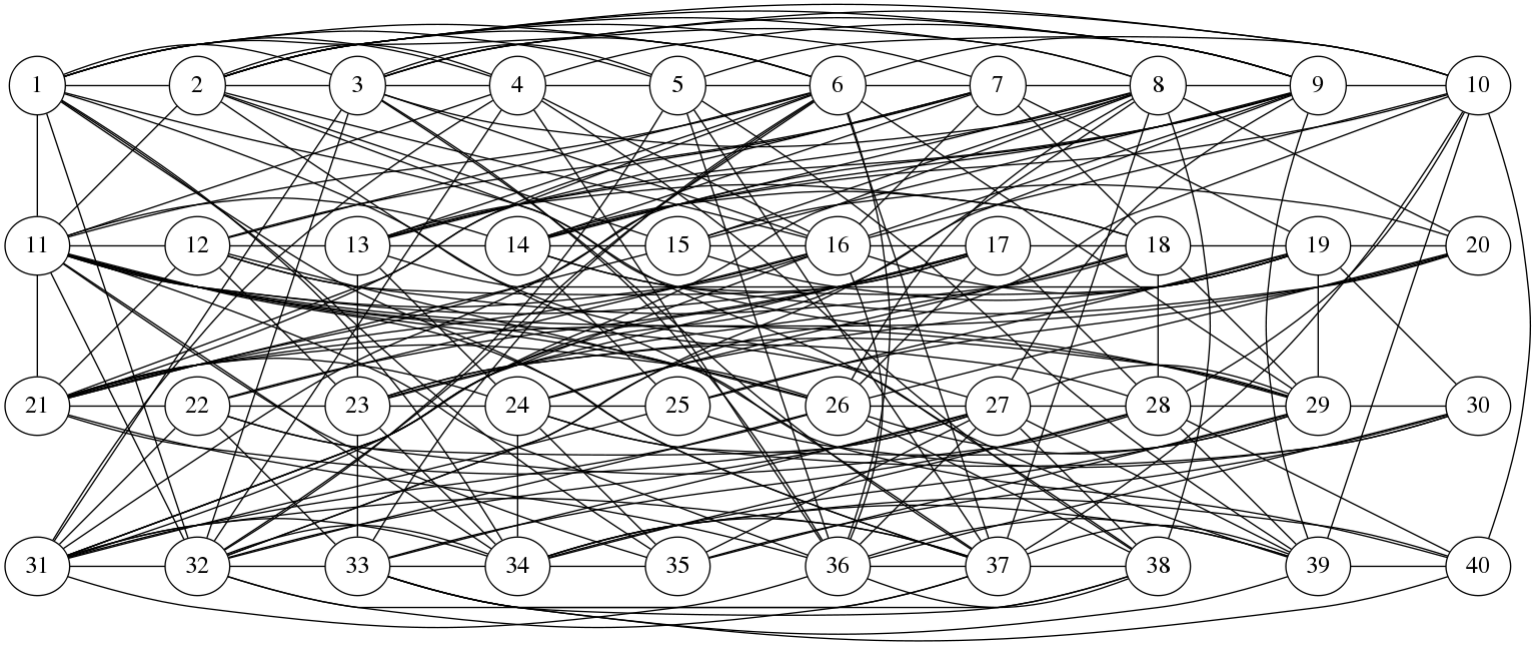


The Parallel Ocean Program, NP = 128

Моделирование процессов в мировом океане

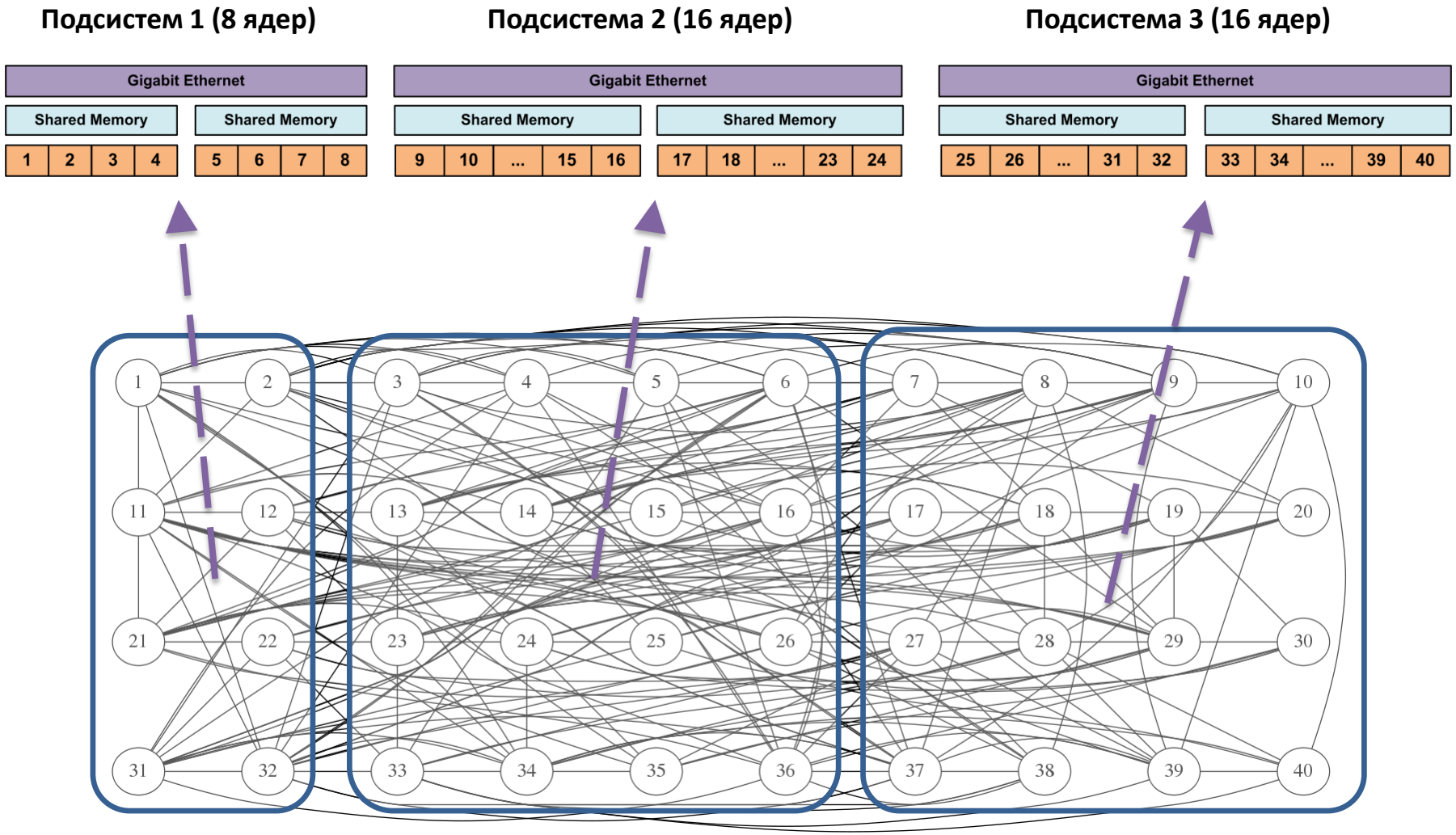
Иерархический метод вложения программ





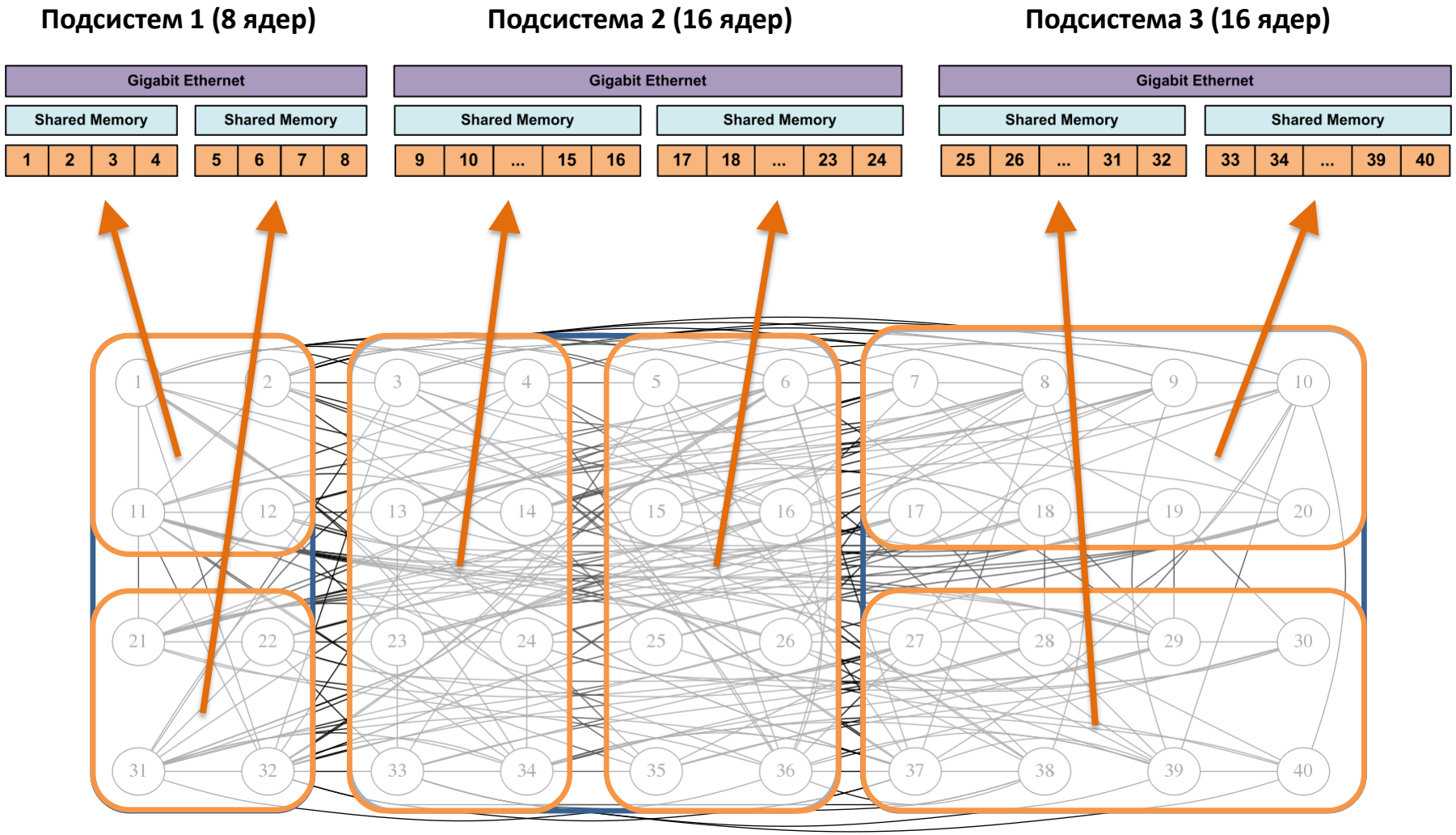
The Parallel Ocean Program (POP), NP = 40

Иерархический метод вложения программ



The Parallel Ocean Program (POP), NP = 40

Иерархический метод вложения программ



The Parallel Ocean Program (POP), NP = 40