Argonne NATIONAL LABORATORY

Introduction to MPI

Slides Available at http://www.mcs.anl.gov/~balaji/permalinks/argonne14_mpi.php

Rajeev Thakur

Argonne National Laboratory

Email: thakur@mcs.anl.gov

Web: http://www.mcs.anl.gov/~thakur

Pavan Balaji

Argonne National Laboratory

Email: balaji@anl.gov

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

Ken Raffenetti

Argonne National Laboratory

Email: raffenet@mcs.anl.gov

Web: http://www.mcs.anl.gov/~raffenet

Wesley Bland

Argonne National Laboratory

Email: wbland@mcs.anl.gov

Web: http://www.mcs.anl.gov/~wbland

Xin Zhao

University of Illinois, Urbana-Champaign

Email: xinzhao3@illinois.edu

Web: http://web.engr.illinois.edu/~xinzhao3

U.S. DEPARTMENT OF ENERGY

What we will cover in this tutorial

- **What is MPI?**
- How to write a simple program in MPI
- Running your application with MPICH
- Slightly more advanced topics:
  - Non-blocking communication in MPI
  - Group (collective) communication in MPI
  - MPI Datatypes
- Conclusions and Final Q/A

The switch from sequential to parallel computing

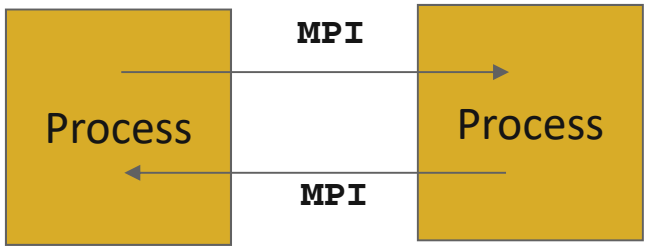
- Moore's law continues to be true, but…
  - Processor speeds no longer double every 18-24 months
  - Number of processing units double, instead
    - Multi-core chips (dual-core, quad-core)
  - No more automatic increase in speed for software
- Parallelism is the norm
  - Lots of processors connected 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
    - By airline industry to build newer models of flights

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)

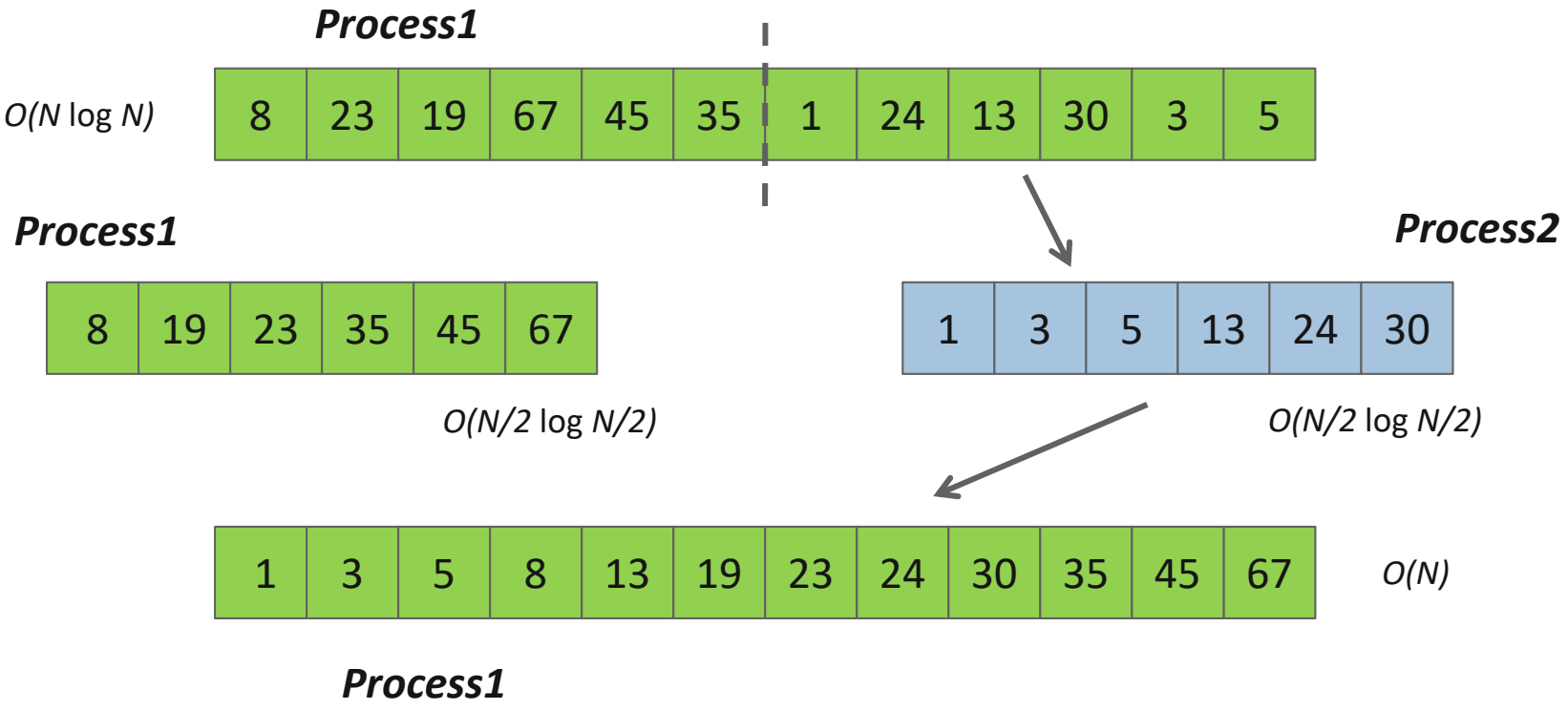
The Message-Passing Model

- 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

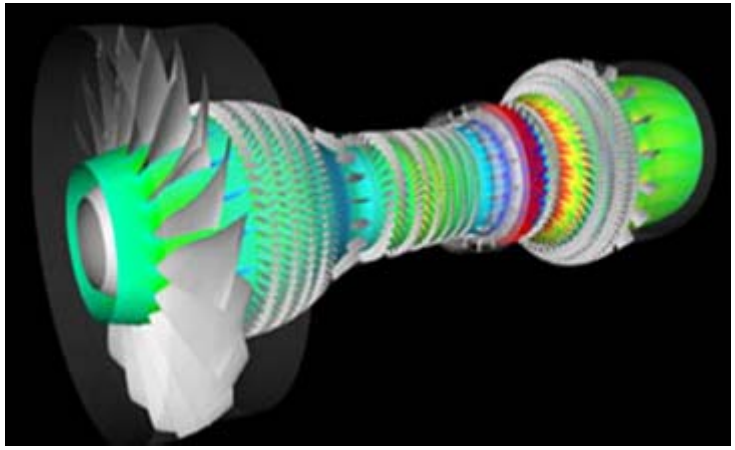
- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable)
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of message-passing issues
  - Lacked vendor support
  - Were not implemented at the most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

What is MPI?

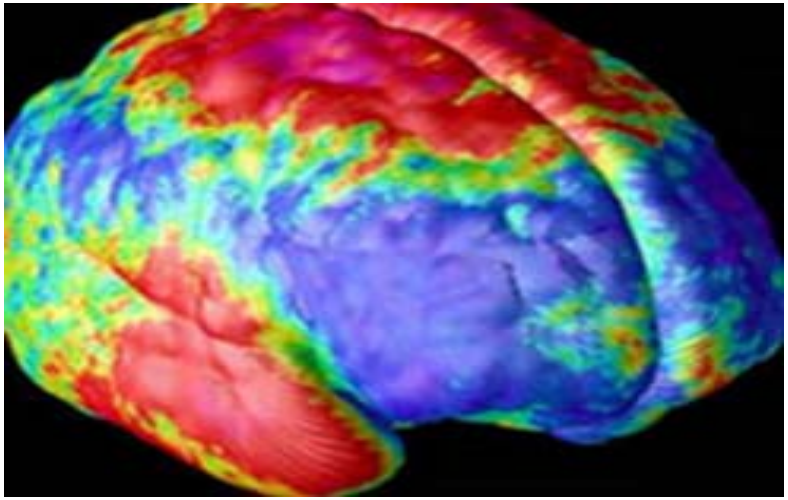
- MPI: Message Passing Interface
  - The MPI Forum organized in 1992 with broad participation by:
    - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
    - Portability library writers: PVM, p4
    - Users: application scientists and library writers
    - MPI-1 finished in 18 months
  - Incorporates the 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

Applications (Science and Engineering)

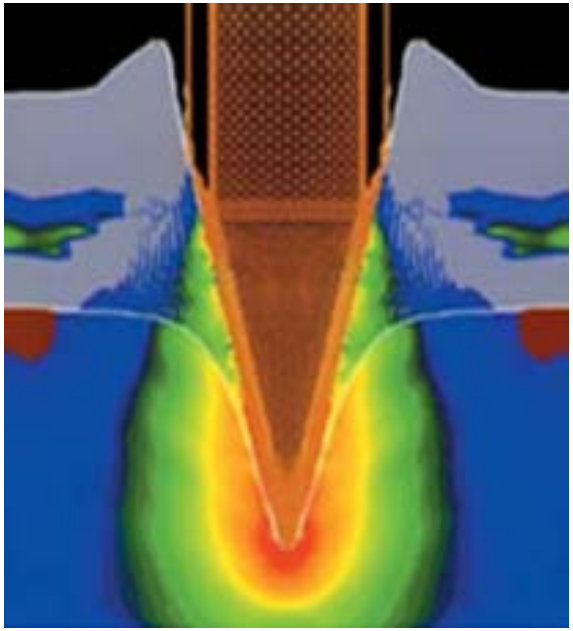
- MPI is widely used in large scale parallel applications in science and engineering
  - Atmosphere, Earth, Environment
  - Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
  - Bioscience, Biotechnology, Genetics
  - Chemistry, Molecular Sciences
  - Geology, Seismology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



Turbo machinery (Gas turbine/compressor)

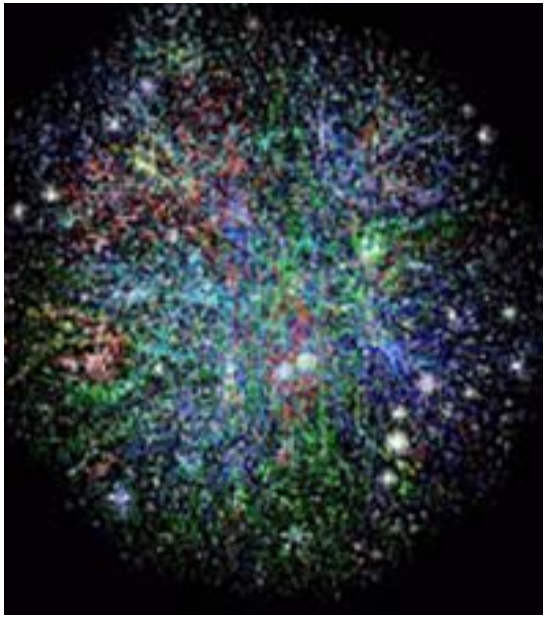


Biology application



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
  - 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

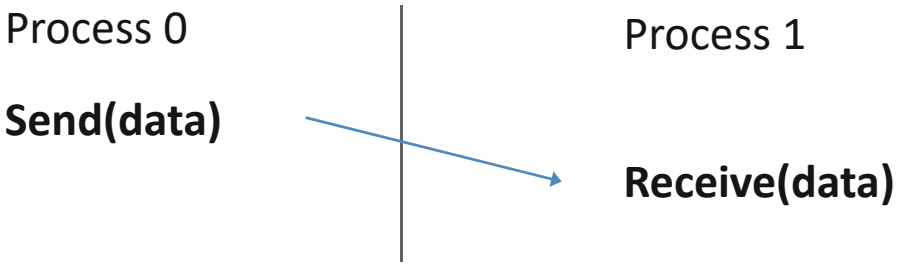
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

What we will cover in this tutorial

- What is MPI?
- **How to write a simple program in MPI**
- Running your application with MPICH
- Slightly more advanced topics:
  - Non-blocking communication in MPI
  - Group (collective) communication in MPI
  - MPI Datatypes
- Conclusions and Final Q/A

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 -n 16 ./test`

Process Identification

- 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

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;
}
```

Basic requirements for an MPI program

Code Example

- *intro-hello.c*

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 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) 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 (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

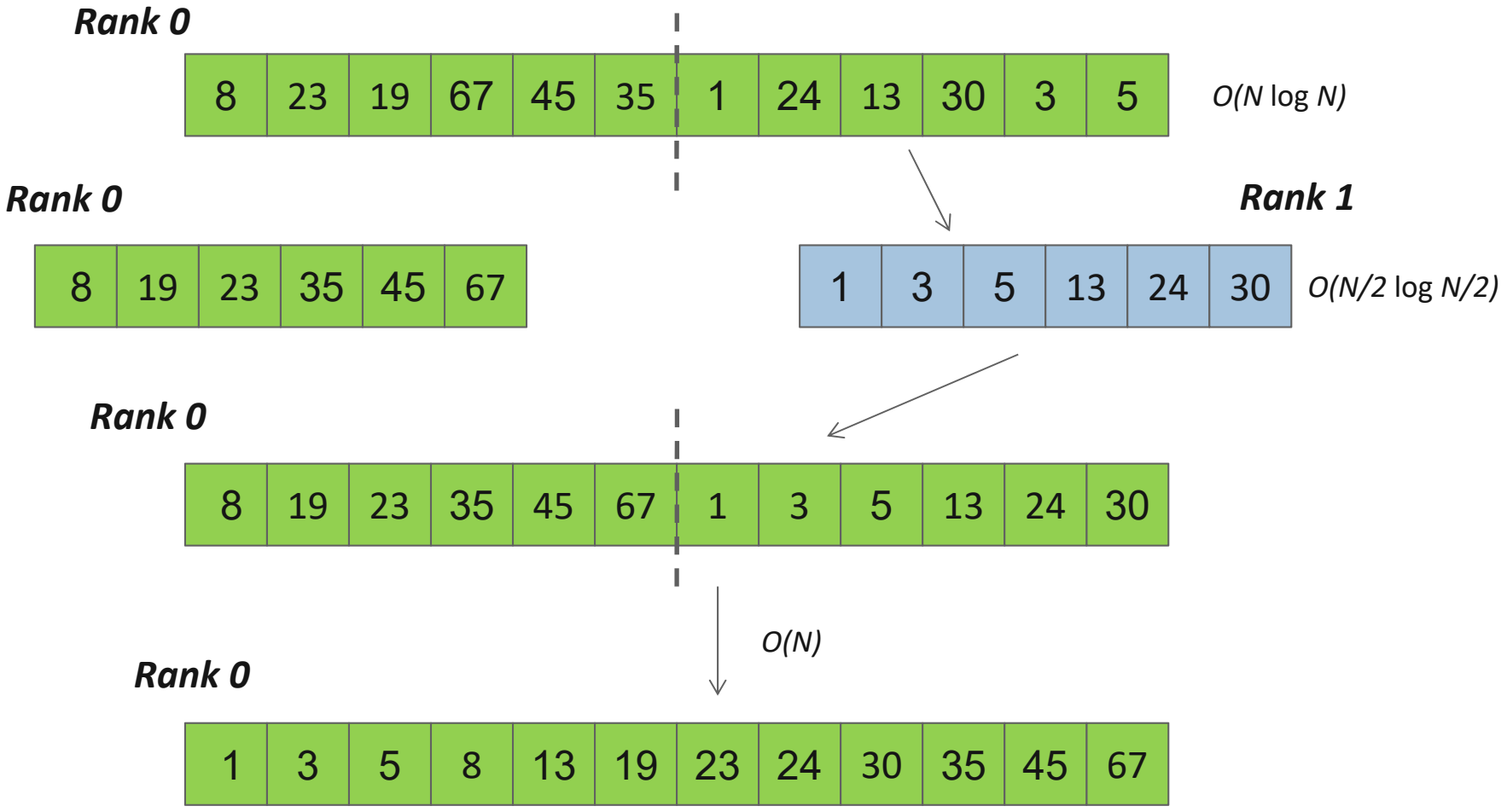
Simple Communication in MPI

```
#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;
}
```

Code Example

- *intro-sendrecv.c*

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);

        /* 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,
                                 MPI_STATUS_IGNORE);
        sort(b, 500);
        MPI_Send(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize(); return 0;
}
```

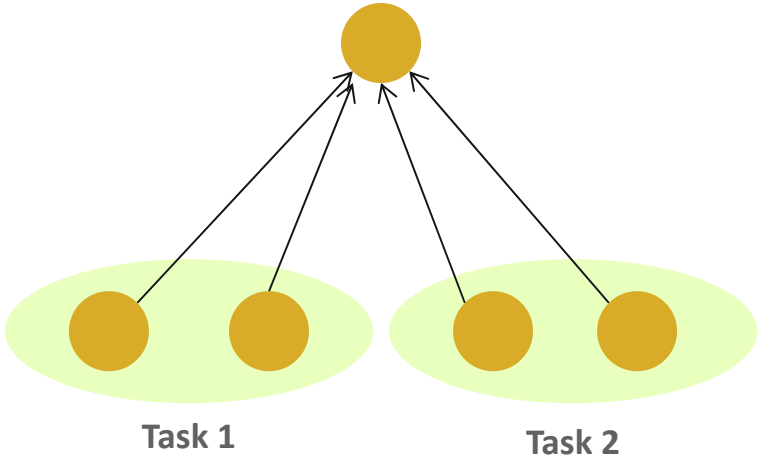
Status Object

- 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 source 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)  /* worker process */
        MPI_Send(data, rand() % 100, MPI_INT, 0, group_id,
                 MPI_COMM_WORLD);
    else {  /* master process */
        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.MPI_SOURCE, status.MPI_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

What we will cover in this tutorial

- What is MPI?
- How to write a simple program in MPI
- **Running your application with MPICH**
- Slightly more advanced topics:
  - Non-blocking communication in MPI
  - Group (collective) communication in MPI
  - MPI Datatypes
- Conclusions and Final Q/A

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, and MPI-3.0)
- Active development lead by Argonne National Laboratory and University of Illinois at Urbana-Champaign
  - Several close collaborators who contribute many features, bug fixes, testing for quality assurance, etc.
    - IBM, Microsoft, Cray, Intel, Ohio State University, Queen's University, Myricom and many others
- Current release is MPICH-3.1.1

Getting Started with MPICH

- 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 -xzvf mpich-3.1.1.tar.gz`
  - `cd mpich-3.1.1`
  - `./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`

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 -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.
  - **`mpiexec -hosts h1,h2,h3,h4 -n 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 -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

Interaction with Resource Managers

- Resource managers such as SGE, PBS, SLURM or Loadleveler are common in many managed clusters
  - MPICH automatically detects them and interoperates with them
- For example with PBS, you can create a script such as:

```
#! /bin/bash

cd $PBS_O_WORKDIR
# No need to provide -np or -hostfile options
mpiexec ./test
```

- Job can be submitted as: `qsub -l nodes=2:ppn=2 test.sub`
  - "mpiexec" will automatically know that the system has PBS, and ask PBS for the number of cores allocated (4 in this case), and which nodes have been allocated
- The usage is similar for other resource managers

Debugging MPI programs

- Parallel debugging is trickier than debugging serial programs
  - Many processes computing; getting the state of one failed process is usually hard
  - MPICH provides in-built support for debugging
    - It natively interoperates with commercial parallel debuggers such as Totalview and DDT
- Using MPICH with totalview:
  - **`totalview -a mpiexec -n 6 ./test`**
- Using MPICH with ddd (or gdb) on one process:
  - **`mpiexec -n 4 ./test : -n 1 ddd ./test : -n 1 ./test`**
  - Launches the 5th process under "ddd" and all other processes normally

What we will cover in this tutorial

- What is MPI?
- How to write a simple program in MPI
- Running your application with MPICH
- **Slightly more advanced topics:**
  - **Non-blocking communication in MPI**
  - Group (collective) communication in MPI
  - MPI Datatypes
- Conclusions and Final Q/A

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

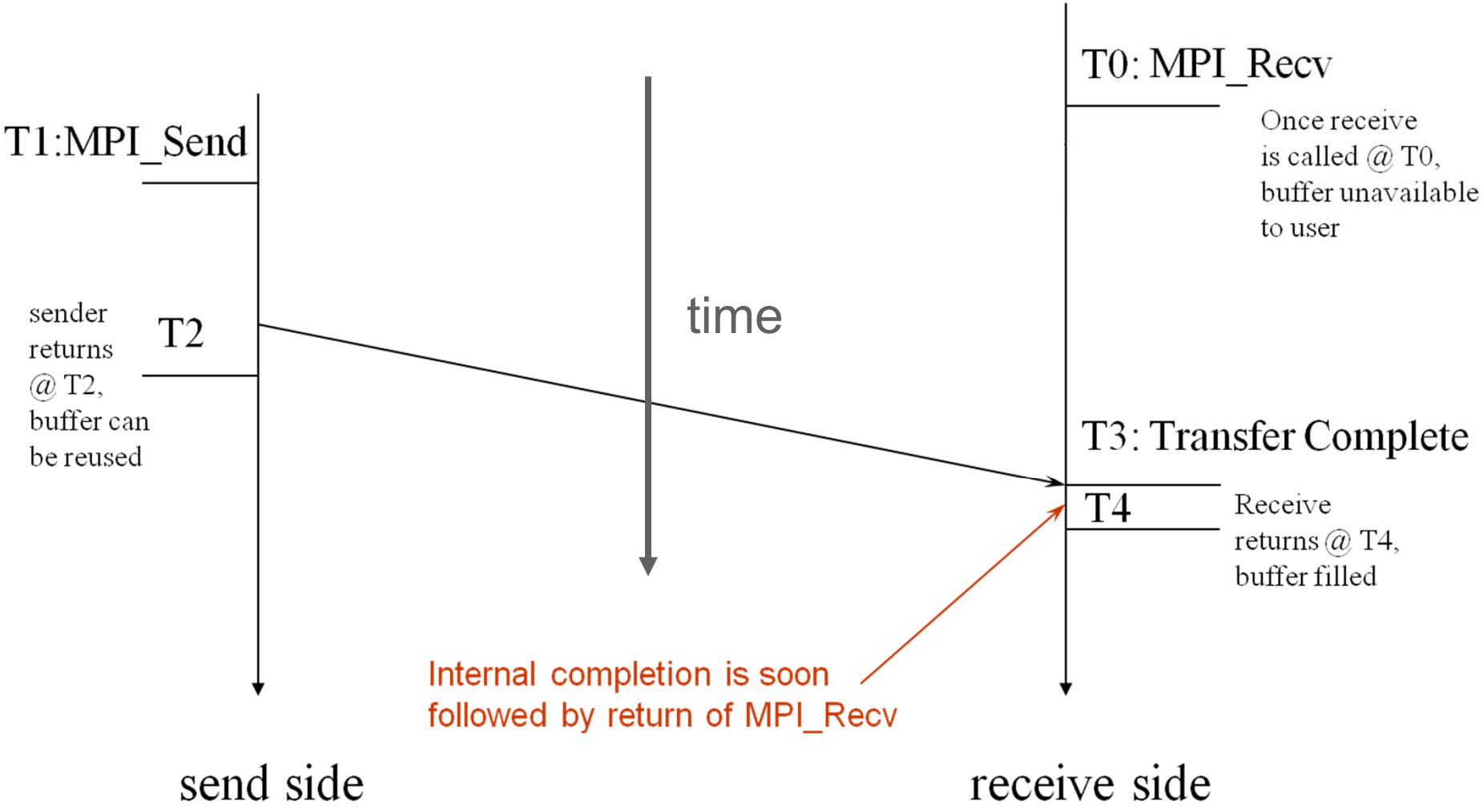
- In blocking communication.
  - **MPI_SEND** does not return until buffer is empty (available for reuse)
  - **MPI_RECV** does not return until buffer is full (available for use)
- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked 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) {
        MPI_SEND(..to rank 1..)
        MPI_RECV(..from rank 1..)
else if (rank == 1) {
        MPI_SEND(..to rank 0..)
        MPI_RECV(..from rank 0..)
}
```

Usually deadlocks →

← reverse send/recv

Blocking Send-Receive Diagram



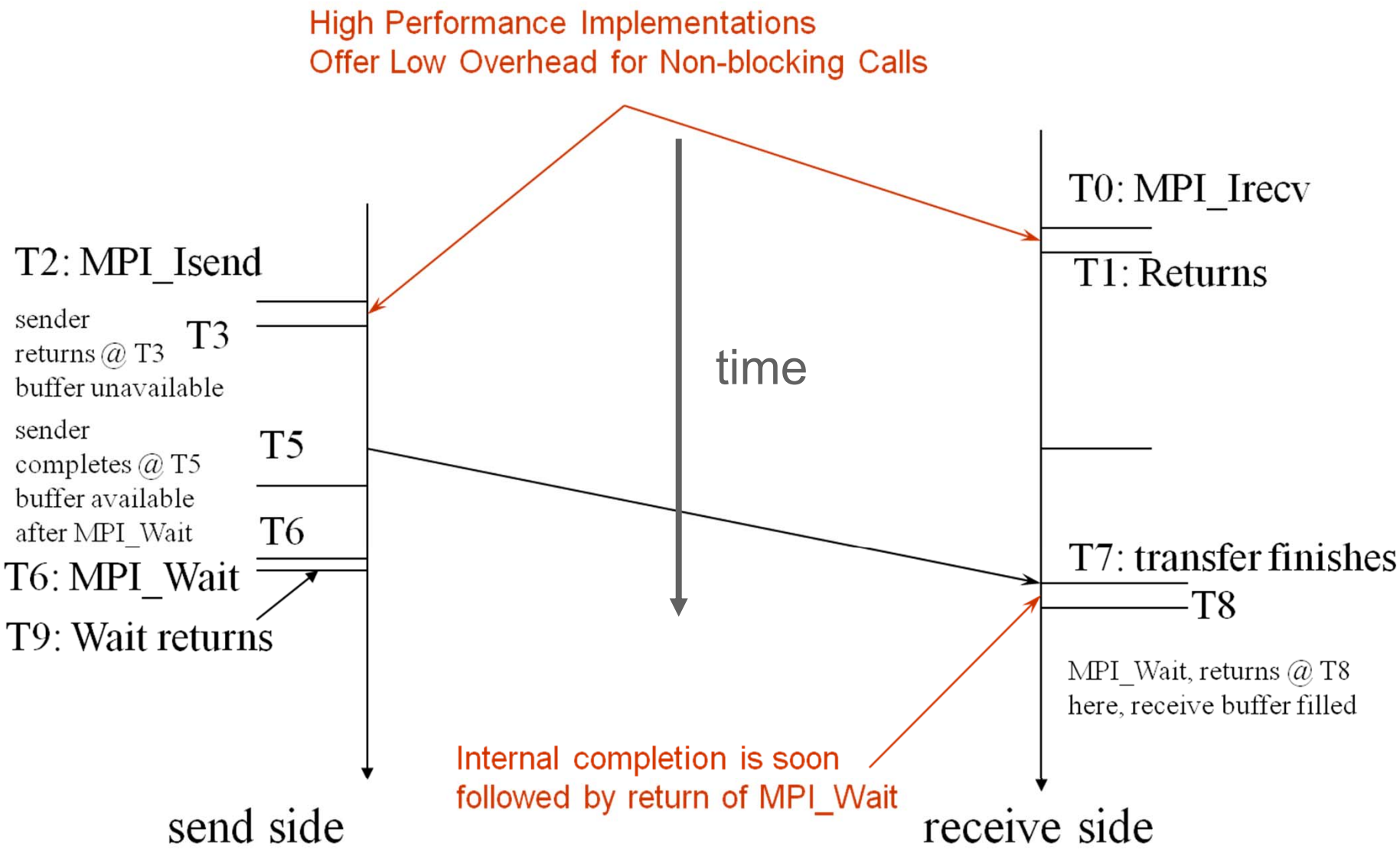
Non-Blocking Communication

- Non-blocking (asynchronous) 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)`
- 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**

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

Non-Blocking Send-Receive Diagram



Message Completion and Buffering

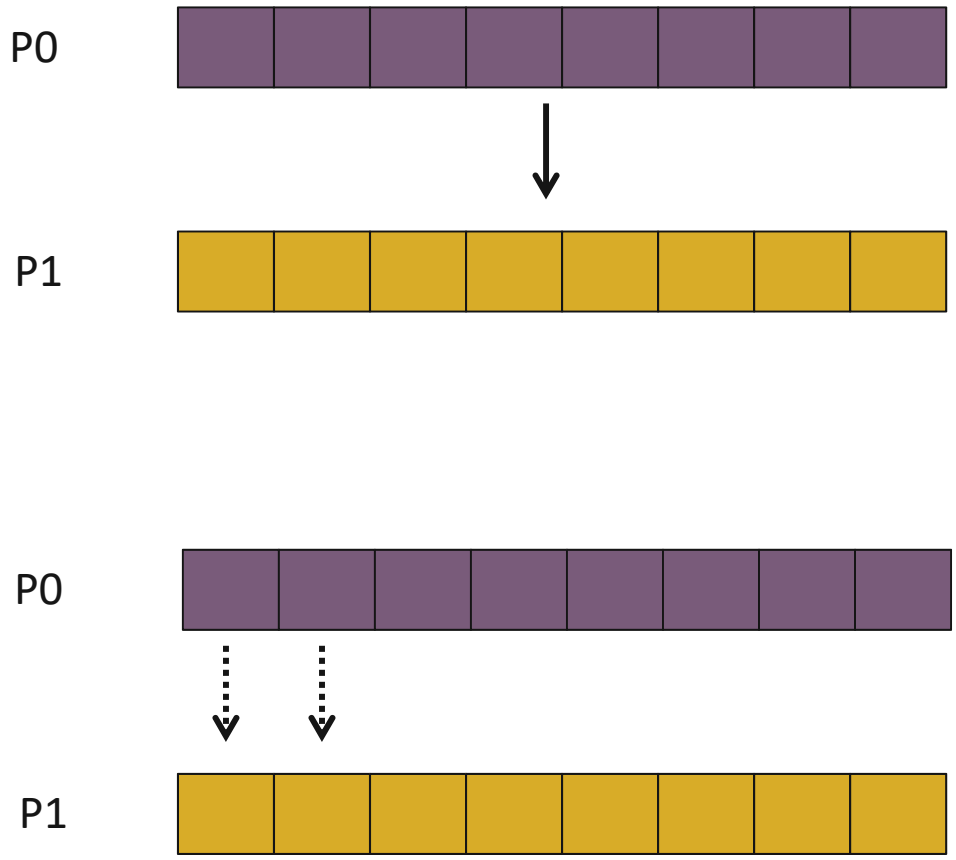
- For a communication to succeed:
  - Sender must specify a valid destination rank
  - Receiver must specify a valid source rank (including MPI_ANY_SOURCE)
  - The communicator must be the same
  - Tags must match
  - Receiver's buffer must be large enough
- A send has completed when the user supplied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

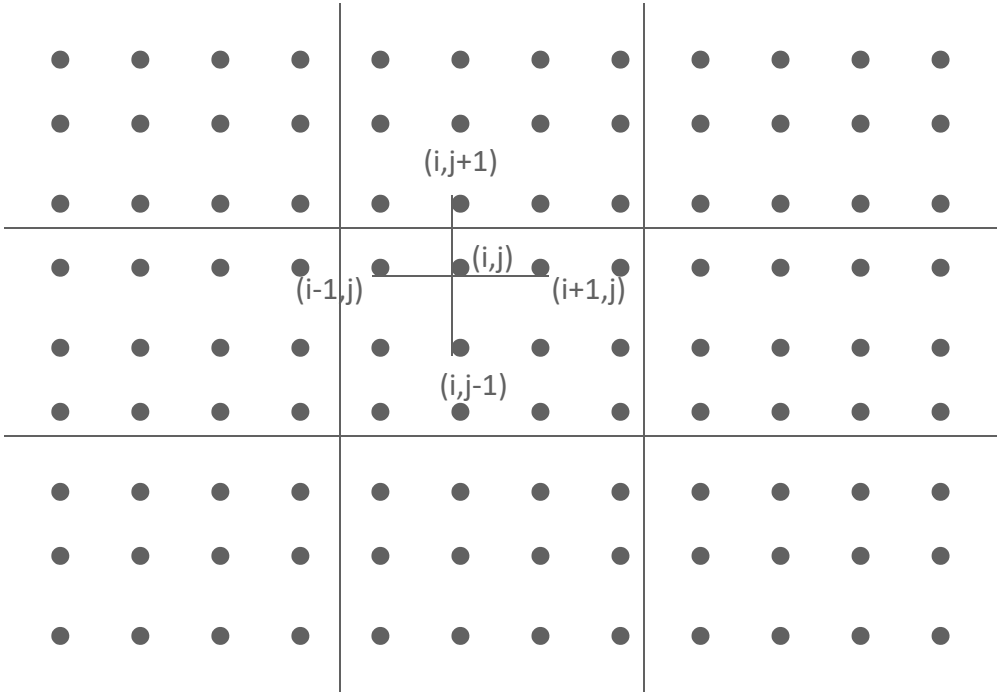
A Non-Blocking communication example



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...]
}
```

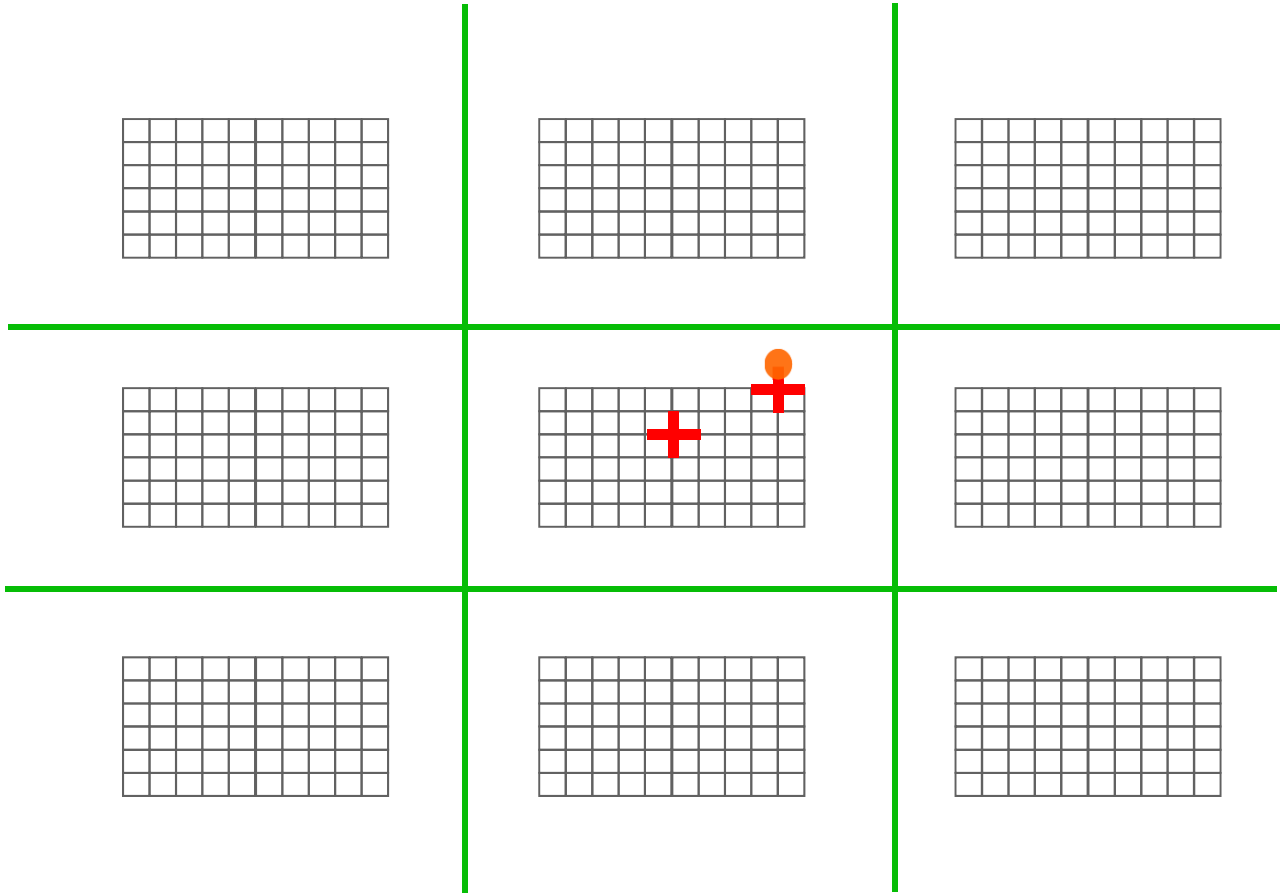
2D Poisson Problem



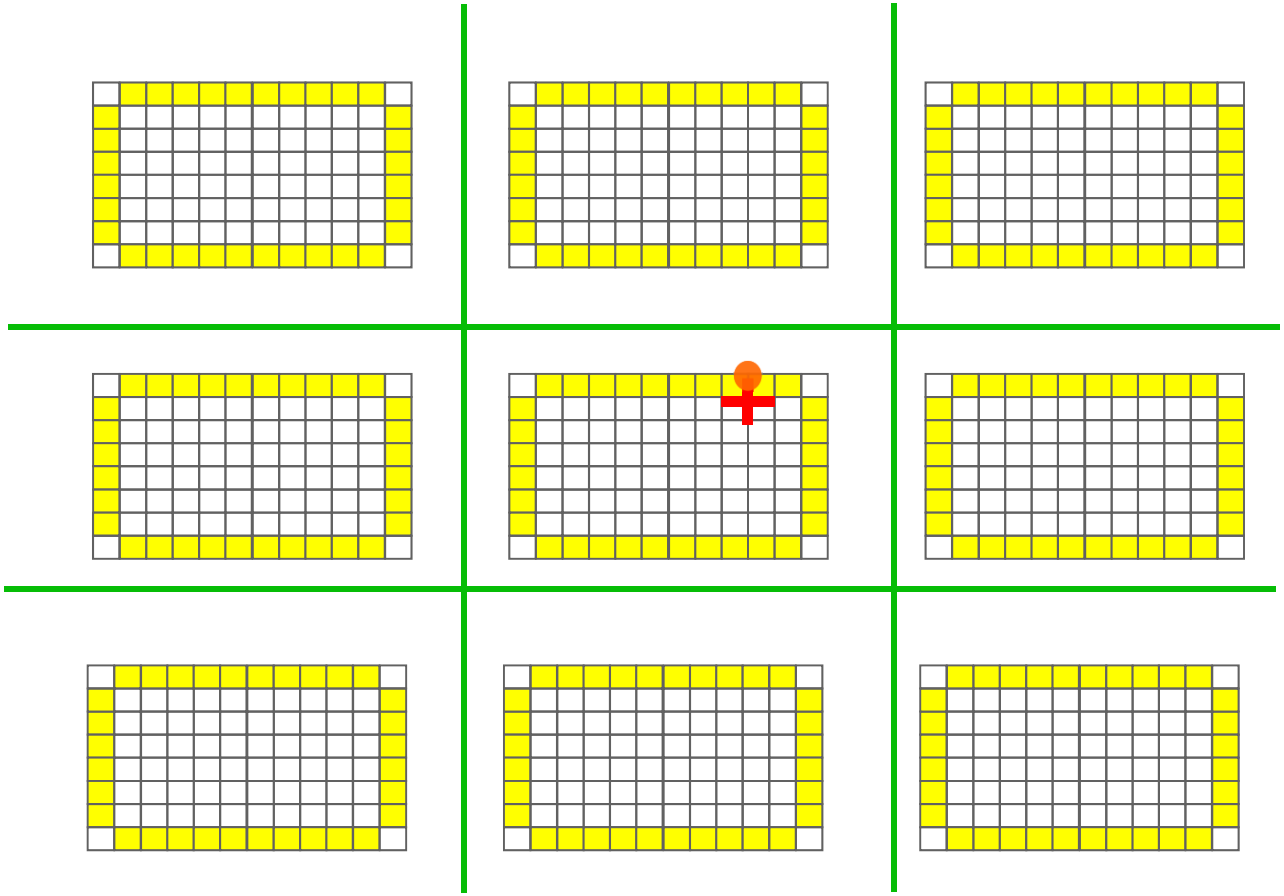
Regular Mesh Algorithms

- 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

Necessary Data Transfers

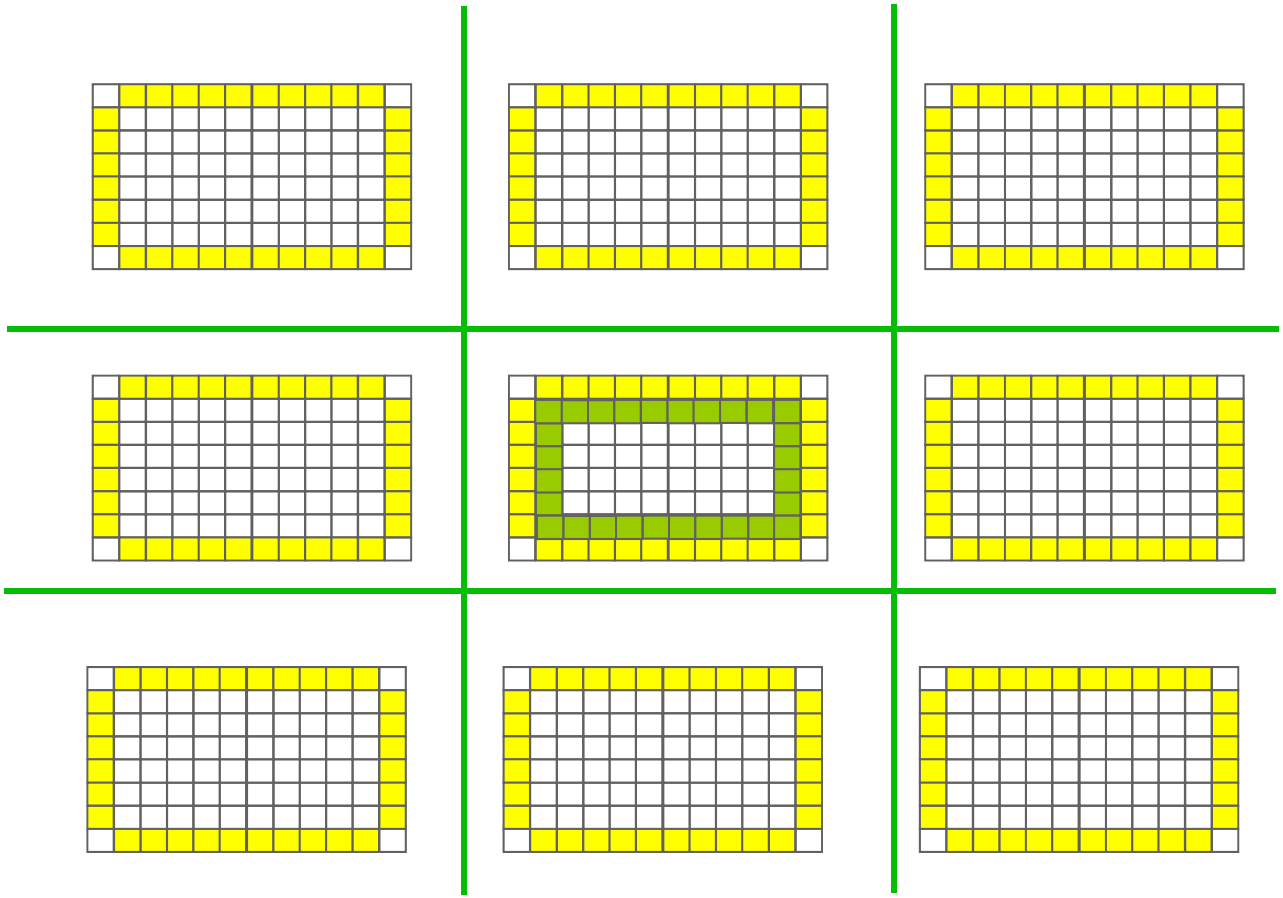


Necessary Data Transfers



Necessary Data Transfers

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

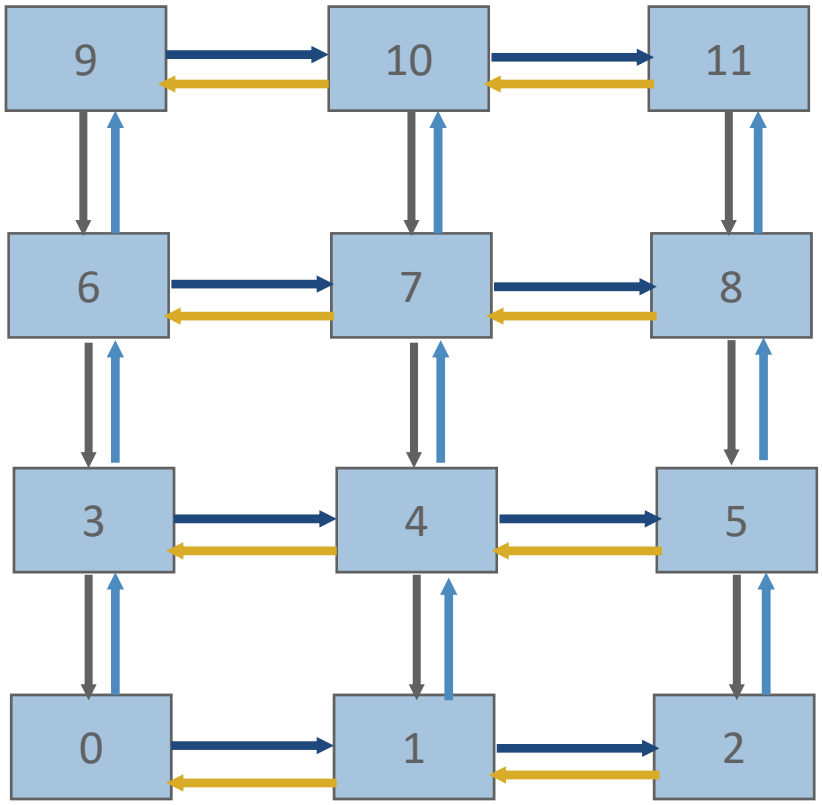


Understanding Performance: Unexpected Hot Spots

- Basic performance analysis looks at two-party exchanges
- Real applications involve many simultaneous communications
- Performance problems can arise even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - Blocking operations may cause unavoidable memory stalls

Mesh Exchange

- Exchange data on a mesh



Sample Code

```
for (i = 0; i < n_neighbors; i++) {
  MPI_Send(edge, len, MPI_DOUBLE, nbr[i], tag, comm);
}
for (i = 0; i < n_neighbors; i++) {
  MPI_Recv(edge, len, MPI_DOUBLE, nbr[i], tag, comm, status);
}
```

- What is wrong with this code?

Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of

```
if (has up nbr)
   MPI_Recv( … up … )
…
if (has down nbr)
   MPI_Send( … down … )
```

sequentializes (all except the bottom process blocks)

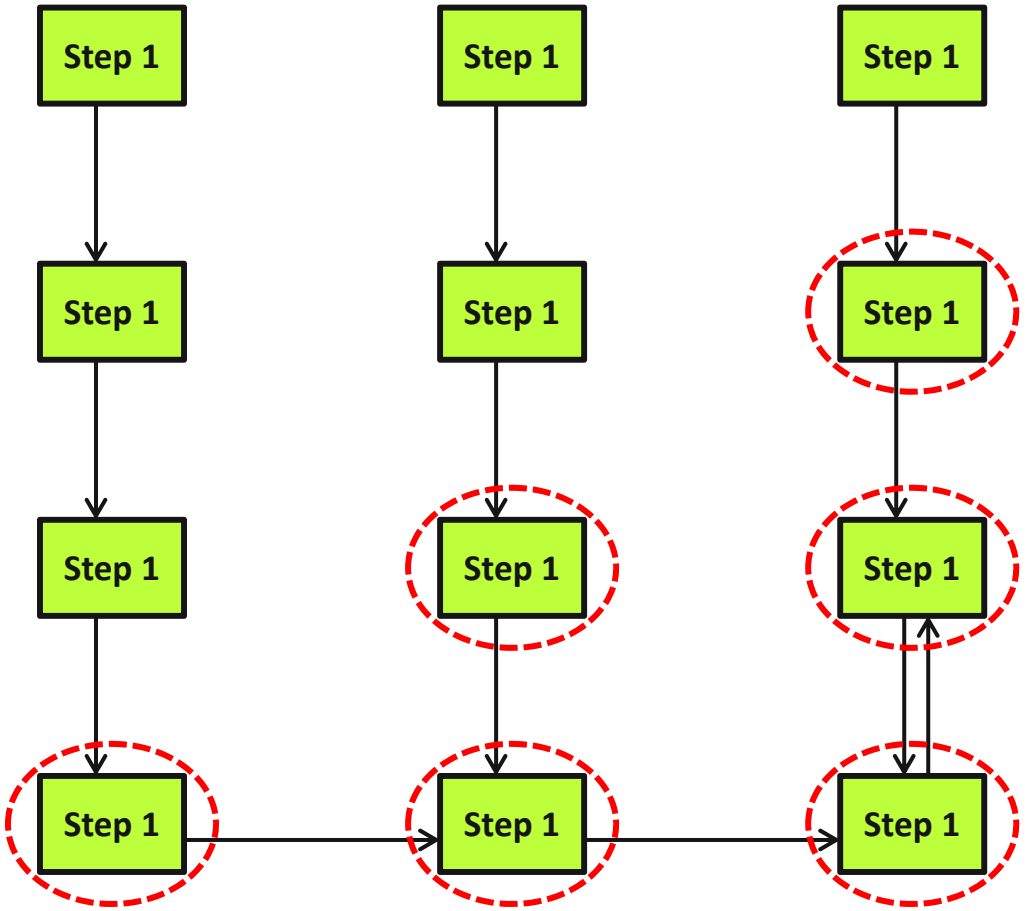
Fix 1: Use Irecv

```
for (i = 0; i < n_neighbors; i++) {
        MPI_Irecv(edge, len, MPI_DOUBLE, nbr[i], tag,
                        comm, requests[i]);
}
for (i = 0; i < n_neighbors; i++) {
        MPI_Send(edge, len, MPI_DOUBLE, nbr[i], tag, comm);
}
MPI_Waitall(n_neighbors, requests, statuses);
```

- Does not perform well in practice. Why?

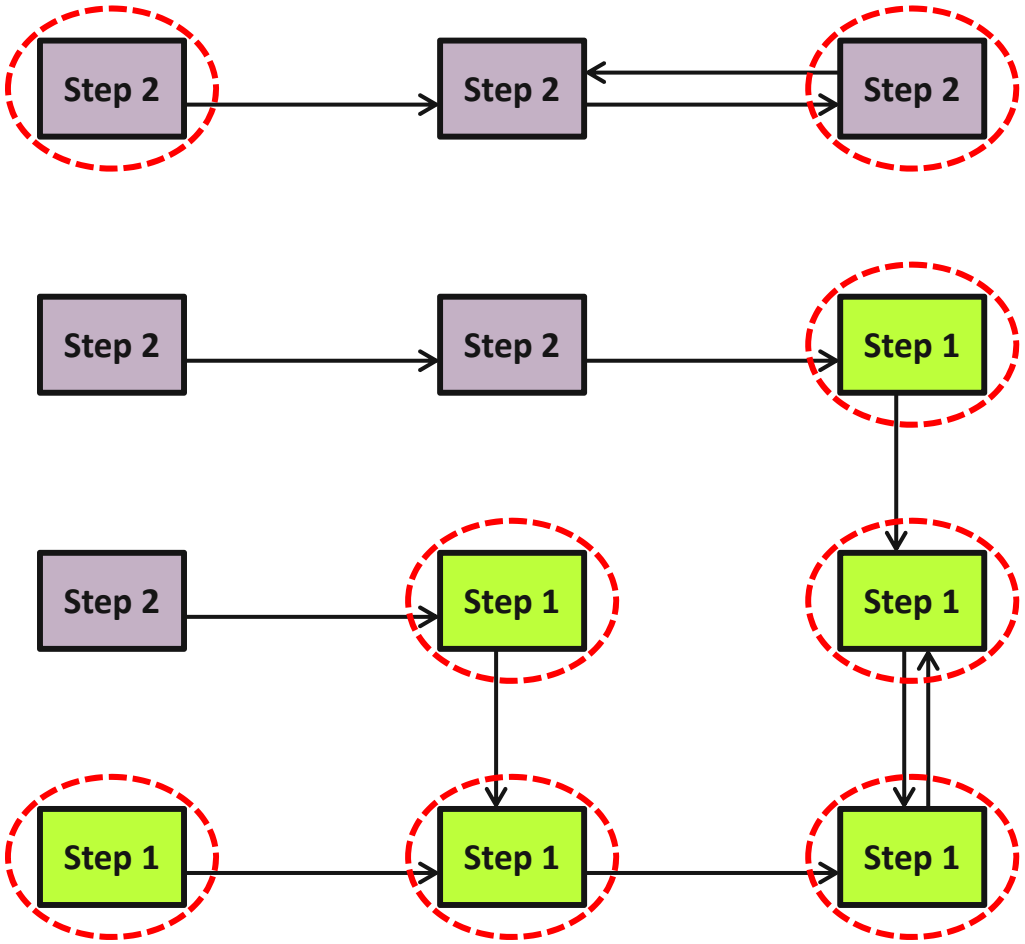
Mesh Exchange

- Exchange data on a mesh



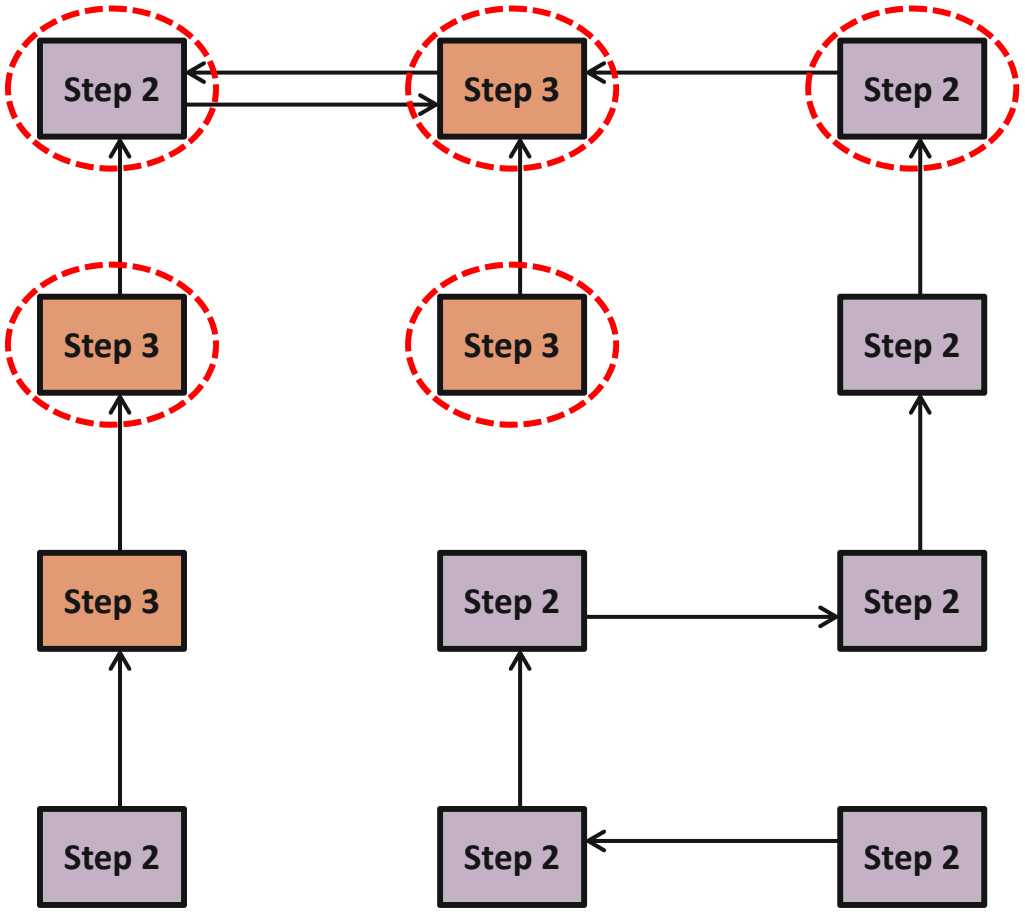
Mesh Exchange

- Exchange data on a mesh



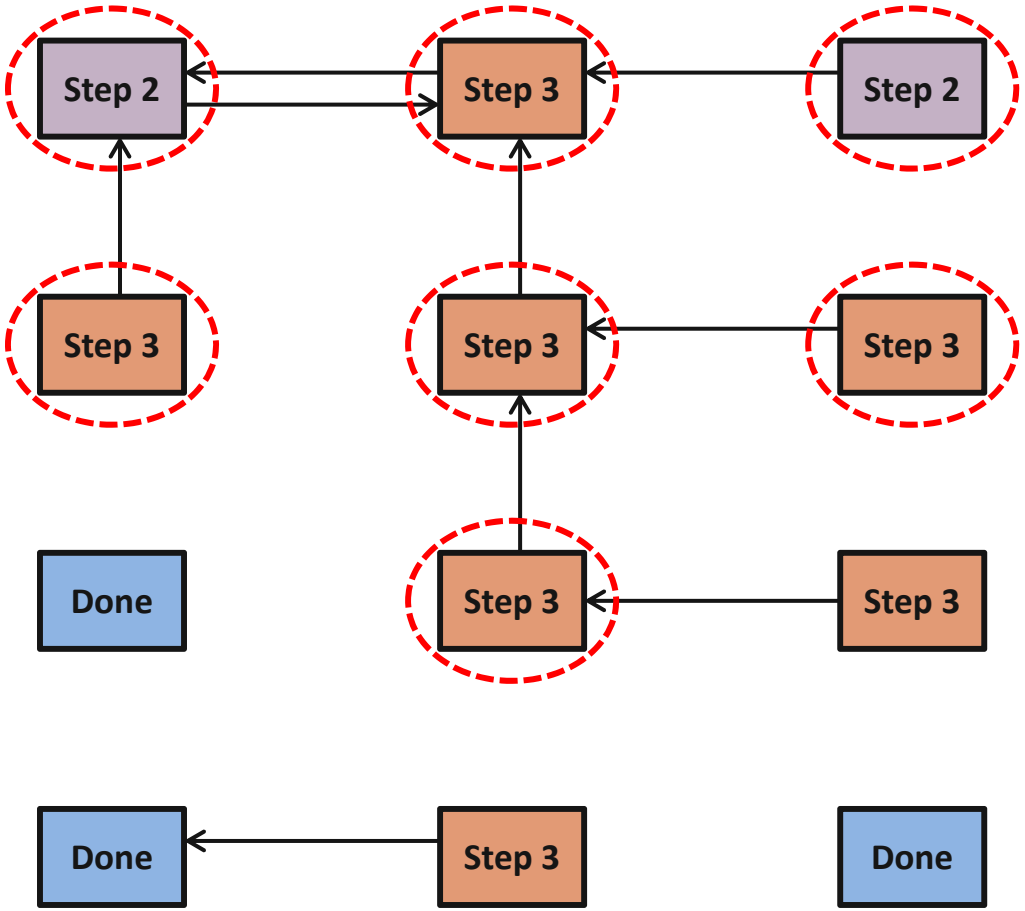
Mesh Exchange

- Exchange data on a mesh



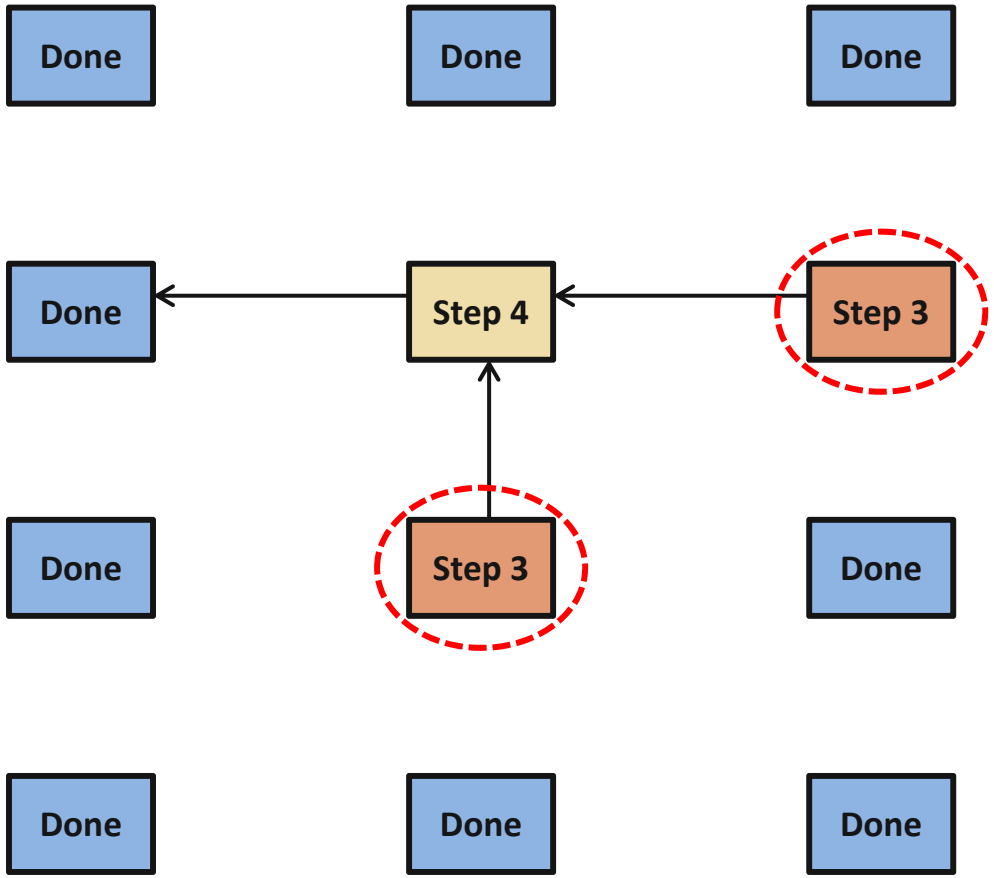
Mesh Exchange

- Exchange data on a mesh



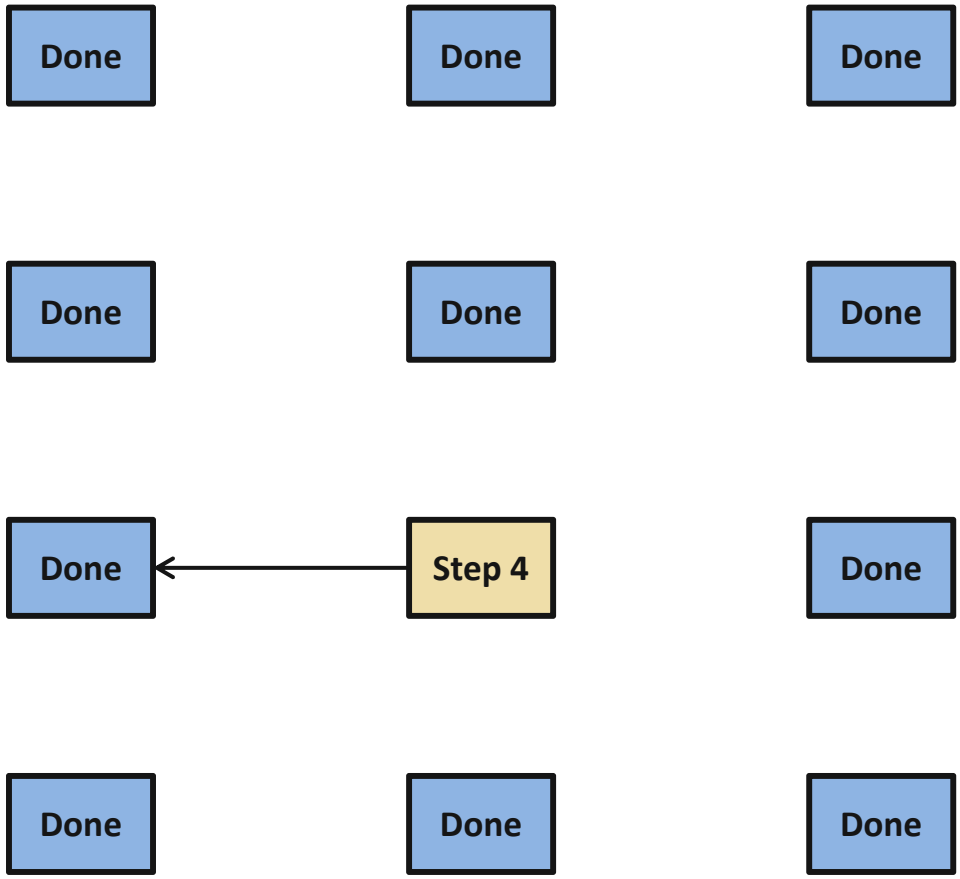
Mesh Exchange

- Exchange data on a mesh

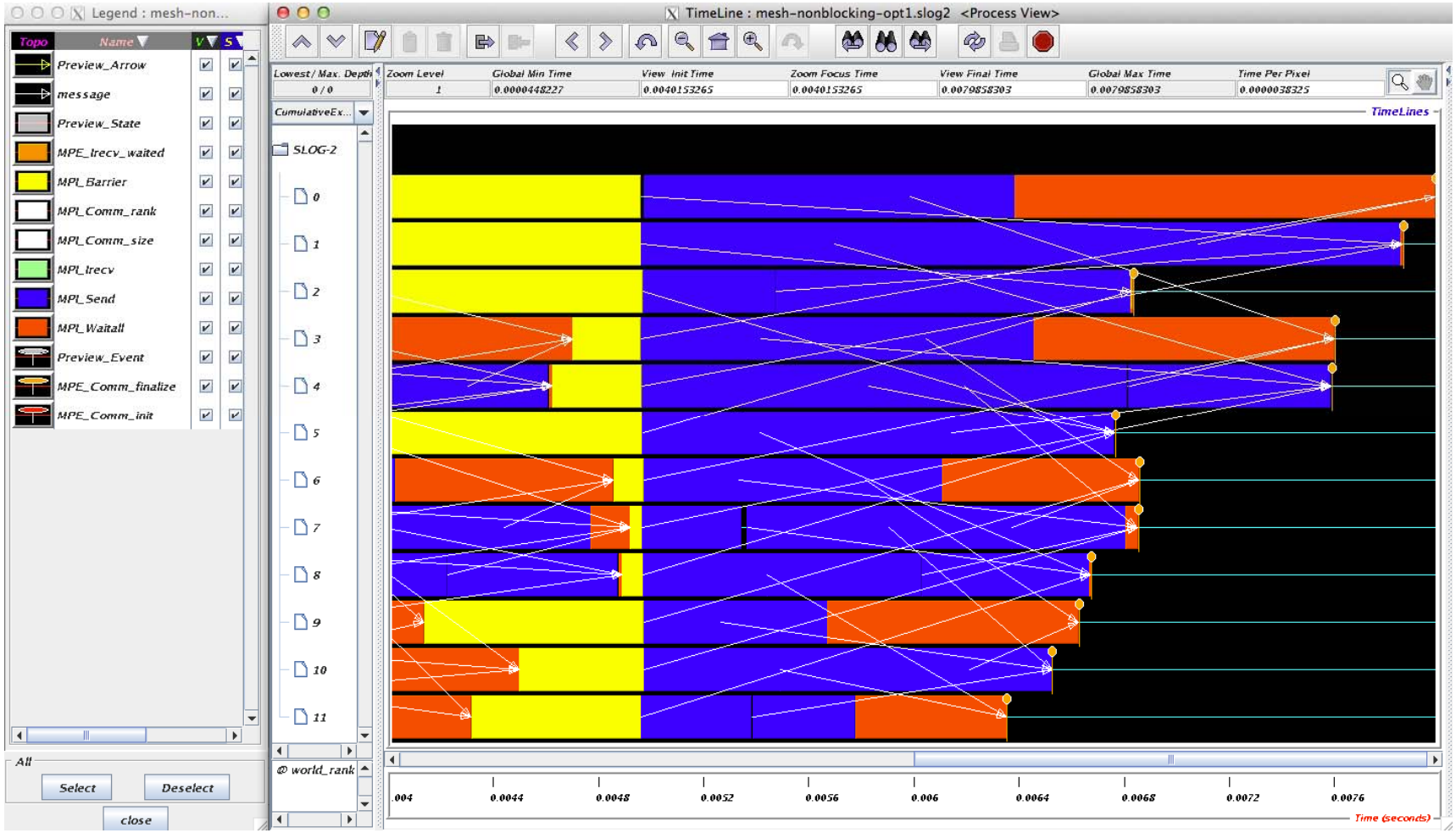


Mesh Exchange

- Exchange data on a mesh



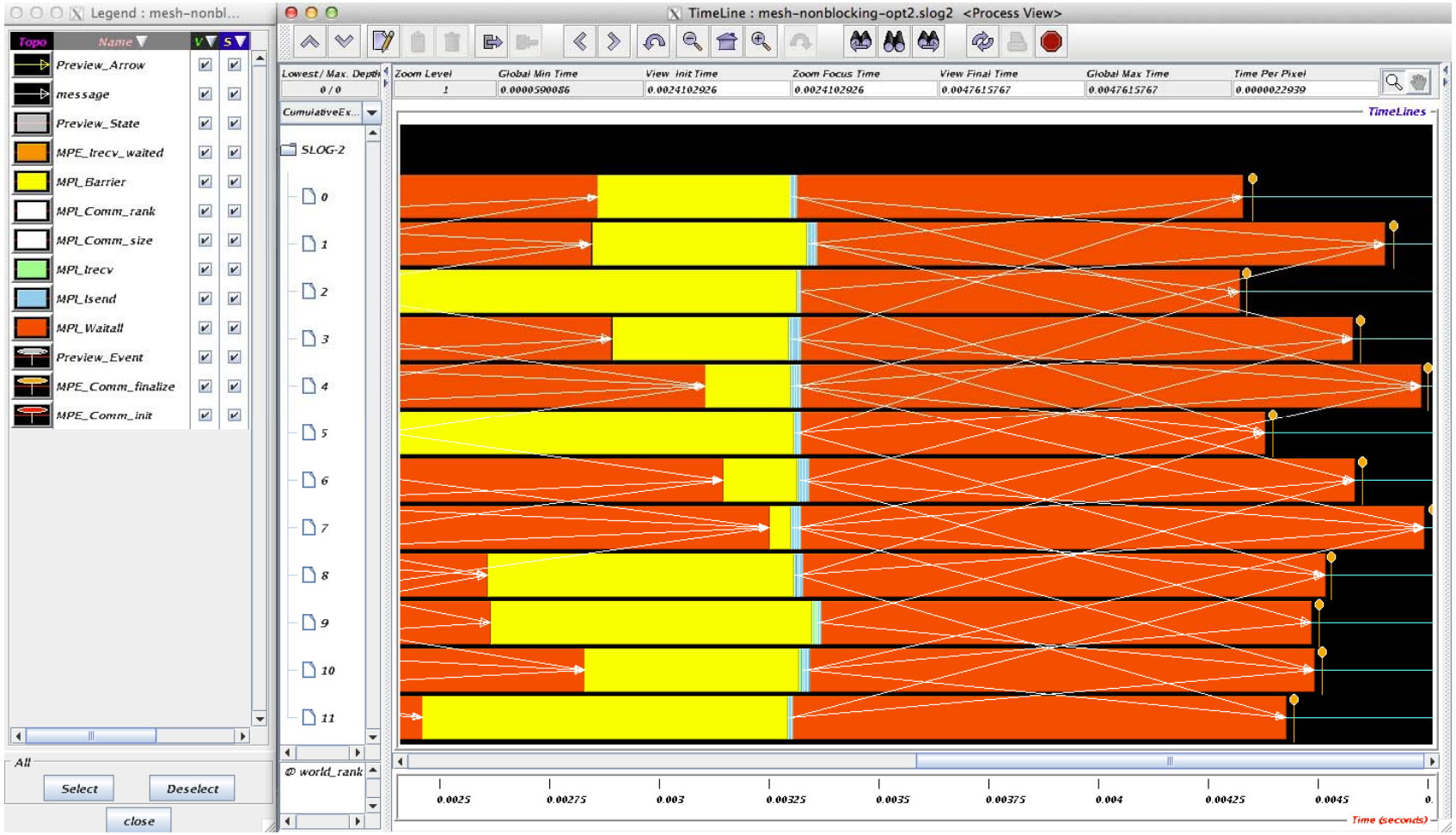
Timeline from IB Cluster



Fix 2: Use Isend and Irecv

```
for (i = 0; i < n_neighbors; i++) {
    MPI_Irecv(edge, len, MPI_DOUBLE, nbr[i], tag,
              comm, requests[i]);
}
for (i = 0; i < n_neighbors; i++) {
    MPI_Isend(edge, len, MPI_DOUBLE, nbr[i], tag, comm,
              requests[n_neighbors + i]);
}
MPI_Waitall(2 * n_neighbors, requests, statuses);
```

Timeline from IB Cluster



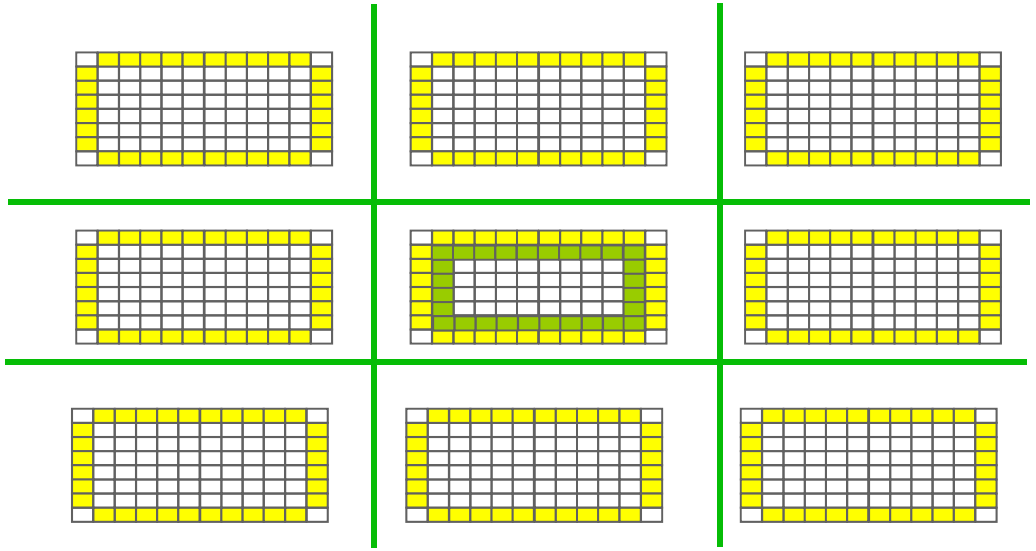
Note processes 4 and 7 are the only interior processors; these perform more communication than the other processors

Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization
- In many cases, there is more synchronization than required
- Use non-blocking operations and `MPI_Waitall` to defer synchronization
- Tools can help out with identifying performance issues
  - MPE, Tau and HPCToolkit are popular profiling tools
  - Jumpshot tool uses their datasets to show performance problems graphically

Code Example

- *stencil_mpi_nonblocking.c*
- Non-blocking sends and receives
- Manually packing and unpacking the data
- Additional communication buffers are needed



- Display message queue state using Totalview
  - totalview mpiexec –a –n 4 ./stencil_mpi_nonblocking 300 250 100 2 2

What we will cover in this tutorial

- What is MPI?
- How to write a simple program in MPI
- Running your application with MPICH
- **Slightly more advanced topics:**
  - Non-blocking communication in MPI
  - **Group (collective) communication in MPI**
  - MPI Datatypes
- 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 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.

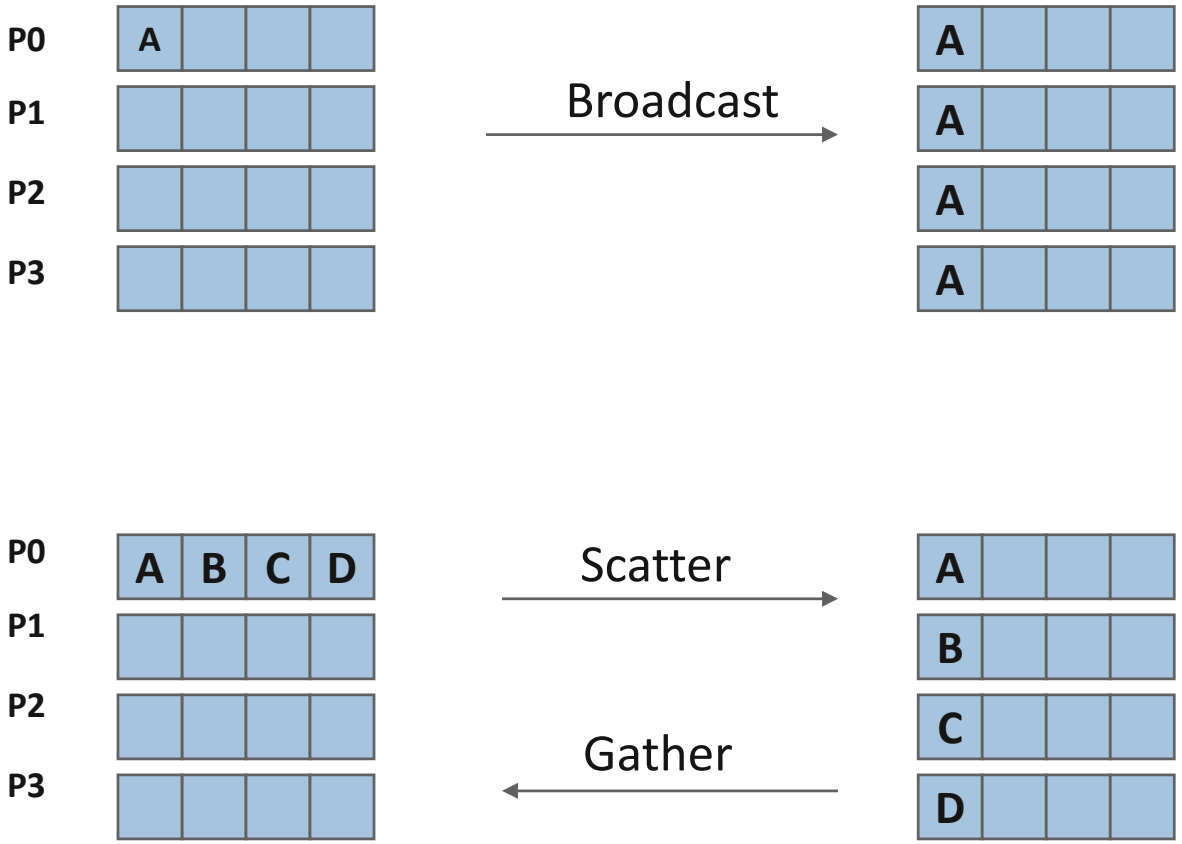
MPI Collective Communication

- 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

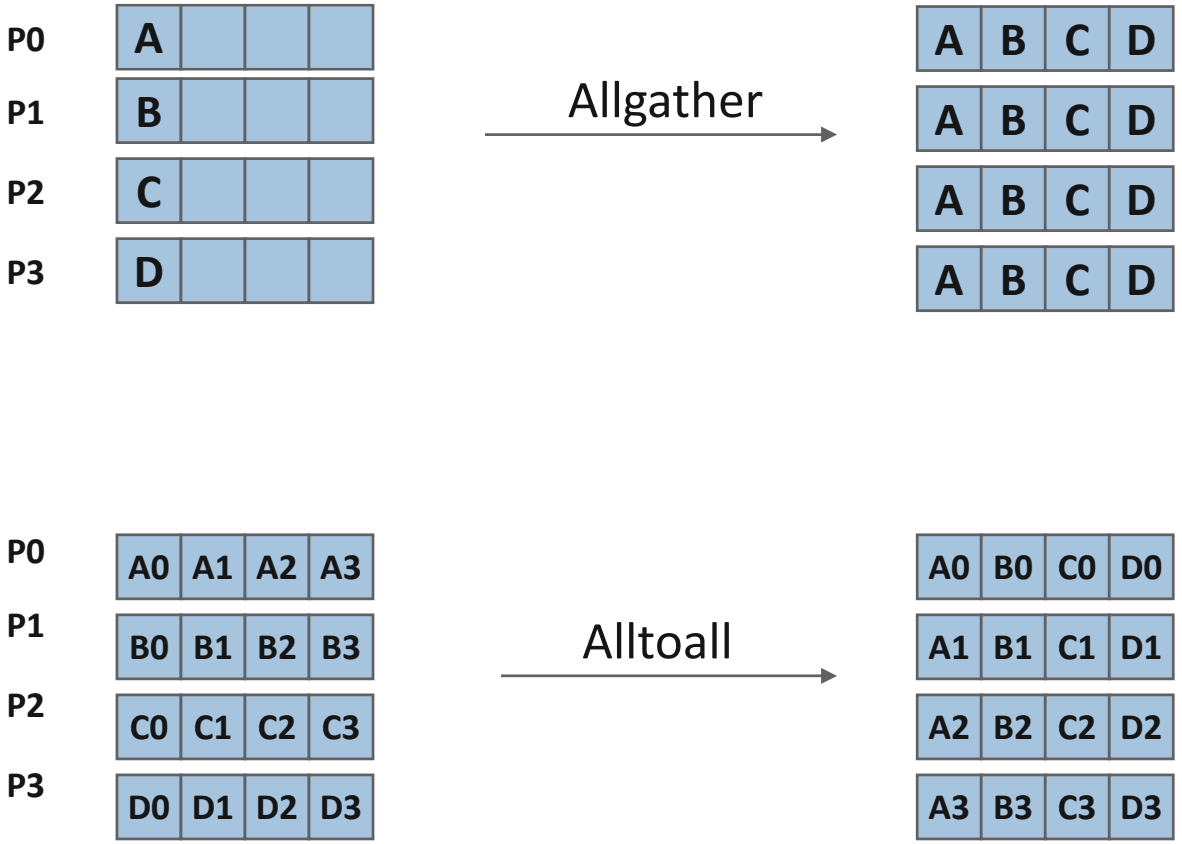
Synchronization

- **`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

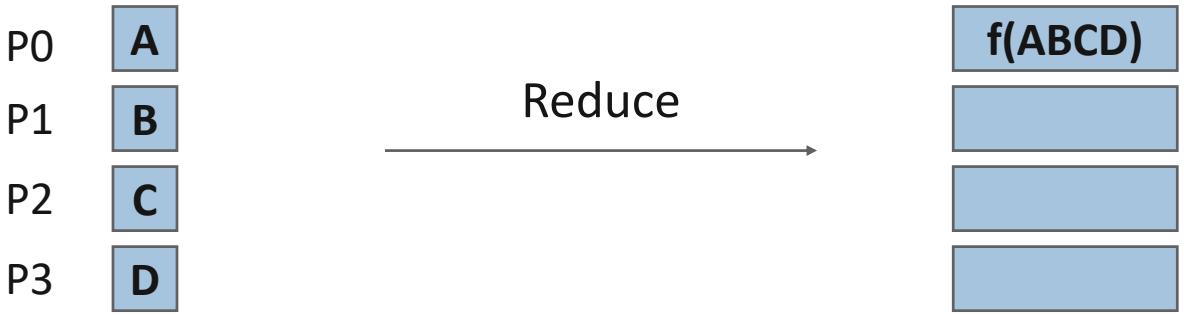
Collective Data Movement

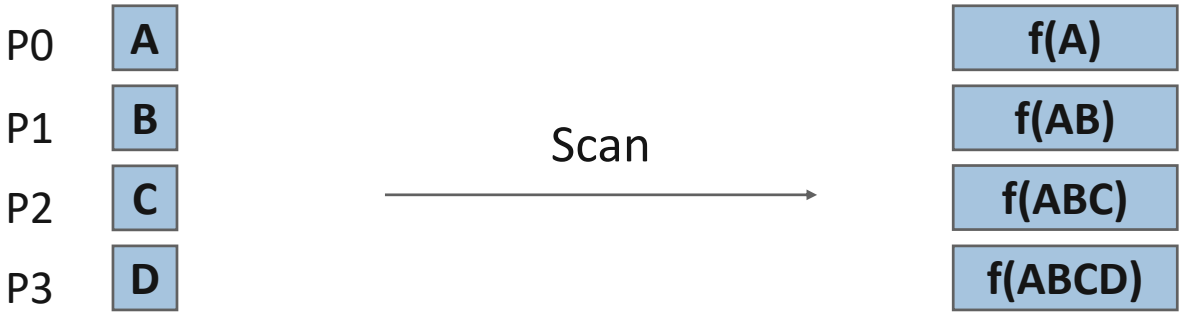


More Collective Data Movement



Collective Computation





MPI Collective Routines

- Many Routines: **MPI_ALLGATHER, MPI_ALLGATHERV, MPI_ALLREDUCE, MPI_ALLTOALL, MPI_ALLTOALLV, MPI_BCAST, MPI_GATHER, MPI_GATHERV, MPI_REDUCE, MPI_REDUCESCATTER, MPI_SCAN, MPI_SCATTER, MPI_SCATTERV**
- **"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

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:

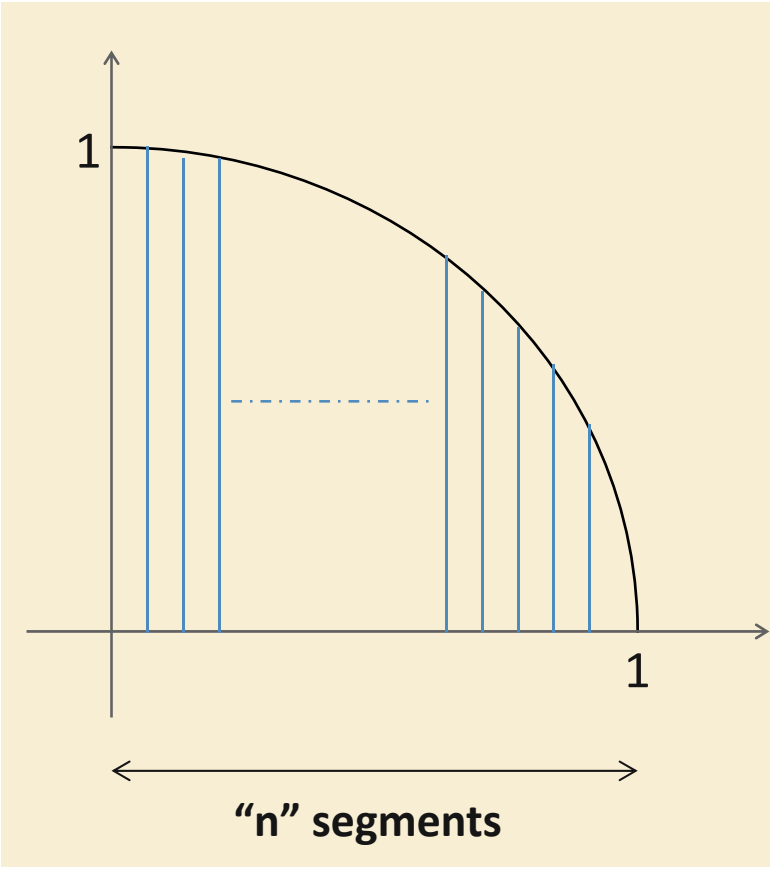
```
inoutvec[i] = invec[i] op inoutvec[i];
```

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)]^2)

Example: PI in C

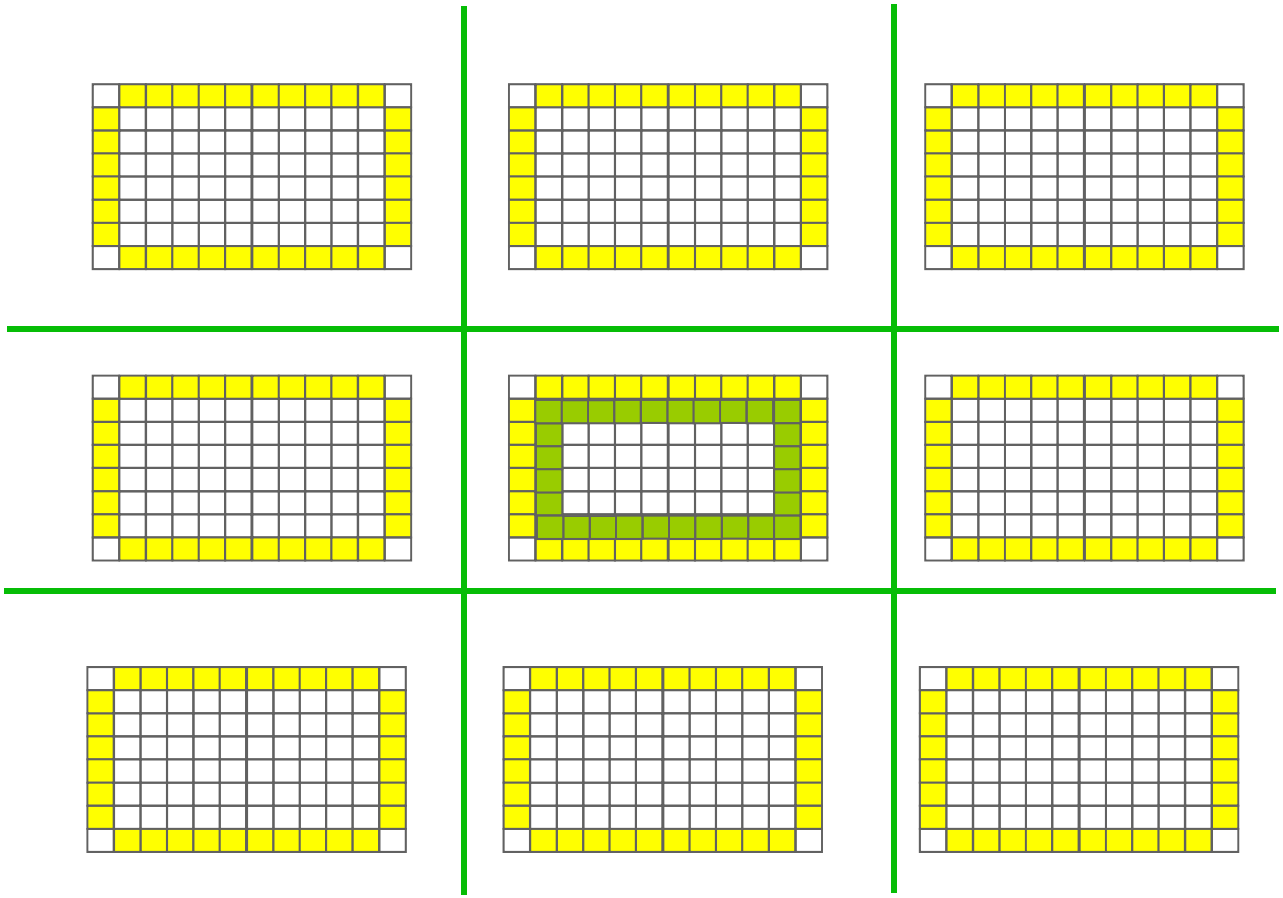
```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[])
{
    [...snip...]
    /* Tell all processes, the number of segments you want */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    w   = 1.0 / (double) n;
    mypi = 0.0;
    for (i = rank + 1; i <= n; i += size)
        mypi += w * sqrt(1 - (((double) i / n) * ((double) i / n));
    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...]
}
```

What we will cover in this tutorial

- What is MPI?
- How to write a simple program in MPI
- Running your application with MPICH
- **Slightly more advanced topics:**
  - Non-blocking communication in MPI
  - Group (collective) communication in MPI
  - **MPI Datatypes**
- Conclusions and Final Q/A

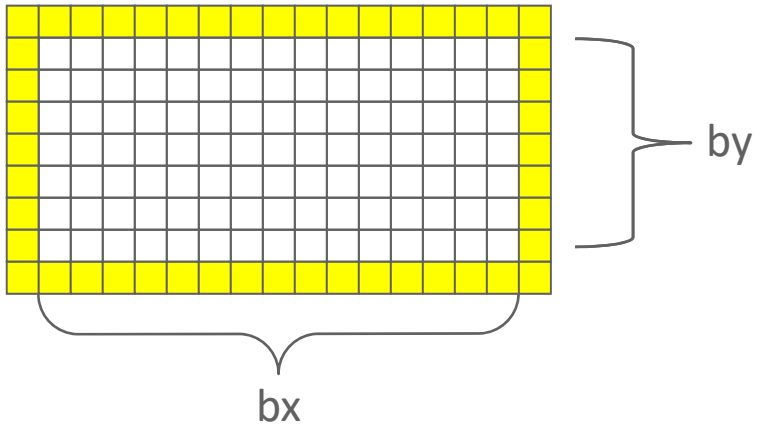
Necessary Data Transfers

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



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)



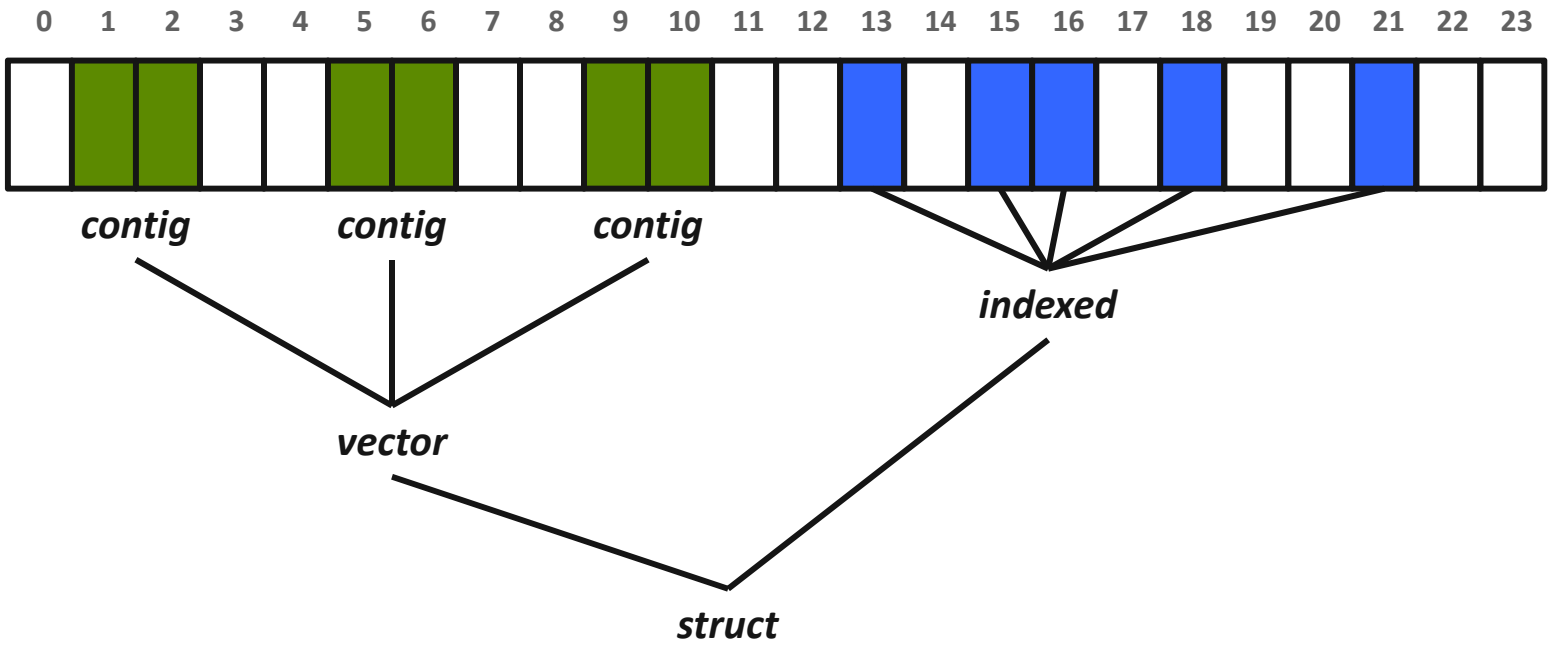
Introduction to Datatypes in MPI

- 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)

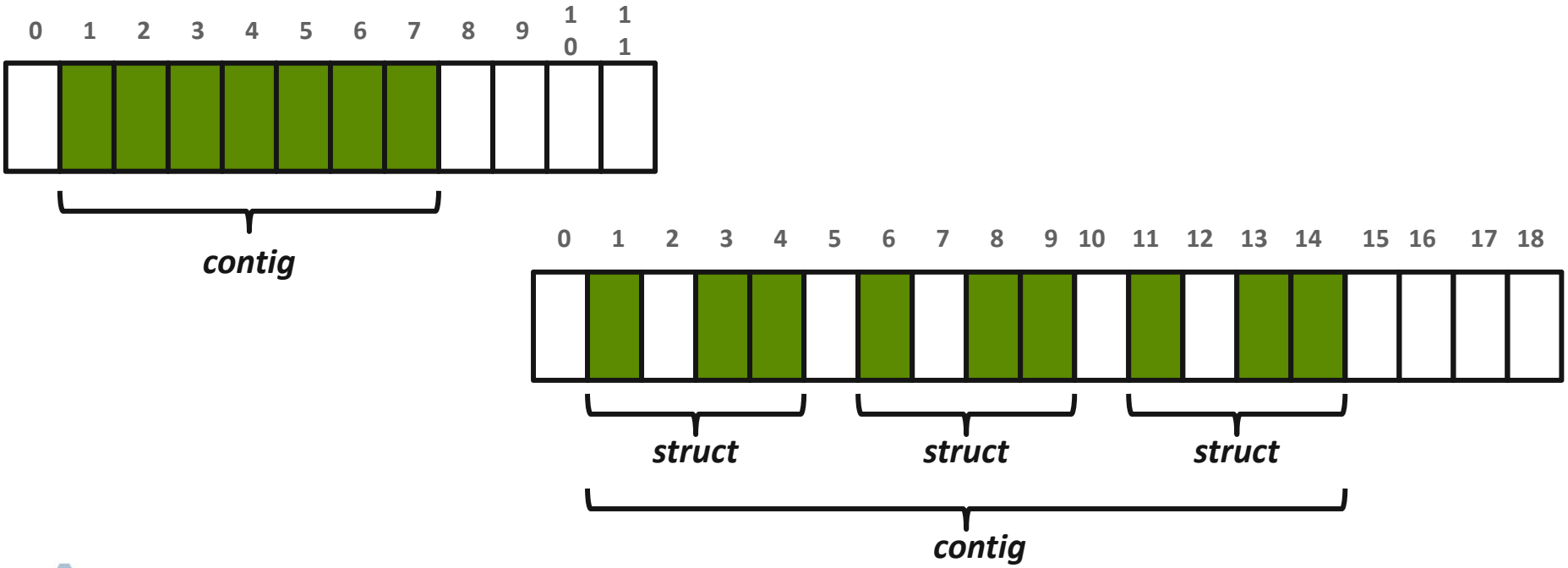
Derived Datatype Example



MPI_Type_contiguous

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

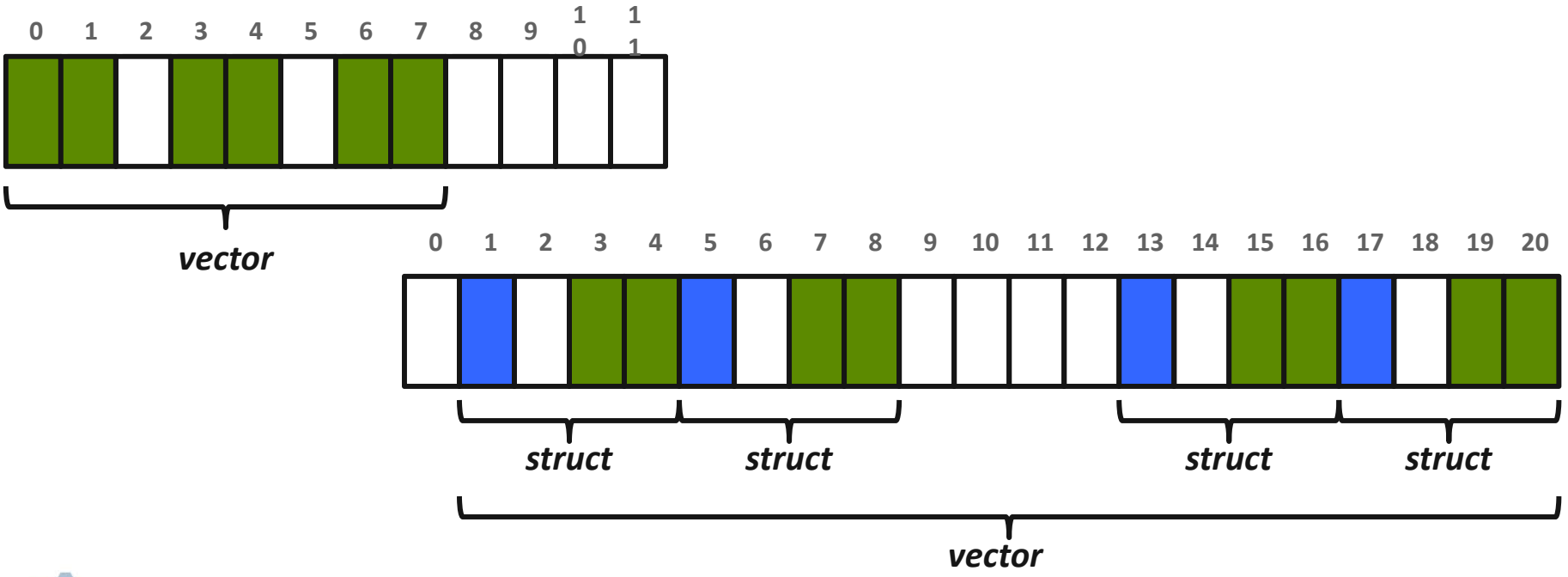
- Contiguous array of oldtype
- Should not be used as last type (can be replaced by count)



MPI_Type_vector

MPI_Type_vector(int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)

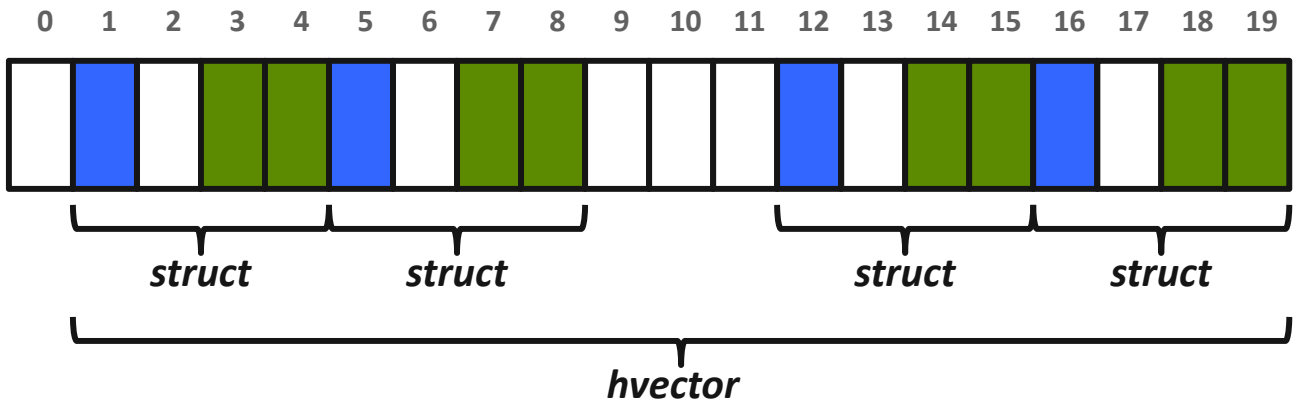
- Specify strided blocks of data of oldtype
- Very useful for Cartesian arrays



MPI_Type_create_hvector

```
MPI_Type_create_hvector(int count, int blocklength,
                        MPI_Aint stride, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
```

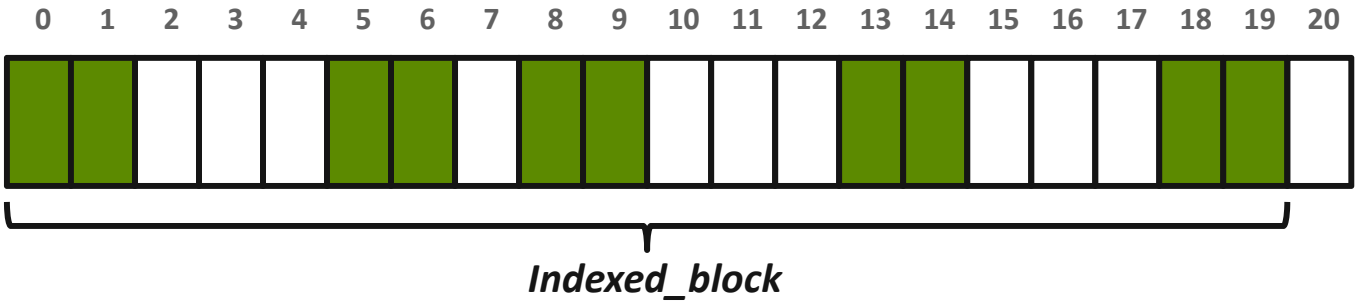
- Create non-unit strided vectors
- Useful for composition, e.g., vector of structs



MPI_Type_create_indexed_block

```
MPI_Type_create_indexed_block(int count, int blocklength,
                              int *array_of_displacements, MPI_Datatype oldtype,
                              MPI_Datatype *newtype)
```

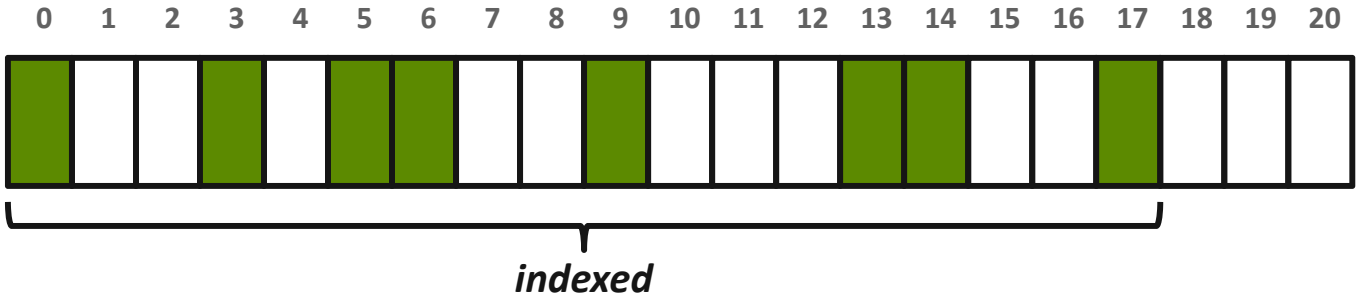
- Pulling irregular subsets of data from a single array
  - dynamic codes with index lists, expensive though!
  - blen=2
  - displs={0,5,8,13,18}



MPI_Type_indexed

```
MPI_Type_indexed(int count, int *array_of_blocklengths,
                 int *array_of_displacements, MPI_Datatype oldtype,
                 MPI_Datatype *newtype)
```

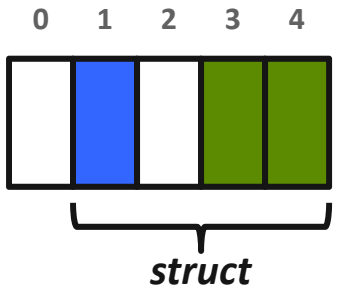
- Like indexed_block, but can have different block lengths
  - blen={1,1,2,1,2,1}
  - displs={0,3,5,9,13,17}



MPI_Type_create_struct

```
MPI_Type_create_struct(int count, int array_of_blocklengths[],
                       MPI_Aint array_of_displacements[],
                       MPI_Datatype array_of_types[],
                       MPI_Datatype *newtype)
```

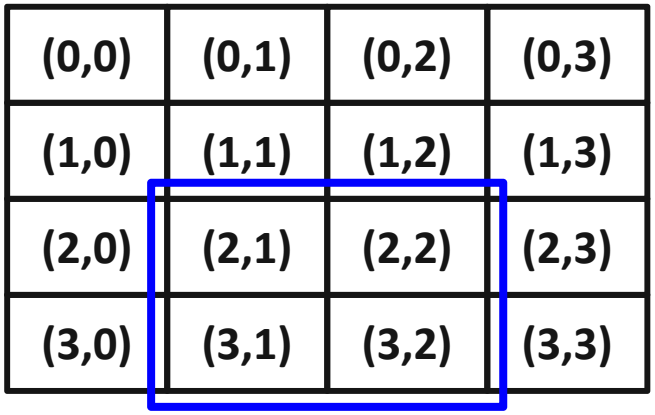
- Most general constructor, allows different types and arbitrary arrays (also most costly)



MPI_Type_create_subarray

```
MPI_Type_create_subarray(int ndims, int array_of_sizes[],
                         int array_of_subsizes[], int array_of_starts[],
                         int order, MPI_Datatype oldtype,
                         MPI_Datatype *newtype)
```

- Convenience function for creating datatypes for array segments
- Specify subarray of n-dimensional array (sizes) by start (starts) and size (subsize)



MPI_BOTTOM and MPI_Get_address

- MPI_BOTTOM is the absolute zero address
  - Portability (e.g., may be non-zero in globally shared memory)
- MPI_Get_address
  - Returns address relative to MPI_BOTTOM
  - Portability (do not use “&” operator in C!)
- Very important to
  - build struct datatypes
  - If data spans multiple arrays

Commit, Free, and Dup

- 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
- MPI_Type_dup
  - Duplicates a type
  - Library abstraction (composability)

Other DDT Functions

- Pack/Unpack
  - Mainly for compatibility to legacy libraries
  - You should not be doing this yourself
- Get_envelope/contents
  - Only for expert library developers
  - Libraries like MPITypes[1] make this easier
- MPI_Create_resized
  - Change extent and size (dangerous but useful)

http://www.mcs.anl.gov/mpitypes/

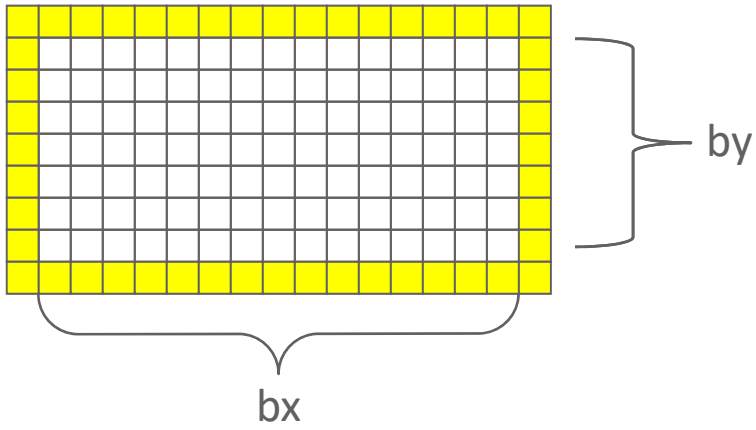
Datatype Selection Order

- Simple and effective performance model:
  - More parameters == slower
- **predefined < contig < vector < index_block < index < struct**
- Some (most) MPIs are inconsistent
  - But this rule is portable
- Advice to users:
  - Try datatype “compression” bottom-up

W. Gropp et al.: Performance Expectations and Guidelines for MPI Derived Datatypes

Code Example

- *stencil-mpi-ddt.c*
- Non-blocking sends and receives
- Data location specified by MPI datatypes
- Manual packing of data no longer required



What we will cover in this tutorial

- What is MPI?
- How to write a simple program in MPI
- Running your application with MPICH
- Slightly more advanced topics:
  - Non-blocking communication in MPI
  - Group (collective) communication in MPI
  - MPI Datatypes
- **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 scientific applications with great success
- Your application can be next!

Web Pointers

- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPICH : http://www.mpich.org
- MPICH mailing list: discuss@mpich.org
- MPI Forum : http://www.mpi-forum.org/

- Other MPI implementations:
  - MVAPICH (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