Programming Models Case Study: MPI

Ruini Xue

School of Computer Science and Engineering

University of Electronic Science and Technology of China

2016

What is MPI?

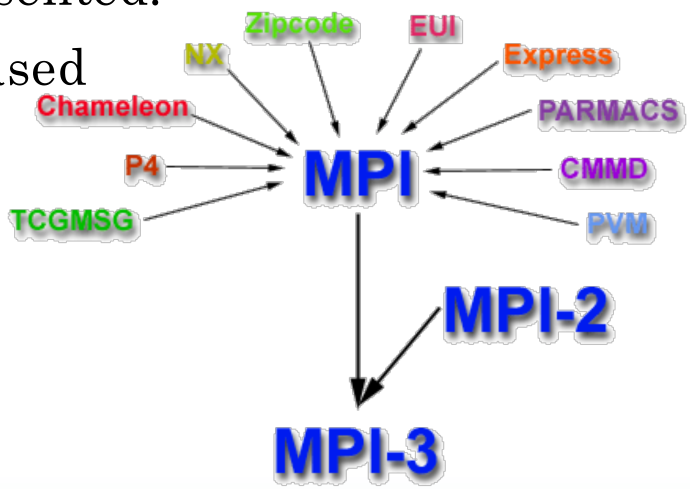
- Message Passing Interface
- a **specification** for the developers and users of message passing libraries.
  - NOT a library, but the specification of what such a library should be.
- addresses the message-passing parallel programming model: *data is moved from the address space of one process to that of another process through cooperative operations on each process*.
  - Network programming: send, recv
- not an IEEE or ISO standard, but the de-facto "industry standard"

MPI

- The programming model clearly remains a ***distributed memory model*** however, runs on virtually any hardware platform:
  - Distributed Memory
  - Shared Memory
  - Hybrid
- All parallelism is explicit
  - the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

MPI History

- 1980s - early 1990s: Distributed memory, parallel computing develops, as do a number of incompatible software tools for writing such programs. Recognition of the need for a standard arose.
- Apr 1992: Workshop on Standards for Message Passing. Preliminary draft proposal developed.
- Nov 1992: MPI draft proposal (MPI1) from ORNL presented, form the *MPI Forum*, comprised of about 175 individuals from 40 organizations.
- Nov 1993: SC'93 draft MPI standard presented.
- May 1994: Final version of MPI-1.0 released
- MPI-1.1 (Jun 1995),
  - MPI-1.2 (Jul 1997), MPI-1.3 (May 2008)
- 1996, MPI-2, brand new
  - MPI-2.1 (Sep 2008), MPI-2.2 (Sep 2009)
- Sep 2012: The MPI-3.0



Why MPI instead of send/recv?

- 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, e.g,. PVM
- Portability
  - little or no need to modify source code when porting application to a different platform that supports MPI.
- Performance Opportunities
  - Vendor implementations to exploit native hardware features to optimize performance. E.g., Infiniband
- Functionality
  - over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
- Availability
  - A variety of implementations are available, both vendor and public domain.

Famous Implementation

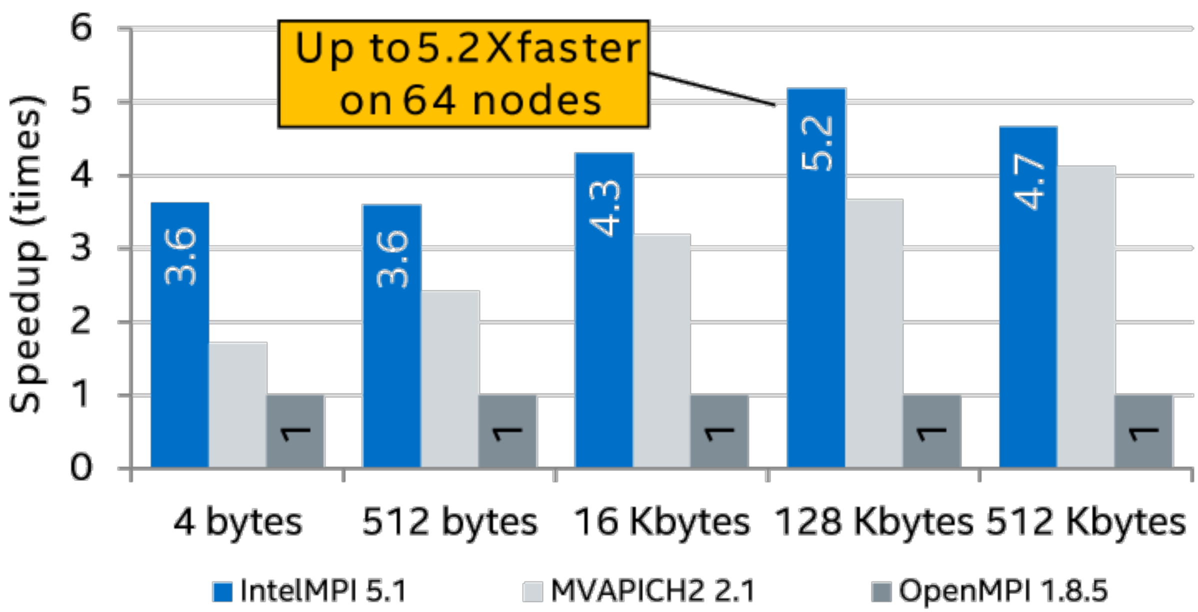
- C, Fortran bindings
  - C++, Java, Python
- MPICH @ ANL
  - high performance and widely portable implementation of the Message Passing Interface (MPI) standard
- MVAPICH @ Ohio State University
  - MPI over InfiniBand, 10GigE/iWARP and RoCE
- LAM/MPI → Open-MPI
  - an open source MPI implementation that is developed and maintained by a consortium of academic, research, and industry partners.
- Intel® MPI Library
  - Making applications perform better on Intel® architecture-based clusters with multiple fabric flexibility

Intel MPI

Superior Performance with Intel® MPI Library 5.1

1792 Processes, 64 nodes (InfiniBand + shared memory), Linux* 64

Relative (Geomean) MPI Latency Benchmarks (Higher is Better)

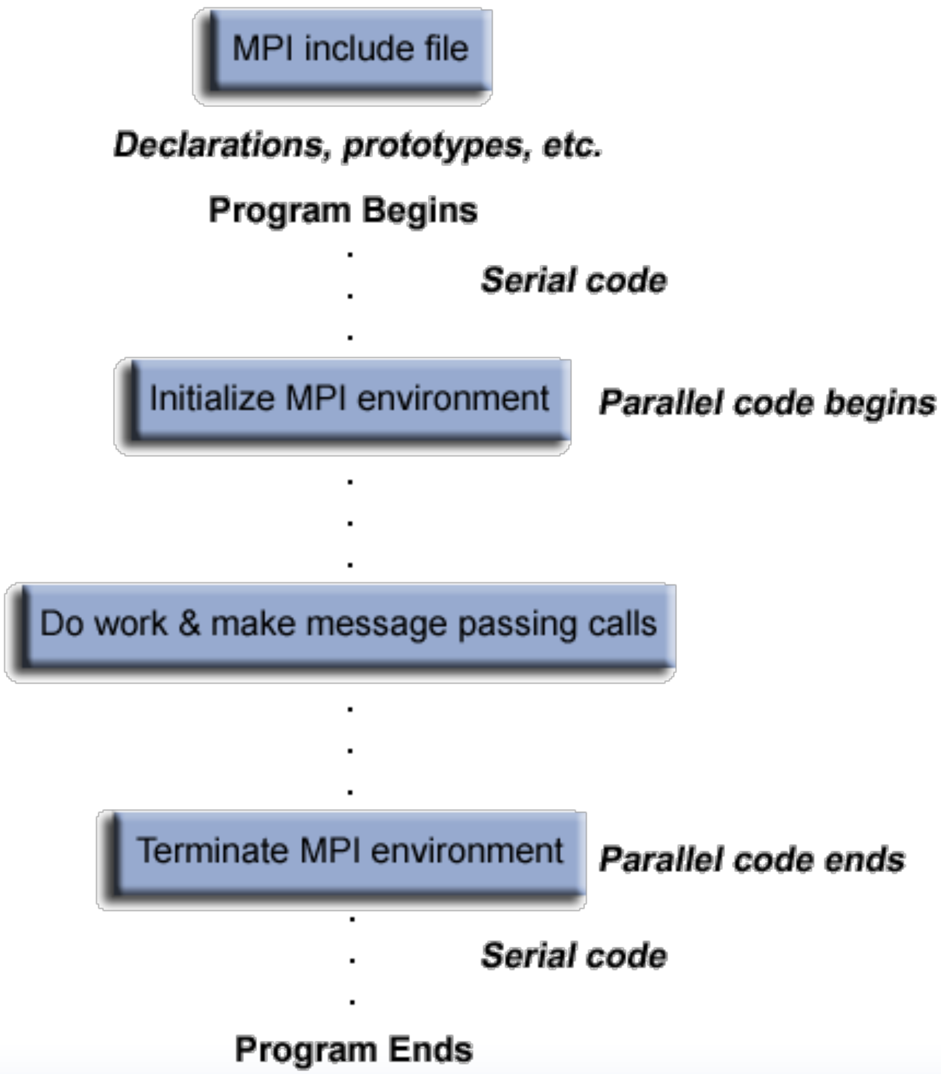


Configuration: Hardware: CPU: Dual Intel® Xeon E5-2697v3@2.60Ghz; 64 GB RAM. Interconnect: Mellanox Technologies* MT27500 Family [ConnectX*-3].
Software: RHEL 6.5; OFED 3.5-2; Intel® C/C++ Compiler XE 15.0.3; Intel® MPI Library 5.1; Intel® MPI Benchmarks 4.1.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. * Other brands and names are the property of their respective owners. Benchmark Source: Intel Corporation

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

MPI Program Structure



- SPMD: single program multiple data
- All processes are clones of each other. (Same executable)
- The first process that starts (usually designated proc 0) can be assigned the tasks of I/O, distributing data among the other procs etc.
- Everybody usually does work but not required
- An SPMD app. can function as a Manager/Worker code

Example: Hello World

```
/*
 "Hello World" MPI Test Program
*/
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[])
{
  char idstr[32];
  char buff[BUFSIZE];
  int numprocs;
  int myid;
  int i;
  MPI_Status stat;
  /* MPI programs start with MPI_Init; all 'N' processes exist thereafter */
  MPI_Init(&argc,&argv);
  /* find out how big the SPMD world is */
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  /* and this processes' rank is */
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

Example

```c
/* At this point, all programs are running equivalently, the rank
   distinguishes the roles of the programs in the SPMD model, with
   rank 0 often used specially... */
if(myid == 0)
{
  printf("%d: We have %d processors\n", myid, numprocs);
  for(i=1;i<numprocs;i++)
  {
    sprintf(buff, "Hello %d! ", i);
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
  }
  for(i=1;i<numprocs;i++)
  {
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
    printf("%d: %s\n", myid, buff);
  }
}
```

Example

```
  else
  {
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strncat(buff, idstr, BUFSIZE-1);
    strncat(buff, "reporting for duty", BUFSIZE-1);
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
  }

  /* MPI programs end with MPI Finalize; this is a weak synchronization point */
  MPI_Finalize();
  return 0;
}
```

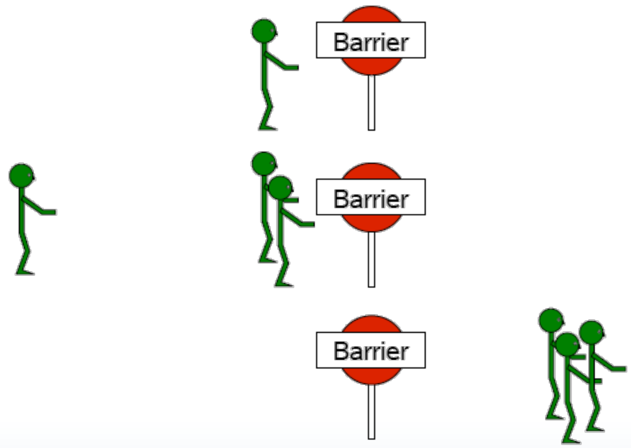
Result

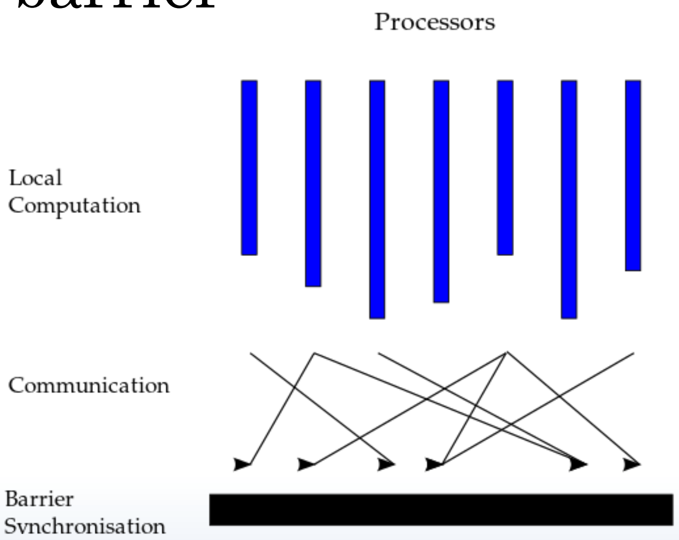
- When run with two processors this gives the following output.

```
0: We have 2 processors
0: Hello 1! Processor 1 reporting for duty
```

Key concepts

- BSP model: Bulk Synchronous Parallel (Parallelism)
  - Concurrent computation: a set of processor-memory pairs
  - Communication: a communications network that delivers messages in a point-to-point manner
  - Barrier synchronization: a mechanism for the efficient barrier synchronization for all or a subset of the processes.
- Supersteps: comp. → comm. → barrier



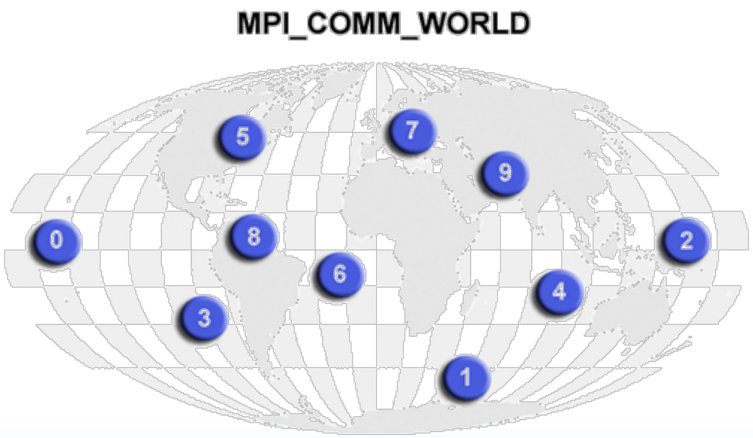


Further reading

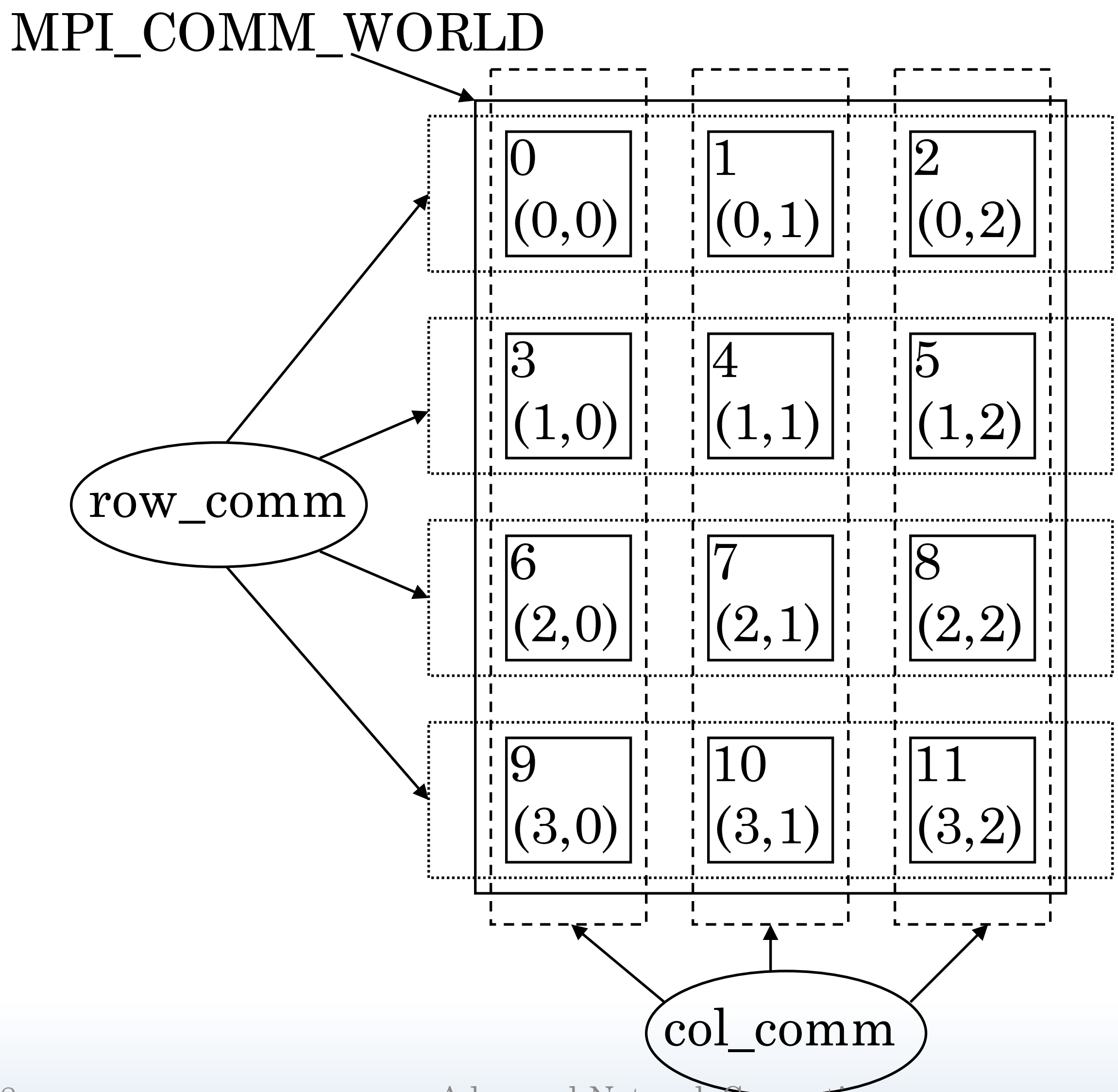
- LogP model
- PRAM model
- Efficient Implementation of **MPI_Barrier**

Communicator

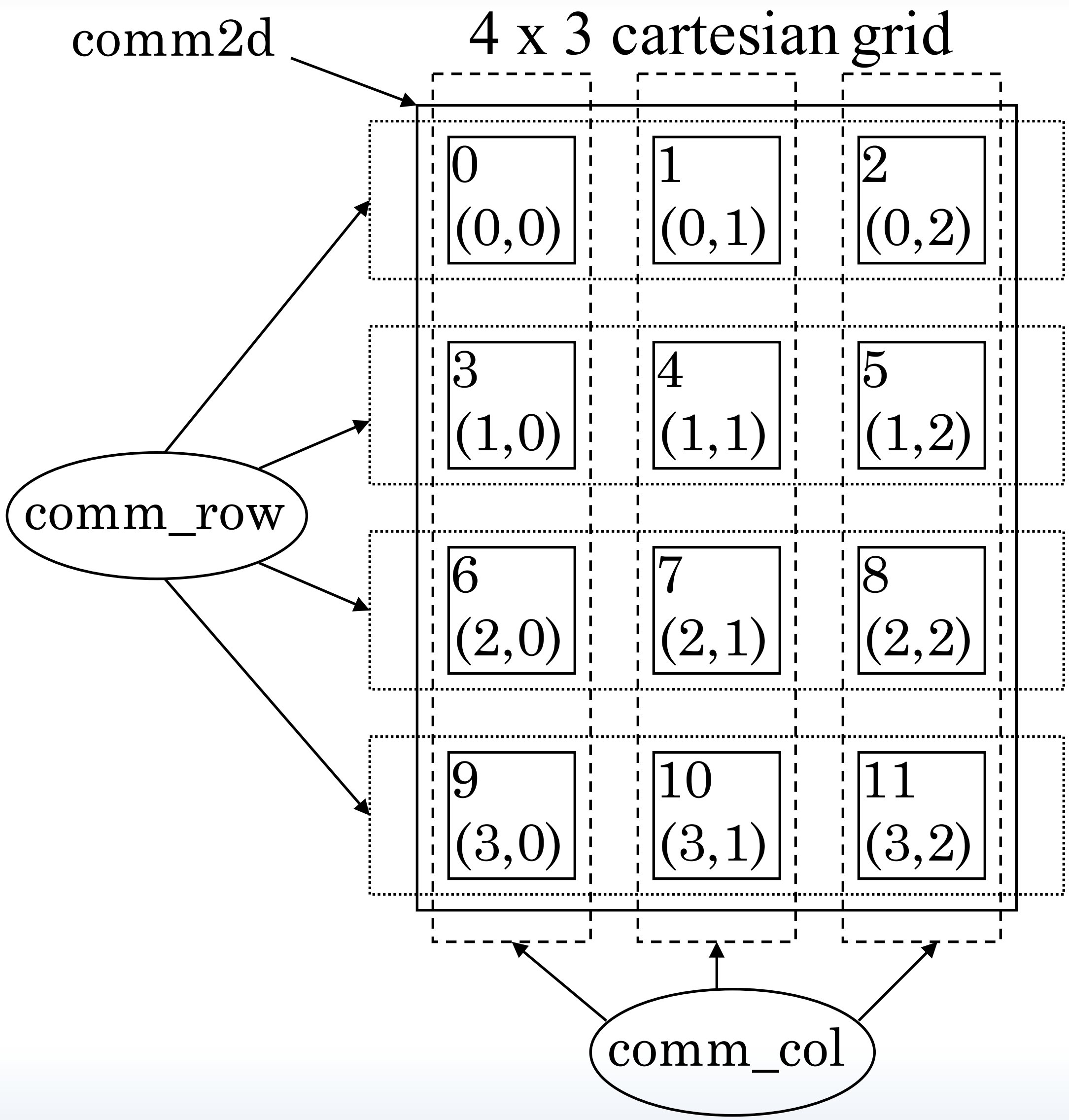
- Define which collection of processes may communicate with each other.
- Required by most routines
  - `MPI_Bsend(&buf, count, type, dest, tag, comm)`
- Predefined: **MPI_COMM_WORLD**
- User defined comm.: a group of processes.



New group and communicator



Cartesian Topologies



Rank

- `MPI_send(&buf, count, type, dest, tag, comm)`
- `MPI_recv(&buf, count, type, src,  tag, comm)`
- Unique integer id of an process, in a communicator.

- Q:
  - Can dests in different comms be the same?
  - Are the same dests in different comms the same process?

Tag

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **MPI_ANY_TAG** as the tag in a receive.
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes.

Identification

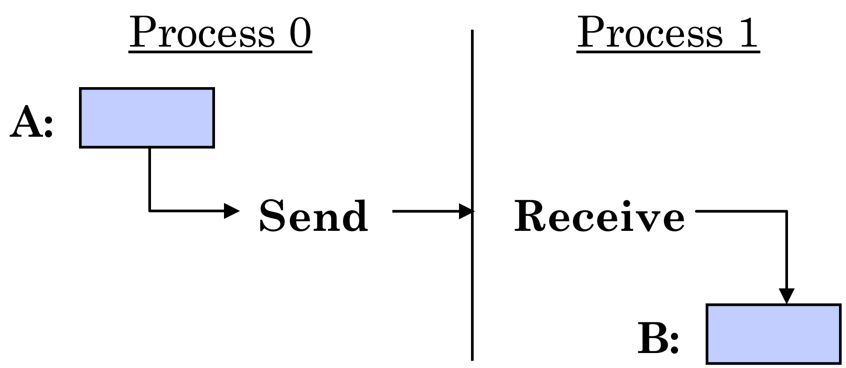
- Specify a process
  - <communicator, rank>
- Specify a message
  - <communicator, rank, tag>
- Wildcard
  - `MPI_ANY_SOURCE`
  - `MPI_ANY_TAG`
- Why no `MPI_ANY_DESTINATION`?

Communication

- Point-to-Point
  - One process to one process
  - Blocking and non-blocking
    - int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
    - int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
- Collective
  - communication among all processes in a process group (which can mean the entire process pool or a program-defined subset)
  - MPI_Bcast, MPI_Reduce, MPI_Alltoall

Sending and Receiving Messages

- Basic message passing process. Send data from one process to another



- Questions
  - To whom is data sent?
  - Where is the data?
  - What type of data is sent?
  - How much of data is sent?
  - How does the receiver identify it?

Message Organization in MPI

- Message is divided into data and envelope
- data
  - buffer
  - count
  - datatype
- envelope
  - process identifier (source/destination rank)
  - message tag
  - communicator
- Follows standard arg list order for most functions ie
  - MPI_SEND(buf, count, datatype, destination, tag, communicator)

Traditional Buffer Specification

- Sending and receiving only a contiguous array of bytes
- Hides the real data structure from hardware which might be able to handle it directly
- Requires pre-packing dispersed data
  - Rows of a matrix stored column-wise
  - General collections of structures
- Prevents communications between machines with different representations (even lengths) for same data type, except if the user works this out
- Buffer in MPI documentation can refer to:
  - User defined variable, array, or structure (most common)
  - MPI system memory used to process data (hidden from user)

Generalizing the Buffer Description

- Specified in MPI by starting address, count, and datatype, where datatype is as follows:
  - Elementary (all C and Fortran datatypes)
  - Contiguous array of datatypes
  - Strided blocks of datatypes
  - Indexed array of blocks of datatypes
  - General structure
- Datatypes are constructed recursively
- Specifying application-oriented layout of data allows maximal use of special hardware
- Elimination of length in favor of count is clearer
  - Traditional: send 20 bytes
  - MPI: send 5 integers

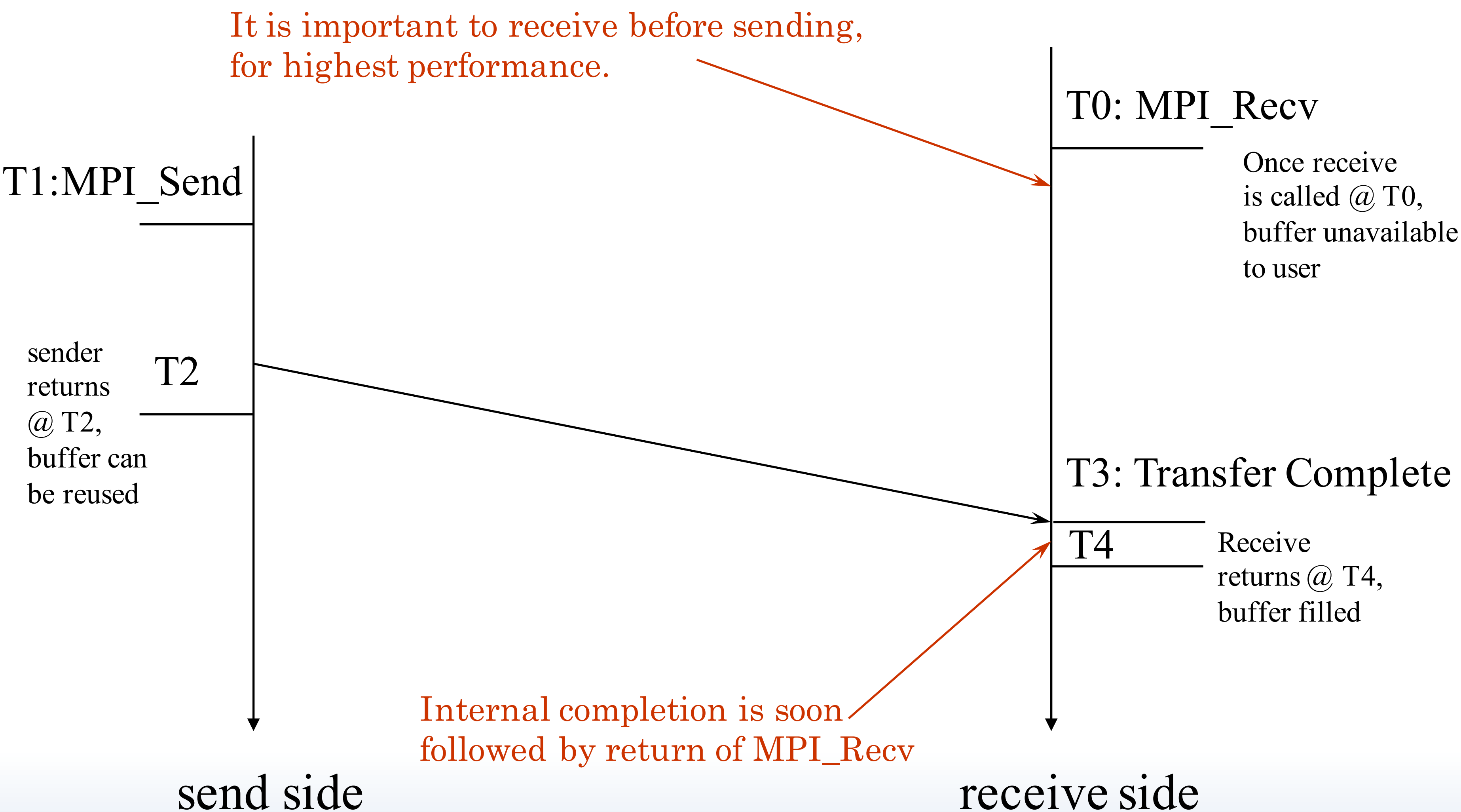
MPI C Datatypes

MPI datatype	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_BYTE	
MPI_PACKED	

Blocking Communication

- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked until the receive buffer is filled
- Completion of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks If (my_proc.eq.0) Then
- Call mpi_send(..)
- Call mpi_recv(…)
- Usually deadlocks--> Else
- Call mpi_send(…) <--- UNLESS you reverse send/recv
- Call mpi_recv(….)
- Endif

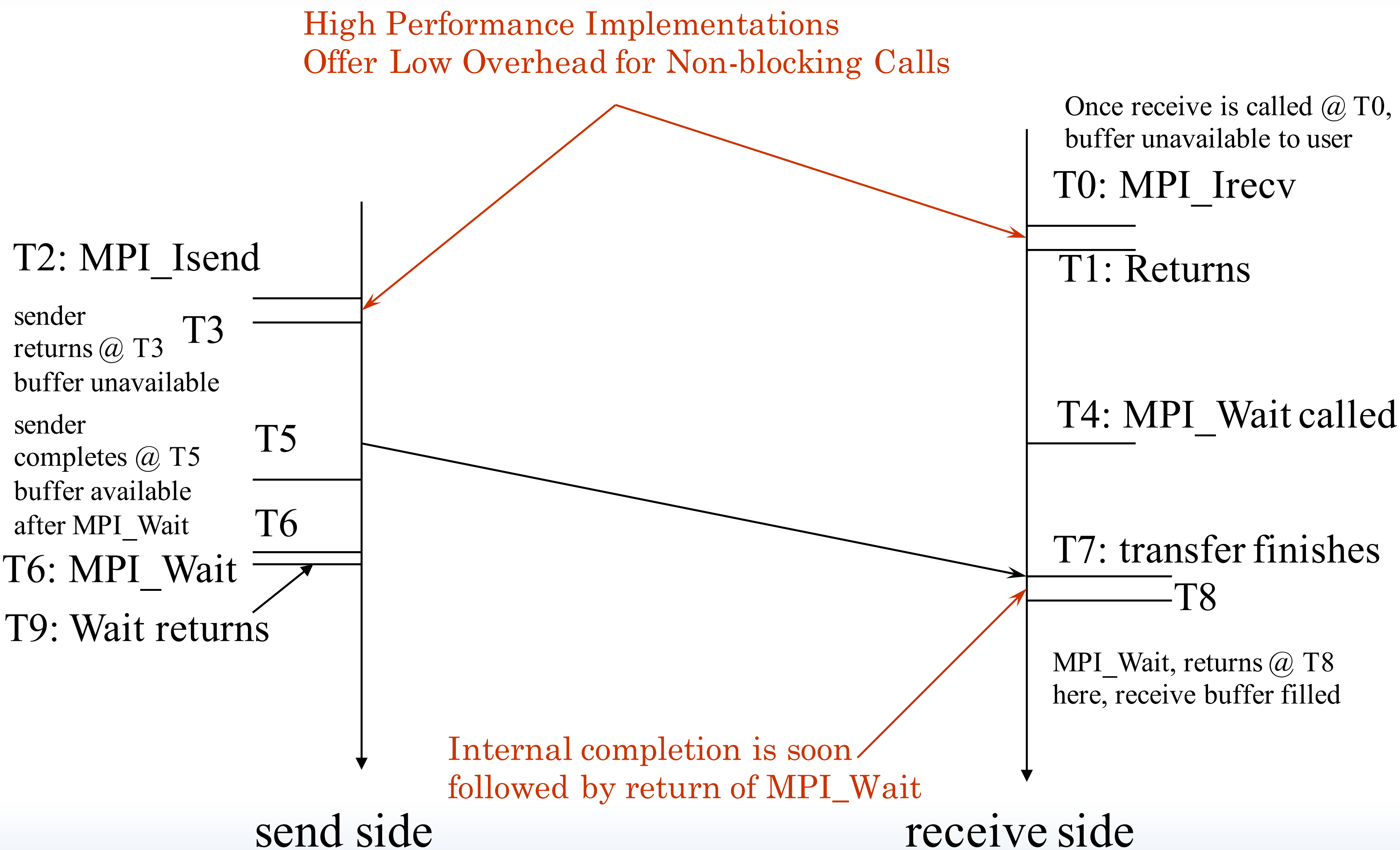
Blocking Send-Receive Diagram (Receive before Send)



Non-Blocking Communication

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

Non-Blocking Send-Receive Diagram



Multiple Completions

- It is often desirable to wait on multiple requests
- An example is a worker/manager program, where the manager waits for one or more workers to send it a message
  - `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

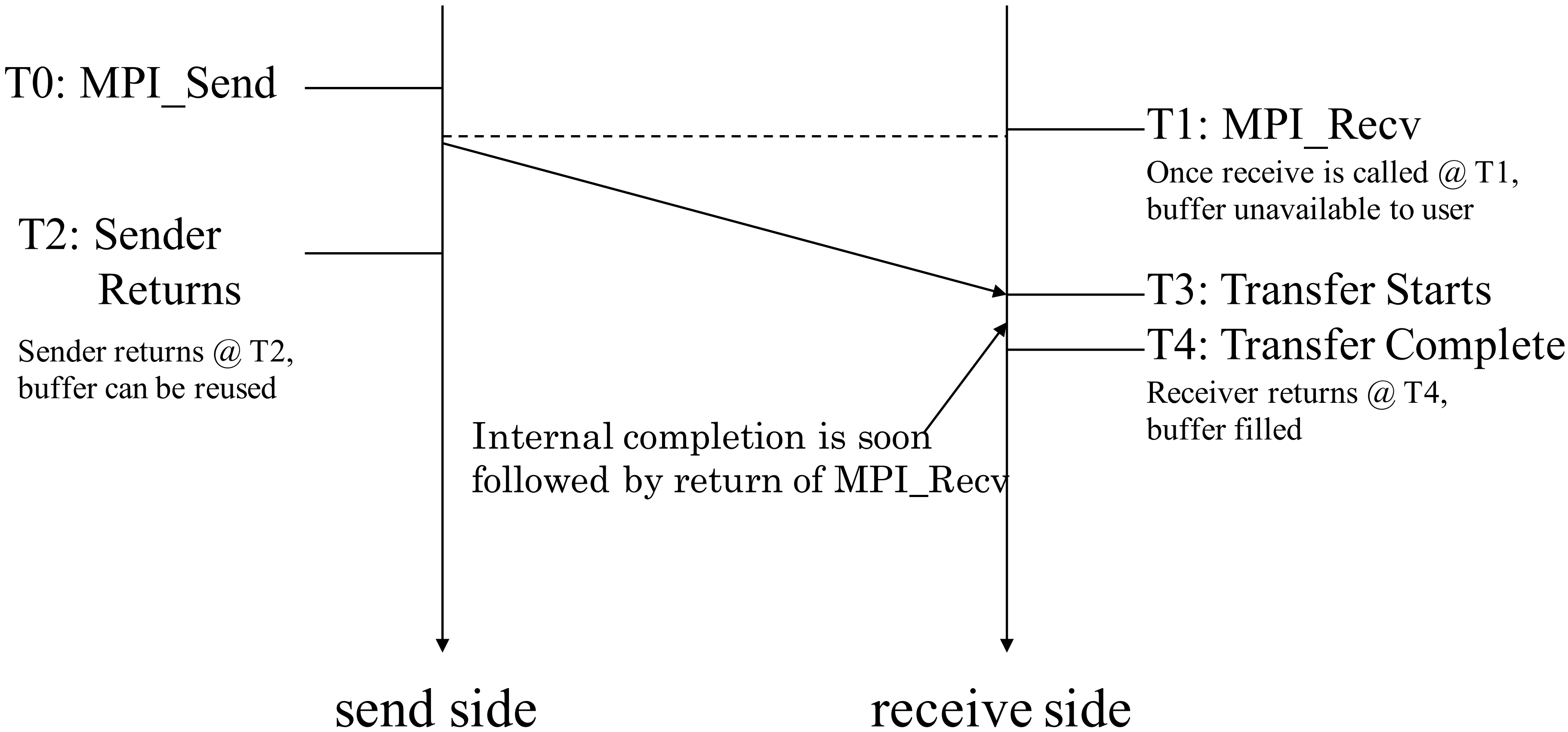
Probing the Network for Messages

- MPI_PROBE and MPI_IPROBE allow the user to check for incoming messages without actually receiving them
- MPI_IPROBE returns "flag == TRUE" if there is a matching message available.
- MPI_PROBE will not return until there is a matching receive available
- MPI_IPROBE (source, tag, communicator, flag, status)
  MPI_PROBE (source, tag, communicator, status)

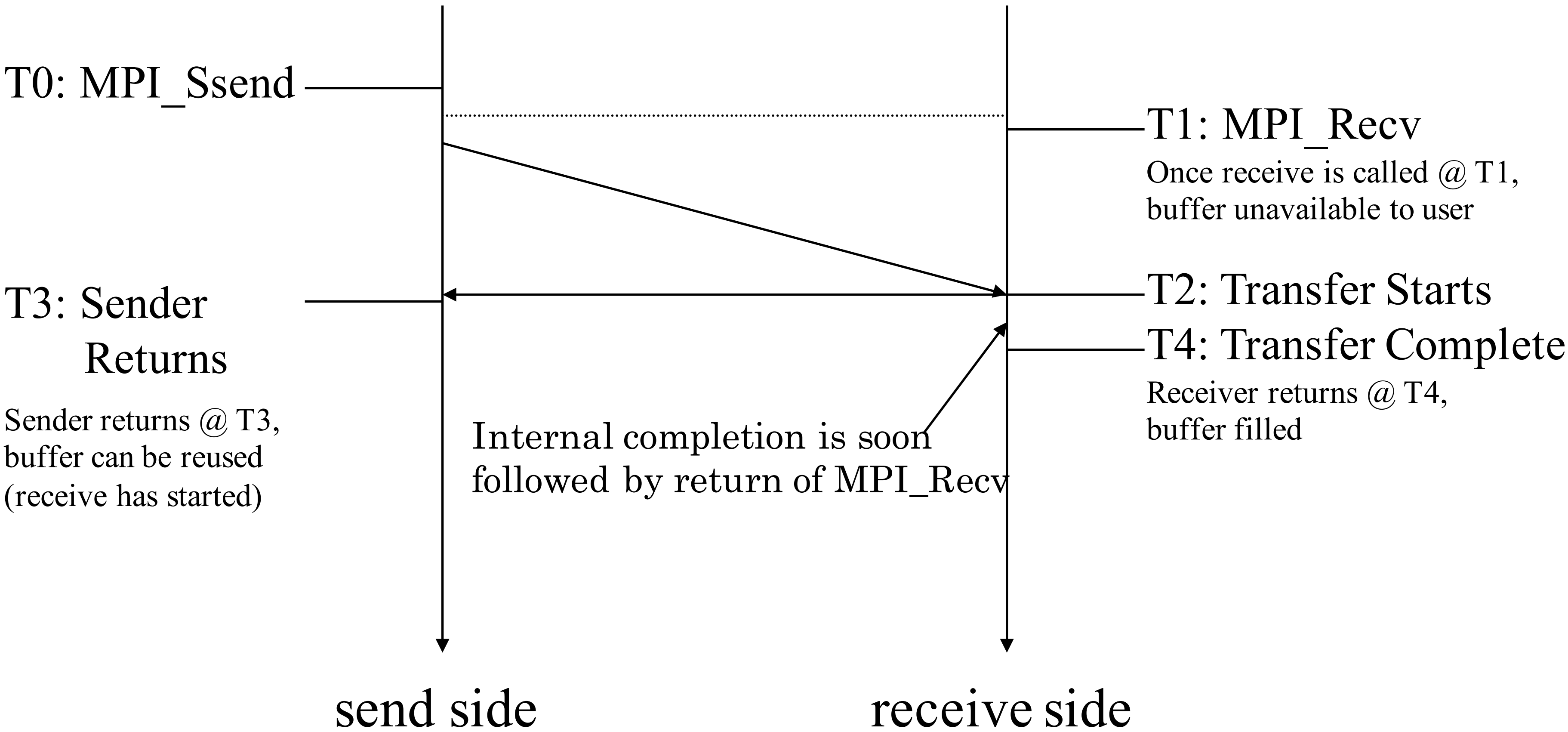
Send Modes

- Standard mode (**`MPI_Send, MPI_Isend`**)
  - The standard **`MPI_Send`**, the send will not complete until the send buffer is empty
- Synchronous mode (**`MPI_Ssend, MPI_Issend`**)
  - The send does not complete until after a matching receive has been posted
- Buffered mode (**`MPI_Bsend, MPI_Ibsend`**)
  - User supplied buffer space is used for system buffering
  - The send will complete as soon as the send buffer is copied to the system buffer
- Ready mode (**`MPI_Rsend, MPI_Irsend`**)
  - The send will send eagerly under the assumption that a matching receive has already been posted (an erroneous program otherwise)

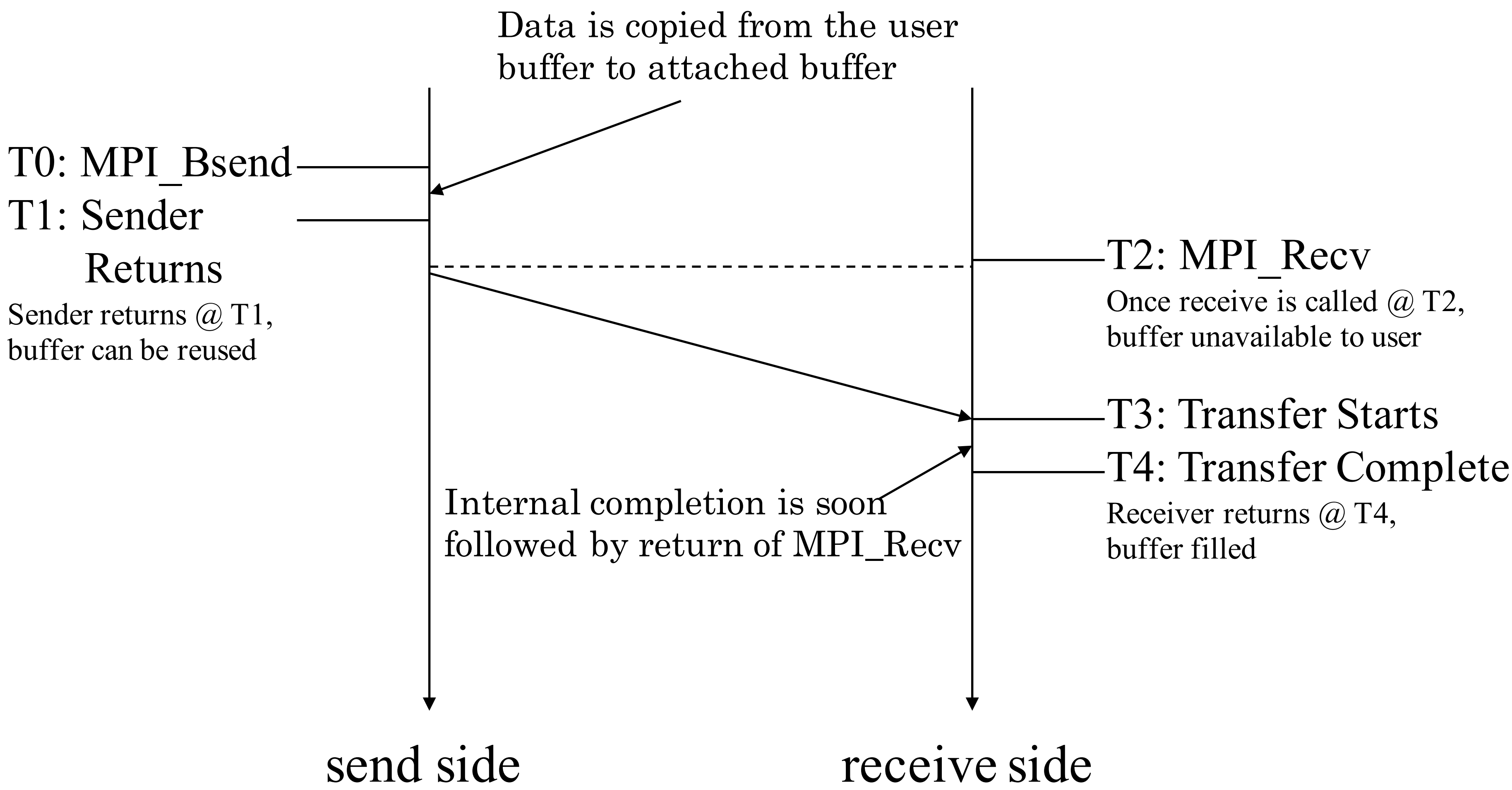
Standard Send-Receive Diagram



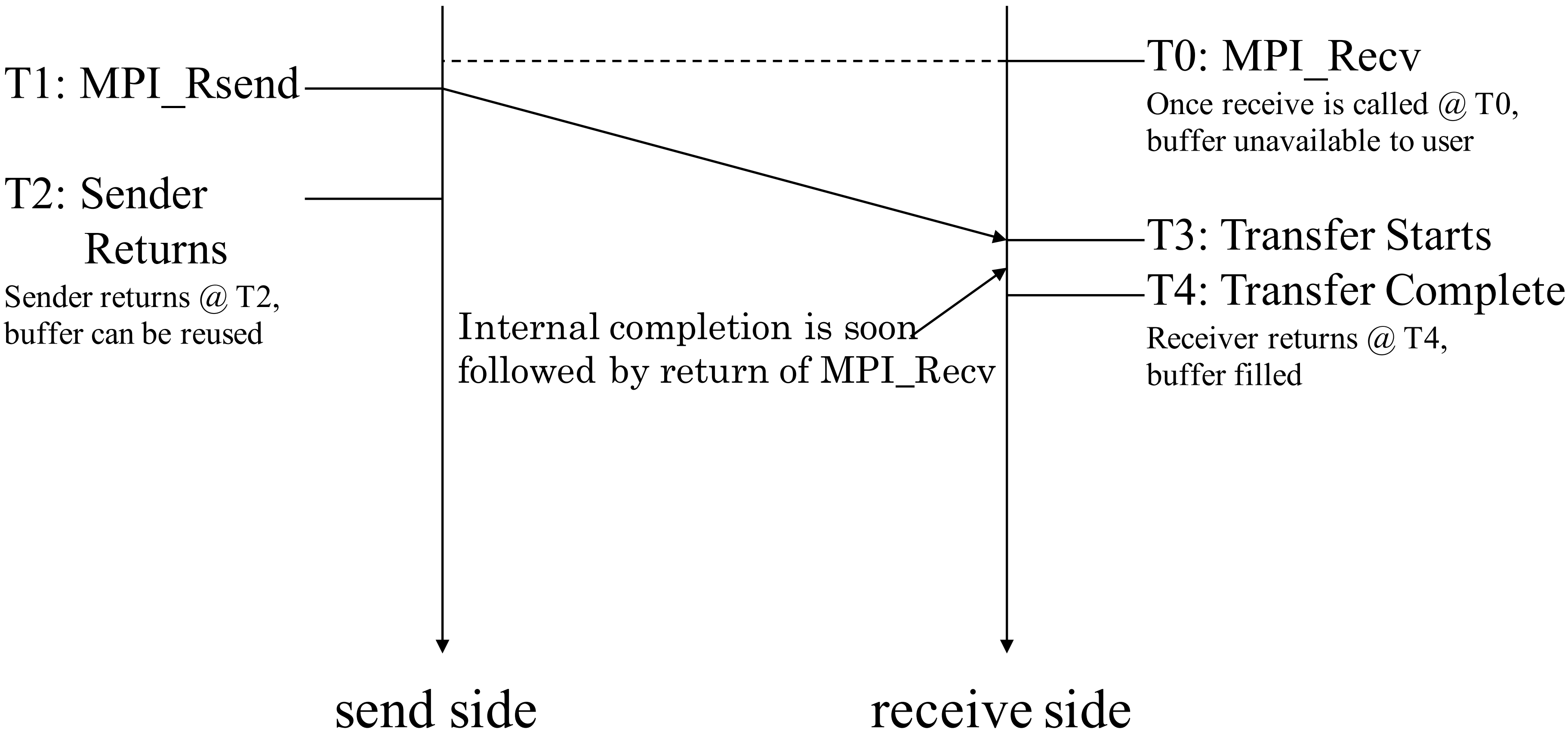
Synchronous Send-Receive Diagram



Buffered Send-Receive Diagram



Ready Send-Receive Diagram

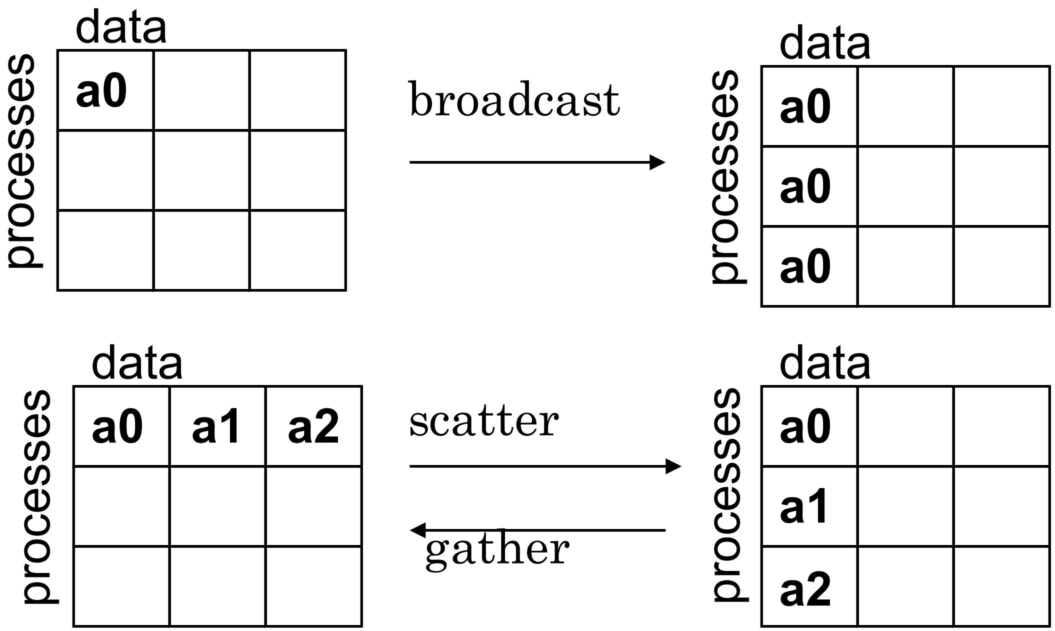


Collective Communications

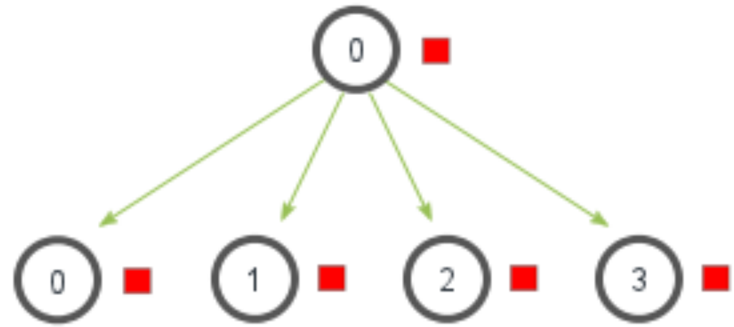
- Communication is coordinated among a group of processes, as specified by communicator, not on all processes
- All collective operations are blocking and no message tags are used (in MPI-1)
- All processes in the communicator group must call the collective operation
- Three classes of collective operations
  - Data movement
  - Collective computation
  - Synchronization

MPI Basic Collective Operations

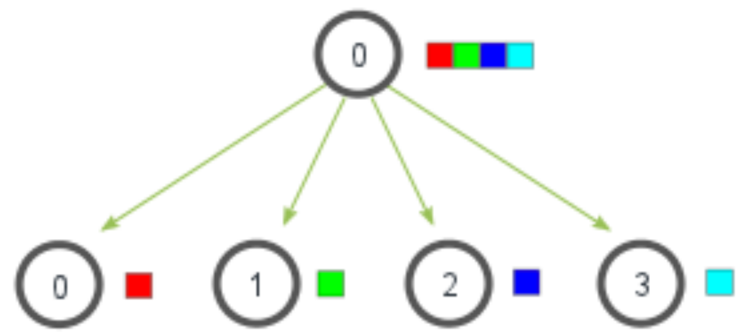
- Two simple collective operations
  - MPI_BCAST(start, count, datatype, root, comm)
  - MPI_REDUCE(start, result, count, datatype, operation, root, comm)
- The routine MPI_BCAST sends data from one process to all others
- The routine MPI_REDUCE combines data from all processes, using a specified operation, and returns the result to a single process



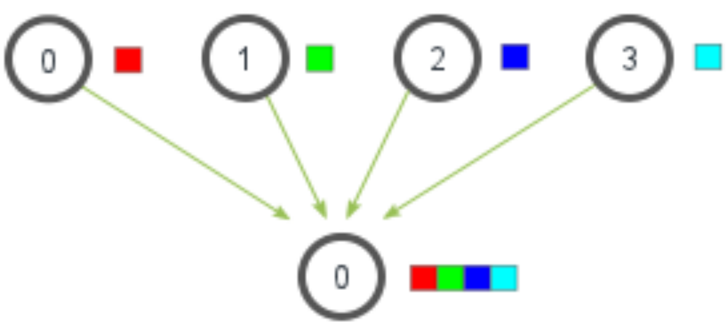
MPI_Bcast



MPI_Scatter



MPI_Gather



data (processes × data)

a0		
b0		
c0		

allgather →

data (processes × data)

a0	b0	c0
a0	b0	c0
a0	b0	c0

data (processes × data)

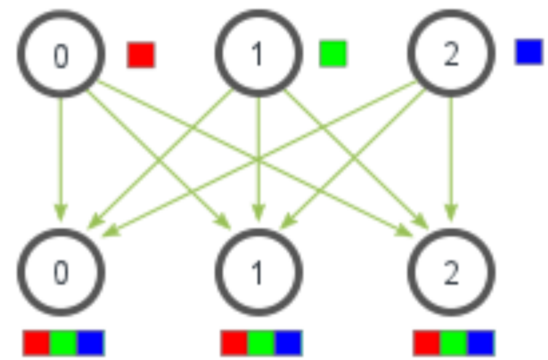
a0	a1	a2
b0	b1	b2
c0	c1	c2

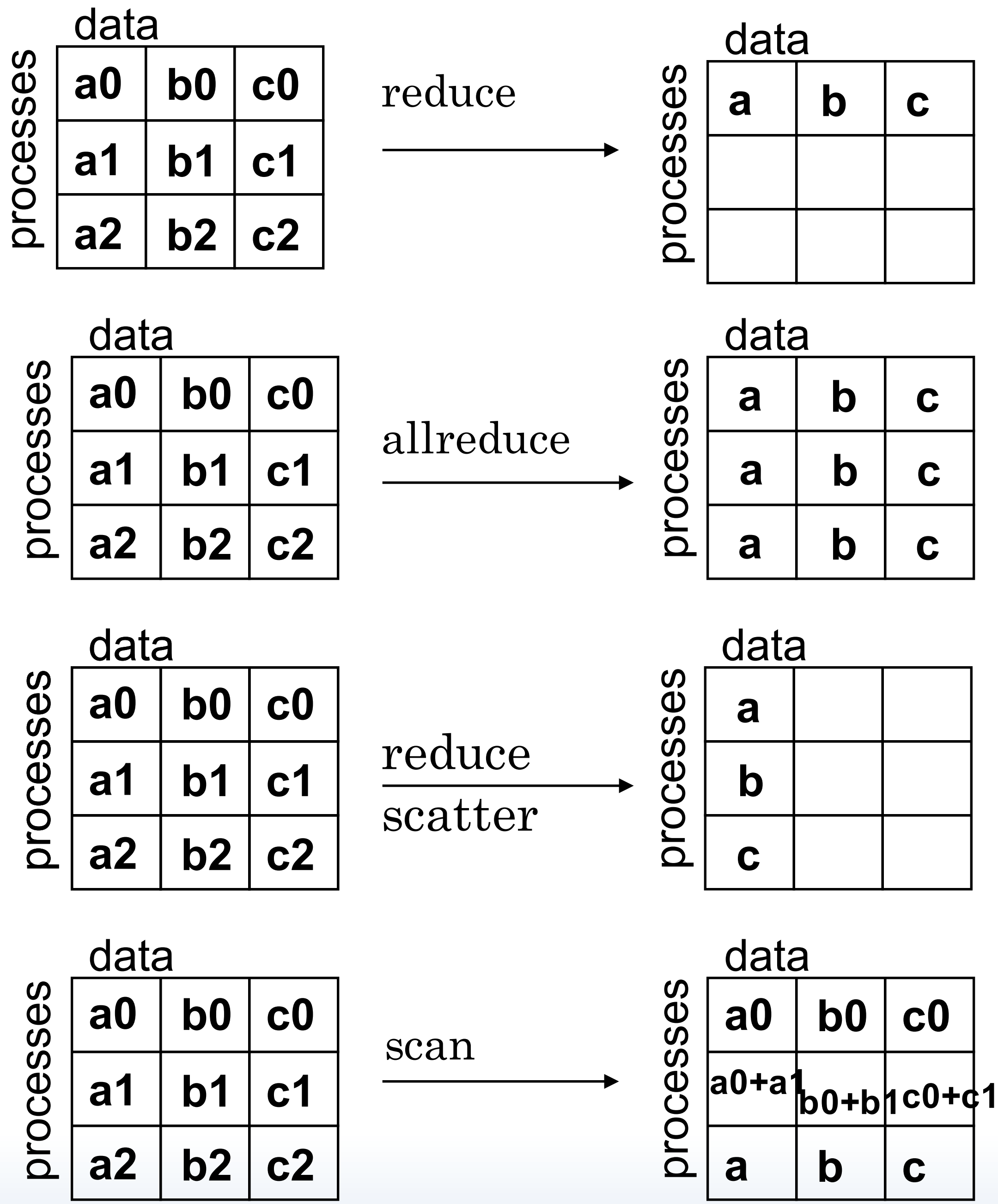
alltoall →

data (processes × data)

a0	b0	c0
a1	b1	c1
a2	b2	c2

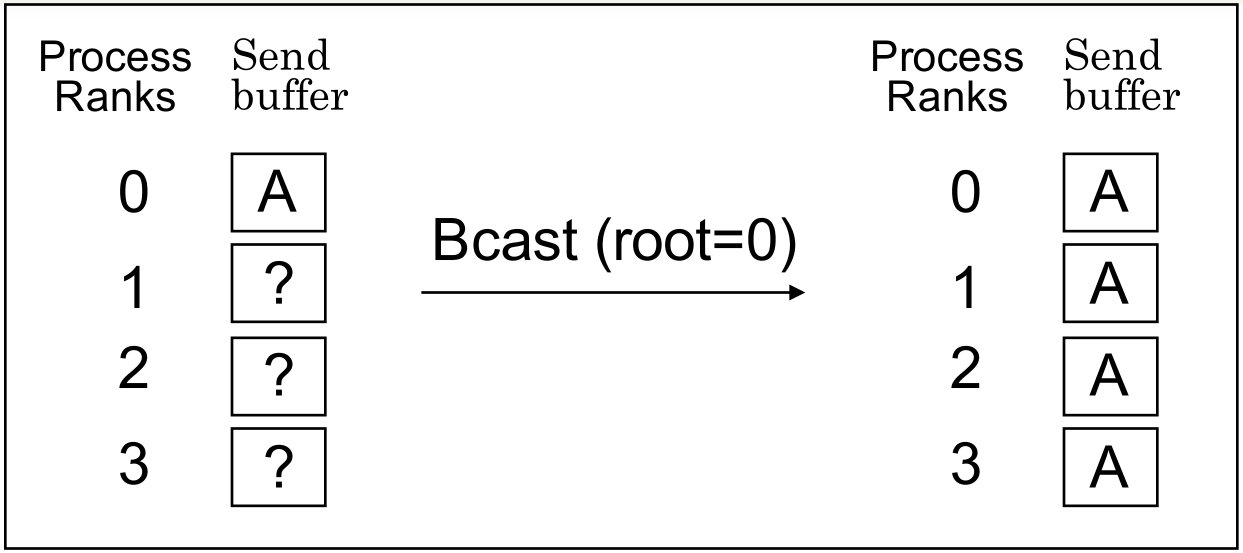
MPI_Allgather

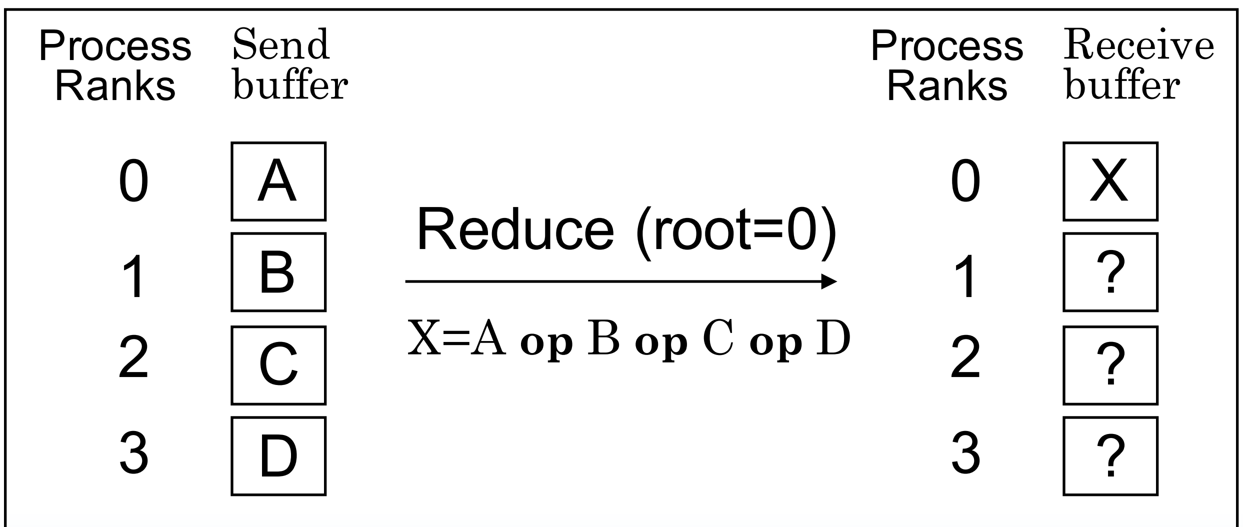




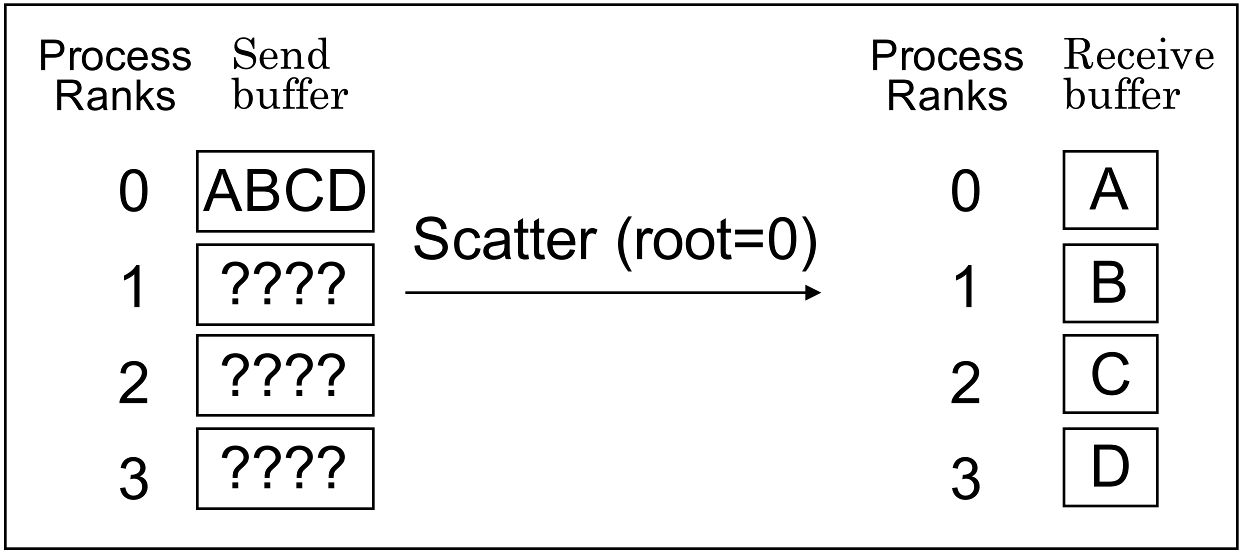
Note:
a=sum($\mathbf{a}_i$,all i)
b=sum($\mathbf{b}_i$, all i
c=sum($\mathbf{c}_i$, all i)
(here i=0..2)

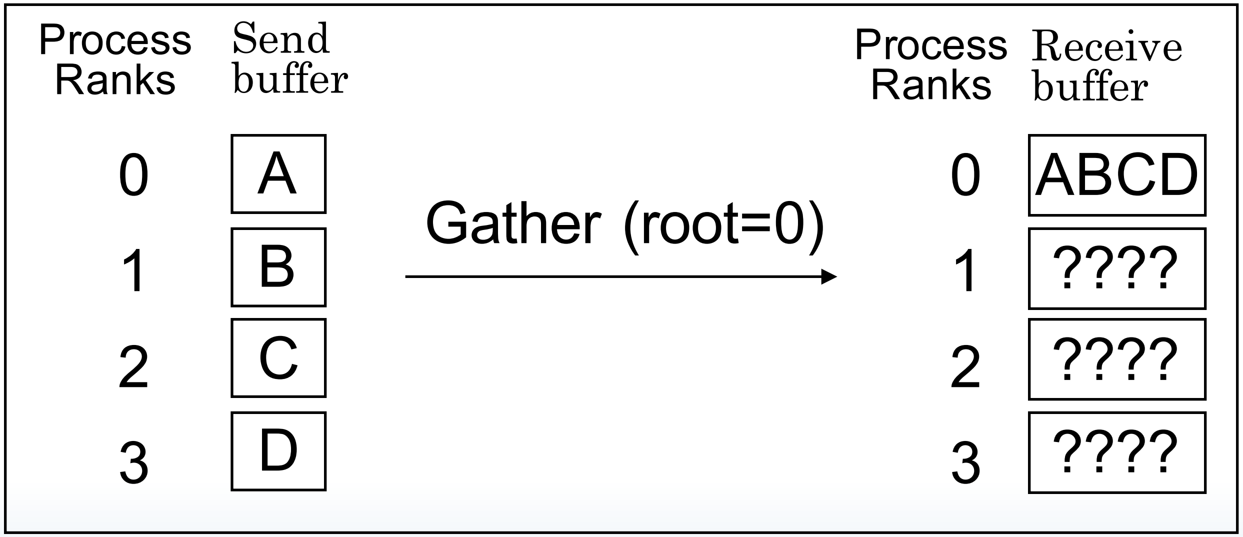
Broadcast and Reduce



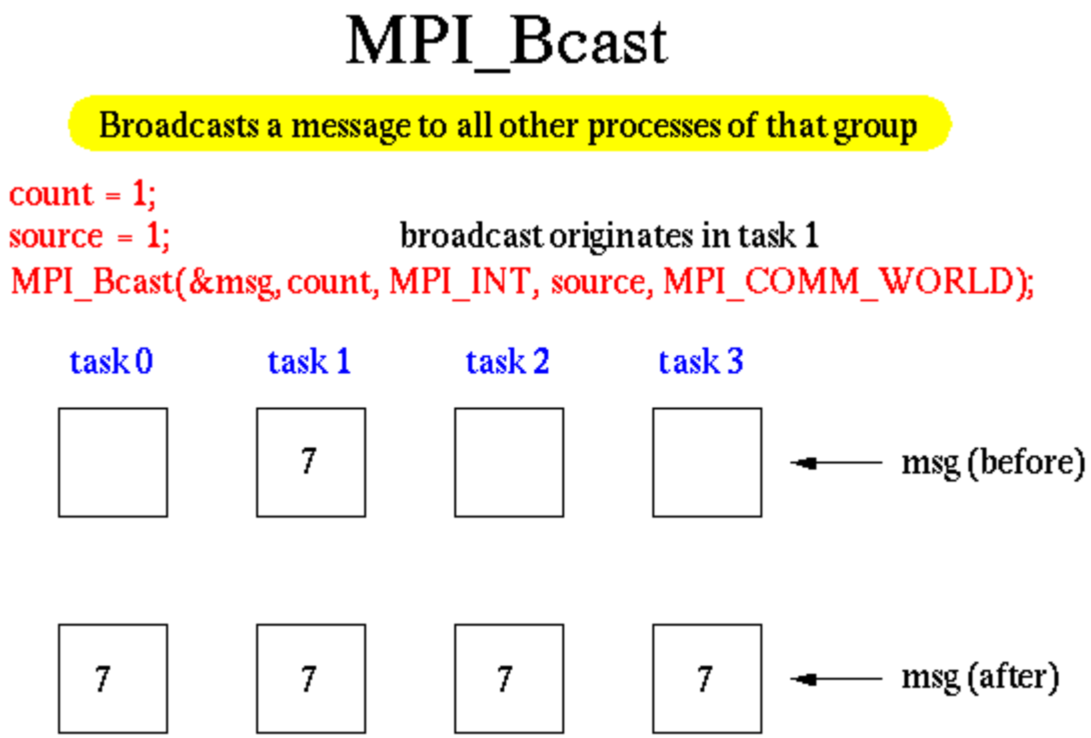


Scatter and Gather

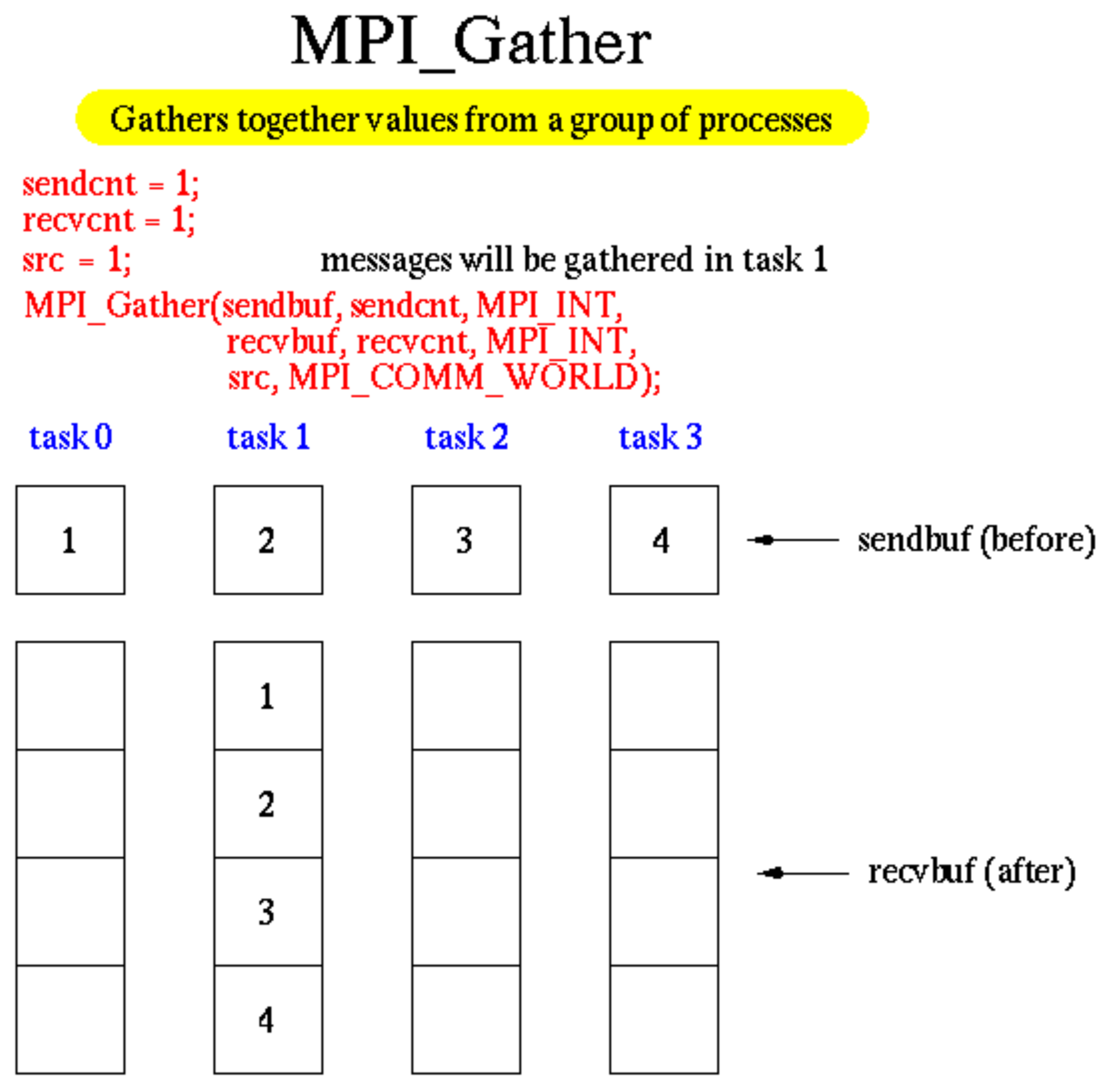




MPI_Bcast


MPI_Bcast
Broadcasts a message to all other processes of that group
count = 1;
source = 1;
broadcast originates in task 1
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
7
msg (before)
7
7
7
7
msg (after)

MPI_Gather

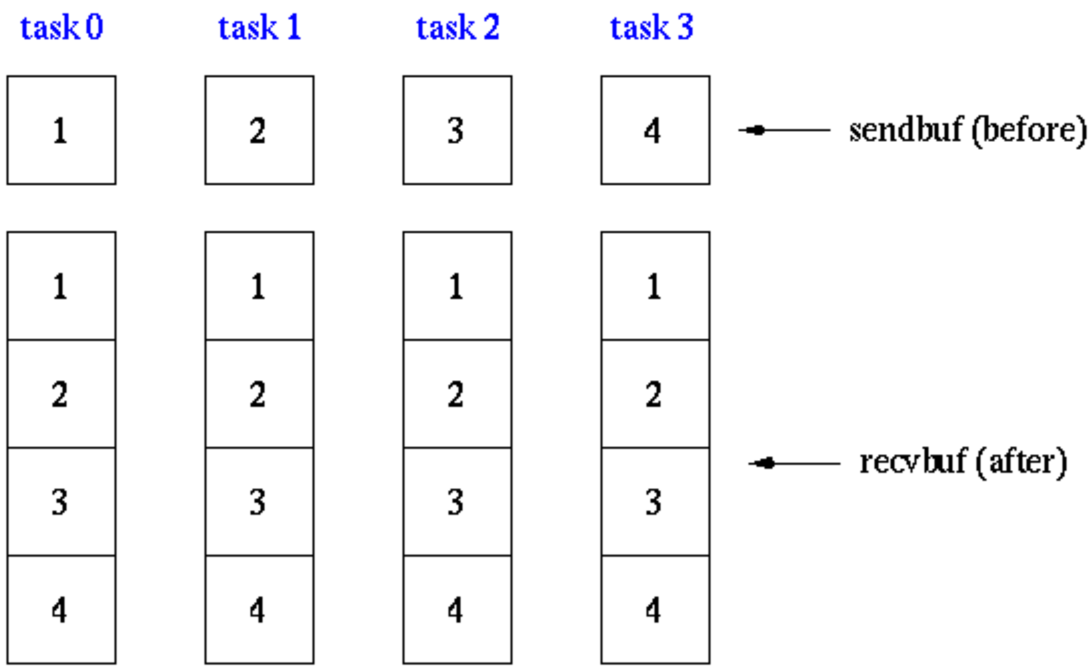

MPI_Gather
Gathers together values from a group of processes
sendcnt = 1;
recvcnt = 1;
src = 1;
messages will be gathered in task 1
MPI_Gather(sendbuf, sendcnt, MPI_INT,
recvbuf, recvcnt, MPI_INT,
src, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
1
2
3
4
recvbuf (after)

MPI_Allgather

MPI_Allgather

Gathers together values from a group of processes and distributes to all

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT,
              recvbuf, recvcnt, MPI_INT,
              MPI_COMM_WORLD);
```



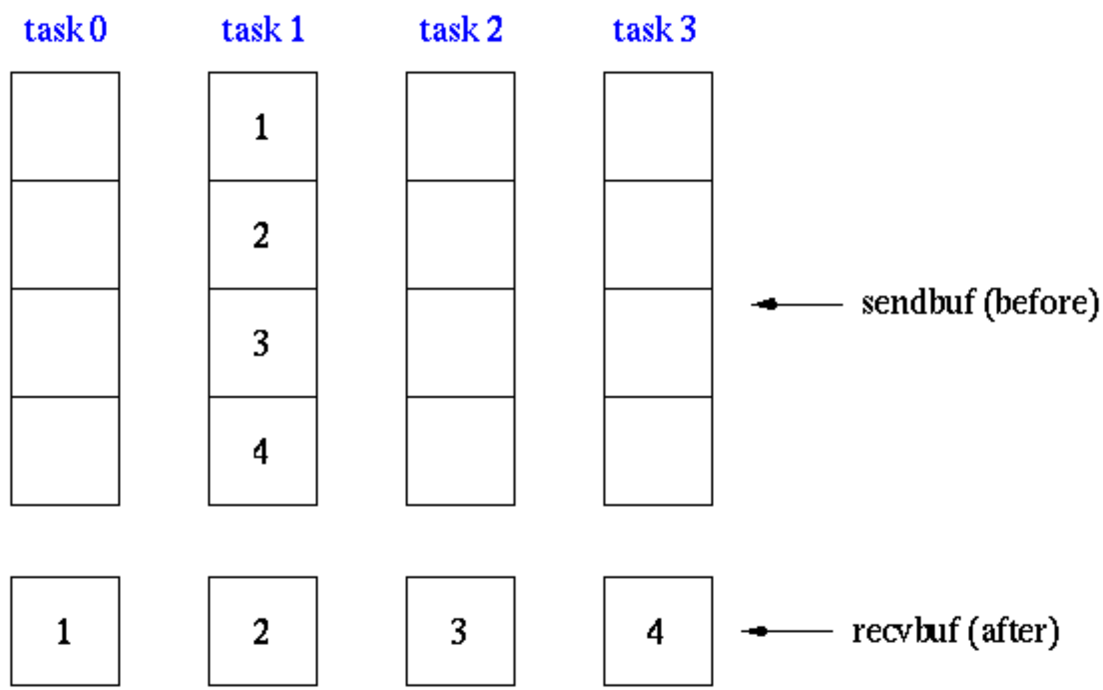
MPI_Scatter

MPI_Scatter

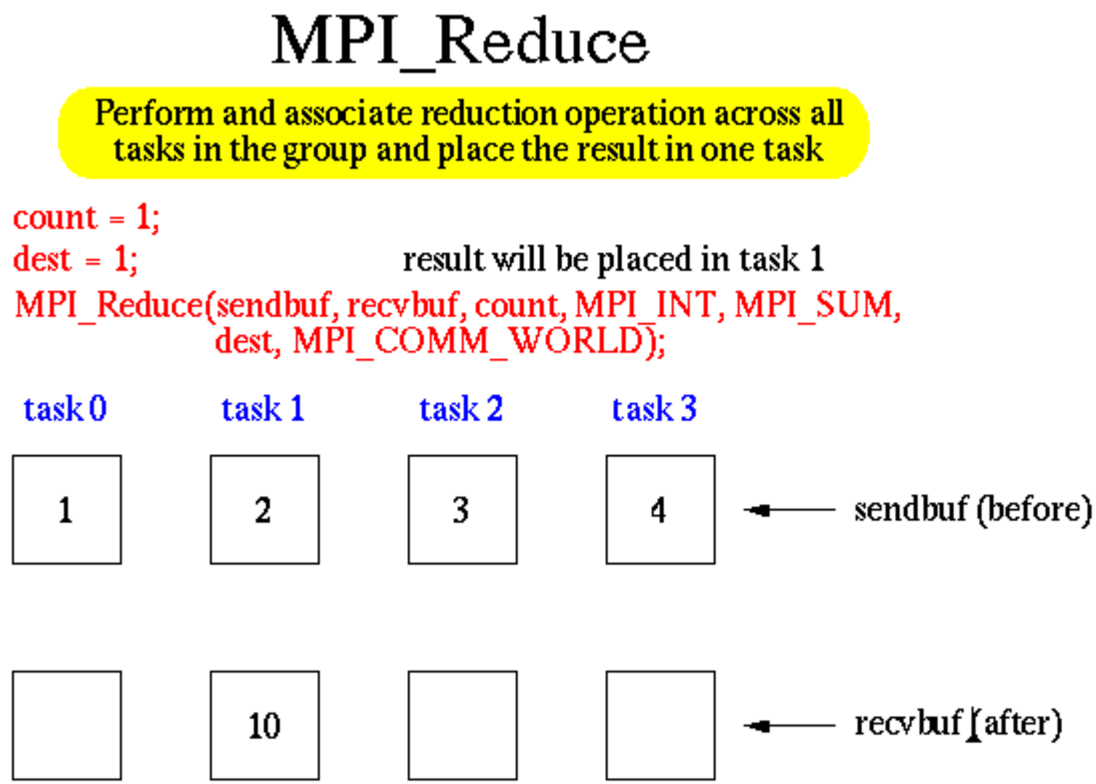
Sends data from one task to all other tasks in a group

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Scatter(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT,
            src, MPI_COMM_WORLD);
```

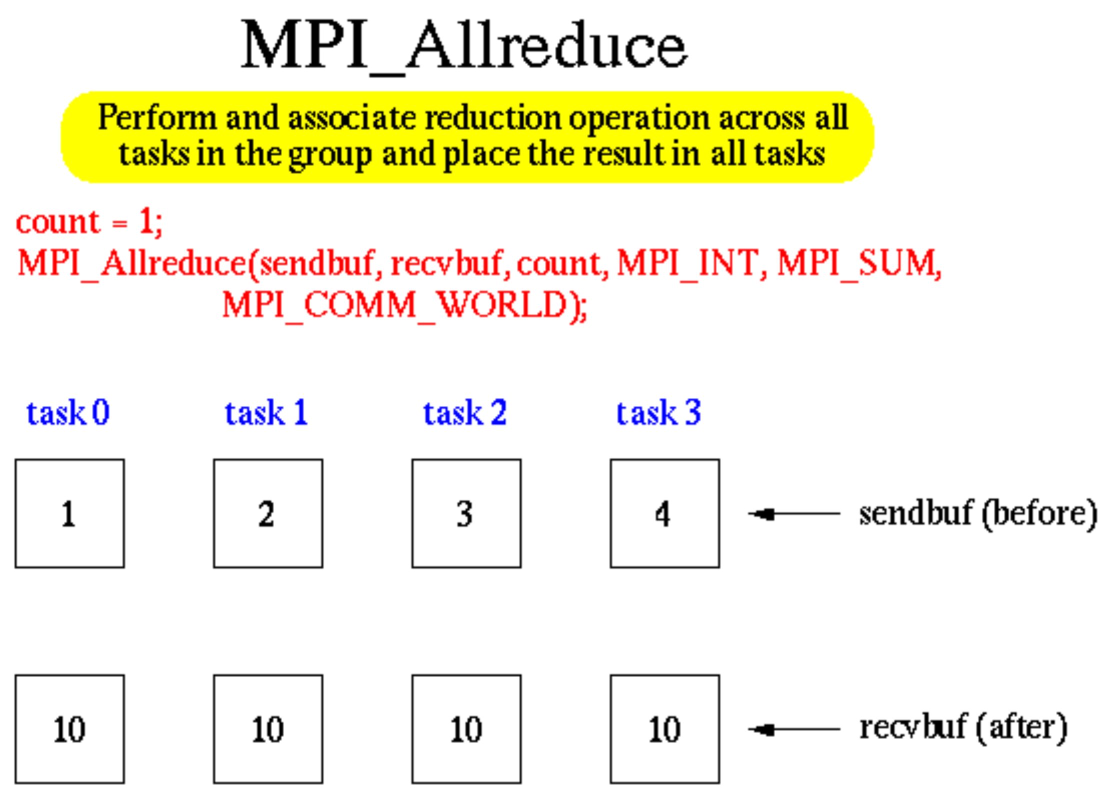
task 1 contains the message to be scattered



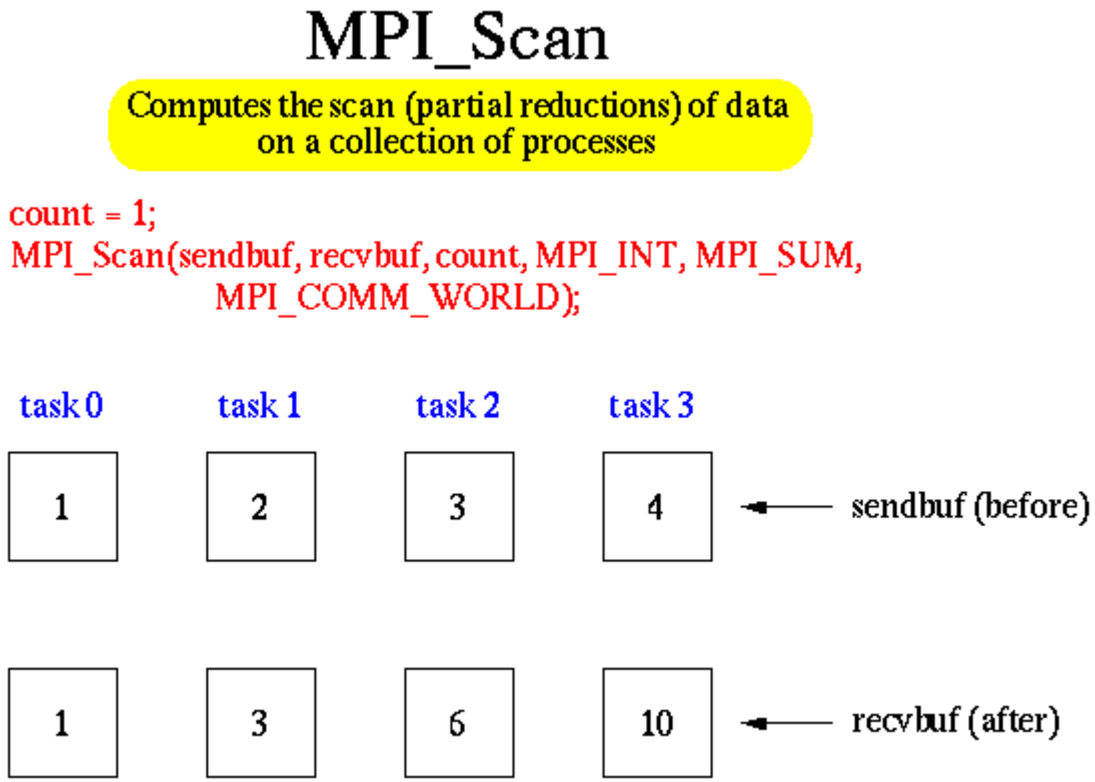
MPI_Reduce


MPI_Reduce
Perform and associate reduction operation across all tasks in the group and place the result in one task
count = 1;
dest = 1;
result will be placed in task 1
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, dest, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
10
recvbuf (after)

MPI_Allreduce



MPI_Scan

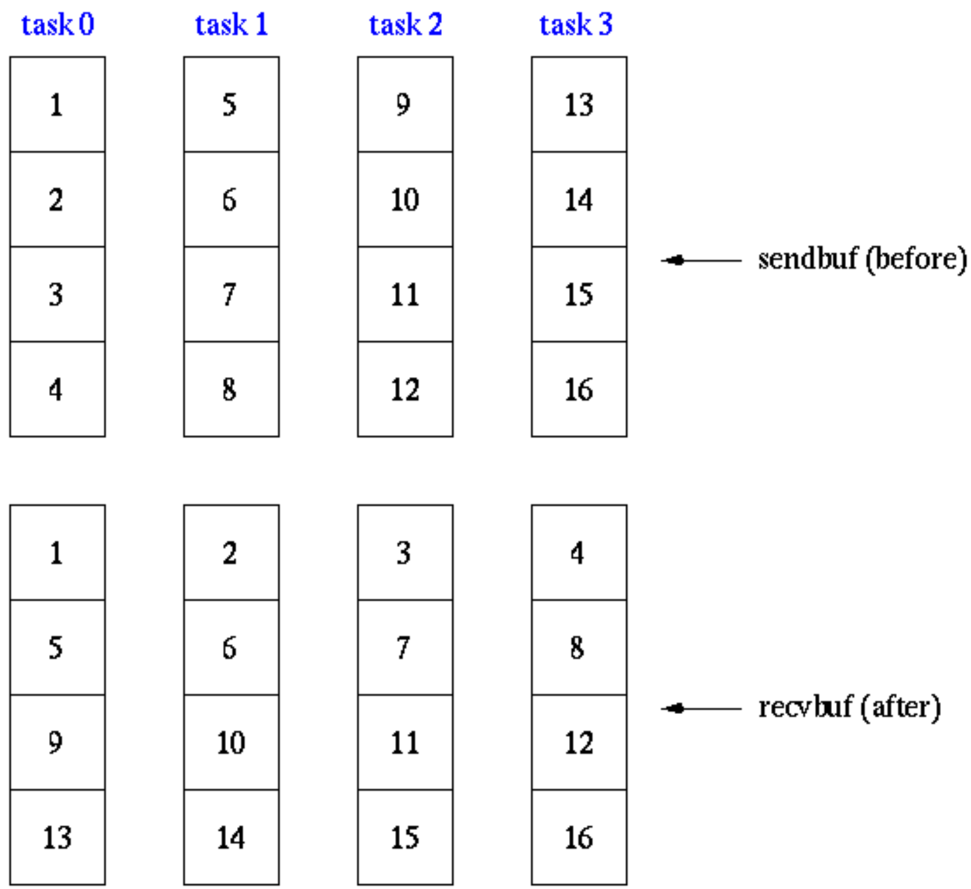

MPI_Scan
Computes the scan (partial reductions) of data on a collection of processes
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
1
3
6
10
recvbuf (after)

MPI_A

MPI_Alltoall

Sends data from all to all processes. Each process performs a scatter operation.

```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
             recvbuf, recvcnt, MPI_INT,
             MPI_COMM_WORLD);
```



MPI Collective Routines

- Several routines:

MPI_ALLGATHER	**MPI_ALLGATHERV**	**MPI_BCAST**
MPI_ALLTOALL	**MPI_ALLTOALLV**	**MPI_REDUCE**
MPI_GATHER	**MPI_GATHERV**	**MPI_SCATTER**
MPI_REDUCE_SCATTER	**MPI_SCAN**	
MPI_SCATTERV	**MPI_ALLREDUCE**	

- **All** versions deliver results to all participating processes
- **"V"** versions allow the chunks to have different sizes
- **MPI_ALLREDUCE, MPI_REDUCE, MPI_REDUCE_SCATTER,** and **MPI_SCAN** take both built-in and user-defined combination functions

Built-In Collective Computation Operations

MPI Name	Operation
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 (xor)
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise xor
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

Synchronization

- MPI_BARRIER (comm)
- Function blocks until all processes in "comm" call it
- Often not needed at all in message-passing codes
- When needed, mostly for highly asynchronous programs or ones with speculative execution
- Try to limit the use of explicit barriers. They can severely impact performance if overused.
  - The T3E is an exception: It has hardware support for barriers. Most machines don't

Question

- What’s the benefits of non-blocking comm. over blocking routines?
  - For the sake of performance, a huge topic: comp./comm. overlap
- Can collective communication routines be non-blocking? and why?
  - Complexity.
  - MPI-2 tries to address them.

Derived datatypes

- Recall how to transfer a buffer from one process to another one?
  - MPI_Send(&buffer, count, type, …)
- What if to transfer separated data items? E.g., odd columns?
  - Do it in a loop?
  - Or copy the items to a continuous buffer?
- Derived datatype

Column 1	Column 2	Column 3	Column 4	Column 5

Datatypes

- MPI datatypes have two main purposes:
  - Heterogeneity --- parallel programs between different processors
  - Noncontiguous data --- structures, vectors with non-unit stride, etc.
- Basic/primitive datatypes, corresponding to the underlying language, are predefined
- Users can construct new datatypes at run time → derived datatypes
- Datatypes can be constructed recursively
- Avoids explicit packing/unpacking of data by user
- A derived datatype can be used in any communication operation instead of primitive datatype
  - MPI_SEND (buf, 1, mytype, …..)
  - MPI_RECV (buf, 1, mytype, …..)

Datatypes in MPI

- Elementary: Language-defined types
  - MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, etc.
- Vector: Separated by constant "stride"
  - MPI_TYPE_VECTOR
- Contiguous: Vector with stride of one
  - MPI_TYPE_CONTIGUOUS
- Hvector: Vector, with stride in bytes
  - MPI_TYPE_HVECTOR
- Indexed: Array of indices (for scatter/gather)
  - MPI_TYPE_INDEXED
- Hindexed: Indexed, with indices in bytes
  - MPI_TYPE_HINDEXED
- Struct: General mixed types (for C structs etc.)
  - MPI_TYPE_STRUCT

Example: naïve solution

- `int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *`**`recvbuf`**`, int recvcount, MPI_Datatype recvtype, int `**`root`**`, MPI_Comm comm)`
- **`recvbuf`**: address of receive buffer, significant only at **`root`**

```
int array[100];
int root, total_p, *receive_array;

MPI_Comm_size(comm, &total_p);
receive_array=malloc(total_p*100*sizeof(*receive_array));
MPI_Gather(array, 100, MPI_INT, receive_array, 100, MPI_INT,
root, comm);
```

Example: derived datatype

```
MPI_Datatype newtype;
MPI_Type_contiguous(100, MPI_INT, &newtype);
MPI_Type_commit(&newtype);

MPI_Gather(array, 1, newtype, receive_array, 1, newtype, root,
comm);
```

```
int array[100];
int root, total_p, *receive_array;

MPI_Comm_size(comm, &total_p);
receive_array=malloc(total_p*100*sizeof(*receive_array));
MPI_Gather(array, 100, MPI_INT, receive_array, 100, MPI_INT,
root, comm);
```

```
count = 4;   blocklength = 1;   stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,
&columntype);
```

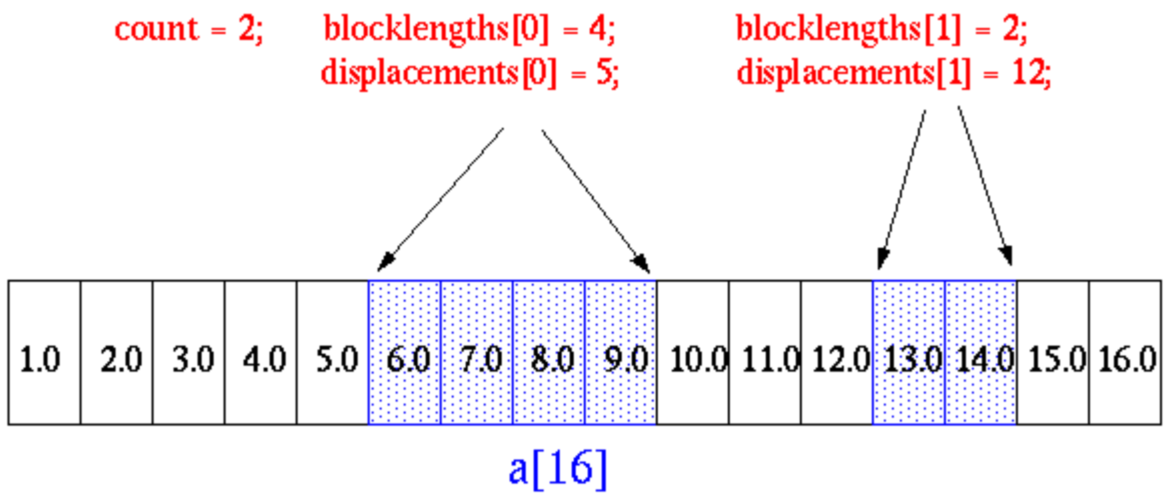
1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);
```

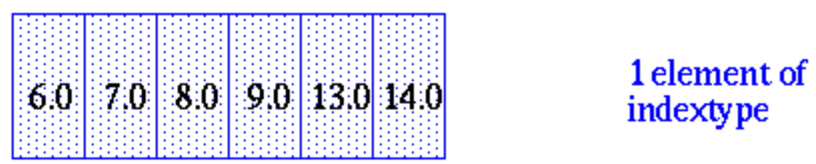
2.0	6.0	10.0	14.0

1 element of columntype




```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



Pack and Unpack

- If you neither like derived datatype, nor like manage new buffer manually.

- We explicitly pack non-contiguous data into a contiguous buffer for transmission, then unpack it at the other end.
- When sending/receiving packed messages, must use MPI_PACKED datatype in send/receive calls.

MPI: Packing

int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)

- will pack the information specified by *inbuf* and *incount* into the buffer space provided by *outbuf* and *outsize*
- the current packing call will pack this data starting at offset *position* in the *outbuf*

MPI: Unpacking

int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

- unpacks message to *outbuf*.
- updates the *position* argument so it can be used in a subsequent call to MPI_Unpack.

Example

```
int i;
char c[100];
char buffer[110];
int position=0;

//pack
MPI_Pack(&i,1,MPI_INT,buffer,110,&position,MPI_COMM_WORLD);
MPI_Pack(c,100,MPI_CHAR,buffer,110,&position,MPI_COMM_WORLD);
//send
MPI_Send(buffer,position,MPI_PACKED,1,0,MPI_COMM_WORLD);
...
```

Example cnt'd

```
...
//correspoding receive
//position=0
MPI_Recv(buffer,110,MPI_PACKED,1,0,MPI_COMM_WORLD,&status);
//and unpack
MPI_Unpack(buffer,110,&position,&i,1,MPI_INT,MPI_COMM_WORLD);
MPI_Unpack(buffer,110,&position,c,100,MPI_CHAR,MPI_COMM_WORLD
  );
...
```

MPI: Other Pack/Unpack Calls

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)

- allows you to find out how much space (bytes) is required to pack a message.
- enables user to manage buffers for packing.

Derived Datatypes vs. Pack/Unpack

- Pack/Unpack is quicker/easier to program.
- Derived datatypes are more flexible in allowing complex derived datatypes to be defined.
- Pack/Unpack has higher overhead.
- Derived datatypes are better if the datatype is regularly reused.

Six Function MPI-1 Subset

- MPI is simple. These six functions allow you to write many programs:

- MPI_Init()
- MPI_Finalize()
- MPI_Comm_size()
- MPI_Comm_rank()
- MPI_Send()
- MPI_Recv()

Starting the MPI Environment

- MPI_INIT ( )

  Initializes MPI environment. This function must be called and must be the first MPI function called in a program (exception: MPI_INITIALIZED)

  Syntax

  - int MPI_Init (int *argc, char ***argv)

  - MPI_INIT (IERROR)
  - INTEGER IERROR

- NOTE: Both C and Fortran return error codes for all calls.

Exiting the MPI Environment

- MPI_FINALIZE ( )

  Cleans up all MPI state. Once this routine has been called, no MPI routine (even MPI_INIT) may be called

  - Syntax
    - int MPI_Finalize ( );
    - MPI_FINALIZE (IERROR)
    - INTEGER IERROR

  - MUST call MPI_FINALIZE when you exit from an MPI program

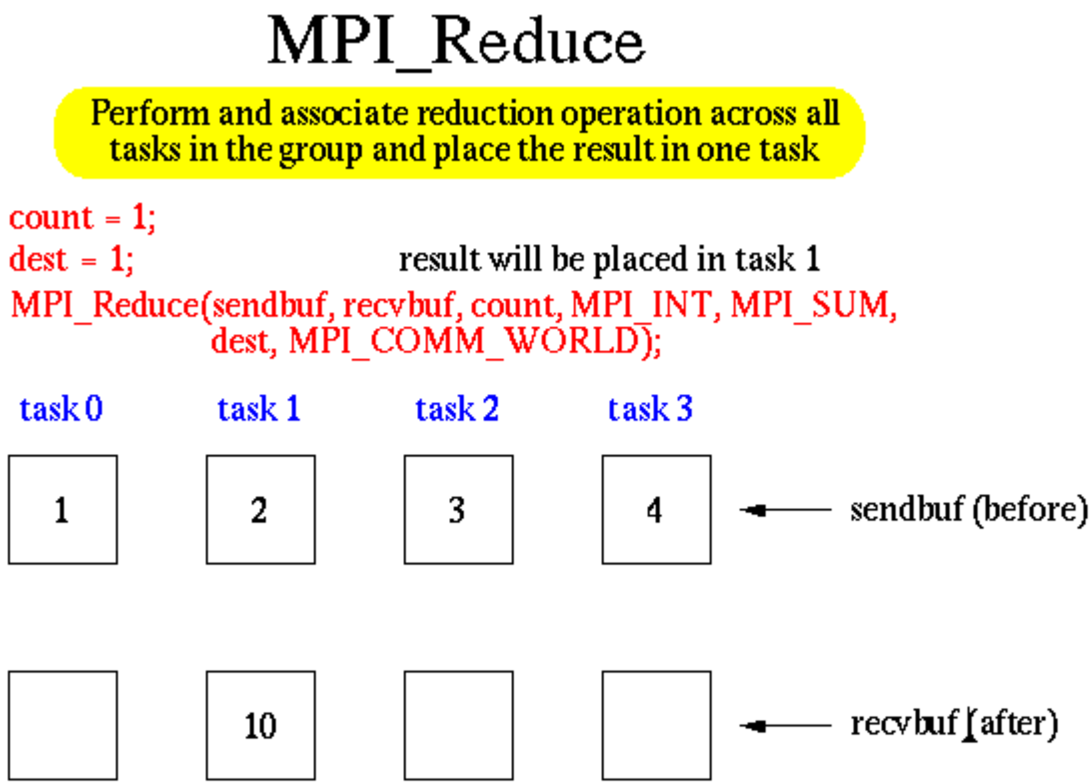
Finding Out About the Parallel Environment

- Two of the first questions asked in a parallel program are:
  - "How many processes are there?"
  - "Who am I?"
- "How many" is answered with the function call MPI_COMM_SIZE
- "Who am I" is answered with the function call MPI_COMM_RANK
  - The rank is a number between zero and (size - 1)

Exercise

- Install MPICH (or openmpi)
- Write an MPI application to calculate π

MPI_Reduce


MPI_Reduce
Perform and associate reduction operation across all tasks in the group and place the result in one task
count = 1;
dest = 1;
result will be placed in task 1
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, dest, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
10
recvbuf (after)

MPI: PI

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

static long num_steps = 100000;
double step;

void main (int argc, char** argv) {
 int i, id, num_procs;
 double x, pi, sum = 0.0;

 MPI_Init(&argc, &argv);

 step = 1.0/(double) num_steps;

 MPI_Comm_rank(MPI_COMM_WORLD, &id);
 MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

MPI: PI

```
for (i=(id+1);i<= num_steps; i+=num_procs){
   x = (i-0.5)*step;
   sum = sum + 4.0/(1.0+x*x);
 }

 MPI_Reduce(&sum, &pi, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
 if (id == 0){
   pi *= step;
   printf("pi is %f \n",pi);
 }
}
```

Reference

- http://www.mpi-forum.org
- http://www.llnl.gov/computing/tutorials/mpi/
- http://mvapich.cse.ohio-state.edu/
- http://www-unix.mcs.anl.gov/mpi/tutorial/
- MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/)
- Open MPI (http://www.open-mpi.org/)