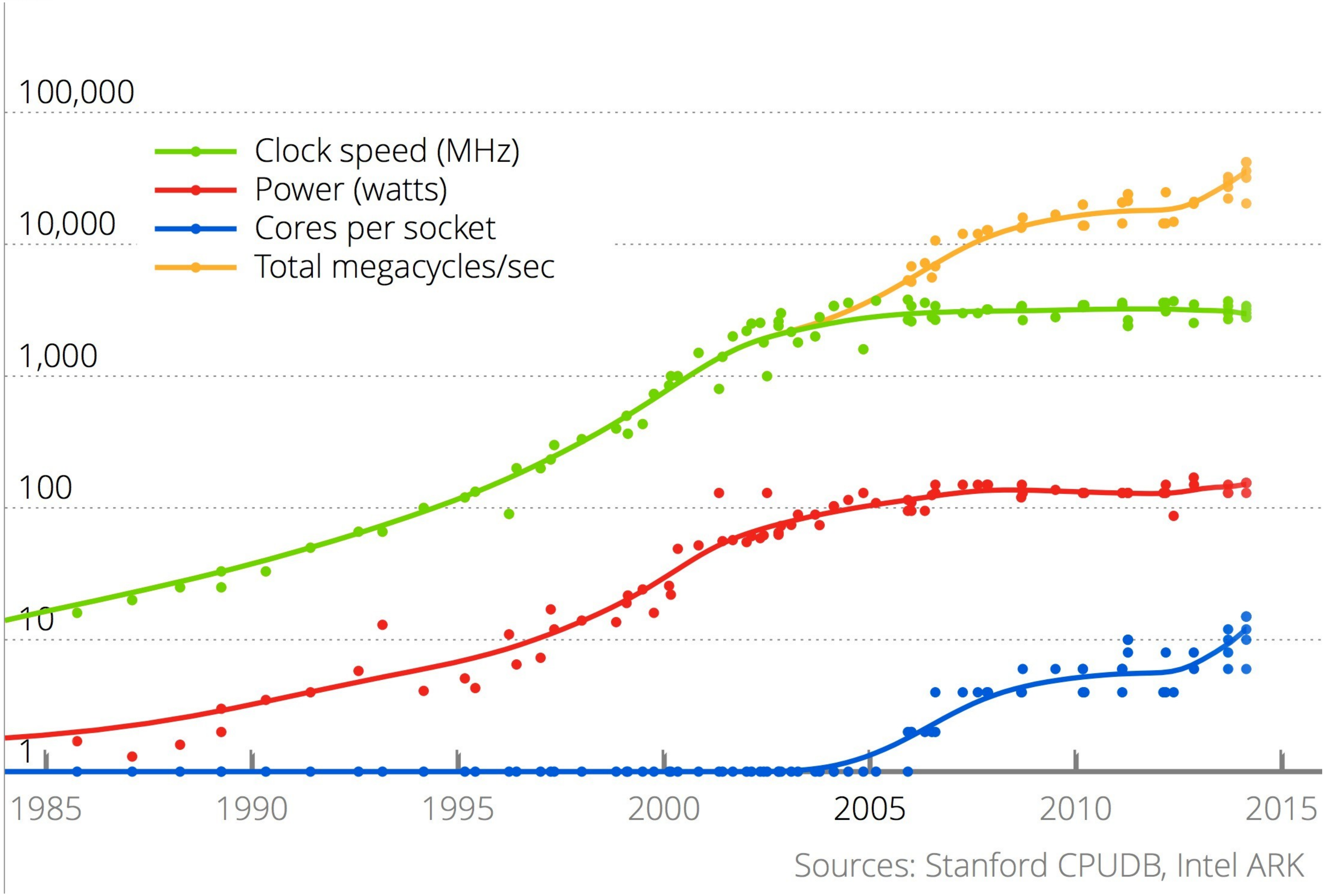
TIMO SCHNEIDER <TIMOS@INF.ETHZ.CH>

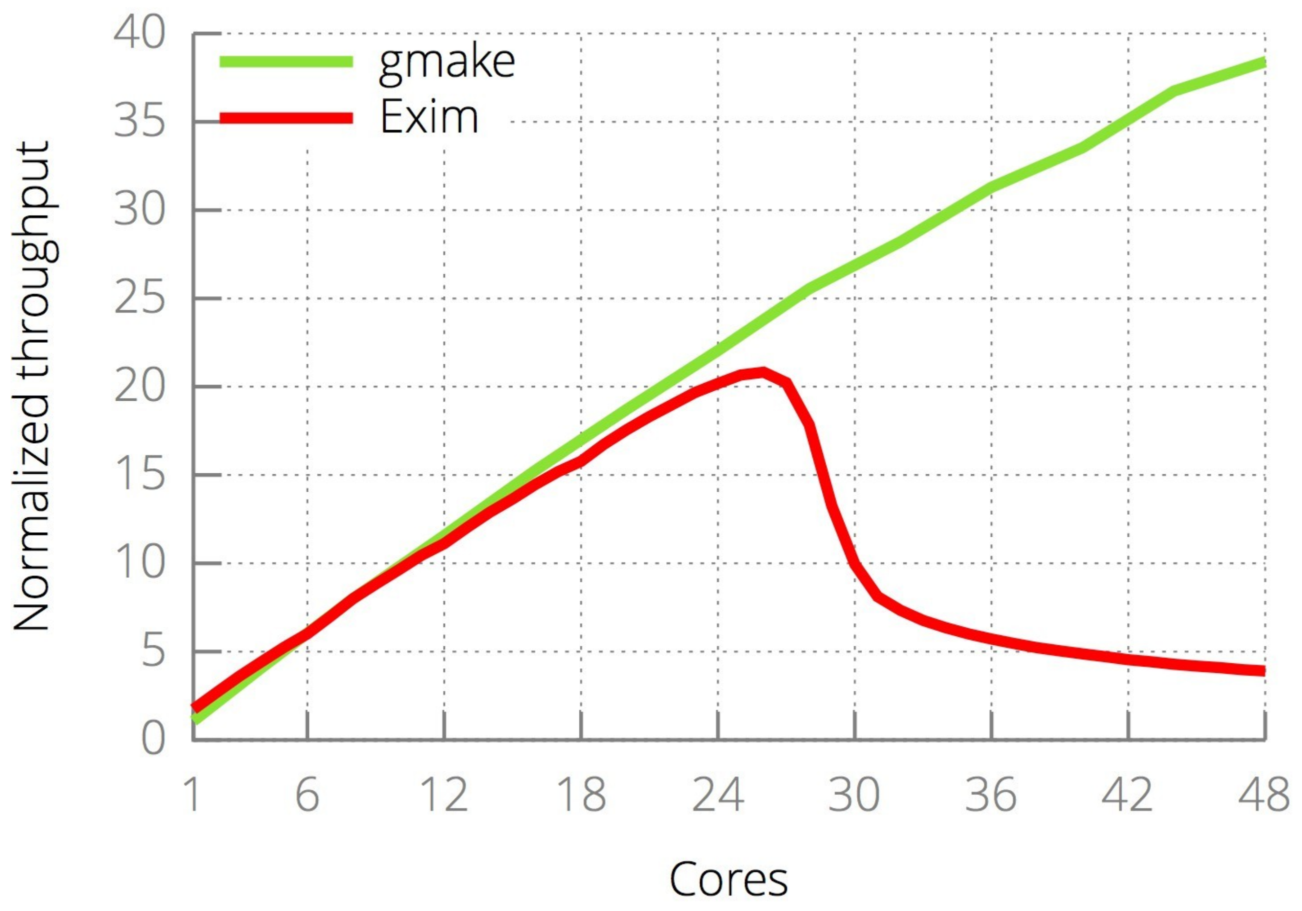
MPI Tutorial

Design of Parallel and High-Performance Computing

Slides credits: Pavan Balaji, Torsten Hoefler
https://htor.inf.ethz.ch/teaching/mpi_tutorials/ppopp13/2013-02-24-ppopp-mpi-basic.pdf



Sources: Stanford CPUDB, Intel ARK

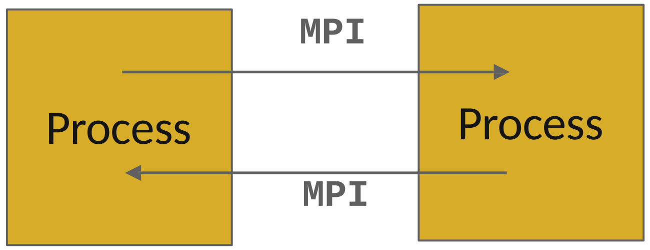

gmake
Exim
Normalized throughput
0
5
10
15
20
25
30
35
40
Cores
1
6
12
18
24
30
36
42
48

Sample Parallel Programming Models

- Shared Memory Programming
  - Processes share memory address space (threads model)
  - Application ensures no data corruption (Lock/Unlock)
- Transparent Parallelization
  - Compiler works magic on sequential programs
- Directive-based Parallelization
  - Compiler needs help (e.g., OpenMP)
- Message Passing
  - Explicit communication between processes (like sending and receiving emails)

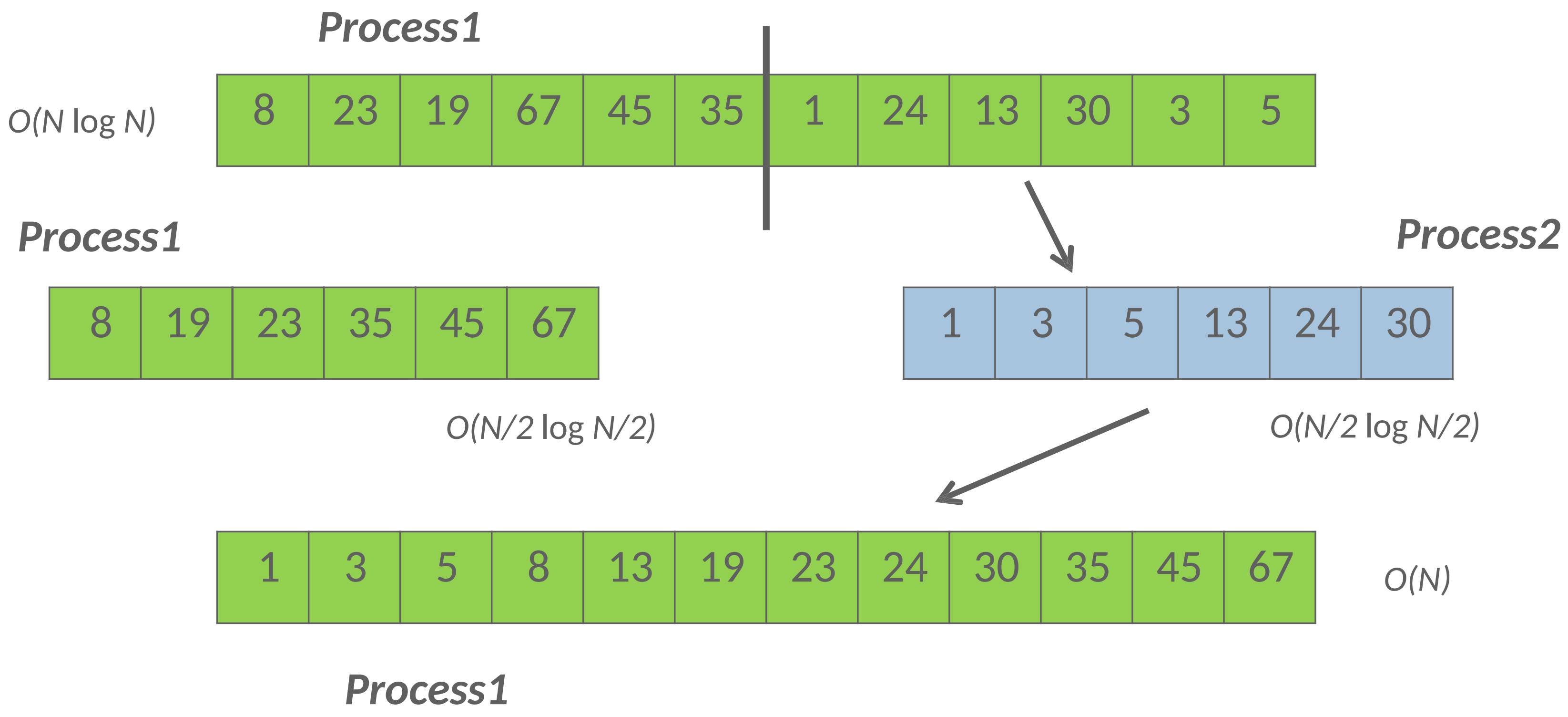
The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Inter-process communication consists of…
  - synchronization
  - movement of data from one process’s address space to another’s.



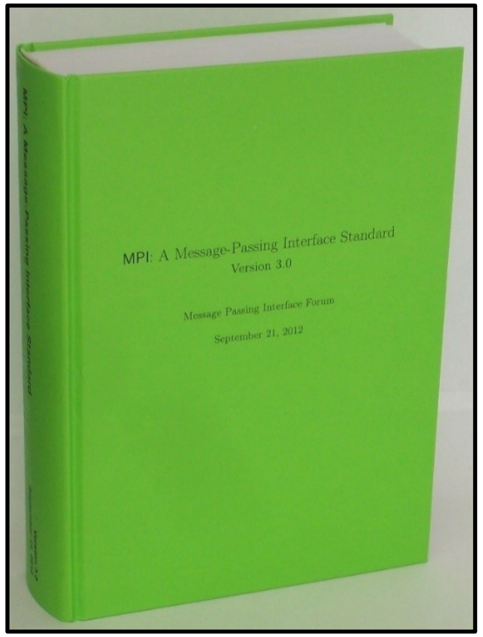
The Message-Passing Model (an example)

- Each process has to send/receive data to/from other processes
- Example: Sorting Integers



Standardizing Message-Passing Models with MPI

- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable)
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of message-passing issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The MPI Forum was a collection of vendors, portability writers and users that wanted to standardize all these efforts


MPI: A Message-Passing Interface Standard
Version 3.0
Message Passing Interface Forum
September 21, 2012

What is MPI?

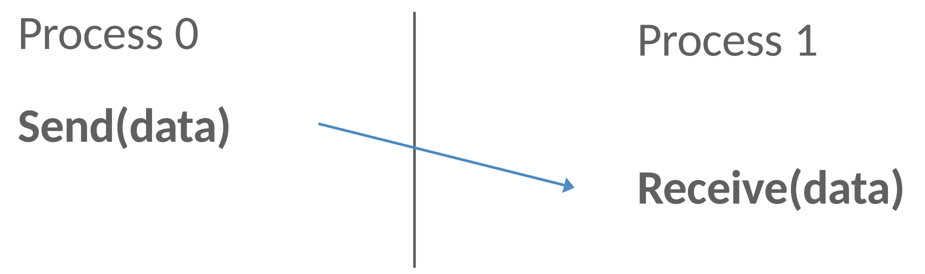
- MPI: Message Passing Interface
  - The MPI Forum organized in 1992 with broad participation by:
    - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
    - Portability library writers: PVM, p4
    - Users: application scientists and library writers
    - MPI-1 finished in 18 months
  - Incorporates the best ideas in a “standard” way
    - Each function takes fixed arguments
    - Each function has fixed semantics
      - Standardizes what the MPI implementation provides and what the application can and cannot expect
      - Each system can implement it differently as long as the semantics match
- MPI is not…
  - a language or compiler specification
  - a specific implementation or product

Reasons for Using MPI

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries
- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance
- **Functionality** – Rich set of features
- **Availability** - A variety of implementations are available, both vendor and public domain
  - MPICH/Open MPI are popular open-source and free implementations of MPI
  - Vendors and other collaborators take MPICH and add support for their systems
    - Intel MPI, IBM Blue Gene MPI, Cray MPI, Microsoft MPI, MVAPICH, MPICH-MX

MPI Basic Send/Receive

- Simple communication model



- Application needs to specify to the MPI implementation:
  1. How do you compile and run an MPI application?
  2. How will processes be identified?
  3. How will “data” be described?

Compiling and Running MPI applications (more details later)

- MPI is a library
  - Applications can be written in C, C++ or Fortran and appropriate calls to MPI can be added where required
- Compilation:
  - Regular applications:
    - `gcc test.c -o test`
  - MPI applications
    - `mpicc test.c -o test`
- Execution:
  - Regular applications
    - `./test`
  - MPI applications (running with 16 processes)
    - `mpiexec –np 16 ./test`

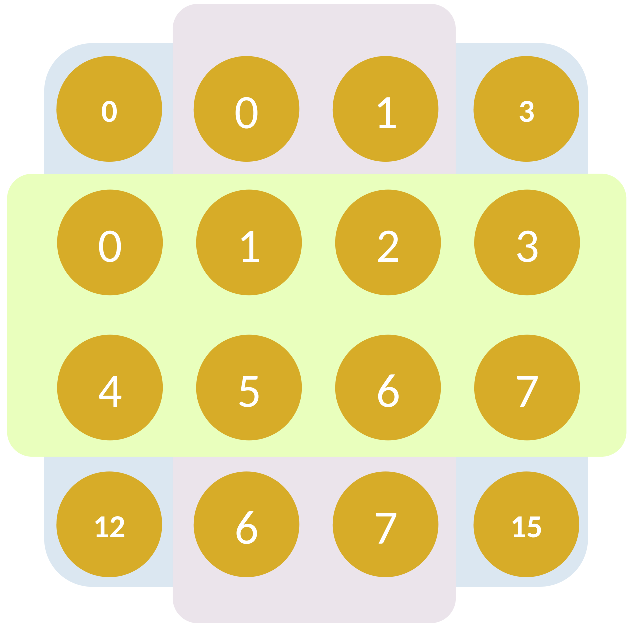
Process Identification

- MPI processes can be collected into groups
  - When an MPI application starts, the group of all processes is initially given a predefined name called **`MPI_COMM_WORLD`**
  - The same group can have many names, but simple programs do not have to worry about multiple names
- A process is identified by a unique number within each communicator, called *rank*
  - For two different communicators, the same process can have two different ranks: so the meaning of a “rank” is only defined when you specify the communicator

Communicators

```
mpiexec  -np  16  ./test
```

Communicators do not need to contain all processes in the system



When you start an MPI program, there is one predefined communicator **MPI_COMM_WORLD**

Every process in a communicator has an ID called as “rank”

Can make copies of this communicator (same group of processes, but different “aliases”)

The same process might have different ranks in different communicators

Communicators can be created “by hand” or using tools provided by MPI (not discussed in this tutorial)

Simple programs typically only use the predefined communicator **MPI_COMM_WORLD**

Simple MPI Program Identifying Processes

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

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

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Basic requirements for an MPI program

Data Communication

- Data communication in MPI is like email exchange
  - One process sends a copy of the data to another process (or a group of processes), and the other process receives it
- Communication requires the following information:
  - Sender has to know:
    - Whom to send the data to (receiver's process rank)
    - What kind of data to send (100 integers or 200 characters, etc)
    - A user-defined "tag" for the message (think of it as an email subject; allows the receiver to understand what type of data is being received)
  - Receiver "might" have to know:
    - Who is sending the data (OK if the receiver does not know; in this case sender rank will be **`MPI_ANY_SOURCE`**, meaning anyone can send)
    - What kind of data is being received (partial information is OK: I might receive *up to* 1000 integers)
    - What the user-defined "tag" of the message is (OK if the receiver does not know; in this case tag will be **`MPI_ANY_TAG`**)

More Details on Using Ranks for Communication

- When sending data, the sender has to specify the destination process' rank
  - Tells where the message should go
- The receiver has to specify the source process' rank
  - Tells where the message will come from
- `MPI_ANY_SOURCE` is a special "wild-card" source that can be used by the receiver to match any source

More Details on Describing Data for Communication

- MPI Datatype is very similar to a C or Fortran datatype
  - `int MPI_INT`
  - `double MPI_DOUBLE`
  - `char MPI_CHAR`
- More complex datatypes are also possible:
  - E.g., you can create a structure datatype that comprises of other datatypes
  - Or, a vector datatype for the columns of a matrix
- The “count” in `MPI_SEND` and `MPI_RECV` refers to how many datatype elements should be communicated

More Details on User “Tags” for Communication

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message
- For example, if an application is expecting two types of messages from a peer, tags can help distinguish these two types
- Messages can be screened at the receiving end by specifying a specific tag
- **`MPI_ANY_TAG`** is a special “wild-card” tag that can be used by the receiver to match any tag

MPI Basic (Blocking) Send

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

- The message buffer is described by (**`buf, count, datatype`**).
- The target process is specified by **`dest`** and **`comm`**.
  - **`dest`** is the rank of the target process in the communicator specified by **`comm`**.
- **`tag`** is a user-defined “type” for the message
- When this function returns, the data has been delivered to the system and the buffer can be reused.
  - The message may not have been received by the target process.

MPI Send Modes

Standard (MPI_Send)	The sending process returns when the system can buffer the message or when the message is received and the buffer is ready for reuse.
Buffered (MPI_Bsend)	The sending process returns when the message is buffered in an application-supplied buffer.
Synchronous (MPI_Ssend)	The sending process returns only if a matching receive is posted and the receiving process has started to receive the message.
Ready (MPI_Rsend)	The message is sent as soon as possible.

MPI Basic (Blocking) Receive

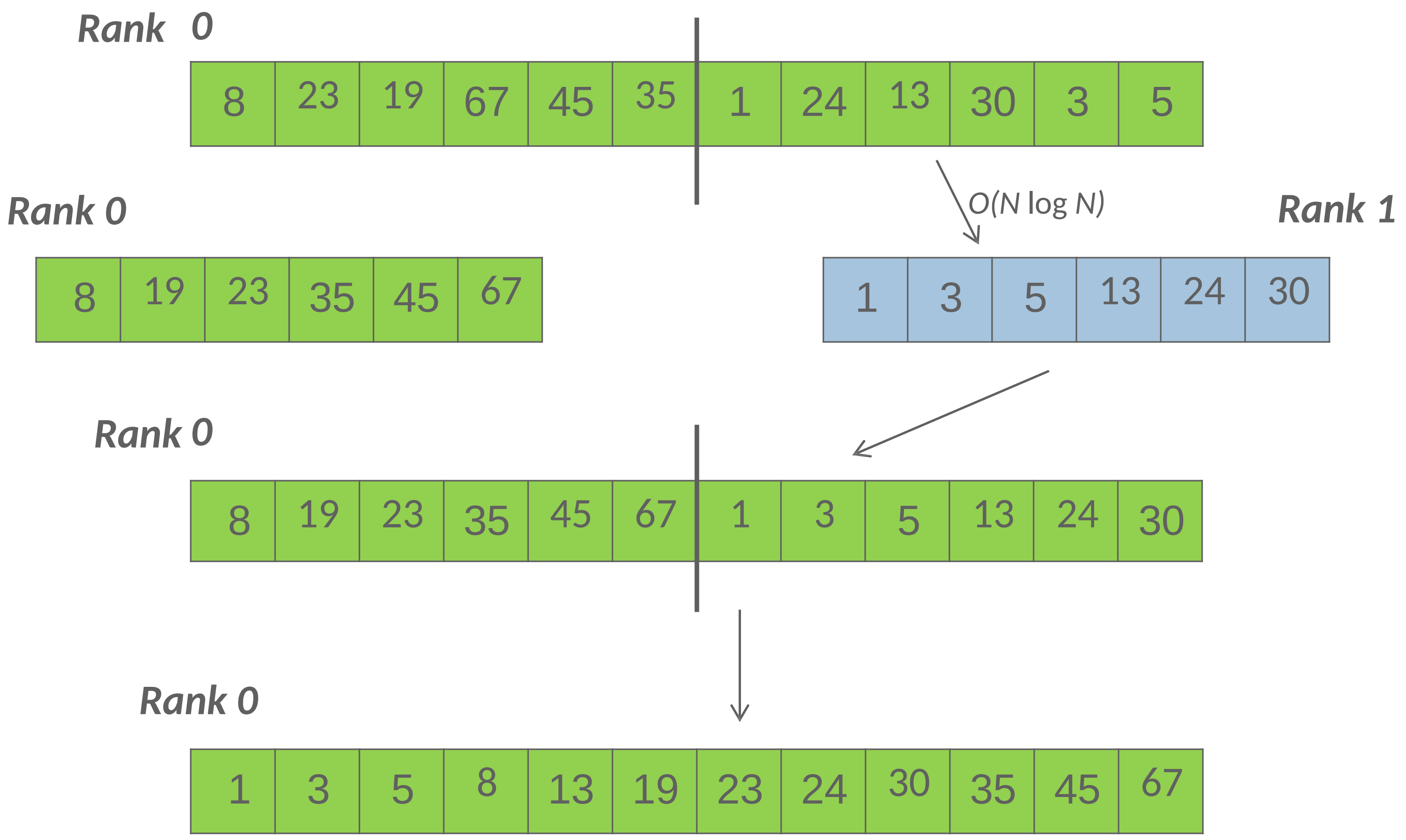
`MPI_RECV(buf, count, datatype, source, tag, comm, status)`

- Waits until a matching (on **`source`**, **`tag`**, **`comm`**) message is received from the system, and the buffer can be used.
- **`source`** is rank in communicator **`comm`**, or **`MPI_ANY_SOURCE`**.
- Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.
- **`status`** contains further information:
  - Who sent the message (can be used if you used **`MPI_ANY_SOURCE`**)
  - How much data was actually received
  - What tag was used with the message (can be used if you used **`MPI_ANY_TAG)`**
  - **`MPI_STATUS_IGNORE`** can be used if we don't need any additional information

Simple Communication in MPI

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

int main(int argc, char ** argv)
{
    int rank, data[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Finalize();
return 0;
}
```

Parallel Sort using MPI Send/Recv



Parallel Sort using MPI Send/Recv (contd.)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
     int rank;
     int a[1000], b[500];

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);  if (rank
     == 0) {
          MPI_Send(&a[500], 500, MPI_INT, 1, 0,
          MPI_COMM_WORLD);
          sort(a, 500);
          MPI_Recv(b, 500, MPI_INT, 1, 0,
          MPI_COMM_WORLD, &status);

          /* Serial: Merge array b and sorted part of
          array a */
     }
     else if (rank == 1) {
          MPI_Recv(b, 500, MPI_INT, 0, 0,
          MPI_COMM_WORLD, &status);
          sort(b, 500);
          MPI_Send(b, 500, MPI_INT, 0, 0,
          MPI_COMM_WORLD);
     }

     MPI_Finalize(); return 0;
}
```

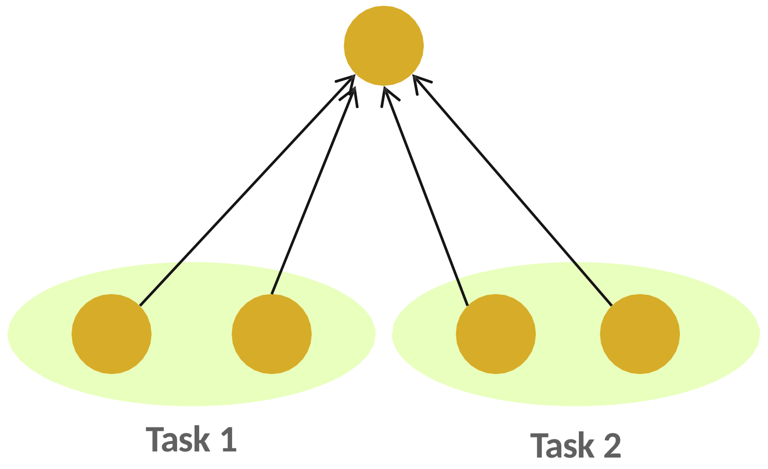
Status Object

- The status object is used after completion of a receive to find the actual length, source, and tag of a message
- Status object is MPI-defined type and provides information about:
  - The source process for the message (`status.MPI_SOURCE`)
  - The message tag (`status.MPI_TAG`)
  - Error status (`status.MPI_ERROR`)
- The number of elements received is given by:

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

`status`	return status of receive operation (status)
`datatype`	datatype of each receive buffer element (handle)
`count`	number of received elements (integer)(OUT)

Using the “status” field



- Each “worker process” computes some task (maximum 100 elements) and sends it to the “master” process together with its group number: the “tag” field can be used to represent the task
  - Data count is not fixed (maximum 100 elements)
  - Order in which workers send output to master is not fixed (different workers = different src ranks, and different tasks = different tags)

Using the “status” field (contd.)

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

int main(int argc, char ** argv)
{
    [...snip...]

    if (rank != 0)
        MPI_Send(data, rand() % 100, MPI_INT, 0, group_id, MPI_COMM_WORLD);
    else {
        for (i = 0; i < size – 1 ; i++) {
            MPI_Recv(data, 100, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_INT, &count);
            printf(“worker ID: %d; task ID: %d; count: %d\n”, status.source, status.tag, count);
        }
    }

    [...snip...]
}
```

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - `MPI_INIT` – initialize the MPI library (must be the first routine called)
  - `MPI_COMM_SIZE` - get the size of a communicator
  - `MPI_COMM_RANK` – get the rank of the calling process in the communicator
  - `MPI_SEND` – send a message to another process
  - `MPI_RECV` – send a message to another process
  - `MPI_FINALIZE` – clean up all MPI state (must be the last MPI function called by a process)
- For performance, however, you need to use other MPI features

Getting Started with MPI

- **Download your preferred MPI implementation. E.g.:**
  - MPICH: https://www.mpich.org (current stable: 3.2)
  - OpenMPI: https://www.open-mpi.org (current stable: 3.0)
- **Build (e.g., MPICH – pretty much the same for the others)**
  - `tar -xzvf mpich-3.2.tar.gz`
  - `cd mpich-3.2`
  - `./configure –-prefix=/where/to/install/mpich`
  - `make`
  - `make install`
  - `Add /where/to/install/mpich/bin to your PATH`

Compiling MPI programs with MPICH

- Compilation Wrappers
  - For C programs: `mpicc test.c –o test`
  - For C++ programs: `mpicxx test.cpp –o test`
  - For Fortran 77 programs: `mpif77 test.f –o test`
  - For Fortran 90 programs: `mpif90 test.f90 –o test`
- You can link other libraries are required too
  - – To link to a math library: `mpicc test.c –o test -lm`
- You can just assume that “mpicc” and friends have replaced
  - your regular compilers (gcc, gfortran, etc.)

Running MPI programs with MPICH

- Launch 16 processes on the local node:
  - `mpiexec –np 16 ./test`
- Launch 16 processes on 4 nodes (each has 4 cores)
  - `mpiexec –hosts h1:4,h2:4,h3:4,h4:4 –np 16 ./test`
    - Runs the first four processes on h1, the next four on h2, etc.
  - `mpiexec –hosts h1,h2,h3,h4 –np 16 ./test`
    - Runs the first process on h1, the second on h2, etc., and wraps around
    - So, h1 will have the 1st, 5th, 9th and 13th processes
- If there are many nodes, it might be easier to create a host file
  - `cat hf`

    ```
    h1:4  h2:2
    ```
  - `mpiexec –hostfile hf –np 16 ./test`

Blocking vs. Non-blocking Communication

- `MPI_SEND/MPI_RECV` are blocking communication calls
  - Return of the routine implies completion
  - When these calls return the memory locations used in the message transfer can be safely accessed for reuse
  - For "send" completion implies variable sent can be reused/modified
  - Modifications will not affect data intended for the receiver
  - For "receive" variable received can be read
- `MPI_ISEND/MPI_IRECV` are non-blocking variants
  - Routine returns immediately – completion has to be separately tested for
  - These are primarily used to overlap computation and communication to improve performance

Blocking Communication

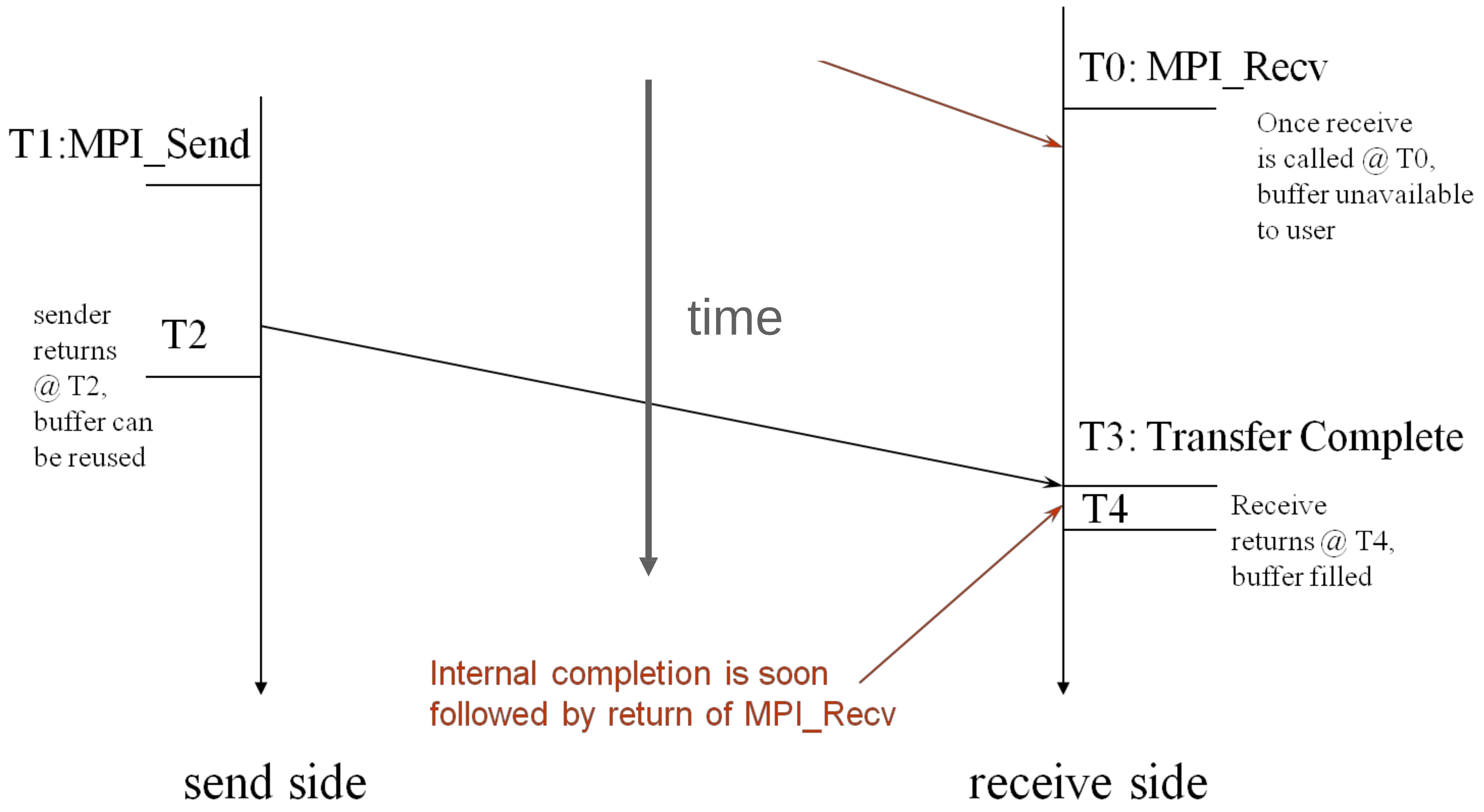
- In blocking communication.
  - **`MPI_SEND`** does not return until buffer is empty (available for reuse)
  - **`MPI_RECV`** does not return until buffer is full (available for use)
- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked until the receive buffer is filled
- Exact completion semantics of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks

```
If (rank == 0) Then
        Call mpi_send(..)
        Call mpi_recv(..)
Else
        Call mpi_send(..)
        Call mpi_recv(..)
Endif
```

Usually deadlocks ✉

 UNLESS you reverse send/recv

Blocking Send-Receive Diagram


T0: MPI_Recv
Once receive is called @ T0, buffer unavailable to user
T1:MPI_Send
sender returns @ T2, buffer can be reused
T2
time
T3: Transfer Complete
T4
Receive returns @ T4, buffer filled
Internal completion is soon followed by return of MPI_Recv
send side
receive side

Non-Blocking Communication

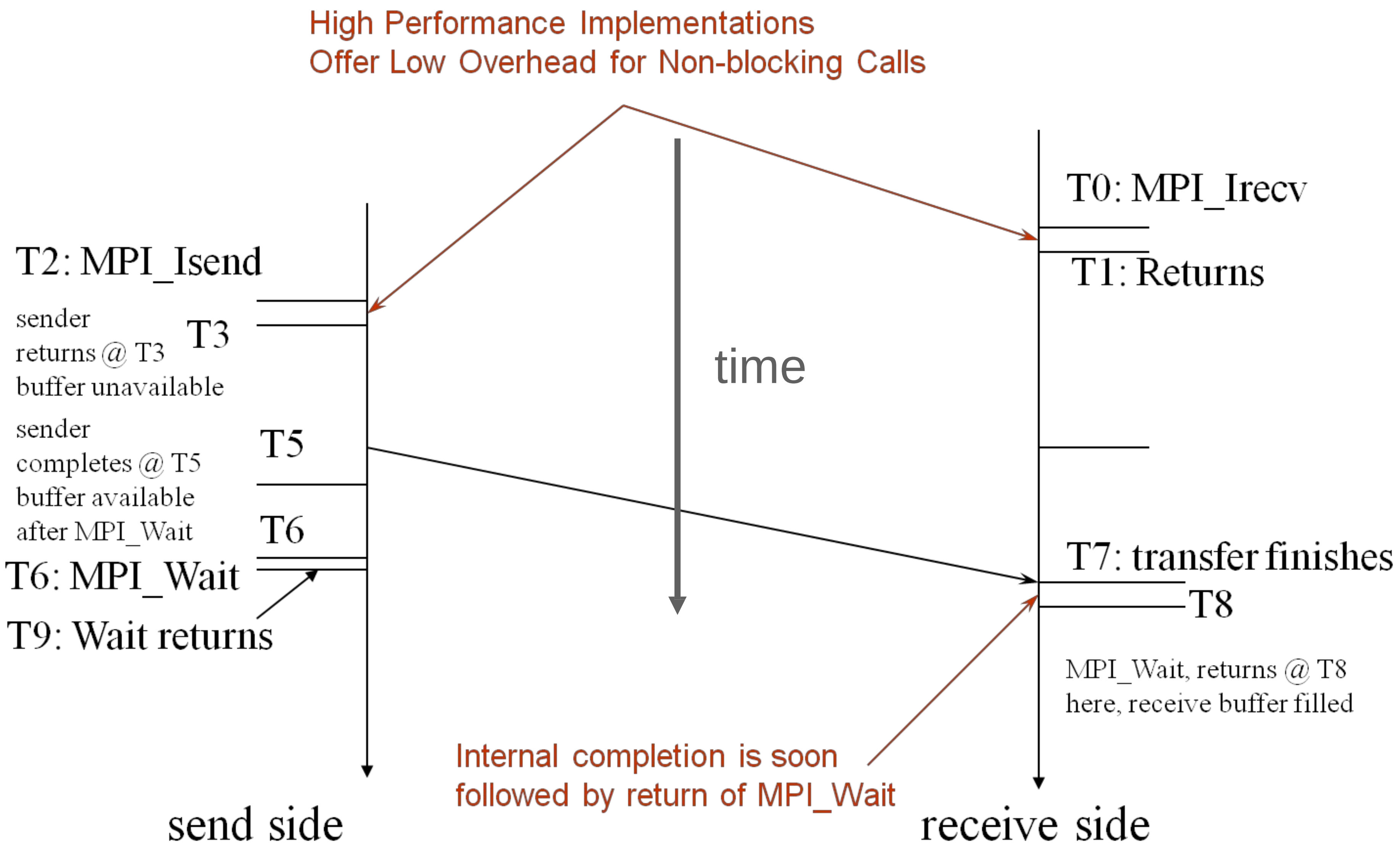
- Non-blocking (asynchronous) operations return (immediately) “request handles” that can be waited on and queried
  - `MPI_ISEND(start, count, datatype, dest, tag, comm, request)`
  - `MPI_IRECV(start, count, datatype, src, tag, comm, request)`
  - `MPI_WAIT(request, status)`
- Non-blocking operations allow overlapping computation and communication
- One can also test without waiting using **MPI_TEST**
  - **MPI_TEST(request, flag, status)**
- Anywhere you use **MPI_SEND** or **MPI_RECV**, you can use the pair of **MPI_ISEND/MPI_WAIT** or **MPI_IRECV/MPI_WAIT**
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution instead of barriers

Blocking	MPI_Send	MPI_Bsend	MPI_Ssend	MPI_Rsend	MPI_Recv
Non-blocking	MPI_Isend	MPI_Ibsend	MPI_Issend	MPI_Irsend	MPI_Irecv

Multiple Completions

- It is sometimes desirable to wait on multiple requests:
  - `MPI_Waitall(count, array_of_requests, array_of_statuses)`
  - `MPI_Waitany(count, array_of_requests, &index, &status)`
  - `MPI_Waitsome(count, array_of_requests, array_of_indices,  array_of_statuses)`
- There are corresponding versions of **`test`** for each of these

Non-Blocking Send-Receive Diagram



Message Completion and Buffering

- For a communication to succeed:
  - Sender must specify a valid destination rank
  - Receiver must specify a valid source rank (including MPI_ANY_SOURCE)
  - The communicator must be the same
  - Tags must match
  - Receiver's buffer must be large enough
- A send has completed when the user supplied buffer can be reused

```
*buf =3;
MPI_Send(buf, 1, MPI_INT …)
*buf = 4; /* OK, receiver will always
 receive 3 */
```

```
*buf =3;
MPI_Isend(buf, 1, MPI_INT …)
*buf = 4; /*Not certain if receiver
 gets 3 or 4 or anything else */
MPI_Wait(…);
```

- Just because the send completes does not mean that the receive has completed
  - Message may be buffered by the system
  - Message may still be in transit

A Non-Blocking communication example

```
int main(int argc, char ** argv)
{
    [...snip...]
    if (rank == 0) {
        for (i=0; i< 100; i++) {
            /* Compute each data element and send it out */  data[i] = compute(i);
            MPI_ISend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request[i]);
        }
        MPI_Waitall(100, request, MPI_STATUSES_IGNORE)
    }
    else {
        for (i = 0; i < 100; i++)
            MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }
    [...snip...]
}
```

Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in the communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECV`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

MPI Collective Communication

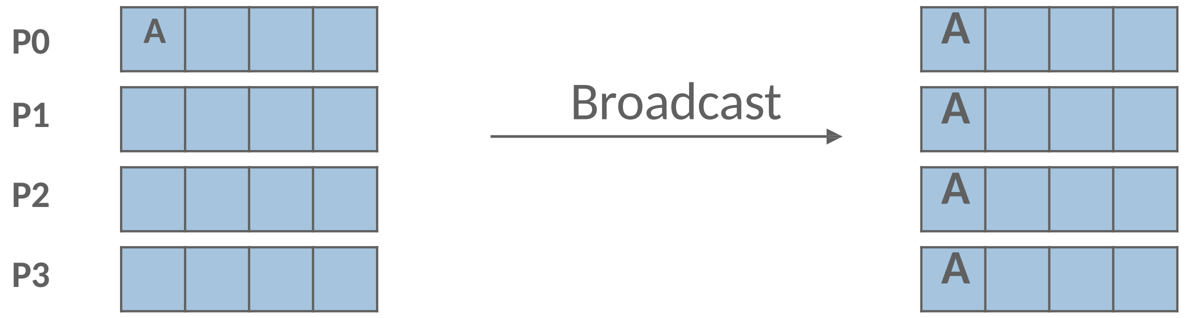
- Communication and computation is coordinated among a group of processes in a communicator
- Tags are not used; different communicators deliver similar functionality
- Non-blocking collective operations in MPI-3
  - Covered in the advanced tutorial (but conceptually simple)
- Three classes of operations: synchronization, data movement, collective computation

Synchronization

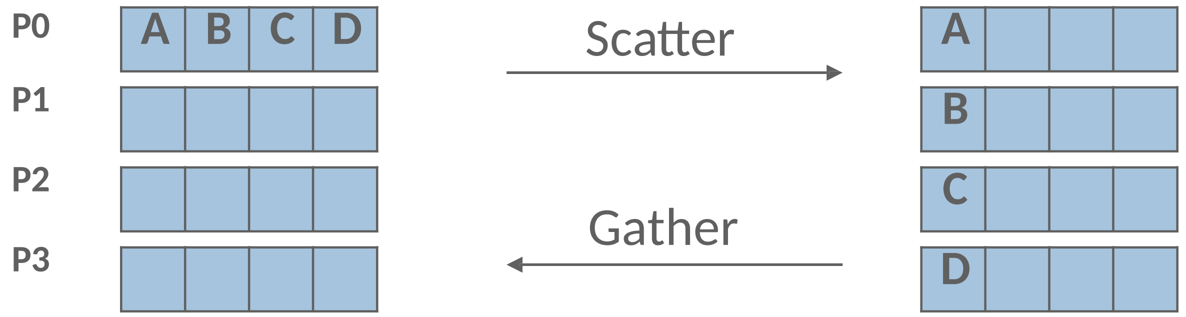
- `MPI_BARRIER(comm)`
  - Blocks until all processes in the group of the communicator **`comm`** call it
  - A process cannot get out of the barrier until all other processes have reached barrier

Collective Data Movement

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



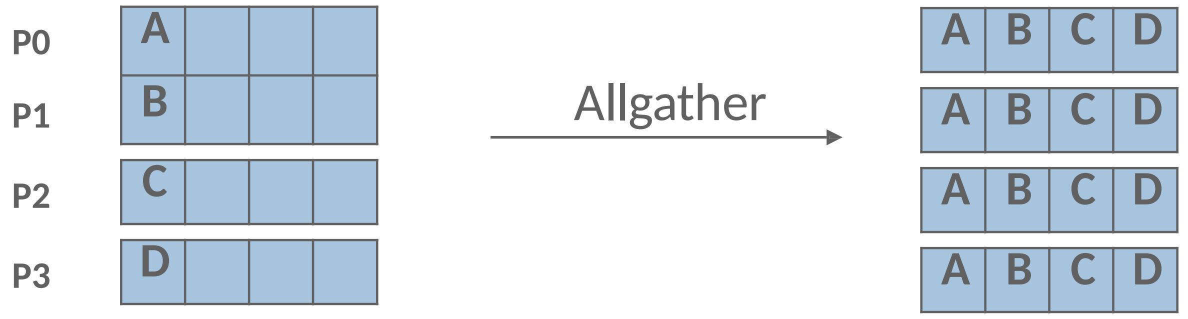
```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount,
               MPI_Datatype recvtype, int root, MPI_Comm comm)
```



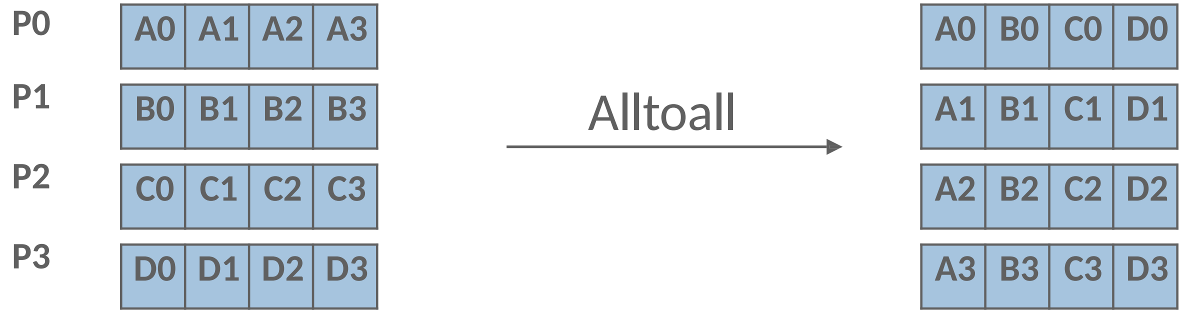
```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount,
               MPI_Datatype recvtype, int root, MPI_Comm comm)
```

More Collective Data Movement

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

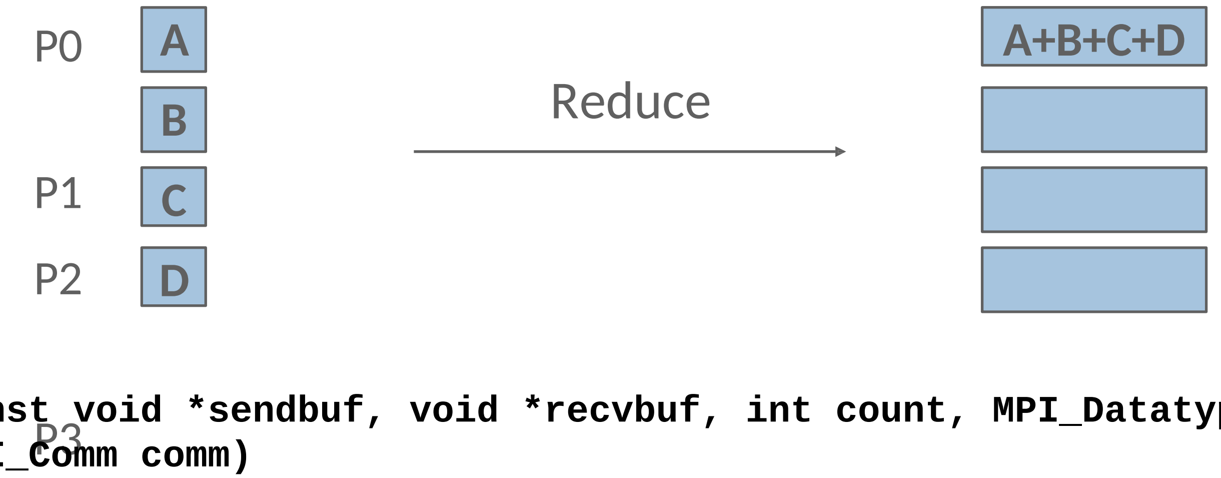


```
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int
recvcount,
                 MPI_Datatype recvtype, MPI_Comm comm)
```

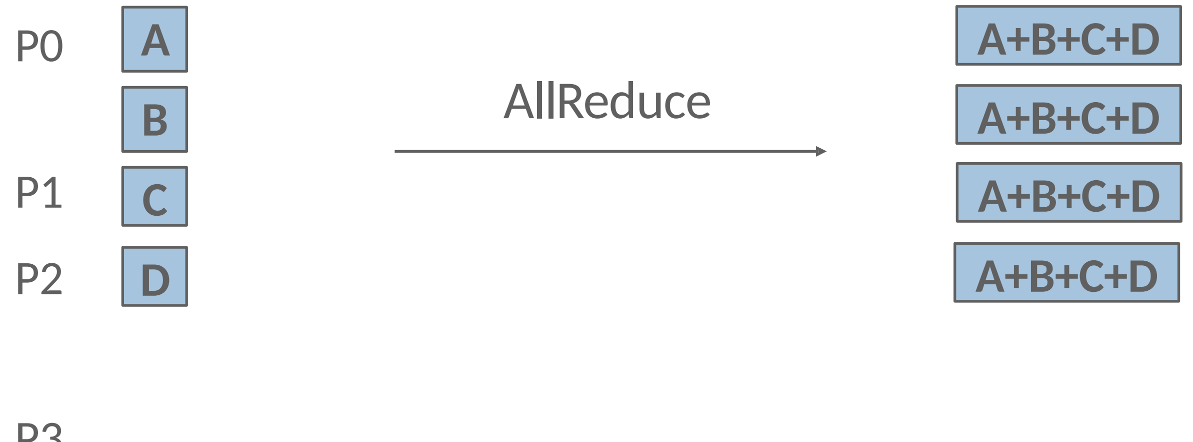


Collective Computation

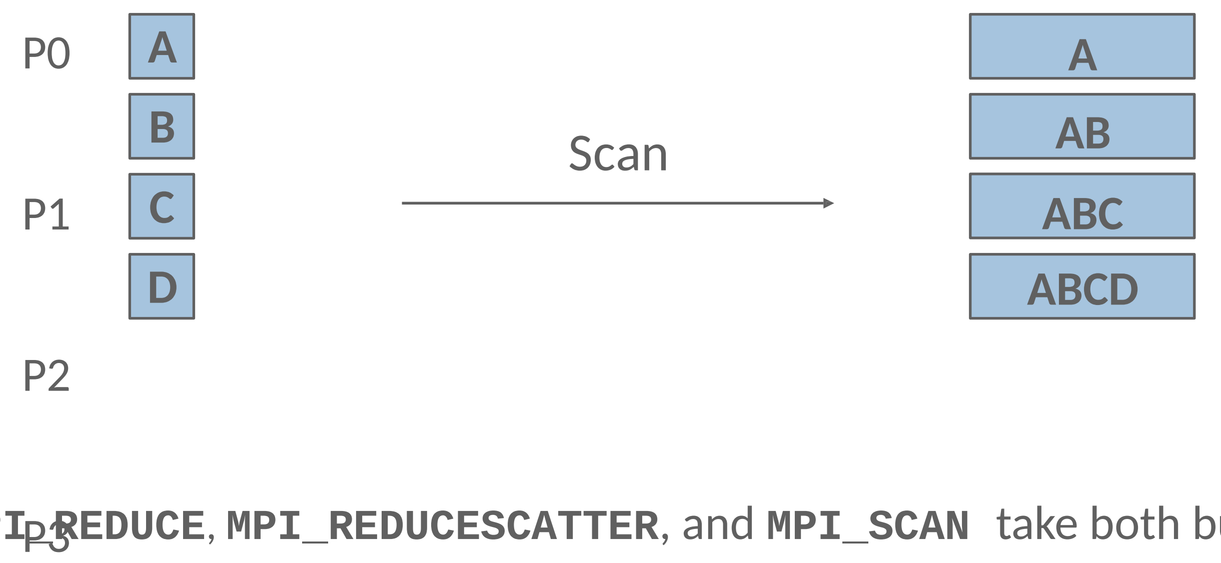
```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)
```



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



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



- **`MPI_ALLREDUCE`**, **`MPI_REDUCE`**, **`MPI_REDUCESCATTER`**, and **`MPI_SCAN`** take both built-in and user-defined combiner functions

MPI Built-in Collective Computation Operations

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Defining your own Collective Operations

- Create your own collective computations with:

```
MPI_OP_CREATE(user_fn, commutes, &op);
MPI_OP_FREE(&op);

user_fn(invec, inoutvec, len, datatype);
```

- The user function should perform:

```
 inoutvec[i] = invec[i] op   inoutvec[i];
```

  for i from 0 to len-1

- The user function can be non-commutative, but must be associative

Homework

- Download π-seq from the website
- Make it parallel using MPI!
- Did the perfomance improve? How did you measure it?