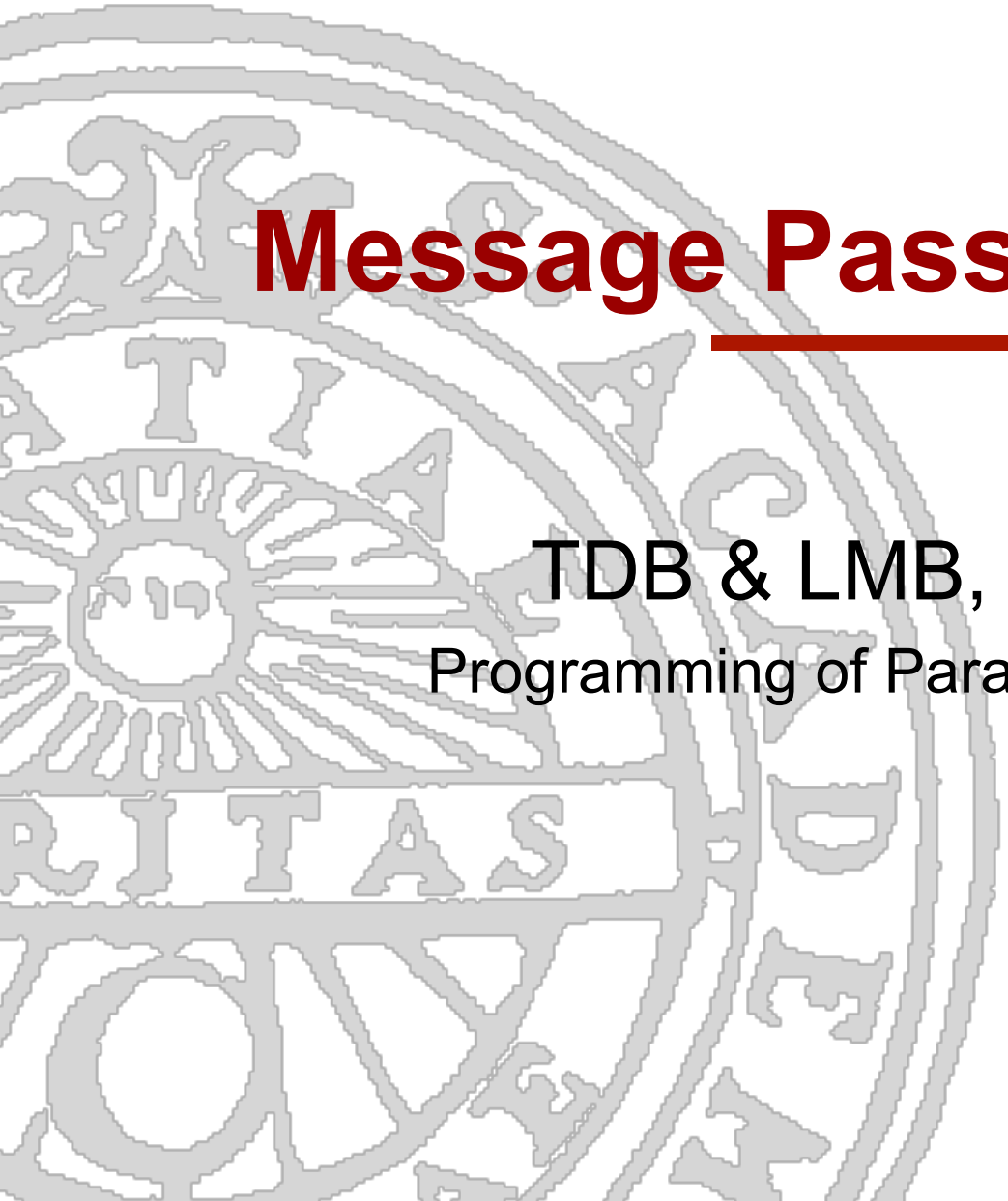


MPI

Message Passing Interface

Jing Liu

TDB & LMB, Uppsala University

Programming of Parallel Computers,Jan, 2015

What to Know...

- Preliminary knowledge
  - Programming in FORTRAN/C/C++/Java
  - Basics in hardware - CPU, RAM, Network
- Outcomes:
  - MPI: ready to go

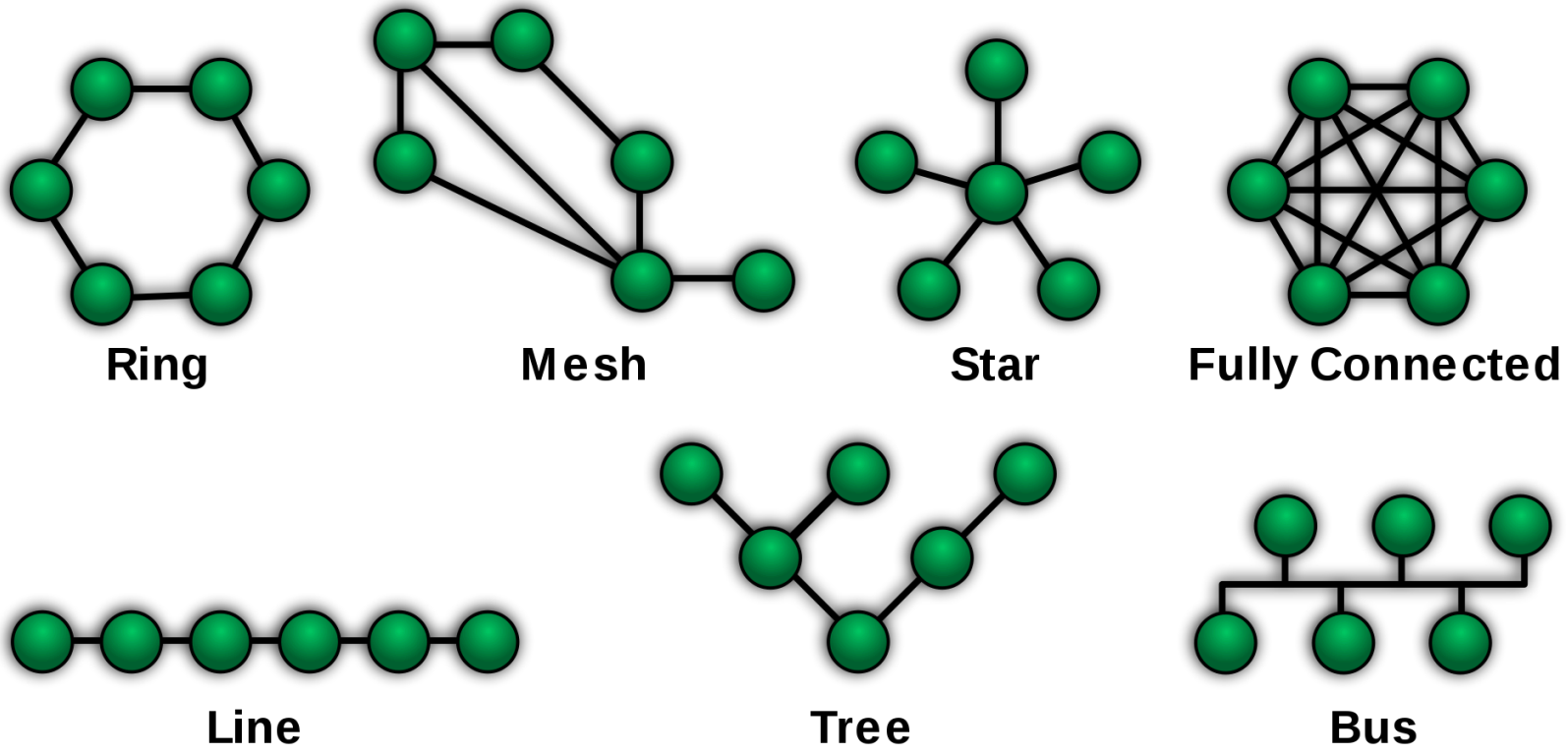
UPPSALA UNIVERSITET

Dep. of Information Technology

Outline

- Introduction and motivation
- Code Body
- Communicators
- Send and Receive
- Other Point-to-Point Functions
- Global Functions
- Datatypes
- Topology

Distributed Computing



Distributed Computing -- Paradigms

- Communication Models:
  - Message Passing
  - Shared Memory
- Computation Models:
  - Functional Parallel
  - Data Parallel

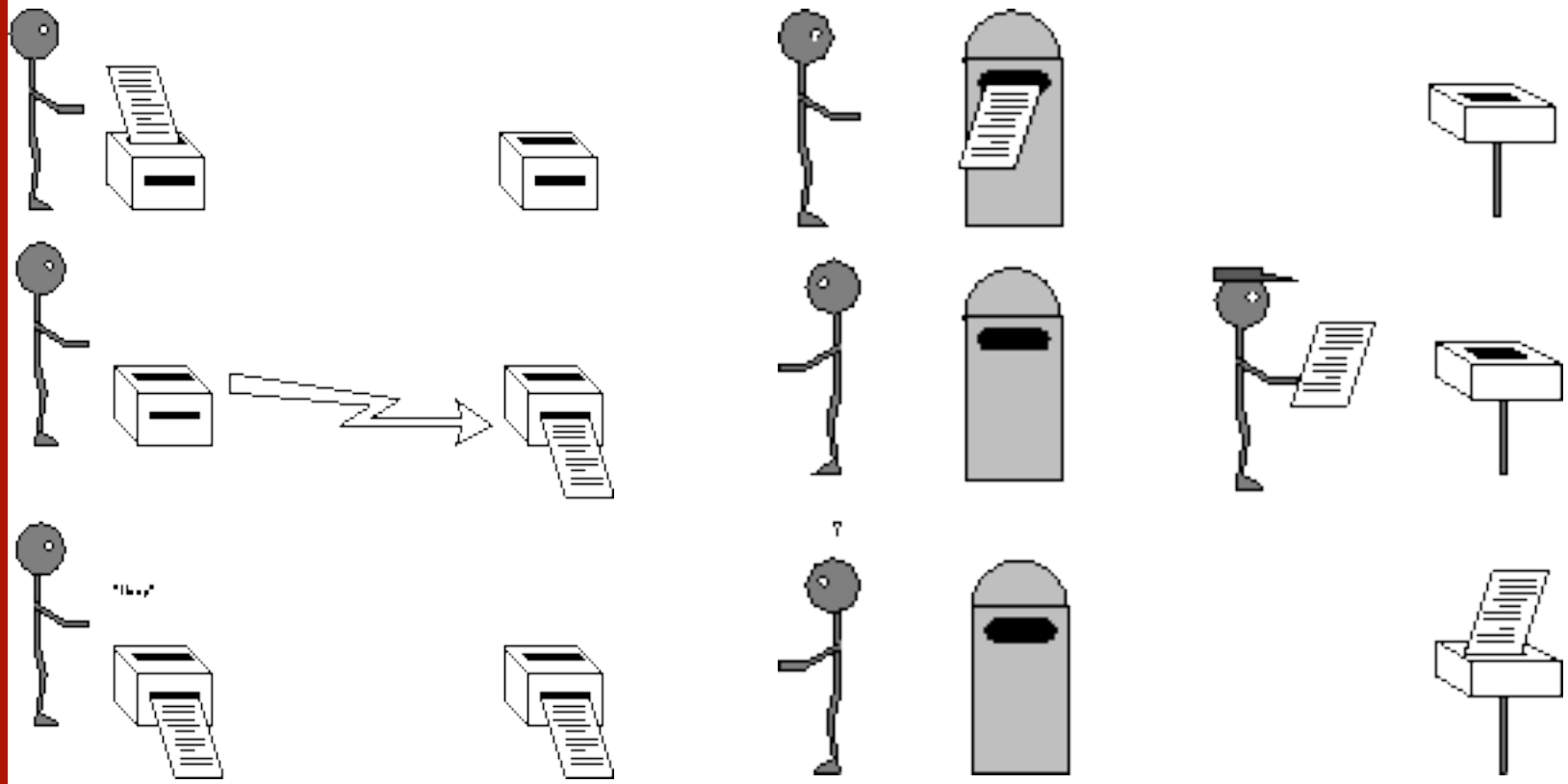
Distributed Computing

- How to communicate?
  - Sending data
  - Receiving data
  - Waiting for data
  - Waiting for synchronization

UPPSALA UNIVERSITET

Dep. of Information Technology

Synchronous Vs. Asynchronous



UPPSALA UNIVERSITET

Dep. of Information Technology

What is MPI?

- A message-passing library specifications:
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or produce
- For parallel computers, clusters, and heterogeneous networks.
- Designed to permit the development of parallel software libraries.

Brief History

- 1992 - draft of the project
- 1994 - first version MPI 1.0
  - Point-to-point
  - Global communication, groups
- 1997 - MPI 2.0
  - One-sided communication
  - Dynamic management
- 2008/09 - MPI 2.1 and 2.2
- 2012 - MPI-3.0
  - New one-sided communication

UPPSALA UNIVERSITET

Dep. of Information Technology

Outline

- Introduction and motivation
- Code Body
- Communicators
- Send and Receive
- Other Point-to-Point Functions
- Global Functions
- Datatypes
- Topology

Hello MPI

```
#include <mpi.h> // header file to use MPI
int main(int argc, char** argv) {
  // Initialize the MPI environment
  MPI_Init(NULL, NULL);
  // Get the number of processes
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  // Get the rank of the process
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  // Get the name of the processor
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  MPI_Get_processor_name(processor_name, &name_le

  // Print off a hello world message
  printf("Hello MPI from processor %s, rank %d"
         " out of %d processors\n",
         processor_name, world_rank, world_size);
  // Finalize the MPI environment.
  MPI_Finalize();
}
```

A MPI program

- Code body MUST have:
  - Header file: #include <mpi.h>
  - MPI Init(&agrc, &argv);
  - MPI Finalize();
- Compilation
  - Mpicc -o program program.c
  - mpiCC -o program program.cpp
- Execution
  - mpirun -np N ./program

UPPSALA UNIVERSITET

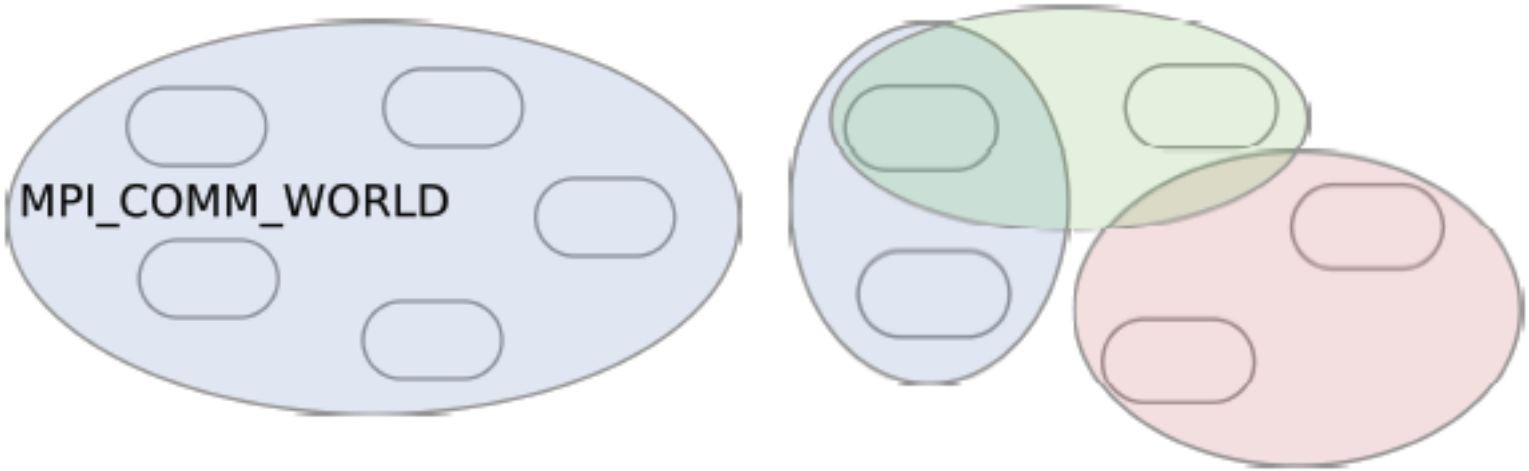
Dep. of Information Technology

Outline

- Introduction and motivation
- Code Body
- Communicators
- Send and Receive
- Other Point-to-Point Functions
- Global Functions
- Datatypes
- Topology

Communicator

- Groups and communicators are two important concepts



  - How to range your datasets / communication patterns/ algorithm/ …

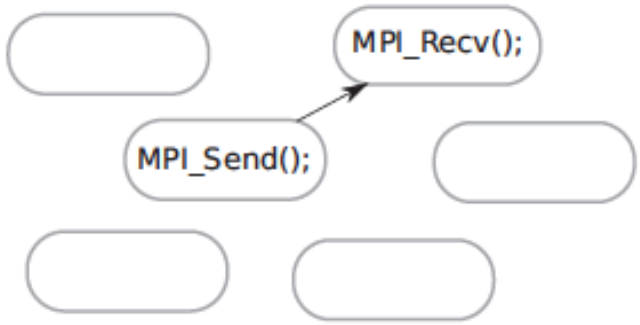
UPPSALA UNIVERSITET

Dep. of Information Technology

Outline

- Introduction and motivation
- Code Body
- Communicators
- Send and Receive
- Other Point-to-Point Functions
- Global Functions
- Datatypes
- Topology

Send and Receive

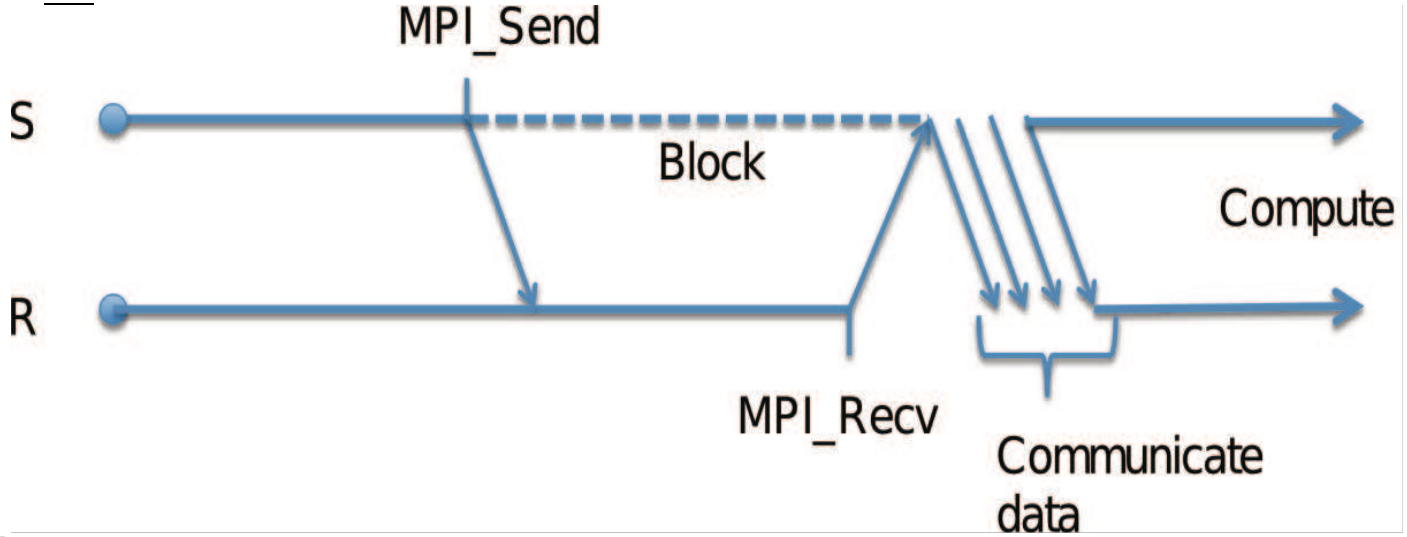


- Point to point communication
- 4 basic communication modes
  - Standard: MPI_Send / MPI_Isend
  - Synchronous: MPI_Ssend / MPI_Issend
  - Ready: MPI_Rsend / MPI_Irsend
  - Buffered: MPI_Bsend / MPI_Ibsend
- Blocking vs. Non-Blocking

Read more about communication mode in section 3.4 at MPI document

Send and Receive (cont.)

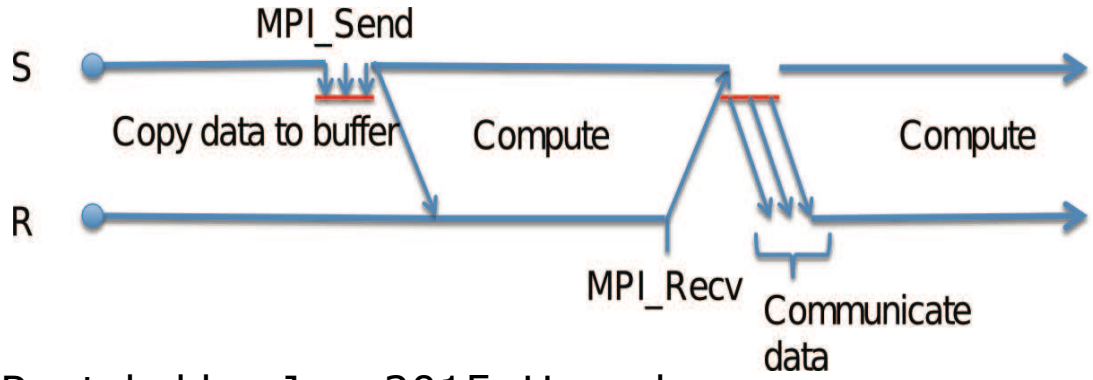
- MPI_Send – standard



  - S is blocked until data has been sent
  - MPI_Recv can be post before MPI_Send
  - R & S shake hand,

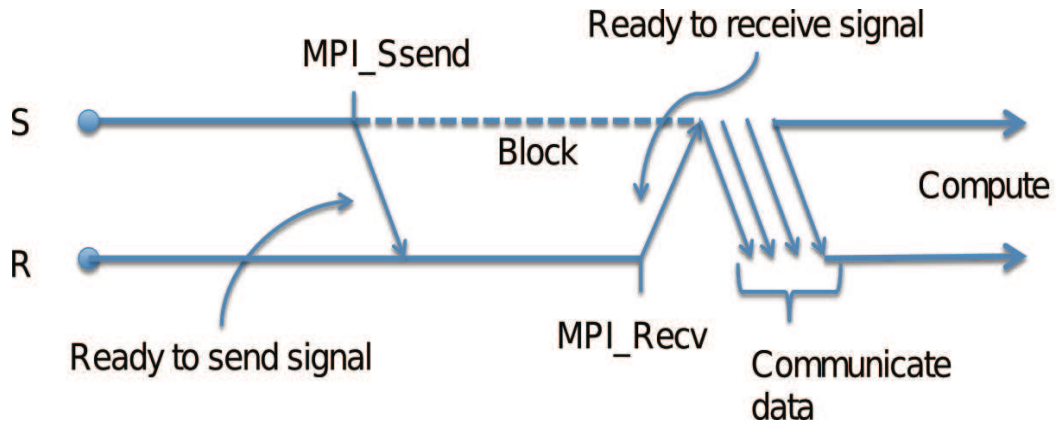
Send and Receive (cont.)

- MPI_Send with a very small dataset
  - An eager protocol may be used.
  - S assume that R can store a small message
  - R has the responsibility to buffer the message upon its arrival, especially if the receive operation has not been posted.



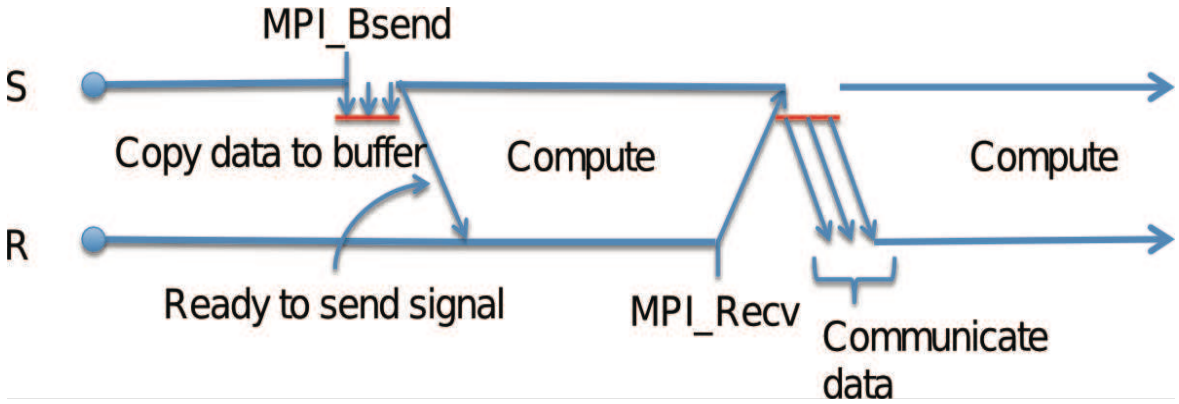
Send and Receive (cont.)

- MPI_Ssend
  - Synchronous
  - S waits until the receive has been posted on the receiving end.



Send and Receive (cont.)

- Buffered MPI_Bsend
  - S returns after coping data to a user-supplied communication buffer.
  - Safe to modify the original data.
  - S blocks also when data is transferring.

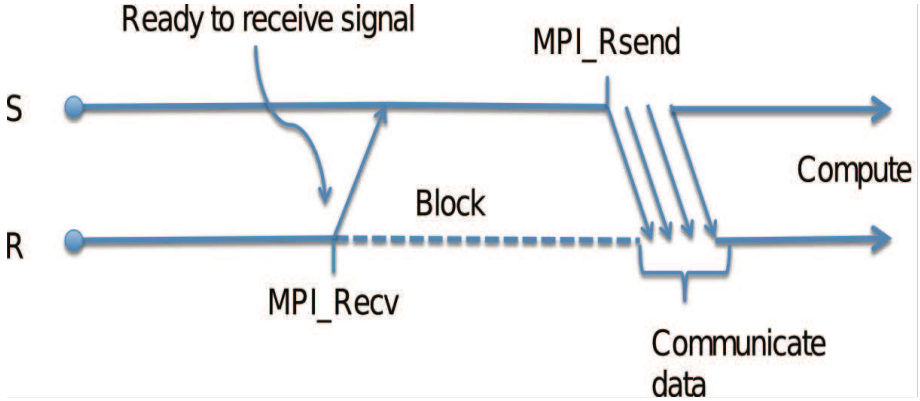


Send and Receive (cont.)

- MPI_Bsend cont.
  - Explicitly allocate the buffer first.
    - int buflen = totlen*sizeof(double) + MPI_BSEND_OVERHEAD;
    - double* buffer = malloc(buflen);
    - MPI_Buffer_attach(buffer, buflen);
    - MPI_Bsend(data,count,type,dest,tag,comm);
  - Explicitly use wait/test to ensure that the communication has completed and it is safe to modify the data.

Send and Receive (cont.)

- MPI_Rsend
  - Ready mode
  - It notifies the system that a matching receive is already posted. That information can save some overhead.
  - Replace MPI_Send()

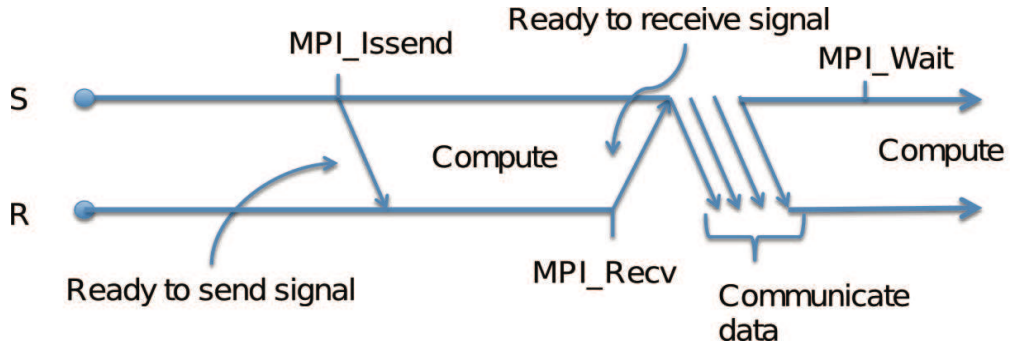


Send and Receive (cont.)—blocking message send

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

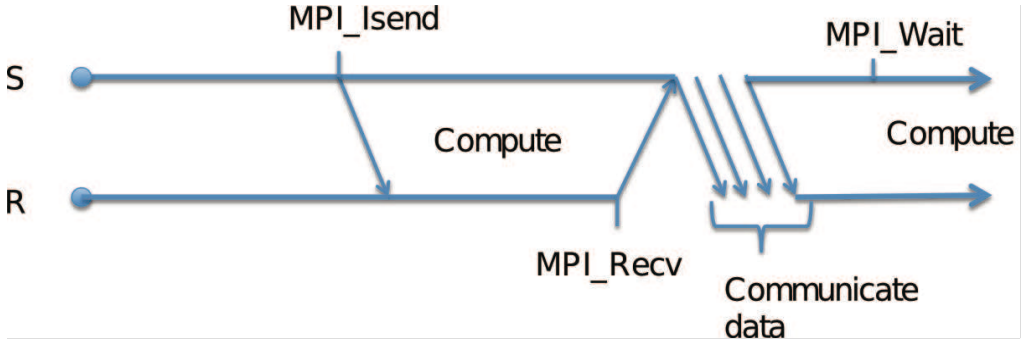
Send and Receive (cont.) – Non-blocking

- MPI_Issend
  - Initiates a synchronous mode send
  - S is not blocked
  - Explicitly use wait/test to ensure that the data buffer is safe to use.



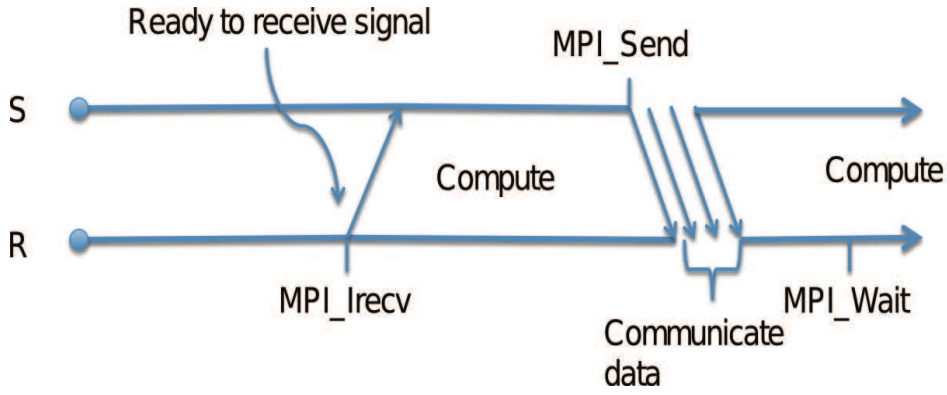
Send and Receive (cont.) – Non-blocking

- MPI_Isend
  - Similar to MPI_Issend
  - May return before data is copied out of buffer
  - Explicitly use wait/test to ensure that the data buffer is safe to use.



Send and Receive (cont.) – Non-blocking

- Non-blocking MPI_Irecv
  - May return before the message is received into the buffer.
  - Explicitly use wait/test to ensure data is safe to use
  - No blocking



Send and Receive (cont.)

- All send calls need to be matched with a receive call!
- Blocking suspends the execution until the message (data) buffer is safe to use (been sent/ received/copied).
- Non-blocking calls returns immediately after initiating the communication.
- Deadlock will occur if no matching send/ receive!

Send and Receive (cont.)

- Tips:
  - Use non-blocking operations if possible
  - Post non-blocking operations as early as possible, so that communications overlap with computations.
  - In most cases, the standard non-blocking operations are sufficient.

Find APIs here

Outline

- Introduction and motivation
- Code Body
- Communicators
- Send and Receive
- Other Point-to-Point Functions
- Global Functions
- Datatypes
- Topology

Other P2P Functions

- MPI_Wait / MPI_Waitall / MPI_Waitany / MPI_Waitsome
  - Wait for (a specified / all / any/ some specified) request(s) to complete
- MPI_Test / MPI_Testall / MPI_Testany / MPI_Testsome
  - Test the completion of (a specified / all / any/ some specified) request(s)
  - It returns immediately

Other P2P operations

- MPI_Probe, MPI_Iprobe
  - Allow checking of incoming messages without actual receipt of them
  - May allocate memory for the receive buffer, according to the probing.
- MPI_Cancel
  - Cancel a pending communication
  - User resources need to be free

Outline

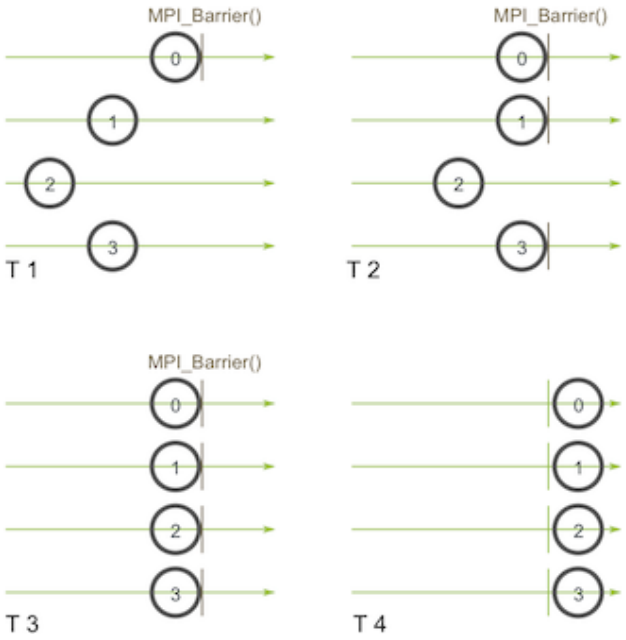
- Introduction and motivation
- Code Body
- Communicators
- Send and Receive
- Other Point-to-Point Functions
- Global Functions
- Datatypes
- Topology

Global functions

- Global functions act on every process in a group (communicator).
  - Barrier: all process wait
  - Broadcast: send to all
  - Reduce: collect data
  - Scatter: send to all
  - Gather: receive from all

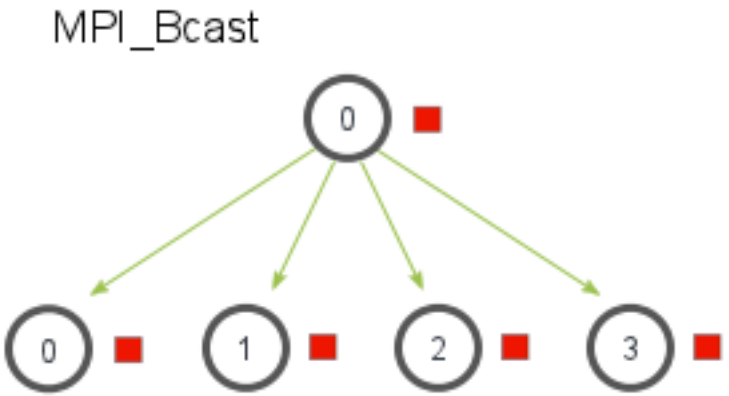
Global functions (cont.)

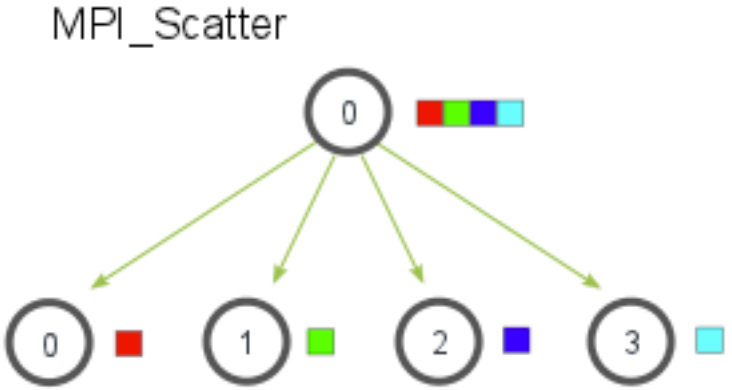
- int MPI Barrier(MPI_Comm comm);
  - Wait until every processes post this function



Global functions(cont.)

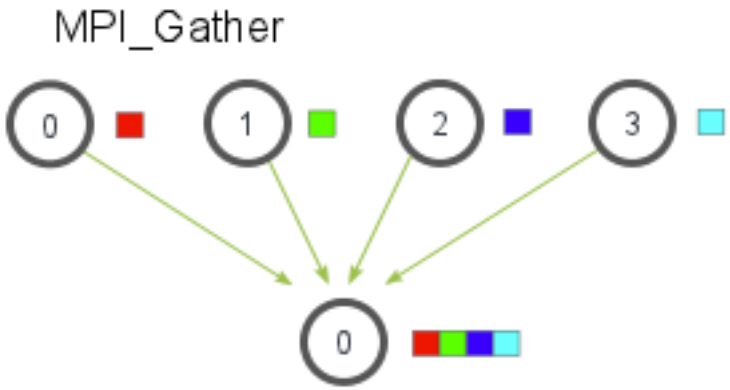
- Broadcast
  - int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
- Scatter
  - int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int rcount, MPI_Datatype recvtype, int root, MPI_Comm comm);





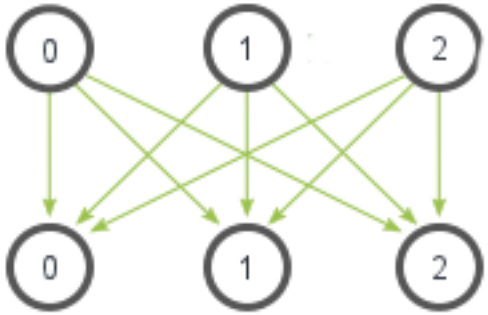
Global functions(cont.)

- MPI_Gather
  - Inverse of MPI_Scatter
  - Highly useful for parallel sorting and searching
- int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int rcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

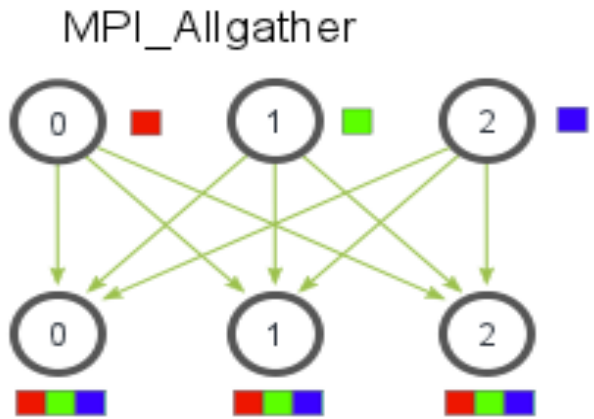


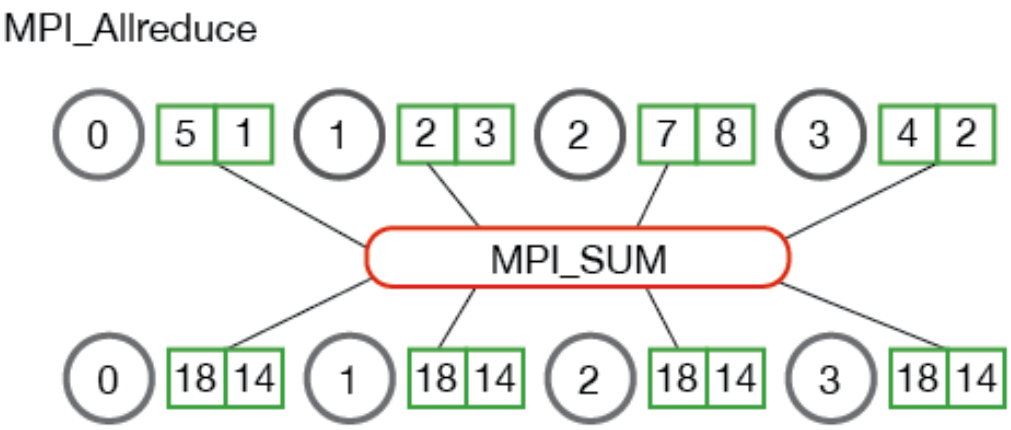
Global functions(cont.)

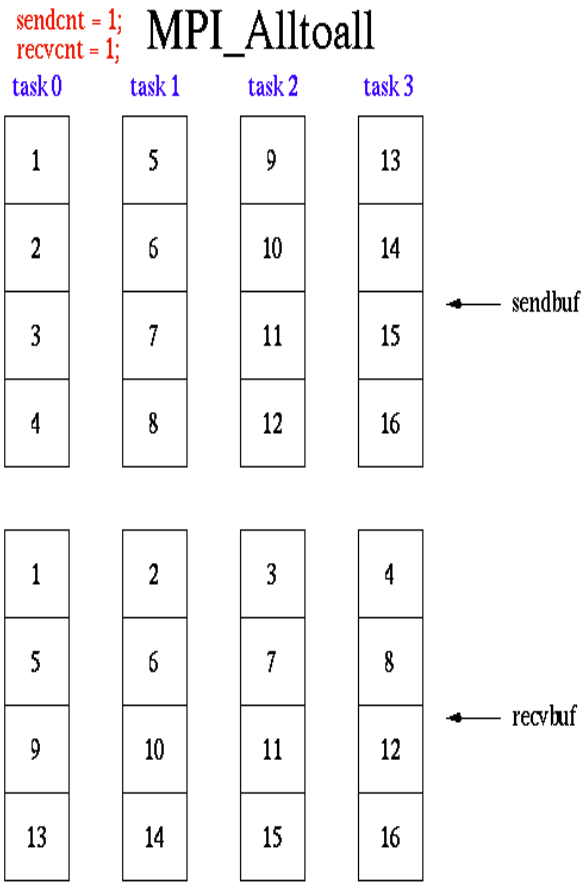
- MPI_All*
  - Many-to-Many communication pattern
    - MPI_Allreduce
    - MPI_Allgather
    - MPI_Alltoall



Global functions(cont.)





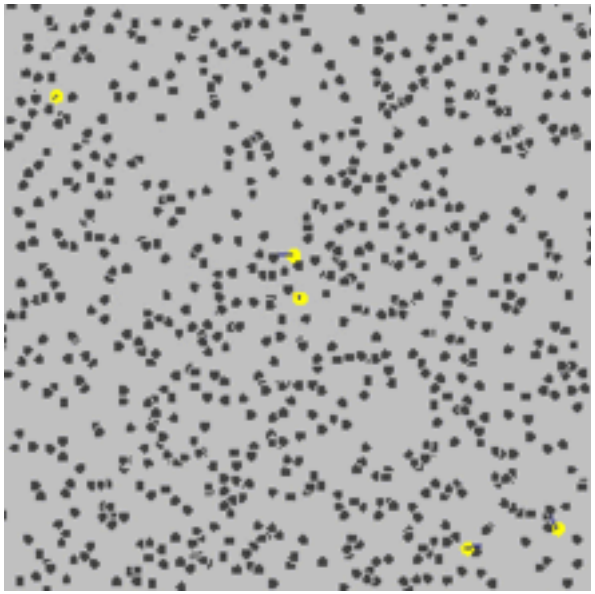


Global functions(cont.)

- For the MPI_*reduce
  - MPI_MAX – Returns the maximum element.
  - MPI_MIN – Returns the minimum element.
  - MPI_SUM – Sums the elements.
  - MPI_PROD – Multiplies all elements.
  - MPI_LAND – Performs a logical “and” across the elements.
  - MPI_LOR – Performs a logical “or” across the elements.
  - MPI_BAND – Performs a bitwise “and” across the bits of the elements.
  - MPI_BOR – Performs a bitwise “or” across the bits of the elements.
  - MPI_MAXLOC – Returns the maximum value and the rank of the process that owns it.
  - MPI_MINLOC – Returns the minimum value and the rank of the process that owns it.

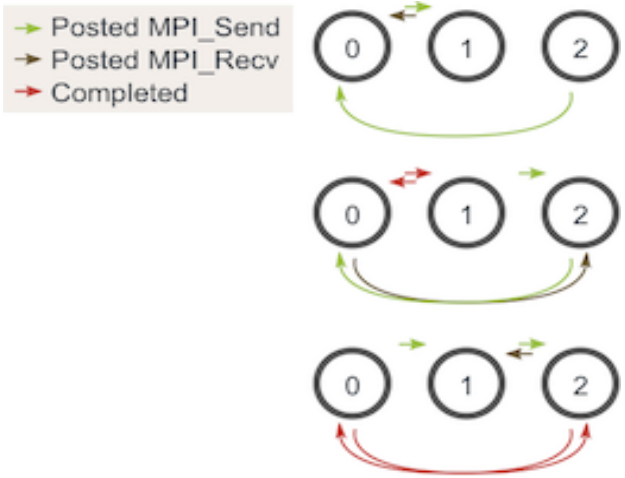
P2P case study

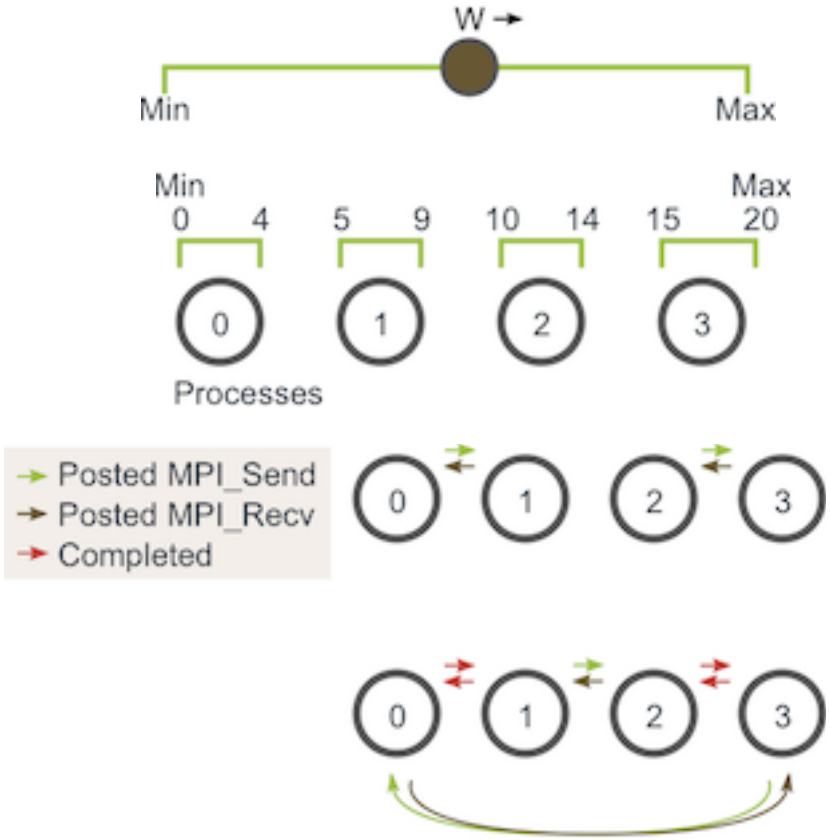
- Random walk and particle tracing
  - Particle moves randomly in a fluid – Brownian motion
  - Performing parallel particle tracing can be very difficult
    - Randomly move among grids
    - Unbalanced computation
  - Random walk



P2P case study

- A simple random walk in MPI
  - MPI_Send
  - MPI_Recv
  - MPI_Probe





J. Liu, J.Rantakokko, Jan, 2015, Uppsala

Global functions—case study

- Some optimization is done with the global functions
- Compare MPI_Bcast to for loop Send/ Recv
- mpirun -n 16 ./DemoBcast2 1000000 10

  Data size = 4000000 bytes, Trials = 10 times

  Avger SR_bcast time = 0.015581 seconds

  Avg MPI_Bcast time = 0.004403 seconds

UPPSALA UNIVERSITET

Dep. of Information Technology

Outline

- Introduction and motivation
- Code Body
- Communicators
- Send and Receive
- Other Point-to-Point Functions
- Global Functions
- Datatypes
- Topology

Datatypes

MPI datatypeÅ	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_PACKED	-
MPI_BYTE	-

Datatypes (cont.)

- Derived Datatype
  - **MPI_Type_contiguous**
    - Produces a new datatype by making count copies of an existing data type.
  - **MPI_Type_vector**
    - Similar to contiguous, but allows for regular gaps (stride) in the displacements.
  - **MPI_Type_indexed**
    - An array of displacements of the input data type is provided as the map for the new data type.

Datatypes (cont.)

- Derived Datatype (cont.)
  - **MPI_Type_struct**
    - The most general of all derived datatypes. The new data type is formed according to completely defined map of the component data types.
- Allocate / de-allocate datatype
  - int MPI_Type_commit (MPI_datatype *datatype)
  - int MPI_Type_free (MPI_datatype *datatype)

Read more at Derived Data Types with MPI

Datatypes (cont.)

- int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - MPI_Datatype newtype;
  - MPI Type vector(3,2,4,MPI_INT,&newtype);
  - MPI_Commit(&newtype);
  - MPI_Send(&A[0][1],1,newtype,1,0,comm)

1	2	3	4
5	6	7	8
9	10	11	12

- Sends new array [2 3 6 7 10 11] to process 1
  - count * blocklength = # of elements to be sent

Datatypes (cont.)

- int MPI_Type_indexed(int count, const int *blocklengths, const int *disp, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - Use different block sizes
  - Use different displacement

  Demo

UPPSALA UNIVERSITET

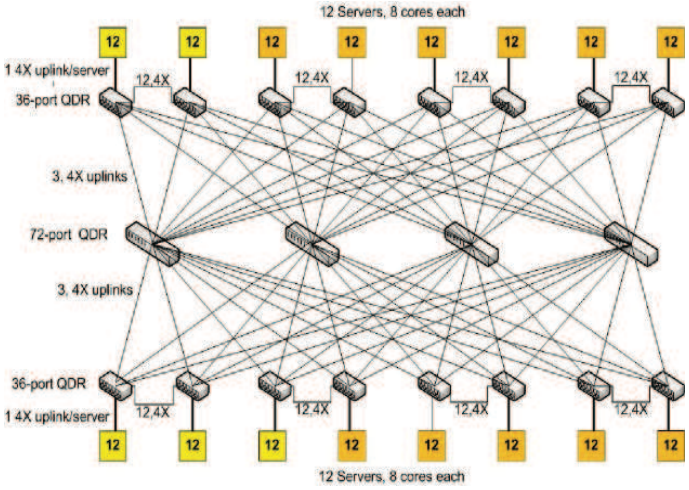
Dep. of Information Technology

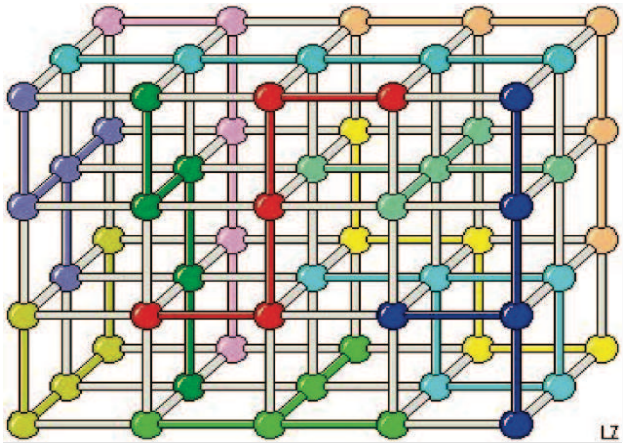
Outline

- Introduction and motivation
- Code Body
- Communicators
- Send and Receive
- Other Point-to-Point Functions
- Global Functions
- Datatypes
- Topology

Topology

- Virtual Topology VS physical topology
  - Use different communicator
  - Optimizing your algorithm communications





Topology (cont)

- Cartesian Topology

int MPI_Cart_create (MPI_Comm commold , int ndims , int *dims , int * periods, int reorder, MPI_Comm *commcart)

  - Creates a new communicator with Cartesian topology of arbitrary dimension
    - commold input communicator
    - ndims number of dimensions of Cartesian grid
    - dims array of size ndims specifying the number of processes in each dimension
    - periods logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
    - reorder ranking of initial processes may be reordered (true) or not (false)
    - comm cart communicator with new Cartesian topology

Topology (cont)

- old_comm = MPI_COMM_WORLD;
- ndims = 2;
- dim_size[0] = 3;
- dim_size[1] = 2;
- periods[0] = 0;
- periods[1] = 0;
- reorder = 0;
- MPI_Cart_create(old_comm, ndims, dim_size,periods, reorder, &new_comm);

Topology (cont)

- MPI_Cart_rank
  - Determines process rank in communicator given Cartesian location
- MPI_Cart_coords
  - Determines process coordinates in Cartesian topology given rank in group
- Demo time!!

MPI TIMING

- MPI_Wtime returns an elapsed time (in second) on the calling processor

```
double start_time;
start_time = MPI_Wtime();
...
// Compute
...
double finish_time;
finish_time = MPI_Wtime();
// Elapsed time
double elapsed_time;
elapsed_time = finish_time - start_time;
```

UPPSALA UNIVERSITET

Dep. of Information Technology

Read more on:

- MPI APIs and examples
- MPICH official website
- OpenMPI v1.8