# PARALLEL PROGRAMMING WITH MPI

Part 1

# INTRODUCTION TO PARALLEL COMPUTING

# PARALLEL COMPUTING MEANS:

- Executing more than one process at a time that are related (solving the same problem).
- Processes may or may not need to communicate with one another while they are running.
- Degree of communication required and organization of code determines the type of parallelization chosen.

# TYPES OF PARALLELISM

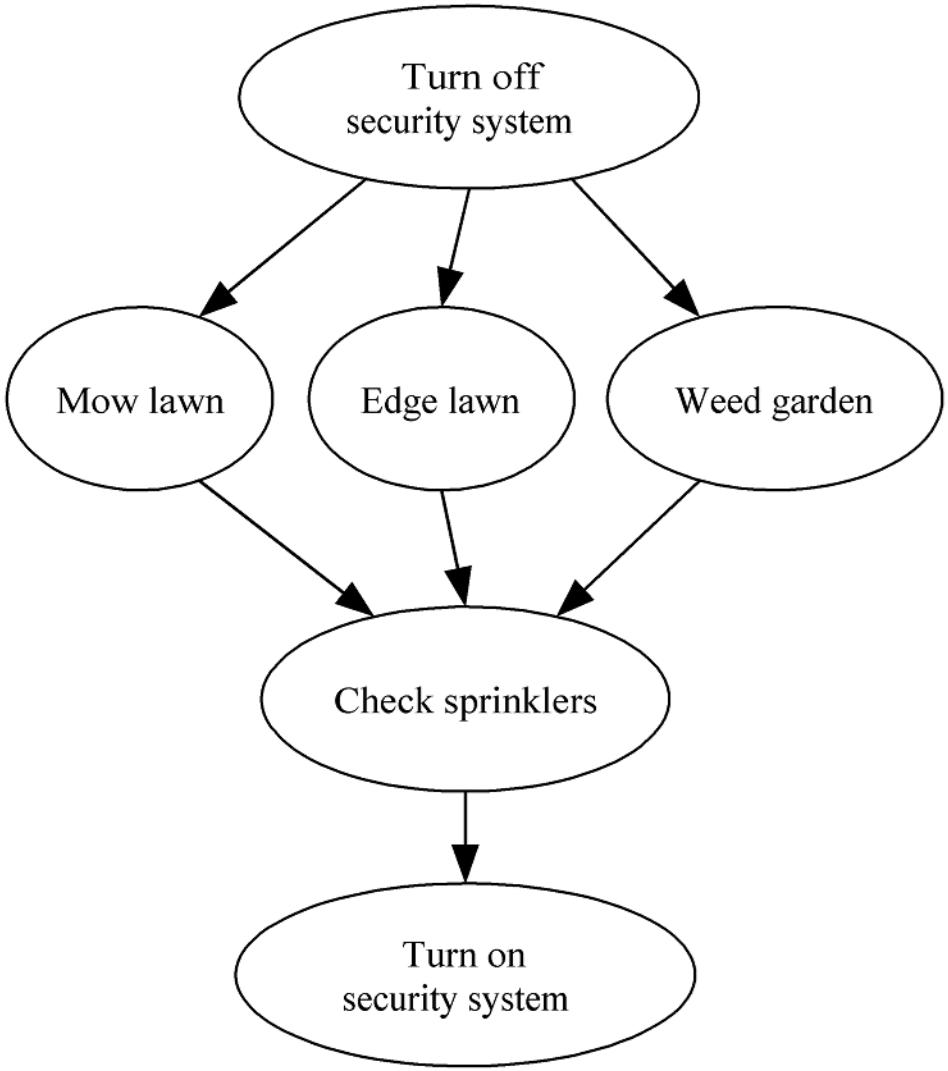
- Embarrassingly parallel (high-throughput computing)
  - Independent processes with little (or no) need to communicate.
- Data parallelism
  - Divide the data into smaller parts. Work on each part individually, then if necessary collect results and go to next phase.
- Task parallelism
  - Perform multiple tasks at the same time on the data.

# EXAMPLE

- The landscape service has several tasks to perform.
  - Turn off security system to access client's garage
  - Mow lawn
  - Edge lawn
  - Weed garden
  - Turn on water to sprinklers, check sprinklers, turn off water.
  - Turn security system back on
- What can be done in parallel (assuming sufficient staff) and what must be serial?

# DATA DEPENDENCE GRAPH

- Directed graph
- Vertices = tasks
- Edges = dependences



# TASK PARALLELISM

- Independent tasks apply different operations to different data elements

$$a \leftarrow 2$$
$$b \leftarrow 3$$
$$m \leftarrow (a + b) / 2$$
$$s \leftarrow (a^2 + b^2) / 2$$
$$v \leftarrow s - m^2$$

- First and second statements
- Third and fourth statements

# DATA PARALLELISM

- Independent tasks apply same operation to different elements of a data set

```
for i ← 0 to 99 do
   a[i] ← b[i] + c[i]
endfor
```

- Safe to perform operations concurrently
- Pop quiz: can you think of an analogy to data parallelism that could be applied to the landscaping problem? Assume you can buy any required "hardware" system.

# PARALLEL PROGRAMMING METHODS

# APPROACHES TO PROGRAMMING PARALLEL COMPUTERS

- Extend compilers: translate sequential programs into parallel programs automatically
- Extend languages: add parallel operations
- Add parallel language layer as a library invoked by sequential languages
- Define totally new parallel language and compiler system

# STRATEGY 1: EXTEND COMPILERS

- Parallelizing compiler
  - Detect parallelism in sequential program
  - Produce parallel executable program
- Historical example: High-Performance Fortran
  - Effort to make automatically parallelizing compiler
  - Never caught on, but some constructs and concepts are in Fortran 90 and up
- Modern example:
  - Compilers that autogenerate parallel threads (e.g. OpenMP).

# EXTEND COMPILERS (CONT.)

- Advantages
  - Can leverage millions of lines of existing serial programs
  - Saves time and labor
  - Requires no retraining of programmers
  - Sequential programming is easier than parallel programming
- Disadvantages
  - Parallelism opportunities may be irretrievably lost when programs are written in sequential languages
  - Performance of parallelizing compilers on broad range of applications still up in air

# STRATEGY 2: EXTEND LANGUAGE

- Add functions to a sequential language
  - Create and terminate processes
  - Synchronize processes
  - Allow processes to communicate
- Advantages
  - Easiest, quickest, and least expensive
  - Allows existing compiler technology to be leveraged
  - New libraries can be ready soon after new parallel computers are available
- Disadvantages
  - Can be very difficult to implement, thus compromising availability and uptake

# EXTEND LANGUAGE (CONT)

- Example: Co-Array Fortran
  - Originally an extended version of Fortran
  - Now in the 2008 standard
- UPC
  - Unified Parallel C
  - Extended version of language
  - Just recently updated to UPC++

# STRATEGY 3: ADD A PARALLEL PROGRAMMING LAYER

- Lower layer
  - Usually in the form of libraries
  - Core of computation
  - Process manipulates its portion of data to produce its portion of result
- Upper layer
  - Handled by programmer invoking the library procedures
  - Creation and synchronization of processes
  - Partitioning of data among processes

# STRATEGY 4: CREATE A PARALLEL LANGUAGE

- Develop a parallel language “from scratch”
- Examples: Occam, Chapel
- Advantages
  - Allows programmer to communicate parallelism to compiler
  - Improves probability that executable will achieve high performance
- Disadvantages
  - Requires development of new compilers
  - New languages may not become standards
  - Programmer resistance

# CURRENT STATUS

- Low-level approach is most popular
  - Augment existing language with low-level parallel constructs
  - MPI and OpenMP are examples
- Advantages of low-level approach
  - Efficiency
  - Portability
- Disadvantage: More difficult to program and debug

# HARDWARE

# INTERCONNECTION NETWORKS

- Uses of interconnection networks
  - Connect processors to shared memory
  - Connect processors to each other
- Can be internal (inside a node) or external

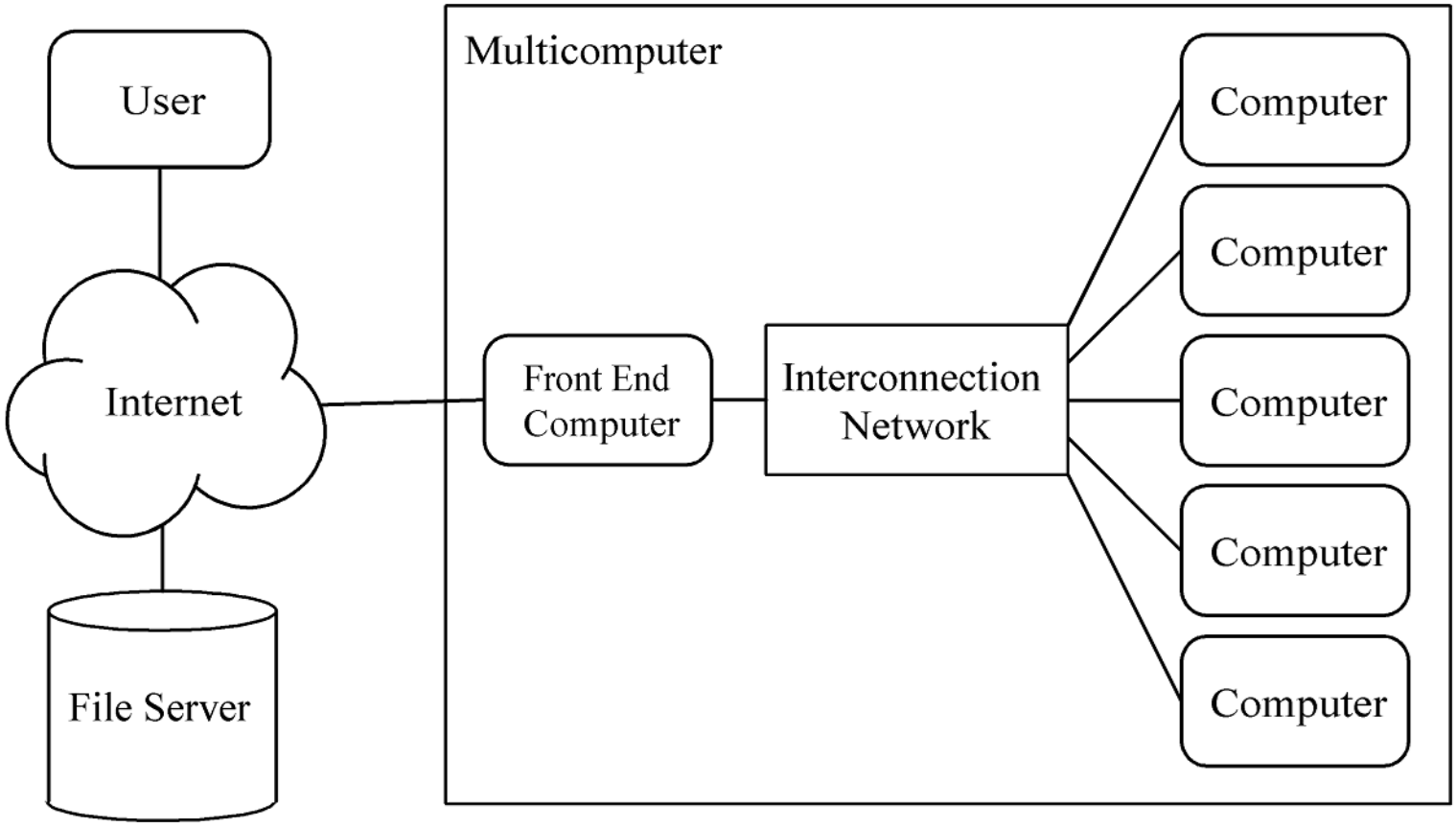
# DISTRIBUTED MEMORY CLUSTERS

- Distributed memory multiple-CPU computer
- Each connected computer is called a **node.**
- Processes interact through message passing

# DISTRIBUTED MEMORY PROGRAMMING MODEL

- Communicate sending “messages”
- A message is an array of bytes
  - send (int dest, char *buf, int len);
  - receive (int &dest, char *buf, int &len);

# SCHEMATIC


User
Internet
File Server
Multicomputer
Front End Computer
Interconnection Network
Computer
Computer
Computer
Computer
Computer


CRAY
CRAY
CRAY

# DEVELOPING PARALLEL STRATEGIES

# GRANULARITY

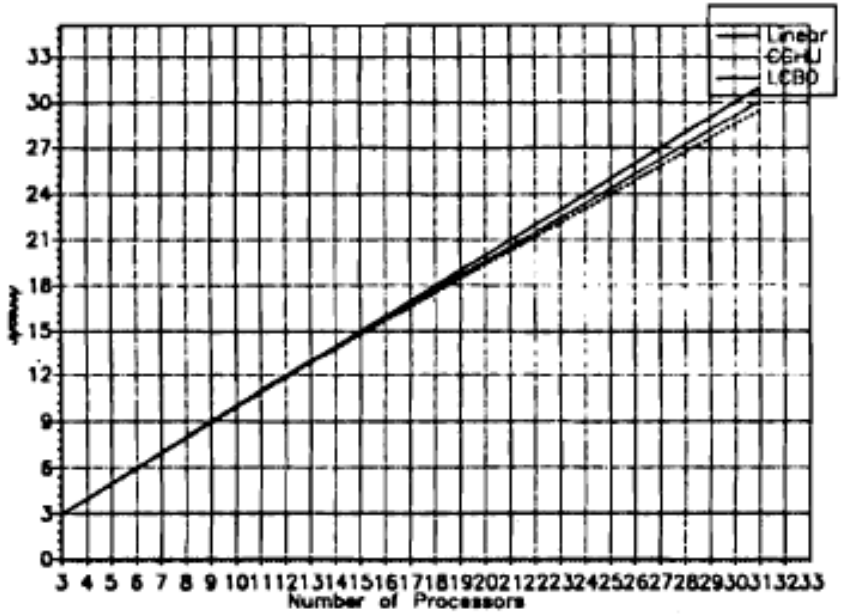
- An important factor in parallelization strategies is the *granularity* of the data.
- Coarse grained: lots of computation, little (or no) communication
- Fine grained: more evenly distributed between computation and communication.
- The granularity determines whether special hardware might be required.

# ALGORITHM SELECTION

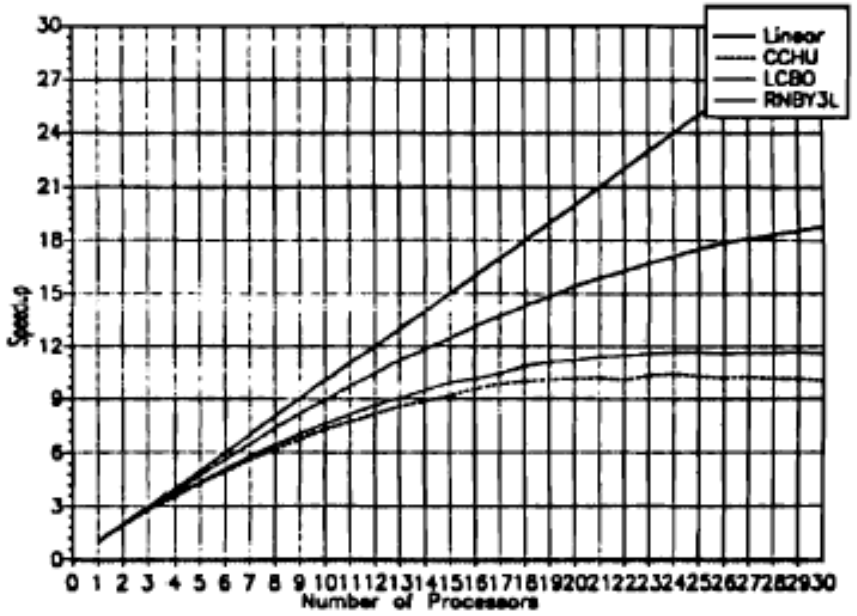
- Some algorithms that are fast in serial cannot be parallelized efficiently
- Some algorithms that are relatively slow in serial are easily parallelizable
- It may be necessary to change your algorithm for a parallel code
- Granularity plays an important role in algorithm parallelizability

# EXAMPLE

- Two gene sequence alignment algorithms
- Smith-Waterman
  - Compares segments of all possible length
  - Optimizes similarity metric
- FASTA
  - Local sequence alignment
  - Heuristics

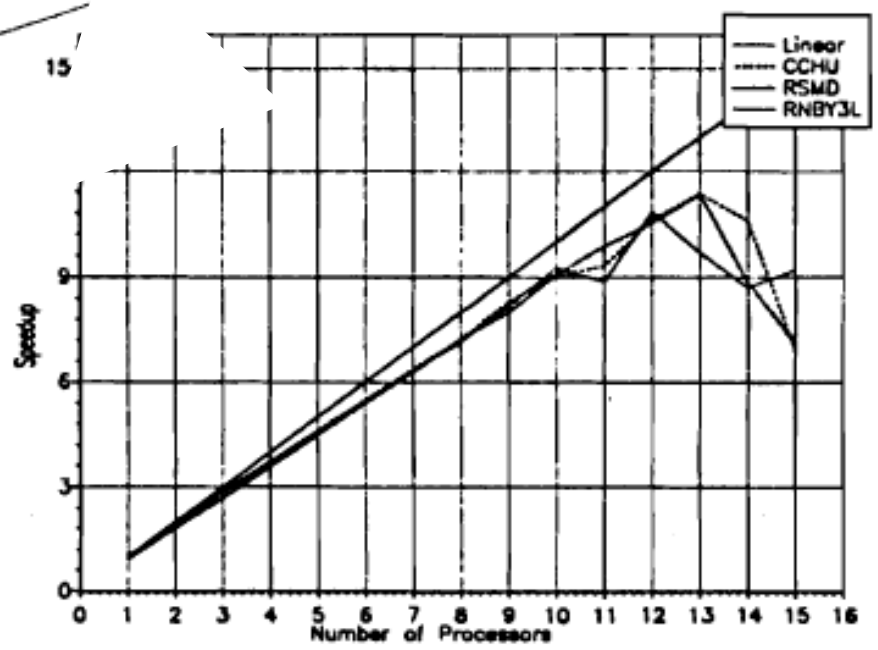



(a) Smith-Waterman

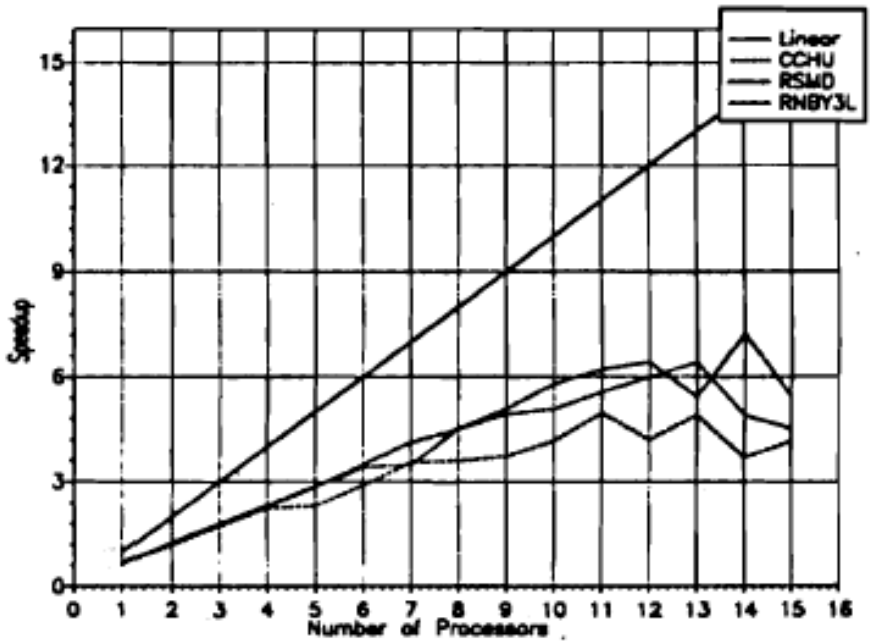



(b) FASTA

iPSC/2 *scanlib* speedups. Speedup is determined by dividing the Mentat time by the sequential C execution time.




(a) Smith-Waterman




(b) FASTA

Sun IPC network *scanlib* speedups. Speedup is determined by dividing the Mentat time by the sequential C execution time.

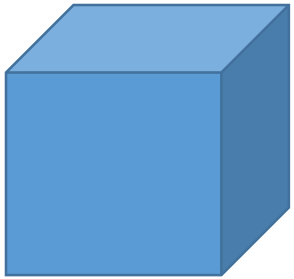
# LOAD BALANCING

- In parallel codes, the runtime is determined by the slowest process
- The computational load should be distributed so that each process does approximately the same share
- Example: with a fixed grid size a general-circulation model must do more computations over land than over ocean. Possible solution: smaller grid sizes over land.

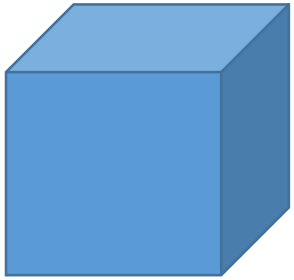
# PARTITIONING

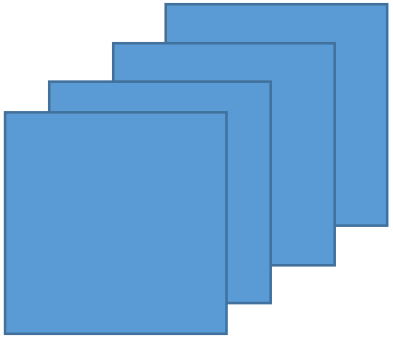
- Dividing computation and data into pieces
- Domain decomposition
  - Divide data into pieces
  - Determine how to associate computations with the data
- Functional decomposition
  - Divide computation into pieces
  - Determine how to associate data with the computations

# EXAMPLE DOMAIN DECOMPOSITIONS

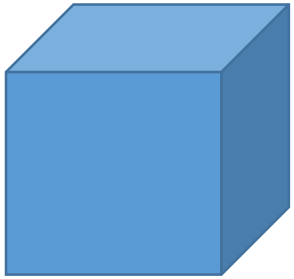


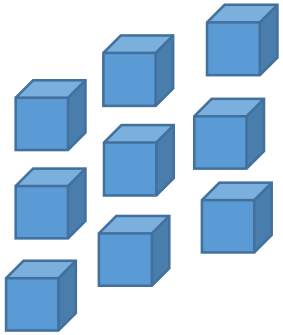
1D





2D





3D

# PARTITIONING CHECKLIST

- ✓Maximize computation to communication ratio
- ✓Minimize redundant computations and redundant data storage
- ✓Quantity of work on each process should be roughly the same size
- ✓Number of tasks is generally an increasing function of problem size

# COMMUNICATION

- Determine values passed among tasks
- Local communication
  - Task needs values from a small number of other processes
- Global communication
  - Significant number of processes contribute data to perform a computation
- Goals:
  - Balance communication operations among tasks
  - As much as possible, each task communicates only with a small group of neighbors

# AGGLOMERATION

- Grouping tasks into larger tasks
- Goals
  - Improve performance
  - Maintain scalability of program
  - Simplify programming
- Due to overhead, it is better to send fewer, larger messages than more, smaller messages
- In MPI programming, the goal is often to create one agglomerated task per processor

# MAPPING

- Process of assigning tasks to processors.
- Threading (multicore): mapping done by operating system.
- Distributed memory system: user chooses how many processes, how many nodes, how many cores per node. These choices can affect performance.
- Conflicting goals of mapping
  - Maximize processor utilization
  - Minimize interprocess communication
- Optimal mapping probably unsolvable in general. Must use heuristics and approximations. Frequently requires experimentation (scaling studies).

# INTRODUCTION TO MPI

# MESSAGE PASSING FOR DISTRIBUTED MEMORY

- The model is *nodes* (computing systems) connected by an *interconnection network*
- Nodes consist of processor(s)/memory/network
- N nodes on the machine
- K processes are distributed to a subset of these N nodes
- Processes communicate by sending/receiving messages.
- A message is a stream of bytes.

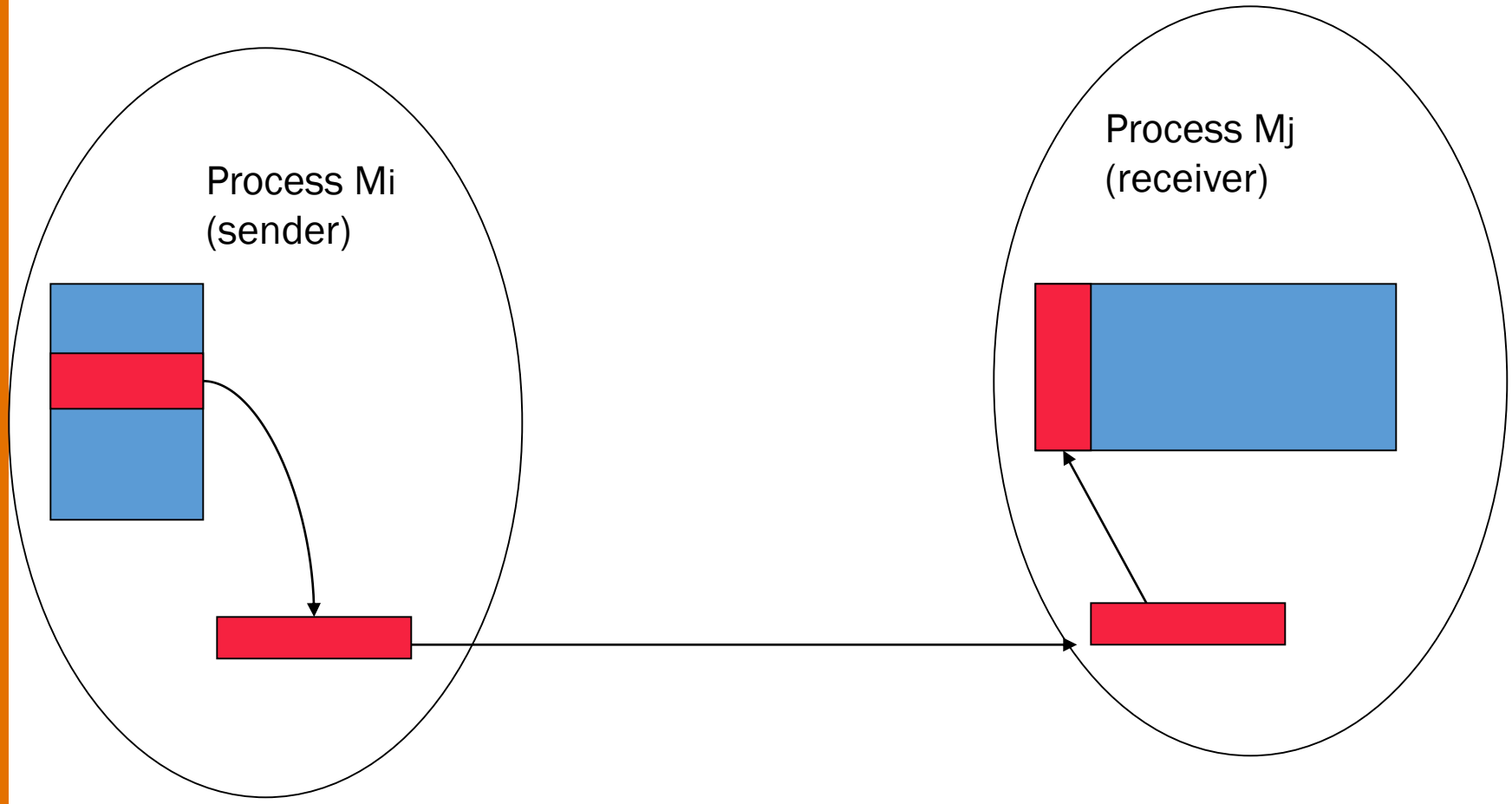
# NETWORK CHARACTERISTICS

- Latency
  - Time from the request for the data to the arrival of the first byte
- Bandwidth
  - Number of bytes per second that can be transmitted
- High latency,  medium to high bandwidth (inexpensive)
  - Most Ethernet
- Low latency, high bandwidth (expensive)
  - Infiniband, OmniPath

# DISTRIBUTED MEMORY PROGRAMMING MODEL

- Each node runs M copies of your executable. The total number of processes is M*Nnodes.
- The executables are built using a communication library.
- Processes communicate by sending “messages.”
- A message is an array of bytes
  - send (int dest, char *buf, int len);
  - receive (int &dest, char *buf, int &len);

# COMMUNICATING WITH MESSAGES



# NAMING

- How do you “name” sources and targets?
  - IP address and port pairs
  - Logical task number
  - Process ID
  - Mailbox
  - Other

# MPI

- MPI stands for Message Passing Interface
- MPI is a standard established by a committee of users and vendors
- MPI is the dominant communication library
- MPI is written in C and ships with bindings for Fortran. Bindings have been written for many other languages including Python and R. C++ programmers should use the C functions.

# MPI

- Usually when MPI is run the number of processes is determined and fixed for the lifetime of the program.
- MPI3 standard can spawn new processes but in a resource managed environment the total number must be requested in advance.
- MPI programs run under the control of an *executor*, usually called **mpiexec** or **mpirun.** In our local environment we use the SLURM process manager **srun.**
  - mpiexec –np 16 myprog
  - When running with srun under SLURM the executor does *not* require the -np flag; it computes the number of processes from the task manager.
- Each process is running a copy of your program.
- Each copy has its own global variables, stack, heap, and program counter.

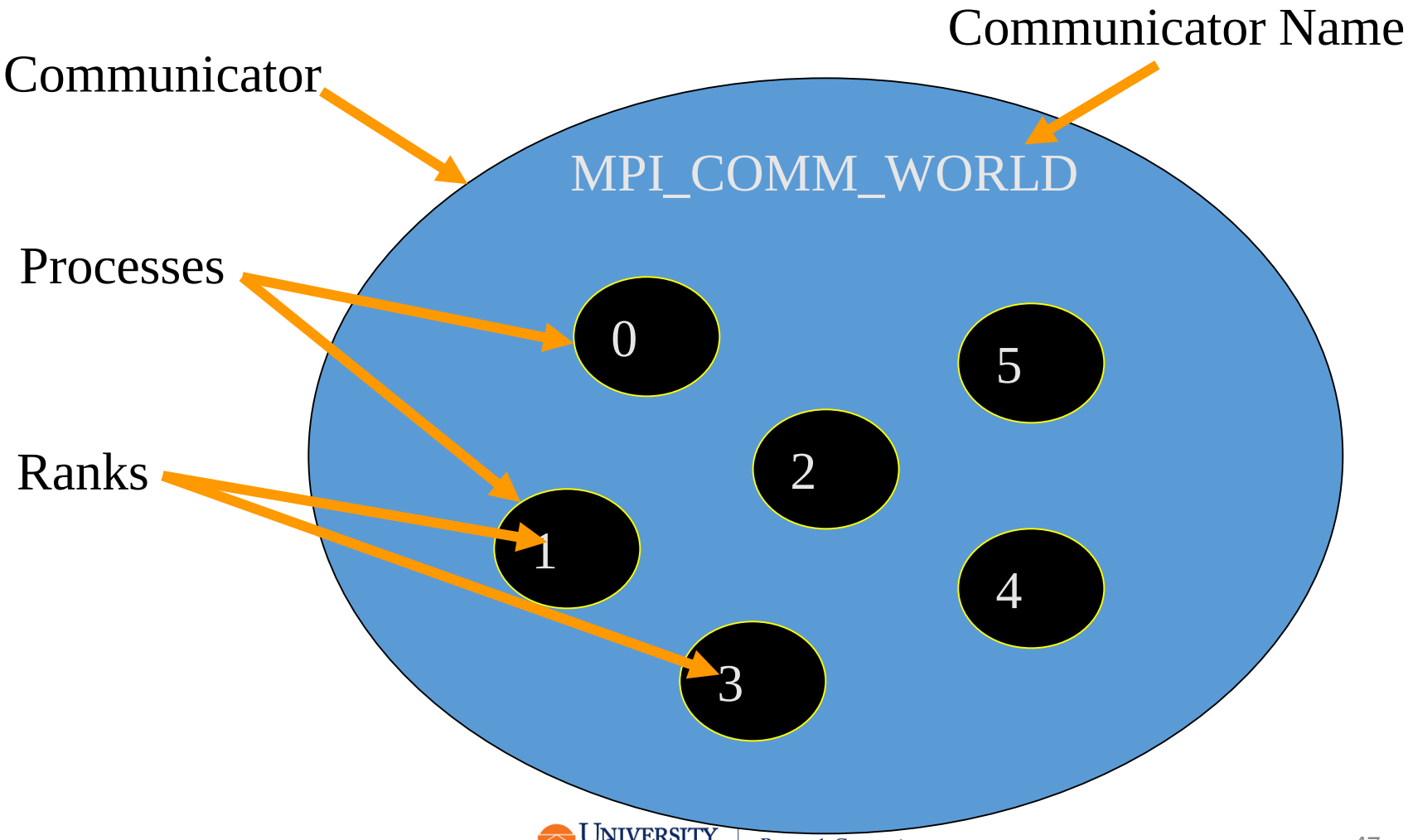
# C VERSUS FORTRAN VERSUS PYTHON

- Many of the examples in this lecture are C or C++ code
  - With some Fortran and Python examples thrown in
  - All of the C functions work the same for Fortran
    - And mostly the same for Python
- The C++ bindings have been deprecated, so we use the C routines for C++ also.
- For Fortran and Python MPI function usage, see the resources at the HPC Bootcamp Collab site under Resources/docs/Fortran or Resources/docs/Python.

# COMMUNICATORS

- Communicator: opaque object that provides message-passing environment for processes
- MPI_COMM_WORLD
  - Default communicator
  - Includes all processes
- It is possible to create new communicators
  - We will not discuss this

# COMMUNICATOR



# MESSAGE ENVELOPES

- A message is uniquely identified by:
  - Source process ID
  - Destination process ID
  - Communicator
  - Tag
- The "tag" can often be set to an arbitrary value such as zero. It is needed only in cases where there may be multiple messages from the same source to the same destination in a short time interval, or a more complete envelope is desired for some reason.

# MESSAGE BUFFERS

- MPI documentation refers to "send buffers" and "receive buffers."
- These refer to *variables* in the program whose contents are to be sent or received. These variables must be set up by the programmer.
- The send and receive buffers cannot be the same unless the special "receive buffer" MPI_IN_PLACE is specified.
- Remember that the buffers we use when we call MPI routines are *variables*.

# INITIALIZE MPI

```
MPI_Init(&argc, &argv);
```

```
call MPI_Init(ierr)
```

- First MPI function called by each process
- Not necessarily first executable statement
- Allows system to do any necessary setup
- Establishes default communicator
- Python: mpi4py calls this when a communicator object is instantiated

# DETERMINE NUMBER OF PROCESSES

```
MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
call MPI_Comm_size(MPI_COMM_WORLD,p,ierr)
```

- First argument is communicator
- Number of processes returned through second argument (an integer)

# DETERMINE PROCESS RANK

```
MPI_Comm_rank(MPI_COMM_WORLD, &id);
```

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

- First argument is communicator
- Process rank (in range 0, 1, …, *p*-1) returned through second argument. It must be an integer.

# SHUTTING DOWN MPI

```
MPI_Finalize();
```

```
call MPI_Finalize()
```

- Call after all other MPI library calls
- Allows system to free up MPI resources

# MPI IN C

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]) {
  int rank, npes;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  if ( rank == 0 ) {
      printf("Running on %d Processes\n",npes);
  }
 fflush(stdout);
  printf("Greetings from process %d\n",rank);
  MPI_Finalize();
}
```

Name this file mpi1.c

# MPI IN FORTRAN

Name this file mpi1.f90

```
program hello
use mpi
integer :: myrank, nprocs
integer :: err
call MPI_INIT(err)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, err)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, err)
if ( myrank .eq. 0 ) then
  print *, 'Running on ',nprocs,' Processes'
endif
print *, 'Greetings from process ', myrank
call flush(6)
call MPI_FINALIZE(err)
end
```

# MPI IN PYTHON

Name this file mpi1.py

```
from mpi4py import MPI
import sys

myrank = MPI.COMM_WORLD.Get_rank()
nprocs = MPI.COMM_WORLD.Get_size()
if myrank == 0:
    sys.stdout.write("Running on %d processes\n" %(nprocs))

sys.stdout.write ("Greetings from process %d\n"%(myrank))
```

# TRY IT!

- On the frontend interactive.hpc.virginia.edu
- Run
  - `module load gcc openmpi`
  - Python: Use Anaconda
  - `module load mpi4py/3.0.0-py3.6`
  - This will also load the correct MPI libraries.

# RUN IT

- IMPORTANT! Use mpiexec and –np only on the frontends!! Use for short tests only!!
- C compiling
  - `mpicc –o mpihello mpi1.c`
- C++ compiling
  - `mpicxx –o mpihello mpi1.cxx`
- Fortran compiling
  - `mpif90 –o mpihello mpi1.f90`
- Both
  - `mpiexec –np 4 ./mpihello`
- Python
  - `mpiexec –np 4 python mpi1.py`

# SUBMIT IT

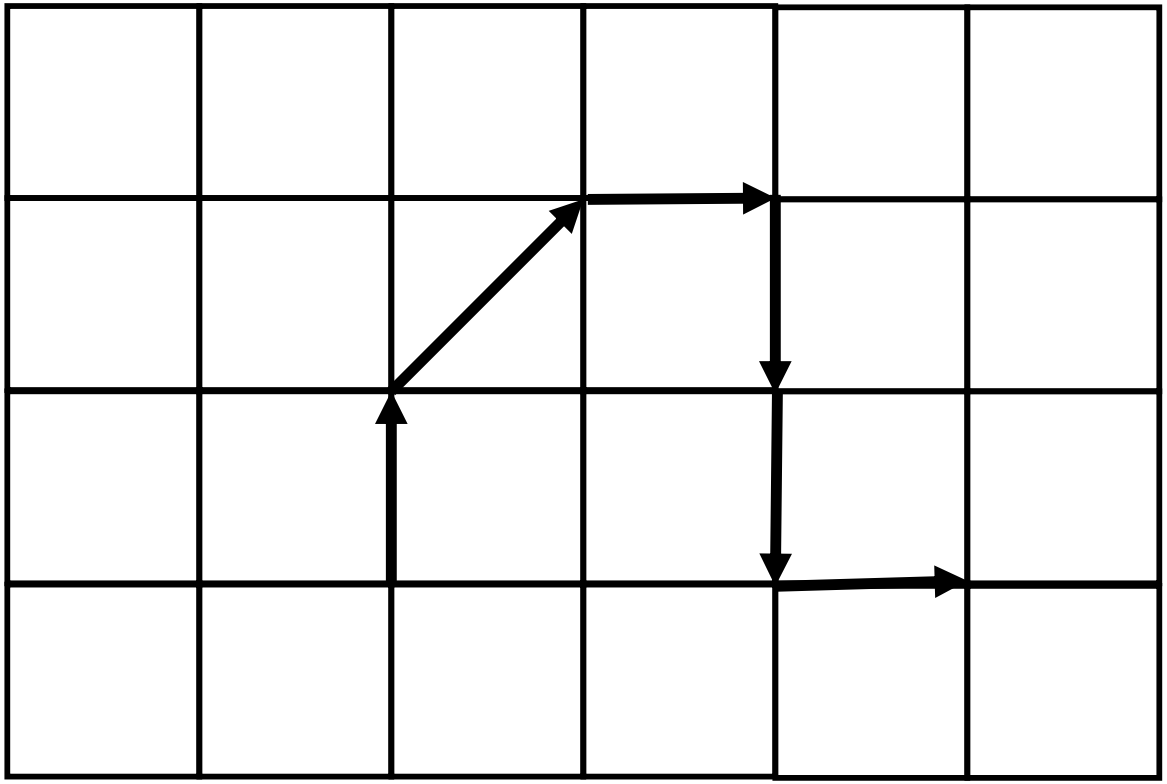
- Write a SLURM script to run your program.
- Request 1 node and 5 cores on the parallel partition.
- The executor will know how many cores were requested from SLURM.
  - `srun ./mpihello`
- Or
  - `srun python mpihello.py`

# EXAMPLE 1

Simple static work distribution

No communication

# TWO-DIMENSIONAL LATTICE RANDOM WALK



# SOLUTION METHOD

- At each step, the "walker" can move left, right, up, or down, with equal probability.
- We seek the distance from the origin after N steps.
- Theoretically, for N steps the distance is $\sqrt{N}$
- We want to test this empirically. But one trial does not give us very good results. We want to run a lot of trials and average the results.

# AGGLOMERATION AND MAPPING

- Properties of parallel algorithm
  - Each trial is independent of the others
  - Fixed number of tasks
  - No communications between tasks to obtain independent result.

# SERIAL CODE (C++)

```
/* Solves a random-walk problem by a brute-force method.
 *
*/

#include <iostream>
#include <string>
#include <sstream>
#include <random>
#include <cmath>

using namespace std;

main(int argc, char **argv) {

   random_device rd;
   mt19937 rng(rd());
   uniform_int_distribution<int> choice(1,4);

   int N;

   if (argc != 2) {
      cout<<"0,0,0\n";
      return 1;
   }
   else {
      string steps=argv[1];
      stringstream ssteps;
      ssteps<<steps;
      ssteps>>N;
   }

   int x=0;
   int y=0;

   int direction;
```

# SERIAL CODE (CONTINUED)

```
 for (int i=1; i<=N; ++i) {
      direction=choice(rng);
      switch (direction) {
         case 1:
           x+=1;
           break;
         case 2:
           x-=1;
           break;
         case 3:
           y+=1;
           break;
         case 4:
           y-=1;
           break;
      }
   }


   double eucDist=sqrt(x*x+y*y);
   cout<<N<<","<<sqrt((double)N)<<","<<eucDist<<"\n";


   return 0;

}
```

# FORTRAN CODE DOWNLOAD RANDOM.F90 MODULE WITH LABS

```
program random_walk
use random
implicit none


  integer                 :: nargs
  character(len=10)       :: ns,nexp
  integer                 :: m, n, ntrials, nsteps, step
  real, dimension(2)      :: pos, move
  real                    :: distance, avg



  nargs=command_argument_count()
  if ( nargs .ne. 1 ) then
     write(*,*) 0, 0., 0.
     stop
  else
     call get_command_argument(1,ns)
     read(ns,'(i10)') nsteps
  endif


  call set_random_seed()


  pos=[0.,0.]
```

# FORTRAN CODE (CONTINUED)

```
  avg=0.

   do n=1,nsteps
      step=randint(1,4)
      if (step==1) then
         move=[0.,1.]
      else if (step==2) then
         move=[0.,-1.]
      else if (step==3) then
         move=[1.,0.]
      else
         move=[-1.,0.]
      endif
      pos=pos+move
   enddo

   distance=sqrt(pos(1)**2+pos(2)**2)

   write(*,*) nsteps, sqrt(real(nsteps)), distance

end program
```

# PYTHON CODE

```
import numpy as np
import random
import sys


if len(sys.argv)>1:
   Nsteps=int(sys.argv[1])
else:
   print 0, 0., 0.
   sys.exit()


pos=np.array([0,0])


moves=[np.array([0,1]),np.array([0,-1]),np.array([1,0]),np.array([-1,0])]


for n in range(Nsteps):
    move=random.choice(moves)
    pos+=move


distance=np.sqrt(pos[0]**2+pos[1]**2)
print Nsteps, np.sqrt(Nsteps),
distance
```

# ADD MPI (ONLY C++ SHOWN)

```
/* Solves a random-walk problem by a brute-force method.
 *
*/

#include <iostream>
#include <string>
#include <sstream>
#include <random>
#include <cmath>
#include <mpi.h>

using namespace std;

main(int argc, char **argv) {

   random_device rd;
   mt19937 rng(rd());
   uniform_int_distribution<int> choice(1,4);

   int npes, rank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   int N;

   if (argc != 2) {
      cout<<rank<<":0,0,0\n";
      MPI_Finalize();
      return 1;
   }
   else {
      string steps=argv[1];
      stringstream ssteps;
      ssteps<<steps;
      ssteps>>N;
   }
```

# MPI (CONTINUED)

```
    int x=0;
    int y=0;

    int direction;

    for (int i=1; i<=N; ++i) {
       direction=choice(rng);
       switch (direction) {
          case 1:
            x+=1;
            break;
          case 2:
            x-=1;
            break;
          case 3:
            y+=1;
            break;
          case 4:
            y-=1;
            break;
       }
    }

    double eucDist=sqrt(x*x+y*y);
    cout<<rank<<":"<<N<<","<<sqrt((double)N)<<","<<eucDist<<"\n";

    MPI_Finalize();

    return 0;

}
```

# RUN IT

- mpicxx –std=c++11 -o randc mpirandom_walk.cxx
  - Newer compilers won't need the –std=c++11 flag
- mpirun -np 4 ./randc 1000000

0:1000000,1000,898.186

1:1000000,1000,189.589

2:1000000,1000,1235.87

3:1000000,1000,391.479

- I have to compute the average manually
  - 678.781

# TRY WITH -NP 8

0:1000000,1000,1011.24
5:1000000,1000,1569.2
3:1000000,1000,1753.23
2:1000000,1000,967.042
1:1000000,1000,1212.17
6:1000000,1000,418.708
7:1000000,1000,881.862
4:1000000,1000,744.641

- The first number is the rank. Why is it jumbled?
- MPI output is *nondeterministic* unless forced to order it (which requires barriers).

# GLOBAL COMMUNICATIONS

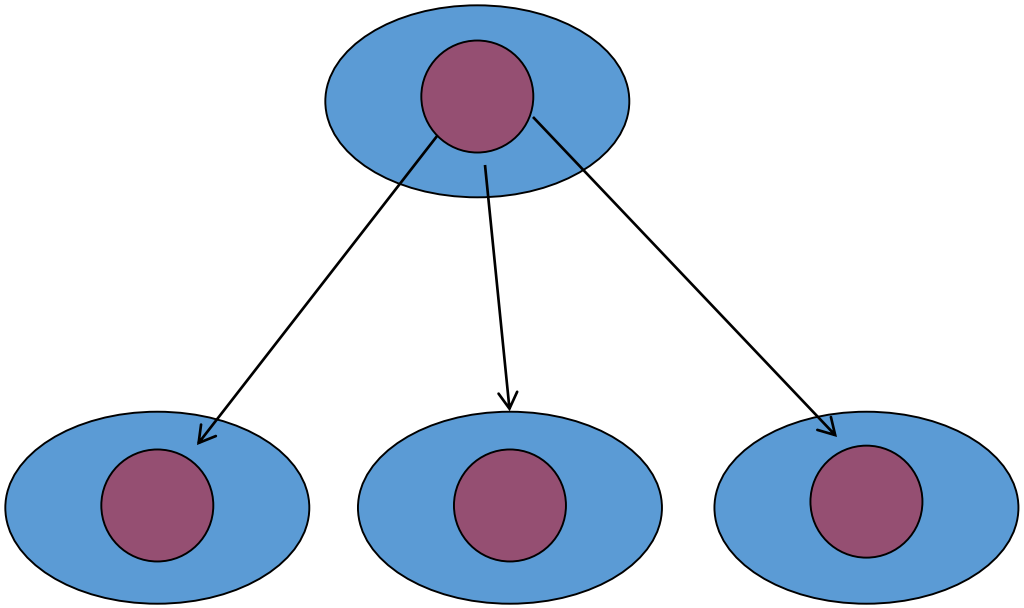
# TYPES OF GLOBAL COMMUNICATIONS

- One-to-many
  - Broadcast
  - Scatter
- Many-to-one
  - Scatter
  - Reduction
- Many-to-many
  - Alltoall
  - Barrier

# MPI_BARRIER

- A barrier causes all processes to pause until each process has invoked the barrier function.
- Barriers can be used to synchronize but reduce parallel efficiency.
- Global communications have implicit barriers.
- C/C++: `MPI_Barrier(MPI_COMM_WORLD);`
- Fortran: `call MPI_Barrier(MPI_COMM_WORLD,ierr)`
- Python: `MPI_COMM_WORLD.Barrier()`

# BROADCAST



# PROTOTYPE FOR MPI_BCAST

```
int MPI_Bcast (
    void            *buffer,
    int             count,
    MPI_Datatype    datatype,
    int             root,
    MPI_Comm        communicator)
```

- The data is sent from the *root* node to all other nodes
- In the *root* node, *buffer* is read
  - In all other nodes, *buffer* is written to

# FORTRAN SYNTAX

```
call MPI_Bcast(vals,ncount,MPI_TYPE,root,   &
                           MPI_COMM_WORLD,err)
```

- Broadcast is one-to-many
- The same values are sent to each process from root
- Scatter is similar but breaks the values into parts.

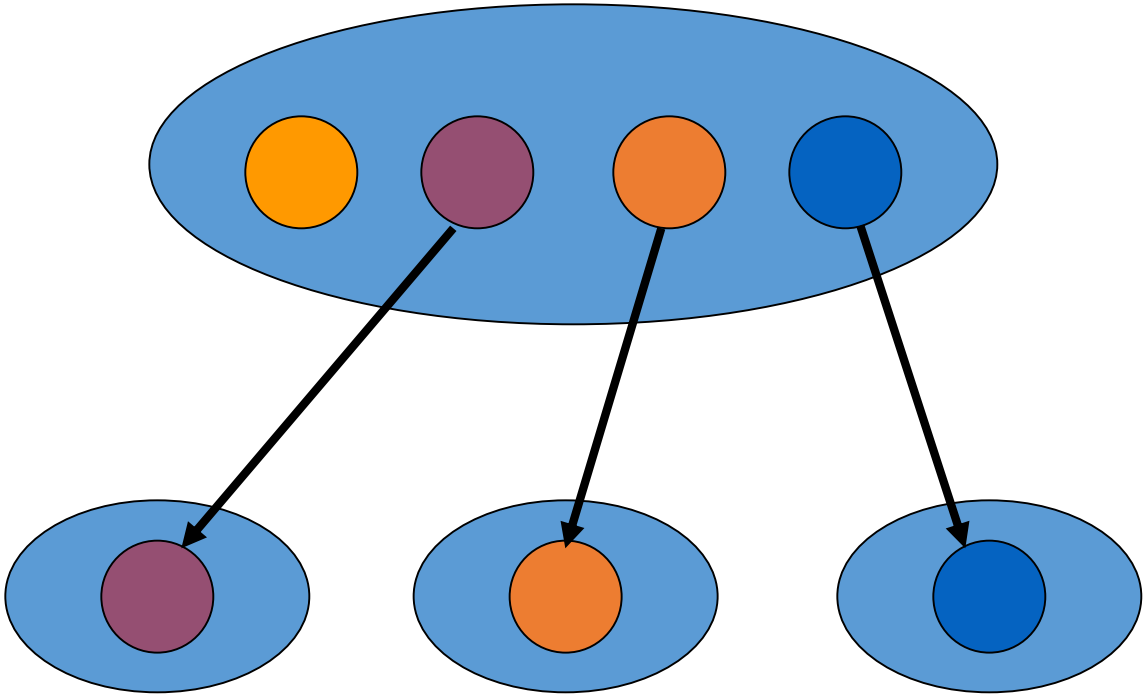
# PYTHON SYNTAX

- Pretty simple:

```
comm.Bcast([sendvals,MPI.DOUBLE],root=0)
```

- `sendvals` should be a Numpy array, even if it has only one element.

# SCATTER



# PROTOTYPE FOR MPI_SCATTER

```
int MPI_Scatter (
    void          *sendbuffer,
    int           count,
    MPI_Datatype  datatype,
    void          *recvbuffer,
    int           count,
    MPI_Datatype  datatype,
    int           root,
    MPI_Comm      communicator)
```

- The data is sent from the *root* node to all other nodes. Each process sends *count* items which are distributed in rank order.
- In the *root* node, *sendbuffer* contains all the data so is larger than recvbuffer (count*nprocs)

# FORTRAN SYNTAX

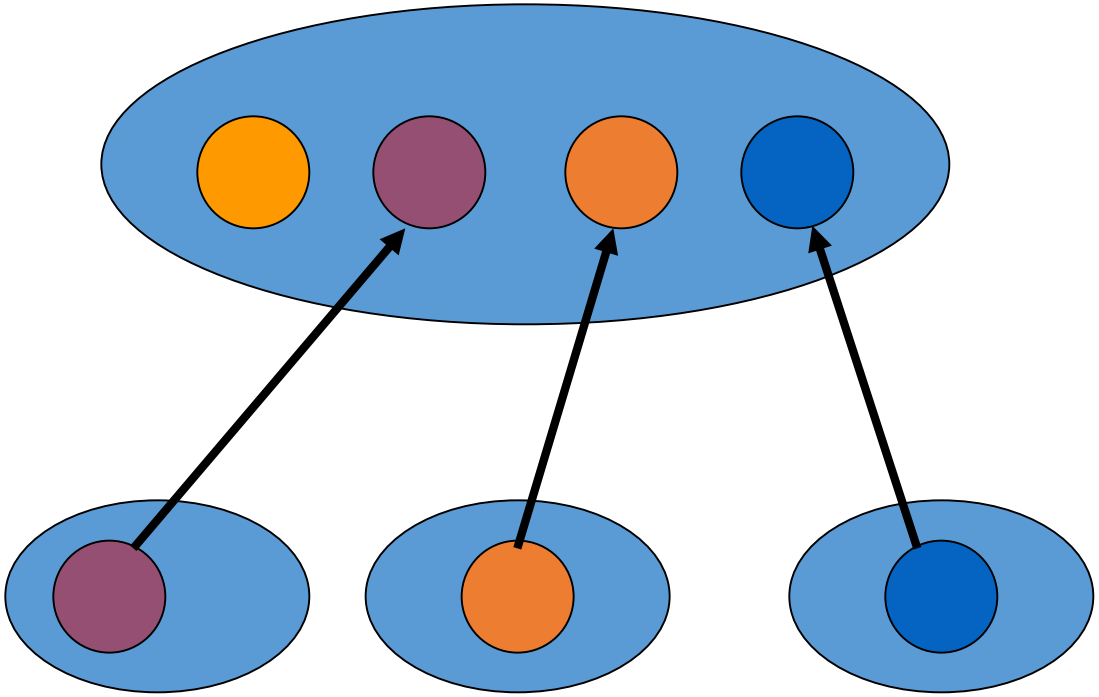
```
call MPI_Scatter (vals,ncount,MPI_TYPE, &
                  rvals,ncount,MPI_TYPE,root,  &
                                MPI_COMM_WORLD,err)
```

# PYTHON SYNTAX

- Both buffers should be Numpy arrays (type all in one line in code):

```
comm.Scatter([sendvals,MPI.DOUBLE],[recvals,MPI.DOUBLE,
             root=0)
```

# GATHER



# PROTOTYPE FOR MPI_GATHER

```
int MPI_Gather (
    void          *recvbuffer,
    int           count,
    MPI_Datatype  datatype,
    void          *sendbuffer,
    int           count,
    MPI_Datatype  datatype,
    int           root,
    MPI_Comm      communicator)
```

- The data is sent from each node to the *root* node. The data are equally gathered from all processes and collected in rank order.
- In the *root* node, *buffer* is read. It must be declared to be large enough to accommodate all the data.

# FORTRAN SYNTAX

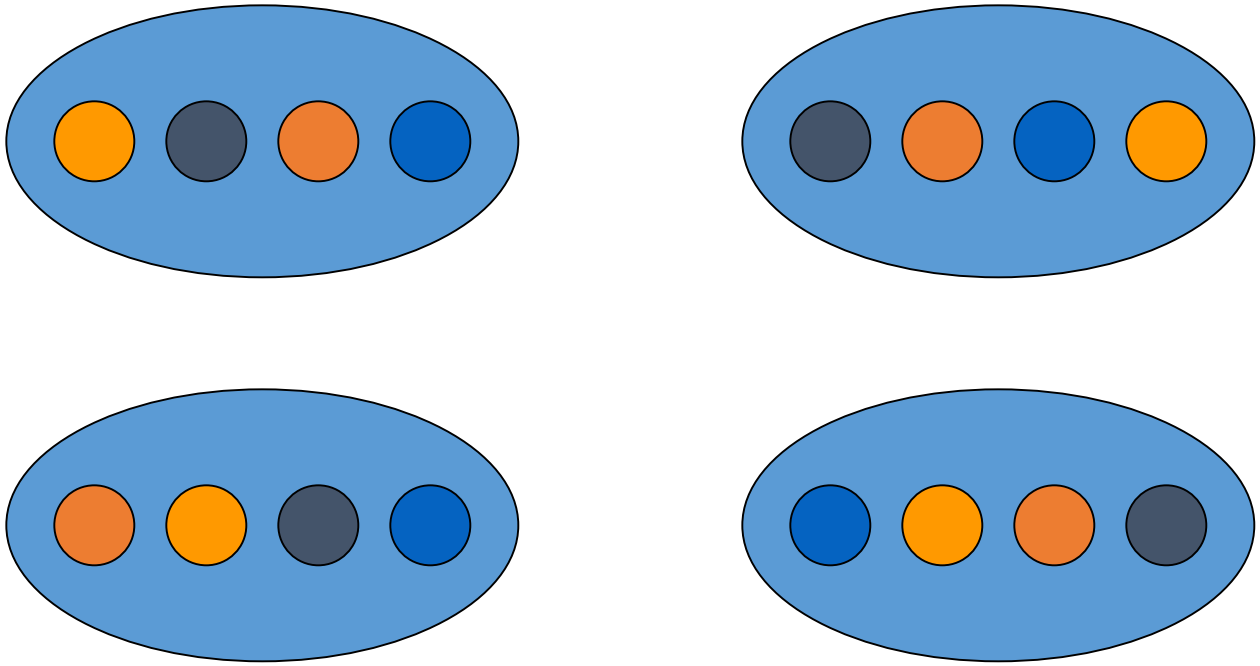
```
call MPI_Gather(vals,ncount,MPI_TYPE, &
                rvals,ncount,MPI_TYPE,root,  &
                              MPI_COMM_WORLD,err)
```

# PYTHON SYNTAX

- Both buffers should be Numpy arrays (type all in one line in code):

```
comm.Gather([sendvals,MPI.DOUBLE],[recvals,MPI.DOUBLE,
            root=0)
```

# ALLGATHER



# ALLGATHER SYNTAX

- Allgather is identical to gather but omitting the root variable. Data are gathered on process 0 then the accumulated buffer is broadcast to all other processes.
- Use allgather rather than gather/bcast if all data need to know the total.

```
MPI_Allgather(&svals,ncount,MPI_TYPE,
              &rvals,ncount,MPI_TYPE,
                 MPI_COMM_WORLD)
```

# PROTOTYPE OF MPI_REDUCE (C)

```
int MPI_Reduce (
   void *operand,/* addr of 1st reduction element */
   void *result, /* addr of 1st reduction result */
   int count,    /* reductions to perform */
   MPI_Datatype type, /* type of elements */
   MPI_Op operator, /* reduction operator */
    int root,   /* process getting result(s) */
    MPI_Comm comm  /* communicator */
)
```

# PROTOTYPE OF MPI_REDUCE (FORTRAN)

- `MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierr)`
  - `integer count, datatype, op, root, comm, ierr`
  - `<type> sendbuf(<length>), recvbuf(<length>)`
- Example:

```
call MPI_REDUCE(myval,val,1,MPI_REAL,      &
        MPI_SUM,0,MPI_COMM_WORLD,ierr)
```

# PROTOTYPE OF MPI_REDUCE (PYTHON)

- `Reduce (sendobj, recvobjc, operation, root)`
  - Note that the lower-case 'reduce' handles pickled objects; use the title-case 'Reduce' with NumPy arrays.
  - Type all in one line (PowerPoint and Python don't get along).

- `MPI.COMM_WORLD.Reduce ([hits,MPI.DOUBLE], [total_hits,MPI.DOUBLE], op=MPI.SUM, root=0)`
  - Both hits and total_hits are initalized numpy arrays of one element.
    - total_hits can have any value (as it's being overwritten), but it must be initialized.
    - When creating a numpy array, by default it creates it as a double.
  - The number of elements that are being reduced is based on the size of the hits and total_hits arrays.

# MPI_DATATYPE OPTIONS FOR C

- MPI_CHAR
- MPI_DOUBLE
- MPI_FLOAT
- MPI_INT
- MPI_LONG
- MPI_LONG_DOUBLE
- MPI_SHORT
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_UNSIGNED_SHORT

# MPI_DATATYPE OPTIONS FOR FORTRAN

- MPI_CHARACTER
- MPI_DOUBLE_PRECISION
- MPI_REAL
- MPI_INTEGER
- MPI_LOGICAL
- MPI_COMPLEX
- MPI_DOUBLE_COMPLEX

# MPI_OP OPTIONS (C AND FORTRAN)

- MPI_BAND
- MPI_BOR
- MPI_BXOR
- MPI_LAND
- MPI_LOR
- MPI_LXOR
- MPI_MAX
- MPI_MAXLOC
- MPI_MIN
- MPI_MINLOC
- MPI_PROD
- MPI_SUM

# ENHANCING THE RANDOM-WALK PROGRAM

I'd rather not do that average myself

Incorporate sum-reduction into program

Reduction is a collective communication

# OUR CALL TO MPI_REDUCE

```
MPI_Reduce (&eucDist,
            &total,
            1,
            MPI_DOUBLE,
            MPI_SUM,
            0,
            MPI_COMM_WORLD);
```

**Only process 0 will get the result**

```
if (rank==0) {
    double avg=total/npes;
    cout<<N<<","<<sqrt((double)N)<<","<<avg<<"\n";
 }
```

# IMPROVED VERSION

```
/* Solves a random-walk problem by a brute-force method.
 *
*/

#include <iostream>
#include <string>
#include <sstream>
#include <random>
#include <cmath>
#include <mpi.h>


using namespace std;


main(int argc, char **argv) {


   random_device rd;
   mt19937 rng(rd());
   uniform_int_distribution<int> choice(1,4);


   int npes, rank;


   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &npes);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);


   int N;


   if (argc != 2) {
      if (rank==0) {
         cout<<":0,0,0\n";
      }
      MPI_Finalize();
      return 0;
   }
   else {
      string steps=argv[1];
      stringstream ssteps;
      ssteps<<steps;
      ssteps>>N;
   }
```

# REDUCTION (CONTINUED)

```
  int x=0;
  int y=0;

   int direction;

   for (int i=1; i<=N; ++i) {
      direction=choice(rng);
      switch (direction) {
         case 1:
           x+=1;
           break;
         case 2:
           x-=1;
           break;
         case 3:
           y+=1;
           break;
         case 4:
           y-=1;
           break;
      }
   }

   double eucDist=sqrt(x*x+y*y);

   double total;
   MPI_Reduce(&eucDist,&total,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

   if (rank==0) {
      double avg=total/npes;
      cout<<N<<","<<sqrt((double)N)<<","<<avg<<"\n";
   }

   MPI_Finalize();

   return 0;

}
```

# RESULT

- mpirun -np 8 ./randc 100000
- 100000,316.228,252.55

# BENCHMARKING THE PROGRAM

- `MPI_Barrier` – barrier synchronization
- `MPI_Wtick` – timer resolution
- `MPI_Wtime` – current time

- Weak scaling: N steps on each process

# BENCHMARKING CODE

```
double elapsed_time;
…
MPI_Init (&argc, &argv);
…
elapsed_time=MPI_Wtime()-elapsed_time;
…
MPI_Reduce (…);
elapsed_time += MPI_Wtime();
```

Generally we look only at time from the root process.

# JOB SCRIPT

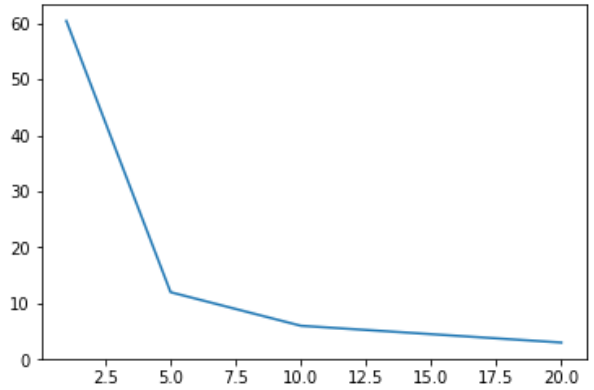
```
#!/bin/bash
#SBATCH -A rivanna-training
#SBATCH -t 00:10:00
#SBATCH -p parallel
#SBATCH -N 1
#SBATCH --ntasks-per-node=20

module load gcc
module load openmpi
srun ./randc  100000000
```

# BENCHMARKING RESULTS

Running the random walk for 1,000,000,000 steps

| Processors | Time (sec) |
|---|---|
| 1 | 60.423 |
| 5 | 12.0012 |
| 10 | 6.00945 |
| 20 | 3.01335 |



# CREDITS

- These slides were developed by
  - Andrew Grimshaw, UVA Department of Computer Science
  - Aaron Bloomfield, UVA Department of Computer Science
  - Katherine Holcomb
- Some slides based on material in *Parallel Computing: Theory and Practice*, by Michael J. Quinn.