Introduction to Parallel Programming with MPI

Slides are available at

http://www.mcs.anl.gov/~balaji/tmp/csdms-mpi-basic.pdf

Pavan Balaji

Argonne National Laboratory

balaji@mcs.anl.gov

http://www.mcs.anl.gov/~balaji

General principles in this tutorial

- Everything is practically oriented
- We will use lots of real example code to illustrate concepts
- At the end, you should be able to use what you have learned and write real code, run real programs
- Feel free to interrupt and ask questions
- If my pace is too fast or two slow, let me know

About Myself

- Assistant Computer Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory
- Research interests in parallel programming, message passing, global address space and task space models
- Co-author of the MPICH implementation of MPI
- Participate in the MPI Forum that defines the MPI standard
  - Co-author of the MPI-2.1 and MPI-2.2 standards
  - Lead the hybrid programming working group for MPI-3
  - Active participant in the remote memory access (global address space runtime system) working group for MPI-3

What we will cover in this tutorial

- **What is MPI?**
- Fundamental concepts in MPI
- Point-to-point communication in MPI
- Group (collective) communication in MPI
- A sneak peak at MPI-3
- Conclusions and Final Q/A

The switch from sequential to parallel computing

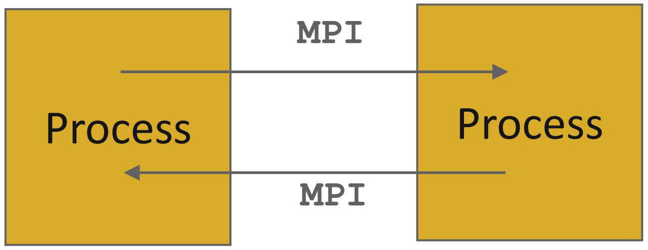
- Moore's law continues to be true, but…
  - Processor speeds no longer double every 18-24 months
  - Number of processing units double, instead
    - Multi-core chips (dual-core, quad-core)
  - No more automatic increase in speed for software
- Parallelism is the norm
  - Lots of processors connected over a network and coordinating to solve large problems
  - Used every where!
    - By USPS for tracking and minimizing fuel routes
    - By automobile companies for car crash simulations

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)

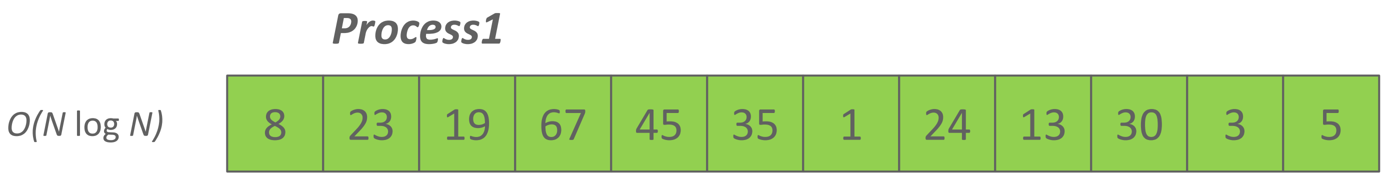
The Message-Passing Model

- 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.



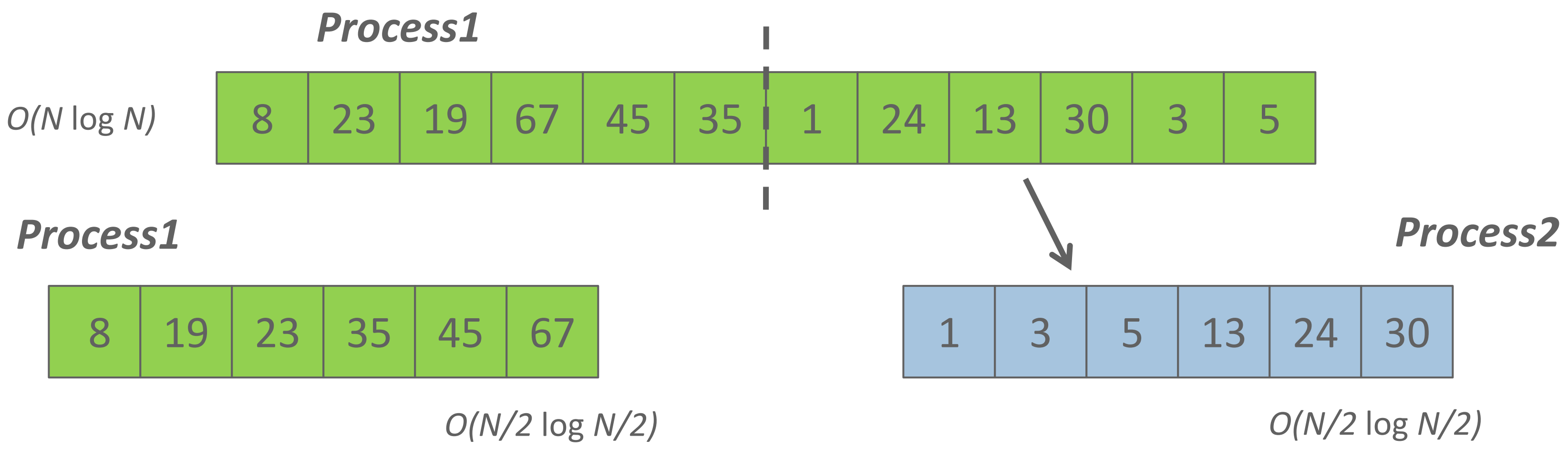
The Message-Passing Model (an example)

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



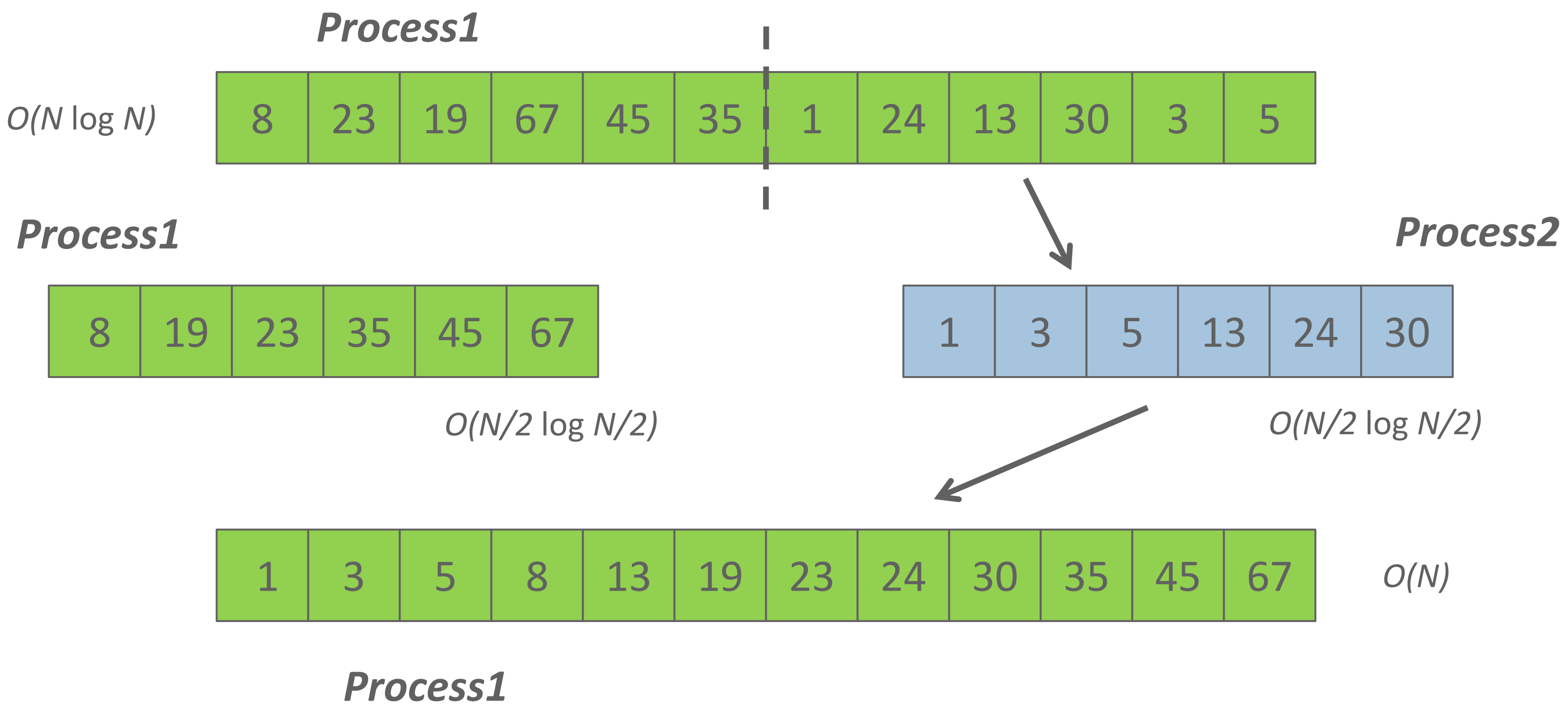
The Message-Passing Model (an example)

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



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

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

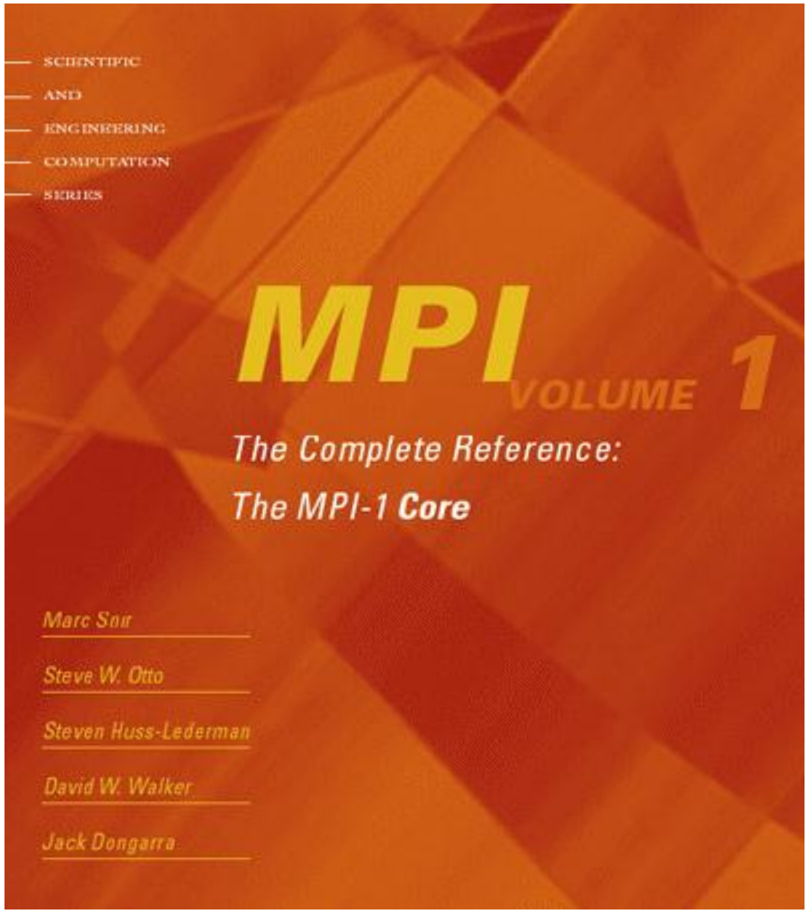
What is in MPI-1

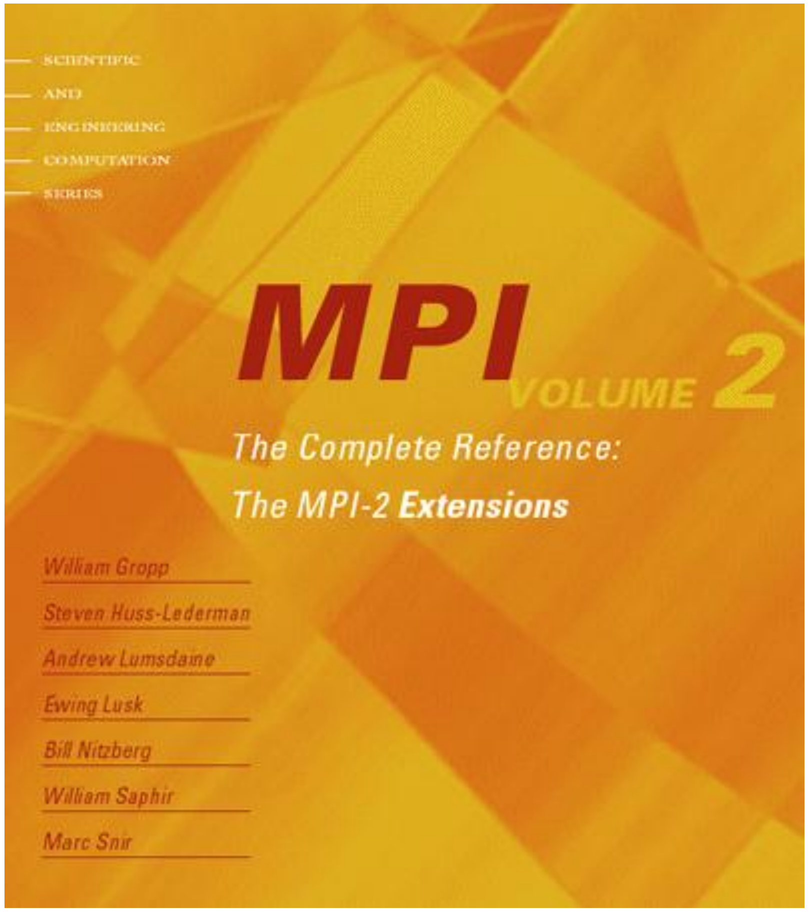
- Basic functions for communication (100+ functions)
- Blocking sends, receives
- Nonblocking sends and receives
- Variants of above
- Rich set of collective communication functions
  - Broadcast, scatter, gather, etc
  - Very important for performance; widely used
- Datatypes to describe data layout
- Process topologies
- C and Fortran bindings
- Error codes and classes

Following MPI Standards

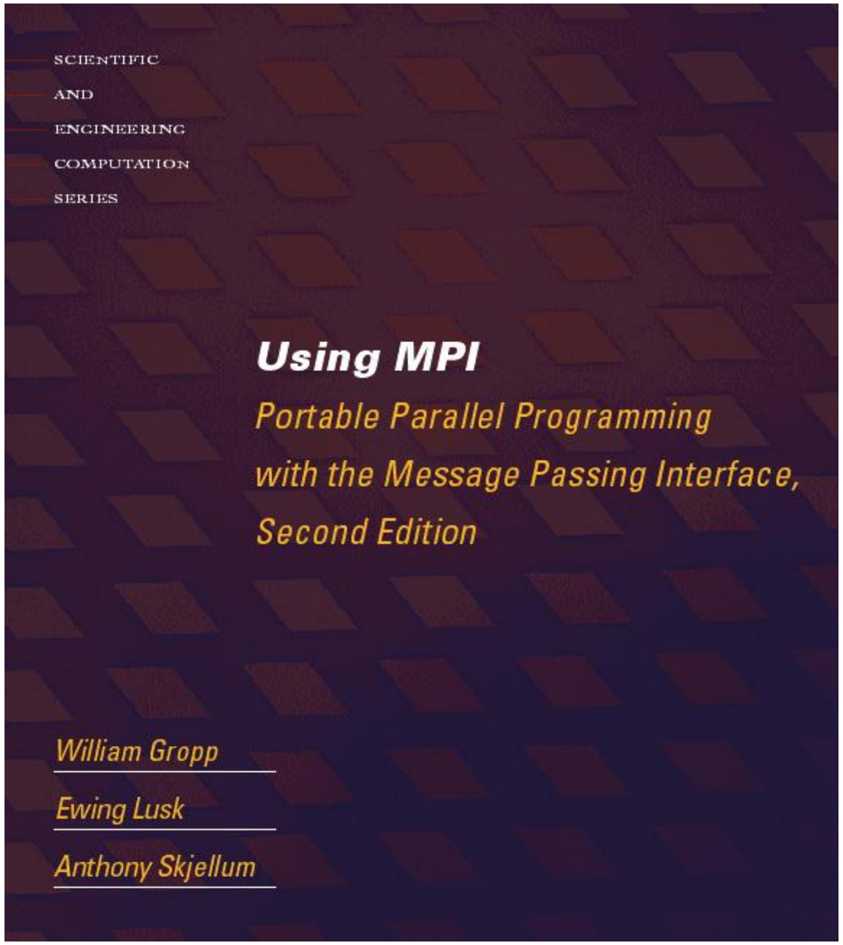
- MPI-2 was released in 2000
  - Several additional features including MPI + threads, MPI-I/O, remote memory access functionality and many others
- MPI-2.1 (2008) and MPI-2.2 (2009) were recently released with some corrections to the standard and small features
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

The MPI Standard (1 & 2)





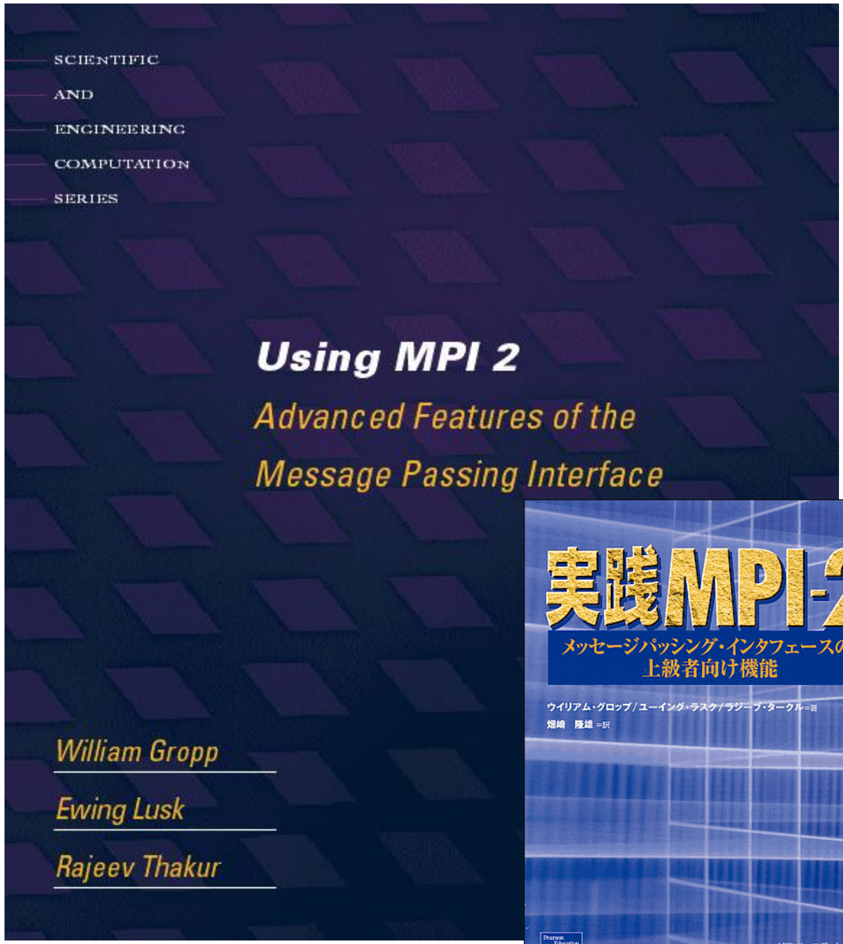
Tutorial Material on MPI-1 and MPI-2


SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using MPI

Portable Parallel Programming with the Message Passing Interface, Second Edition

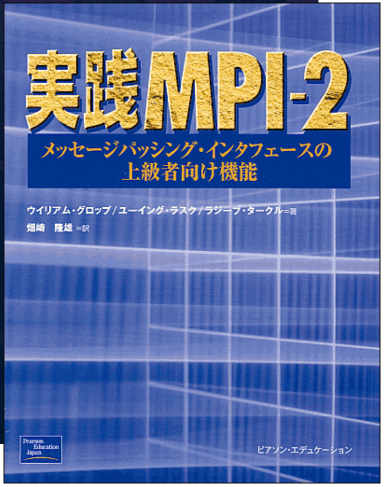
William Gropp
Ewing Lusk
Anthony Skjellum


SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using MPI 2

Advanced Features of the Message Passing Interface

William Gropp
Ewing Lusk
Rajeev Thakur


実践MPI-2
メッセージパッシング・インタフェースの
上級者向け機能

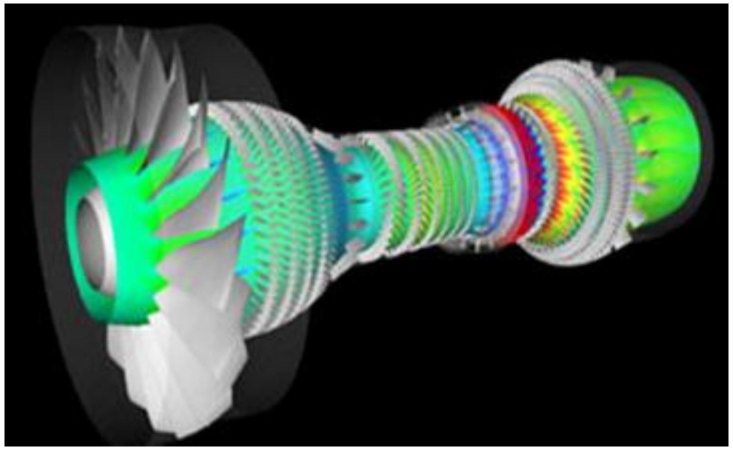
ウイリアム・グロップ/ユーイング・ラスク/ラジーブ・タークル=著
畑崎　隆雄 =訳

Pearson Education Japan
ピアソン・エデュケーション

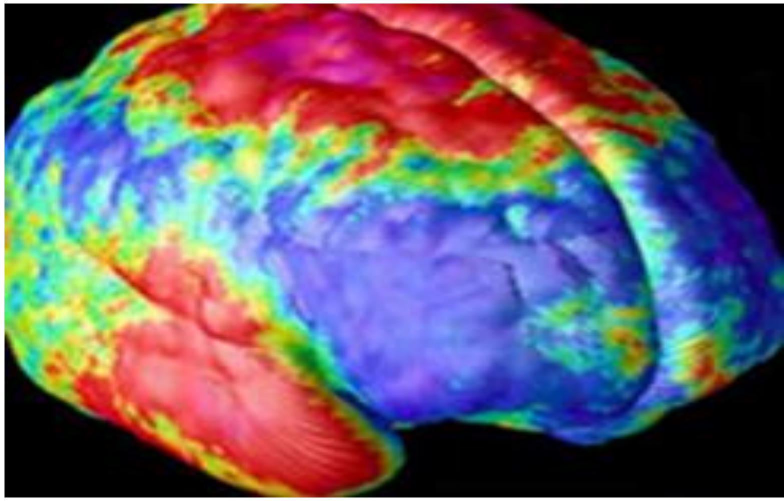
http://www.mcs.anl.gov/mpi/usingmpi
http://www.mcs.anl.gov/mpi/usingmpi2

Applications (Science and Engineering)

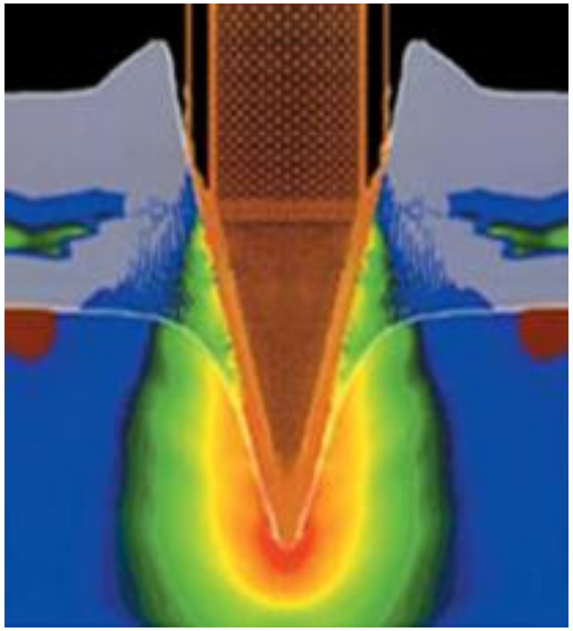
- MPI is widely used in large scale parallel applications in science and engineering
  - Atmosphere, Earth, Environment
  - Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
  - Bioscience, Biotechnology, Genetics
  - Chemistry, Molecular Sciences
  - Geology, Seismology
  - Mechanical Engineering - from prosthetics to spacecraft
  - Electrical Engineering, Circuit Design, Microelectronics
  - Computer Science, Mathematics



Turbo machinery (Gas turbine/compressor)

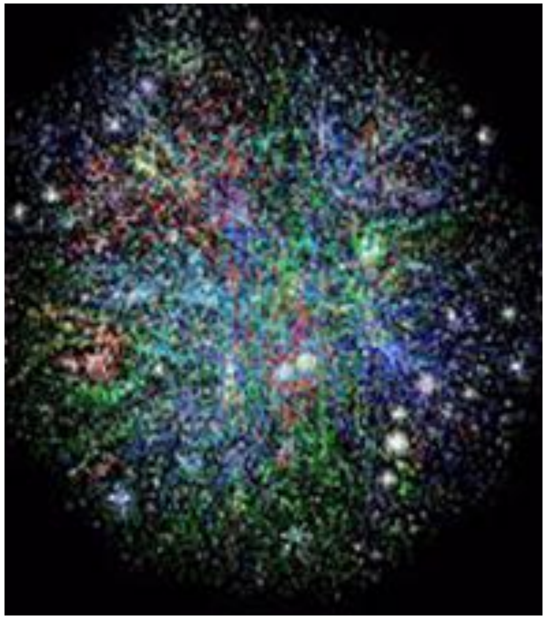


Biology application



Drilling application

Transportation & traffic application



Astrophysics application

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
  - MPICH2 is a popular open-source and free implementation of MPI
  - Vendors and other collaborators take MPICH2 and add support for their systems
    - Intel MPI, IBM Blue Gene MPI, Cray MPI, Microsoft MPI, MVAPICH2, MPICH2-MX

Important considerations while using MPI

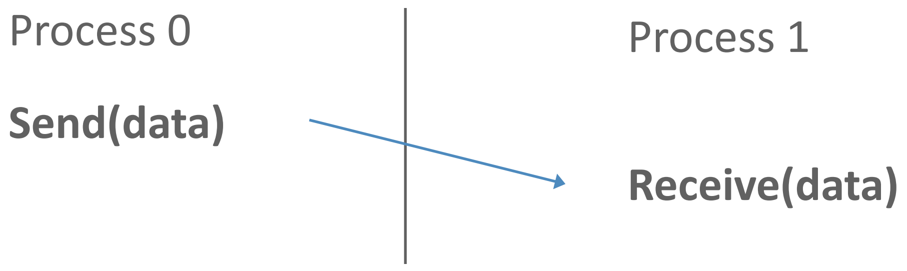
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

What we will cover in this tutorial

- What is MPI?
- **Fundamental concepts in MPI**
- Point-to-point communication in MPI
- Group (collective) communication in MPI
- A sneak peak at MPI-3
- Conclusions and Final Q/A

MPI Basic Send/Receive

- Simple communication model

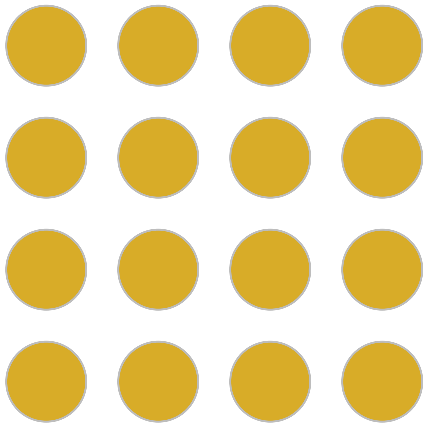


- Application needs to specify to the MPI implementation:
  1. How will processes be identified?
  2. How will “data” be described?

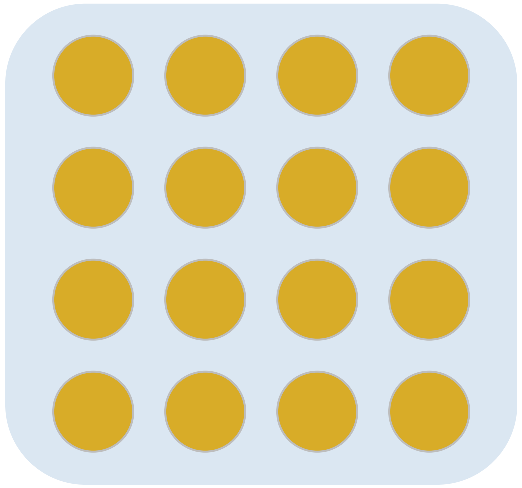
Process Identification

- MPI Processes can be collected into *groups*
- Each group of processes can have multiple *contexts*
- A group and context together form a *communicator*
  - Simple programs typically use just one context
  - In this case, a communicator is equivalent to a group of processes
- A process is identified by its *rank* within its communicator
- There is a default communicator that contains all initial processes, called **`MPI_COMM_WORLD`**

Communicator = Group of processes + Context

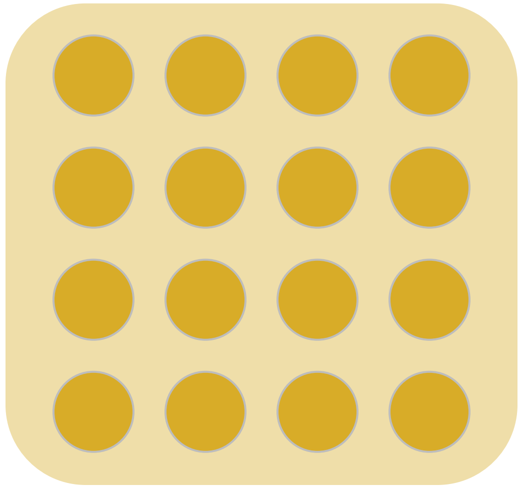


Communicator = Group of processes + Context



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

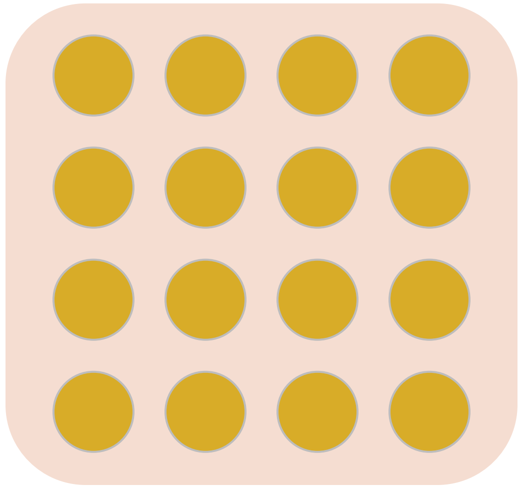
Communicator = Group of processes + Context



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

Can make copies of this communicator (same group of processes, but different contexts)

Communicator = Group of processes + Context

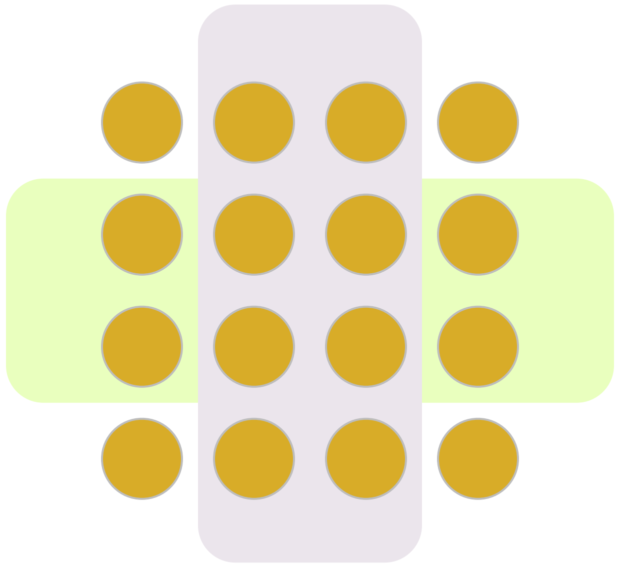


When you start an MPI program, there is one predefined communicator MPI_COMM_WORLD

Can make copies of this communicator (same group of processes, but different contexts)

Communicator = Group of processes + Context

Communicators do not need to contain all processes in the system



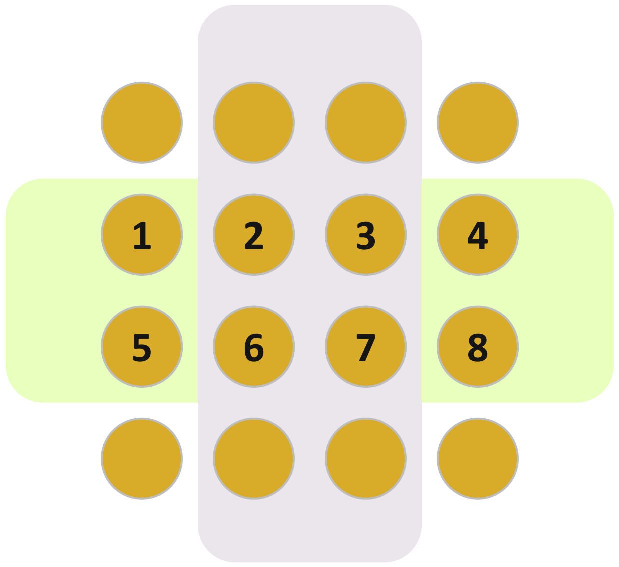
When you start an MPI program, there is one predefined communicator MPI_COMM_WORLD

Can make copies of this communicator (same group of processes, but different contexts)

Communicator = Group of processes + Context

Communicators do not need to contain all processes in the system

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



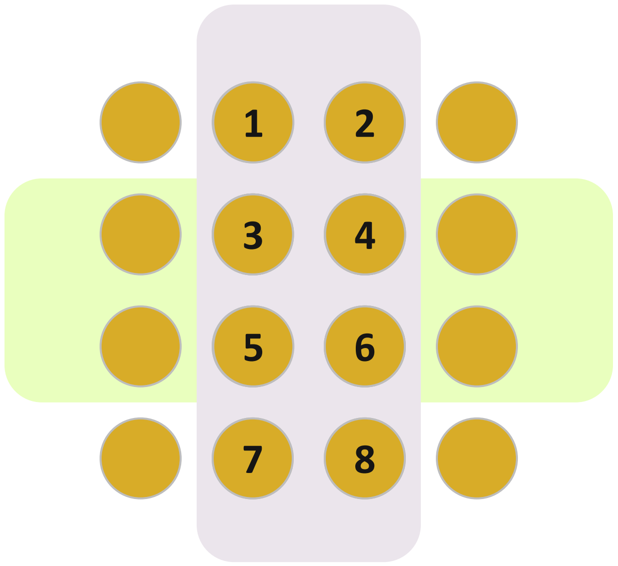
When you start an MPI program, there is one predefined communicator MPI_COMM_WORLD

Can make copies of this communicator (same group of processes, but different contexts)

Communicator = Group of processes + Context

Communicators do not need to contain all processes in the system

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



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

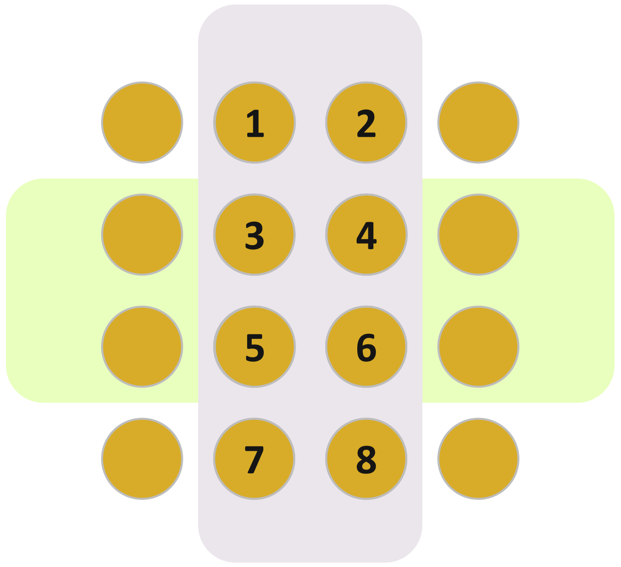
Can make copies of this communicator (same group of processes, but different contexts)

The same process might have different ranks in different communicator

Communicator = Group of processes + Context

Communicators do not need to contain all processes in the system

Every process in a communicator has an ID called as "rank"



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

Can make copies of this communicator (same group of processes, but different contexts)

The same process might have different ranks in different communicator

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

Source and Destination Ranks

- 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

Simple MPI Program

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

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


    MPI_Init(&argc, &argv);





    MPI_Finalize();
    return 0;
}
```

Basic requirements for an MPI program

Simple MPI Program

```c
#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;
}
```

Data Description: MPI Datatypes

- 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 → a char, an int and a double.
  - 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

Recognizing/Screening different types of data: MPI Tags

- 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

What we will cover in this tutorial

- What is MPI?
- Fundamental concepts in MPI
- **Point-to-point communication in MPI**
- Group (collective) communication in MPI
- A sneak peak at MPI-3
- Conclusions and Final Q/A

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 Basic (Blocking) Receive

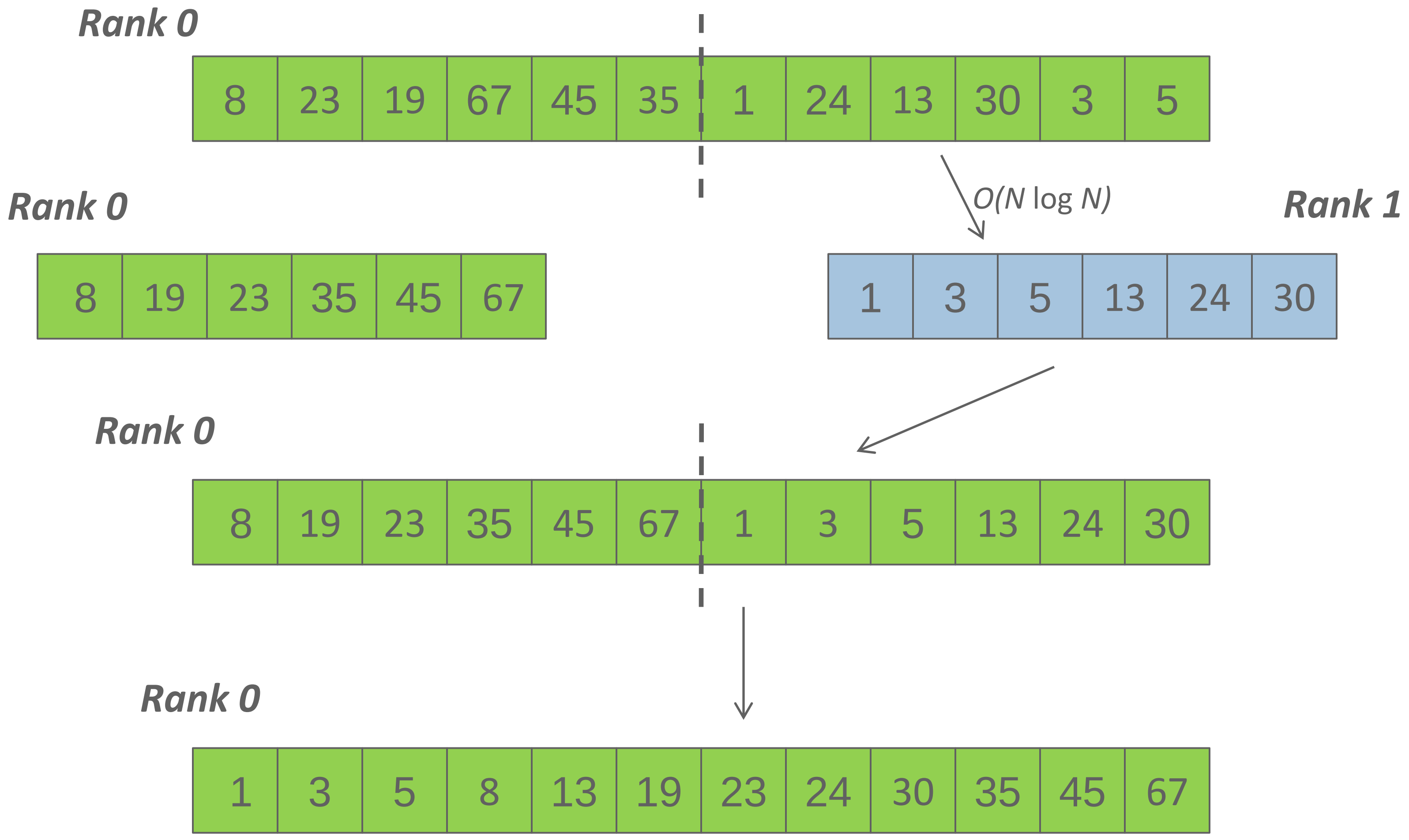
`MPI_RECV(buf, count, datatype, source, tag, comm, status)`

- 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
  - How much data was actually received
  - `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
        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>

void sort(int * x, int count, int * y)
{
    /* Sort the array ‘x’ of count elements and
     * place the sorted array in y */
}

int main( int argc, char ** argv )
{
    int rank, size;
    int a[1000], b[1000]; /* Array of 1000 integers */
    MPI_Status status;

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

Parallel Sort using MPI Send/Recv (contd.)

```
    if (rank == 0) {
        MPI_Send(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD);
        sort(&a[0], 500, &b[0]);
        MPI_Recv(&b[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD,
                  &status);
    }
    else {
        MPI_Recv(&a[0], 500, MPI_INT, 0, 0, MPI_COMM_WORLD,
                  &status);
        sort(&a[0], 500, &b[0]);
        MPI_Send(&b[0], 500, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    for (x = 0, i = 0, j = 500; i < 500 || j < 1000; x++) {
        if (b[i] < b[j]) a[x] = b[i++];
        else a[x] = b[j++];
    }
    MPI_Finalize(); return 0;
}
```

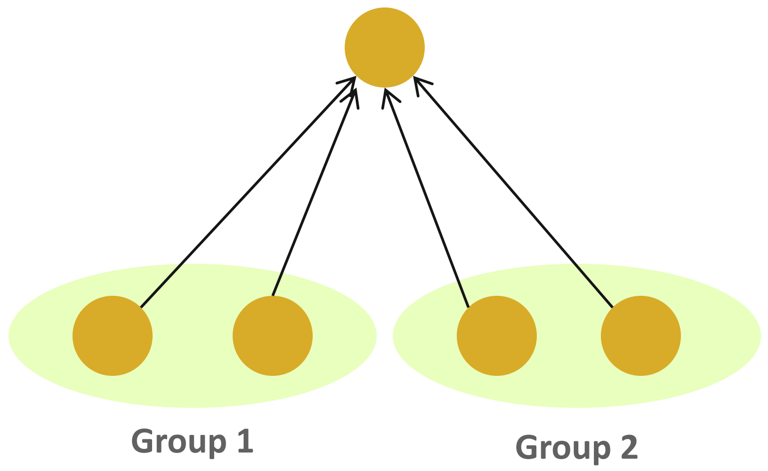
Status Object

- 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.source`)
  - The message tag (`status.tag`)
- 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 data (maximum 100 elements) and sends it to the “master” process together with its group number: the “tag” field can be used to represent the group ID
  - 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 groups = 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 {
        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; group 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

Blocking vs. Non-blocking Communication

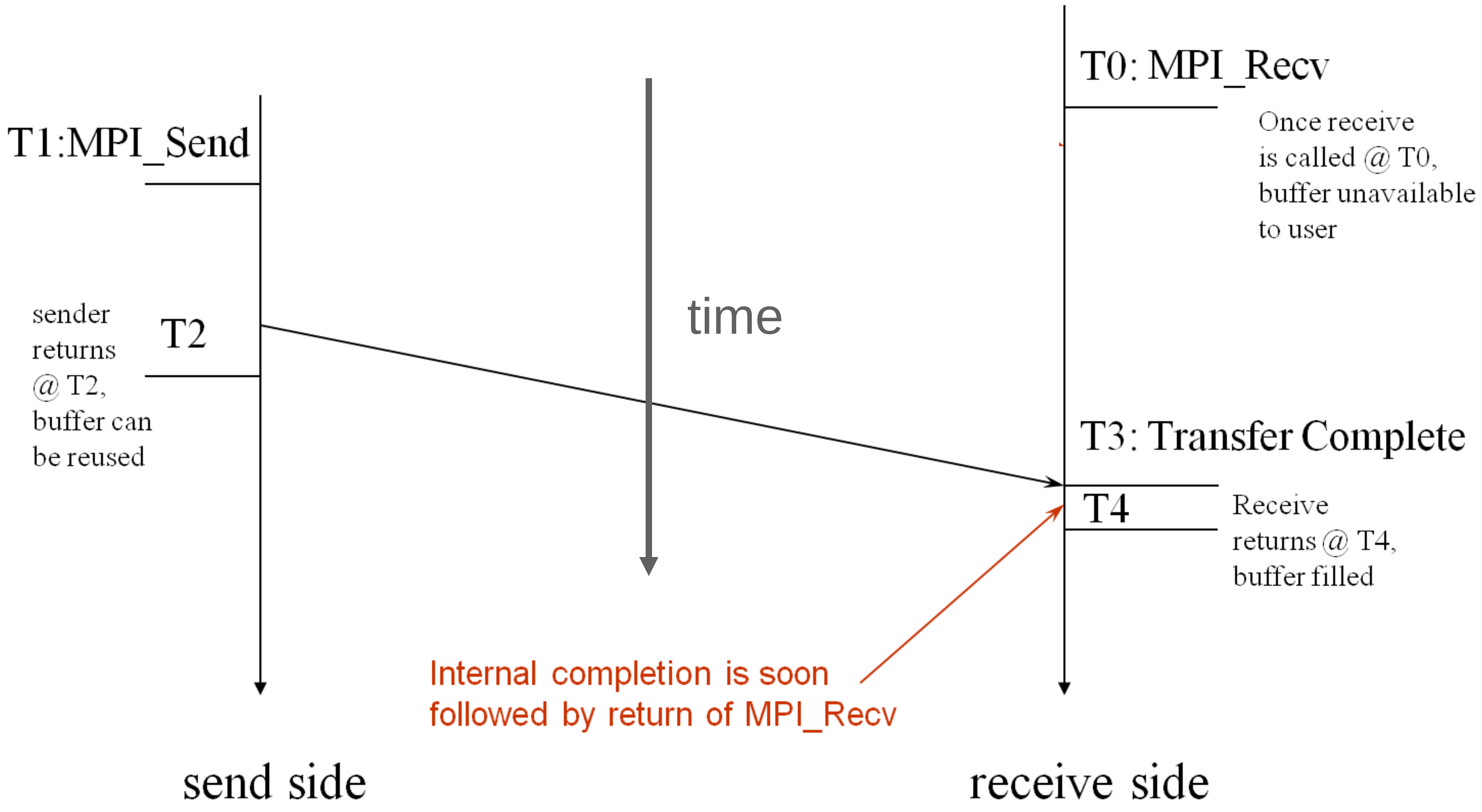
- When these calls return the memory locations used in the message transfer can be safely accessed for reuse
  - Modifications will not affect data intended for the receiver
  - For "send" completion implies variable sent can be reused/modified
  - For "receive" variable received can be read
- `MPI_SEND/MPI_RECV` are blocking communication calls
  - Return of the routine implies completion
- Non-blocking variants of these are also available
  - `MPI_ISEND/MPI_IRECV`
  - 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 complete until buffer is empty (available for reuse)
  - **`MPI_RECV`** does not complete 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
- Completion 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(..)
Usually deadlocks →      Else
                               Call mpi_send(..)     ← UNLESS you reverse send/recv
                               Call mpi_recv(..)
                         Endif
```

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

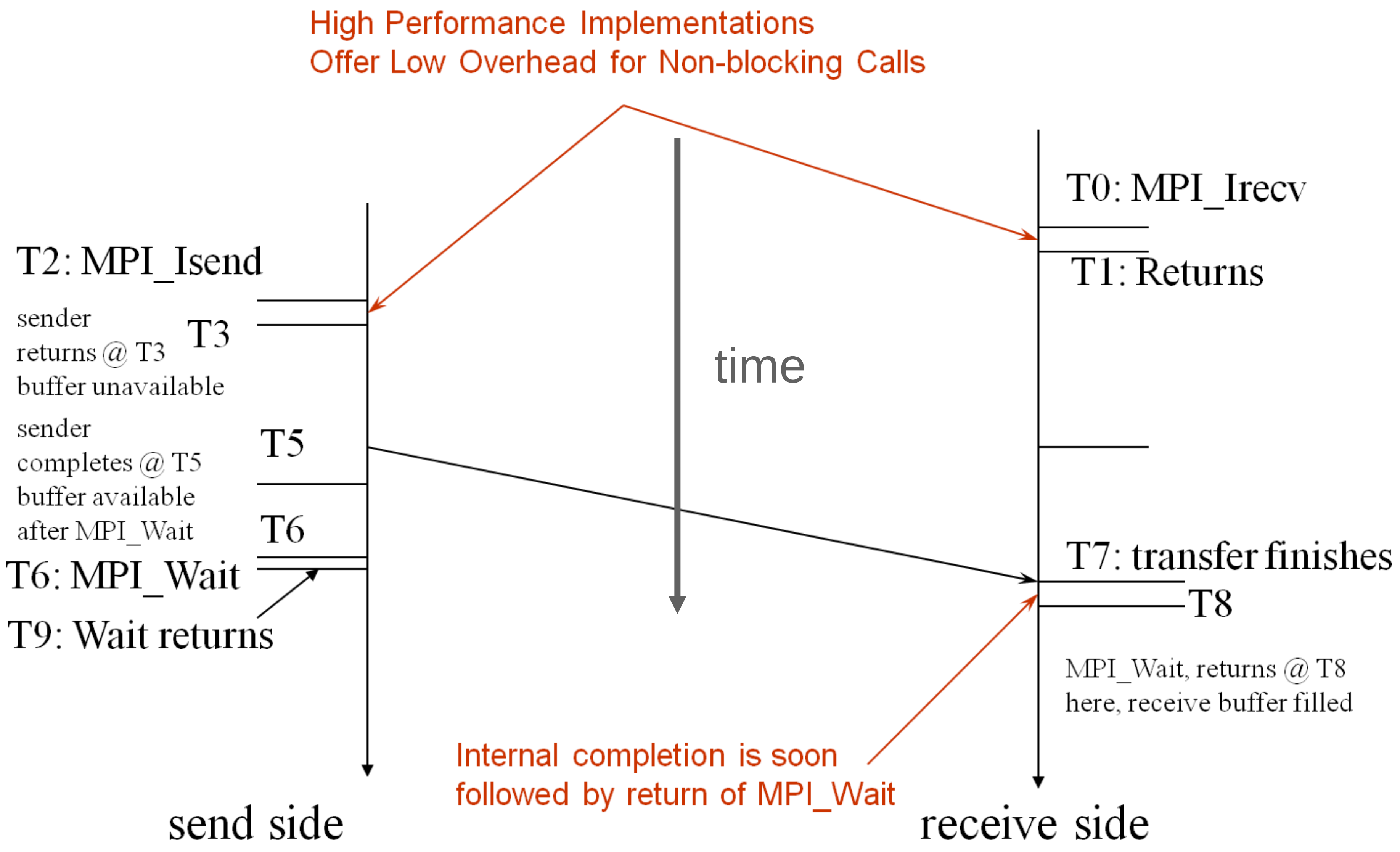
Non-Blocking Communication

- 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

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

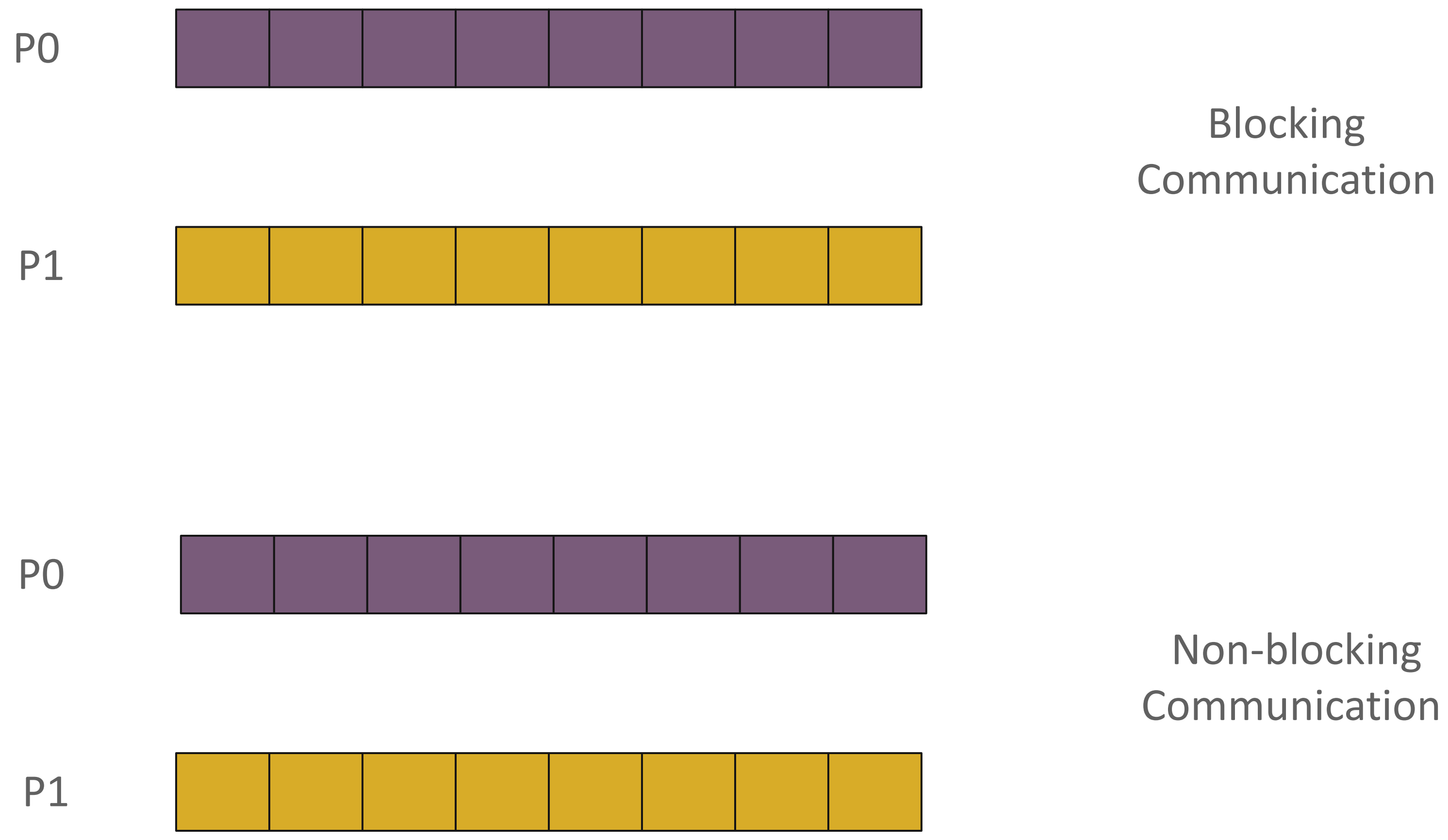
- For a communication to succeed:
  - Sender must specify a valid destination rank
  - Receiver must specify a valid source rank
  - The communicator must be the same
  - Tags must match
  - Receiver's buffer must be large enough
- A send has completed when the user supplied buffer can be reused

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

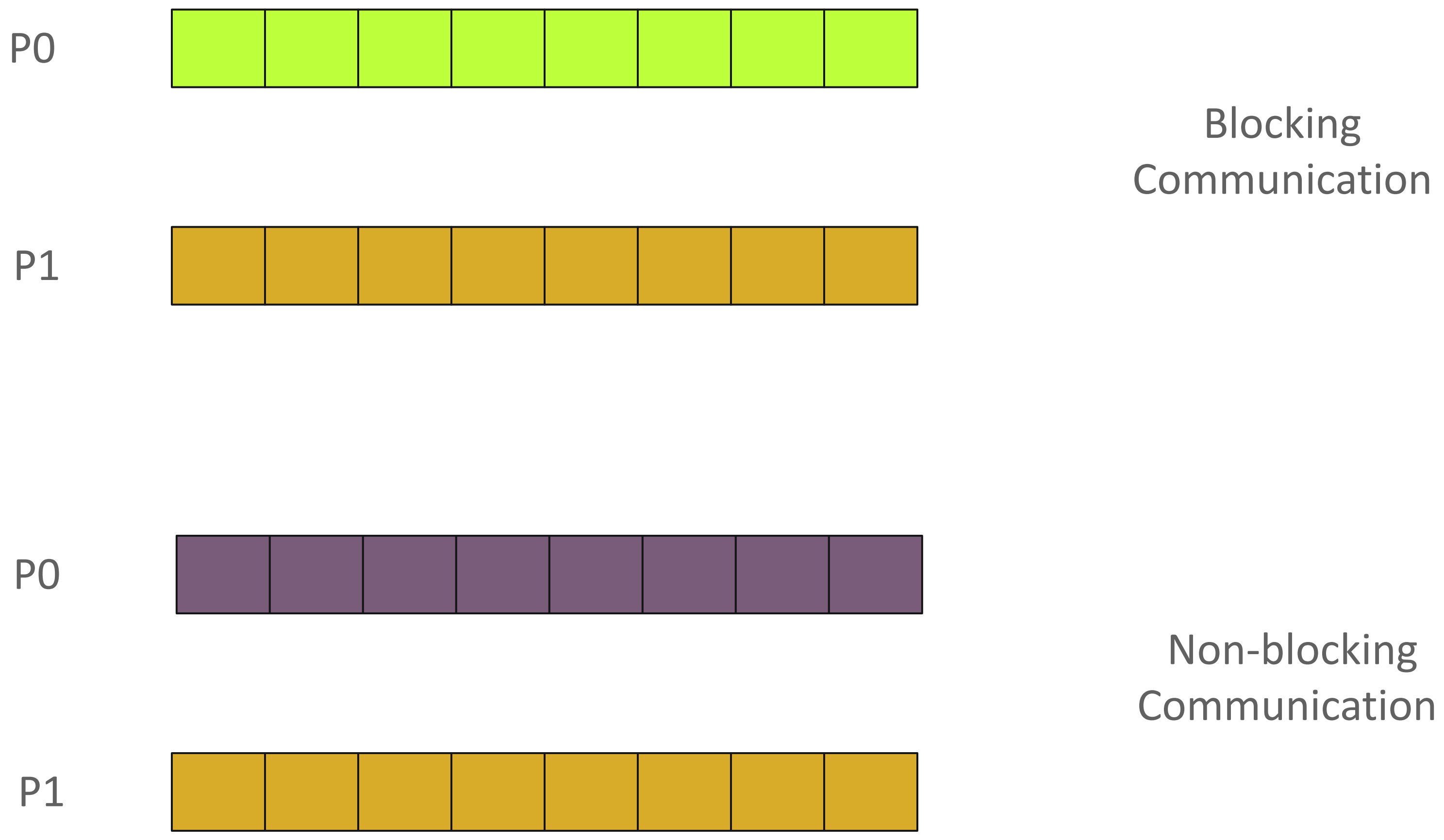
```
*buf =3;
MPI_Isend(buf, 1, MPI_INT …)
*buf = 4; /*Not certain if receiver
gets 3 or 4 */
MPI_Wait(…);
```

- 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

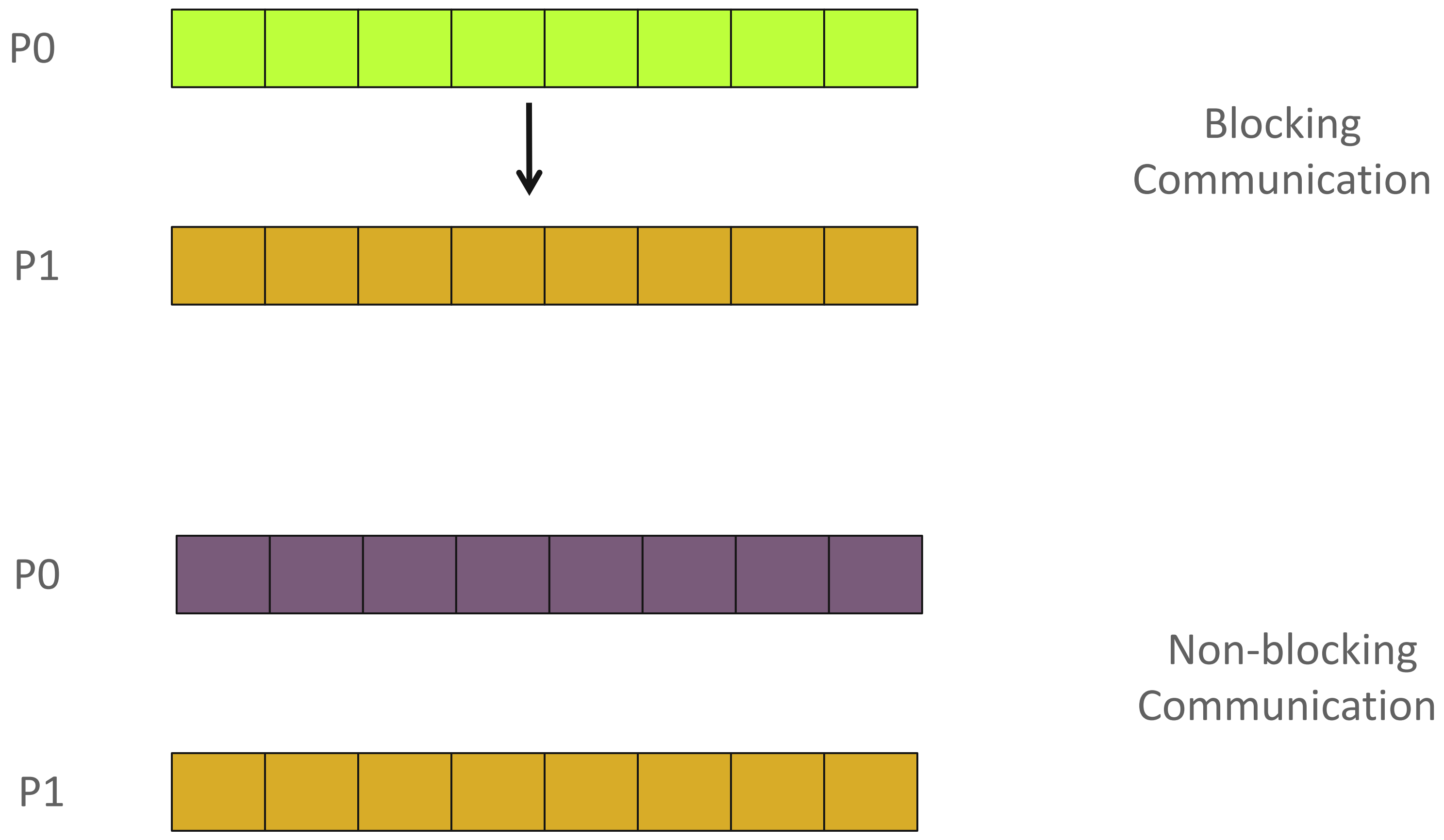
A Non-Blocking communication example



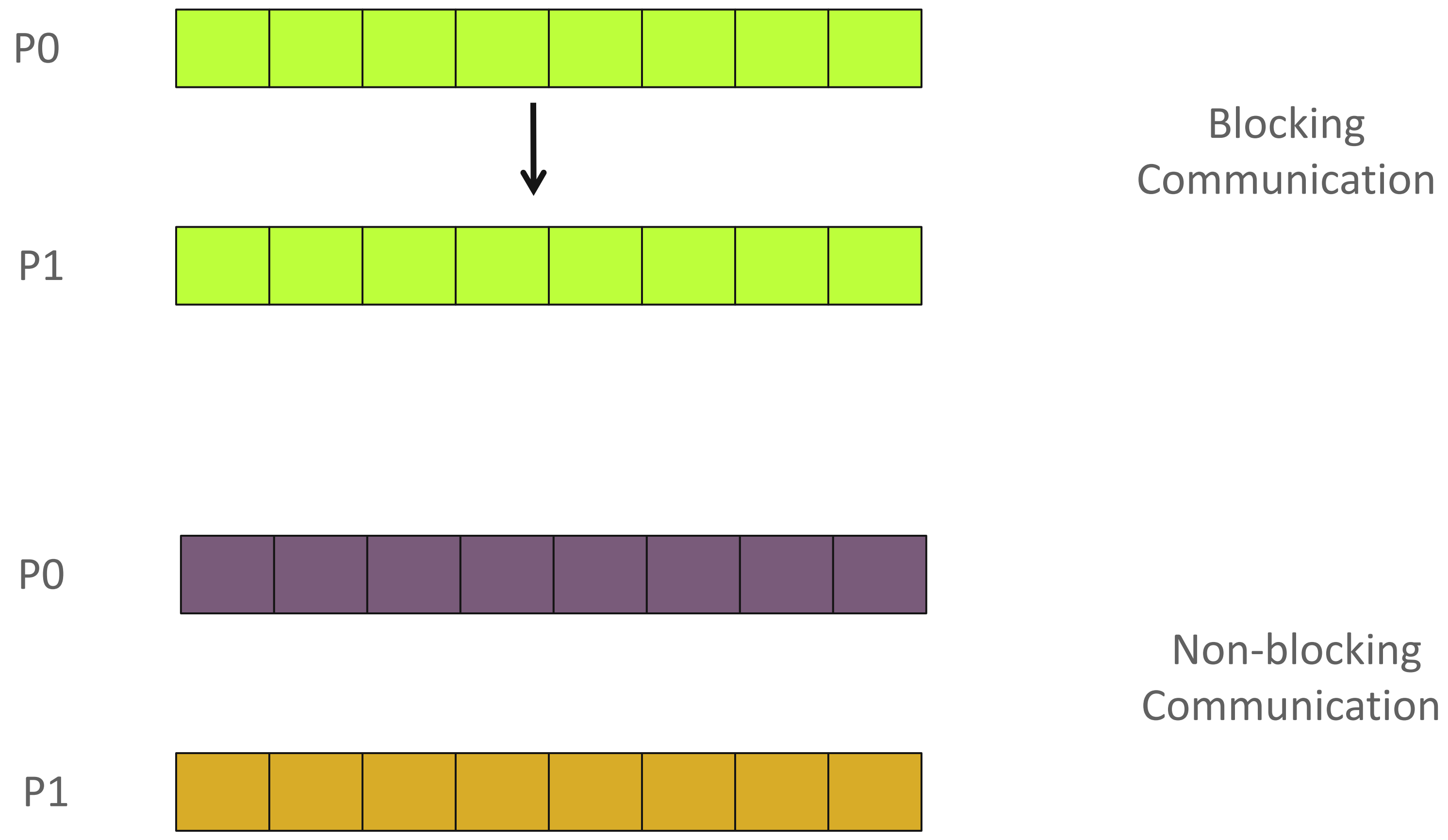
A Non-Blocking communication example



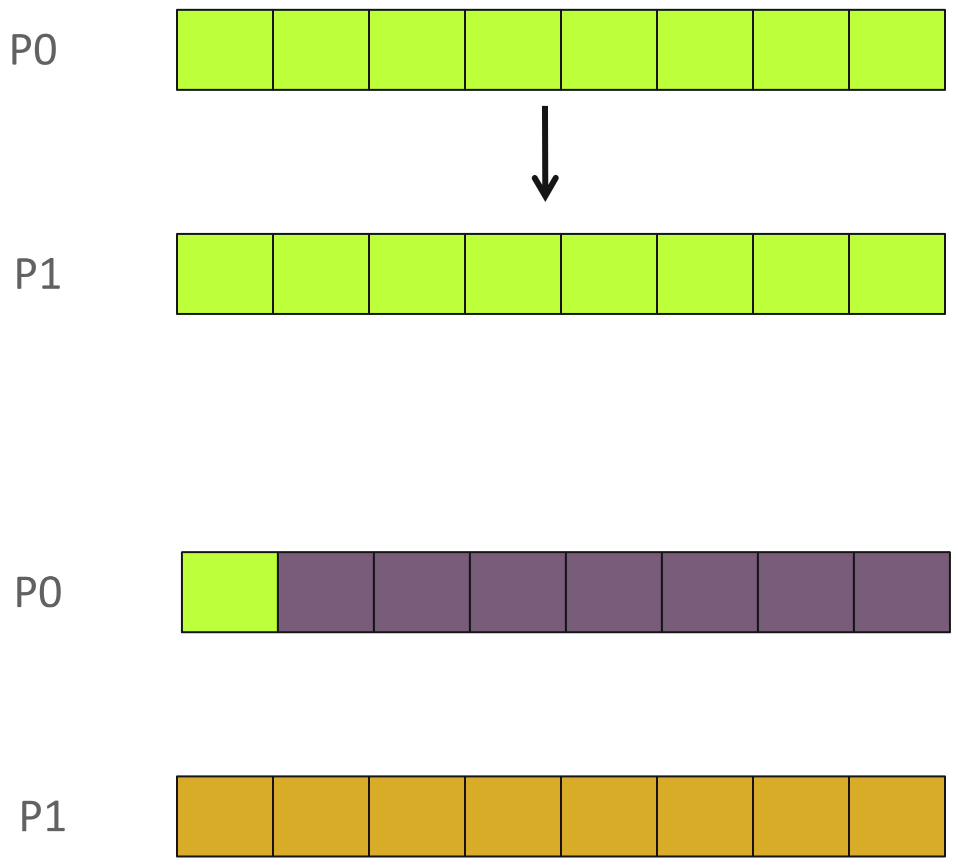
A Non-Blocking communication example



A Non-Blocking communication example



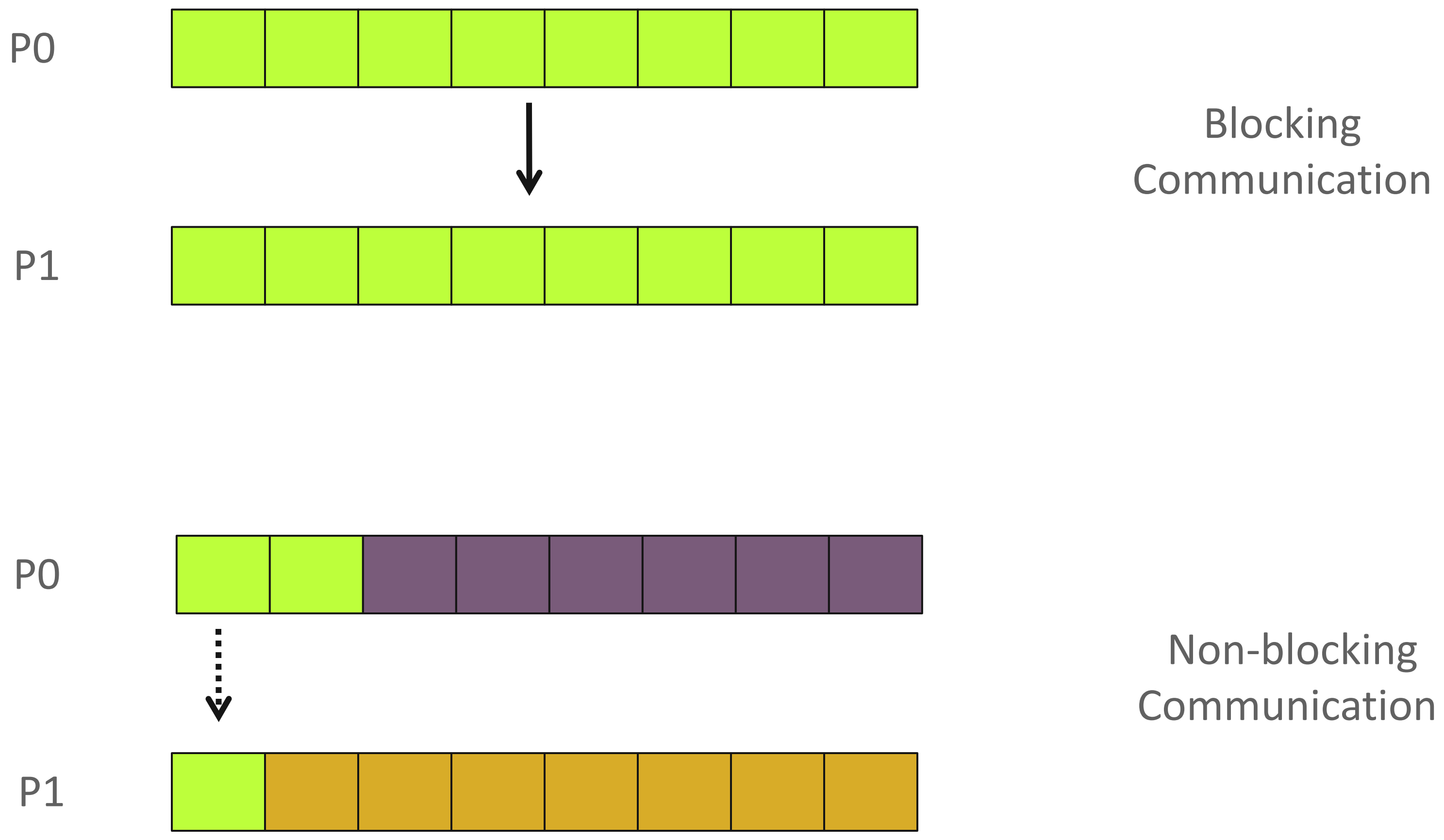
A Non-Blocking communication example



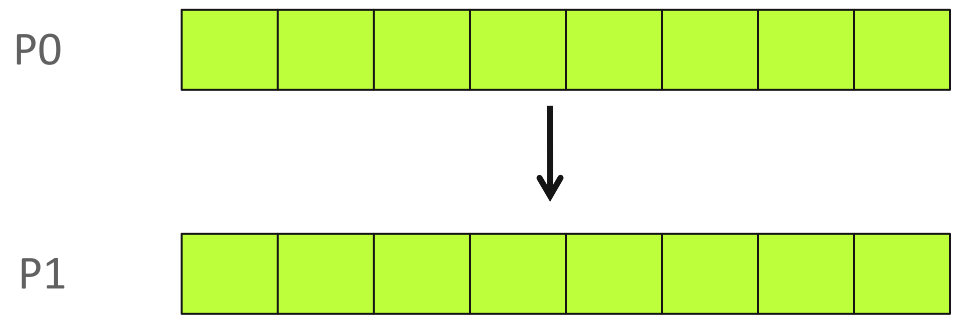
Blocking Communication

Non-blocking Communication

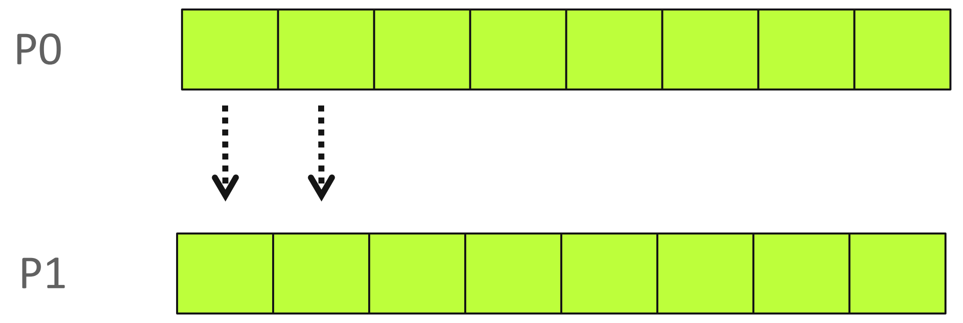
A Non-Blocking communication example



A Non-Blocking communication example



Blocking Communication



Non-blocking Communication

A Non-Blocking communication example (contd.)

```
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...]
}
```

What we will cover in this tutorial

- What is MPI?
- Fundamental concepts in MPI
- Point-to-point communication in MPI
- **Group (collective) communication in MPI**
- A sneak peak at MPI-3
- Conclusions and Final Q/A

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 communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

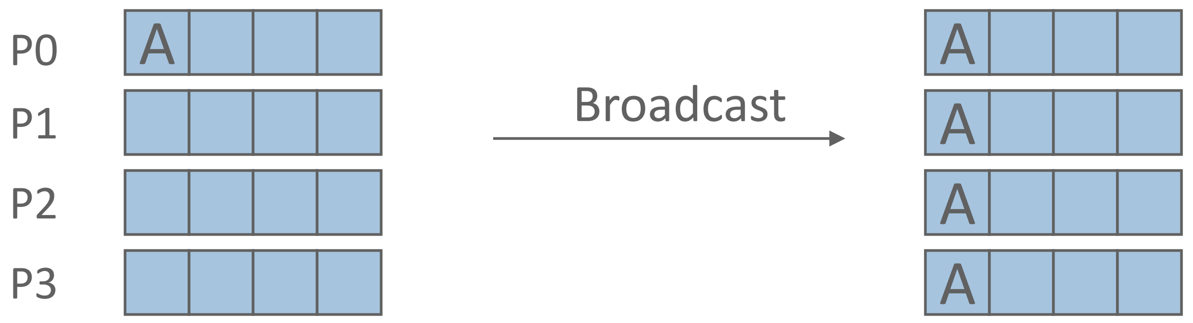
MPI Collective Communication

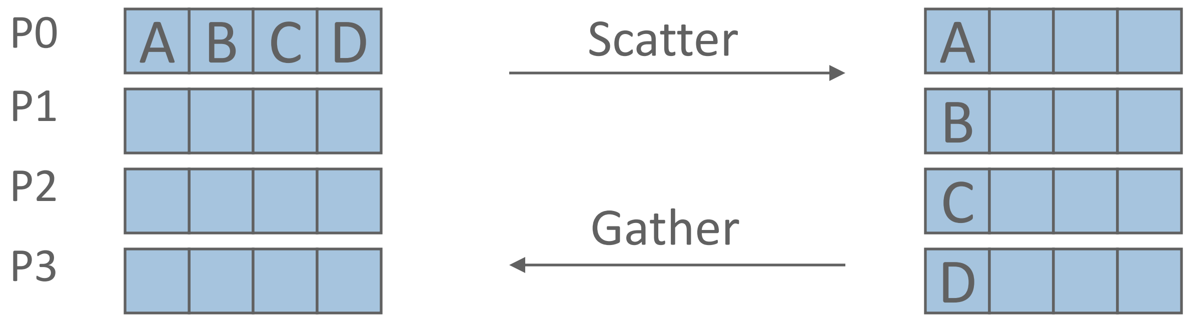
- Communication and computation is coordinated among a group of processes in a communicator
- Tags are not used; different communicators deliver similar functionality
- No non-blocking collective operations
  - (they are being added in MPI-3)
- Three classes of operations: synchronization, data movement, collective computation

Synchronization

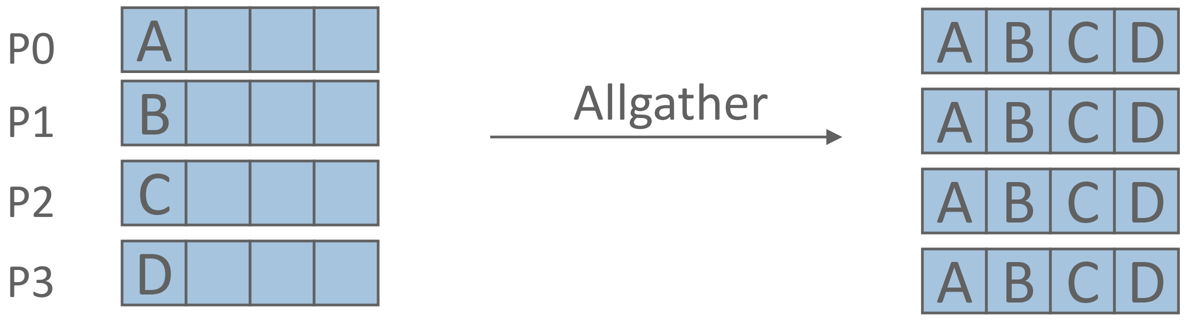
- `MPI_Barrier(comm)`
- Blocks until all processes in the group of the communicator `comm` call it
- A process cannot get out of the barrier until all other processes have reached barrier

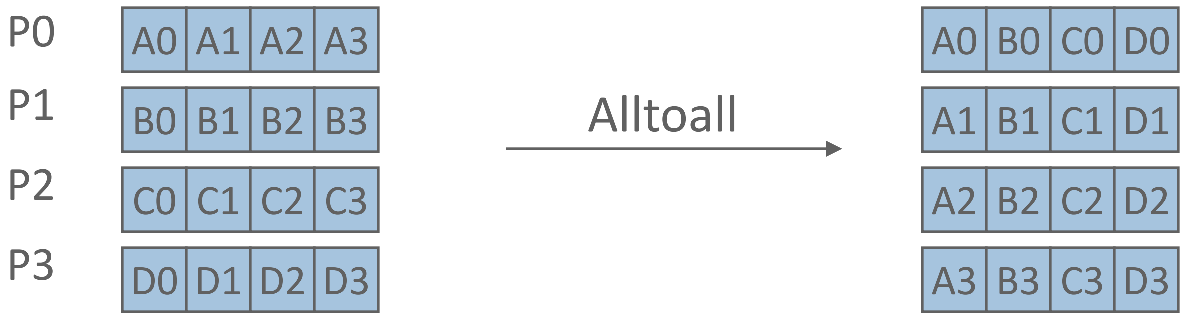
Collective Data Movement



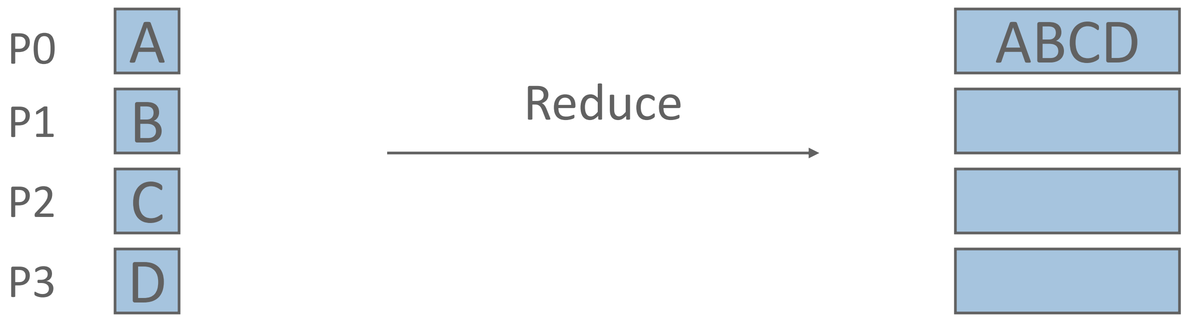


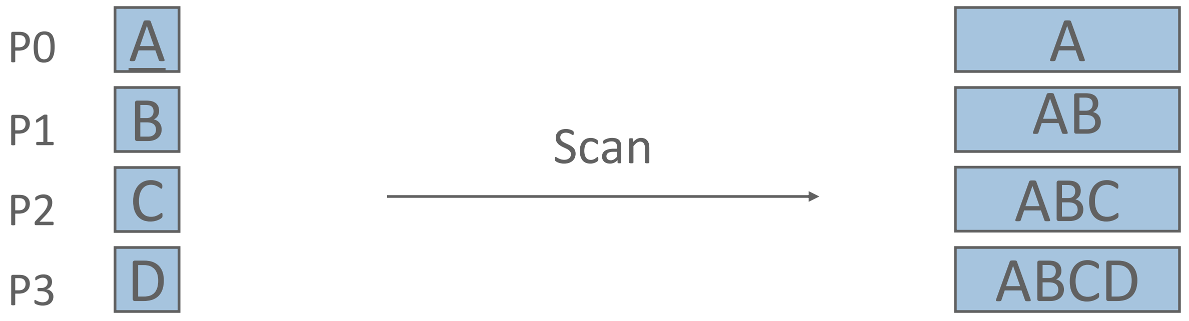
More Collective Data Movement





Collective Computation





MPI Collective Routines

- Many Routines: **`Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, ReduceScatter, Scan, Scatter, Scatterv`**
- **`“All”`** versions deliver results to all participating processes.
- **`“V”`** versions (stands for vector) allow the hunks to have different sizes.
- **`Allreduce`**, **`Reduce`**, **`ReduceScatter`**, and **`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:

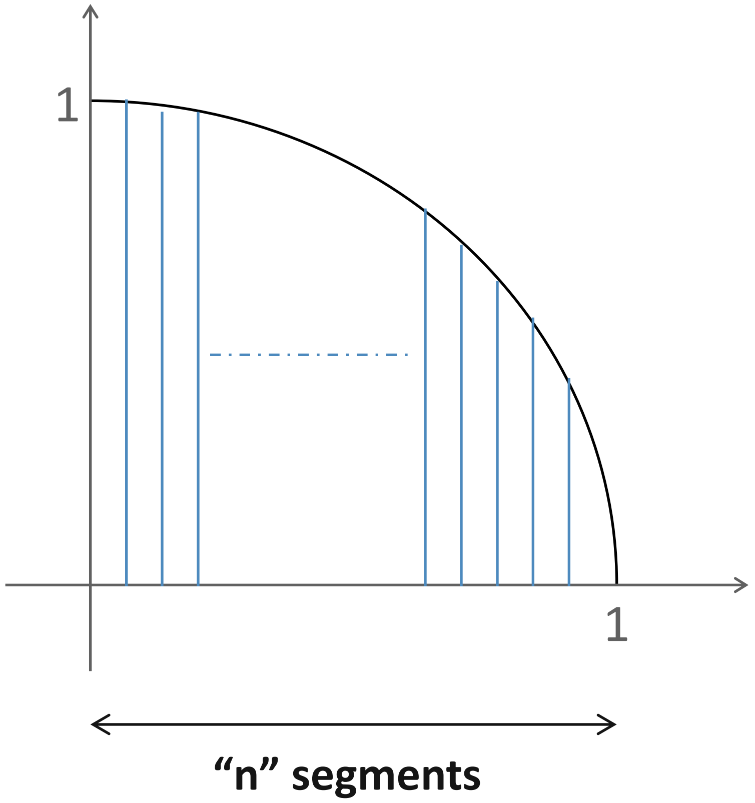
```
inoutvec[i]  =  invec[i]  op  inoutvec[i];
```

  for i from 0 to len-1

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

Example: Calculating Pi

- Calculating pi via numerical integration
  - Divide interval up into subintervals
  - Assign subintervals to processes
  - Each process calculates partial sum
  - Add all the partial sums together to get pi



1. Width of each segment (w) will be 1/n
2. Distance (d(i)) of segment "i" from the origin will be "i * w"
3. Height of segment "i" will be sqrt(1 – d(i))

Example: PI in C

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
    [...snip...]
    /* Tell all processes, the number of segments you want */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    w   = 1.0 / (double) n;
    mypi = 0.0;
    for (i = myid + 1; i <= n; i += numprocs)
        mypi += sqrt(1 - (w * i));
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n", pi,
               fabs(pi - PI25DT));
    [...snip...]
}
```

What we will cover in this tutorial

- What is MPI?
- Fundamental concepts in MPI
- Point-to-point communication in MPI
- Group (collective) communication in MPI
- **A sneak peak at MPI-3**
- Conclusions and Final Q/A

Ongoing effort in multiple areas

- Collective Communication
  - Non-blocking collective operations
  - Sparse collective operations
- Remote Memory Access
  - Improvements Get/Put model
- Hybrid Programming
  - Improved interoperability with threads
  - Interoperability with shared memory programming
  - Interoperability with PGAS models
- Fault Tolerance

What we will cover in this tutorial

- What is MPI?
- Fundamental concepts in MPI
- Point-to-point communication in MPI
- Group (collective) communication in MPI
- A sneak peak at MPI-3
- **Conclusions and Final Q/A**

Conclusions

- Parallelism is critical today, given that that is the only way to achieve performance improvement with the modern hardware
- MPI is an industry standard model for parallel programming
  - A large number of implementations of MPI exist (both commercial and public domain)
  - Virtually every system in the world supports MPI
- Gives user explicit control on data management
- Widely used by many many scientific applications with great success
- Your application can be next!

Web Pointers

- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPICH2 : http://www.mcs.anl.gov/research/projects/mpich2/
- MPICH mailing list: mpich-discuss@mcs.anl.gov
- MPI Forum : http://www.mpi-forum.org/

- Other MPI implementations:
  - MVAPICH2 (MPICH on InfiniBand) : http://mvapich.cse.ohio-state.edu/
  - Intel MPI (MPICH derivative): http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI (MPICH derivative)
  - Open MPI : http://www.open-mpi.org/

- Several MPI tutorials can be found on the web