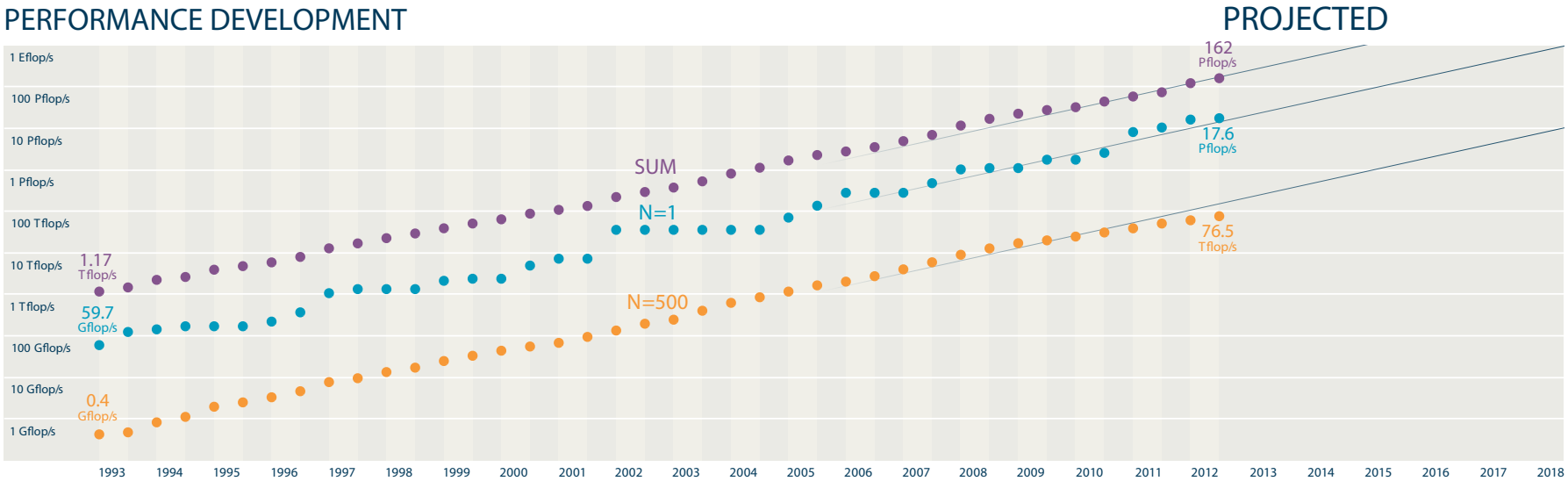
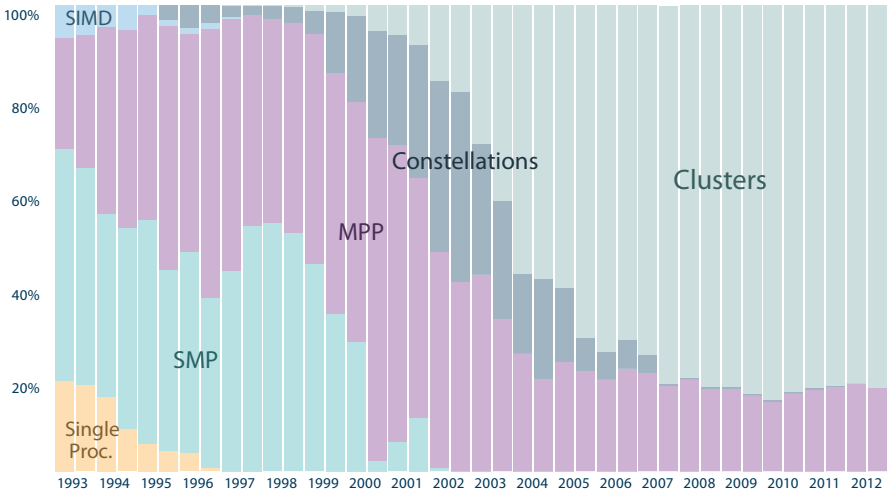
Introduction to MPI

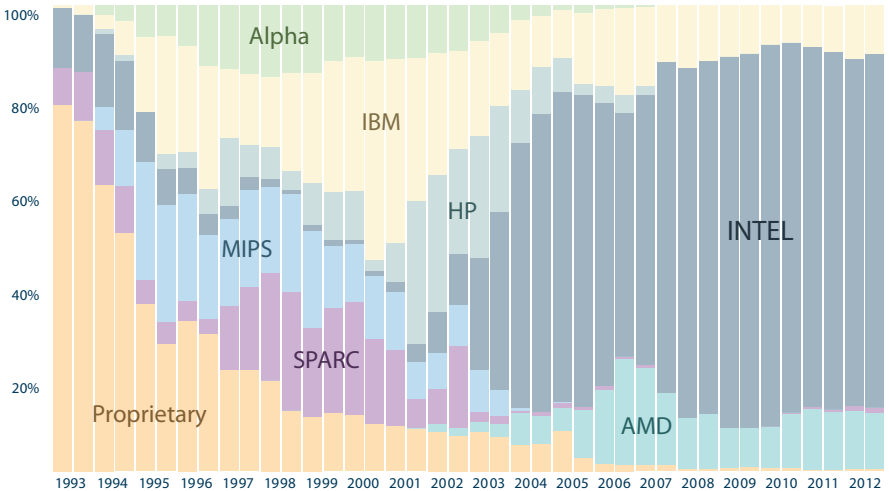
May 20, 2013

Daniel J. Bodony
Department of Aerospace Engineering
University of Illinois at Urbana-Champaign

Top500.org


PERFORMANCE DEVELOPMENT
PROJECTED
1 Eflop/s
100 Pflop/s
10 Pflop/s
1 Pflop/s
100 Tflop/s
10 Tflop/s
1 Tflop/s
100 Gflop/s
10 Gflop/s
1 Gflop/s
SUM
N=1
N=500
1.17 Tflop/s
59.7 Gflop/s
0.4 Gflop/s
162 Pflop/s
17.6 Pflop/s
76.5 Tflop/s
1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018


ARCHITECTURES
100%
80%
60%
40%
20%
SIMD
SMP
Single Proc.
MPP
Constellations
Clusters
1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012


CHIP TECHNOLOGY
100%
80%
60%
40%
20%
Alpha
IBM
HP
MIPS
SPARC
Proprietary
AMD
INTEL
1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012

Types of Parallelism

SISD	Single Instruction–Single Data	single core PCs
SIMD	Single Instruction–Multiple Data	GPUs
MISD	Multiple Instruction–Single Data	Esoteric
MIMD	Multiple Instruction–Multiple Data	multi-core PCs; most parallel computers

Types of Parallelism

SISD	Single Instruction–Single Data	single core PCs
SIMD	Single Instruction–Multiple Data	GPUs
MISD	Multiple Instruction–Single Data	Esoteric
MIMD	Multiple Instruction–Multiple Data	multi-core PCs; most parallel computers

$$\text{MIMD} \implies \text{Need to manage tasks and data}$$

$$\implies \text{MPI}$$

What is MPI?

MPI—Message Passing Interface

- Developed in early 1990s
- "...a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs" [a]
- Describes a means to pass information between processes

[a] Blaise, `https://computing.llnl.gov/tutorials/mpi`

What is MPI?

MPI—Message Passing Interface

- Developed in early 1990s
- "...a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs" [a]
- Describes a means to pass information between processes

[a] Blaise, `https://computing.llnl.gov/tutorials/mpi`

MPI implementations

- Open source
  - MPICH (DOE's Argonne) `http://www.mpich.org`
  - MVAPICH (OSU) `http://mvapich.cse.ohio-state.edu/overview/mvapich2`
  - OpenMPI (community) `http://www.open-mpi.org`
- Vendor-specific (IBM, SGI, ...)

Why Use MPI?

1. Available: on nearly every parallel computer (including single CPU, multi-core systems)
2. Flexible: runs equally well on distributed memory, shared memory, and hybrid machines
3. Portable: same code runs on different machines without modification[1]
4. Fast: vendors *can* implement (parts of it) in hardware
5. Scalable: demonstrated to work well on 1,000,000+ processes
6. Stable: future exascale machines are expected to have MPI
7. Multi-lingual: MPI available in Fortran 77/90+, C, C++, Python, ...

[1] Usually, unless you and/or the implementation has a bug

MPI $\Longrightarrow$ Explicit Parallelism

Within MPI you, the programmer, must explicitly include all parallelism, including:

- Task decomposition (how the data is to be distributed)
- Task granularity (problem size per MPI process)
- Message organization & handling (send, receive)
- Development of parallel algorithms (*e.g.*, sparse mat-vec multiply)

MPI $\Longrightarrow$ Explicit Parallelism

Within MPI you, the programmer, must explicitly include all parallelism, including:

- Task decomposition (how the data is to be distributed)
- Task granularity (problem size per MPI process)
- Message organization & handling (send, receive)
- Development of parallel algorithms (*e.g.*, sparse mat-vec multiply)

Other programming models, like OpenMP, can handle the parallelism at the compile stage (but usually at a loss of performance or scalability)

Using MPI & OpenMP together has some advantages on emerging computers (*e.g.*, $\sim$1,000 nodes with $\sim$100s cores each)

MPI Program Structure in Fortran

```
Program NULL
  Include 'mpif.h'
  Implicit None
  Integer :: rank, numprocs, ierr
  Call MPI_Init(ierr)
  Call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
  Call MPI_Comm_size( MPI_COMM_WORLD, numprocs, ierr )
  Write (*,'(2(A,I5),A)') 'Processor ', rank, ' of ', &
    numprocs, ' ready.'
  < do stuff >
  Call MPI_Finalize ( ierr )
End Program
```

MPI Program Structure in C

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main (int argc, char *argv[])
{
  int ierr, rank, numprocs;
  ierr = MPI_Init();
  ierr = MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
  ierr = MPI_Comm_size ( MPI_COMM_WORLD, &numprocs );
  printf("Processor %d of %d ready.\n"; rank, numprocs);
  < do stuff >
  ierr = MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Dissecting The Program Structure

Structure of an MPI program (in any language):

1. Initialize code in *serial*
2. Initialize MPI
3. Do good work in parallel
4. Finalize MPI
5. Finalize code in *serial*

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main (int argc, char *argv[])
{
  int ierr, rank, numprocs;
  ierr = MPI_Init();
  ierr = MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
  ierr = MPI_Comm_size ( MPI_COMM_WORLD, &numprocs );
  printf("Processor %d of %d ready.\n"; rank, numprocs);
  < do stuff >
  ierr = MPI_Finalize();
  return EXIT_SUCCESS;
}
```

The MPI commands above define the Environment Management routines.

MPI_COMM_WORLD and Communicators

MPI Communicators

- Establish a group of MPI processes
- MPI_COMM_WORLD is the default and contains all processes
- Can be created, copied, split, destroyed
- Serve as an argument for most point-to-point and collective operations

Communicator datatypes

- Fortran: Integer (*e.g.*, `Integer :: mycomm`)
- C: MPI_Comm (*e.g.*, `MPI_Comm mycomm;`)
- C++: (*e.g.*, `MPI::Intercomm mycomm;`)

Communicator Examples :: MPI_COMM_WORLD

30	31	32	33	34	35
24	25	26	27	28	29
18	19	20	21	22	23
12	13	14	15	16	17
6	7	8	9	10	11
0	1	2	3	4	5

MPI_COMM_WORLD contains all 36 processors, ranks 0 through 35

Communicator Examples :: Other Possibilities

30	31	32	33	34	35
24	25	26	27	28	29
18	19	20	21	22	23
12	13	14	15	16	17
6	7	8	9	10	11
0	1	2	3	4	5

Each color could be a different communicator

Communicator Examples :: Other Possibilities

Each color (including white) could be a different communicator

Interlude #1—Parallel 'Hello, World!'

Write a MPI program that prints "Hello, World!" as below:

```
rank 0:  ‘‘Hello, World!’’
rank 1:  ‘‘Hello, World!’’
...
rank <numprocs-1>:  ‘‘Hello, World!’’
```

Do the processes write in ascending order?

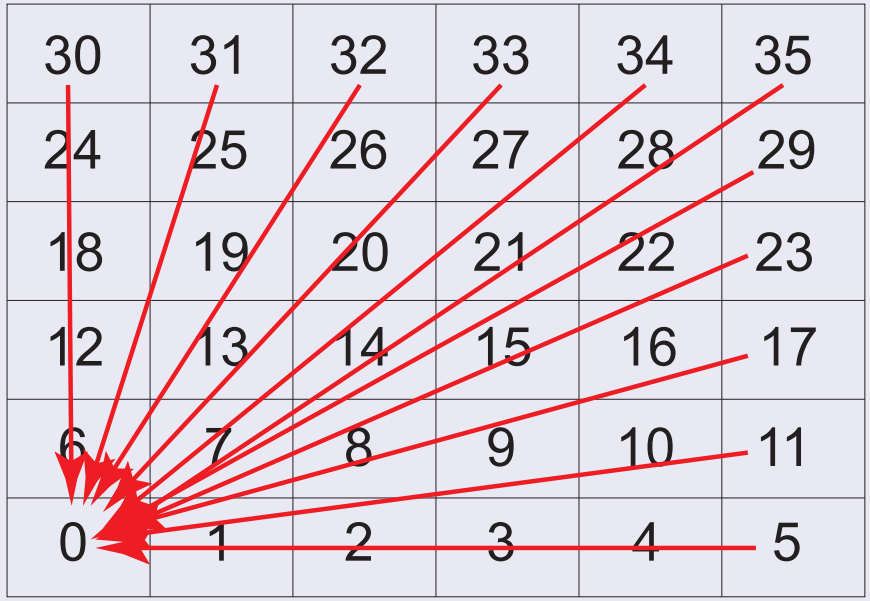
You have 30 minutes for this task.

Interprocess Communication in MPI

There are two primary types of communication

Point-to-point

Collective



Point-to-Point Send/Recv Communication in MPI

Process 13 'Send'

```
...
if (rank == 13) &
     Call MPI_Send (sbuf, count, type, dest, &
                    tag, comm, ierr)
...
```

variable	comment
sbuf	sending variable
count	sbuf element size
type	sbuf variable type
dest	process id in comm
tag	message tag
comm	communicator

Process 27 'Recv'

```
...
if (rank == 27) &
     Call MPI_Recv (rbuf, count, type, source, &
                    tag, comm, status, ierr)
...
```

variable	comment
rbuf	receiving variable
count	rbuf element size
type	rbuf variable type
source	process id in comm
tag	message tag
comm	communicator
status	message status

Point-to-Point Send/Recv Communication in MPI

```
Call MPI_Send (sbuf, count, type, dest, tag, comm, ierr)
```

Example:

```
Integer :: N, dest, tag, ierr
Real(KIND=8), Pointer :: sbuf(:)
...
N = 100
tag = 1
dest = 27
allocate(sbuf(N))
Call MPI_Send (sbuf, N, MPI_REAL8, dest, tag, &
               MPI_COMM_WORLD, ierr)
...
```

Point-to-Point Send/Recv Communication in MPI

```
Call MPI_Recv (rbuf, count, type, source, tag, comm, status, i
```

Example:

```
Integer :: N, dest, tag, ierr
Real(KIND=8), Pointer :: rbuf(:)
Integer :: status(MPI_STATUS_SIZE)
...
N = 100
tag = 1
source = 13
allocate(rbuf(N))
Call MPI_Recv (rbuf, N, MPI_REAL8, source, tag, &
               MPI_COMM_WORLD, status, ierr)
...
```

Note: `status` contains source id, tag, and count of the message received (useful for debugging when sending multiple messages)

MPI Predefined Datatypes[2]

Fortran	C	Comment
MPI_CHARACTER	MPI_CHAR	Character
MPI_INTEGER	MPI_INT	Integer
MPI_REAL	MPI_FLOAT	Single precision
MPI_REAL8	—	`Real(KIND=8)` in Fortran
MPI_DOUBLE_PRECISION	MPI_DOUBLE	Double precision
MPI_LOGICAL	MPI_C_BOOL	Logical
...		

[2] There exist many more!

Interlude #2a—Send/Recv Example

Write a MPI program that passes one integer from process 0 to process `numprocs-1` through each process (1, 2, ..., `numprocs-2`) and adds one to it after each `MPI_Recv`.

You have 30 minutes for this task.

MPI_Send and MPI_Recv Are Blocking

When `MPI_Send` or `MPI_Recv` are called, that process waits and is blocked from performing further communication and/or computation.

Advantage: buffers `sbuf` and `rbuf` are able to be reused once the call finishes.

Disadvantage: code waits for matching Send/Recv pair to complete.

MPI provides non-blocking communication routines `MPI_Isend` and `MPI_Irecv`.

```
Call MPI_Isend (sbuf, count, datatype, dest, tag, comm, &
                request, ierr)
Call MPI_Irecv (rbuf, count, datatype, src, tag, comm, &
                request, ierr)
Call MPI_Wait (request, status, ierr)
```

Must watch buffer use closely.

Non-Blocking Point-to-Point Communication

Communication and computation can overlap—the key to getting good scalability!

High-Level Example

```
...
Call MPI_Isend (sbuf, count, datatype, dest, tag(1), comm, &
                request(1), ierr)
Call MPI_Irecv (rbuf, count, datatype, src, tag(2), comm, &
                request(2), ierr)

< do work while the Send and Recv complete >

Call MPI_Waitall (2, request, status, ierr)
...
```

Interlude #2b—Blocking vs. Nonblocking Send/Recv

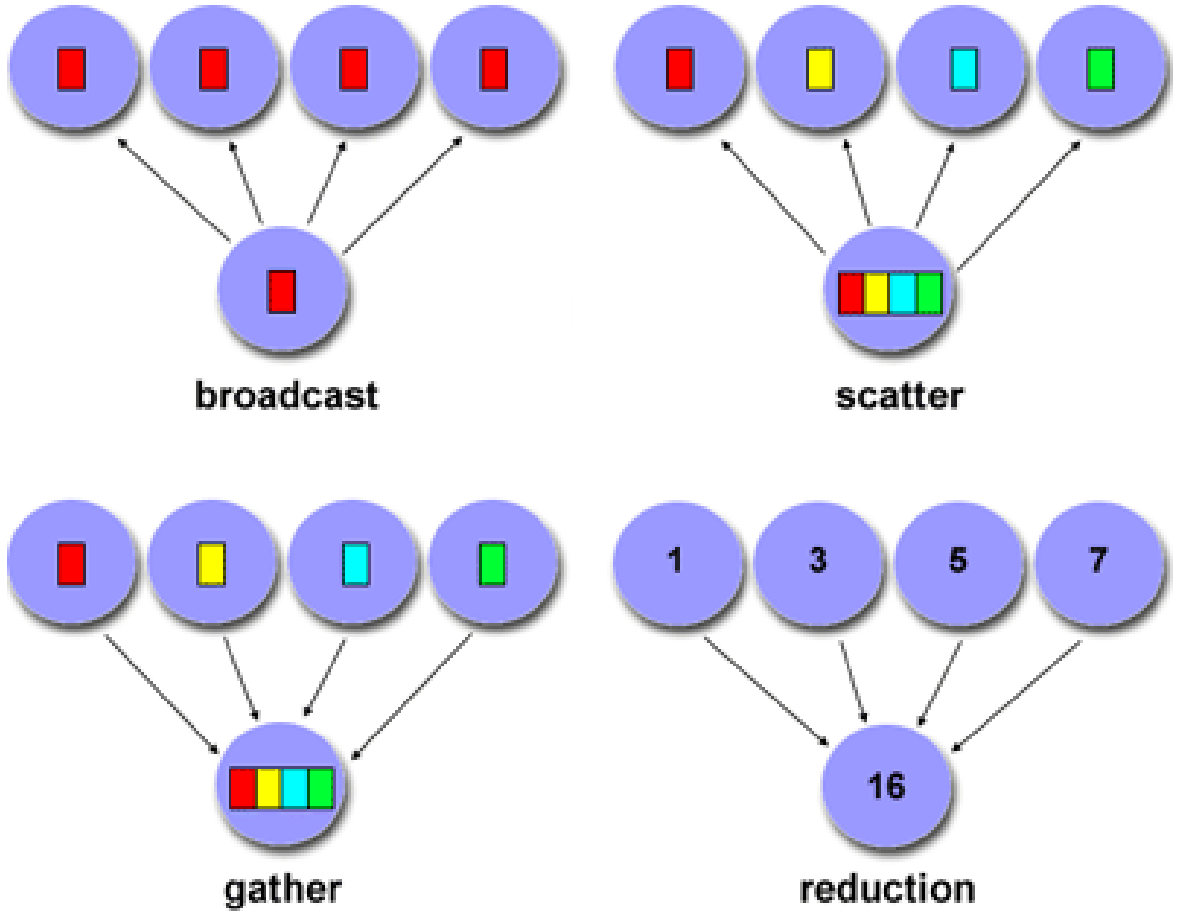
Compile and run Blaise Barney's `mpi_bandwidth.[c,f]` and `mpi_bandwidth_nonblock.[c,f]` to observe the differences between blocking and non-blocking communications.

file	url
mpi_bandwidth.c	`https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_bandwidth.c`
mpi_bandwidth_nonblock.c	`https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_bandwidth_nonblock.c`
mpi_bandwidth.f	`https://computing.llnl.gov/tutorials/mpi/samples/Fortran/mpi_bandwidth.f`
mpi_bandwidth_nonblock.f	`https://computing.llnl.gov/tutorials/mpi/samples/Fortran/mpi_bandwidth_nonblock.f`

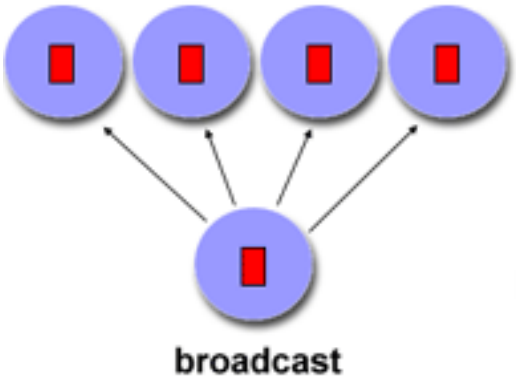
You have 15 minutes for this task.

Collective Communications

In many circumstances, all processes on a communicator must exchange data $\Longrightarrow$ collective communication.



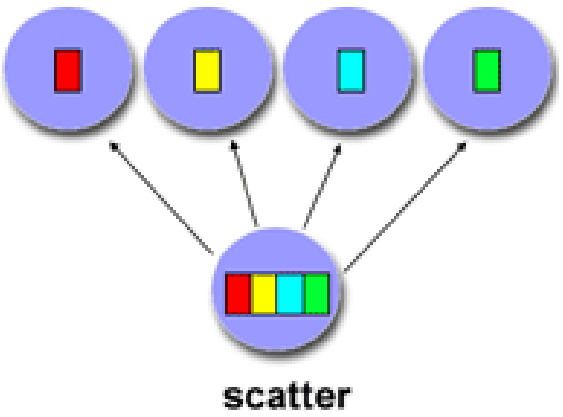
MPI_Bcast




```
Call MPI_BCAST (buffer, count, datatype, &
                root, comm, ierr)

MPI_Bcast (&buffer, count, datatype,
           root, comm);
```

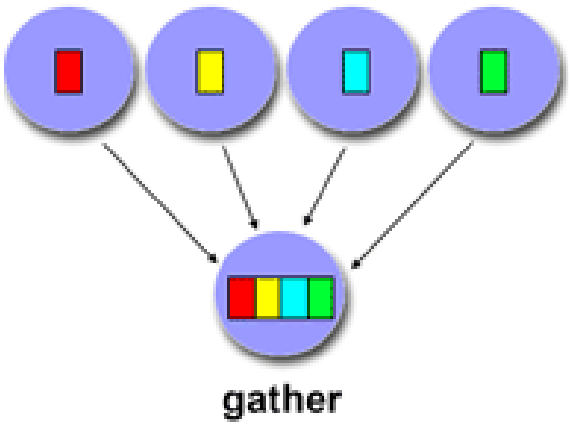
MPI_Scatter




```
Call MPI_SCATTER (sbuf, scount, stype, &
                  rbuf, rcount, rtype, &
                  root, comm, ierr)

MPI_Scatter (&sbuf, scount, stype,
             &rbuf, rcount, rtype,
             root, comm);
```

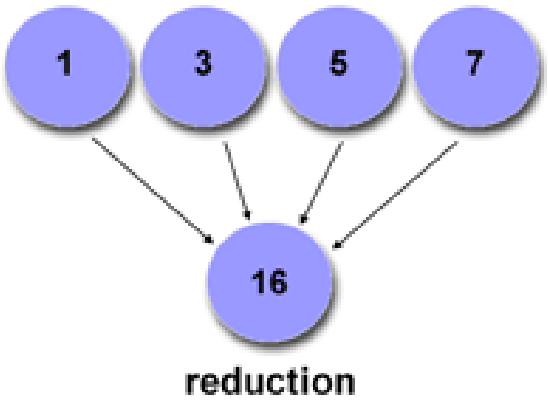
MPI_Gather




```
Call MPI_GATHER (sbuf, scount, stype, &
                 rbuf, rcount, rtype, &
                 root, comm, ierr)
```

```
MPI_Gather (&sbuf, scount, stype,
            &rbuf, rcount, rtype,
            root, comm);
```

MPI_Reduce




```
Call MPI_REDUCE (sbuf, rbuf, rcount, &
                 rtype, op, root, &
                 comm, ierr)

MPI_Reduce (&sbuf, &rbuf, rcount, &
            rtype, op, root, &
            comm);
```

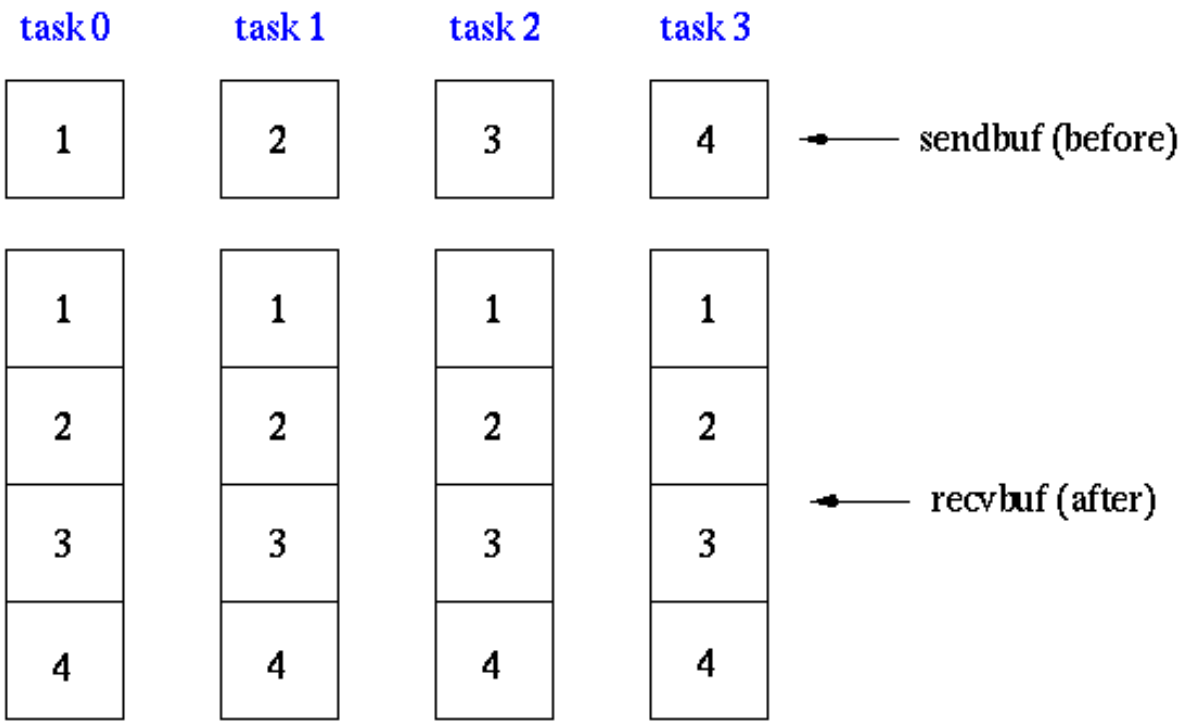
op	description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
⋮	

'All' Variants of MPI Collective Routines

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);
```

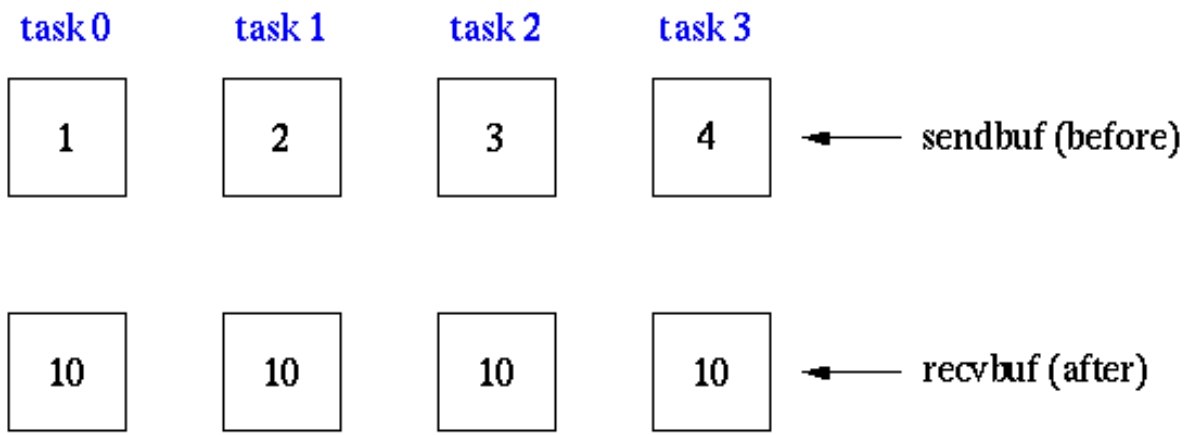


'All' Variants of MPI Collective Routines

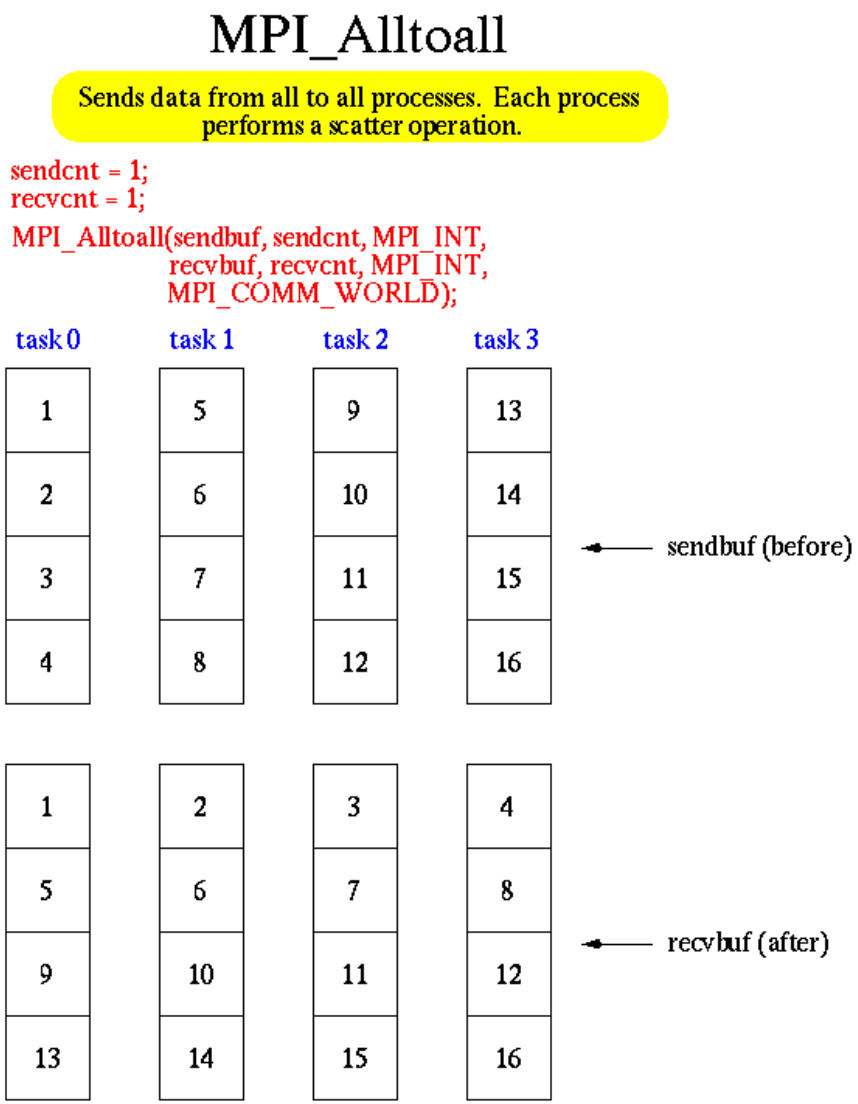
MPI_Allreduce

Perform and associate reduction operation across all tasks in the group and place the result in all tasks

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
              MPI_COMM_WORLD);
```



‘All’ Variants of MPI Collective Routines



Useful for matrix transpose and FFTs. (NOTE: It’s future on exascale machines is not clear.)

Interlude #3—Parallel Inner Product

Develop a parallel code to take the scalar product of two $N \times 1$ vectors $\vec{x}$ and $\vec{y}$, *i.e.*, $\vec{x}^T \vec{y}$. Choose $N > 5N_p$, where N_p is the number of MPI processes. Have the scalar answer stored on all processors.

You have 30 minutes for this task.

Interlude #4—Parallel Matrix Transpose

Use `MPI_Alltoall` to take the transpose of a $10N_p \times 10N_p$ matrix, where N_p is the number of MPI processes used.

You have 30 minutes for this task.

Summary

What MPI commands did we learn?

- MPI_Init
- MPI_Comm_rank, MPI_Comm_size
- MPI_Finalize
- MPI_Send, MPI_Recv
- MPI_Isend, MPI_Irecv
- MPI_Waitall
- MPI_Scatter, MPI_Bcast
- MPI_Gather, MPI_Reduce
- MPI_Allreduce, MPI_Allscatter
- MPI_Alltoall
- MPI_Barrier

These commands will get you very far in your parallel program development.

Additional Resources

1. http://www.mcs.anl.gov/research/projects/mpi/tutorial/index.html
2. https://computing.llnl.gov/tutorials/parallel_comp/
3. https://computing.llnl.gov/tutorials/mpi/
4. http://www.mpi-forum.org/