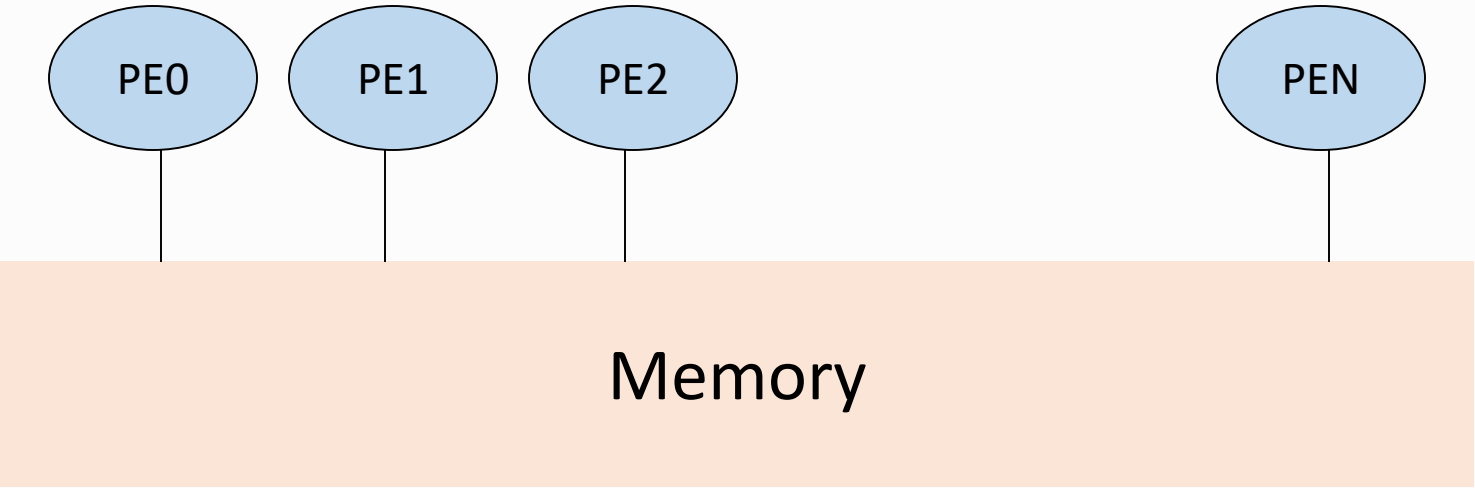
Message Passing in Distributed Systems

Distributed memory machines and communication

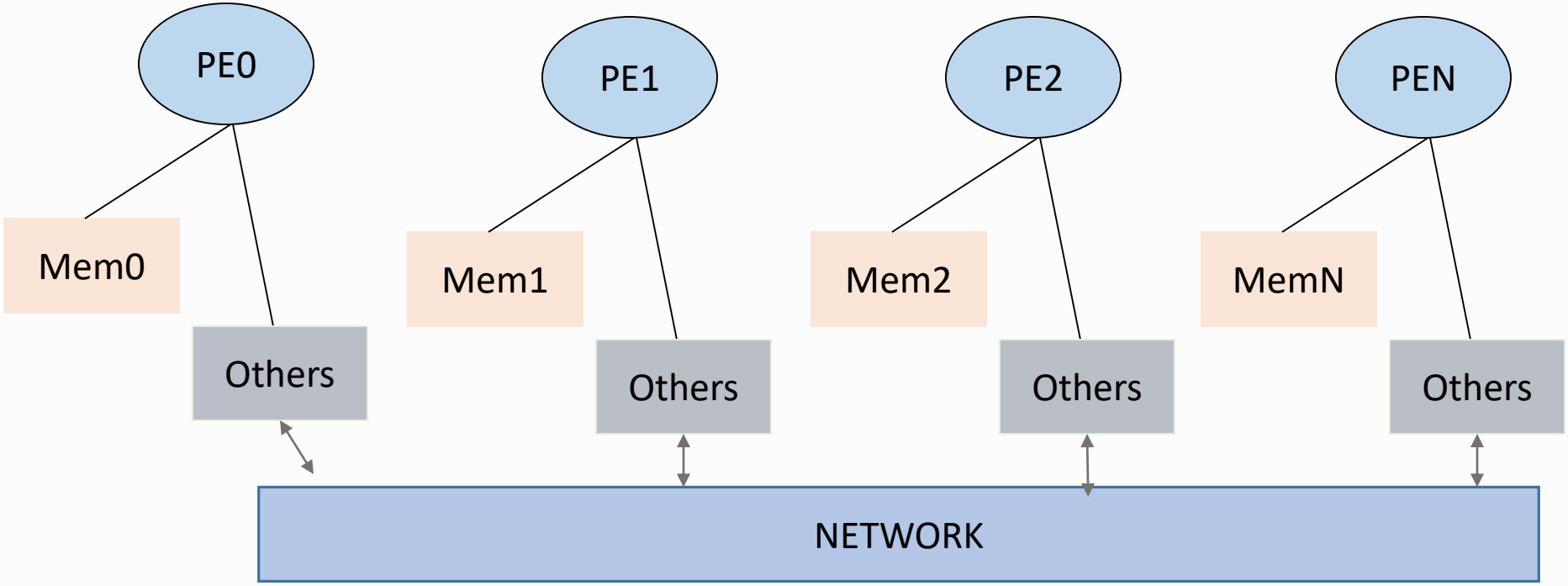
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Shared Memory Systems



- Advantages: Easy to parallelize, all data available nearby
- Limitations: Complex and expensive to build, restricted to certain system sizes

Distributed Memory Systems



- Locality of data is exposed, cheap to build
- But what are the negative points?

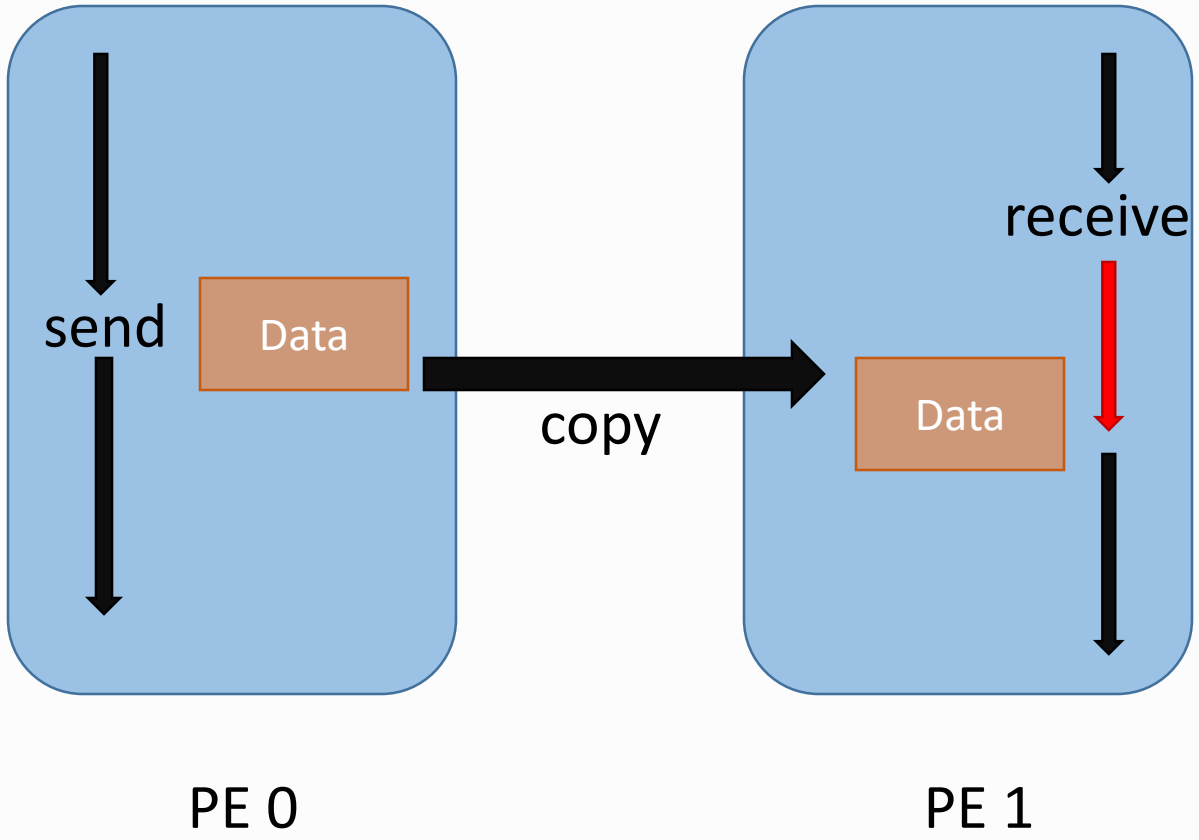
Message passing

- Exemplified by MPI: (**M**essage **P**assing **I**nterface)
- Work unit: Processes
- Data units:
  - Decomposed, so that each process has its own data unit
  - No shared data
  - Coordination by: exchanging “messages” via send/recv calls
  - Analogy: mail

Anatomy of message passing

- Typical clusters today:
  - Ethernet or more sophisticated network card and a interconnect
  - Messages are sent as packets over the network
- Co-processors
  - Network Interface Card (NIC) is an example
  - Offload work of communication to co-processors

Message Passing

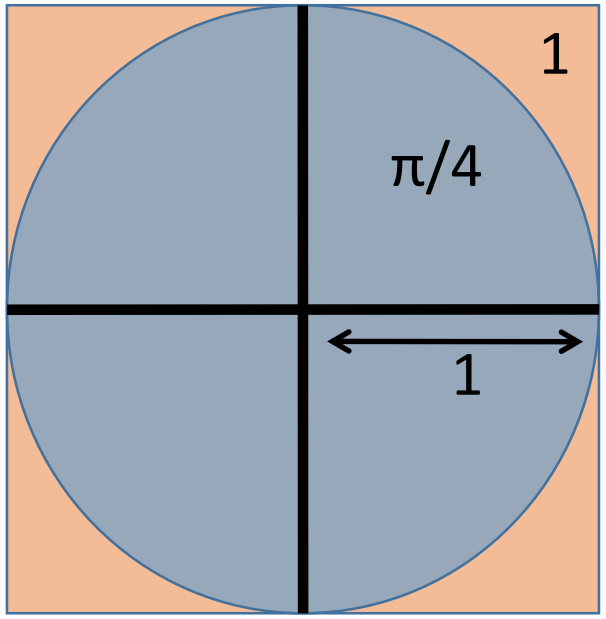


Basic Message Passing

- We will describe a hypothetical message passing system
  - With just a few calls that define the model
  - Later, we will look at real message passing models (e.g. MPI), with a more complex sets of calls
- Basic calls:
  - send(int proc, int tag, int size, char *buf);
  - recv(int proc, int tag, int size, char * buf);
    - recv may return the actual number of bytes received in some systems
  - tag and proc may be wildcarded in a recv:
    - recv(ANY, ANY, 1000, &buf);
- Broadcast:
- Other global operations (reductions)

Pi with message passing

```
int count, c1, i;
main() {
  Seed s = makeSeed(myProcessor);
  for (i=0; i<100000/P; i++) {
    x = random(s);
    y = random(s);
    if (x*x + y*y < 1.0)
      count++;
  }
  send(0, 1, 4, &count);
  if (myProcessorNum() == 0) {
    globalCount = 0.0;
    for (i=0; i<maxProcessors(); i++) {
      recv(i, 1, 4, c);
      globalCount += c;
    }
    printf("pi=%f\n", 4*globalCount/100000);
  }
} /* end function main */
```



Collective calls

- Message passing is often, but not always, used for SPMD style of programming:
  - SPMD: Single process multiple data
  - All processors execute essentially the same program, and same steps, but not in lockstep
- All communication is *almost in lockstep*
- **Collective calls:**
  - global reductions (such as max or sum)
  - syncBroadcast (often just called broadcast):
    - syncBroadcast(whoAmI, dataSize, dataBuffer);
      - whoAmI: sender or receiver

Basic MPI

Sending and Receiving Messages

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Standardization of message passing

- Historically:
  - nxlib (On Intel hypercubes)
  - ncube variants
  - PVM
  - Everyone had their own variants
- MPI standard:
  - Vendors, ISVs, and academics got together
  - with the intent of standardizing current practice
  - Ended up with a large standard
  - Popular, due to vendor support
  - Support for
    - communicators: avoiding tag conflicts, ..
    - Data types:
    - ..

A Simple subset of MPI

- These six functions allow you to write many programs:
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv

MPI Init and Finalize

- MPI_Init(int argc, char **argv)
- Initializes the MPI library.
- MPI_Finalize()
- Terminates use of MPI library.

All MPI calls must occur between these two, temporally.

MPI Process Identification

MPI_Comm_size(comm, &size)

Determines the number of processes.

MPI_Comm_rank(comm, &pid)

Pid is the process identifier of the caller.

A simple MPI program

```
#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("Hello world! I'm %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

MPI Basic Send

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

buf: address of send buffer

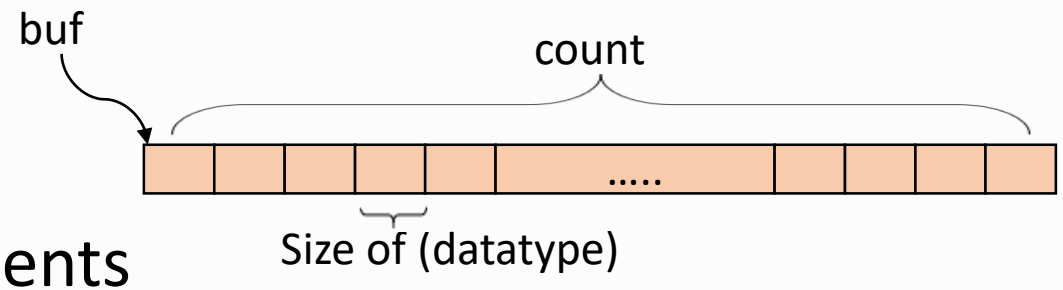
count: number of elements

datatype: data type of send buffer elements

dest: process id of destination process

tag: message tag (ignore for now)

comm: communicator (ignore for now)



MPI Basic Receive

MPI_Recv(buf, count, datatype, source, tag, comm, &status)

- buf: address of receive buffer
- count: size of receive buffer in elements
- datatype: data type of receive buffer elements
- source: source process id or MPI_ANY_SOURCE
- tag and comm: ignore for now
- status: status object

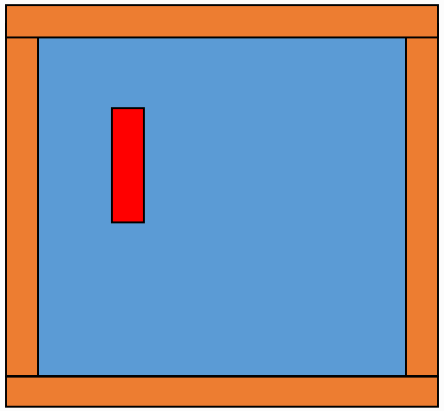
Running a MPI Program

- Example: `mpirun -np 2 hello`
- Interacts with a daemon process on the hosts.
- Causes a process to be run on each of the hosts.

Other Operations

- Collective Operations
  - Broadcast
  - Reduction
  - Scan
  - All-to-All
  - Gather/Scatter
- Support for Topologies
- Buffering issues: optimizing message passing
- Data-type support

Example : Jacobi relaxation



Red and blue boundaries held at fixed values (say temperature)

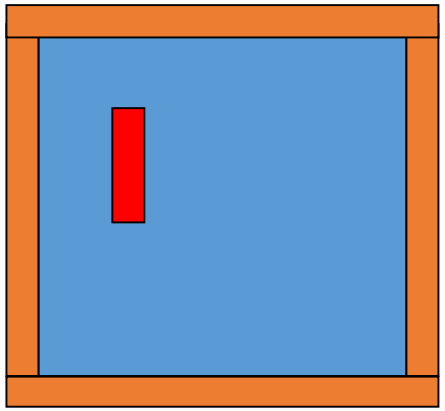
Discretization: divide the space into a grid of cells.

For all cells except those on the boundary: iteratively compute temperature as average of their neighboring cells'

```
Pseudocode:

A, Anew: NxN 2D-array of (FP) nos.

Begin Loop (how many times?)

  for each I = 1, N

    for each J between 1, N

      Anew[I,J] = average of 4
neighbors and itself.

  Swap Anew and A

End Loop
```

How to parallelize?

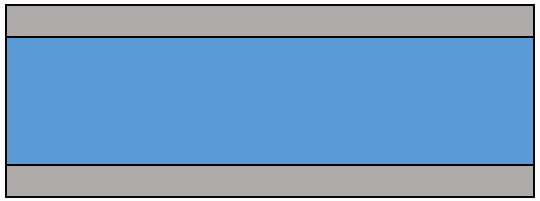
- Decide to decompose data:
  - What options are there? (e.g. 16 processors)
    - Vertically
    - Horizontally
    - In square chunks
  - Pros and cons:



- Identify communication needed
  - Let us assume we will run for a fixed number of iterations
  - What data do I need from others?
  - From whom specifically?
  - Reverse the question: Who needs my data?
  - Express this with sends and recvs..

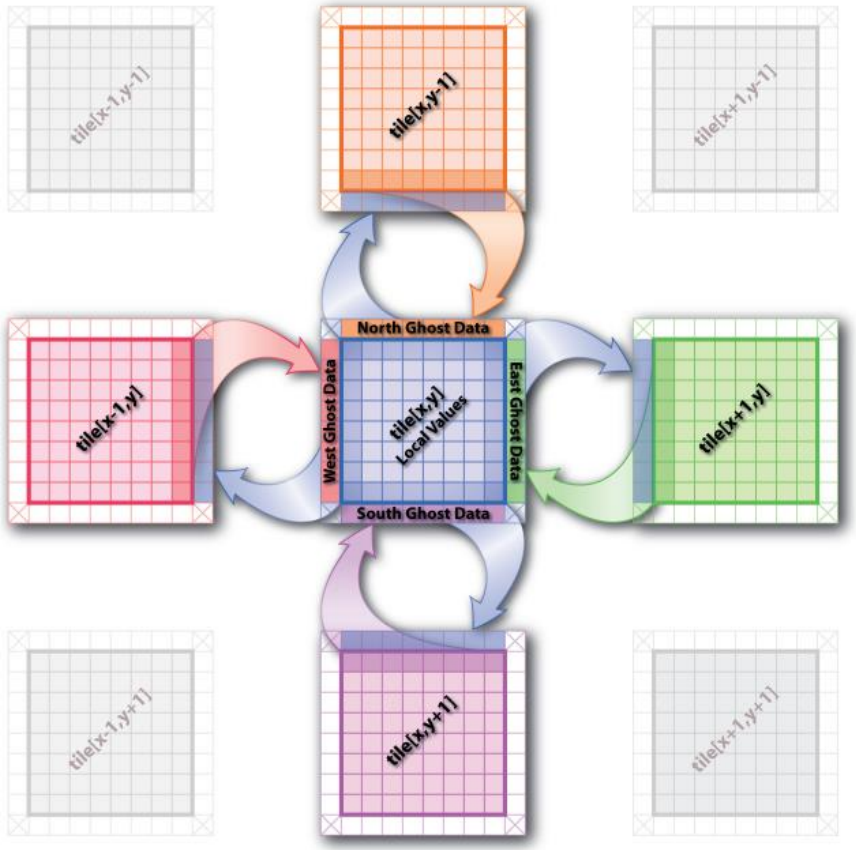
Ghost cells: a common apparition

- The data I need from neighbors
  - But that I don't modify (therefore "don't own")
- Can be stored in my data structures
  - So that my inner loops don't have to know about communication at all..
  - They can be written as if they are sequential code.



Ghost Cells

5 Point Stencil

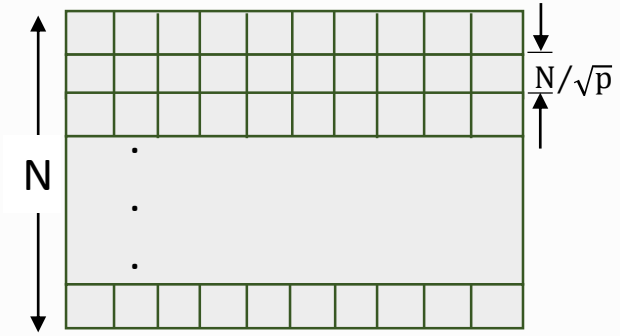


```
Assume P ranks gridded as p1 x p2
For each rank proc,tag,size,buf
myx = myrank / p2; myy = myrank % p2;
Begin Loop
  if(myx != 0) send(RANK(myx-1, myy), 0, N, northData)
  if(myx != p1-1) send(RANK(myx+1, myy), 1, N, southData)
  if(myy != 0) send(RANK(myx, myy-1), 2, N, eastData)
  if(myx != p2-1) send(RANK(myx, myy+1), 3, N, westData)

  if(myx != p1-1) recv(RANK(myx+1, myy), 0, N, newSouthData)
  if(myx != 0) recv(RANK(myx-1, myy), 1, N, newNorthData)
  if(myx != p2-1) recv(RANK(myx, myy+1), 2, N, newWestData)
  if(myy != 0) recv(RANK(myx, myy-1), 3, N, newEastData)
  For data on me
    update Anew using A and received data
  End For
End Loop
```

Exercise: code this with MPI

Comparing the decomposition options

- What issues?
  - Communication cost is the main one here
  - How many messages? How many bytes?
- Options:
  1. Each process owns a block of rows
  2. Each process owns a tile
- Compare the options:
  - Communication volume (number of bytes)
  - Number of messages: 2 vs 4
  - Restrictions of machine configuration: tile decomposition needs a square number of processors
  - Ease of coding
  - Cache performance for copying



Send/Receive Variants

Buffers, copies, and waits

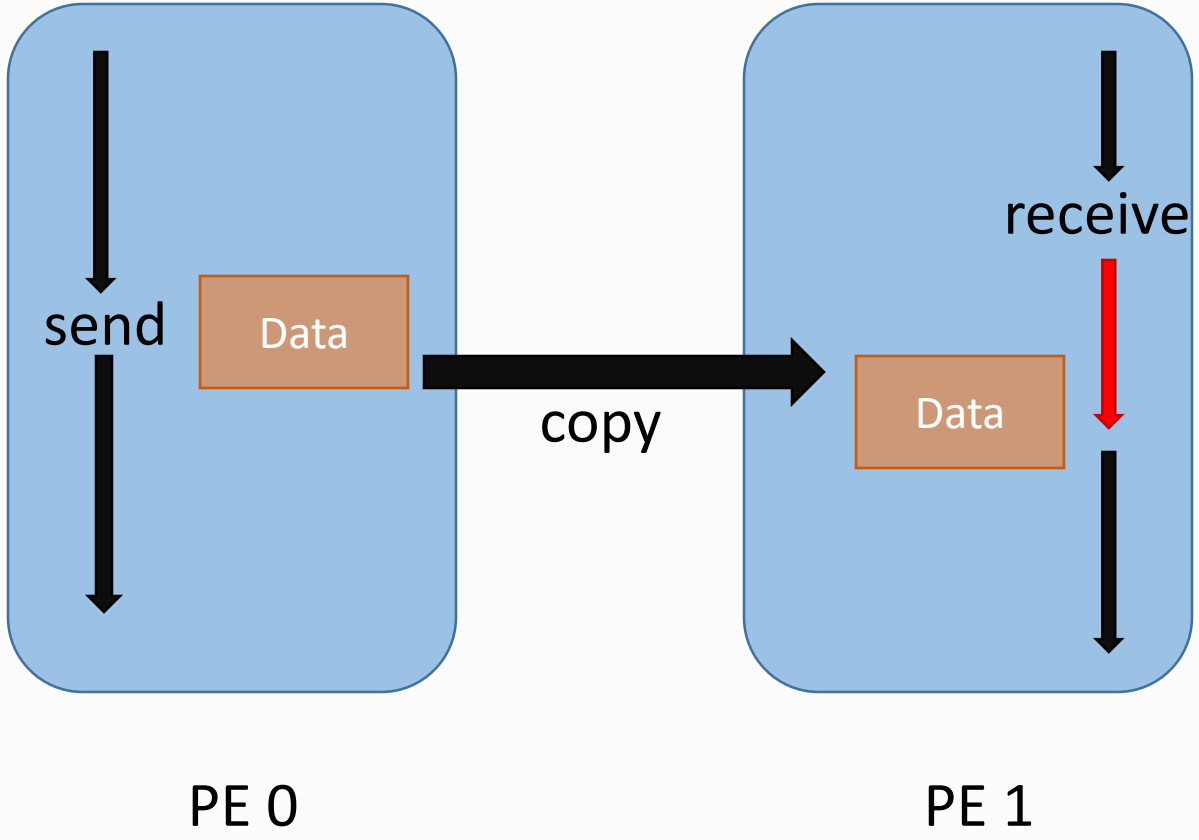
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Implementation and efficiency issues

- It is useful to understand some of these to see the motivation for the variety of send/recv call variants
- Issues:
  - Data copying cost
  - Buffer availability and allocation
  - Packetization
  - Tag matching
  - Progress engine

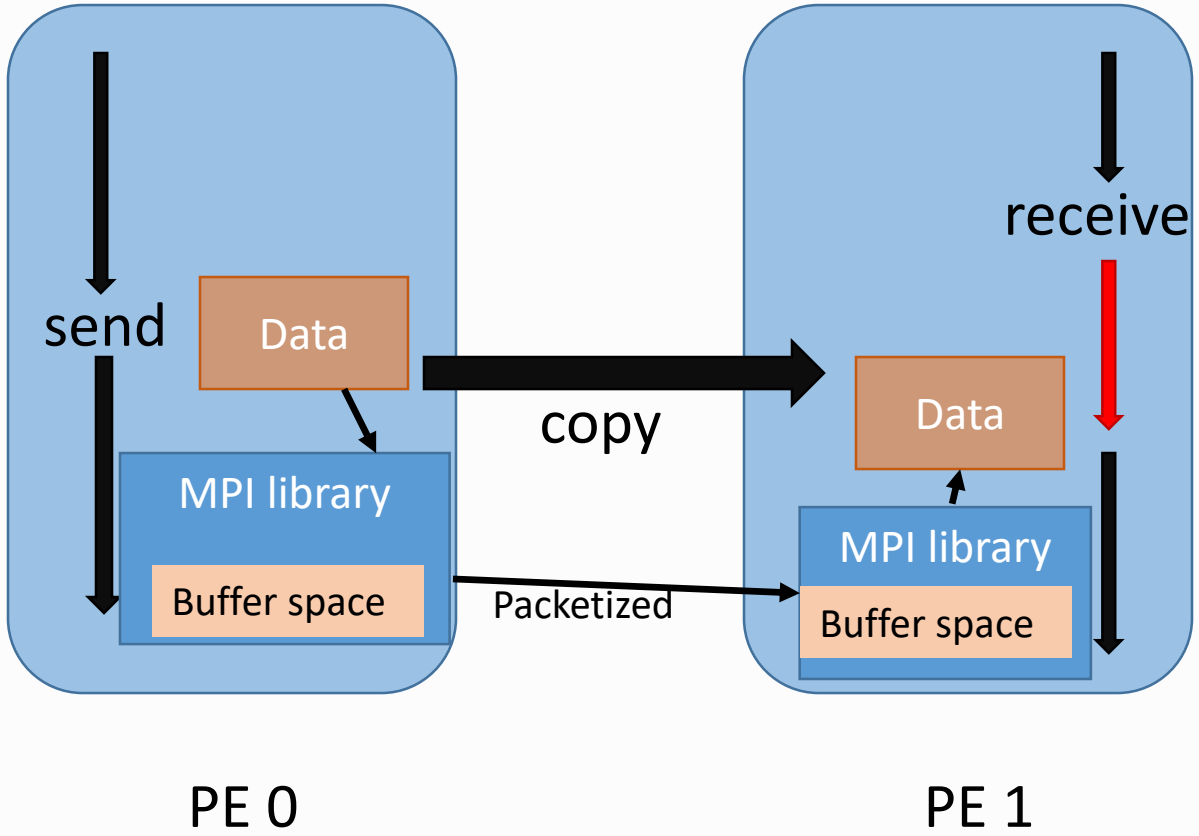
Recall: Message Passing

Involves copying data from user's data space on the source processor to user's data space on the other processor



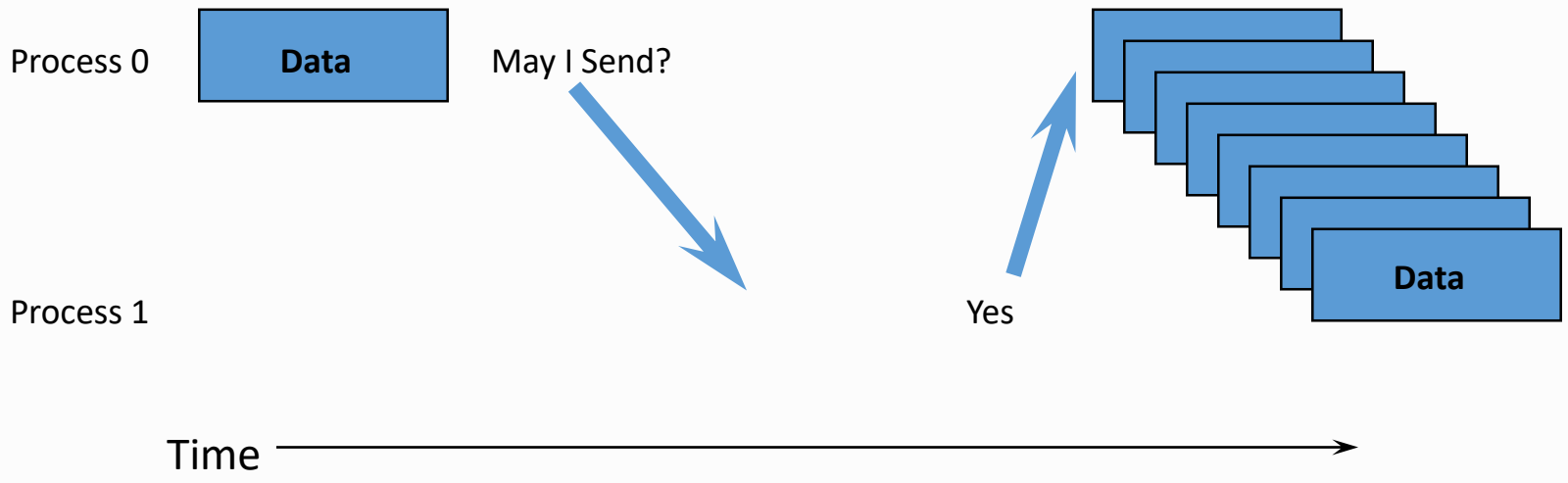
Recall: Message Passing

Involves copying data from user's data space on the source processor to user's data space on the other processor



Implementing MPI

- Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

(Gropp, ~2007)

Implementation and efficiency issues

- It is useful to understand some of these to see the motivation for the variety of send/recv call variants
- Issues:
  - Data copying cost: if you copy into MPI buffers at source and destination
  - Buffer availability and allocation
  - Packetization: who will pay attention to incoming packets? Where do they go?
  - Tag matching
  - Progress engine

Messaging protocols

- Message consists of “envelope” (header) and data
  - Envelope contains tag, communicator, length, source information, plus implementation-specific private data
- MPI implementations often use different protocols for different messages,
  - for a good tradeoff between performance and buffer memory
  - Short
    - Message data (message for short) sent with envelope
  - Eager
    - Message sent assuming destination can store (by allocating memory if needed)
  - Rendezvous
    - Header sent first
    - Message not sent until destination sends an ok-to-send reply

Messaging protocol: programmer's control

- What can you do as a programmer with the knowledge of protocols?
  - Understand why the program performs a certain way
  - Influence it by changing the threshold message size when it switches from one protocol to the other

MPI terminology

- Basic terms
  - Immediate- Operation does not wait for completion (aka nonblocking)
  - Synchronous - Completion of send *requires* initiation (but not completion) of receive
  - Buffered – The MPI implementation can use a buffer the user has already supplied to it for its internal copies
  - Ready - *Correct* send requires a matching receive already be posted
    - I.e. programmer guarantees that such a receive has been posted by the time the send call is made

  - Asynchronous - communication and computation take place simultaneously, ***not*** an MPI concept (implementations *may* use asynchronous methods)

Basic Send/Receive modes

- MPI_Send
  - Sends data. May wait for matching receive. Depends on implementation, message size, and possibly history of computation
- MPI_Recv
  - Receives data
- MPI_Ssend (synchronous send)
  - Waits for matching receive to start
- MPI_Rsend (ready send)
  - Expects matching receive to be pre-posted

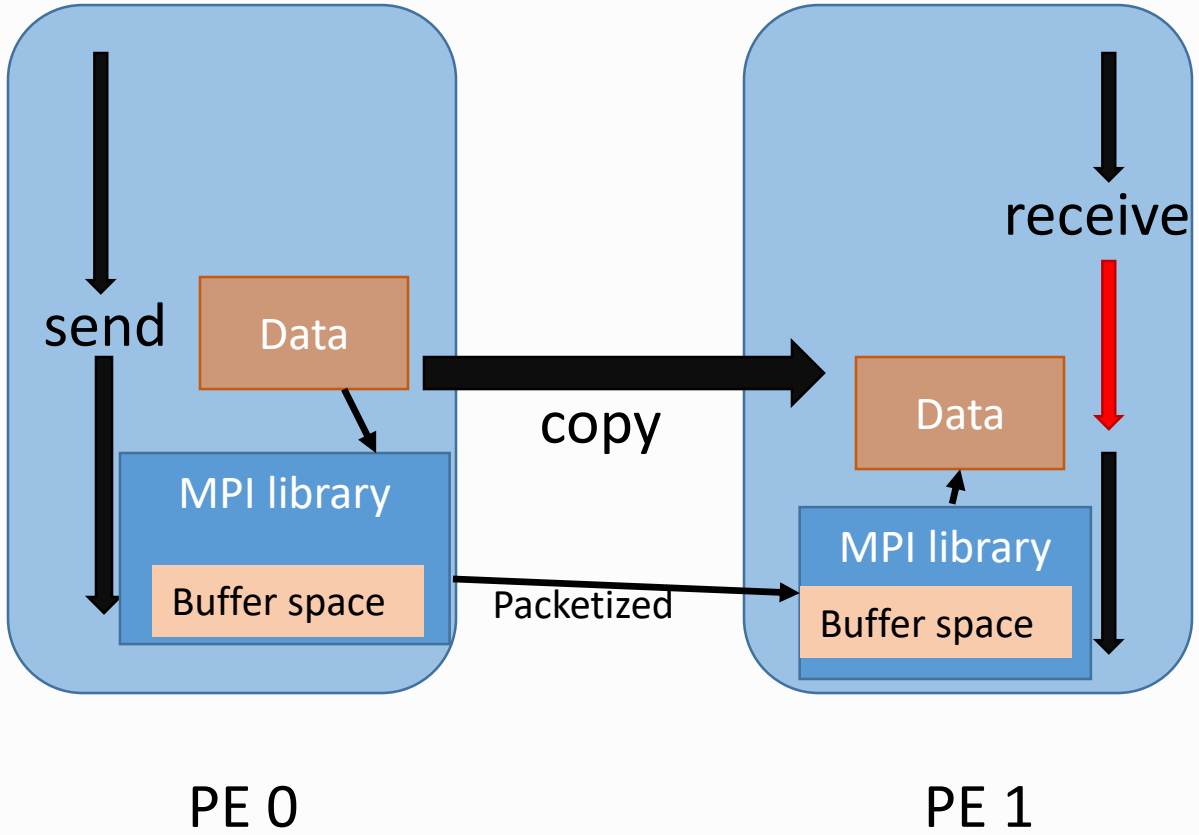
Nonblocking Modes

- MPI_Isend
  - Returns immediately, does not complete until send buffer available for reuse
- MPI_Irsend
  - Expects matching receive to be posted when called
- MPI_Issend
  - Does not complete until buffer available *and* matching receive posted
- MPI_Irecv
  - Does not complete until receive buffer available for use

Recall: Message Passing

There is a problem in both cases:

- Either the sender arrives first, or
- Receiver arrives first



Completion

- MPI_Test
  - Nonblocking test for the completion of a nonblocking operation
- MPI_Wait
  - Blocking test
- MPI_Testall, MPI_Waitall
  - For all in a collection of requests
- MPI_Testany, MPI_Waitany
- MPI_Testsome, MPI_Waitsome
- MPI_Cancel (MPI_Test_cancelled)

Persistent Communications

- MPI_Send_init
  - Creates a request (like an MPI_Isend) but doesn't start it
- MPI_Start
  - Actually begin an operation
- MPI_Startall
  - Start all in a collection
- Also MPI_Recv_init, MPI_Rsend_init, MPI_Ssend_init, MPI_Bsend_init

Testing for Messages

- MPI_Probe
  - Blocking test for a message in a specific communicator
- MPI_Iprobe
  - Nonblocking test
- Can use wildcard tag and source for dynamic communication patterns
- No way to test in all/any communicator

Buffered Communications

- MPI_Bsend
  - May use user-defined buffer
- MPI_Buffer_attach
  - Defines buffer for all buffered sends
- MPI_Buffer_detach
  - Completes all pending buffered sends and releases buffer
- MPI_Ibsend
  - Nonblocking version of MPI_Bsend

Why so Many Forms

- Each represents a different tradeoff in ease of use, efficiency, or correctness
- Smaller sets can provide full functionality
- Need all to tune with
- What about asynchrony?
  - Implementation may be asynchronous or not
  - User may insert MPI_Test or MPI_Iprobe calls to ensure progress

References

- Gropp, W. (~2007).