

Parallel Computing: Intro to MPI

Fall Break Bootcamp 2022

October 19, 2022

Stéphane Ethier

(ethier@pppl.gov)

Princeton Plasma Physics Laboratory

Why Parallel Computing?

Why not run n instances of my code? Isn't that parallel computing?

YES... but

- You want to speed up your calculation because it takes a week to run!
- Your problem size is too large to fit in the memory of a single node
- Want to use those extra cores on your “multicore” processor
- **Solution:**
 - Split the work between several processor cores so that they can work in parallel
 - Exchange data between them when needed
- **How?**
 - Message Passing Interface (**MPI**) on **distributed memory systems** (works also on shared memory nodes)
 - **OpenMP** directives on **shared memory node**
 - and some other methods not as popular (pthreads, Intel TBB, Fortran Co-Arrays)

Programming for HPC: MPI+X

Top 5 of the Nov 2020 List of the top supercomputers in the world (www.top500.org)



PRESENTED BY



FIND OUT MORE AT

top500.org



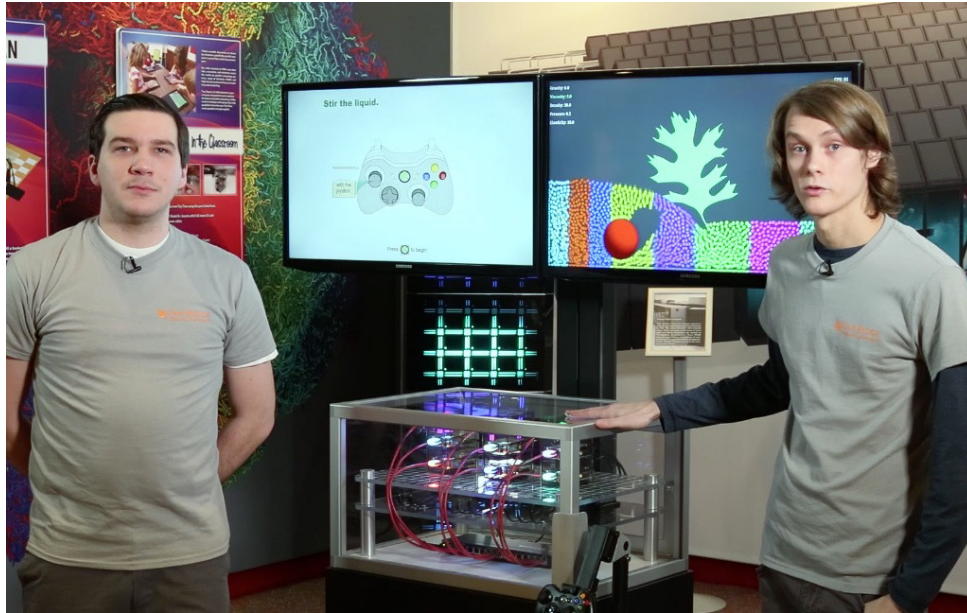
NOVEMBER 2020	SYSTEM	SPECS	SITE	COUNTRY	CORES	RMAX PFLOP/S	POWER MW
1	Fugaku 158,976 nodes	Fujitsu A64FX (48C, 2.2GHz), Tofu Interconnect D	RIKEN R-CCS	Japan	7,630,848	442.0	29.9
2	Summit 4,608 nodes	IBM POWER9 (22C, 3.07GHz), NVIDIA Volta GV100 (80C), Dual-Rail Mellanox EDR Infiniband	DOE/SC/ORNL	USA	2,414,592	148.6	10.1
3	Sierra 4,320 nodes	IBM POWER9 (22C, 3.1GHz), NVIDIA Tesla V100 (80C), Dual-Rail Mellanox EDR Infiniband	DOE/NNSA/LLNL	USA	1,572,480	94.6	7.44
4	Sunway TaihuLight	Shenwei SW26010 (260C, 1.45 GHz) Custom Interconnect	NSCC in Wuxi	China	10,649,600	93.0	15.4
5	Selene	NVIDIA DGX A100, AMD EPYC 7742 (64C, 2.25GHz), NVIDIA A100, Mellanox HDR Infiniband	NVIDIA Corporation	USA	555,520	63.4	2.65

Languages and libraries for parallel computing

- MPI for distributed-memory parallelism (runs everywhere except GPUs)
- Multithreading or “shared memory parallelism”
 - Directive-based OpenMP (deceptively easy) www.openmp.org (!\$OMP DO)
 - POSIX pthread programming (explicit parallelism, somewhat harder than MPI since one needs to manage threads access to memory).
 - GPGPU (General-Purpose Graphical Processing Unit) programming with **CUDA** (nvidia), **OpenACC** or **OpenMP**
- PGAS global address space SPMD languages (using GASNet layer or other)
 - Efficient single-sided communication on globally-addressable memory
 - FORTRAN 2008 co-arrays
 - UPC (<http://upc.lbl.gov/>): Similar to co-array Fortran but for C.

MPI works on all parallel systems

Even on **Tiny-Titan!** 9 Raspberry Pis connected together



<https://www.olcf.ornl.gov/2014/06/02/titans-tiny-counterpart-engages-educates/>

Reason to use MPI: Scalability and portability

Distributed memory parallel computers (inter-node parallelism)

- Each (operating system) process has its own virtual memory and cannot access the memory of other processes
- A copy of the same executable runs on each MPI process (processor core)
- Any data to be shared must be explicitly transmitted from one to another

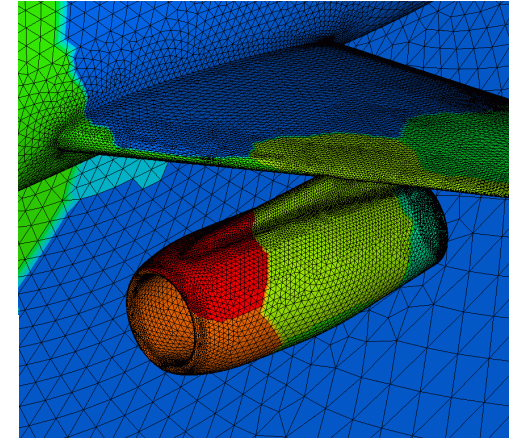
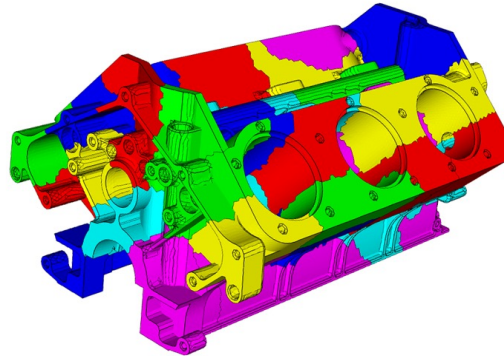
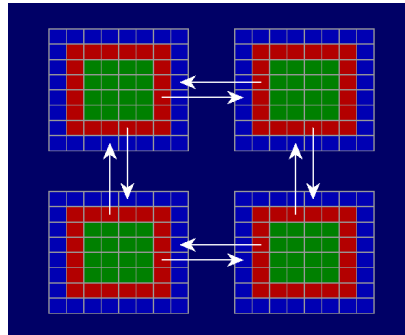
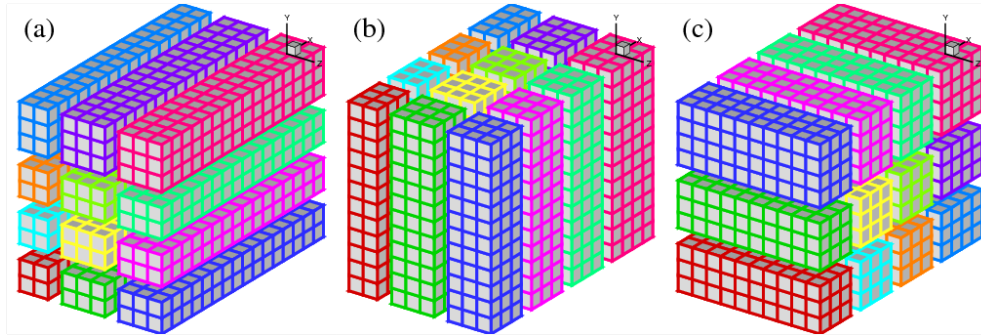
Most message passing programs use the *single program multiple data (SPMD) model*

- Each process executes the same set of instructions asynchronously
- Parallelization is achieved by letting each processor core operate on a different piece of data
- Not to be confused with SIMD: Single Instruction Multiple Data *a.k.a vector computing*

How to split the work between processors?

Domain Decomposition

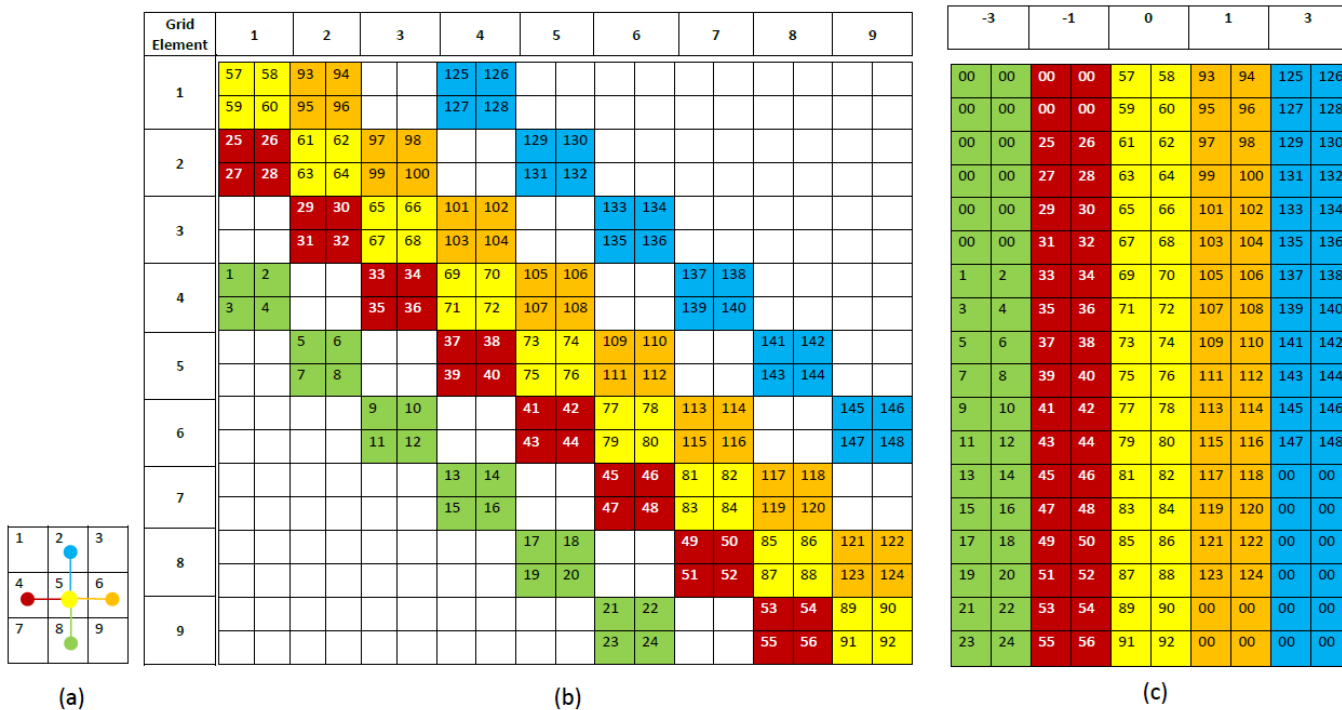
- Most widely used method for grid-based calculations



How to split the work between processors?

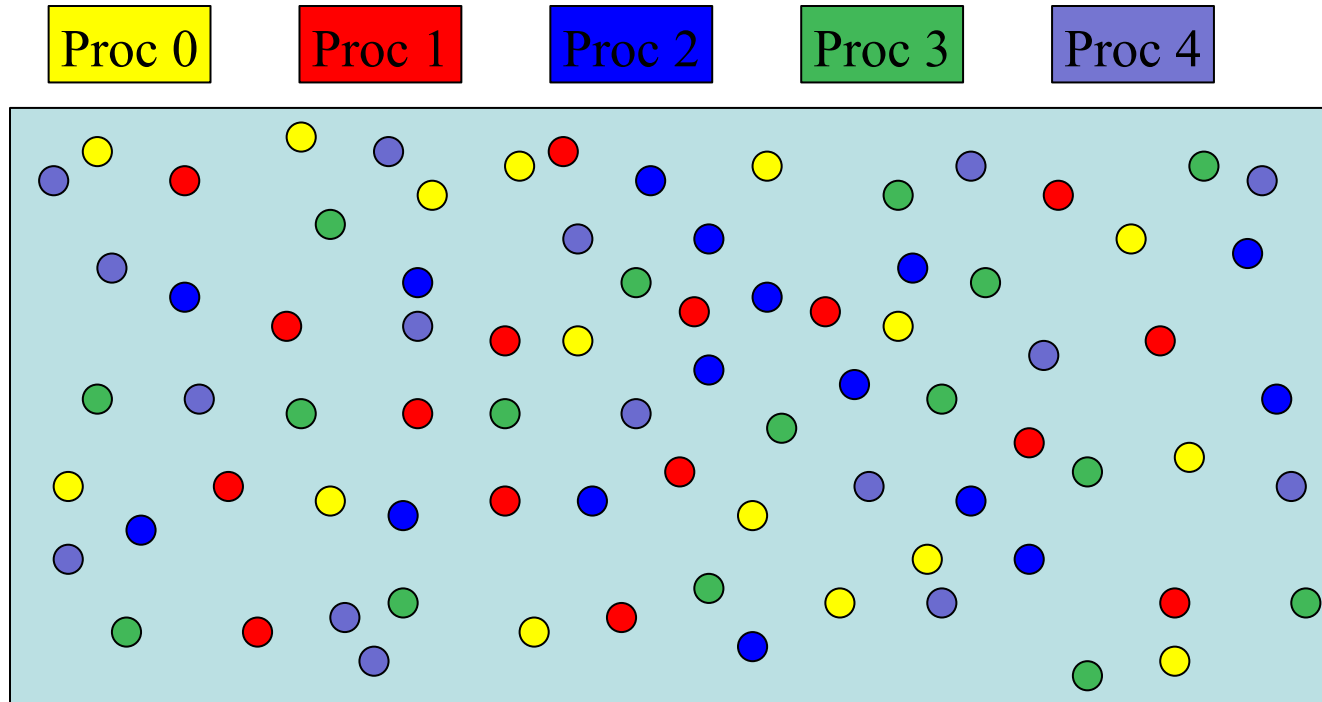
Split matrix elements in PDE solves

- See PETSc project: <https://www.mcs.anl.gov/petsc/>



How to split the work between processors? “Coloring”

- Useful for particle simulations (Particle-in-Cell, MD)



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 set of functions (C) and subroutines (Fortran) used for exchanging data between processes.
- An MPI library exists on ALL parallel computing platforms so it is highly portable.
- The scalability of MPI is not limited by the number of processors/cores on one computation node, as opposed to shared memory parallel models.
- Also available for Python (mpi4py.scipy.org), R (Rmpi), Lua, and Julia! (if you can call C functions, you can use MPI...)

MPI standard

- MPI standard is a specification of what MPI is and how it should behave. Vendors have some flexibility in the implementation (e.g. buffering, collectives, topology optimizations, etc.).
- This tutorial focuses on the functionality introduced in the original MPI-1 standard
- MPI-2 standard introduced additional support for
 - Parallel I/O (many processes writing to a single file). Requires a parallel filesystem to be efficient
 - One-sided communication: `MPI_Put`, `MPI_Get`
 - Dynamic Process Management
- MPI-3 standard starting to be implemented by compilers vendors
 - Non-blocking collectives
 - Improved one-sided communications
 - Improved Fortran bindings for type check
 - And more (see <http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf>)

How much do I need to know?

- MPI has about 400 functions/subroutines
- You can do **everything** with about **6 functions** although your code will be complex and hard to read
- Collective functions, which involve communication between several MPI processes, are **EXTREMELY** useful since they simplify the coding, and vendors optimize them for best performance on their interconnect hardware
- One can access flexibility when required.
- No need to master all parts of MPI to use it successfully
- **The way you split the work in your program is more important!!**

Compiling and linking an MPI code

- First things first: load your favorite compiler module and MPI on adroit
 - **module load intel/2021.1.2 intel-mpi/intel/2021.3.1**(or openmpi)
 - **module load openmpi/gcc/4.1.0** (uses the OS gcc and gfortran)
- Need to tell the compiler where to find the MPI include files and how to link to the MPI libraries.
- Fortunately, all MPI implementations come with scripts that take care of these issues:
 - **mpicc mpi_code.c -o a.out**
 - **mpiCC mpi_code_C++.C -o a.out**
 - **mpif90 mpi_code.f90 -o a.out**
- Use “**mpicc -show**” to display the actual compile line

Makefile

- Always a good idea to have a Makefile
- Here is a very simple one:

```
%cat Makefile
```

```
CC=mpicc
```

```
CFLAGS=-O
```

```
% : %.c
```

```
    $(CC) $(CFLAGS) $< -o $@
```

How to run an MPI executable

- The implementation supplies scripts to launch the MPI parallel calculation, for example:

```
mpirun -np #proc a.out  
mpiexec -n #proc a.out  
srun -n #proc a.out (SLURM batch system, Princeton systems)
```

} MPICH, OPENMPI

- A copy of the same program runs on each processor core within its own process (private address space).
- Each process works on a subset of the problem.
- Exchange data when needed
 - Can be exchanged through the network interconnect
 - Or through the shared memory on SMP machines (Bus?)
- Easy to do coarse grain parallelism = scalable

SLURM Batch System

- Submit a job script: `sbatch script`
- Check status of jobs: `squeue -a` (for all jobs)
- Stop a job: `scancel job_id`

```
#!/bin/bash
# parallel job using 16 processors. and runs for 4 hours (max)
#SBATCH -N 2 # node count
#SBATCH --ntasks-per-node=8
#SBATCH -t 4:00:00
# sends mail when process begins, and
# when it ends. Make sure you define your email
#SBATCH --mail-type=begin
#SBATCH --mail-type=end
#SBATCH --mail-user=yourNetID@princeton.edu
module load openmpi/gcc/4.1.0
srun ./a.out
```


Example code: calculating π using numerical integration (C version)

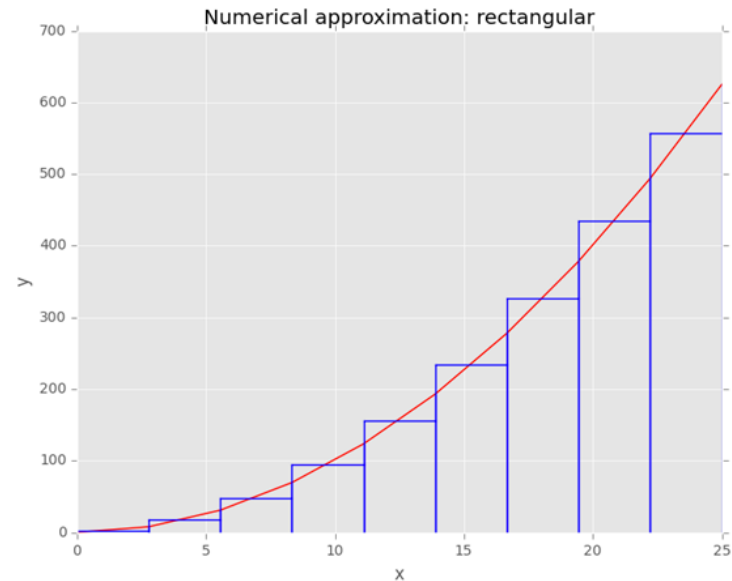
```
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    FILE *ifp;

    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;
}
```

$$\int_0^1 \frac{dx}{1+x^2} = \frac{\pi}{4}$$



Example code: calculating π using numerical integration (Fortran version)

```
program fpi
  double precision  PI25DT
  parameter          (PI25DT = 3.141592653589793238462643d0)
  double precision  mypi, pi, h, sum, x, f, a
  integer n, myid, numprocs, i, j, ierr

  open(12,file='nslices.in',status='old')
  read(12,*) n
  close(12)
  write(*,*)' number of intervals=',n
c
  h = 1.0d0/n
  sum = 0.0d0
  do i = 1, n
    x = h * (dble(i) - 0.5d0)
    sum = sum + 4.d0/(1.d0 + x*x)
  enddo
  mypi = h * sum
c
  pi = mypi
  write(*,*)' pi=',pi,' Error=',abs(pi - PI25DT)

end
```

Hands-on exercise #1

1. Log into adroit: `ssh -X username@adroit.princeton.edu`
2. `module load intel intel-mpi`
3. Copy files from my directory:
`cp -r /home/ethier/ Fall_Break_2022 .` (don't forget the dot)
4. “cd” into `Intro_MPI/C` or `Fortran`
5. Examine the “`Makefile`” and “`slurm_script`”
6. Examine the first example “`cpi_1.c`” or “`fpi_1.c`”
7. Build the example: `make cpi_1` (make `fpi_1`)
8. Run the example: `./cpi_exe` or `./fpi_exe`
9. Run it again via the slurm script: `sbatch slurm_script`
10. Look in the file `output.log`. What's the difference?

MPI Communicators

- A communicator is an identifier associated with a group of processes
 - Each process has a unique rank within a specific communicator (the rank starts from 0 and has a maximum value of $(n\text{processes}-1)$).
 - Internal mapping of processes to processing units
 - Always required 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

```
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
```

Header file

```
Program mpi_code
! Load MPI definitions
use mpi

! 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
```

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

Initialization

```
Program mpi_code
```

```
! Load MPI definitions
```

```
use mpi
```

```
! Initialize MPI
```

```
call MPI_Init(ierr)
```

```
! Get the number of processes
```

```
call MPI_Comm_size(MPI_COMM_WORLD, nprocs, 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
```

- 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
```

```
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
```


What is my rank?

```
Program mpi_code
```

```
! Load MPI definitions
```

- 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

```
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
```

Termination

```
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)
```

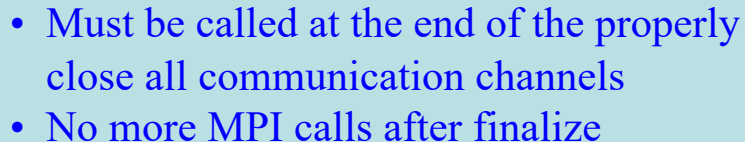
```
  call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
```

```
  Do work and make message passing calls...
```

```
! Finalize
```

```
  call MPI_Finalize(ierr)
```

```
end program mpi_code
```

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

A sample MPI program in C

```
#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;
}
```

Hands-on exercise #2

1. Add the necessary MPI calls to the first exercise code
2. Build your new code: `make cpi_1` (`make fpi_1`)
 - The answer is in `cpi_2.c` and `fpi_2.f` if you run out of time...
3. Run it via the slurm script: `sbatch slurm_script`
4. Look in `output.log`. Is there a difference?

Hands-on exercise #3

1. Now you need to use the MPI task id “`myid`” and the number of MPI tasks “`numprocs`” to split the work between the tasks. Change the `for` or `do` loop accordingly...
2. Build your new code: `make cpi_1` (`make fpi_1`)
 - The answer is in `cpi_3.c` and `fpi_3.f` if you run out of time...
3. Run it via the slurm script: `sbatch slurm_script`
4. Look in `output.log`. What do you observe?

THE TASKS NEED TO COMMUNICATE!

Basic 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 3 types of collective communications
 - Data movement (e.g. MPI_Bcast)
 - Reduction (e.g. MPI_Reduce)
 - Synchronization: MPI_Barrier

Point-to-point communication

Point to point: 2 processes at a time FORTRAN add-ons in RED

“buf” is a
pointer!!

```
MPI_Send(buf, count, datatype, dest, tag, comm, ierr)
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierr)
```

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

where the 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

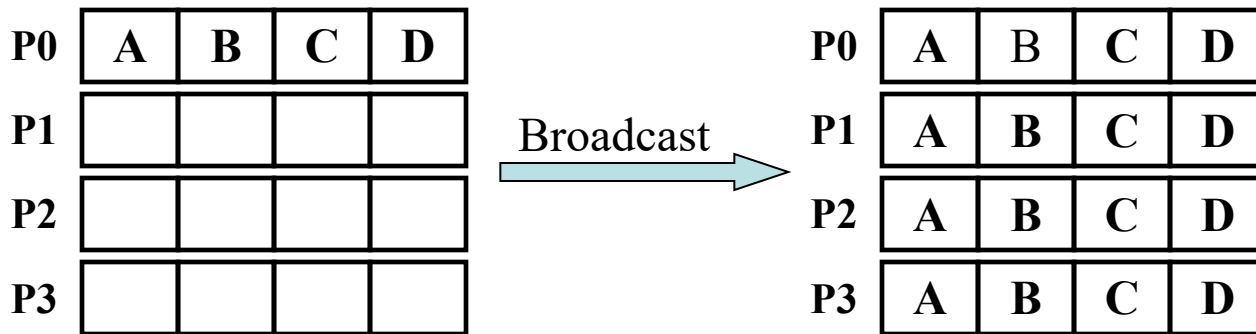
MPI_PROC_NULL

- Can be used as “source” or “destination” in MPI_Send or MPI_Recv (and MPI_Sendrecv)
- Identical behavior as:

```
if (source .ne. MPI_PROC_NULL) then
    call MPI_SEND(..., source, ...)
endif
```


Collective communication: Broadcast

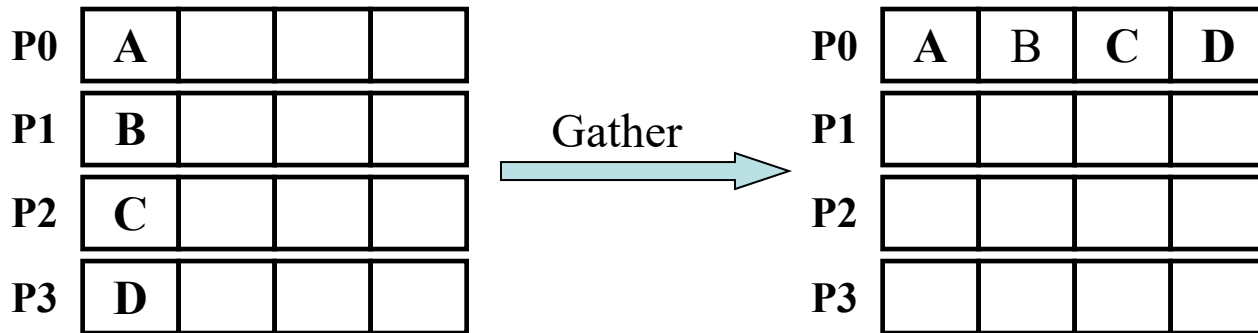
`MPI_Bcast (buffer, count, datatype, root, comm, ierr)`



- 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

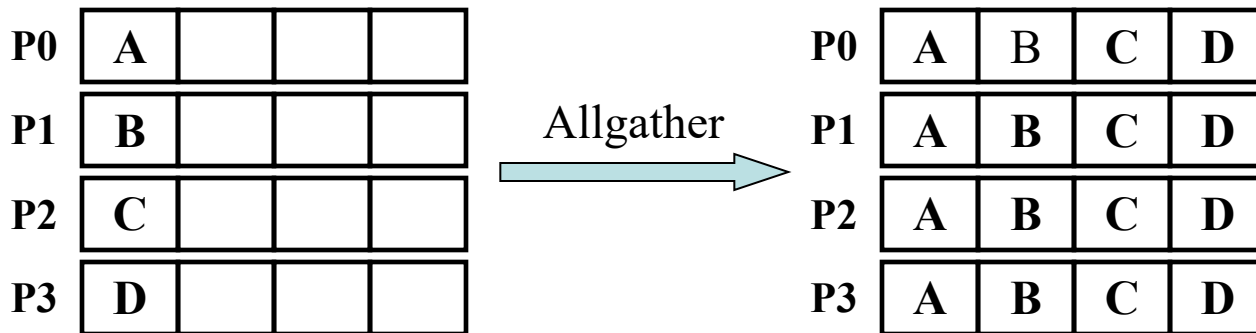
`MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, ierr)`



- 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

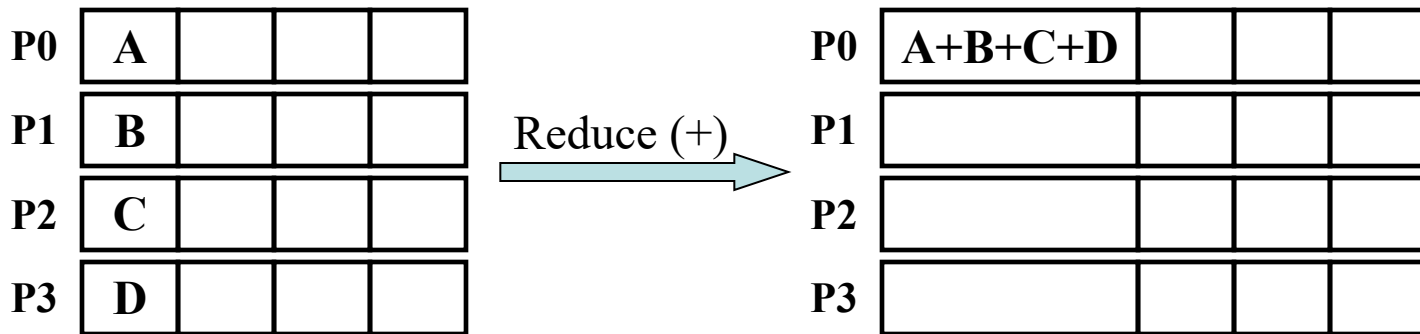
`MPI_Allgather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierr)`



- 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

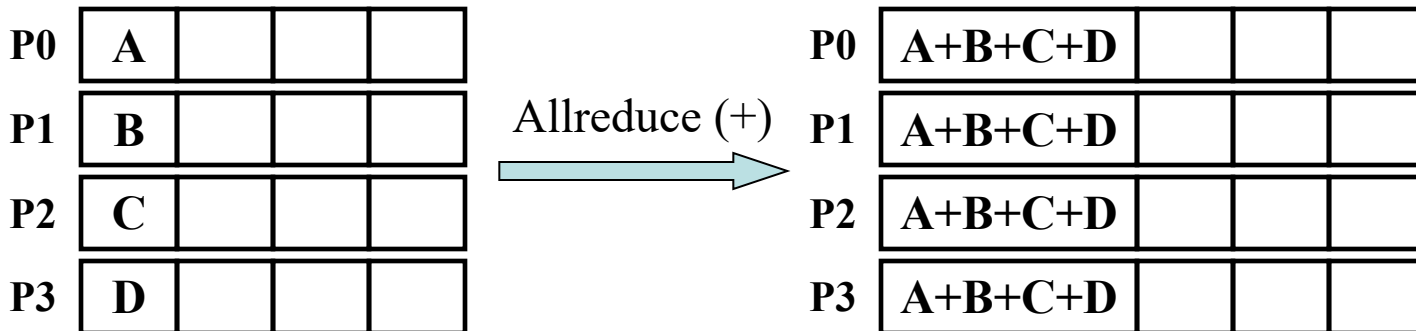
`MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm, ierr)`



- 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
```

Hands-on exercise #4

1. Add the necessary MPI call(s) to get the portions of pi and add them together to get the final (correct) value
2. The answer is in `cpi_4a.c` and `fpi_4a.f` OR `cpi_4b.c` and `fpi_4b.f`
3. Run it via the slurm script: `sbatch slurm_script`
4. Look in `output.log`. Do you get the right answer? Can you think of another MPI call to do this?

Hands-on exercise #5

Let's say that the input file [nslices.dat](#) is very large and that you are using thousands of MPI tasks for your compute intensive code. You probably would not want all the tasks to access this file at the same time since accessing the filesystem is the slowest communication (I/O) operation there is. Do the following:

1. Add code so that only the root process (myid=0) reads the file
2. Add the proper MPI function call so that the root process communicates the content of the file to all the other tasks
3. The answer is in [cpi_5.c](#) and [fpi_5.f](#)

```
#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);
    }
    /* 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;

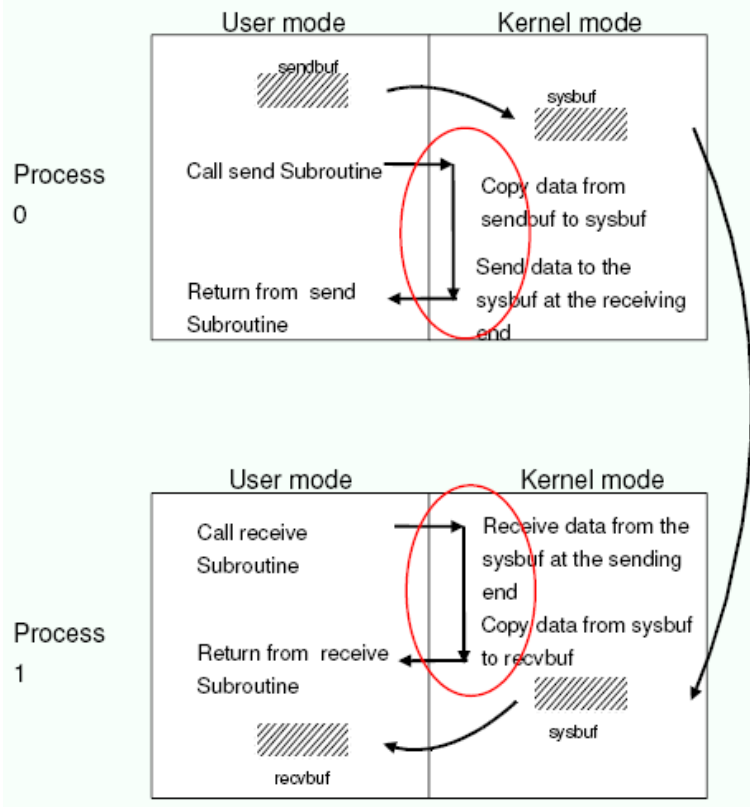
pi = 0.; /* It is not necessary to set pi = 0 */

/* 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));

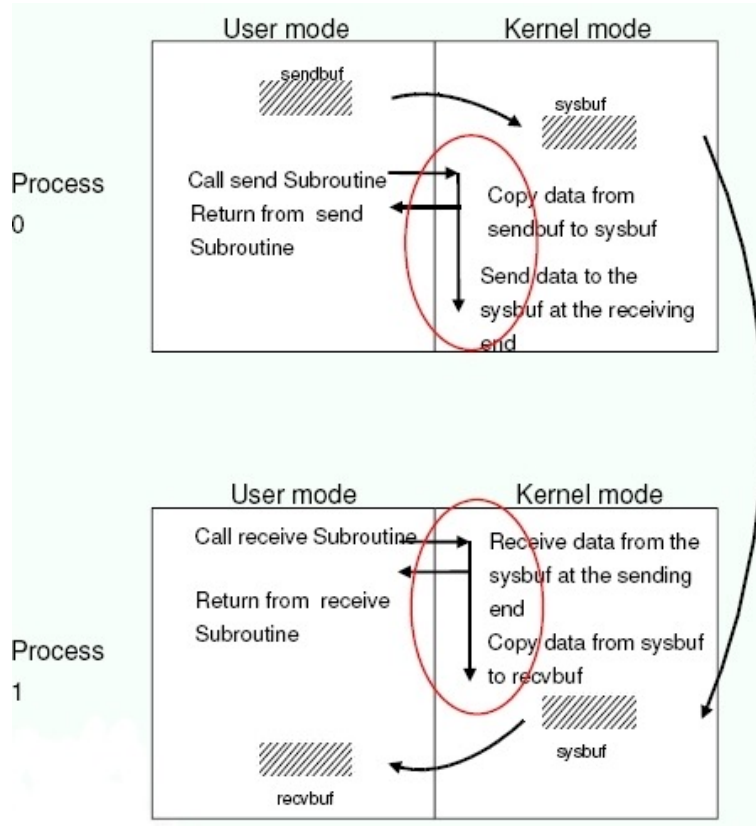
MPI_Finalize();
return 0;
}
```

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

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”](#)

Forced synchronization

```
C:          int MPI_Barrier( MPI_Comm comm )
```

```
Fortran:   call MPI_Barrier( comm, ierr )
```

Blocks until all processes in the **communicator have reached this routine**

- There is an implicit barrier for all blocking collective calls
- MPI_Barrier is sometimes necessary to synchronize processes
- Needed when timing sections of your code
- Frequent synchronizations will slow down your code significantly. **Use barriers sparingly**

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)
```

To detect a “NaN” (not a number):

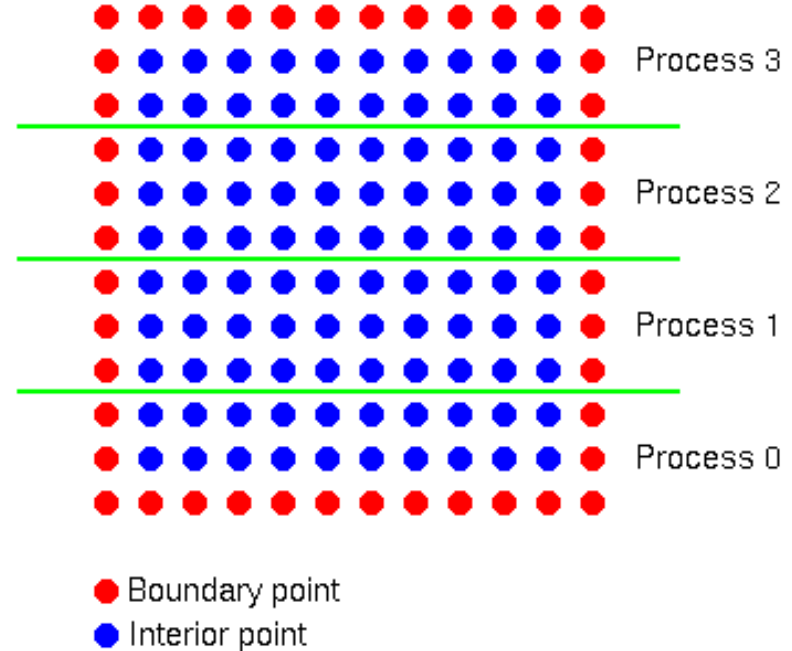
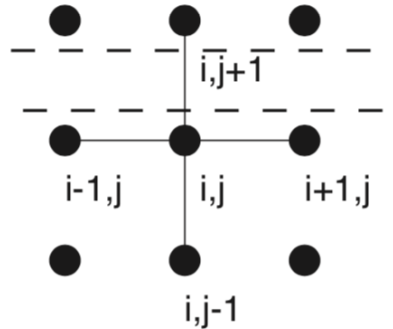
```
C:      if (isnan(var))  
Fortran: if (var /= var)
```

Use a parallel debugger such as **DDT** or Totalview (if available)

Domain decomposition example

Jacobi solver

```
while (not converged) {  
  for (i,j)  
    xnew[i][j]= (x[i+1][j] + x[i-1][j]  
                + x[i][j+1] + x[i][j-1])/4;  
  for (i,j)  
    x[i][j] = xnew[i][j];  
}
```

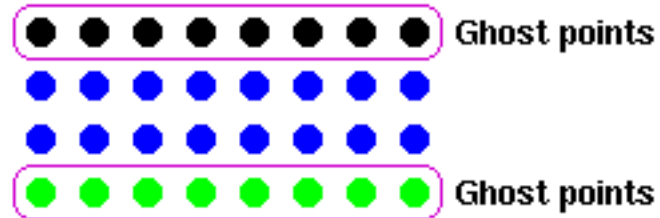


Setting up “ghost” cells

X, showing decomposition
by color



X_{local} for Blue processor



Hands-on exercise #6

Have a look at the file **jacobi_MPI.c**

1. Compile it with `make jacobi_MPI` (executable still named `cpi_exe`)
 2. Run it via the slurm script: `sbatch slurm_script`
 3. Look at `output.log`. Why does it work?
- Can you replace `MPI_Send` and `MPI_Recv` with `MPI_Sendrecv`?
 - Hint: you will need `MPI_PROC_NULL`
 - Replace blocking send/recv with non-blocking
 - When going 3D, easier to use `MPI_Cart_Create`

References

- Just google “mpi tutorial”, or “mpi documentation”, or “mpi standard”...
- <https://computing.llnl.gov/tutorials/mpi/>
- <http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html> (old but still relevant)
- <http://www.mpi-forum.org> (location of the MPI standard)

- MPI on Linux clusters:
 - MPICH (<https://www.mpich.org/>)
 - Open MPI (<http://www.open-mpi.org/>)
- Books:
 - Using MPI “Portable Parallel Programming with the Message-Passing Interface” by William Gropp, Ewing Lusk, and Anthony Skjellum
 - Using MPI-2 “Advanced Features of the Message-Passing Interface”

Works with Python too!

- <http://mpi4py.scipy.org/docs/usrman/tutorial.html>
- `mpirun -np 4 python script.py`

Script.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

- Uses “pickle” module to get access to C-type contiguous memory buffer
- Evolving rapidly
- On adroit.princeton.edu:
 - **module load openmpi/gcc**
 - **module load conda3**
 - **pip install --user mpi4py**

```
from mpi4py import MPI
import numpy
import time

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

N = numpy.arange(1, dtype=numpy.intc)
if rank == 0:
    N[0] = 1000*1000*100
comm.Bcast([N, 1, MPI.INT], root=0)

start = time.time()

h = 1.0 / N[0]; s = 0.0
for i in range(rank, N[0], size):
    x = h * (i + 0.5)
    s += 4.0 / (1.0 + x**2)
PI = numpy.array(s * h, dtype='d')
PI_sum = numpy.array(0.0, dtype='d')
#comm.Reduce([PI, MPI.DOUBLE], PI_sum, op=MPI.SUM, root=0)
comm.Allreduce([PI, MPI.DOUBLE], PI_sum, op=MPI.SUM)

end = time.time()
print("rank:%d Pi with %d steps is %15.14f in %f secs" %(rank, N[0], PI_sum, end-start))
```

our PI calculation example

Mixing MPI and OpenMP
together in the same application

Why use both MPI and OpenMP in the same code?

- To save memory by not having to replicate data common to all processes, not using ghost cells, sharing arrays, etc.
- To optimize interconnect bandwidth usage by having only one MPI process per NUMA node.
- Although MPI generally scales very well it has its limit, and OpenMP gives another avenue of parallelism.
- Some compilers have now implemented OpenMP-like directives to run sections of a code on general-purpose GPU (GPGPU). Fine-grain parallelism with OpenMP directives is easy to port to GPUs.

Implementing mixed-model

- Easiest and safest way:
 - Coarse grain MPI with fine grain loop-level OpenMP
 - All MPI calls are done outside the parallel regions
 - This is always supported
- Allowing the master thread to make MPI calls inside a parallel region
 - Supported by most if not all MPI implementations
- Allowing ALL threads to make MPI calls inside the parallel regions
 - Requires MPI to be fully thread safe
 - Not the case for all implementations
 - Can be tricky...

Find out the level of support of your MPI library

MPI-2 “Init” functions for multi-threaded MPI processes:

```
int MPI_Init_thread(int * argc, char ** argv[], int thread_level_require,
                   int * thread_level_provided);
int MPI_Query_thread(int * thread_level_provided);
int MPI_Is_main_thread(int * flag);
```

- “Required” values can be:
 - **MPI_THREAD_SINGLE**: Only one thread will execute
 - **MPI_THREAD_FUNNELED**: Only master thread will make MPI-calls
 - **MPI_THREAD_SERIALIZED**: Multiple threads may make MPI-calls, but only one at a time
 - **MPI_THREAD_MULTIPLE**: Multiple threads may call MPI, without restrictions
- “Provided” returned value can be less than “required” value

Compiling and linking mixed code

- Just add the “openmp” compiler option to the compile AND link lines (if separate from each other):
 - `mpicc -qopenmp mpi_omp_code.c -o a.out` (for Intel compiler)
 - `mpif90 -qopenmp mpi_omp_code.f90 -o a.out`
- Dfg
- To run a MPI+OpenMP job, make sure that your SLURM script asks for the total number of threads that you will use in your simulation, which should be **(total number of MPI tasks)*(number of threads per task)**
`#SBATCH --cpus-per-task=${OMP_NUM_THREADS}`
`#SBATCH --ntasks-per-node=(#cores per node/${OMP_NUM_THREADS})`

Thank you for attending...

Happy parallel programming!