

Parallel Programming Using MPI

Short Course on HPC

14th September 2019

Aditya Krishna Swamy

adityaks@iisc.ac.in

SERC, Indian Institute of Science

What is MPI?

- MPI stands for **Message Passing Interface**
- It is a message-passing **specification**, a **standard**, for the vendors to implement
- In practice, MPI is a **library** consisting of C functions and Fortran subroutines (Fortran) used for **exchanging** data between processes
- An MPI library exists on **ALL** parallel computing facilities so it is **highly portable**
- Also available for Python (mpi4py.scipy.org), R (Rmpi)

Why use MPI?

- General
 - MPI-1 '92-'94, MPI-2 ~2008, MPI-3 2012. Has been around for ~25 years
 - Widely used parallel model
 - Libraries and algorithms readily available
 - Very scalable: 1 ~ 300,000 cores
 - Portable
 - Works well with Hybrid models (MPI+X. X=OpenMP, CUDA, OpenACC, OpenCL...)
- Your problem
 - Want to speed up your calculation
 - Want to scale up your problem size
 - Your problem size is too large for a single node

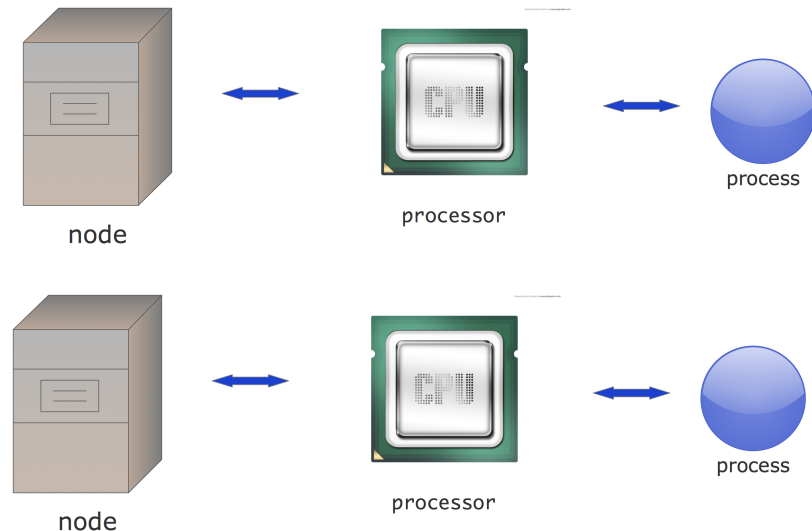
MPI Implementations

- MPICH
 - ANL, (foss) mpich.org/
 - Intel, (free 1-year student license)
 - Cray
 - IBM
- OpenMPI, (foss)
 - open-mpi.org/
- MVAPICH (free, BSD)
 - mvapich.cse.ohio-state.edu/

MPI

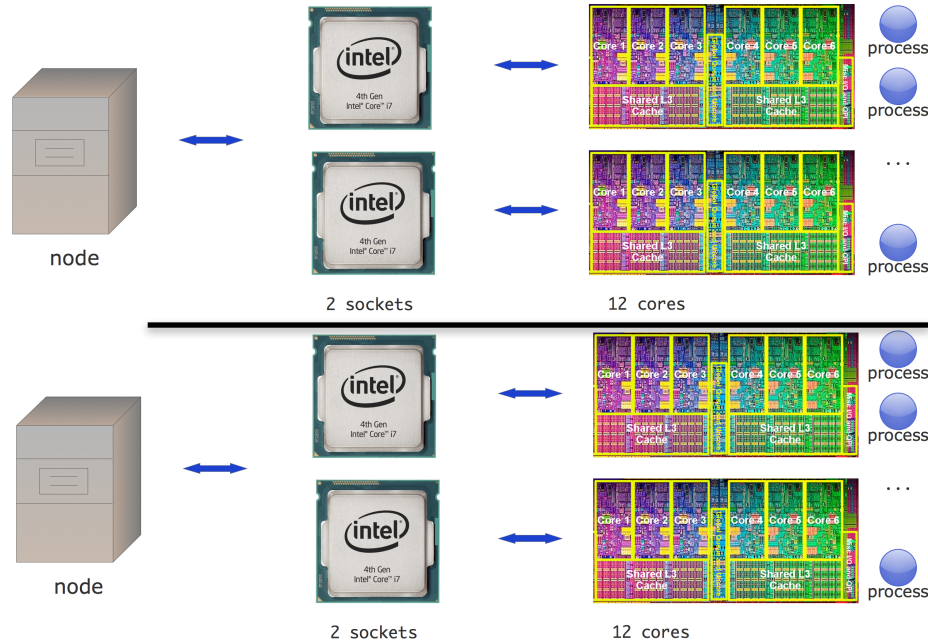
Context: Distributed memory parallel computers

- Each processor has its own memory and cannot access the memory of other processors
- A copy of the same executable runs as MPI process (on each processor core)
- All variables are private to each process
- Any data to be shared must be explicitly transmitted from one to another

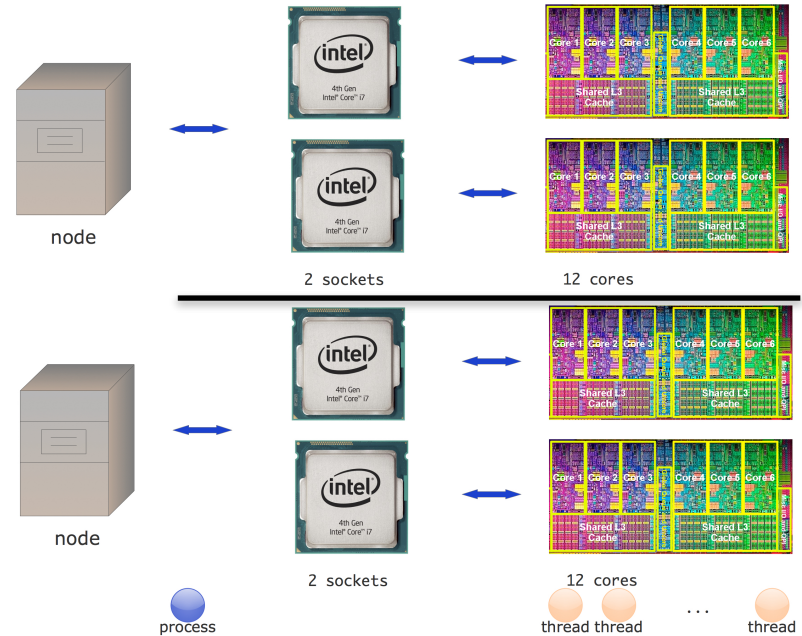


Cluster - 90's

Modern HPC Facilities



MPI only



Hybrid

Terminology

OS/Software

Process	Independent stream of instructions. OS provides dedicated resources
Thread(s)	Created by process Shares resources with process/fellow threads

Hardware

core	“instruction stream processing unit”
“Processor/CPU”	Single die that sits on a “Socket” (has 1 or more cores)
Node	~ “mother board”. With 1 or more sockets
Blade	1 or more Nodes
Cabinet	Several blades

Basic MPI

- Basic functionality in a parallel program
 - Start processes
 - Send messages
 - receive messages
 - Synchronize

A simple program in C

```
#include <stdio>
#include <stdlib>
```

```
int main( int argc, char *argv[] )
{
```

```
    printf("Hello \n")
```

```
return 0;
}
```

Any PC

\$./a.out

Hello

A simple MPI program in C

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

int main( int argc, char *argv[] )
{
    int nproc, myrank;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

    printf("Hello from %d. \n",myrank)

    /* Finalize */
    MPI_Finalize();
    return 0;
}
```

Any Cluster

\$ mpirun -n 8 ./a.out

SahasraT

\$ aprun -n 8 ./a.out

Hello from 0.

Hello from 1.

Hello from 2.

Hello from 3.

Hello from 4.

Hello from 5.

Hello from 6.

Hello from 7.

Header file

- Defines MPI-related parameters and functions
- Must be included in all routines calling MPI functions
- Can also use include file:
include mpif.h

```
#include "mpi.h"
int main( int argc, char *argv[]
{
    int nproc, myrank;
    /* Initialize MPI */
    MPI_Init(&argc,&argv);
    /* Get the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    /* Get my process number (rank) */
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

    Do work and make message passing calls...

    /* Finalize */
    MPI_Finalize();
    return 0;
}
```

Initialization

```
#include "mpi.h"
int main( int argc, char *argv[] )
{
    int nproc, myrank;
    /* Initialize MPI */
    MPI_Init(&argc,&argv);
    /* Get the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    /* Get my process number (rank) */
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

    Do work and make message passing calls...

    /* Finalize */
    MPI_Finalize();
    return 0;
}
```

- Must be called at the beginning of the code before any other calls to MPI functions
- Sets up the communication channels between the processes and gives each one a rank.

How many processes do we have?

- Returns the number of processes available under MPI_COMM_WORLD communicator
- This is the number used on the mpiexec (or mpirun) command:

```
mpiexec -n nproc a.out
```

```
int main(int argc, char** argv) {  
    MPI_Init(&argc, &argv);  
    /* Get the number of processes */  
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);  
    /* Get my process number (rank) */  
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
  
    Do work and make message passing calls...  
  
    /* Finalize */  
    MPI_Finalize();  
    return 0;  
}
```

What is my rank?

```
#include "mpi.h"
int main( int argc, char **argv )
{
    int nproc, myrank;
    /* Initialize MPI */
    MPI_Init(&argc, &argv);
    /* Get the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    /* Get my process number (rank) */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    /* Do work and make message passing calls... */

    /* Finalize */
    MPI_Finalize();
    return 0;
}
```

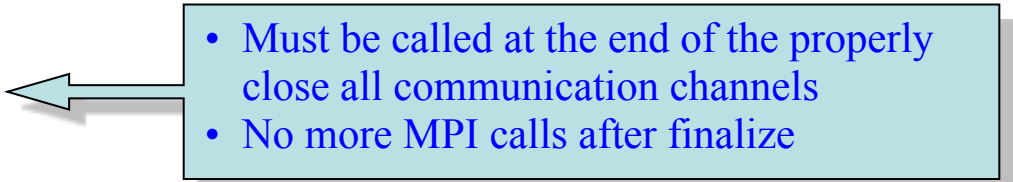
- Get my rank among all of the nproc processes under MPI_COMM_WORLD
- This is a unique number that can be used to distinguish this process from the others

Termination

```
#include "mpi.h"
int main( int argc, char *argv[] )
{
    int nproc, myrank;
    /* Initialize MPI */
    MPI_Init(&argc,&argv);
    /* Get the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    /* Get my process number (rank) */
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

    Do work and make message passing calls...

    /* Finalize */
    MPI_Finalize();
    return 0;
}
```

- 
- Must be called at the end of the properly close all communication channels
 - No more MPI calls after finalize

MPI Communicators

- **An MPI Function:** `MPI_Comm_size(MPI_COMM_WORLD, &nproc);`
- `MPI_COMM_WORLD` - communicator
- A communicator is a group of processes
 - Each process has a unique rank within a specific communicator
 - Rank starts from **0** and has a maximum value of $(nproc-1)$. Fortran programmers beware!
 - Internal mapping of processes to processing units
 - Necessary to specify communicator when initiating a communication by calling an MPI function or routine.
- Default communicator `MPI_COMM_WORLD`, which contains all available processes.
- Several communicators can coexist
 - A process can belong to different communicators at the same time, but has a unique rank in each communicator

A sample MPI program in Fortran 90

```
Program mpi_code
  ! Load MPI definitions
  use mpi (or include mpif.h)

  ! Initialize MPI
  call MPI_Init(ierr)

  ! Get the number of processes
  call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)

  ! Get my process number (rank)
  call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

  Do work and make message passing calls...

  ! Finalize
  call MPI_Finalize(ierr)

end program mpi_code
```

When hello-mpi runs

```
#include "mpi.h"
int main( int argc, char *argv[] ){

    int nproc, myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Hello from %d.\n",myrank);
    MPI_Finalize();
    return 0;
}
```

```
#include "mpi.h"
int main( int argc, char *argv[] ){

    int nproc, myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Hello from %d.\n",myrank);
    MPI_Finalize();
    return 0;
}
```

```
#include "mpi.h"
int main( int argc, char *argv[] ){

    int nproc, myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Hello from %d.\n",myrank);
    MPI_Finalize();
    return 0;
}
```

```
#include "mpi.h"
int main( int argc, char *argv[] ){

    int nproc, myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Hello from %d.\n",myrank);
    MPI_Finalize();
    return 0;
}
```

Any Cluster

\$ mpirun -n 8 ./a.out

SahasraT

\$ aprun -n 8 ./a.out

Hello from 0.

Hello from 4.

Hello from 2.

Hello from 1.

Hello from 3.

Hello from 6.

Hello from 7.

Hello from 5.

```
#include "mpi.h"
int main( int argc, char *argv[] ){

    int nproc, myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Hello from %d.\n",myrank);
    MPI_Finalize();
    return 0;
}
```

```
#include "mpi.h"
int main( int argc, char *argv[] ){

    int nproc, myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Hello from %d.\n",myrank);
    MPI_Finalize();
    return 0;
}
```

```
#include "mpi.h"
int main( int argc, char *argv[] ){

    int nproc, myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Hello from %d.\n",myrank);
    MPI_Finalize();
    return 0;
}
```

```
#include "mpi.h"
int main( int argc, char *argv[] ){

    int nproc, myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Hello from %d.\n",myrank);
    MPI_Finalize();
    return 0;
}
```

How much do I need to know?

- MPI is large: MPI-1 has over 125 functions/subroutines. MPI-3 has over 400
- MPI is small: Can actually most work with about 6 functions!
- Collective functions are **EXTREMELY** useful since they simplify the coding and vendors optimize them for their interconnect hardware
- One can access flexibility when it is required.
- One need not master all parts of MPI to use it.

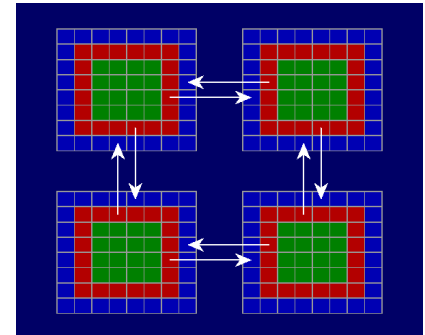
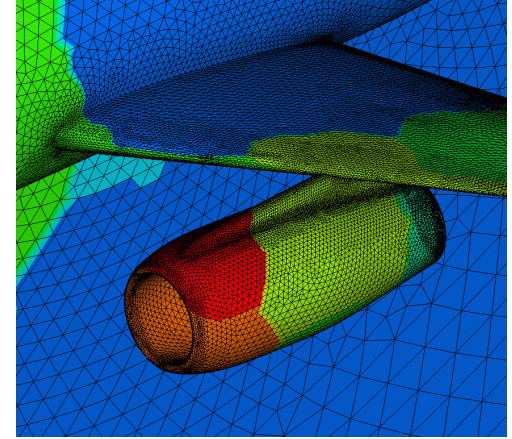
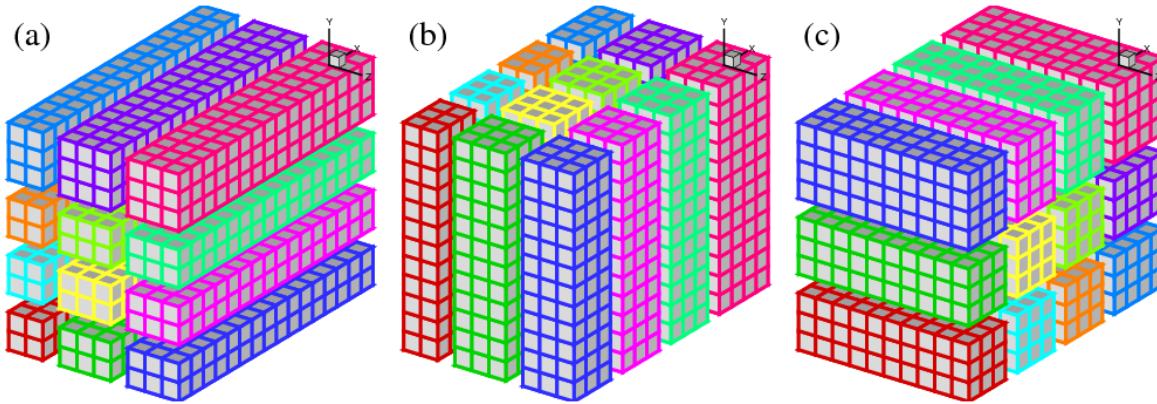
Do I need a Supercomputer?

- To learn MPI and develop MPI application - No. Your laptop/PC will suffice.
 - Any number of MPI processes can be started even on a laptop
 - Same accuracy, but not efficient
- Test your real application - ~~Laptop~~, PC, Workstation, Cluster, Supercomputer
- Production runs - ~~Laptop~~, PC, Workstation, Cluster, Supercomputer

Is communication needed?

Domain Decomposition

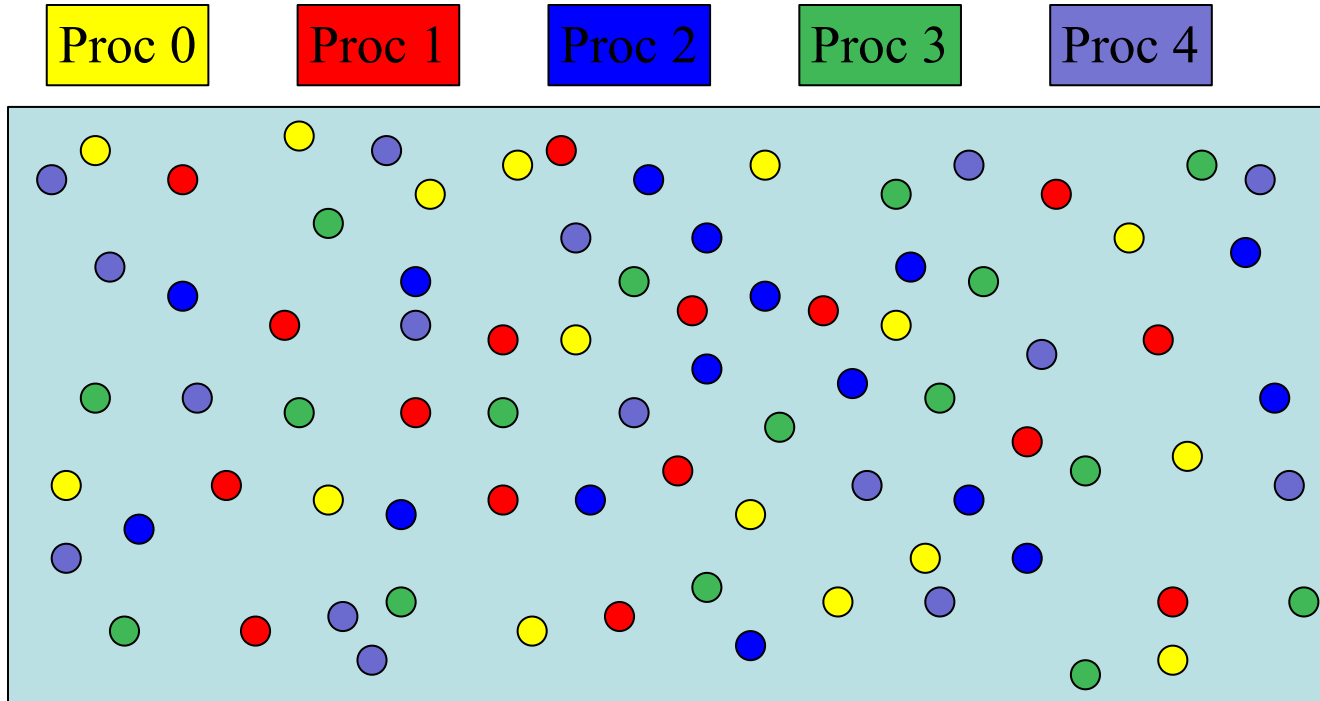
- Most widely used method for grid-based calculations



Is communication needed?

“Coloring”

- Useful for particle simulations



MPI Function Categories

- MPI calls to exchange data
 - Point-to-Point communications
 - Only 2 processes exchange data
 - It is the basic operation of all MPI calls
 - Collective communications
 - A single call handles the communication between all the processes in a communicator
 - There are 2 types of collective communications
 - Data movement (e.g. MPI_Bcast)
 - Reduction (e.g. MPI_Reduce)
- Synchronization:
 - MPI_Barrier
 - MPI_Wait

Send Message: MPI_Send

```
MPI_Send (&numToSend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD) ;
```

&numToSend	Pointer to whatever information to send. In this case, an integer
1	the number of items to send. If sending a vector of 10 int's, we would point to the first one in the above argument and set this to the size of the array.
MPI_INT	the type of object we are sending. In this case, an integer
0	Destination of the message. In this case, Rank 0
10	Message Tag. Useful to identify/sort messages
MPI_COMM_WORLD	We don't have any subsets yet. We just choose the "default"

Point to point: 2 processes at a time

`MPI_Recv(recvbuf, count, datatype, source, tag, comm, status)`

`MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,
recvbuf, recvcount, recvtype, source, recvtag, comm, status)`

Datatypes are:

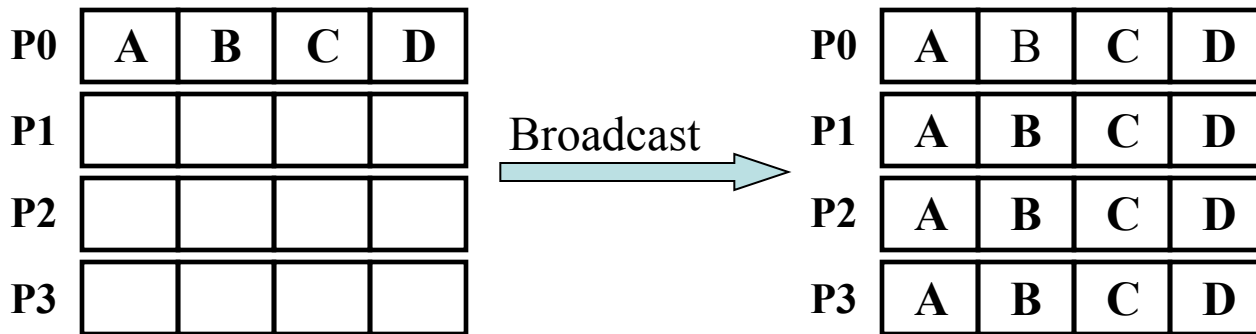
FORTRAN: `MPI_INTEGER`, `MPI_REAL`, `MPI_DOUBLE_PRECISION`,
`MPI_COMPLEX`, `MPI_CHARACTER`, `MPI_LOGICAL`, etc...

C : `MPI_INT`, `MPI_LONG`, `MPI_SHORT`, `MPI_FLOAT`, `MPI_DOUBLE`, etc...

Predefined Communicator: `MPI_COMM_WORLD`

Collective communication: Broadcast

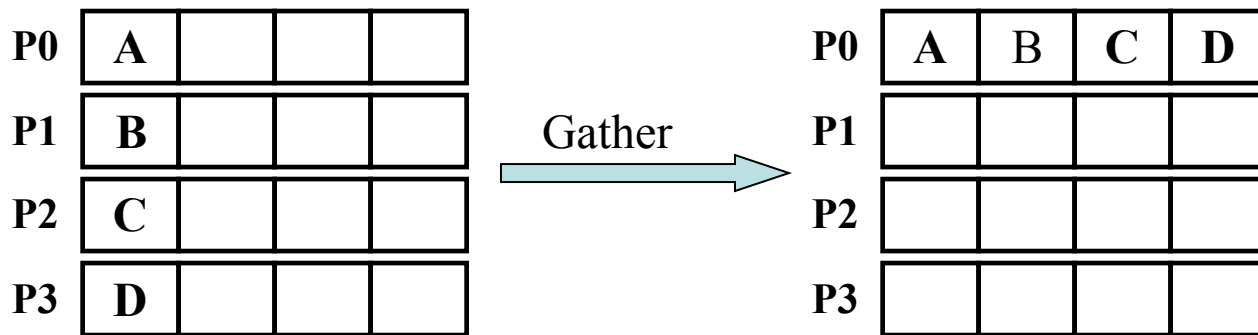
`MPI_Bcast(buffer, count, datatype, root, comm, ierr)`



- One process (called “root”) sends data to all the other processes in the same communicator
- Must be called by ALL processes with the same arguments

Collective communication: Gather

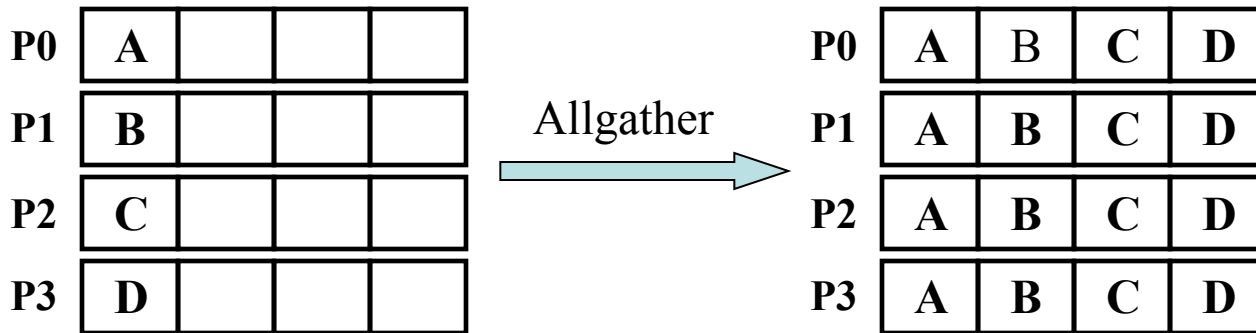
`MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, root, comm, ierr)`



- One root process collects data from all the other processes in the same communicator
- Must be called by all the processes in the communicator with the same arguments
- “sendcount” is the number of basic datatypes sent, not received (example above would be sendcount = 1)
- Make sure that you have enough space in your receiving buffer!

Collective communication: Gather to All

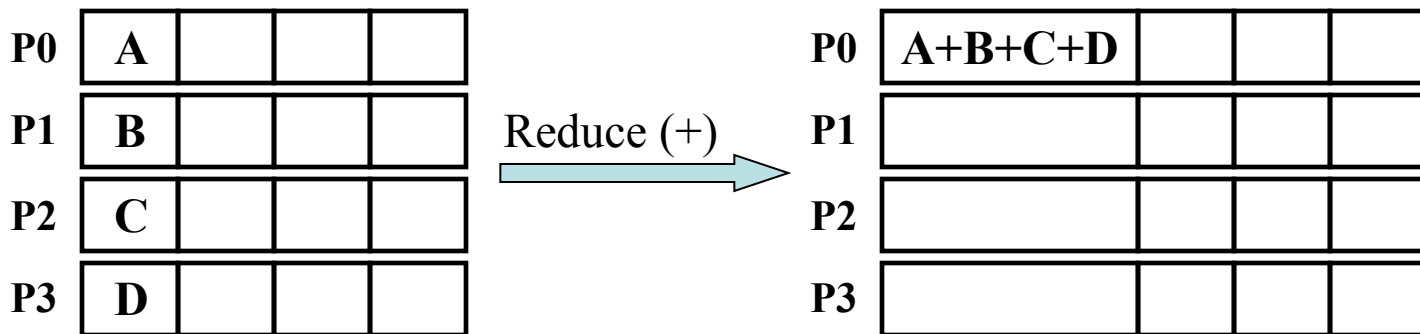
`MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info)`



- All processes within a communicator collect data from each other and end up with the same information
- Must be called by all the processes in the communicator with the same arguments
- Again, sendcount is the number of elements sent

Collective communication: Reduction

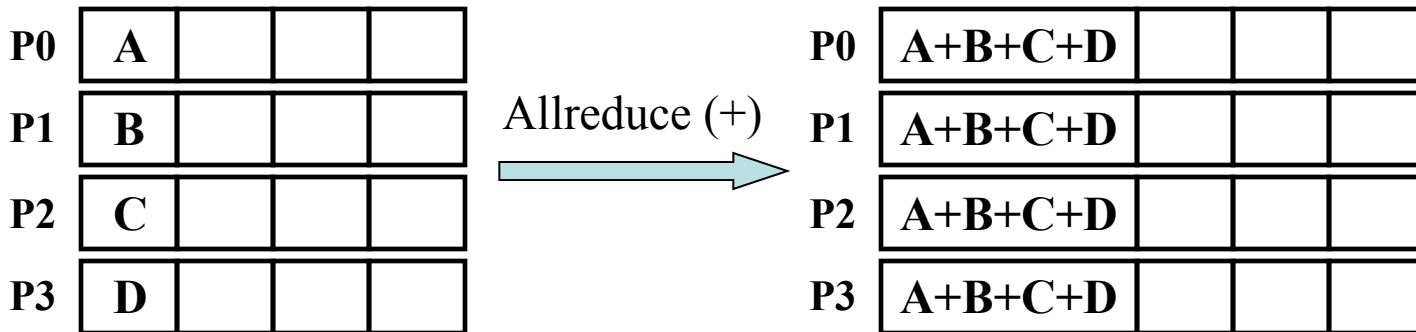
`MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm, ierr)`



- One root process collects data from all the other processes in the same communicator and performs an operation on the received data
- Called by all the processes with the same arguments
- Operations are: `MPI_SUM`, `MPI_MIN`, `MPI_MAX`, `MPI_PROD`, logical AND, OR, XOR, and a few more
- User can define own operation with `MPI_Op_create()`

Collective communication: Reduction to All

`MPI_Allreduce(sendbuf,recvbuf,count,datatype,op,comm,ierr)`



- All processes within a communicator collect data from all the other processes and performs an operation on the received data
- Called by all the processes with the same arguments
- Operations are the same as for MPI_Reduce

More MPI collective calls

One “root” process send a different piece of the data to each one of the other Processes (inverse of gather)

```
MPI_Scatter(sendbuf, sendcnt, sendtype, recvbuf, recvcnt,  
            recvtype, root, comm, ierr)
```

Each process performs a scatter operation, sending a distinct message to all the processes in the group in order by index.

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcnt,  
              recvtype, comm, ierr)
```

Synchronization: When necessary, all the processes within a communicator can be forced to wait for each other although this operation can be expensive

```
MPI_Barrier(comm, ierr)
```

How to time your MPI code

- Several possibilities but MPI provides an easy to use function called “MPI_Wtime()”. It returns the number of seconds since an arbitrary point of time in the past.

FORTTRAN: double precision MPI_WTIME()

C: double MPI_Wtime()

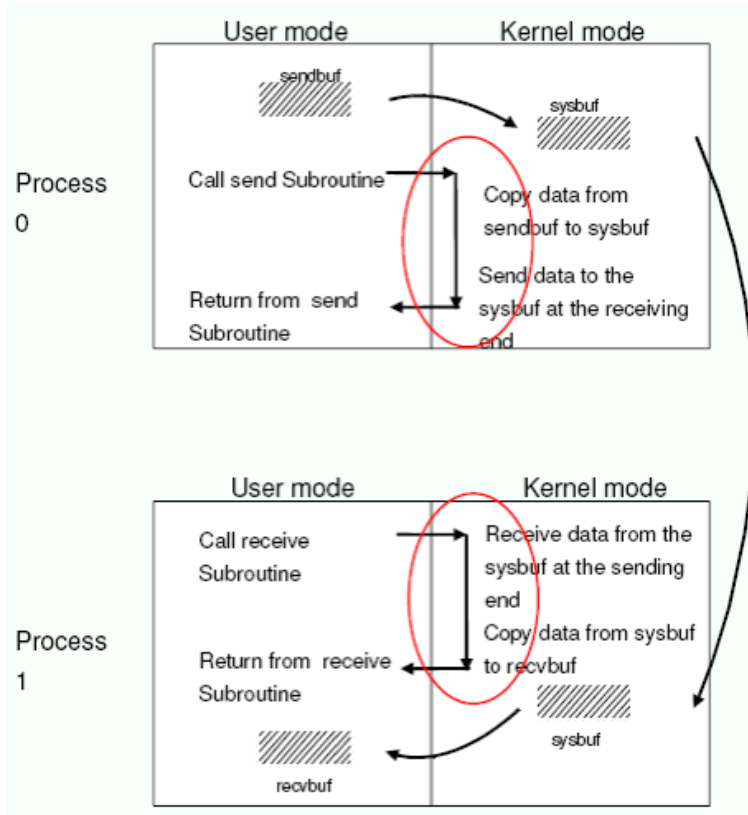
```
starttime=MPI_WTIME()
```

```
... program body ...
```

```
endtime=MPI_WTIME()
```

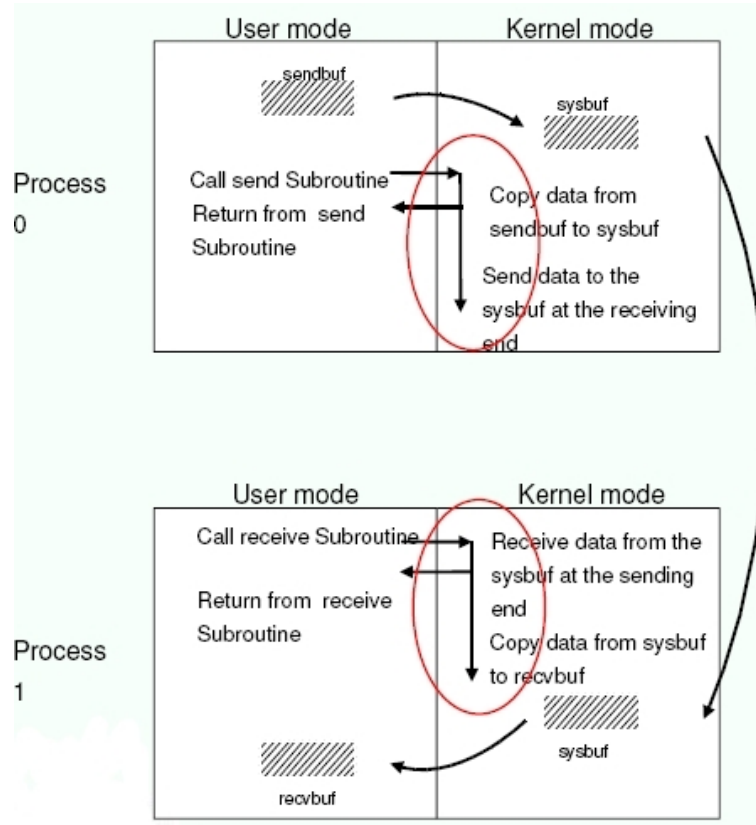
```
elapsetime=endtime-starttime
```

Blocking communications



- The call waits until the data transfer is done
 - The sending process waits until all data are transferred to the system buffer (differences for *eager* vs *rendezvous* protocols...)
 - The receiving process waits until all data are transferred from the system buffer to the receive buffer
- All collective communications are blocking

Non-blocking



- Returns immediately after the data transferred is initiated
- Allows to overlap computation with communication
- Need to be careful though
 - When send and receive buffers are updated before the transfer is over, the result will be wrong

Debugging tips

Use “unbuffered” writes to do “printf-debugging” and always write out the process id:

```
C:      fprintf(stderr,"%d: ...",myid,...) ;  
Fortran: write(0,*)myid,' : ...'
```

If the code detects an error and needs to terminate, use `MPI_ABORT`. The errorcode is returned to the calling environment so it can be any number.

```
C:      MPI_Abort(MPI_Comm comm, int errorcode) ;  
Fortran: call MPI_ABORT(comm, errorcode, ierr)
```

Use a parallel debugger such as Totalview or DDT

References

- Keywords for search “mpi”, or “mpi standard”, or “mpi tutorial”...
- <https://www.mpich.org/static/docs/latest/>
- <http://www.mpi-forum.org> (location of the MPI standard)
- <http://www.llnl.gov/computing/tutorials/mpi/>
- <http://www.nersc.gov/nusers/help/tutorials/mpi/intro/>
- MPI on Linux clusters:
 - MPICH (<http://www-unix.mcs.anl.gov/mpi/mpich/>)
 - Open MPI (<http://www.open-mpi.org/>)

Example: calculating π using numerical integration

```
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    FILE *ifp;

    ifp = fopen("ex4.in", "r");
    fscanf(ifp, "%d", &n);
    fclose(ifp);
    printf("number of intervals = %d\n", n);

    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = 1; i <= n; i++) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;

    pi = mypi;
    printf("pi is approximately %.16f, Error is %.16f\n",
           pi, fabs(pi - PI25DT));
    return 0;
}
```

C version

```

#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i, j, tag, my_n;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, pi_frac, tt0, tt1, ttf;
    FILE *ifp;
    MPI_Status  Stat;
    MPI_Request request;

    n = 1;
    tag = 1;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    tt0 = MPI_Wtime();
    if (myid == 0) {
        ifp = fopen("ex4.in","r");
        fscanf(ifp,"%d",&n);
        fclose(ifp);
        //printf("number of intervals = %d\n",n);
    }
    /* Global communication. Process 0 "broadcasts" n to all other processes */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

```

Root reads input
and broadcast to
all

Each process calculates its section of the integral and adds up results with MPI_Reduce

...

```
h    = 1.0 / (double) n;
sum  = 0.0;
for (i = myid*n/numprocs+1; i <= (myid+1)*n/numprocs; i++) {
    x = h * ((double)i - 0.5);
    sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum;

pi = 0.; /* It is not necessary to set pi = 0 */

/* Global reduction. All processes send their value of mypi to process 0
and process 0 adds them up (MPI_SUM) */
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

ttf = MPI_Wtime();
printf("myid=%d  pi is approximately %.16f, Error is %.16f  time = %10f\n",
      myid, pi, fabs(pi - PI25DT), (ttf-tt0));

MPI_Finalize();
return 0;
}
```

Thank you...

Non-blocking send and receive

Point to point:

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierr)
```

```
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierr)
```

The functions **MPI_Wait** and **MPI_Test** are used to complete a nonblocking communication

```
MPI_Wait(request, status, ierr)
```

```
MPI_Test(request, flag, status, ierr)
```

MPI_Wait returns when the operation identified by “request” is complete. This is a non-local operation.

MPI_Test returns “flag = true” if the operation identified by “request” is complete. Otherwise it returns “flag = false”. This is a local operation.

MPI-3 standard introduces “non-blocking collective calls”

MPI + OpenMP

- By default, MPI assumes no threaded execution.
- `MPI_Init(int *argc, char ***argv)` old.
- `MPI_Init_thread(int *argc, char ***argv, int required, int *provided )`
- `required` = MPI Thread Support level
- Levels of MPI Thread Support
-

Support Levels	Description
MPI_THREAD_SINGLE	Only one thread will execute
MPI_THREAD_FUNNELED	Process may be multi-threaded, but only the main thread will make MPI calls (calls are “funneled” to main thread). *Default*
MPI_THREAD_SERIALIZE	Process may be multi-threaded, and any thread can make MPI calls, but threads cannot execute MPI calls concurrently; they must take turns (calls are “serialized”).
MPI_THREAD_MULTIPLE	Multiple threads may call MPI, with no restriction.