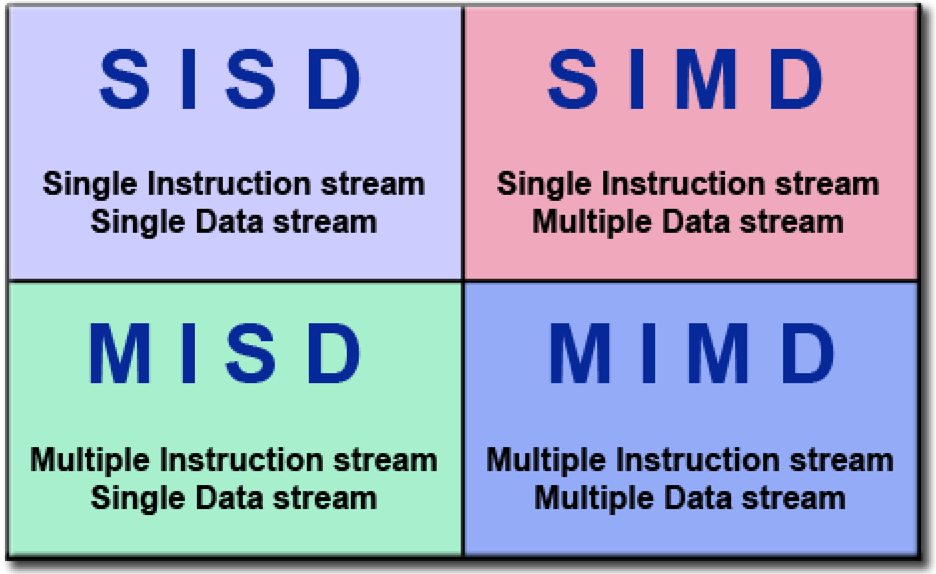
Intro to MPI programming

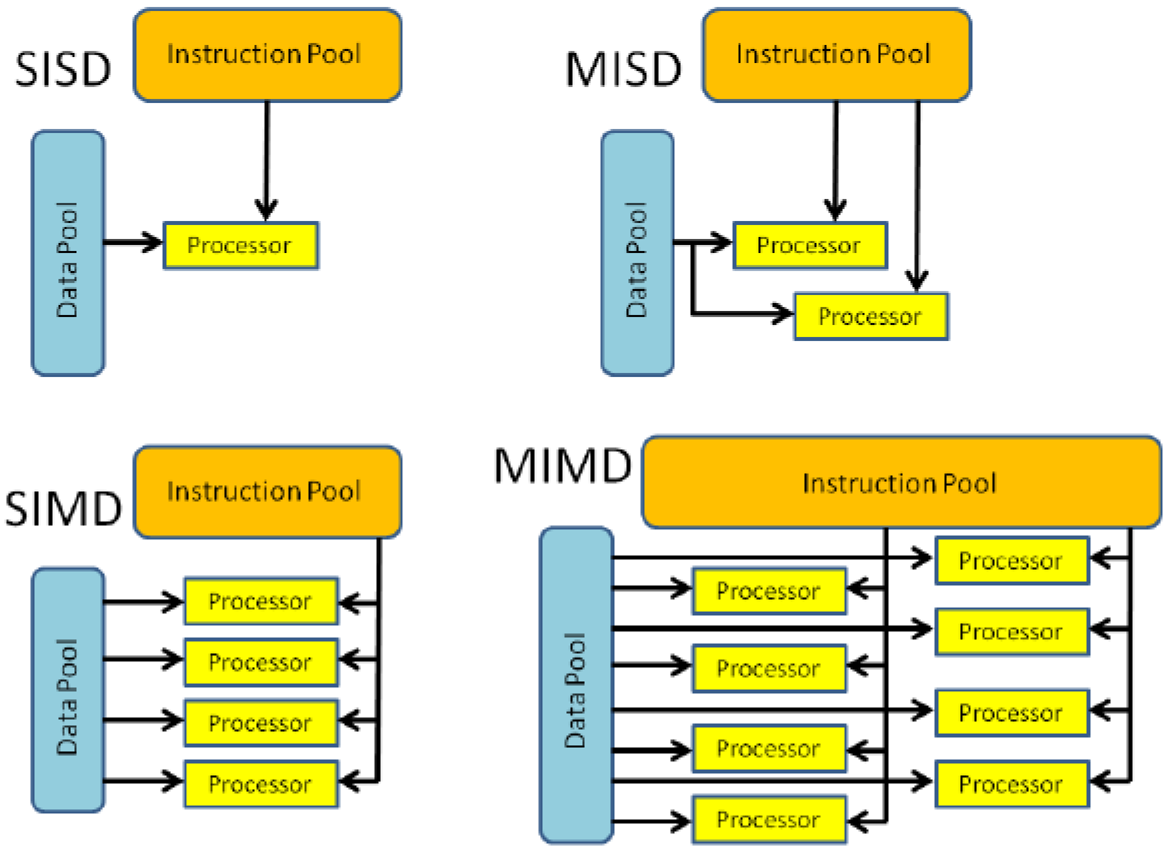
Lab session 1

HPC == Parallel Computers!

- Simultaneous use of multiple compute resources to solve a computational problem
  - Break down the problem
  - Execute concurrently
  - Overall control/coordination mechanism
- A number of different machines
  - Use Flynn's Taxonomy (1996) to classify them



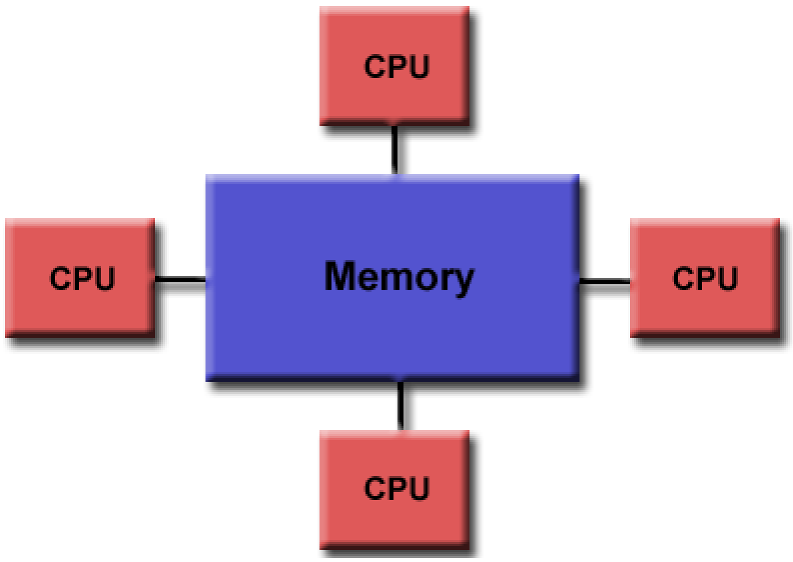
Overview of Flynn’s Computer Architectures

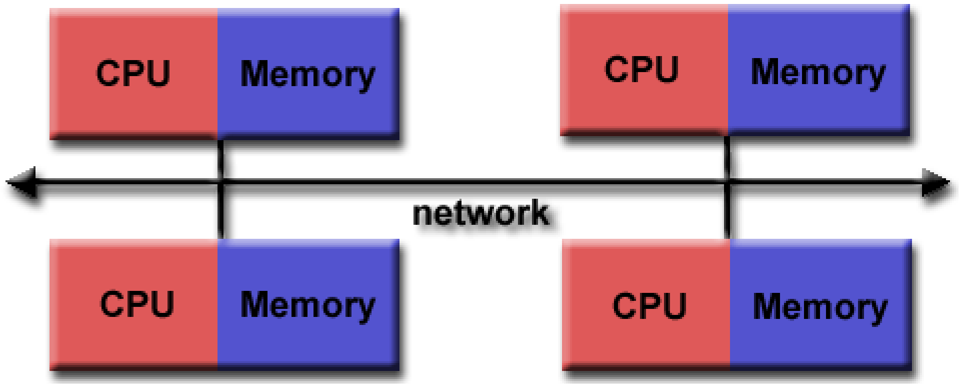


Projectile Monte-Carlo Trajectory Analysis Using a Graphics Processing Unit - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Flynns-Taxonomy-of-Computer-Architectures-13_fig1_268011284 [accessed 17 Feb, 2020]

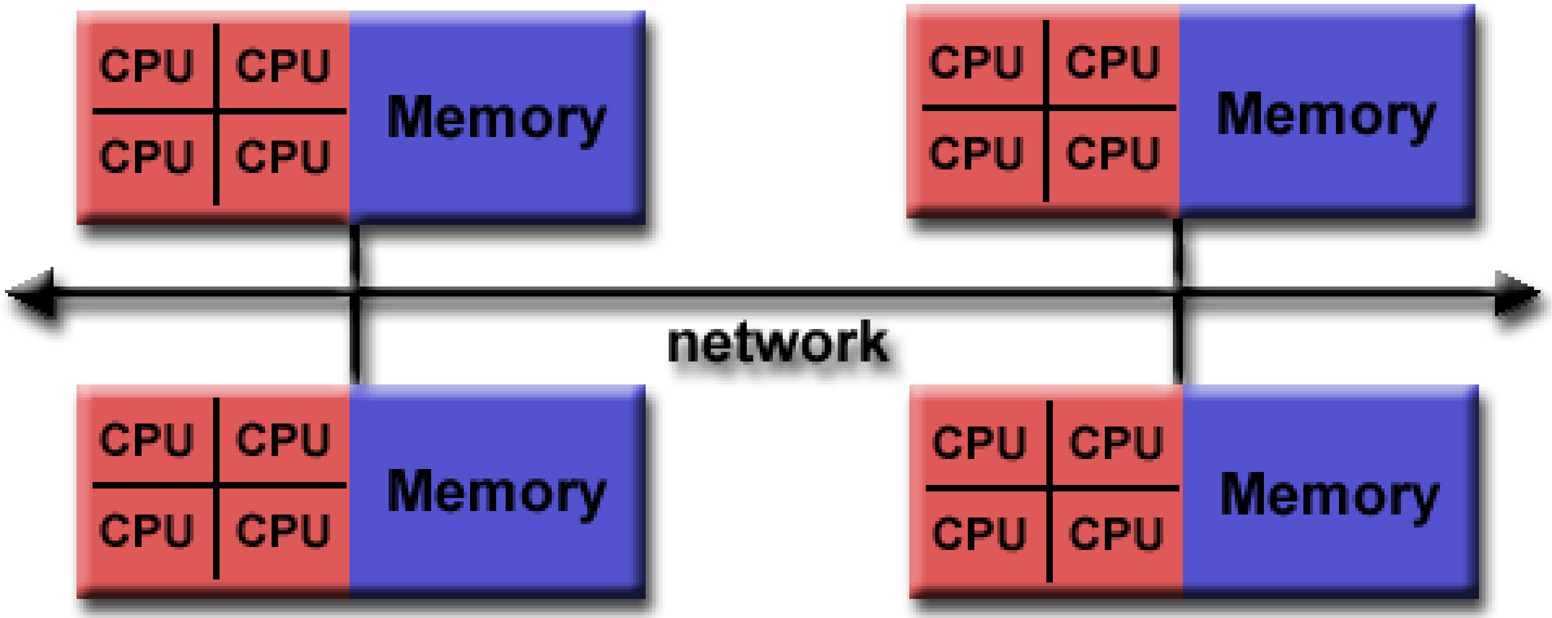
Parallel Computer Memory Architectures

- Memory architecture is probably the key classification criterion for modern parallel computers



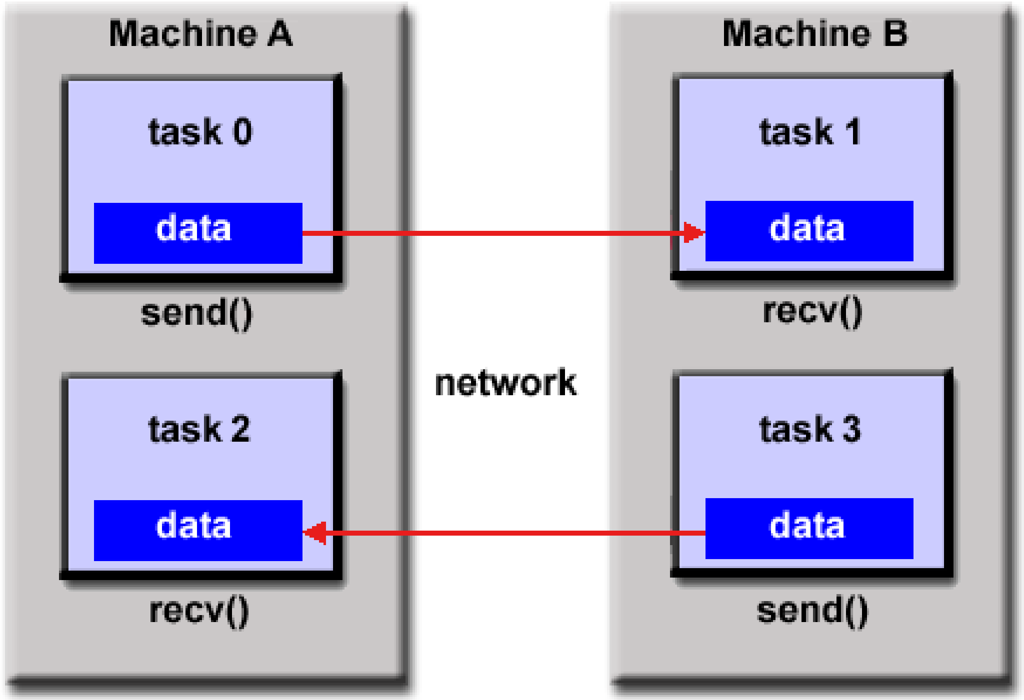


Hybrid Distributed-Shared Memory



Message passing paradigm

- The one that fits the distributed memory computer architecture
- A set of tasks that exchange data through communications by sending and receiving messages
  - Tasks use their own local memory during computation phase
  - Data transfer usually requires cooperative operations
  - Data distribution is manages by the programmer



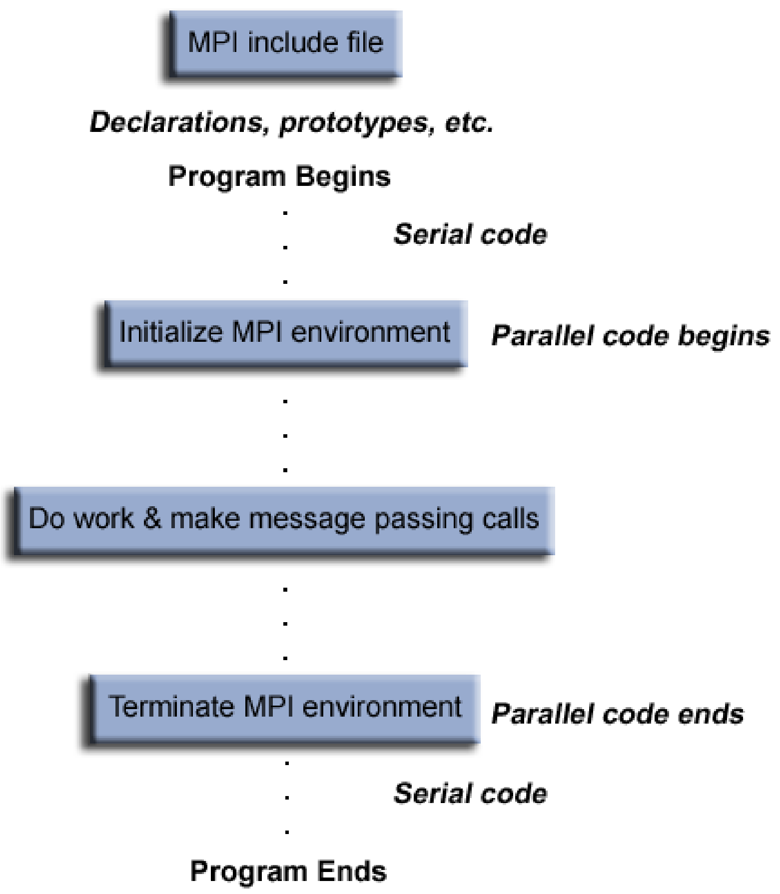
What is MPI?

- **M P I** = **Message Passing Interface**
- MPI is a ***specification*** NOT a library
- MPI primarily addresses the ***message-passing parallel programming model***
- The goal is to provide a widely used standard for writing message passing programs
- Interface specifications have been defined for C and Fortran90 language bindings
- Actual MPI library implementations differ in which version and features of the MPI standard they support.
  - Developers/users will need to be aware of this.

Why should I use MPI?

- International standard
- MPI evolves: MPI 1.0 was first introduced in 1994, most current version is MPI 3.3 (Nov. 2016)
- Available on almost all parallel systems (free MPICH, Open MPI used on many clusters), with interfaces for C/C++ and Fortran
- Supplies many communication variations and optimized functions for a wide range of needs
- Works both on distributed memory (DM) and shared memory (SM) hardware architectures
- Supports large program development and integration of multiple modules

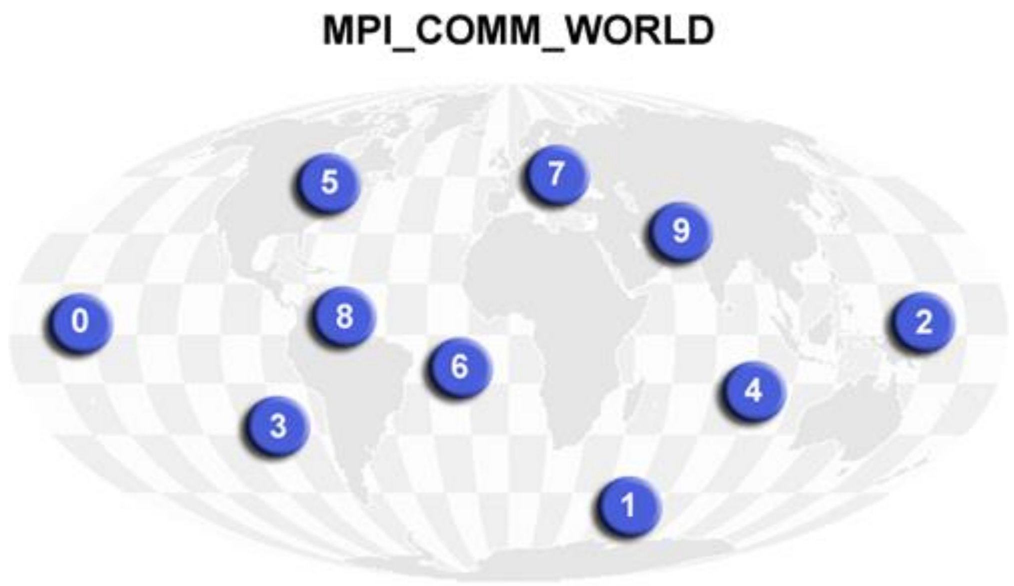
General MPI Program Structure




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

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x’;
MPI_Status Stat;

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

if (rank == 0) {
  dest = 1; source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
&Stat);
} else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
&Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
MPI_Finalize();
}
```

Communicators and groups

- MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.
- Most MPI routines require you to specify a communicator as an argument.
- For now, simply use **MPI_COMM_WORLD** wheneve r a communicator is required - it is the predefined communicator that includes all of your MPI processes.



Basic MPI routines

- MPI_Init: initialize MPI
- MPI_Comm_size: how many Processors?
- MPI_Comm_rank: identify the Processor
- MPI_Send: send data
- MPI_Recv: receive data
- MPI_Finalize: close MPI
- MPI_Get_processor_name: who am I?
- MPI_Wtime: wall clock time in seconds
- MPI_Abort: terminates MPI processes
- MPI_Get_version

MPI basic send/receive

Blocking

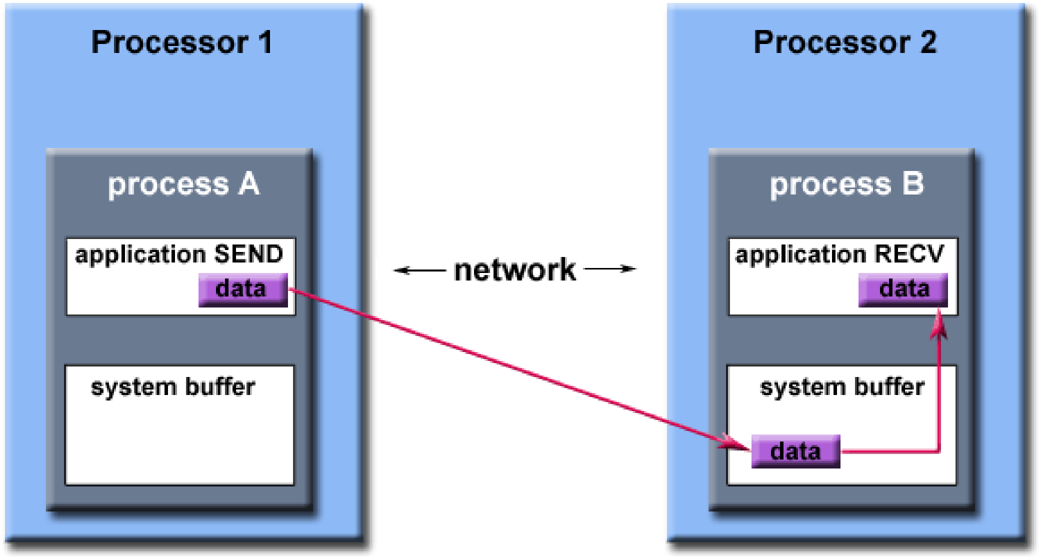
- Will only return when it is safe to modify the application buffer
- Send operation can be asynchronous or synchronous

Non-blocking

- Will return immediately
- Unsafe to modify the application buffer
- MPI_Wait operation needed

Blocking sends	MPI_Send(buffer,count,type,dest,tag,comm)
Non-blocking sends	MPI_Isend(buffer,count,type,dest,tag,comm,request)
Blocking receive	MPI_Recv(buffer,count,type,source,tag,comm,status)
Non-blocking receive	MPI_Irecv(buffer,count,type,source,tag,comm,request)

Buffering



Path of a message buffered at the receiving process

- Opaque to the programmer and managed entirely by the MPI library
- A finite resource that can be easy to exhaust
- Often mysterious and not well documented
- Able to exist on the sending side, the receiving side, or both
- Something that may improve program performance because it allows send - receive operations to be asynchronous.

Setting up the development environment

Ubuntu (Debian, GNU/Linux)

- Install the following packages:
  - openmpi-bin, libopenmpi-dev
- SSH client/server necessary for remote executions
- Other distributions: download & compile from http://www.open-mpi.org

Windows

- Download & execute Windows Binary Installer https://www.open-mpi.org/software/ompi/v1.6/ms-windows.php

Check that everything is OK!

- Compile with *mpicc*

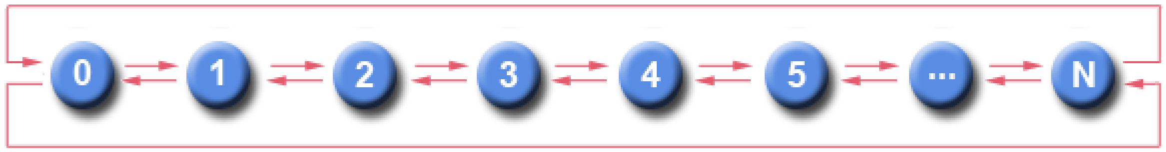
```
mpicc –o example0 example0.cc
```

- Execute with *mpirun*

```
mpirun -np 1 example0
```

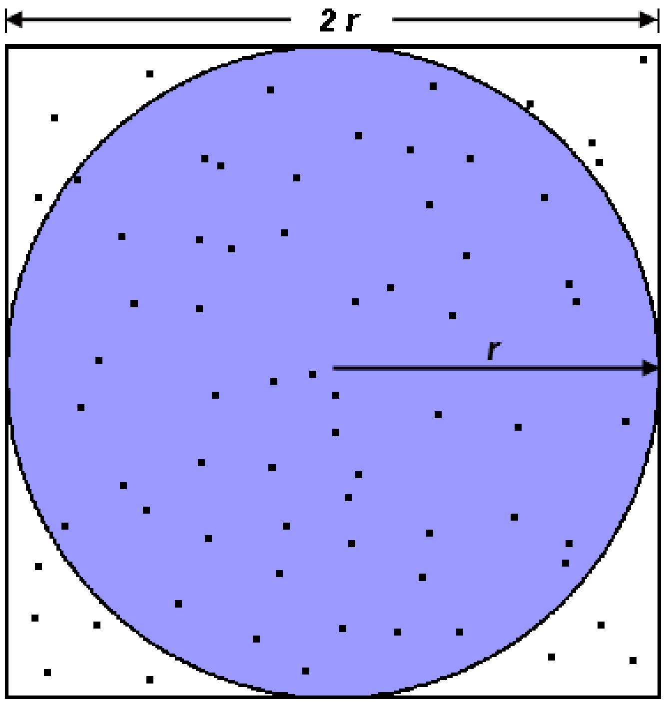
Exercises

- Simple use of send/receive primitives between two processes
- Extend the previous example to N processes organized as a ring
  - Use non-blocking message passing routines + wait



- Aproximate the value of PI using the Monte Carlo method using blocking or non-blocking primitives

Calculate the value of PI



$$A_S = (2r)^2 = 4r^2$$
$$A_C = \pi r^2$$
$$\pi = 4 \times \frac{A_C}{A_S}$$

```
npoints = 10000
circle_count = 0
p = number of tasks
num = npoints/p
find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle then
      circle_count = circle_count + 1
  end do

if I am MASTER receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)
else if I am WORKER
  send to MASTER circle_count
endif
```