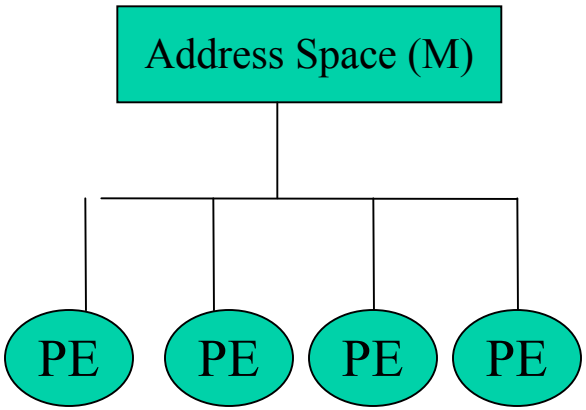
Outline

- Message-Passing Interface
  - Basic concepts
  - Examples
    - Calculating π with integration (a simple program)
    - Calculating π with Monte Carlo (communicators)
    - Performing matrix-vector multiplication
    - Solving the Poisson Equation on a grid (domain decomposition)
    - Parallel IO

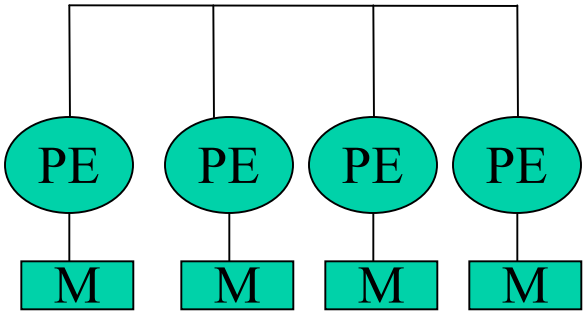
Background (continue)

- Computer architecture
  - Shared-memory model (e.g., CrayC90)



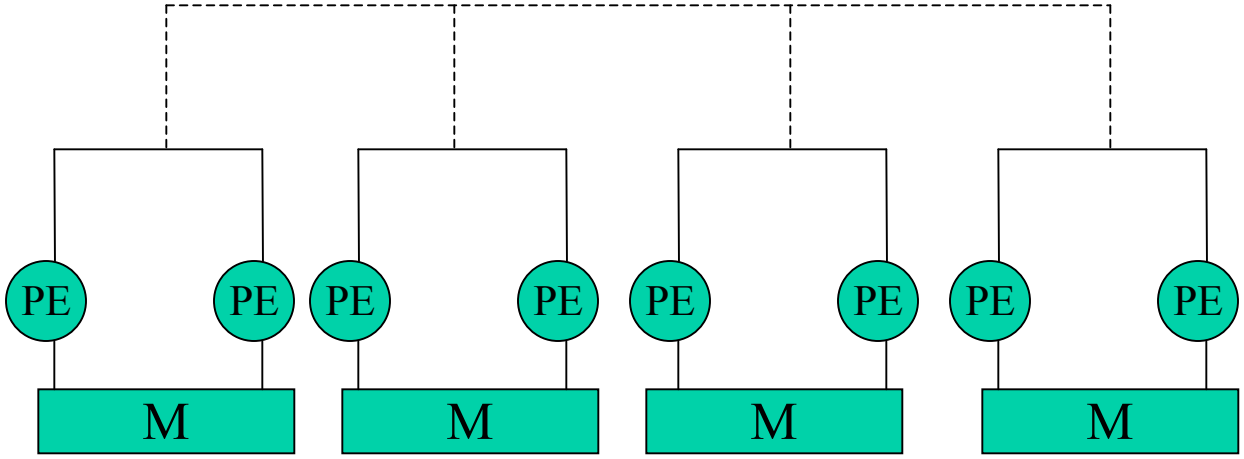
Background (continue)

– Distributed-memory model (e.g., Cray T3D, Linux cluster)



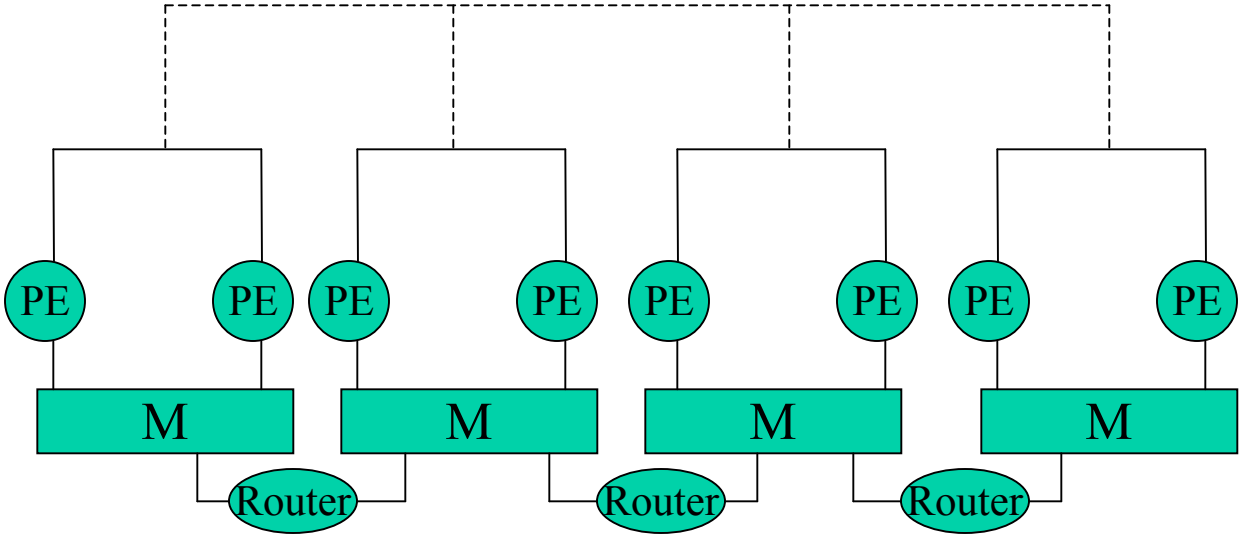
Background (continue)

– Hybrid model (e.q., Compaq SC45)



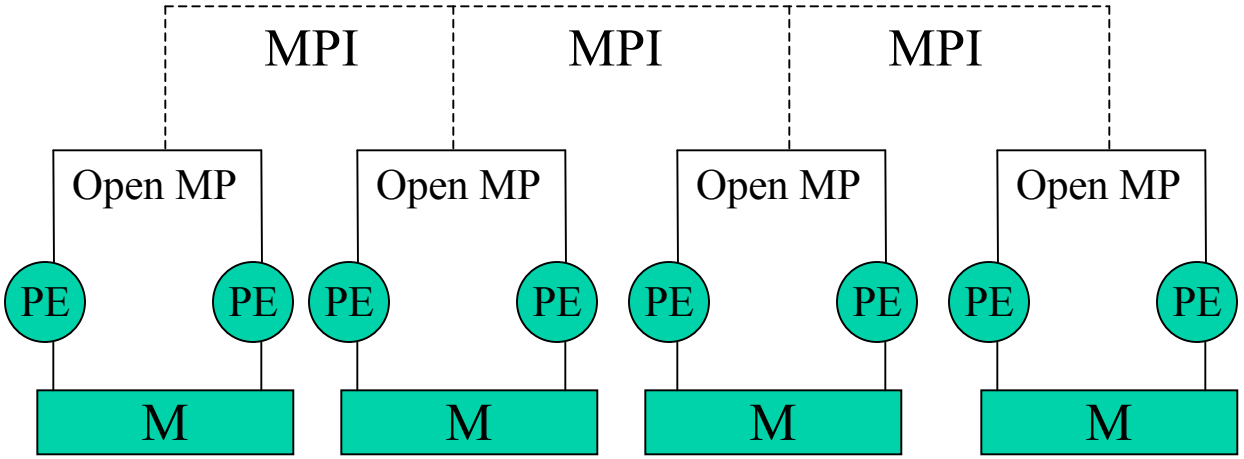
Background (continue)

– Shared-distributed memory model (e.q., SGI Origin2000)



Background (continue)

- Solutions
  - Open MP for shared memory model
  - MPI (Message-Passing Interface) for distributed memory model



What is MPI?

- A *message-passing library specification*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers
- Two popular open-source implementations: MPICH and OpenMPI
- Intel's implementation is widely used in clusters

Why Use MPI?

- MPI provides a powerful, efficient, and *portable* way to express parallel programs
- MPI was explicitly designed to enable libraries which may eliminate the need for many users to learn (much of) MPI
  - Most of large parallel codes are built on top of parallel libraries which is based on MPI

MPI's Pros and Cons

- Advantage
  - **Portable.** The code is portable across different computer platforms (PC/linux, SGI, Sun, IBM workstations, supercomputers)
  - **Performance.** The code is scalable beyond a few hundred or thousand or more processors
- Disadvantage
  - **Difficult.** A developer has to know the algorithms of the code as well as computer architecture to obtain optimal performance

When to use MPI

- Portability and Performance
- Irregular Data Structures
- Building Tools for Others
  - Libraries
- Need to Manage memory on a per processor basis

When *not* to use MPI

- Solution (e.g., library) already exists
  - http://www.mcs.anl.gov/mpi/libraries.html
- Require Fault Tolerance
  - Sockets
- Distributed Computing
  - CORBA, DCOM, etc.

MPI Library Functions

- MPI is large (125 functions)
  - No need to use all
- MPI is small (6 functions)
  - **MPI_INIT** initialize MPI
  - **MPI_COMM_SIZE** find out how many processes there are
  - **MPI_COMM_RANK** find out which process I am
  - **MPI_SEND** send a message
  - **MPI_RECV** receive a message
  - **MPI_FINALIZE** terminate MPI

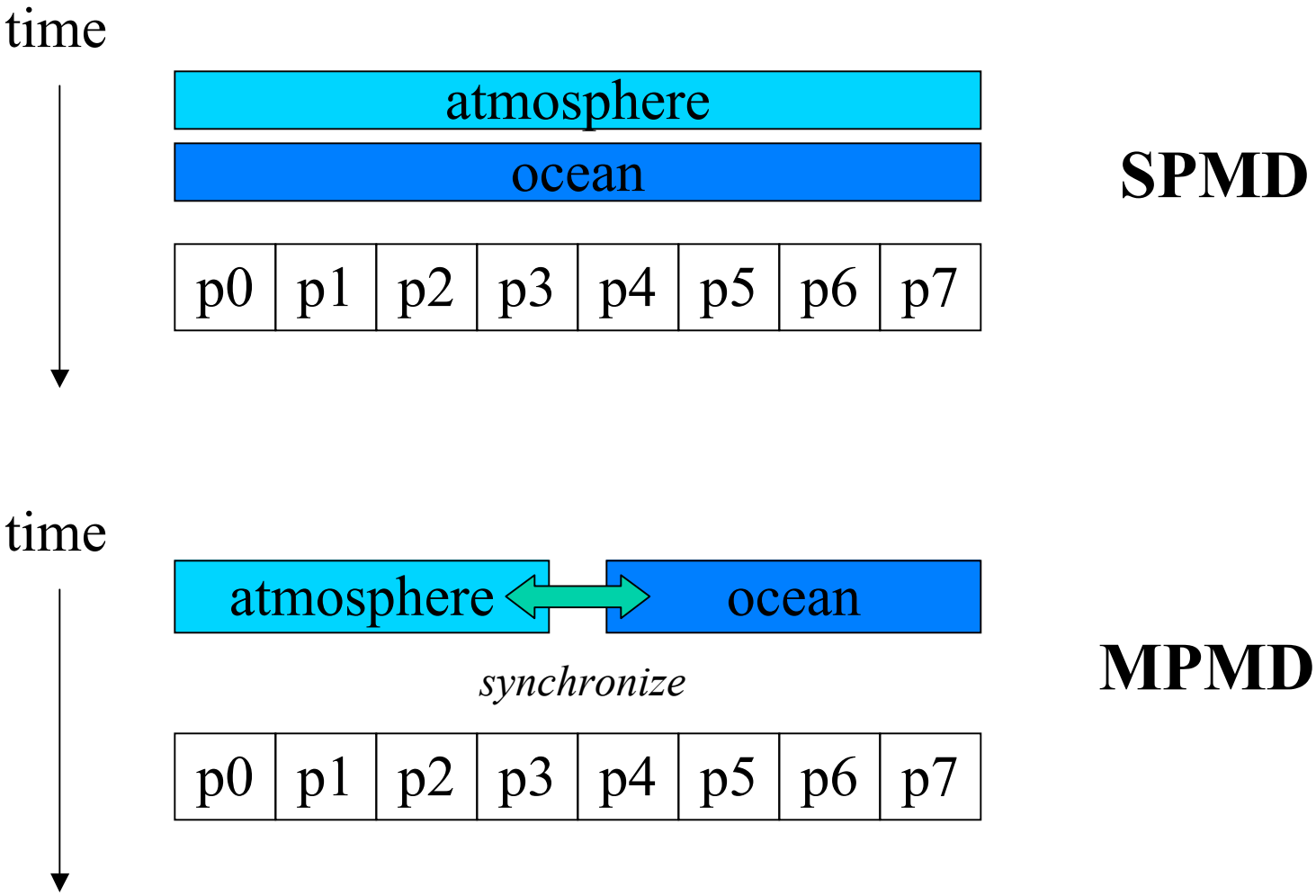
The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's.

Types of Parallel Computing Models

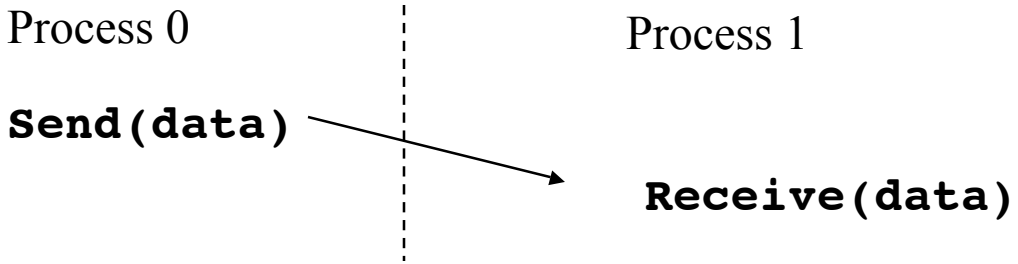
- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
  - Used in IBM Cell's SPU
- Task Parallel - different instructions on different data (MIMD)
- SPMD (single program, multiple data) not synchronized at individual operation level
- MPMD (multiple program, multiple data) synchronized at individual operation level

SPMD vs. MPMD


time
atmosphere
ocean
SPMD
p0 p1 p2 p3 p4 p5 p6 p7
time
atmosphere
ocean
synchronize
MPMD
p0 p1 p2 p3 p4 p5 p6 p7

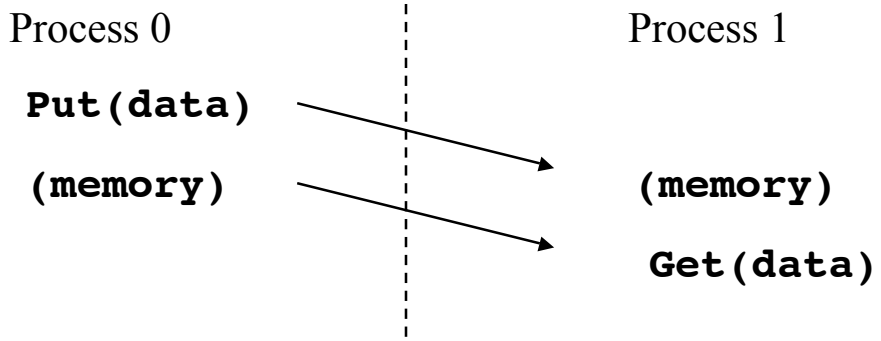
Cooperative Operations for Communication

- The message-passing approach makes the exchange of data *cooperative*.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiving process's memory is made with the receiver's explicit participation.
- Communication and synchronization are combined.



One-Sided Operations for Communication

- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled
  - Enhance performance by reducing latency
- One-sided operations are part of MPI-2.
- IBM Cell's DMA is an example



MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECV`**
- Point-to-point (send/recv) isn't the only way...

Communicator

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

Finding Out the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the *number* of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

Alternative Set of 6 Functions for Simplified MPI

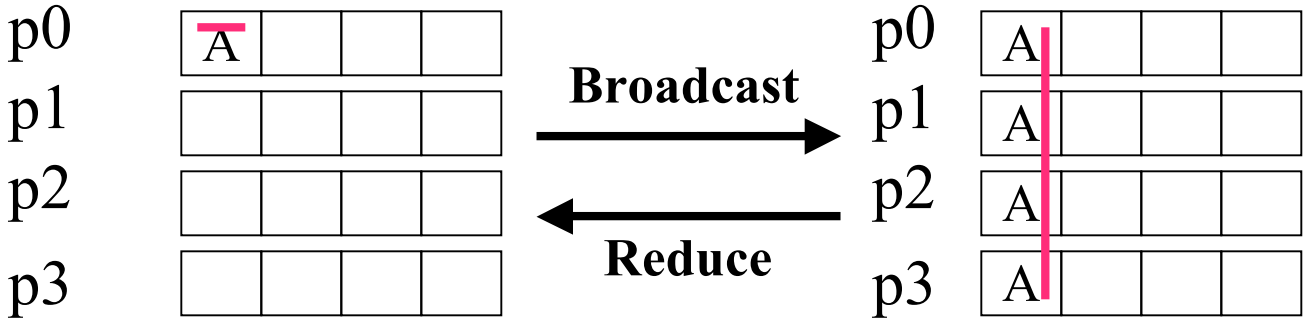
- **MPI_INIT**
- **MPI_FINALIZE**
- **MPI_COMM_SIZE**
- **MPI_COMM_RANK**
- **MPI_BCAST**
- **MPI_REDUCE**

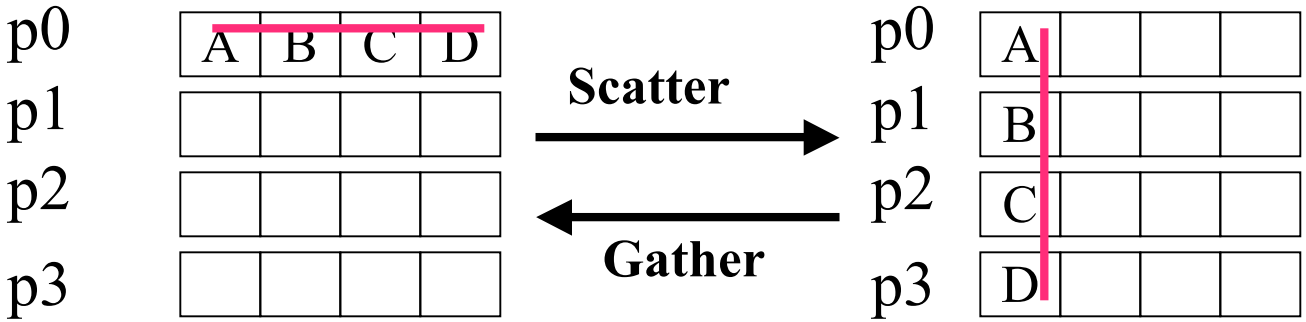
- What else is needed (and why)?
  - MPI_WTtime() timing routine

Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

Collective Operations





Notes on C and Fortran

- C and Fortran bindings correspond closely
- In C:
  - mpi.h must be #included
  - MPI functions return error codes or **`MPI_SUCCESS`**
- In Fortran:
  - mpif.h must be included, or use MPI module (MPI-2)
  - All MPI calls are to subroutines, with a place for the return code in the last argument.
- C++ bindings, and Fortran-90 issues, are part of MPI-2.

Running MPI Programs

- The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
- **`mpiexec <args>`** is part of MPI-2, as a recommendation, but not a requirement
  - You may use mpiexec for MPICH and mpirun for SGI's MPI and Intel's MPI

More details on MPI

Based on “An Introduction to MPI” developed by Gropp and Lusk at ANL

Located at ***http://www-unix.mcs.anl.gov/mpi/***

→ “***Materials for learning MPI***”

→***MPI tutorials***

→***A somewhat longer introduction to MPI***

Examples

Shujia ZHOU UMBC CMSC491A/691A
2009 Spring

Hello (C)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Hello (Fortran)

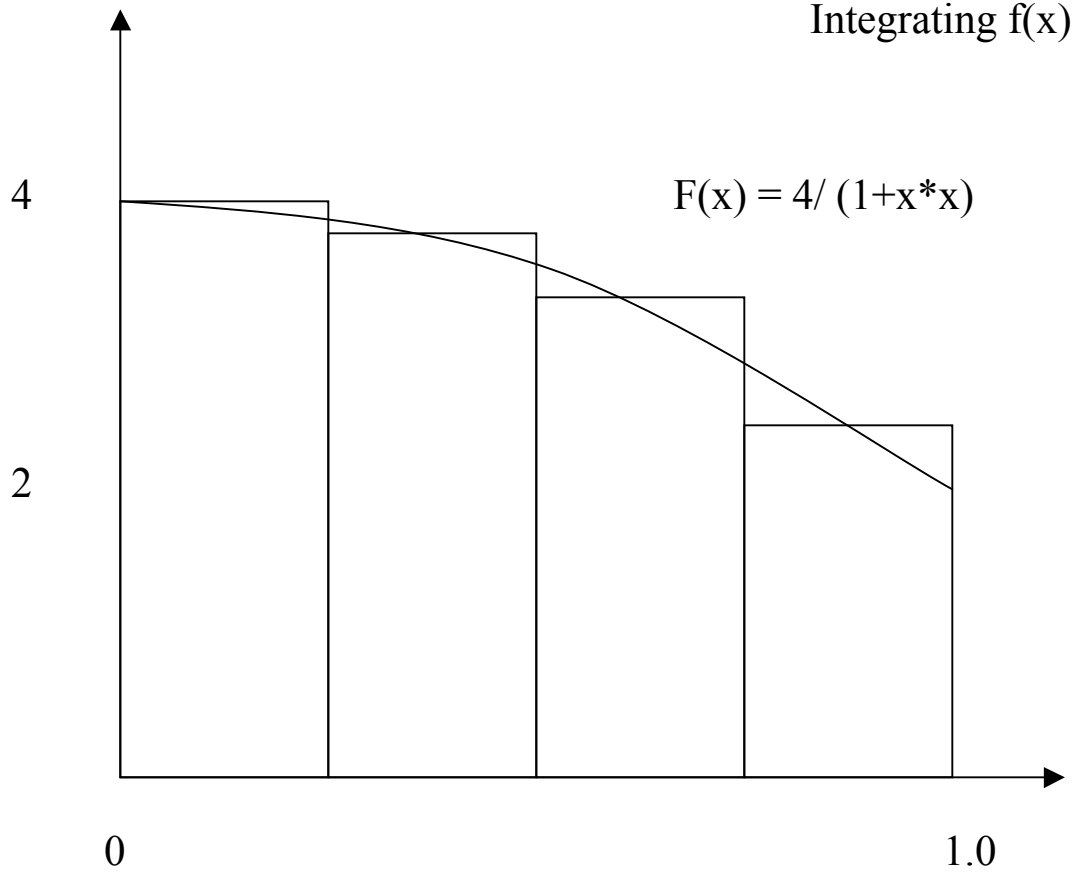
```
program main
use MPI
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

Compile and Run a C version MPI Program in bluegrit

- Ssh bluegrit.cs.umbc.edu
- Ssh b34 (or any node between 34 and 41; PowerPC 2GB, 4 cores, 2.3GHz)
- Compile
  - at one shell of b34 (shell 1)
    - mpicc -o Hello Hello.c
- Run
  - At another shell of b34 (shell 2), request 4 mpi processes
    - Mpd --ncpus=4
  - at shell 1
    - mpirun -np 4 Hello

Integrating to find the value of Pi

Integrating f(x) from 0 to 1 produces pi/4



C program for calculating π

```
#include "mpi.h"
#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;
```

C program for calculating π

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);

while (1) {
      /* processs 0 asks for the input from a user */
      if (myid == 0) {
          printf("Enter the number of intervals: (0 quits) ");
          scanf("%d",&n);
 }

     /* broadcast the value of n to every process */
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

C program for calculating π

```
/* program exits */
if (n == 0)
    break;

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

C program for calculating π

```
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
                        MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",
      pi, fabs(pi - PI25DT));
    } /* end of n!=0 */
  } /* end of while loop */
  MPI_Finalize();
  return 0;
}
```

Homework #1

A particle-system simulation using numerical integration techniques to animate a large set of particles. Numerical integration is implemented using Euler's method of integration. It computes the next value of a function of time, F(t), by incrementing the current value of the function by the product of the time step and the derivative of the function: $F(t + dt) = F(t) + dt*F'(t)$;

Our simple particle system consists of:

- An array of 3-D positions for each particle (pos[])
- An array of 3-D velocities for each particle (vel[])
- An array of masses for each particle (mass[])
- A force vector that varies over time (force)

This programming example is intended to illustrate programming concepts for parallel programming, and is not meant to be a physically realistic simulation. For example, it does not consider:

- How the time-variant force function and the time step, dt, is computed (instead, the example treats them as constants).
- Particle collisions.

In addition, we assume that all 3-D vectors (x,y,z) are expressed as 4-D homogeneous coordinates (x,y,z,1).

We may turn on the interaction among particles in the future home works

```
//this is a serial code

//get END_OF_TIME from a terminal
//#define END_OF_TIME 10
#define PARTICLES 1000000

typedef struct { float x, y, z, w; } vec4D;
vec4D pos[PARTICLES]; // particle positions
vec4D vel[PARTICLES]; // particle velocities
vec4D force; // current force being applied to the particles
float inv_mass[PARTICLES]; // inverse mass of the particles
float dt = 0.01f; // step in time
float temperature;

int main()
{ int i; float time; float dt_inv_mass;

// For each step in time
  for (time=0; time<END_OF_TIME; time += dt)
  { // For each particle
     for (i=0; i<PARTICLES; i++)
```

```
  {
  // Compute the new position and velocity as acted upon by
the force f.
     pos[i].x = vel[i].x * dt + pos[i].x;
     pos[i].y = vel[i].y * dt + pos[i].y;
     pos[i].z = vel[i].z * dt + pos[i].z;
     dt_inv_mass = dt * inv_mass[i];
     vel[i].x = dt_inv_mass * force.x + vel[i].x;
     vel[i].y = dt_inv_mass * force.y + vel[i].y;
     vel[i].z = dt_inv_mass * force.z + vel[i].z;
     temperature = (1.0/PARTICLES)*vel[I].x * vel[I].x +
vel[i].y*vel[i].y + vel[I].z*vel[I].z;
  }
 } return (0);
}
```

Due date: Feb. 23

- Parallelize it with MPI and run with 4 mpi processes and compare the results with the original serial code
- Read in the value of END_OF_TIME from a terminal
- Initialize velocities with the random numbers between 0 and 1
- Initialize the x, y, z force components of each particle with 0.01
- Initialize positions with

```
For (I=-50;I<50;I++) {
 For (j=-50;j<50;j++)   {
  For (k=-50;k<50;k++)  {

        pos[I*(100*100) + j*100 +k]=I*0.1;
        pos[I*(100*100) + j*100 +k]=j*0.1;
        pos[I*(100*100) + j*100 +k]=k*0.1;
   }
  }
}
```