Argonne NATIONAL LABORATORY

Parallel Programming with MPI

Download Slides and Examples at https://anl.box.com/v/2021-IIT-MPI-Lecture

Yanfei Guo

Argonne National Laboratory

Email: yguo@anl.gov

U.S. DEPARTMENT OF ENERGY

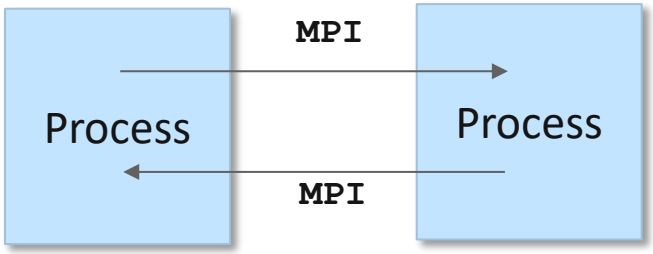
Introduction to MPI

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, hex-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 everywhere!
    - By USPS for tracking and minimizing fuel routes
    - By automobile companies for car crash simulations
    - By airline industry to build newer models of flights

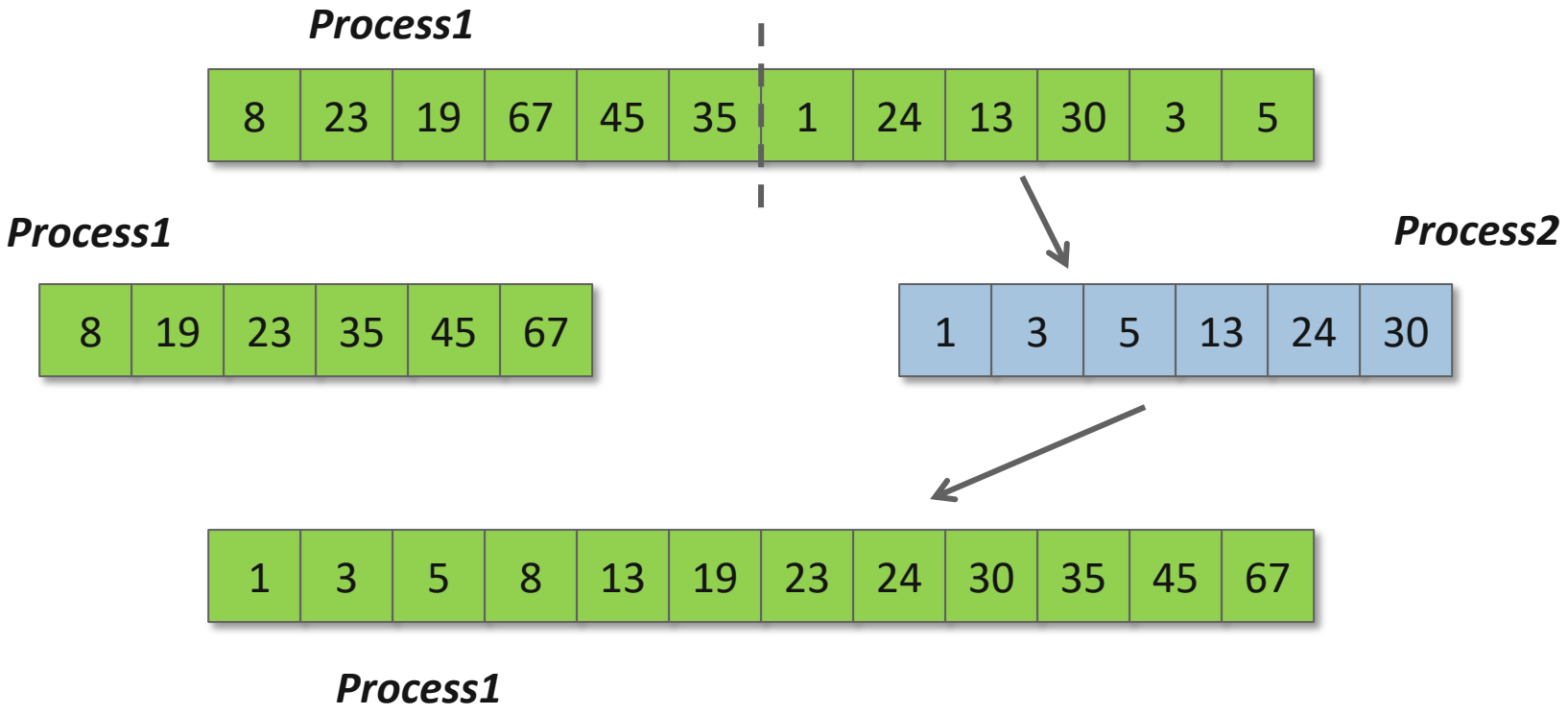
The Message-Passing Model

- 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



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

From MPI-1 to MPI-4.0

- MPI-1 (1994) supports the classical message-passing programming model:
  - Basic point-to-point communication
  - Collectives
  - Datatypes
- MPI-2 (1997) extended the message-passing model
  - Parallel I/O
  - Remote memory operations (one-sided, RMA)
  - Dynamic process management
  - Interaction with threads
- MPI-3, 3.1 (2012, 2015) added several new features to MPI
  - Nonblocking collectives
  - Neighborhood collectives
  - Improved one-sided communication interface
  - Nonblocking collective I/O functions (MPI-3.1)
- MPI-4 (2021) is the latest version

MPI-4.0

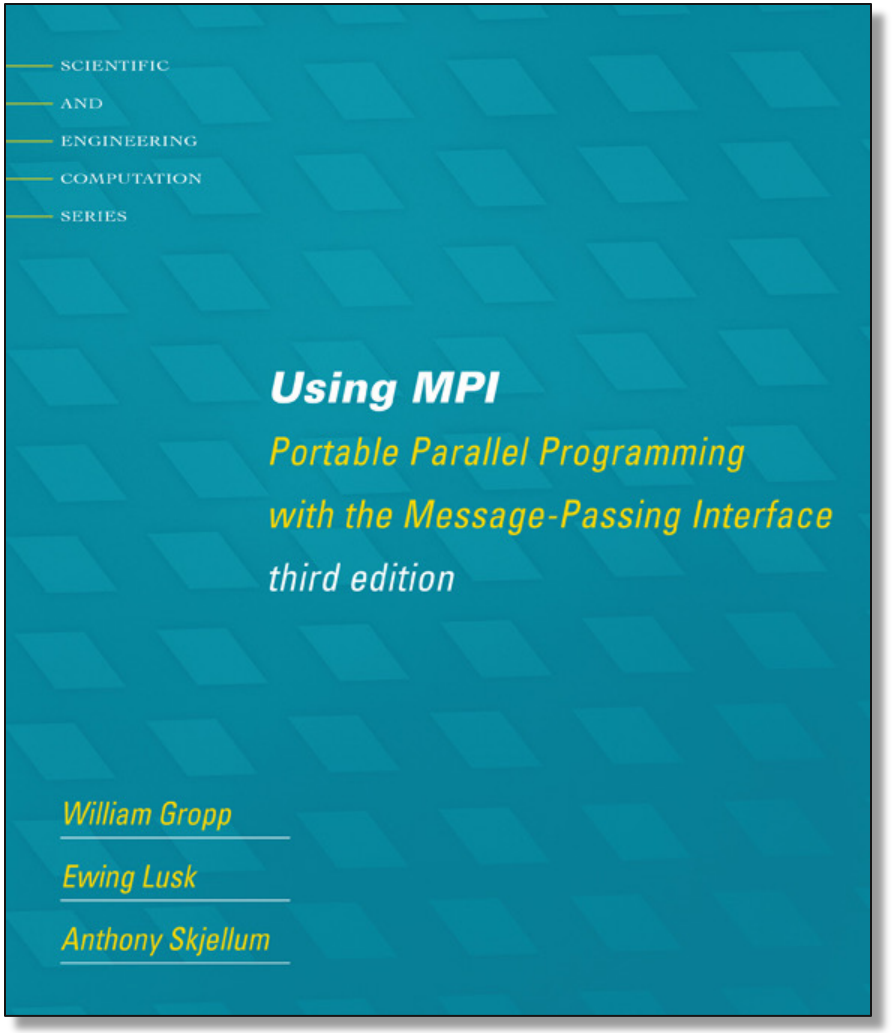
- Major new features and changes
  - Persistent Collectives
  - Partitioned Communication
  - Sessions
  - Big Count
  - Error Handling Improvement
  - Topology Improvement

Web Pointers

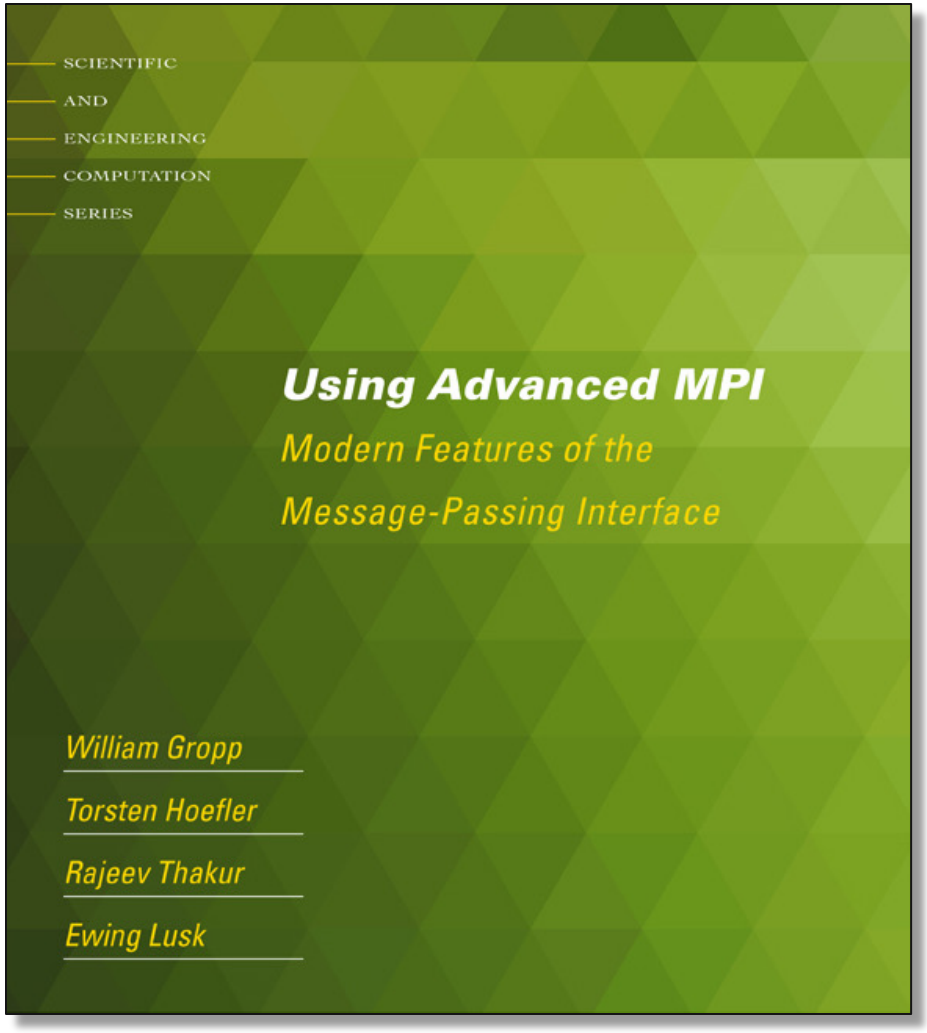
- MPI Standard : https://www.mpi-forum.org/docs/
- MPI implementations:
  - MPICH : http://www.mpich.org
  - MVAPICH : http://mvapich.cse.ohio-state.edu/
  - Intel MPI: http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI: https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
  - Open MPI : http://www.open-mpi.org/
  - IBM MPI, Cray MPI, …
- Several MPI tutorials can be found on the web

  ANL MPI Tutorial Video https://www.youtube.com/watch?v=SGTfkQr5RS8

Tutorial Books on MPI



Basic MPI



Advanced MPI, including MPI-3

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 is a popular open-source and free implementation 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

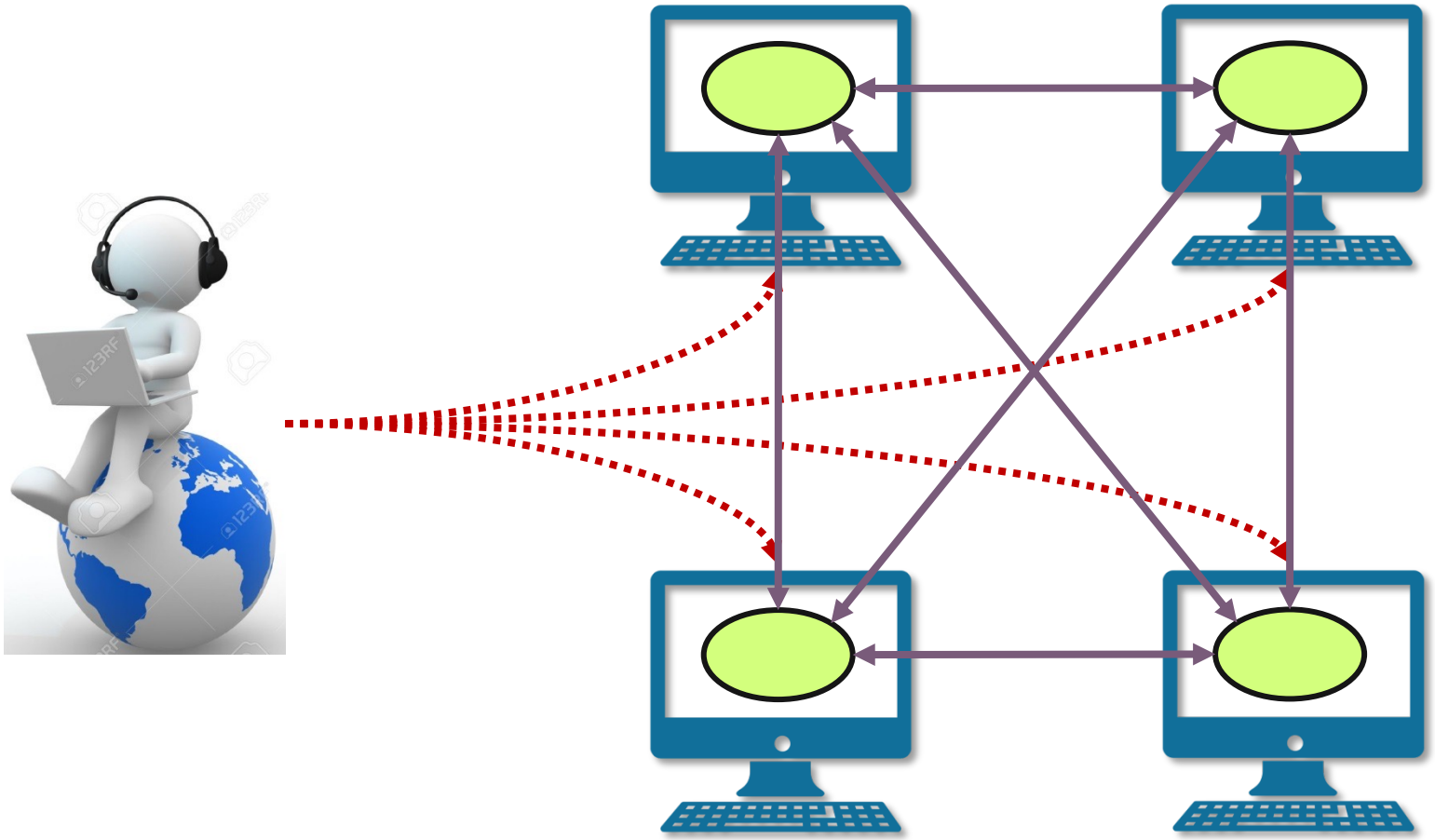
Important considerations while using MPI

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

Outline

- Installing and running MPI
- MPI Communicators
- Blocking Point-to-Point Operations
- Nonblocking Point-to-Point Operations
- Running example: 2D stencil code
- Collective Communication

How does MPI work? (1/2)



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 -n 16 ./test
```

What is MPICH?

- MPICH is a high-performance and widely portable open-source implementation of MPI
- It provides all features of MPI that have been defined so far (including MPI-1, MPI-2.0, MPI-2.1, MPI-2.2, MPI-3.0 and MPI-3.1)
- Active development led by Argonne National Laboratory
  - Several close collaborators who contribute many features, bug fixes, testing for quality assurance, etc.
    - Intel, Cray, Mellanox, Ohio State University, Microsoft, IBM, and many others

Getting Started with MPICH

- From UNIX distributions
  - Most UNIX/Linux distributions package MPICH for easy installation
  - apt-get (Ubuntu/Debian), yum (Fedora, Centos), brew/port (Mac OS)
- Open-source for source-based installation
  - Download MPICH
    - Go to http://www.mpich.org and follow the downloads link
    - The download will be a zipped tarball
  - Build MPICH
    - Unzip/untar the tarball

```
% tar xvfz mpich-<latest_version>.tar.gz
% cd mpich-<latest_version>
% ./configure --prefix=/where/to/install/mpich |& tee c.log
% make |& tee m.log
% make install |& tee mi.log
% Add /where/to/install/mpich/bin to your PATH
```

Setup MPICH Environment (cont'd)

- Linux: Use Docker Image
  1. `mkdir $HOME/mpi-tutorial`
  2. `docker pull pmrs/mpi-tutorial`
  3. `docker run --rm -it -v $HOME/mpi-tutorial:/project pmrs/mpi-tutorial`

     This creates a container and opens a shell.
     It mounts local directory $HOME/mpi-tutorial as /project in the container.

  4. `(in container) cp -r $HOME/examples /project/examples`
- Windows: Use Windows Subsystem for Linux (WSL)

Compiling MPI programs with MPICH

- Compilation Wrappers
  - For C programs:
  - For C++ programs:
  - For Fortran programs:

```
% mpicc test.c -o test
```

```
% mpicxx test.cpp -o test
```

```
% mpifort 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 -n 16 ./test
```

- Launch 16 processes on 4 nodes (each has 4 cores)

```
% mpiexec -hosts h1:4,h2:4,h3:4,h4:4 -n 16 ./test
```

  - Runs the first four processes on h1, the next four on h2, etc.

- If there are many nodes, it might be easier to create a host file

```
% cat hf
      h1:4
      h2:2
% mpiexec -hostfile hf -n 16 ./test
```

Trying some example programs

- MPICH comes packaged with several example programs using almost ALL of MPICH's functionality
- A simple program to try out is the PI example written in C (cpi.c) – calculates the value of PI in parallel (available in the examples directory when you build MPICH)

```
% mpiexec -n 16 ./examples/cpi
```

- The output will show how many processes are running, and the error in calculating PI
- Next, try it with multiple hosts

```
% mpiexec -hosts h1:2,h2:4 -n 16 ./examples/cpi
```

- If things don't work as expected, send an email to discuss@mpich.org

MPI Communicators

Download Slides and Examples at https://anl.box.com/v/2021-IIT-MPI-Lecture

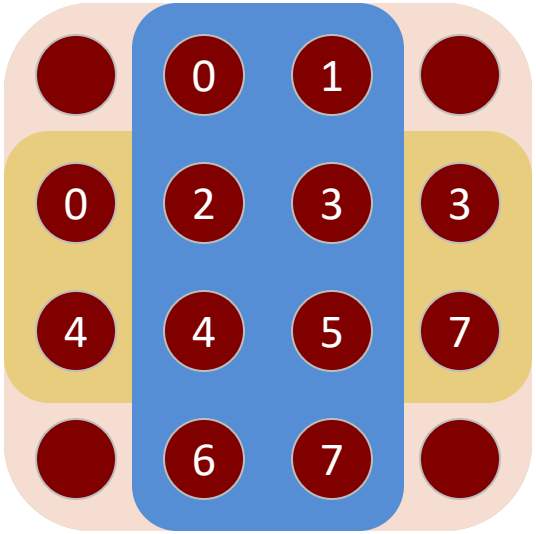
MPI Communicators

- MPI processes can be collected into groups
  - Each group can have multiple colors (some times called context)
  - *Group + color == communicator (it is like a name for the group)*
  - 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 -n 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 a "rank"

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

The same process might have different ranks in different communicators

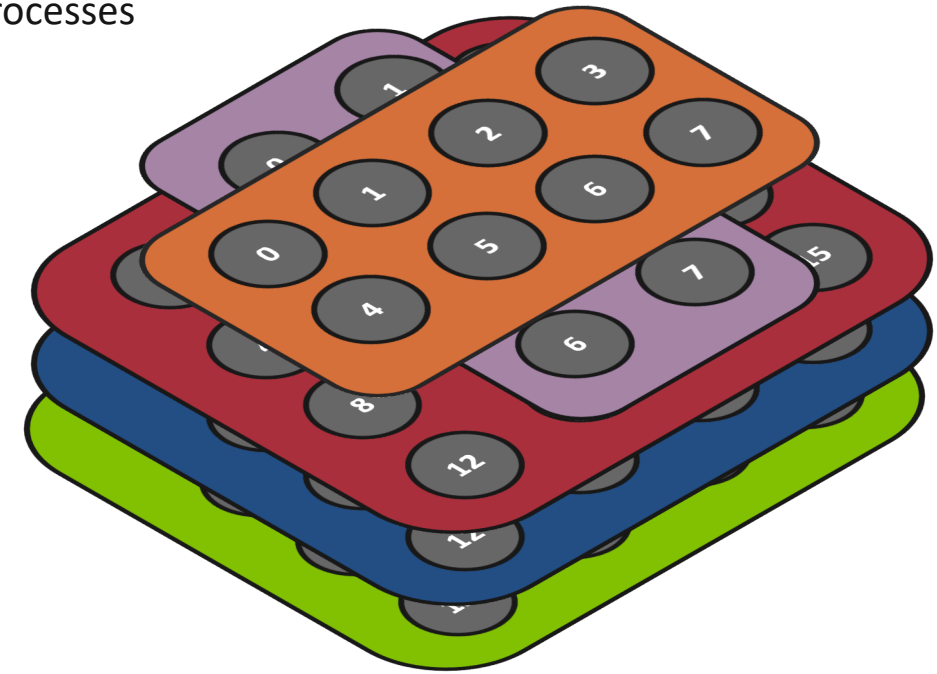
Communicators can be created "by hand" or using tools provided by MPI (not discussed in this tutorial)

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

Communicators

Can be thought of as independent communication layers over a group of processes

Messages in one layer will not affect messages in another



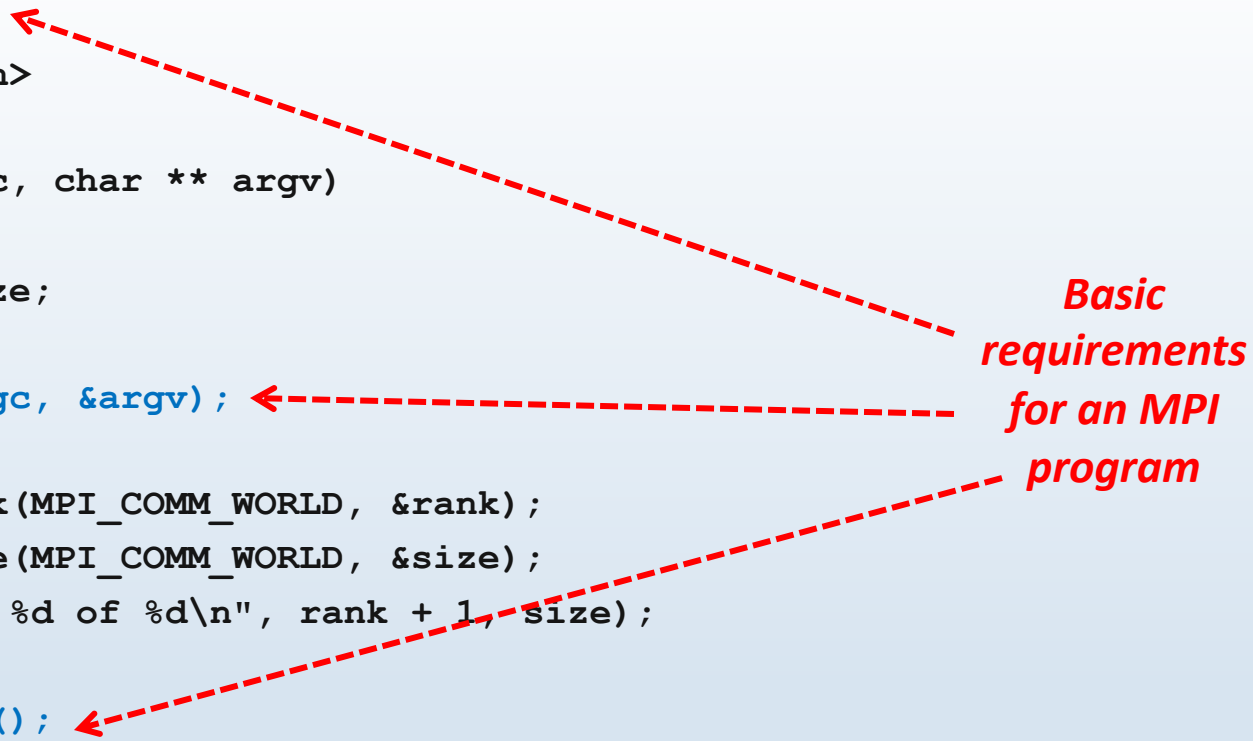
Example: Hello World!

- *communicators/hello.c*
- Basic program where each process prints its rank

Download Examples at https://anl.box.com/v/2019-iit-mpi-lecture

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 + 1, size);

    MPI_Finalize();
    return 0;
}
```

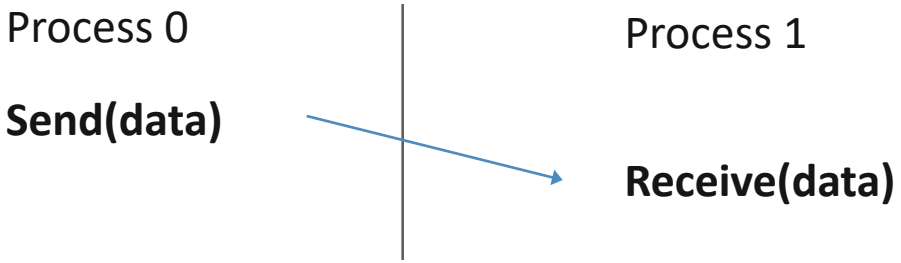


Blocking Point-to-Point Operations

Download Slides and Examples at https://anl.box.com/v/2021-IIT-MPI-Lecture

MPI Basic Send/Receive

- Simple communication model



- 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

MPI Basic (Blocking) Send

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

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 → 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

MPI Basic (Blocking) Receive

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

Example: Basic Send/Receive

- *blocking_p2p/sendrecv.c*
- Simple send/receive program to show basic data transfer

Download Examples at https://anl.box.com/v/2019-iit-mpi-lecture

Simple Communication in MPI

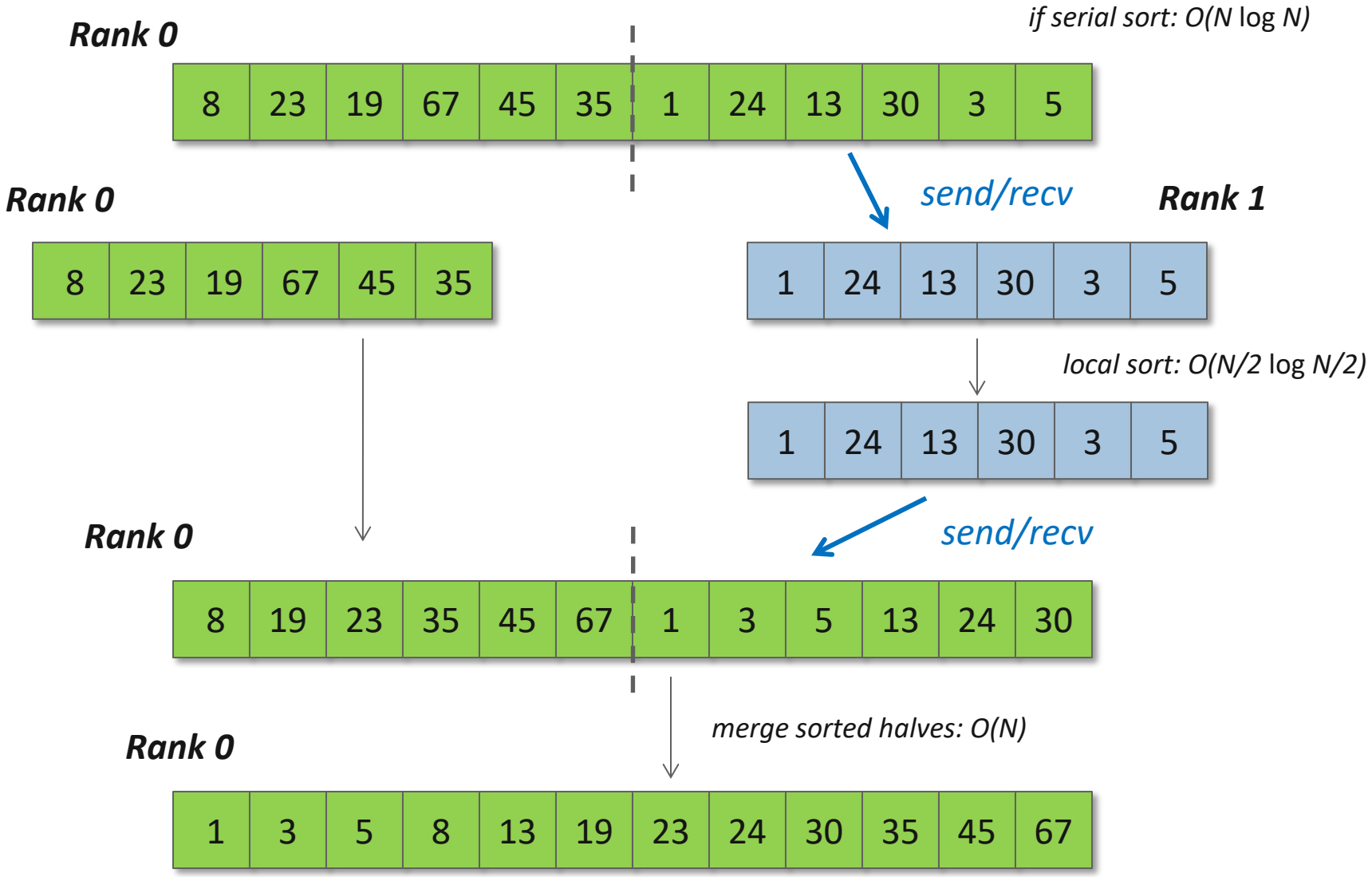
```
#include <mpi.h>
#include <stdio.h>

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 /*dest*/, 0 /*tag*/, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 100, MPI_INT, 0 /*src*/, 0 /*tag*/, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    MPI_Finalize();
    return 0;
}
```

Example: Sorting with Two Processes

- *blocking_p2p/sort_2_procs.c*
- Sorting using two processes

Download Examples at https://anl.box.com/v/2019-iit-mpi-lecture

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, 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, MPI_STATUS_IGNORE);

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

    MPI_Finalize(); return 0;
}
```

Section Summary

- MPI is simple
- Many (if not all) parallel programs can be written using just these six functions
  - **`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`** – receive 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

Nonblocking Point-to-Point Operations

Download Slides and Examples at https://anl.box.com/v/2021-IIT-MPI-Lecture

Blocking vs. Nonblocking 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 nonblocking 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

- Blocking communication is simple to use but can be prone to **deadlocks**

```
if (rank == 0) {
   MPI_SEND(..to rank 1..)
   MPI_RECV(..from rank 1..)
else if (rank == 1) {
   MPI_SEND(..to rank 0..)
   MPI_RECV(..from rank 0..)
}
```

Fix: reverse send/recv

```
if (rank == 0) {
   MPI_SEND(..to rank 1..)
   MPI_RECV(..from rank 1..)
else if (rank == 1) {
   MPI_RECV(..from rank 0..)
   MPI_SEND(..to rank 0..)
}
```

Nonblocking Communication

- Nonblocking operations return (immediately) "request handles" that can be waited on and queried

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
MPI_IRECV(buf, count, datatype, src, tag, comm, request)
MPI_WAIT(request, status)
```

- Nonblocking 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**

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(incount, array_of_requests, outcount,
                array_of_indices, array_of_statuses)
```

- There are corresponding versions of **`TEST`** for each of these

Message Completion and Buffering

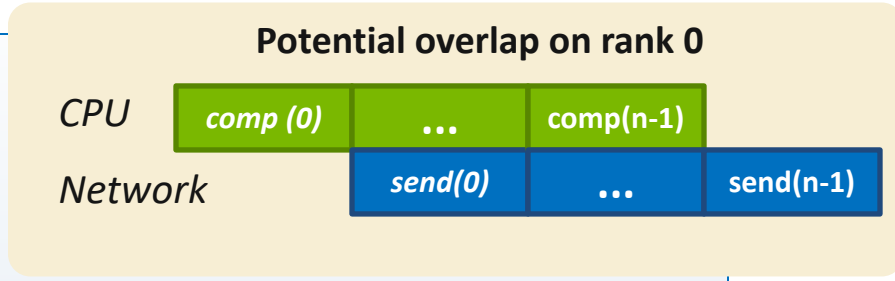
- For a communication to succeed:
  - Sender must specify a valid destination rank
  - Receiver must specify a valid source rank (including **MPI_ANY_SOURCE**)
  - The communicator must be the same
  - Tags must match
  - Receiver's buffer must be large enough
- A send has completed when the user supplied buffer can be reused
- Send completes does not mean that the receive has completed
  - Message may be buffered by the system
  - Message may still be in transit

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

```
*buf = 3;
MPI_Isend(buf, 1, MPI_INT …, req)
*buf = 4; /* Receiver may get 3, 4, or
             anything else */
MPI_Wait(req, status);
```

A Nonblocking 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 if (rank == 1){
        for (i = 0; i < 100; i++)
            MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
    }
    [...snip...]
}
```



Section Summary

- Nonblocking communication is an enhancement over blocking communication
- Allows for computation and communication to be potentially overlapped
  - MPI implementation might, but is not guaranteed to overlap
  - Depends on what capabilities the network provides
  - Depends on how the MPI library is implemented (e.g., some libraries might tradeoff between better overlap and better basic performance)

Running Example: Stencil

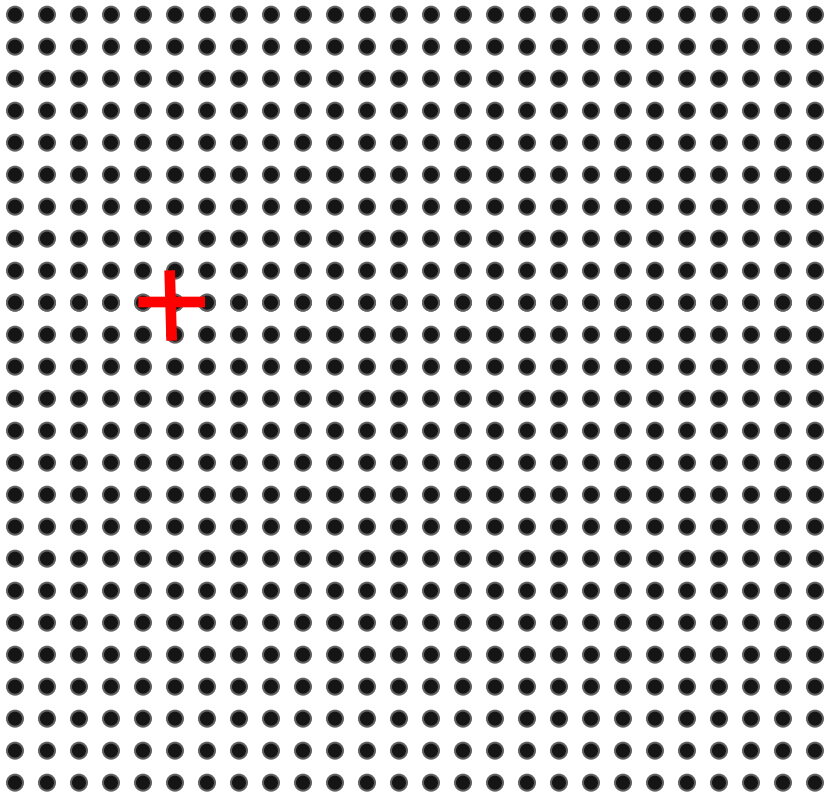
Download Slides and Examples at https://anl.box.com/v/2021-IIT-MPI-Lecture

Running Example: Regular Mesh Algorithms

- Many scientific applications involve the solution of partial differential equations (PDEs)
- Many algorithms for approximating the solution of PDEs rely on forming a set of difference equations
  - Finite difference, finite elements, finite volume
- The exact form of the differential equations depends on the particular method
  - From the point of view of parallel programming for these algorithms, the operations are the same
- Five-point stencil is a popular approximation solution

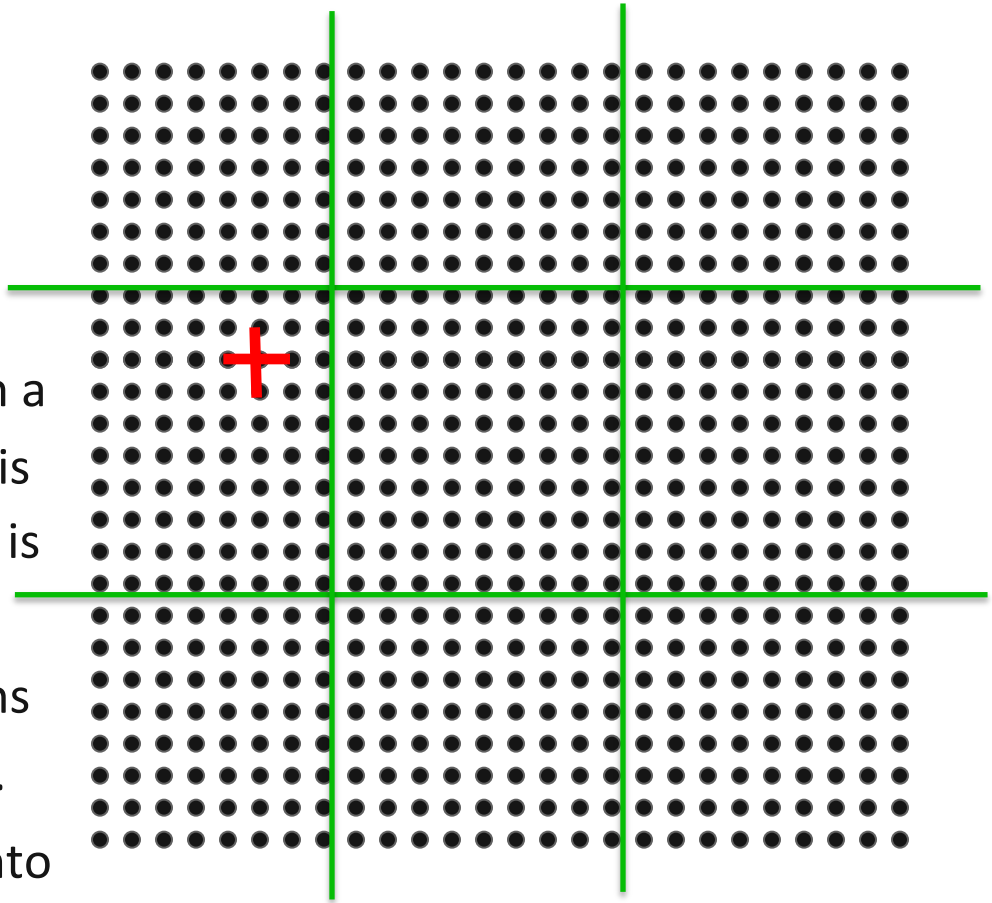
The Global Data Structure

- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.

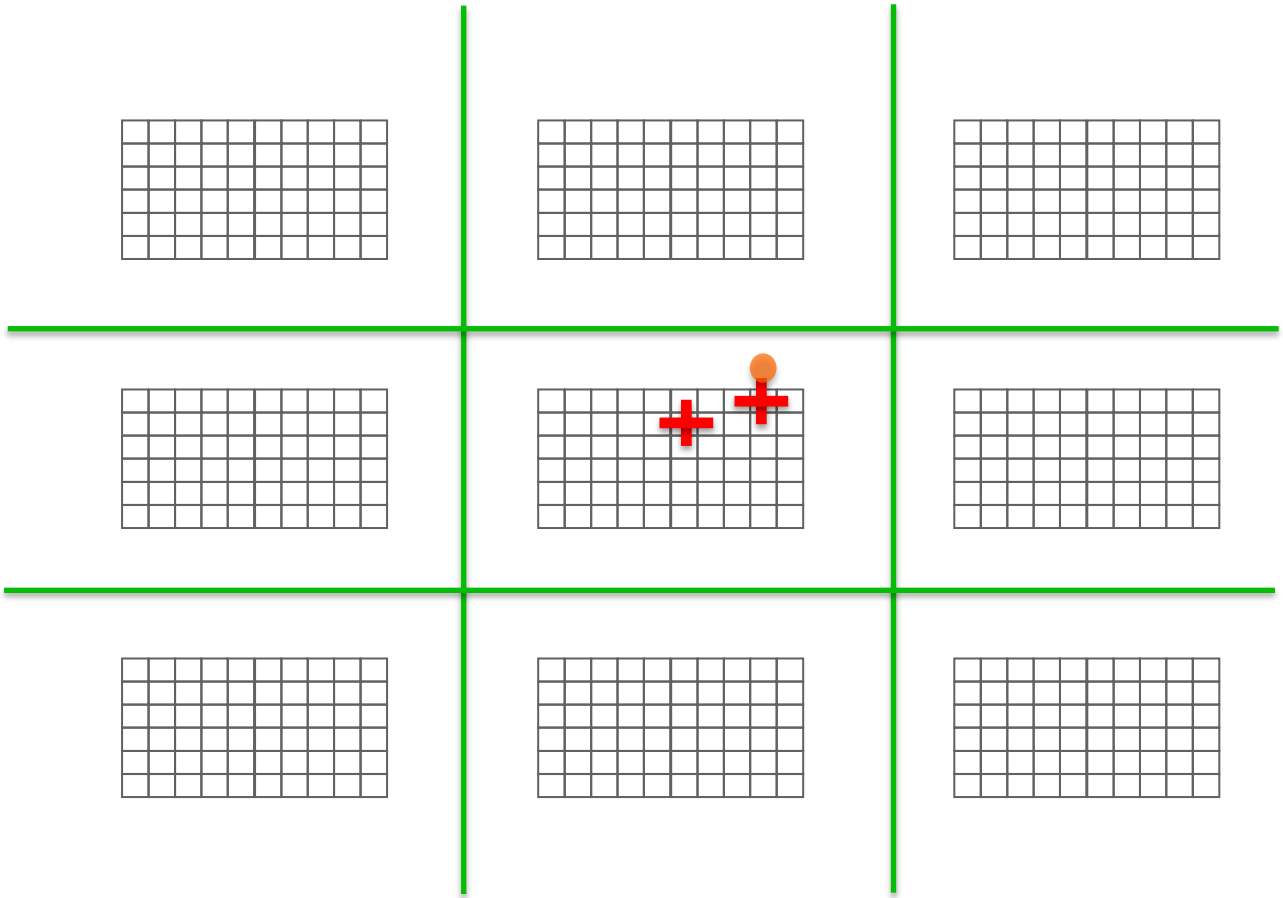


The Global Data Structure

- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.
- **Parallelism:** decompose mesh into equal sized (work) pieces

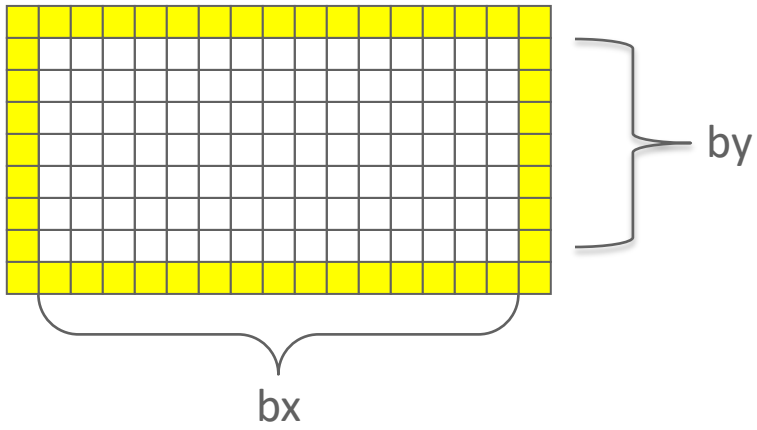


Necessary Data Transfers

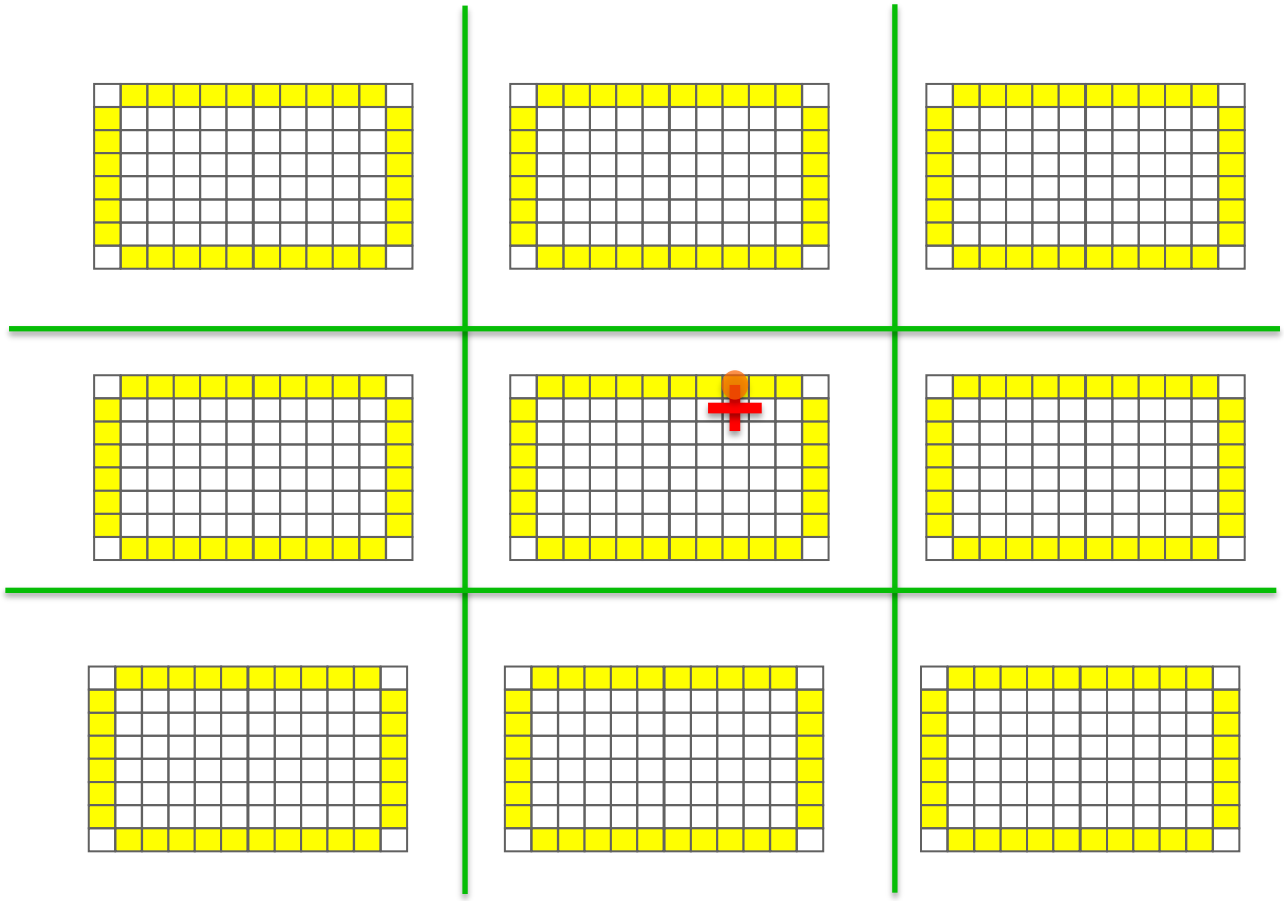


The Local Data Structure

- Each process has its local “patch” of the global array
  - “bx” and “by” are the sizes of the local array
  - Always allocate a halo around the patch
  - Array allocated of size (bx+2)x(by+2)

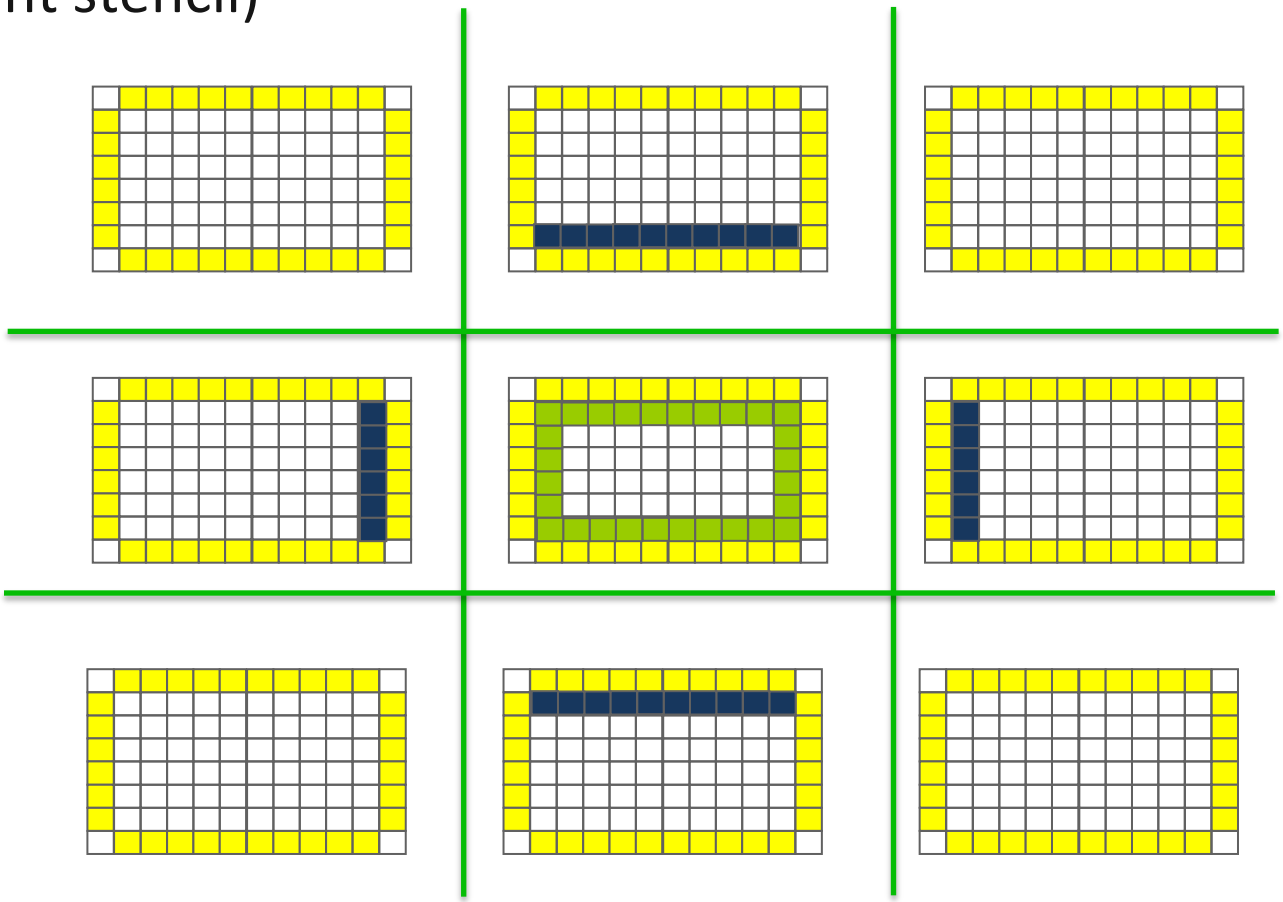


Necessary Data Transfers



Necessary Data Transfers

- Provide access to remote data through a halo exchange (5 point stencil)



Example: Stencil with Nonblocking Send/recv

- *nonblocking_p2p/stencil.c*
- Simple stencil code using nonblocking point-to-point operations

- Usage:
  - mpiexec –n <nproc> ./stencil <n> <energy> <niters> <px> <py>
- Example input:
  - mpiexec –n 8 ./stencil 100 10 1000 4 2

Download Examples at https://anl.box.com/v/2021-IIT-MPI-Lecture

Blocking Collective Operations

Download Slides and Examples at https://anl.box.com/v/2021-IIT-MPI-Lecture

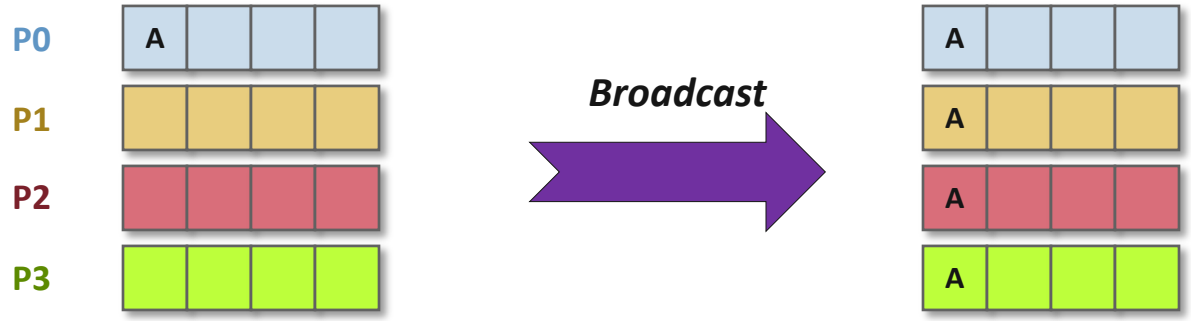
Introduction to Collective Operations in MPI

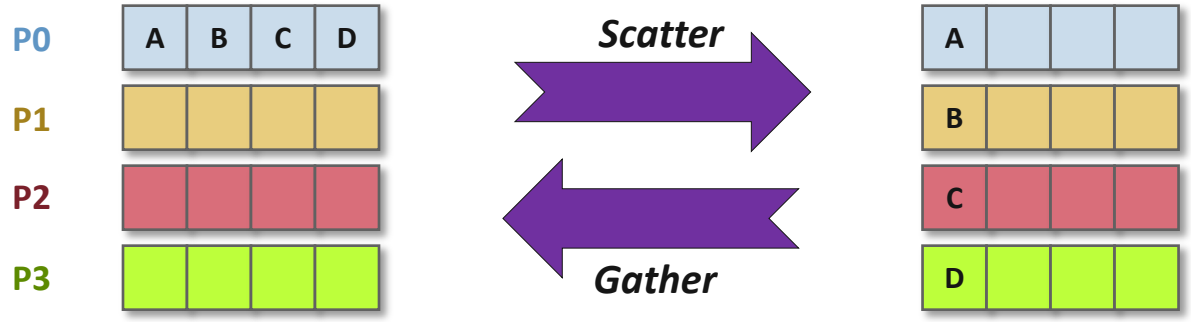
- Collective operations are called by all processes in a communicator.
- In many numerical algorithms, **`SEND/RECV`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.
  - **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
  -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

Synchronization

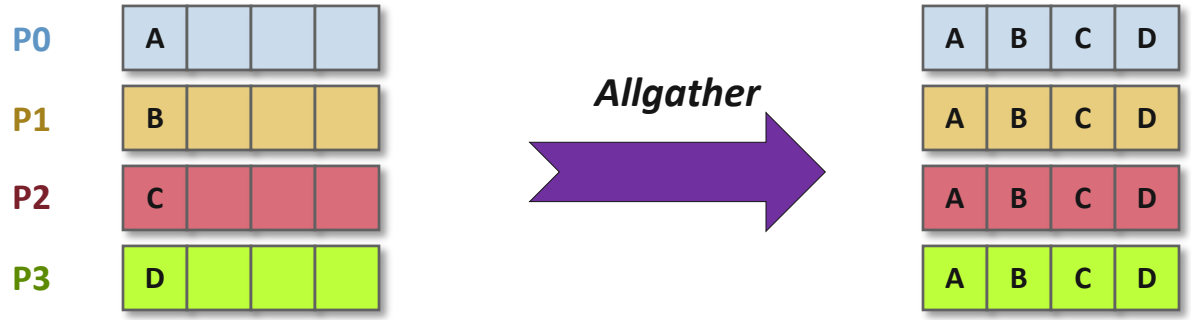
- **`MPI_BARRIER(comm)`**
  - Blocks until all processes in the group of the communicator **`comm`** call it
  - A process cannot get out of the barrier until all other processes have reached barrier

Collective Data Movement

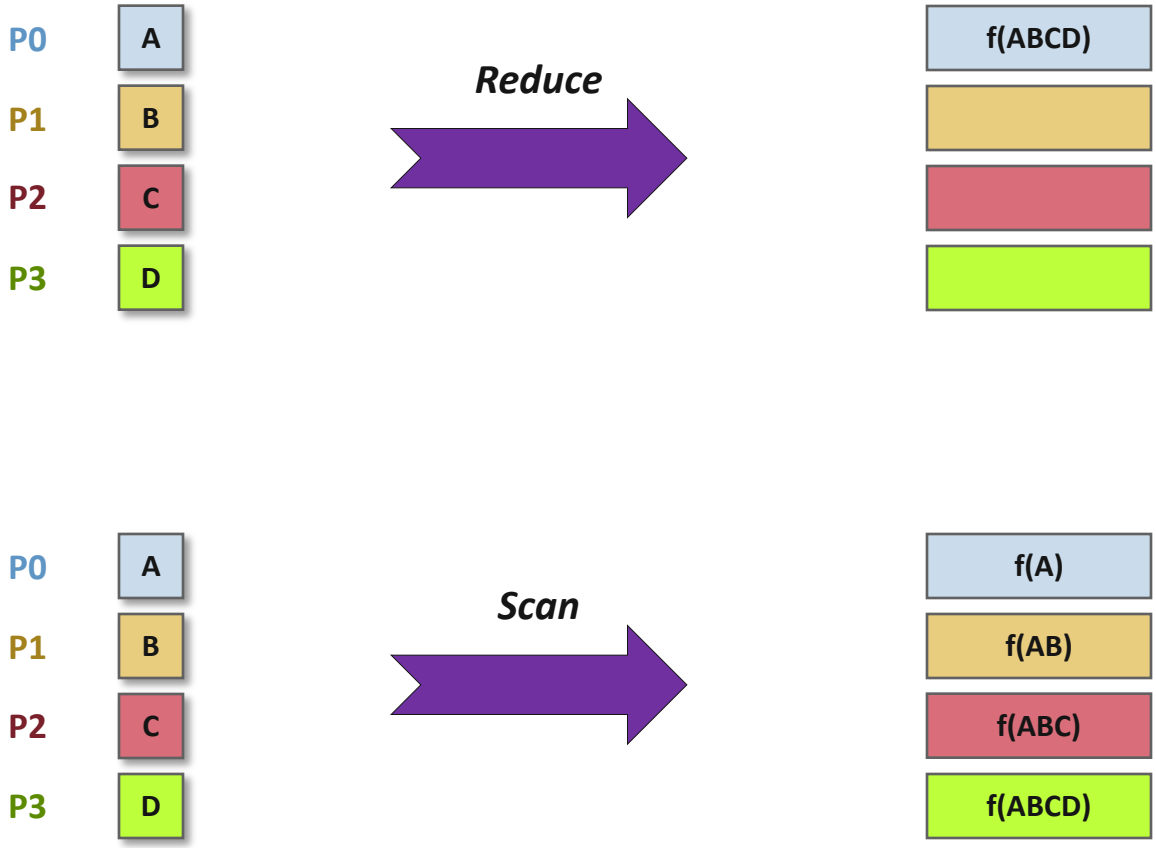




More Collective Data Movement



Collective Computation



MPI Collective Routines

- Many Routines: **MPI_ALLGATHER**, **MPI_ALLGATHER*V***, **MPI_ALLREDUCE**, **MPI_ALLTOALL**, **MPI_ALLTOALL*V***, **MPI_BCAST**, **MPI_GATHER**, **MPI_GATHER*V***, **MPI_REDUCE**, **MPI_REDUCESCATTER**, **MPI_SCAN**, **MPI_SCATTER**, **MPI_SCATTER*V***
- "**All**" versions deliver results to all participating processes
- "***V***" versions (stands for vector) allow the chunks to have different sizes
- **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

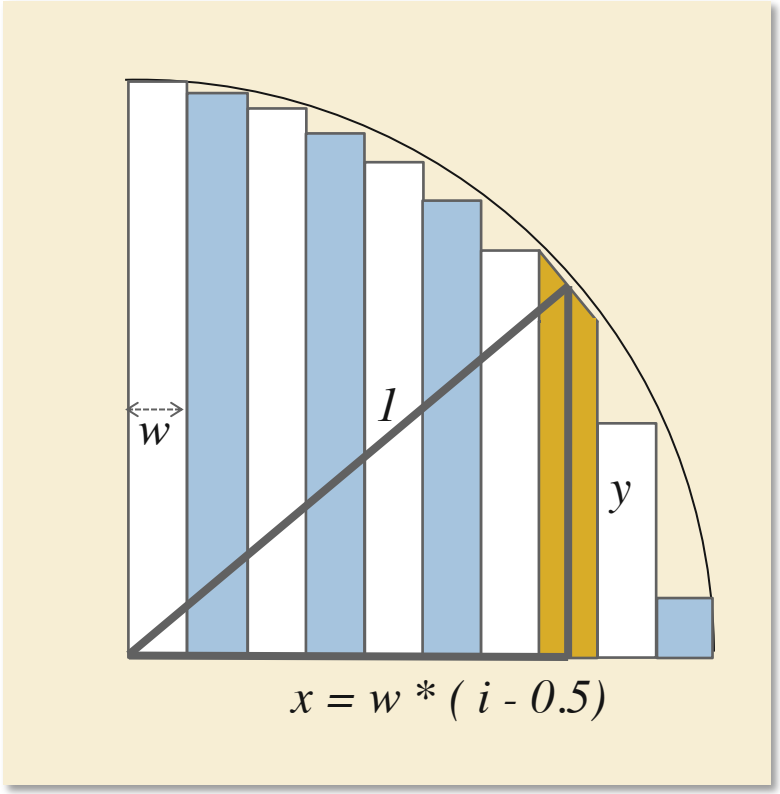
Example: Calculating Pi (1/3)

- *blocking_coll/cpi.c*
- Calculating the approximate value of PI in parallel

Download Examples at https://anl.box.com/v/2021-IIT-MPI-Lecture

Example: Calculating Pi (2/3)

- Calculating the value of “Pi” via numerical integration
  - Value of Pi can be equal to the area of a circle with radius 1
  - Divide each quarter into N strips
  - Distribute strips to processes.
  - Each process calculates area of partial strips and sum them up
  - Add all the partial sums together to get Pi



1. Consider each strip is a trapezoid
2. Width of each strip: w = 1/N
3. Distance of strip “i” from the origin: x = w ∗ (i − 0.5)
4. Height of strip “i”: y = sqrt(1 – x^2)
5. Area of strip “i”: w * y

Example: Calculating Pi (3/3)

```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[])
{
    [...snip...]
    /* Tell all processes, the number of strips you want */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    w   = 1.0 / (double) n;
    mypi = 0.0;
    for (i = rank + 1; i <= n; i += size) {
        x = w * ((double) i - 0.5);
        mypi += sqrt(1.0 - x * x);
    }
    mypi *= w;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
        printf("pi is approximately %.16f, Error is %.16f\n", 4 * pi,
               fabs((4 * pi) - PI25DT));
    [...snip...]
}
```

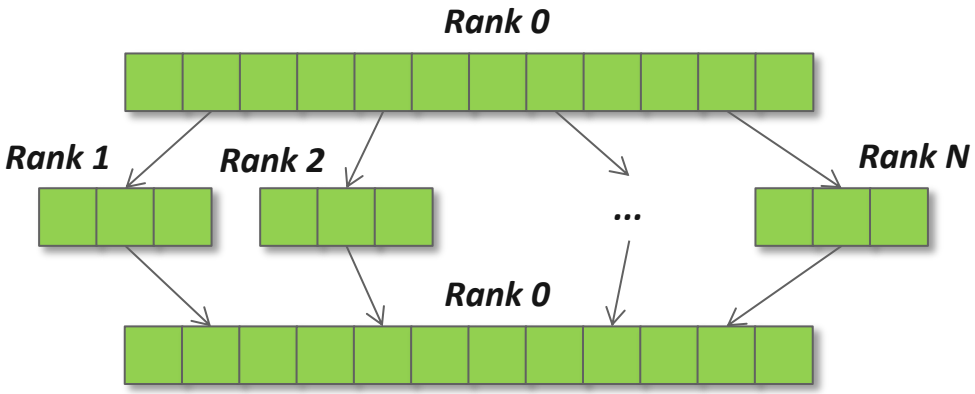
```
% mpicc -o cpi cpi.c -lm
```

Section Summary

- Collectives are a very powerful feature in MPI
- Optimized heavily in most MPI implementations
  - Algorithmic optimizations (e.g., tree-based communication)
  - Hardware optimizations (e.g., network or switch-based collectives)
- Matches the communication pattern of many applications

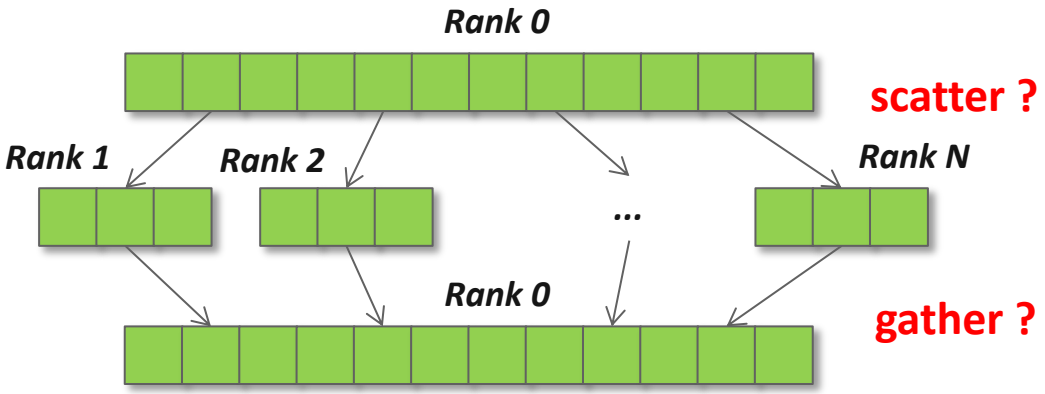
Homework-1: Sorting with Many Processes

- In the two-process version of sorting, only the first two processes are active
  - All other processes are waiting
- Let's try sorting using an arbitrary number of processes
- *Start from blocking_p2p/sort_2_procs.c*



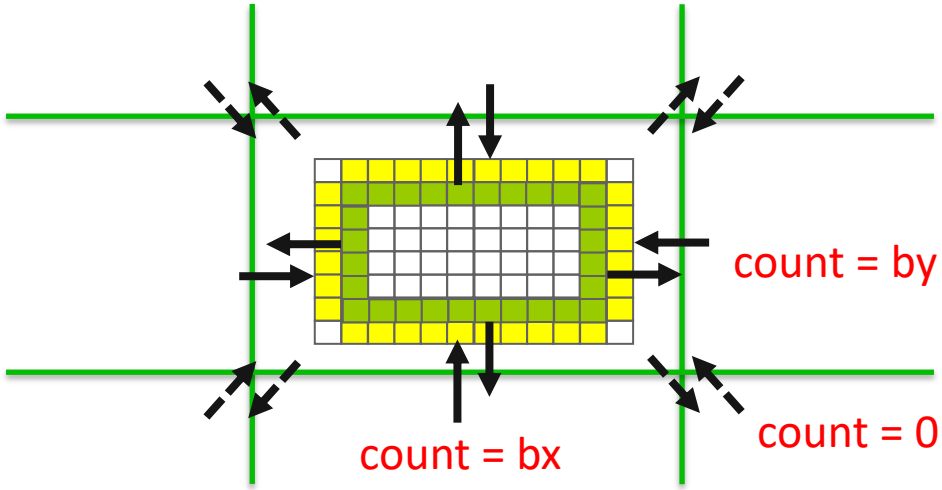
Homework-2: Sorting using Scatter/Gather

- In the send/receive version of sorting, we used multiple send and receive calls:
  - Rank 0 sends a chunk to every process
  - Rank 0 receives the sorted chunk from every process
- Let's try scattering and gathering chunks using collective calls



Homework-3: Stencil using Alltoallv

- In the basic version of the stencil code
  - Used nonblocking send/receive for each direction
- Let's try to use single collective call
- *Start from nonblocking_p2p/stencil.c*



collective ?

Brief Overview of MPI Derived Datatypes

Download Slides and Examples at https://anl.box.com/v/2021-IIT-MPI-Lecture

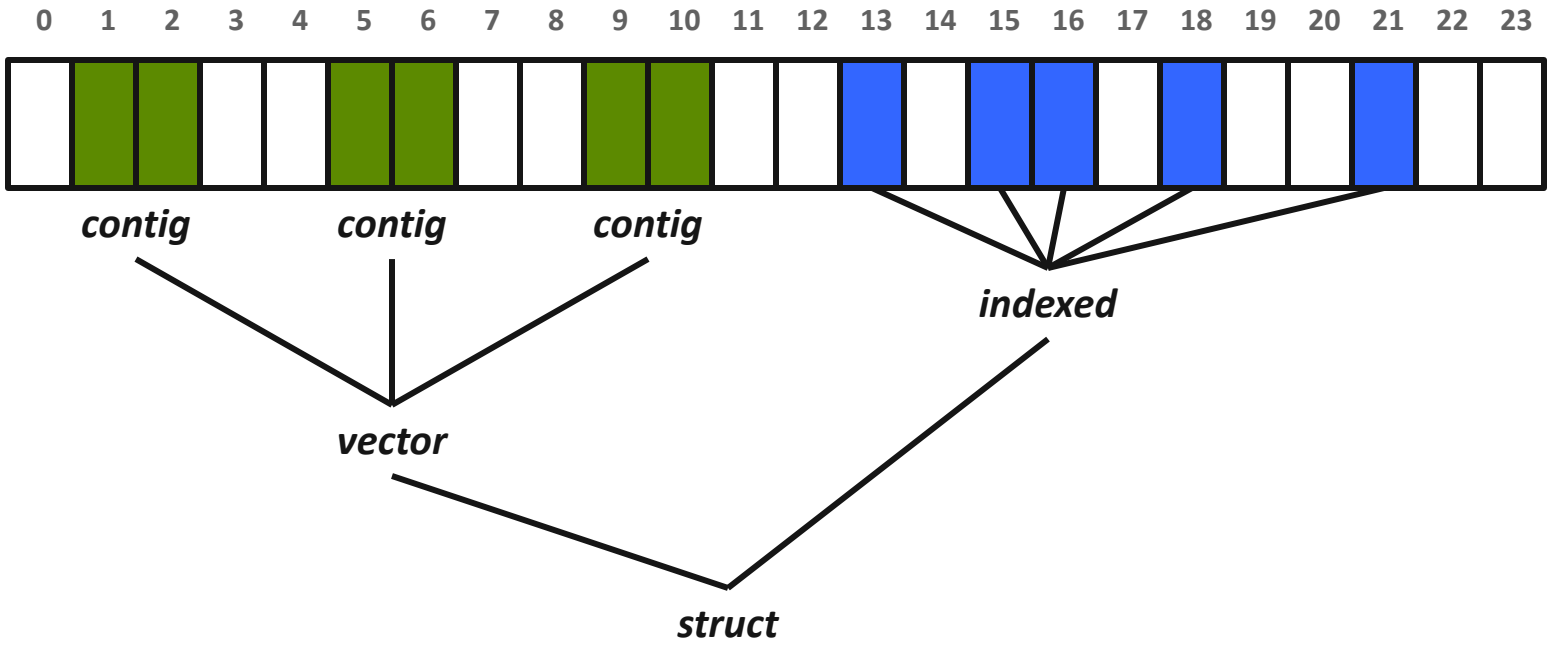
Introduction to Datatypes in MPI

- Datatypes allow to (de)serialize **arbitrary** data layouts into a message stream
  - Networks provide serial channels
  - Same for block devices and I/O
- Several constructors allow arbitrary layouts
  - Recursive specification possible
  - *Declarative* specification of data-layout
    - "what" and not "how", leaves optimization to implementation (*many unexplored* possibilities!)
  - Choosing the right constructors is not always simple

Simple/Predefined Datatypes

- Equivalents exist for all C, C++ and Fortran native datatypes
  - C int → MPI_INT
  - C float → MPI_FLOAT
  - C double → MPI_DOUBLE
  - C uint32_t → MPI_UINT32_T
  - Fortran integer → MPI_INTEGER
- For more complex or user-created datatypes, MPI provides routines to represent them as well
  - Contiguous
  - Vector/Hvector
  - Indexed/Indexed_block/Hindexed/Hindexed_block
  - Struct
  - Some convenience types (e.g., subarray)

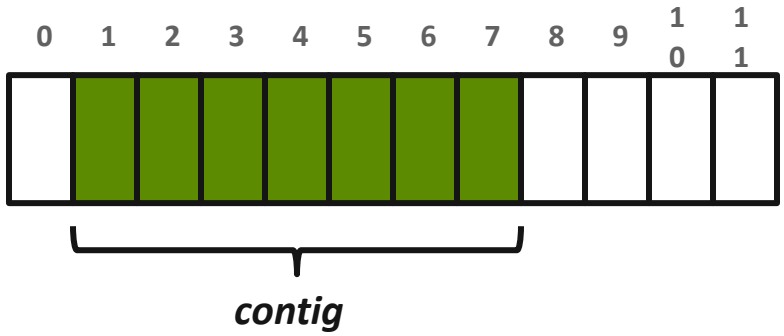
Derived Datatype Example



MPI_Type_contiguous

```
MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

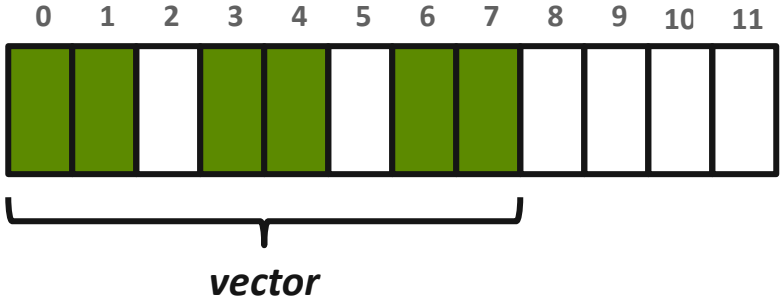
- Contiguous array of oldtype

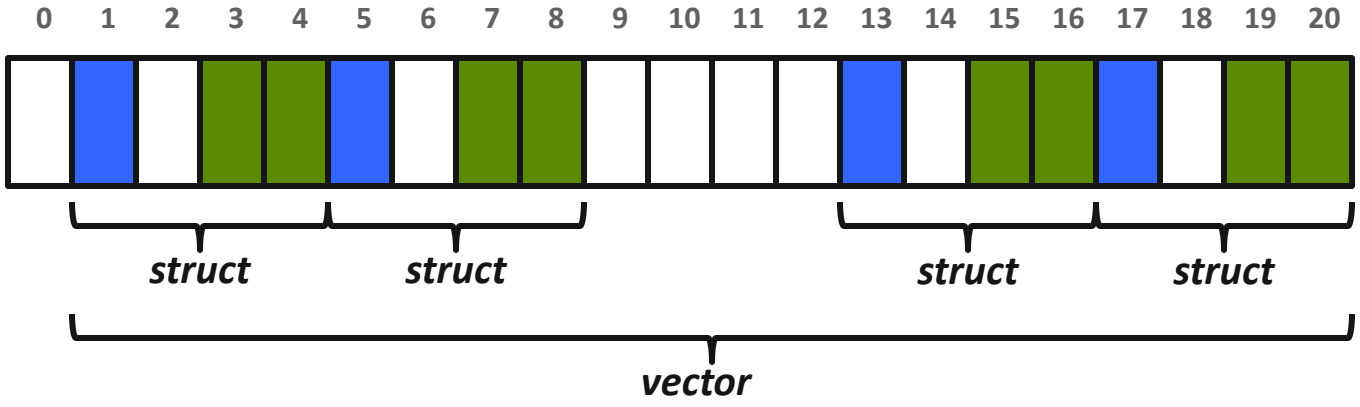


MPI_Type_vector

```
MPI_Type_vector(int count, int blocklen, int stride,
        MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Specify strided blocks of data of oldtype





Commit and Free

- Types must be committed before use
  - Only the ones that are used!
  - MPI_Type_commit may perform heavy optimizations (and will hopefully)
- MPI_Type_free
  - Free MPI resources of datatypes
  - Does not affect types built from it

Homework-4: Stencil with Derived Datatypes

- In the basic version of the stencil code
  - Used nonblocking communication 👍
  - Used manual packing/unpacking of data 👎
- Let's try to use derived datatypes with nonblocking send/receive instead of manually packing/unpacking
- *Start from nonblocking_p2p/stencil.c*