Message Passing Programming with MPI

- Introduction to MPI
- Basic MPI functions
- Most of the MPI materials are obtained from William Gropp and Rusty Lusk's MPI tutorial at http://www.mcs.anl.gov/**mpi**/tutorial/

Message Passing Interface (MPI)

- MPI is an industrial standard that specifies library routines needed for writing message passing programs.
  - Mainly communication routines
  - Also include other features such as topology.
- MPI allows the development of scalable portable message passing programs.
  - It is a standard supported pretty much by everybody in the field.

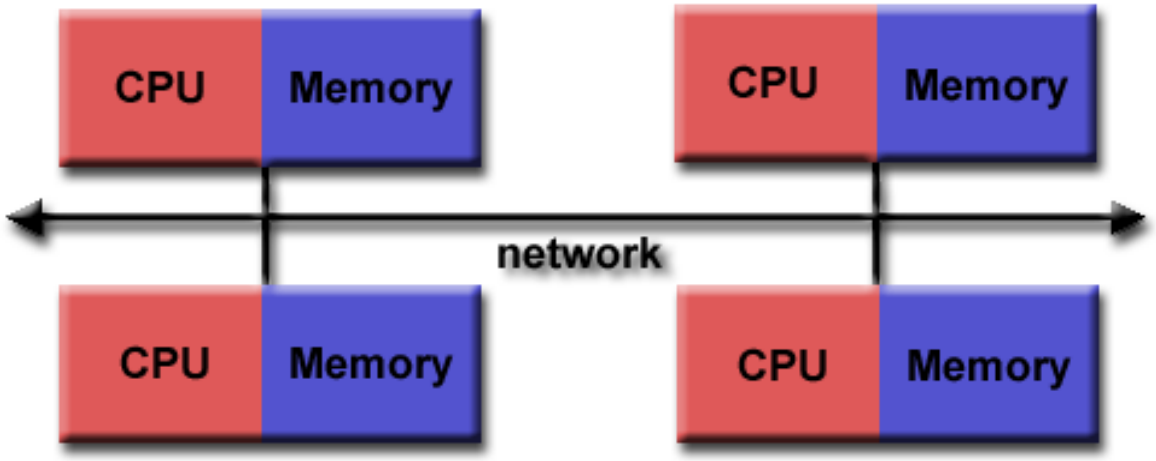
- MPI uses a library approach to support parallel programming.
  - MPI specifies the API for message passing (communication related routines)
  - MPI program = C/Fortran program + MPI communication calls.
  - MPI programs are compiled with a regular compiler(e.g gcc) and linked with an mpi library.

MPI execution model

- The world of parallel multiple instruction, multiple data, or MIMD, computers is, for the most part, divided into **distributed-memory** and **shared-memory** systems.
- From a programmer's point of view, a distributed-memory system consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core.

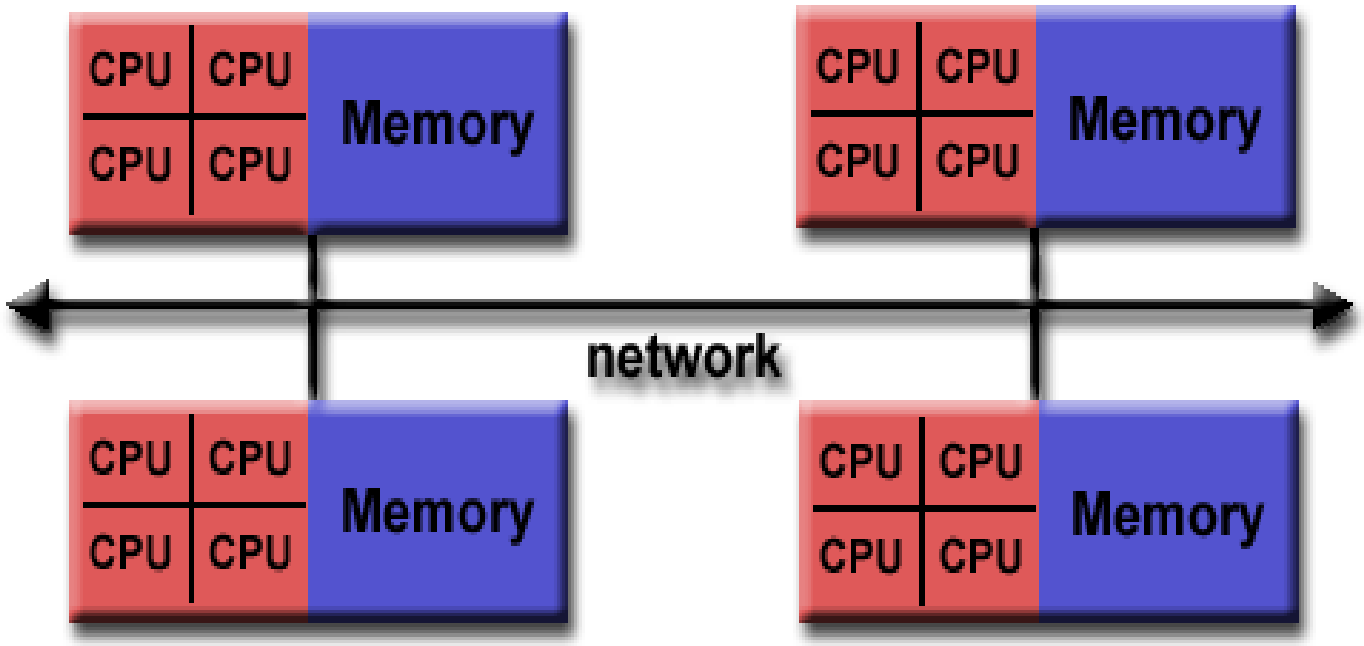
MPI execution model

MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).



MPI execution model

MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.



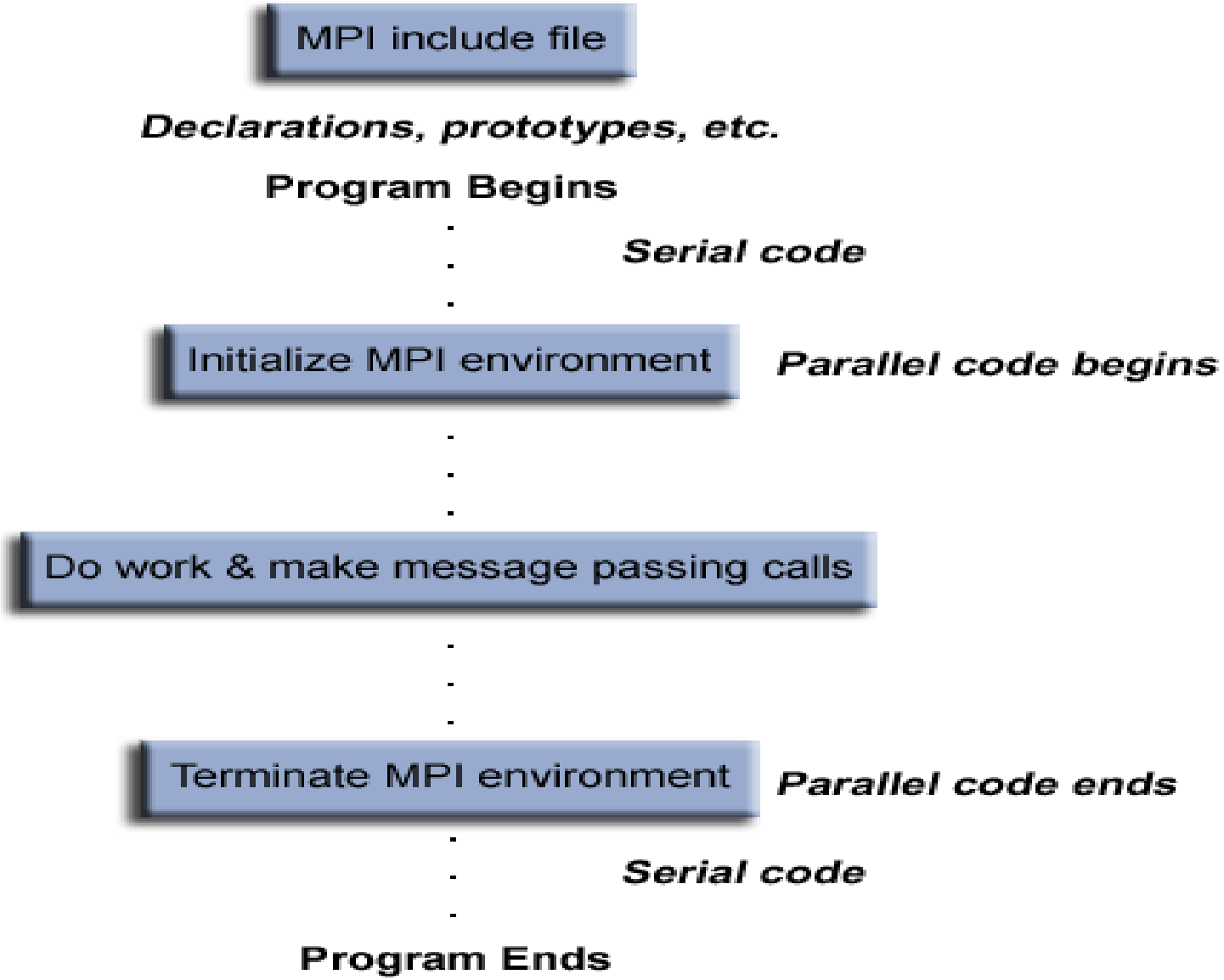
MPI execution model

Today, MPI runs on virtually any hardware platform:

- Distributed Memory
- Shared Memory
- Hybrid

The programming model clearly remains a distributed memory model however, regardless of the underlying physical architecture of the machine.

MPI Program Structure



MPI Program Structre

Required for all programs that make MPI library calls.

C include file	**Fortran include file**
#include "mpi.h"	include 'mpif.h'

Environment Management Routines

This group of routines is used for interrogating and setting the MPI execution environment, and covers an assortment of purposes, such as

- initializing and terminating the MPI environment,
- querying a rank's identity,
- querying the MPI library's version, etc.

Most of the commonly used ones are described below.

Environment Management Routines : MPI_Init

Initializes the MPI execution environment.

This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

MPI_Init (&argc,&argv)

MPI_INIT (ierr)

Environment Management Routines : MPI_Comm_size

Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD.

If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

MPI_Comm_size (comm,&size)

MPI_COMM_SIZE (comm,size,ierr)

Environment Management Routines : MPI_Comm_rank

Returns the rank of the calling MPI process within the specified communicator.

Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD.

This rank is often referred to as a task ID.

If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

MPI_Comm_rank (comm,&rank)

MPI_COMM_RANK (comm,rank,ierr)

Environment Management Routines : MPI_Abort

Terminates all MPI processes associated with the communicator.

In most MPI implementations it terminates ALL processes regardless of the communicator specified.

MPI_Abort (comm,errorcode)

MPI_ABORT (comm,errorcode,ierr)

Environment Management Routines : MPI_Get_processor_name

Returns the processor name.

Also returns the length of the name.

The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size.

What is returned into "name" is implementation dependent - may not be the same as the output of the "hostname" or "host" shell commands.

MPI_Get_processor_name (&name,&resultlength)

MPI_GET_PROCESSOR_NAME (name,resultlength,ierr)

Environment Management Routines : MPI_Get_version

Returns the version and subversion of the MPI standard that's implemented by the library.

MPI_Get_version (&version,&subversion)

MPI_GET_VERSION (version,subversion,ierr)

Environment Management Routines : MPI_Initialized

Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0).

MPI requires that MPI_Init be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call MPI_Init if necessary.

MPI_Initialized solves this problem.

MPI_Initialized (&flag)

MPI_INITIALIZED (flag,ierr)

Environment Management Routines : MPI_Wtime

Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

MPI_Wtime ()

MPI_WTIME ()

Environment Management Routines : MPI_Wtick

Returns the resolution in seconds (double precision) of MPI_Wtime.

MPI_Wtick ()

MPI_WTICK ()

Environment Management Routines : MPI_Finalize

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

MPI_Finalize ()

MPI_FINALIZE (ierr)

Environment Management Routines

```
// required MPI include file
   #include "mpi.h"
   #include <stdio.h>

   int main(int argc, char *argv[]) {
   int  numtasks, rank, len, rc;
   char hostname[MPI_MAX_PROCESSOR_NAME];

   // initialize MPI
   MPI_Init(&argc,&argv);
```

Environment Management Routines

```
// get number of tasks
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
// get my rank
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
// this one is obvious
MPI_Get_processor_name(hostname, &len);
printf ("Number of tasks= %d My rank= %d Running on
%s\n", numtasks,rank,hostname);

    // do some work with message passing
// done with MPI
MPI_Finalize();
}
```

Environment Management Routines : For Compilation

```
$ mpicc -g -Wall -o mpi hello mpi file1.c
```

Typically, mpicc is a script that's a wrapper for the C compiler. A wrapper script is a script whose main purpose is to run some program.

In this case the program is the C compiler.

However, the wrapper simplifies the running of the compiler by telling it where to find the necessary header files and which libraries to link with the object file.

Environment Management Routines : For Compilation

Many systems also support program startup with mpiexec:

$ mpiexec -n <number of processes> ./mpi file1

So to run the program with one process,

$ mpiexec -n 1 ./mpi file1

and to run the program with 4 processes,

$ mpiexec -n 4 ./mpi file1

MPI execution model

- Separate (collaborative) processes are running all the time.
  - 'mpirun –machinefile machines –np 16 a.out' → The same a.out is executed on 16 machines.
  - Different from the OpenMP model.
    - What about the sequential portion of an application?

MPI data model

- No shared memory. Using explicit communications whenever necessary.
- How to solve large problems
  - Logically partition the large array and logically distribute the large array into processes.

- MPI specification is both simple and complex.
  - Almost all MPI programs can be realized with six MPI routines.
  - MPI has a total of more than 100 functions and a lot of concepts.
  - We will mainly discuss the simple MPI, but we will also give a glimpse of the complex MPI.
- MPI is about just the right size.
  - One has the flexibility when it is required.
  - One can start using it after learning the six routines.

The hello world MPI program

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
     printf( "Hello world\n" );
     MPI_Finalize();
     return 0;
}
```

- Mpi.h contains MPI definitioins and types.
- MPI program must start with MPI_init
- MPI program must exit with MPI_Finalize
- MPI functions are just library routines that can be used on top of the regular C, C++, Fortran language constructs.

Compiling, linking and running MPI programs

- MPICH is installed on linprog
- To run a MPI program, do the following:
  - Create a file called .mpd.conf in your home directory with content 'secretword=cluster'
  - Create a file 'hosts' specifying the machines to be used to run MPI programs.
  - Boot the system 'mpdboot –n 3 –f hosts'
  - Check if the system is corrected setup: 'mpdtrace'
  - Compile the program: 'mpicc hello.c'
  - Run the program: mpiexec –machinefile hostmap –n 4 a.out
    - Hostmap specifies the mapping
    - -n 4 says running the program with 4 processes.
  - Exit MPI: mpdallexit

Login without typing password

- Key based authentication
  - Password based authentication is inconvenient at times
    - Remote system management
    - Starting a remote program (starting many MPI processes!)
    - ……
  - Key based authentication allows login without typing the password.
- Key based authentication with ssh in UNIX
  - Remote ssh from machine A to machine B

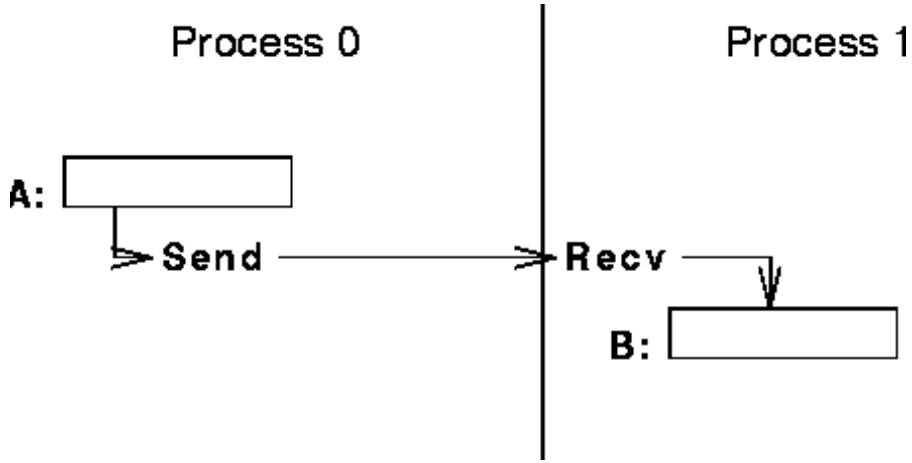
    Step 1: at machine A: ssh-keygen –t rsa

    (do not enter any pass phrase, just keep typing “enter”)

    Step 2: append A:.ssh/id_rsa.pub to B:.ssh/authorized_keys

- MPI uses the SPMD model (one copy of a.out).
  - How to make different process do different things (MIMD functionality)?
    - Need to know the execution environment: Can usually decide what to do based on the number of processes on this job and the process id.
      - How many processes are working on this problem?
        - » MPI_Comm_size
      - What is myid?
        - » MPI_Comm_rank
        - » Rank is with respect to a communicator (context of the communication). MPI_COM_WORLD is a predefined communicator that includes all processes (already mapped to processors).

Sending and receiving messages in MPI



- Questions to be answered:
  - To who are the data sent?
  - What is sent?
  - How does the receiver identify the message

- Send and receive routines in MPI
  - MPI_Send and MPI_Recv (blocking send/recv)
  - Identify peer: Peer rank (peer id)
  - Specify data: Starting address, datatype, and count.
    - An MPI datatype is recursively defined as:
      - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
      - a contiguous array of MPI datatypes
      - a strided block of datatypes
      - an indexed array of blocks of datatypes
      - an arbitrary structure of datatypes
      - There are MPI functions to construct custom datatypes, in particular ones for subarrays
  - Identifying message: sender id + tag

MPI blocking send

MPI_Send(start, count, datatype, dest, tag, comm)

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI blocking receive

MPI_Recv(start, count, datatype, source, tag, comm, status)

- Waits until a matching (both **`source`** and **`tag`**) message is received from the system, and the buffer can be used
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE (a message from anyone)`**
- **`tag`** is a tag to be matched on or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error (result undefined)
- **`status`** contains further information (e.g. size of message, rank of the source)
- See pi_mpi.c and jacobi_mpi.c for the use of MPI_Send and MPI_Recv.

- The Simple MPI (six functions that make most of programs work):
  - **MPI_INIT**
  - **MPI_FINALIZE**
  - **MPI_COMM_SIZE**
  - **MPI_COMM_RANK**
  - **MPI_SEND**
  - **MPI_RECV**
  - Only MPI_Send and MPI_Recv are non-trivial.

The MPI PI program

$$\frac{1}{n}\sum_{i=1}^{n}\frac{4.0}{1+\left(1+\frac{i-0.5}{n}*\frac{i-0.5}{n}\right)}¿$$

```
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;
```

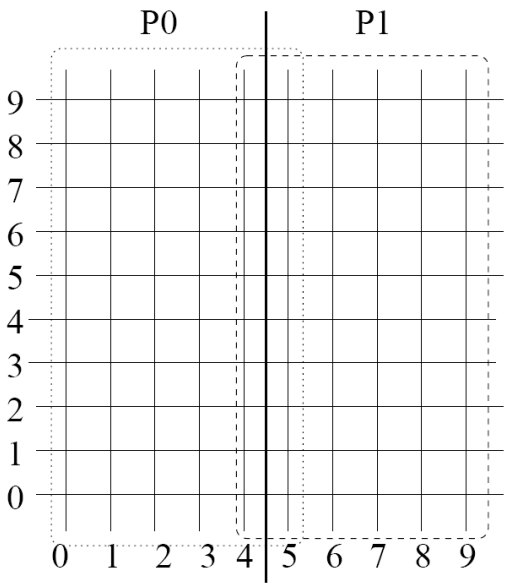
```
h   = 1.0 / (double) n;  sum = 0.0;
  for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  mypi = h * sum;

  if (myid == 0) {
    for (i=1; i<numprocs; i++) {
       MPI_Recv(&tmp, 1, MPI_DOUBLE, i, 0,  MPI_COMM_WORLD,
       &status);
       mypi += tmp;
    }
  } else MPI_Send(&mypi, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
/* see pi_mpi.c */
```

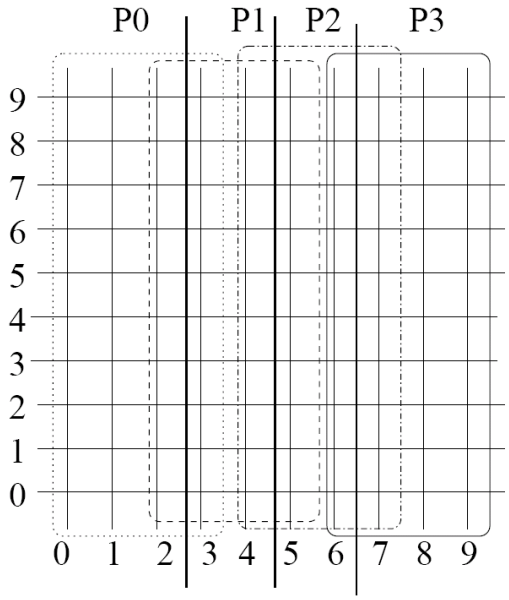
SOR: sequential version

```
for some number of timesteps/iterations {
  for (i=0; i<n; i++ )
        for( j=1, j<n, j++ )
              temp[i][j] = 0.25 *
                    ( grid[i-1][j] + grid[i+1][j]
                    grid[i][j-1] + grid[i][j+1] );
  for( i=0; i<n; i++ )
        for( j=1; j<n; j++ )
              grid[i][j] = temp[i][j];
}
```

SOR: MPI version

- How to partitioning the arrays?
  - double grid[n+1][n/p+1], temp[n+1][n/p+1];





SOR: MPI version

Receive grid[1..n][0] from the process myid-1;

Receive grid[1..n][n/p] from process myid+1;

Send grid[1..n][1] to process myid-1;

Send grid[1..n][n/p-1] to process myid+1;

```
For (i=1; i<n; i++)
    for (j=1; j<n/p; j++)
        temp[i][j] = 0.25* (grid[i][j-1]+grid[i][j+1]
   + grid[i-1][j] + grid[i+1][j]);
```

Sequential Matrix Multiply

```
For (I=0; I<n; I++)
    for (j=0; j<n; j++)
        c[I][j] = 0;
        for (k=0; k<n; k++)
            c[I][j] = c[I][j] + a[I][k] * b[k][j];
```

MPI version? How to distribute a, b, and c? What is the communication requirement?