Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Introduction to MPI

Lecture 4

István Reguly

reguly.istvan@itk.ppke.hu

Shared memory parallelism

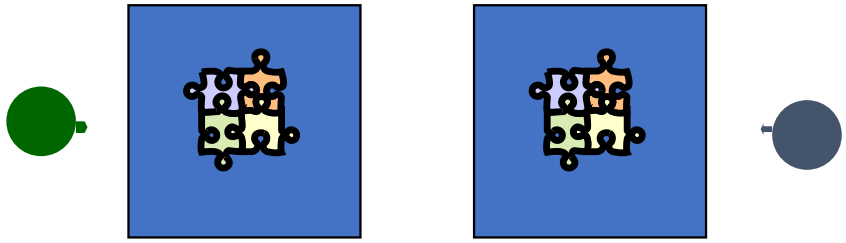
- Multiple people drawing on the same whiteboard
  - Same space
  - Can immediately see, modify what others wrote
  - But have to coordinate to avoid pushing or interfering with each other

Distributed memory parallelism

- People at home, sending letters to each other
  - Each in its own private space
  - No visibility of each other
  - Have to package, send, receive messages to each other

Distributed parallelism

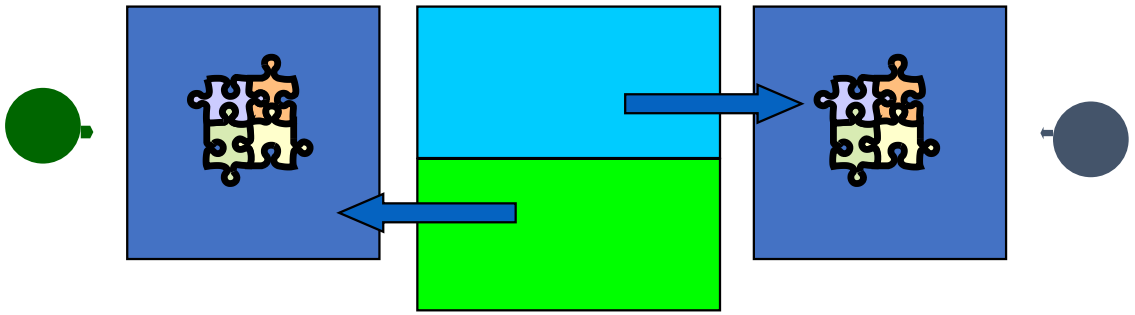
- Let’s try to split the puzzle pieces between two tables and solve it that way: there is now no contention, but the cost of communication is a lot higher – you will have to combine your results at the end



- Additionally, you have to split up your puzzle pieces (decompose) in the beginning. Depending on how much time you spent on a ”good split”, the communication later on may be more or less
  - Perfect split with pieces on the two sides of the final puzzle, vs. random split: it’s a lot easier at the end to put the two halves together, but a lot more work in the beginning

Distributed parallelism

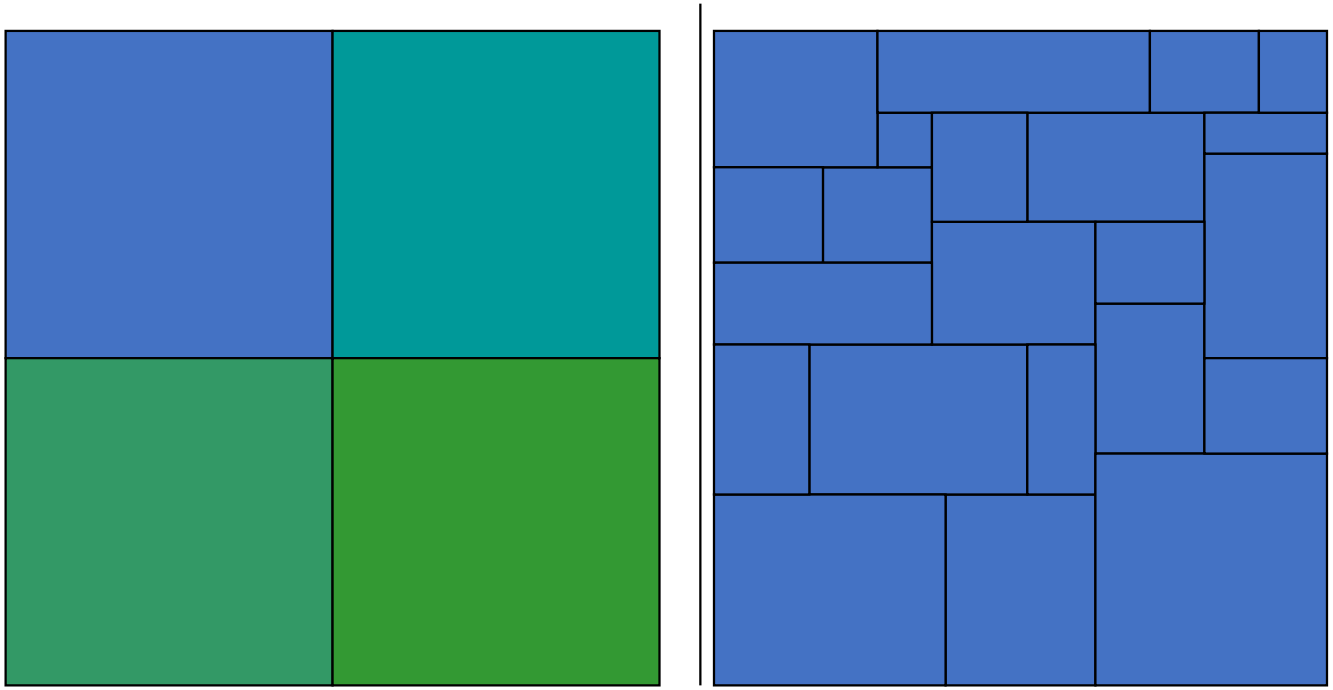
- The initial decomposition determines the load balancing: you try to give everyone the same amount of work to do. If the picture is half grass, half sky, it's easy, and you'll have to communicate only on the grass-sky boundary



- Some problems lend themselves to good decompositions and a good load balance.

Distributed parallelism

- Load balancing can get difficult, particularly if Jane is better at doing puzzles than Joe



Distributed vs. Shared parallelism

- So far we used shared memory parallelism
  - One application process starts, does serial setup, some code regions are parallelised with multiple threads, accessing the same memory
- Distributed memory parallelism in contrast
  - Launches several processes, they all execute the entire application code, and do not see each other's memory
  - Each process has its own unqiue identifier
    - This is used to split up work and coordinate
  - Instead of directly accessing each other's memory, they send messages

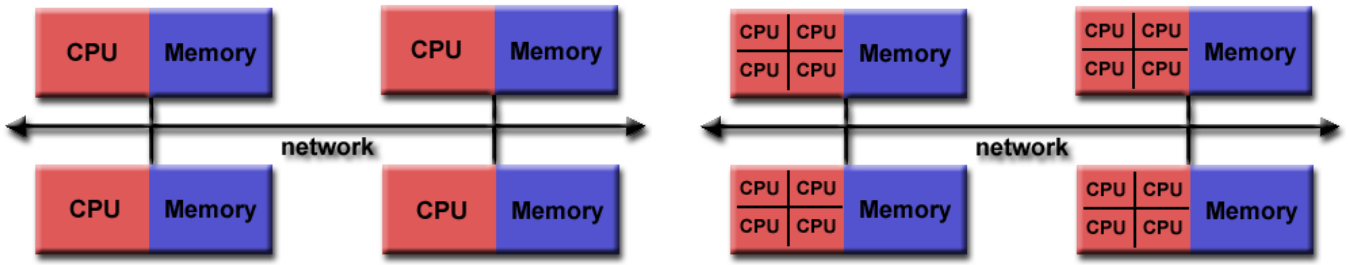
What is MPI?

An interface specification

- MPI=Message Passing Interface
- MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library – but rather the specification of what such a library should be.
- MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to another process through cooperative operations on each process
- Simply stated the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:
  - Practical, Portable, Efficient & Flexible

Programming Model

- Originally, MPI was designed for distributed memory architectures (1980s)



- As architecture trends changed, shared memory SMPs were combined over networks, creating hybrid distributed memory / shared memory systems
- MPI implementations adapted the libraries to handle both automatically, by handling different interconnects and protocols

What is a message?

- A message in most cases, is:
  - Some user-defined data – passed as a pointer, also specifying the size, and datatype
  - A sender (implicit)
  - A receiver (explicitly specified)
  - A „tag” – an index to help distiguish between messages

```
MPI_Send(buffer,count,type,dest,tag,comm)
```

MPI today

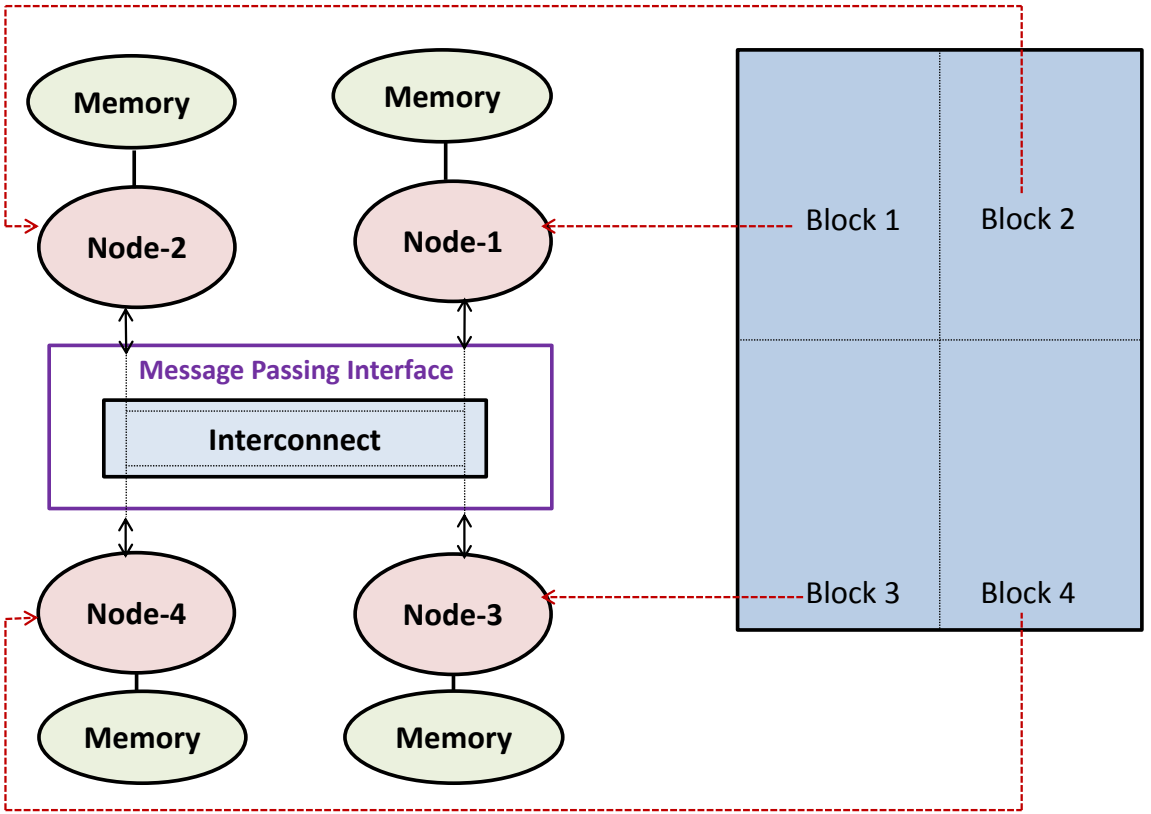
- Runs on virtually any platform:
  - Distributed memory
  - Shared memory
  - Hybrid
- The programming model however clearly remains a ***distributed memory model***, regardless of the underlying physical architecture of the machine.
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

Reasons for using MPI

- Standardization: MPI is the only message passing library that can be considered a standard, and it is supported on virtually all high performance computing platforms
- Portability: There is little or no need to modify your code when you port you application to a different platform that supports the MPI standard
- Performance Opportunities: vendor implementations should be able to exploit native hardware features to optimize performance.
- Functionality: there are over 430 routines defined in MPI-3
- Availability: variety of implementations available, most for free

How do we solve problems with MPI?

Divide & Conquer



OpenMPI

- OpenMPI is a open-source MPI-2 implementation
- Provides a set of compilers:
  - mpicc, mpic++, mpifortran (on Komondor, still cc/CC)
- Provides a tool to run MPI jobs:
  - mpirun (on Komondor, have to use srun)

General MPI program structure

MPI include file

Declarations, prototypes, etc.

Program Begins

. . . *Serial code*

Initialize MPI environment *Parallel code begins*

. . .

Do work & make message passing calls

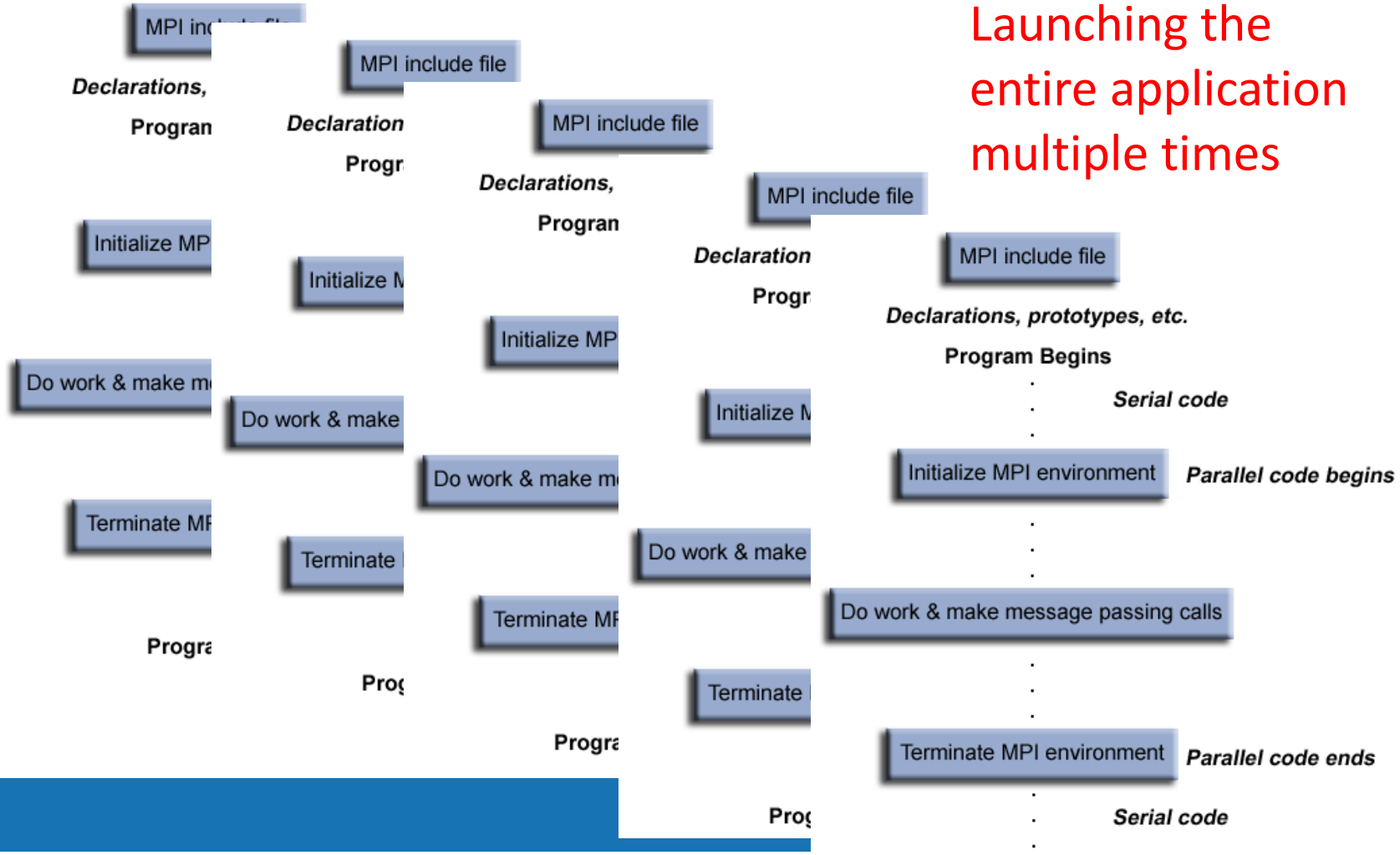
. . .

Terminate MPI environment *Parallel code ends*

. . . *Serial code*

Program Ends

General MPI program structure



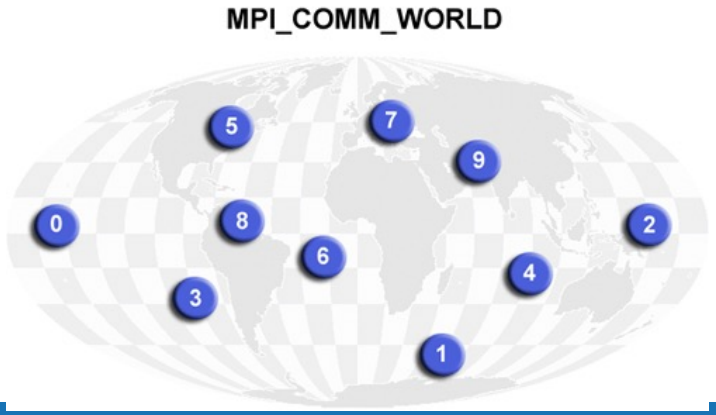
Launching the entire application multiple times

General MPI program structure

- Header file:
  - Required for all programs that make any MPI library calls:
    `#include <mpi.h>`
- Format of MPI Calls:
  - Case-sensitive
  - All functions have the prefix MPI_ (or PMPI_ for profiling)
  - Most return an error code (MPI_SUCCESS if successful)
    Format: `ret = MPI_Xxxx(params,...)`
    Example: `ret = MPI_Bsend(&buf,count,type,dest,tag,comm)`

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, we'll use MPI_COMM_WORLD; a predefined communicator that includes all processes




```
int size;
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Rank

- Within a communicator, every process has its own unique, integer identifier assigned by the system when the communicator is initialized. A rank is sometimes also called a “task ID”. Ranks are contiguous and begin at zero
- Used by the programmer to specify the source and destination messages. Often used conditionally by the application to control program execution (i.e. if rank==0 do this, else do that)

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

Error handling

- Most MPI routines include a return/error code parameter
- According to the MPI standard, the default behaviour of an MPI call is to abort if there is an error. This means you will probably not be able to capture a return code other than MPI_SUCCESS
- The standard does provide means to override this default error handler. More details here.

Environment management routines

- MPI_Init(&argc,&argv)
  - Initializes the MPI execution environment. Must be called in every MPI program, before calling any other MPI functions, and can only be called once.
- MPI_Comm_size(comm, &size)
- MPI_Comm_rank(comm, &rank)
- MPI_Abort(comm, errorcode)
  - Terminates all processes associatied with the communicator, although in most implementations it aborts all processes.
- MPI_Get_processor_name(&name,&length)
  - Returns the processor name. The buffer has to be at least MPI_MAX_PROCESSOR_NAME long

Environment management routines

- MPI_Initialized(&flag)
  - Only exception to the rule that MPI_Init has to be called first – libraries can check if MPI has been initialized
- MPI_Wtime()
  - Return the number of elapsed seconds on the calling processor
- MPI_Finalize()
  - Terminates the MPI execution environment. Should be the last MPI routine called.

Exercise - Hello World

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

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

rc = MPI_Init(&argc,&argv);
if (rc != MPI_SUCCESS) {
  std::cout << “Error starting MPI program. Terminating.\n";
  MPI_Abort(MPI_COMM_WORLD, rc);
  }

MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Get_processor_name(hostname, &len);
std::cout << "Number of tasks= “<< numtasks << “ My rank= “<<rank<<
                    “ Running on “<<hostname<<“\n“;

MPI_Finalize();
}
```

Exercise – Hello world

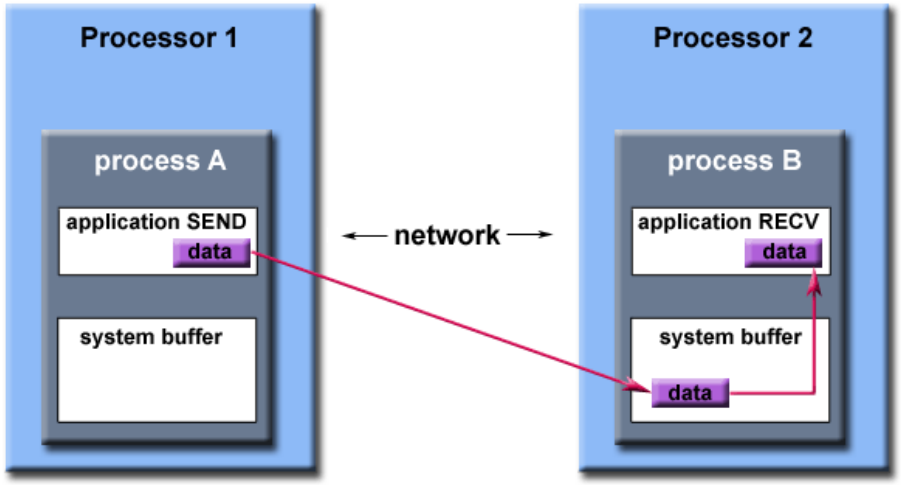
- Take a look at mpi_hello.cpp
- Compile with:
  - CC mpi_hello.cpp -o mpi_hello
- **Run with:**
  - salloc --reservation=p_covidpre_118 -p cpu -c 4 --mem-per-cpu=2000 --time=1:00:00
  - srun --ntasks=4 ./mpi_hello
    - **Launches the application 4x, then connects them**
    - Try with more!

Types of point-to-point communications

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation
- There are different types of send and receive routines used for different purposes:
  - Synchronous send
  - Blocking send/blocking receive
  - Non-blocking send/non-blocking receive
  - Buffered send
  - Combined send/receive
  - “Ready” send
- Any type of send routine can be paired with any kind of receive routine
- Plus operations such as waiting for a message, probing to see whether it has arrived

Buffering

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case... So MPI needs to store data
  - Send occurs 5 seconds before the receive – where is the message in the meantime?
  - Multiple sends arrive at the same receiver, which can only receive one at a time – where are the other four?



Path of a message buffered at the receiving process

Blocking vs non-blocking

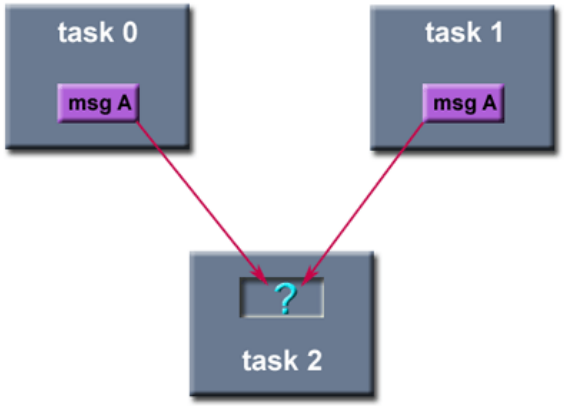
- Most MPI routines can be used both ways
- Blocking:
  - A blocking send routine will only ”return” after it is safe to modify the application buffer (your send data) – modifications will not affect the data intended for the receive task (does not mean it was received)
  - A blocking send can be synchronous which means there is a handshake occurring with the receive task to confirm a safe send
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery
  - A blocking receive only “returns” after the data has arrived and is ready for use by the program

Non-blocking

- Non-blocking send and receive routines behave similarly – they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of the message
- Non-blocking operations simply “request” the MPI library to perform the operation when it is able. The user can not predict when that will happen
- It is unsafe to modify the application buffer (your variable) until you know for a fact that the requested non-blocking operation was actually performed – see wait operations
- Non-blocking operations are primarily used to overlap computation with communication and exploit possible performance gains

Order and fairness

- Order: MPI guarantees that messages will not overtake each other
  - From the same sender to the same destination
  - If a receiver posts a two receives looking for the same message, the first will receive before the second
- Fairness: MPI does not guarantee fairness
  - Two sends, one receive: only one completes, no guarantees which



Blocking routines

- Blocking send

```
MPI_Send(buffer,count,type,dest,tag,comm)
```

- Blocking receive

```
MPI_Recv(buffer,count,type,source,tag,comm,status)
```

- Buffer: the *address* that references data to be sent or to be received to
- Data count: number of data elements to be read from *Buffer*
- Data type: MPI_INT/MPI_FLOAT/MPI_BYTE, etc...
  - Can create custom data types - later

Blocking routines

```
MPI_Recv(buffer,count,type,source,tag,comm,status)
MPI_Send(buffer,count,type,dest,tag,comm)
```

- Destination: *for send routines* - the process where a massage should be delivered (rank)
- Source: *for receive routines* – the originating process' rank. May be MPI_ANY_SOURCE
- Tag: arbitrary non-negative integer that uniquely identifies a message. For a receive, can use MPI_ANY_TAG
- Communicator: the communication context or set of processes, usually MPI_COMM_WORLD
- Status: for a receive, indicates the the source and the tag (MPI_Status struct), can use MPI_STATUS_IGNORE

Blocking routines

Sending 1 int

```
if (rank == 2) {
  int buffer = 23;
  MPI_Send(&buffer,1,MPI_INT,3,22,MPI_COMM_WORLD);}

if (rank == 3) {
  int rbuffer;
  MPI_Recv(&rbuffer,1,MPI_INT,2,22, MPI_COMM_WORLD,MPI_STATUS_IGNORE);}
```

Sending multiple ints

```
if (rank ==2) {
  std::vector<int> buffer(10); //size 10
  MPI_Send(&buffer[0],10,MPI_INT,3,22,MPI_COMM_WORLD);}

if (rank ==3) {
std::vector<int> rbuffer(10); //size 10
MPI_Recv(&rbuffer[0],10,MPI_INT,2,22, MPI_COMM_WORLD,MPI_STATUS_IGNORE);}
```

Exercise

- Make a copy of your hello world program
- Modify it to do the following: after ranks determining their own ID and the number of processes, have each task determine a unique partner to send/receive with. E.g.

```
if (rank < numtasks/2) partner = numtasks/2 + rank;
else if (rank >= numtasks/2) partner = rank - numtasks/2;
```

- Each task sends its partner a single integer (a random number), each task receives from its partner a single integer (the partner's random number)
- Once done, each task shall print something like "task X (randX) is partner with task y (randY)"

Combined Send & Receive

- One MPI call to send and receive a message:

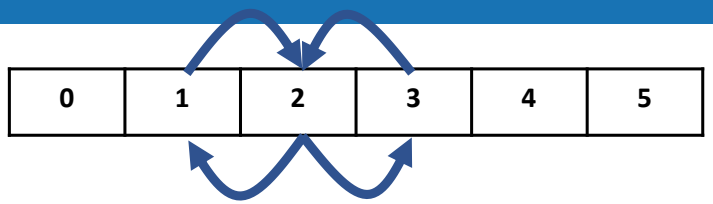
```
MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                                    int dest, int sendtag,
                          void *recvbuf, int recvcount, MPI_Datatype
recvtype,
                                    int source, int recvtag,
                                    MPI_Comm comm, MPI_Status *status)
```

Non-blocking routines

```
MPI_Isend(buffer,count,type,dest,tag,comm,request)
MPI_Irecv(buffer,count,type,source,tag,comm,request)
```

- Request: used by non-blocking send/receive operations. Since these operations may return before the communications even start, the system issues a unique “request number”. We can use it later to determine the completion of the non-blocking operation.
  - Can be passed to MPI_Wait/any/all/some

Non-blocking example




```
MPI_Request reqs[4];

prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

      {  do some work that does not use what you receive  }

MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE);
```

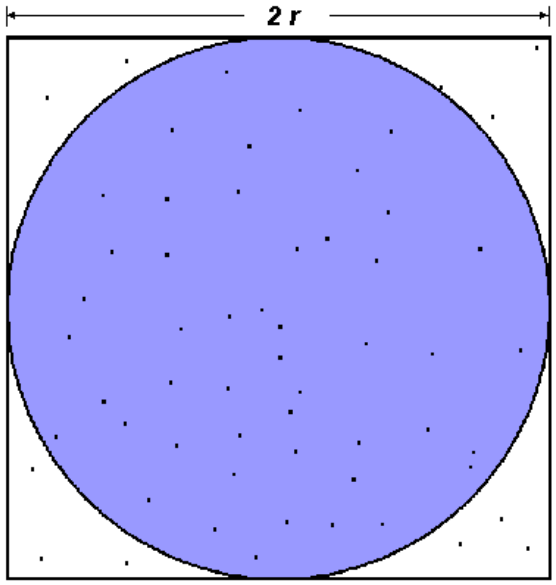
Point-to-point communication

- The value of Pi can be calculated with the Monte-Carlo estimate

```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers
             between -0.5 and 0.5
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```



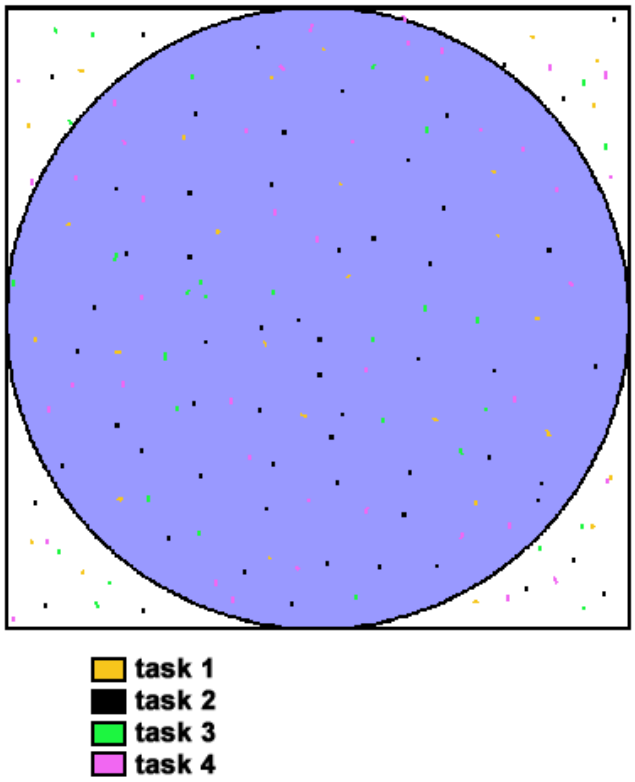
$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

Embarrassingly parallel solution

- Break the loop iterations into chunks that can be executed by different tasks simultaneously
- Each task can do its work without requiring any information from the other – no data dependencies
- Master task receives results from other tasks using send/receive point-to-point operations




```
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 -0.5 and 0.5
  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
```

Exercise

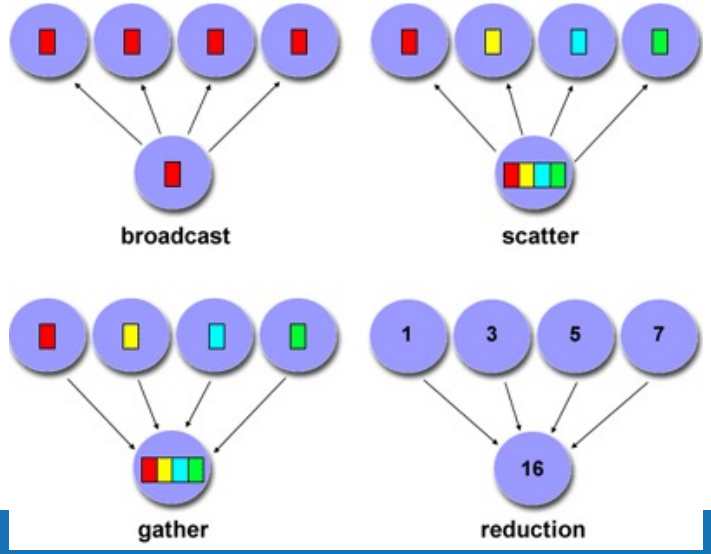
- Convert pi approximation code to use MPI
- Compile with CC comp_pi.cpp -o comp_pi
- Run with
- salloc --reservation=p_covidpre_118 -p cpu -c 4 --mem-per-cpu=2000 --time=1:00:00
  srun --ntasks=1 ./mpi_pi_reduce
  srun --ntasks=2 ./mpi_pi_reduce
  srun --ntasks=4 ./mpi_pi_reduce

Collective communications

- Collective communications must involve all processes within the scope of the communicator
  - Otherwise unexpected behaviour
- Types:
  - Synchronisation: processes wait until all members of the group have reached the synchronisation point
  - Data movement: broadcast, scatter/gather, all-to-all
  - Collective computations: reductions – one member of the group collects data from the other members and performs an operation on that data

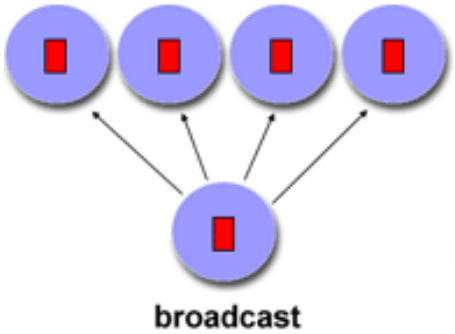
Collective communications

- Collective communication routines do not take message tag arguments
- To work on subsets of processes, one must create a communicator for those
- We mostly use blocking collectives, but as of MPI-3 there are non-blocking operations as well



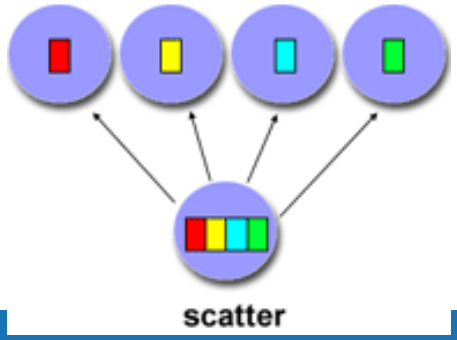
Collective communications

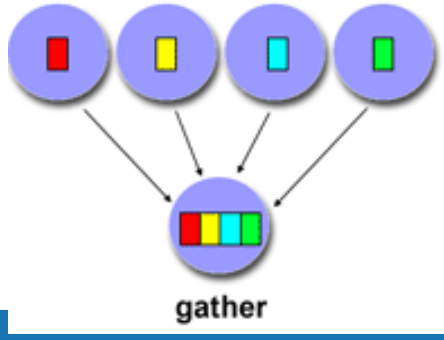
- MPI_Barrier(comm): synchronisation across all ranks of a communicator
- MPI_Bcast(buffer,count,datatype,root,comm): data movement. Broadcasts a message from the process with rank ”root” to all other processes



Collective communications

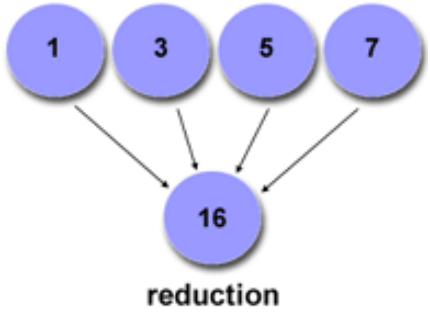
- MPI_Scatter(sendbuf,sendcount,sendtype,recvbuf, recvcount,recvtype,root,comm): distributes distinct messages from a single source to each task in the group
- MPI_Gather: inverse of scatter
- MPI_Allgather: gather to every rank





Collective communications

- MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm): applies reduction on all tasks placing the result on “root”. Ops can be: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, etc.
- MPI_Allreduce: reduction with placement of result on all ranks
- MPI_Reduce_scatter: does a reduction on a vector, then scatters the elements of the vector



Reductions

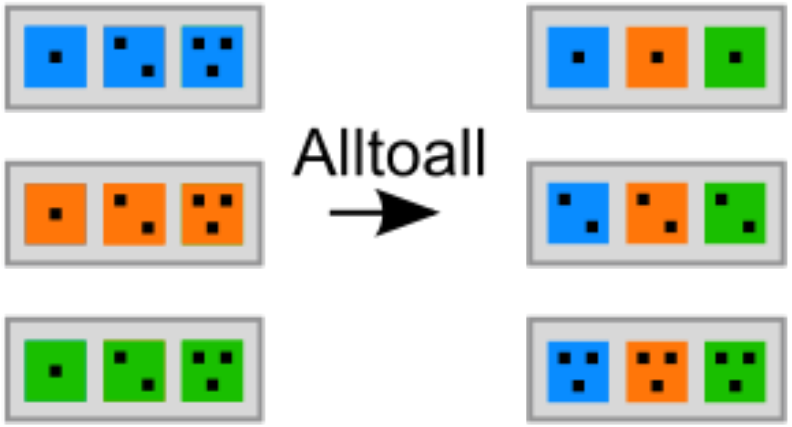
```
MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- All processes send values (`sendbuf`), of length `count` and type `datatype`
- These values are combined with the operations `op`
  - `MPI_MIN, MPI_MAX, MPI_SUM, MPI_PROD, etc.`
- The result is placed in `recvbuf` for each process

```
double my_partial_sum = ...;
double global_sum;
MPI_Allreduce(&my_partial_sum, &global_sum, 1,
              MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
//Now global_sum = sums of partial_sums of each process
```

Collective communications

- MPI_Alltoall: all ranks exchange unique messages



Exercise

- Modify the Pi estimation algorithm to use MPI_Reduce