PARALLEL PROGRAMMING WITH MPI

Part 2

University of Virginia | Research Computing

GLOBAL COMMUNICATIONS REFRESHER

BROADCAST

- Write a program that generates an array of values from 1 to 10 only on the root process
- Broadcast the array to each process.
- Have each process print the array.
- Test it with four processes on the frontend

REDUCTION

- Write a program that generates an array of mpi_rank to mpi_rank +10 on each process.
- Have each process sum its array (Fortran and Python programmers may use the sum intrinsic).
- Perform a reduction to get the overall sum.
- Print the grand sum on all processes. What do you see?
- Have only the master print the grand sum.
- Try replacing Reduce with Allreduce and have all the processes print the grand sum.

GATHER

- Modify your program that creates the arrays mpi_rank to mpi_rank+10 so that they are gathered into the root process.
- Convert gather to allgather.

PERFORMANCE ANALYSIS

SPEEDUP FORMULA

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

EXECUTION TIME COMPONENTS

- For a problem of size *n* on *p* processors

- Inherently sequential computations: $\sigma(n)$
- Potentially parallel computations: $\varphi(n)$
- Communication operations: $\kappa(n,p)$
    - The letters are, respectively, sigma, phi, and kappa

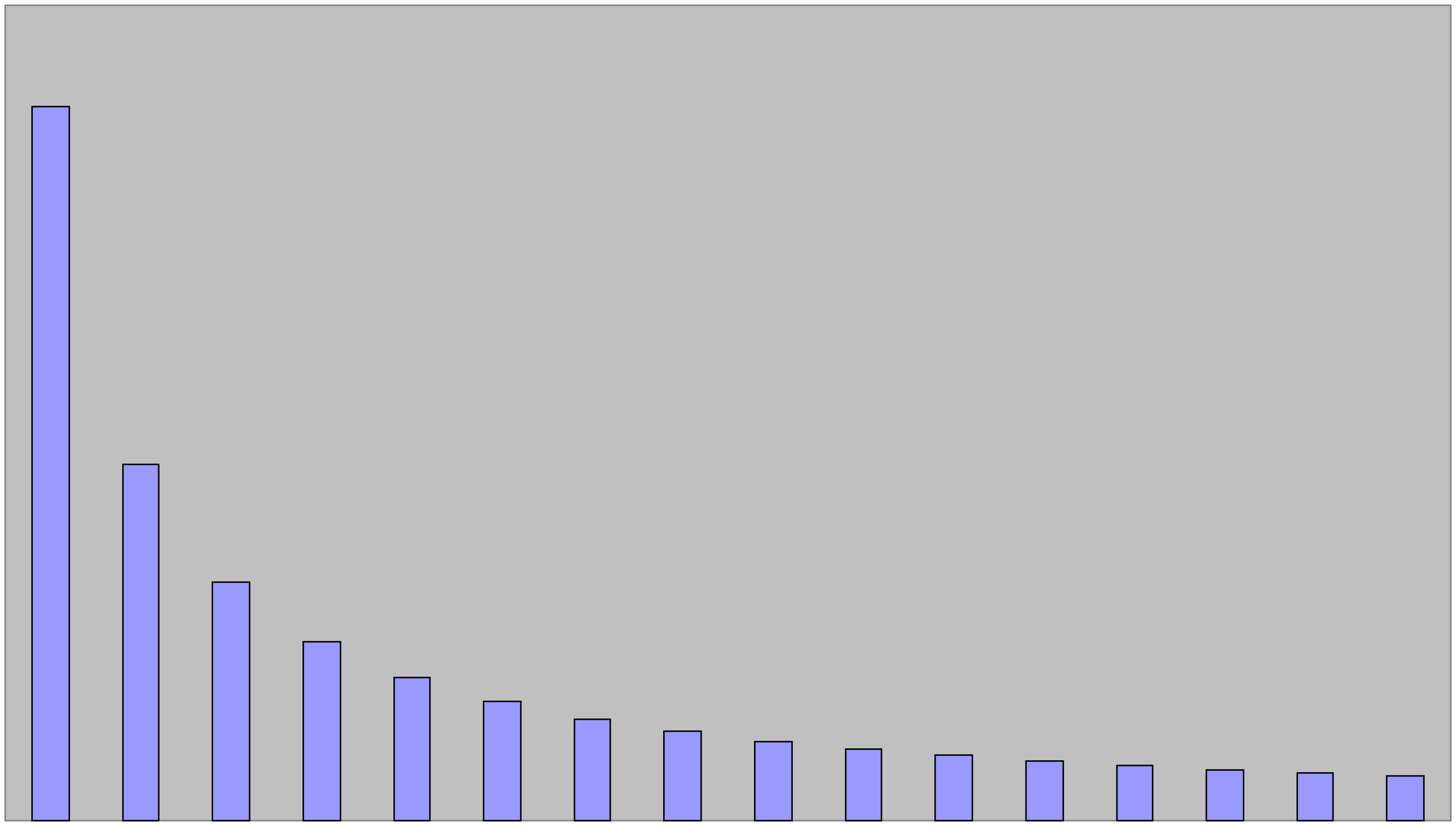
- Single processor is $\sigma(n) + \varphi(n)$
    - No $\kappa(n,p)$
- *p* processors is $\sigma(n) + \varphi(n)/p + \kappa(n,p)$
    - This is the ideal

SPEEDUP EXPRESSION

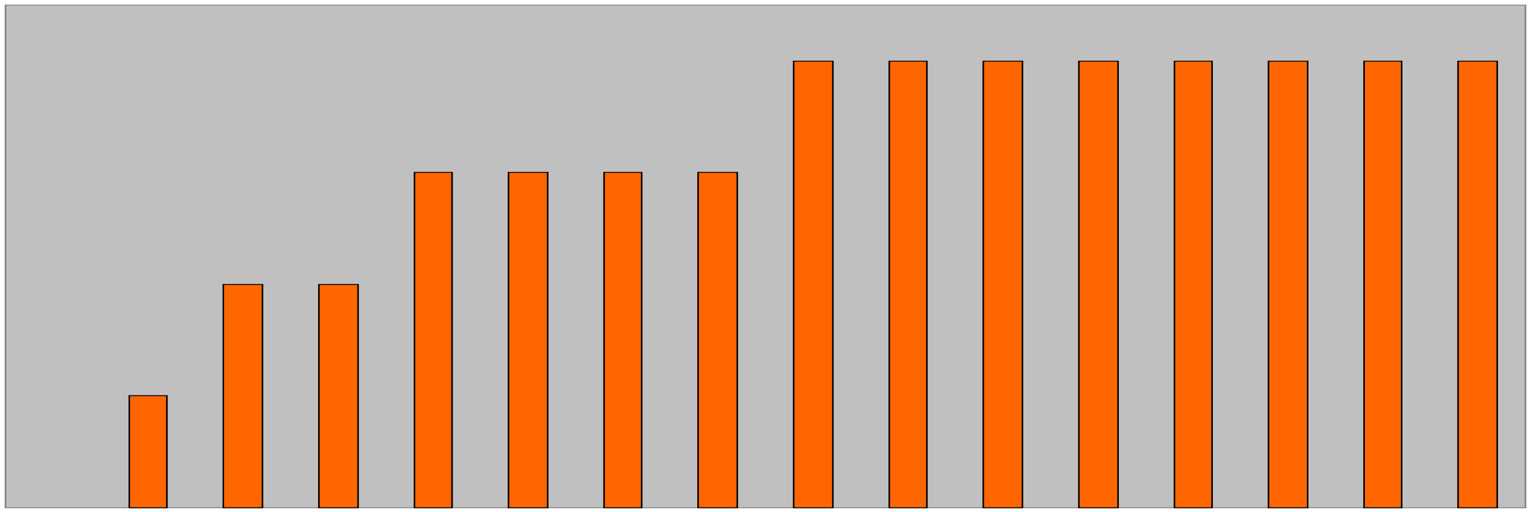
$$y(n,p) = \frac{s(n) + f(n)}{s(n) + f(n)/p + k(n,p)}$$

- Speedups may be less than ideal due to operating system jitter, network noise, etc.
- Occasionally speedup may be *better* than expected due to cache effects (smaller arrays are more cache efficient)

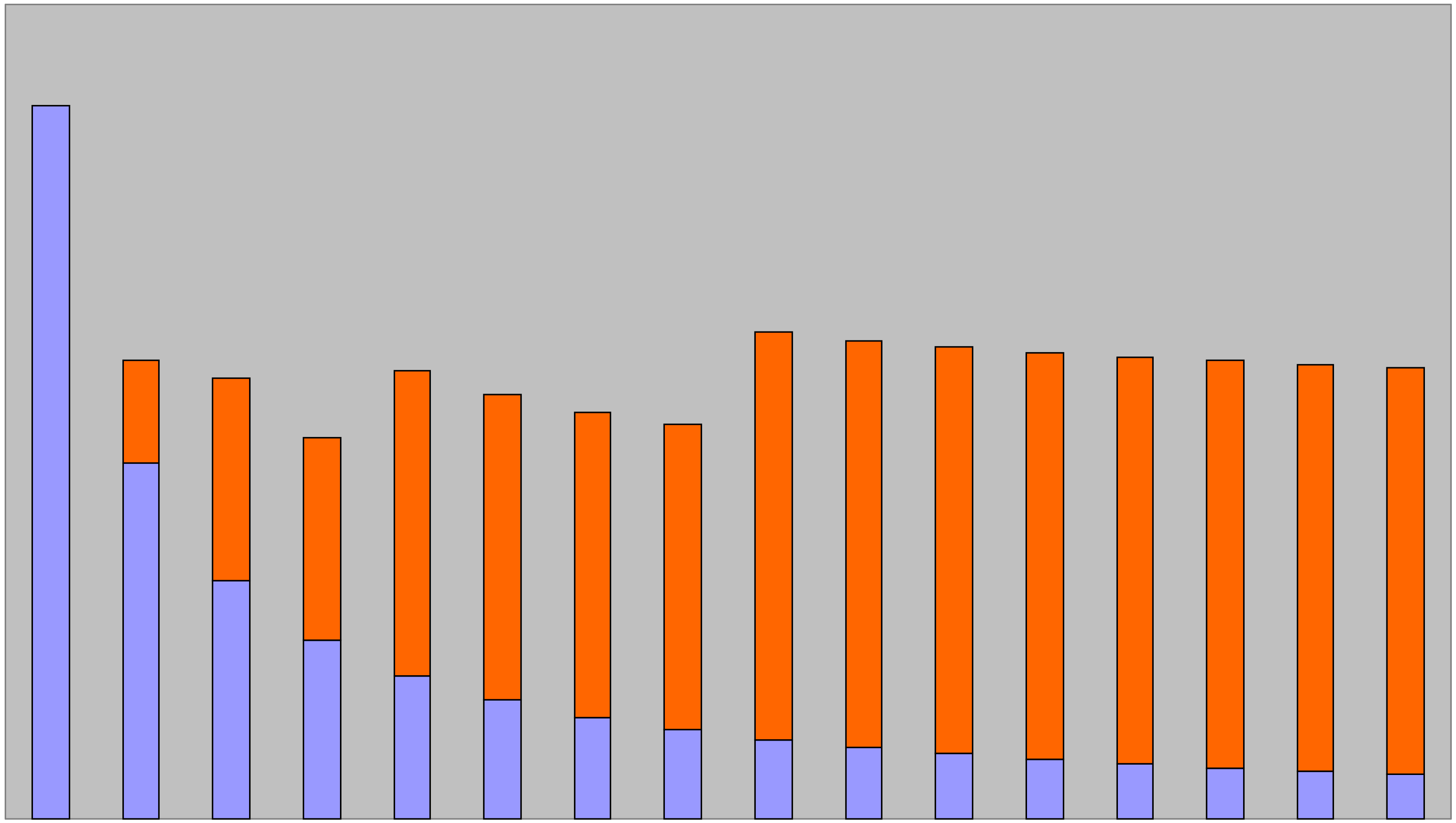
$\varphi(N)/P$: COMPUTATION TIME



$\kappa(N,P)$: COMMUNICATION TIME



$\varphi(N)/P + \kappa(N,P)$: TOTAL TIME TAKEN



EFFICIENCY

$$\text{Efficiency} = \frac{\text{Sequential execution time}}{\text{Processors used} \times \text{Parallel execution time}}$$

$$e(n,p) = \frac{s(n)+f(n)}{p\left(s(n)+f(n)/p+k(n,p)\right)}$$

(epsilon)

Or equivalently

$$e(n,p) = \frac{s(n)+f(n)}{ps(n)+f(n)+pk(n,p)}$$

$0 \le \varepsilon(N,P) \le 1$

$$\varepsilon(n,p) \le \frac{\sigma(n)+\varphi(n)}{p\sigma(n)+\varphi(n)+p\kappa(n,p)}$$

All terms > 0 $\Rightarrow \varepsilon(n,p) > 0$

Denominator > numerator $\Rightarrow \varepsilon(n,p) < 1$

AMDAHL'S LAW

$$y(n,p) = \frac{s(n)+f(n)}{s(n)+f(n)/p+k(n,p)}$$

$$< \frac{s(n)+f(n)}{s(n)+f(n)/p}$$

Let $f = \sigma(n)/(\sigma(n) + \varphi(n))$

f is the fraction of the part that must be done sequentially.
$0 \le f \le 1$

Thus

$$\psi \le \frac{1}{f+(1-f)/p}$$

EXAMPLE 1

- 95% of a program’s execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs? What is the efficiency?
  - *f* = 0.05

$$\psi = \frac{1}{0.05 + (1 - 0.05)/8} \cong 5.9$$

$$\varepsilon = \frac{seq\_time}{p * par\_time} = \frac{t}{8 * t / 5.9} \cong 0.74$$

POP QUIZ

- An oceanographer gives you a serial program and asks you how much faster it might run on 8 processors.
- You can only find one function amenable to a parallel solution.
- Benchmarking and profiling on a single processor reveals 80% of the execution time is spent inside this function.
- What is the best speedup a parallel version is likely to achieve on 8 processors?

POP QUIZ ANSWER

- 20% of a program's execution time is spent within inherently sequential code. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs? What is the efficiency?
  - $f = 0.2$

$$\psi = \frac{1}{0.2 + (1 - 0.2)/8} \cong 3.33$$

$$\varepsilon = \frac{seq\_time}{p * par\_time} = \frac{t}{8 * t / 3.33} \cong 0.42$$

POP QUIZ EXTENDED ANSWER

- 20% of a program’s execution time is spent within inherently sequential code. What is the **limit to the speedup** achievable by a parallel version of the program?
  - *f* = 0.2
  - p->∞

$$\psi \le \lim_{p\to\infty} \frac{1}{0.2+(1-0.2)/p} = \frac{1}{0.2} = 5$$

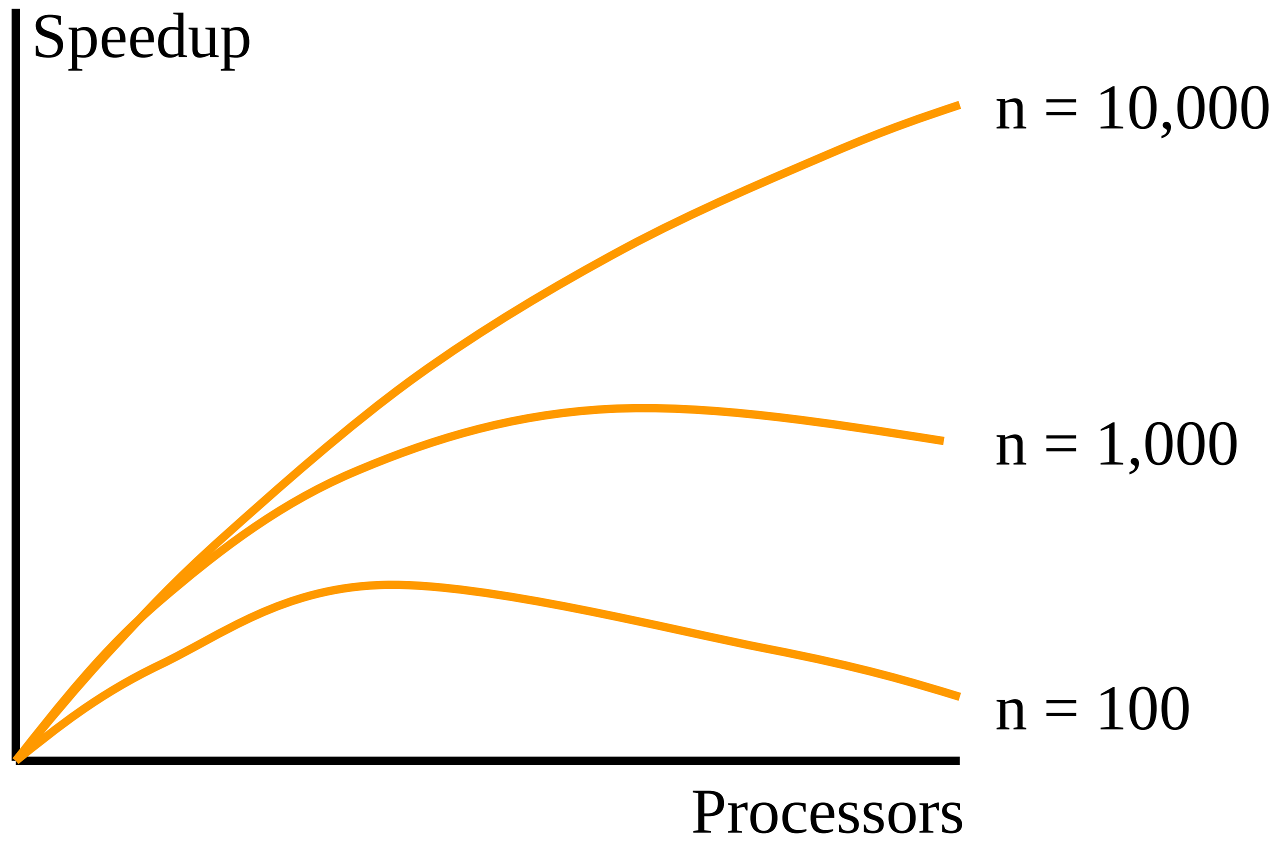
POP QUIZ

- A computer animation program generates a feature movie frame-by-frame.
- Each frame can be generated independently and is output to its own file.
- If it takes 99 seconds to render a frame and 1 second to output it, how much speedup can be achieved by rendering the movie on 100 processors?

AMDAHL EFFECT

- As n (problem size) increases, $\varphi(n)/p$ typically dominates $\kappa(n,p)$ (communication time)
  - In other words, as the problem size increases, the communication drops down as a percentage of time taken
- Thus as n increases, speedup increases

ILLUSTRATION OF AMDAHL EFFECT



SCALABILITY

STRONG SCALING VS. WEAK SCALING

- The Amdahl Effect suggests that more work per process is better.
- Strong scaling: same quantity of work divided among an increasing number of processes.
- Weak scaling: amount of work per process fixed, number of processes increased.

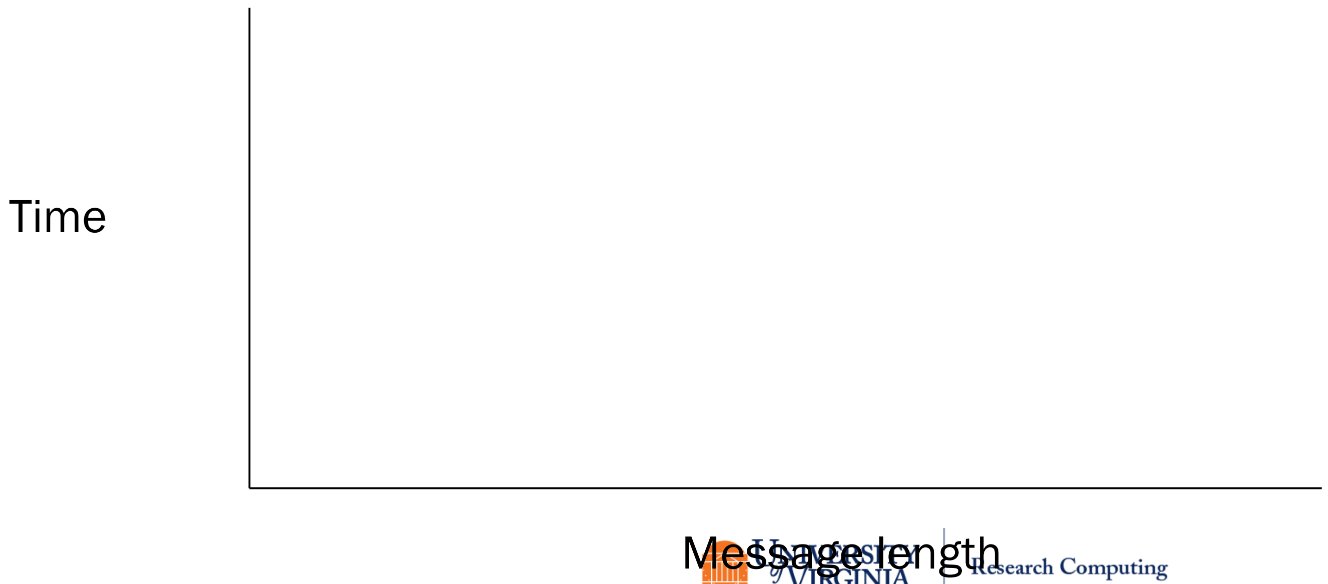
POINT-TO-POINT COMMUNICATIONS

- Individual processes send and receive messages from other processes.
- Send can be
  - Synchronous or asynchronous
  - Buffered or unbuffered
  - Sent to a particular destination
- Receive can be
  - Blocking or non-blocking
  - Buffered or non-buffered
  - Received from a particular source

MESSAGE COST

- One of the most important things to keep in mind

$T_{msg} = \alpha + \beta \text{Bytes}$ (an approximation)



A FEW WARNINGS

- The message cost is rarely linear; there are usually “jaggies” and other discontinuities
- The equation usually only holds on an idle network
- The message cost equation is really a function of the application topology, the network topology, and then number of processors!

NETWORK TYPES

- High latency, low- to moderate-bandwidth:
  - Ethernet
    - 1GE 1 gigabit per second, may be a hub topology
    - 10GE 10 gigabits per second, always switched
    - Typical switch latency for 10GE is approximately 230 ns
- Low-latency, high bandwidth:
  - Most popular (and surviving) is InfiniBand
    - Different ratings are QDR (quad data rate) and FDR (fourteen data rate)
    - Typical switch latency 100 ns for FDR
    - Bandwidth approximately 56 Gb/sec for FDR
  - Intel OmniPath is comparable

POINT-TO-POINT MESSAGES

University of Virginia | Research Computing

FUNCTION MPI_SEND (C)

```
int MPI_Send (
      void             *message,
      int               count,
      MPI_Datatype      datatype,
      int               dest,
      int               tag,
      MPI_Comm          comm
)
```

FUNCTION MPI_SEND (FORTRAN)

- **`MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)`**
  - **`integer count, datatype, dest, tag, comm, ierr`**
  - **`<type> buf(<length>)`**

- Example:
  - `call MPI_SEND(myval,1,MPI_INTEGER,my_rank+1,0, MPI_COMM_WORLD,ierr)`

FUNCTION MPI_SEND (PYTHON, MPI4PY)

- `send (sendobjc, destination, tag)`
  - Note that the lower-case ‘send’ handles pickled objects; use the title-case ‘`Send`’ for NumPy arrays as in the example below. Default for both "destination" and "source" is 0 (root) (thus don't match if only defaults used).
- Example:
  - `MPI.COMM_WORLD.Send ([data,MPI.DOUBLE], rank+1, 0)`
  - `data` is an initialized numpy array
    - When creating a numpy array, by default it creates it as a double. It is advisable to provide an explicit dtype to be sure your types match.
    - To send a scalar create a one-element NumPy array.
  - The number of elements sent is based on the size of the ‘data’ array
  - The error status is returned by the subroutine

FUNCTION MPI_RECV (C)

```
int MPI_Recv (
      void           *message,
      int             count,
      MPI_Datatype    datatype,
      int             source,
      int             tag,
      MPI_Comm        comm,
      MPI_Status     *status
)
```

FUNCTION MPI_RECV (FORTRAN)

- **`MPI_RECV(buf, count, datatype, source, tag, comm, status, ierr)`**
  - **`integer count, datatype, source, tag, comm`**
  - **`integer status(MPI_STATUS_SIZE)`**
  - **`<type> buf(<length>)`**

- Example
  - `call MPI_RECV(myval, 1, MPI_INTEGER, my_rank-1, 0, MPI_COMM_WORLD, status, ierr)`

FUNCTION MPI_RECV (PYTHON)

- **`Recv (recvobjc, destination, tag)`**
  - Note that the lower-case 'recv' handles pickled objects; use the title-case 'Recv' for NumPy arrays

- Example:
  - `MPI.COMM_WORLD.Recv ([data,MPI.DOUBLE], rank-1, 0)`
  - `data` is an initialized numpy array
    - When creating a numpy array, by default it creates it as a double
  - The error status is returned by the subroutine

CODING SEND/RECEIVE

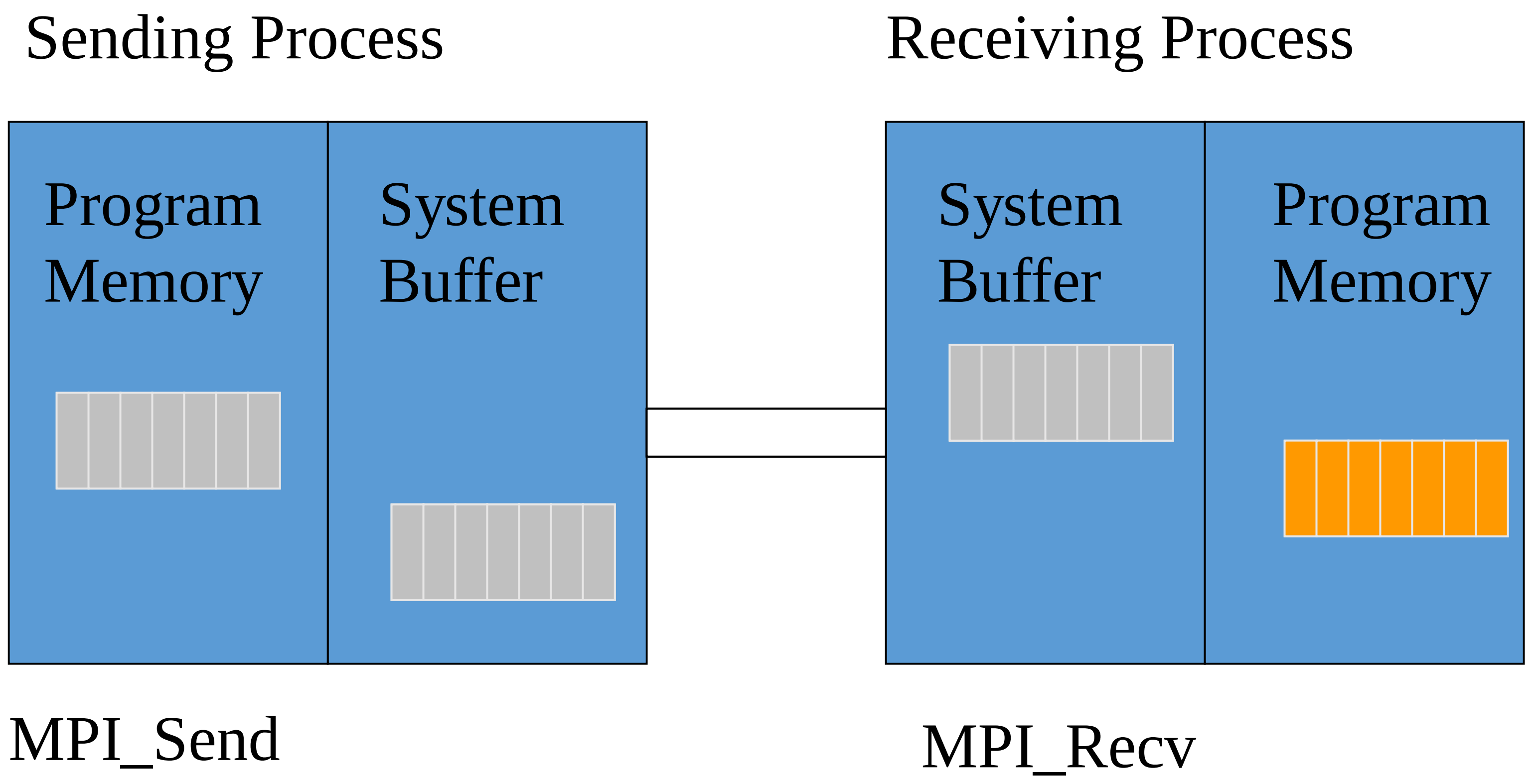
```
…
if (ID == j) {
    …
    Receive from i
    …
}
…
if (ID == i) {
    …
    Send to j
    …
}
…
```

Receive is before Send.
Why does this work?

EXAMPLE (C)

```
#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char** argv) {

// Initialize the MPI environment

MPI_Init(NULL, NULL);

// Find out rank, size

int world_rank;

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

int world_size;

MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// We are assuming at least 2 processes for this task

if (world_size < 2) {
    fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
    MPI_Abort(MPI_COMM_WORLD, 1);

 }

int number;

//Works because 1 will necessarily be in receive state while 0 goes into send state

if (world_rank == 0) {
```

INSIDE MPI_SEND AND MPI_RECV



RETURN FROM MPI_SEND

- Function blocks until message buffer free
- Message buffer is free when
  - Message copied to system buffer, or
  - Message transmitted
- Typical scenario
  - Message copied to system buffer
  - Transmission overlaps computation

RETURN FROM MPI_RECV

- Function blocks until message in buffer
- If message never arrives, function never returns

DEADLOCK

- Deadlock: process waiting for a condition that will never become true
- Easy to write send/receive code that deadlocks
  - Two processes: both receive before send
  - Send tag doesn't match receive tag
  - Process sends message to wrong destination process
  - Both send large messages to each other first, then receive. Too big for buffers.

MPI4PY

```
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank =comm.Get_rank()
#Note alternation in Send/Recv order
if comm.size==2:
   if comm.rank == 0:
      sendmsg = np.array([777],dtype=float)
      comm.Send([sendmsg,MPI.DOUBLE], dest=1, tag=0)
      rec=comm.recv(source=1, tag=1)
   else:
      rec=np.empty(1,dtype=float)
      comm.Recv([rec,MPI.DOUBLE],source=0, tag=0)
      sendmsg = "abc"
      comm.send(sendmsg, dest=0, tag=1)
print rank, rec
```

POINT-TO-POINT EXERCISES

IJOB

- Use ijob for your experiments here
- ijob –A rivanna-training –p standard –c 8

SEND RECEIVE

- Write a program for 2 processes in which rank 0 sends a message to rank 1 and receives a message back from rank 1.
- Modify this code to a "ping pong" where the processes exchange messages some number of times. Try it with 1 first, then try 10 exchanges.

SEND RECEIVE

- Write a program in which each process determines a unique partner to exchange messages. One way to do this is to use

```
if rank < npes//2:
  partner=npes//2 + rank
else
  partner=rank-npes//2
```

- Each tasks sends its rank to its partner. Each task receives the partner's rank.
- Print the message received when done.
- Check that your program works for 1 process.

SENDRECV

- Write a program in which all processes send a message to their left and receive from their right, except for the ends.
  - Make the ends not send any message
  - Make the messages circular, i.e. 0 receives from np-1 and np-1 receives from 0

APPLICATION EXAMPLE: PARTIAL DIFFERENTIAL EQUATIONS

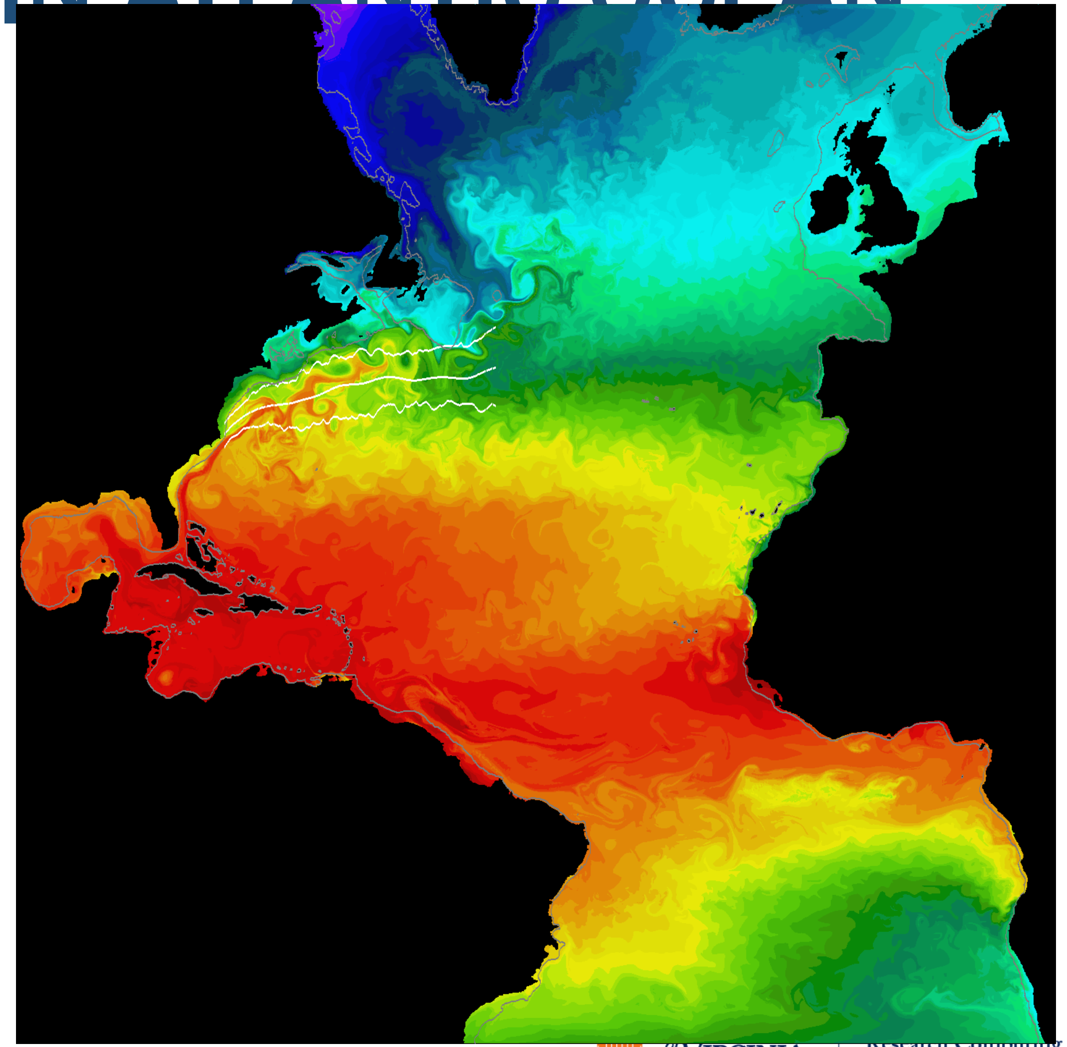
NUMERICAL SOLUTION OF PARTIAL DIFFERENTIAL EQUATIONS

- We will examine point-to-point communications for the example of partial differential equations
- This is a very typical application for P2P messaging
- Definitions:
  - Ordinary differential equation: equation containing derivatives of a function of one variable
  - Partial differential equation: equation containing derivatives of a function of two or more variables

EXAMPLES OF PHENOMENA MODELED BY PDES

- Air flow over an aircraft wing
- Blood circulation in human body
- Water circulation in an ocean
- Bridge deformations as its carries traffic
- Evolution of a thunderstorm
- Oscillations of a skyscraper hit by earthquake
- Strength of a toy

MODEL OF SEA SURFACE TEMPERATURE IN ATLANTIC OCEAN



Courtesy MICOM group at the Rosenstiel School of Marine and Atmospheric Science, University of Miami

SOLVING PDES

- Finite element method
- Finite difference method (our focus)
  - Converts PDE into matrix equation
  - Result is usually a sparse matrix
  - Matrix-based algorithms represent matrices explicitly
  - Matrix-free algorithms represent matrix values implicitly (our focus)

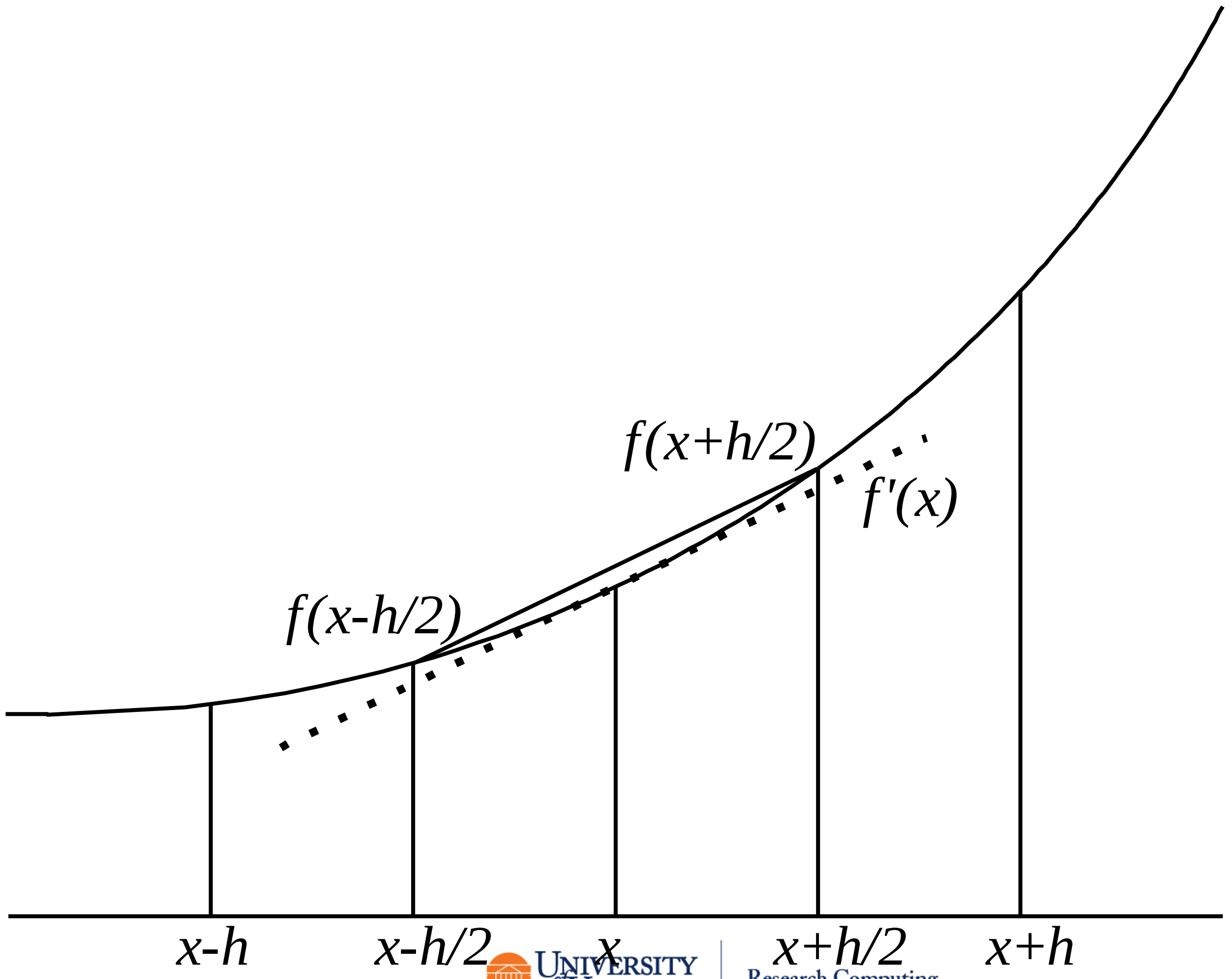
LINEAR SECOND-ORDER PDES

- Linear second-order PDEs are of the form

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H$$

  where A - H are functions of x and y only
- Elliptic PDEs: $B^2 - AC < 0$
- Parabolic PDEs: $B^2 - AC = 0$
- Hyperbolic PDEs: $B^2 - AC > 0$

DIFFERENCE QUOTIENTS



FORWARD-DIFFERENCE FORMULA FOR 1ST DERIVATIVE

$$f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

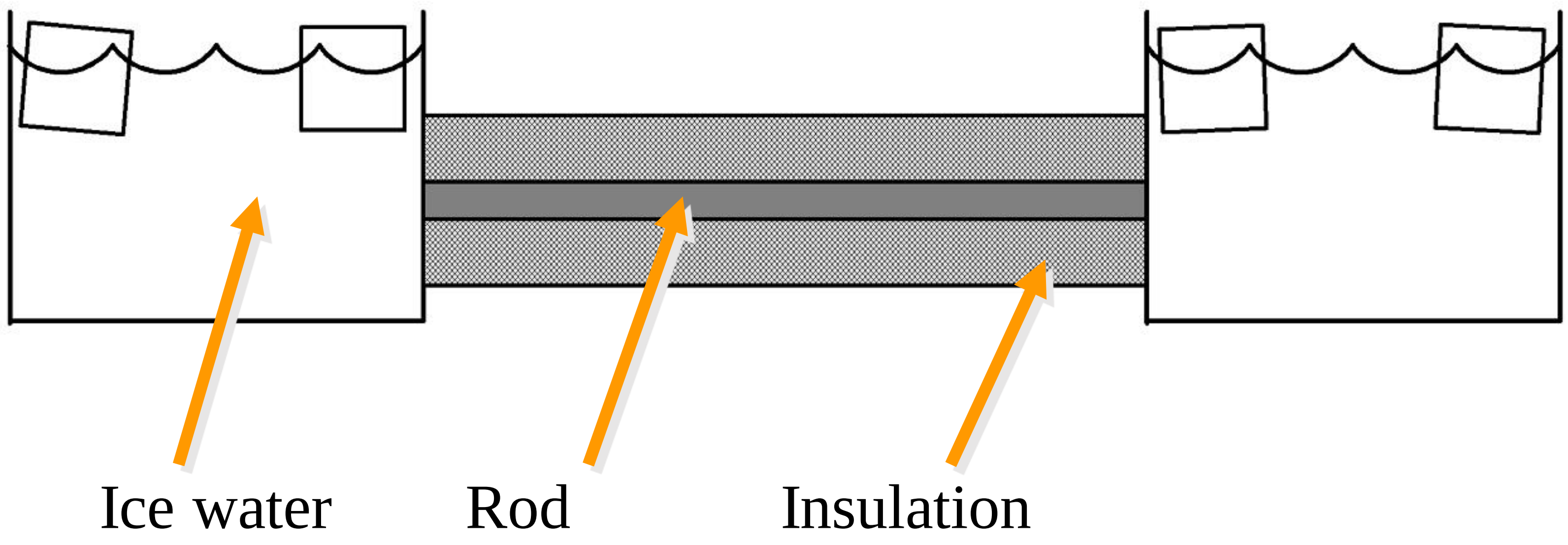
CENTERED-DIFFERENCE FORMULAS FOR 1ST, 2D DERIVATIVES

$$f'(x) \approx \frac{f(x+h/2) - f(x-h/2)}{h}$$

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

HEAT DIFFUSION EQUATIONS

BOUNDARY VALUE PROBLEM



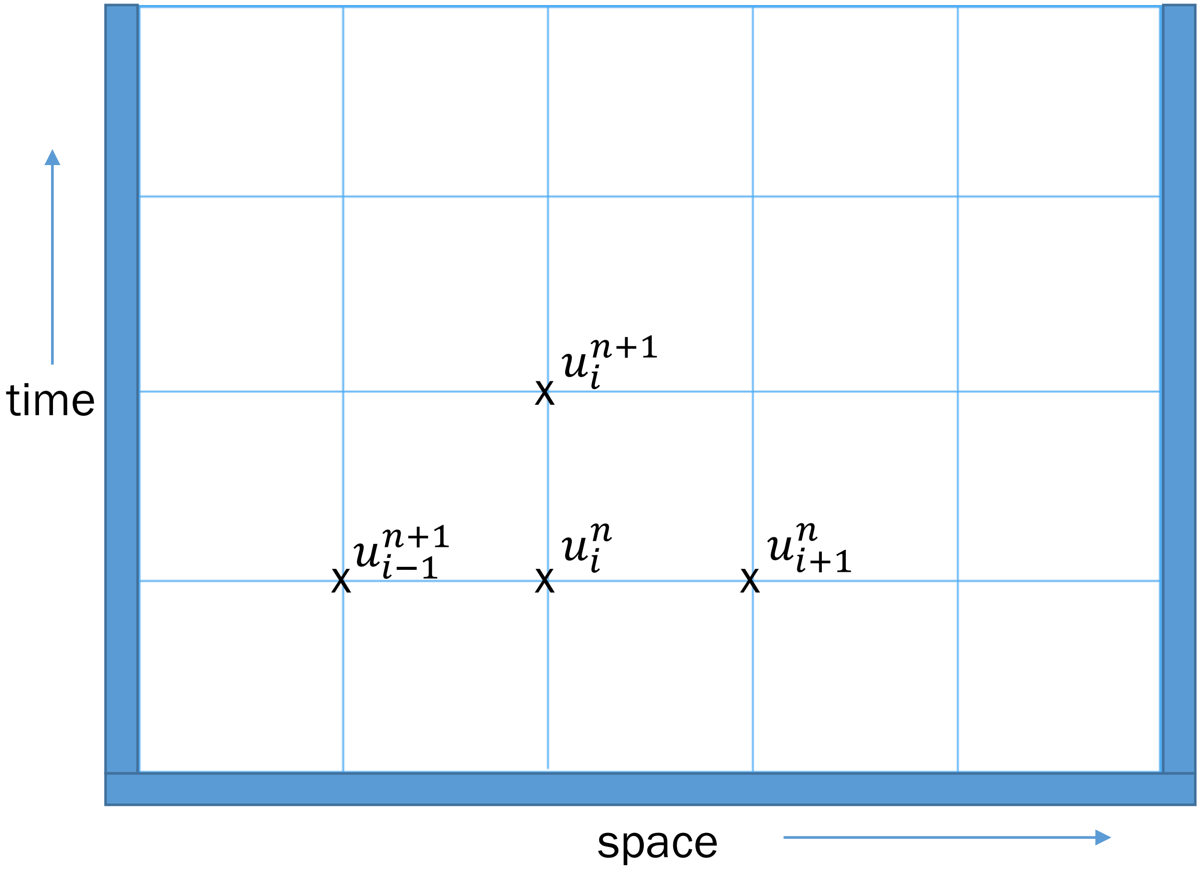
ONE-DIMENSIONAL DIFFUSION EQUATION

- This equation can be represented by the PDE

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial^2 x} + f$$

- In this equation, α is the diffusion coefficient (we will assume it is constant) and f is the forcing function.
- The forcing is often zero.

FINITE-DIFFERENCE APPROXIMATION I=SPACE INDEX, N=TIME INDEX



x_i=iΔx , t_n = nΔt

FINITE DIFFERENCE FORWARD EULER METHOD

- Not the most widely used in practice, but easy to understand.
- We take a forward difference in time and a centered difference in space.

$$u_i^{n+1} = u_i^n + \mathrm{F}(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \Delta t f_i^n$$

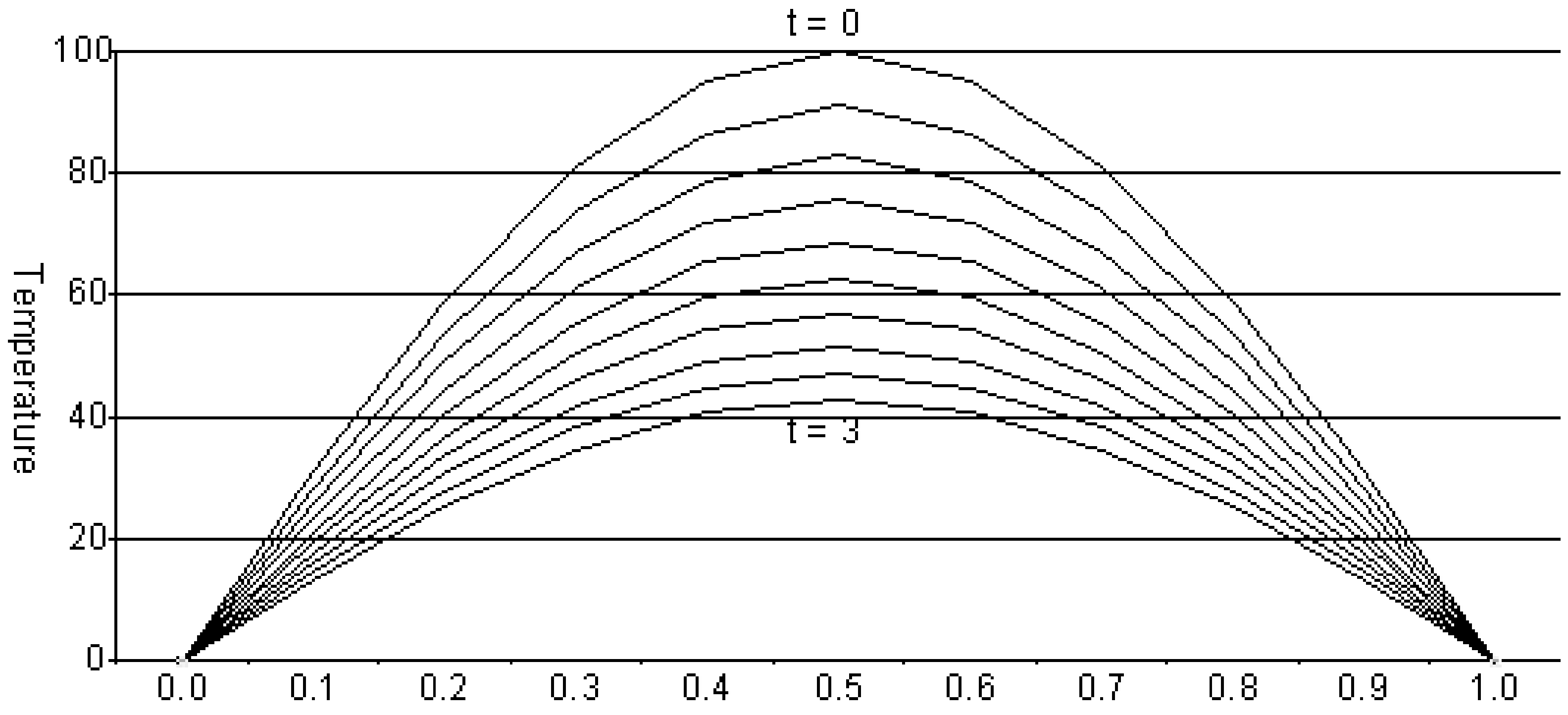
- The constant F is the mesh Fourier number

$$F = \alpha \frac{\Delta t}{\Delta x^2}$$

- For this method to be stable we must choose Δt and Δx such that $F \leq \frac{1}{2}$

ROD COOLS AS TIME PROGRESSES

- Initial temperature in rod: 100 sin (πx)



PARTITIONING

- One data item per grid point in this case (temperature)
- Domain decomposition: divide the grid into subgrids. Assign each to a processor.

COMMUNICATION

- Identify communication pattern between primitive tasks
- Each interior primitive task has three incoming and three outgoing channels

SEQUENTIAL EXECUTION TIME

- t – time to update element
- *n* – number of section
    - Each row has n+1 spots in the matrix
    - But the left and right column are always zero
- *m* – number of iterations
- Sequential execution time: $m\,t(n\text{-}1)$

PARALLEL EXECUTION TIME

- p – number of processors
- λ – message latency

- Parallel execution time $m(t(n-1)/p+2\lambda)$

- But is that faster???

SUMMARY: DESIGN STEPS

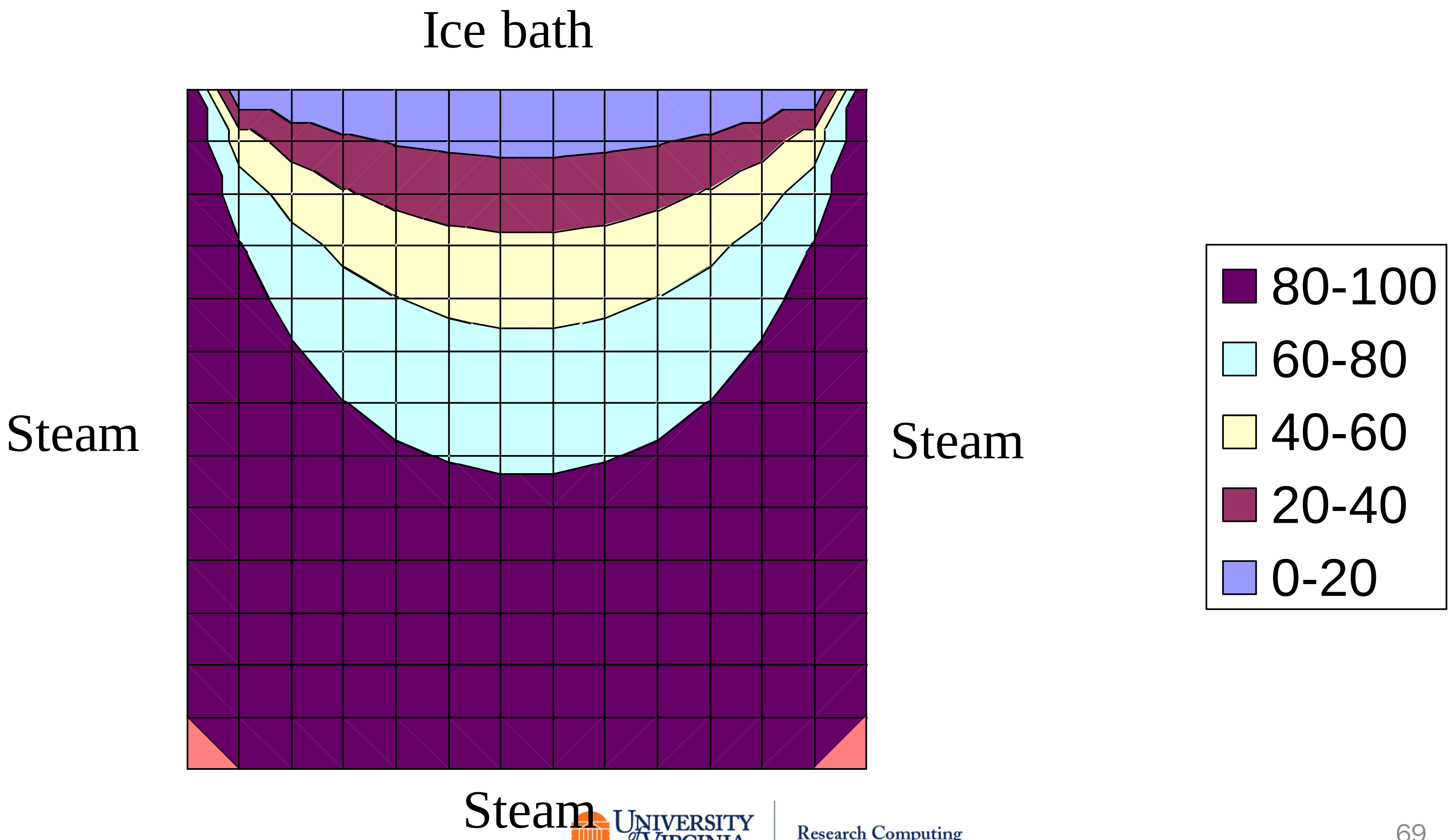
- *Partition* computation
- Analyze *communication*
- *Agglomerate* tasks
- *Map* tasks to processors

- Goals
  - Maximize processor utilization
  - Minimize inter-processor communication

SERIAL PSEUDOCODE (TRANSLATE TO YOUR LANGUAGE)

```
#Determine deltas
dx=float(L)/(Nx+1)
dt=float(T)/(Nt+1)
#Set mesh constant
F = a*dt/dx**2
#Check stability criterion
if ( F>0.5) exit
#Initialize u
u=100.*sin(pi*dx*i) for i in 0/1 to Nx/Nx+1
#Initialize old u
u_old=u

#Start update loop
for t=1,Nt+1 do
  for n=1/2,Nx-1/Nx do
    u[i] = u_n[i] + F*(u_n[i+1] - 2*u_n[i] + u_n[i+1])
  enddo
# Boundary conditions
  u[0/1] = 0; u[Nx/Nx+1] = 0
# Update u_n before next step
  u_n= u
enddo
```

STEADY STATE HEAT DISTRIBUTION PROBLEM



SOLVING THE PROBLEM

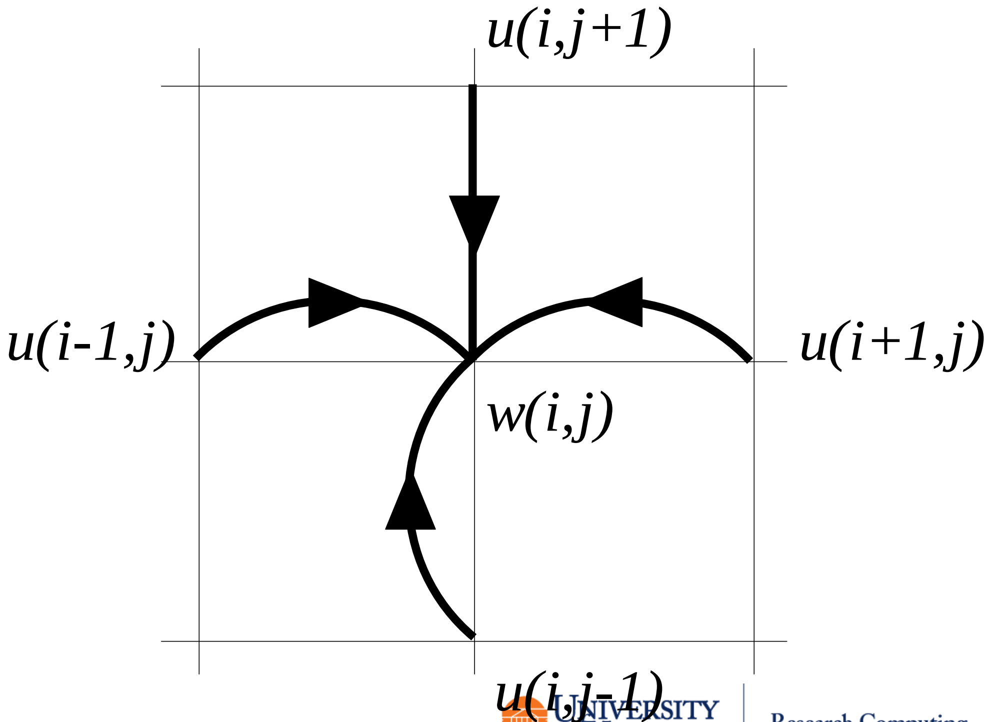
- Underlying PDE is the Poisson equation

$$u_{xx} + u_{yy} = f(x, y)$$

- This is an example of an elliptical PDE
- Will create a 2-D grid
- Each grid point represents value of state state solution at particular (*x*, *y*) location in plate

HEART OF SEQUENTIAL C PROGRAM

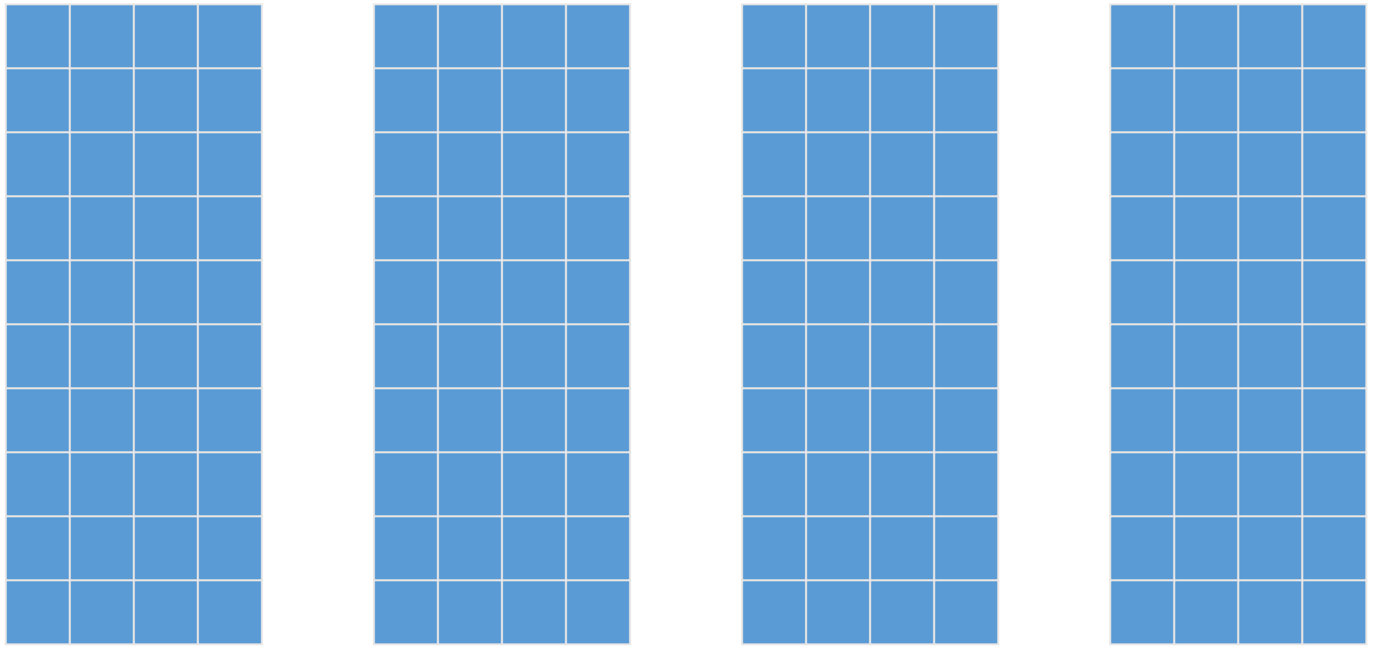
```
w[i][j] = (u[i-1][j] + u[i+1][j] +
                 u[i][j-1] + u[i][j+1]) / 4.0;
```



PARALLEL PROGRAM DESIGN

- Associate primitive task with each element of matrix
- Examine communication pattern
- Agglomerate tasks in same column
- Static number of identical tasks
- Regular communication pattern
- Strategy: agglomerate columns, assign one block of columns to each task

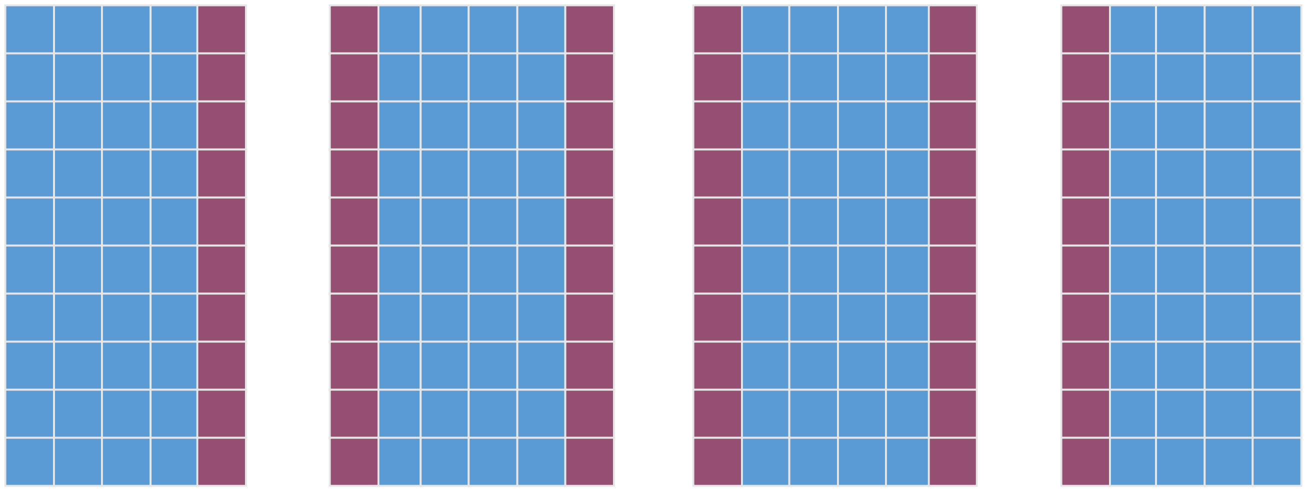
RESULT OF AGGLOMERATION AND MAPPING FORTRAN LAYOUT



GHOST POINTS

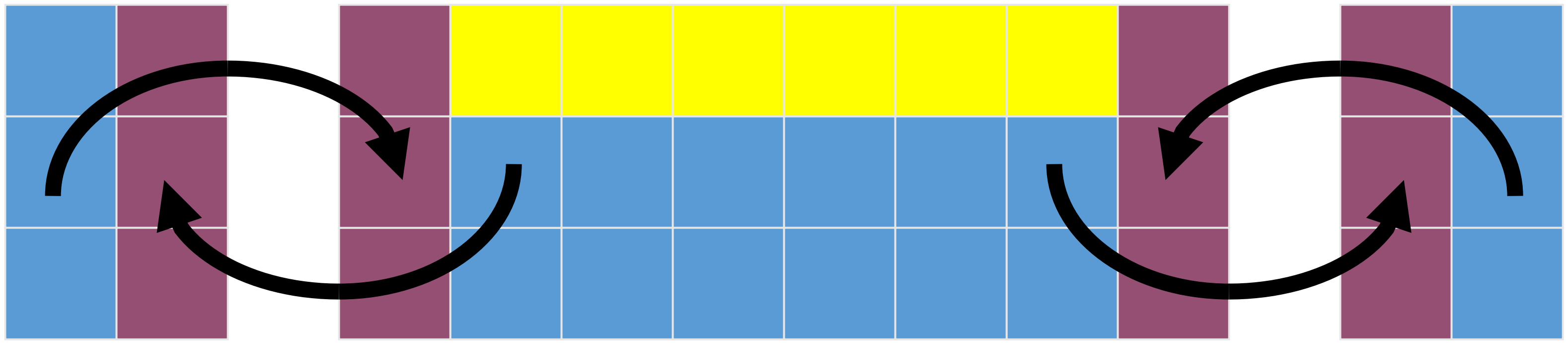
- Ghost points: memory locations used to store redundant copies of data held by neighboring processes
- Allocating ghost points as extra columns simplifies parallel algorithm by allowing same loop to update all cells

MATRICES AUGMENTED WITH GHOST POINTS



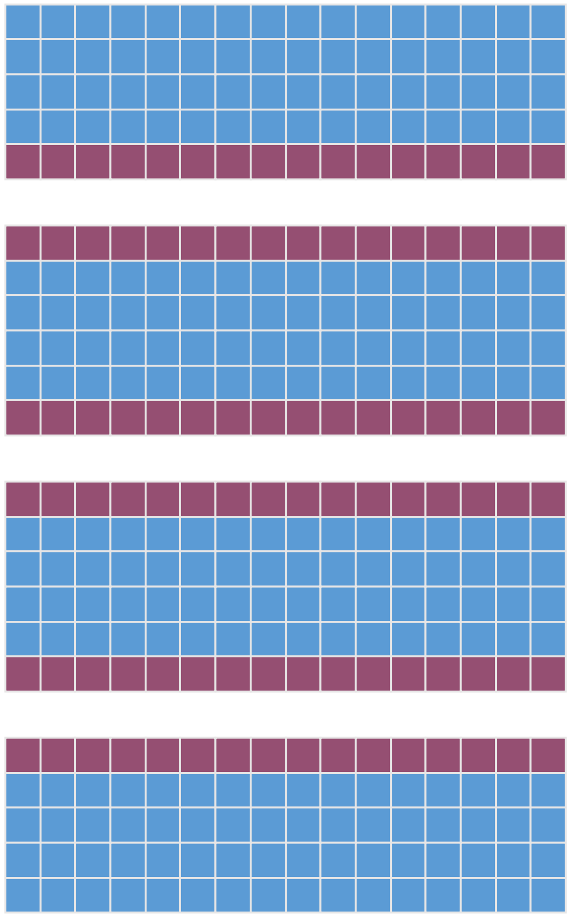
Purple cells are the ghost points.

COMMUNICATION IN AN ITERATION



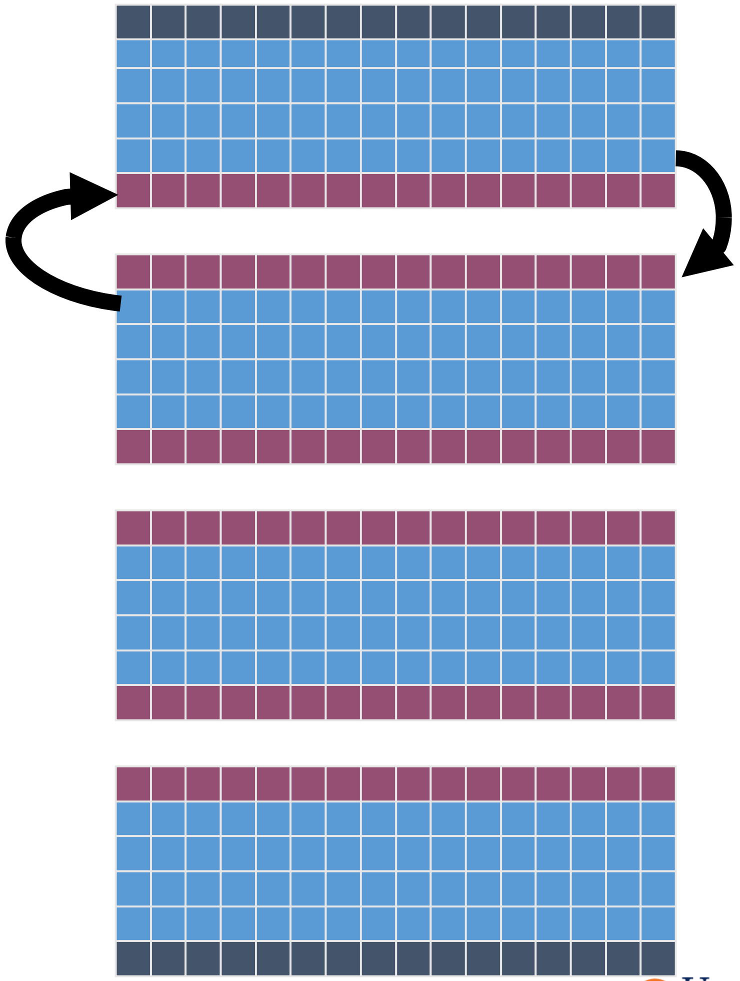
This iteration the process is responsible for computing the values of the yellow cells.

EXAMPLE DECOMPOSITION C/PYTHON LAYOUT



Research Computing

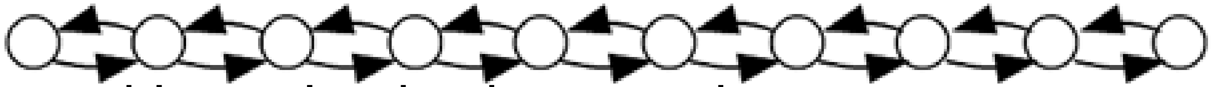
COMMUNICATIONS



Research Computing

HOW TO SYNCHRONIZE THE SENDS AND RECEIVES?

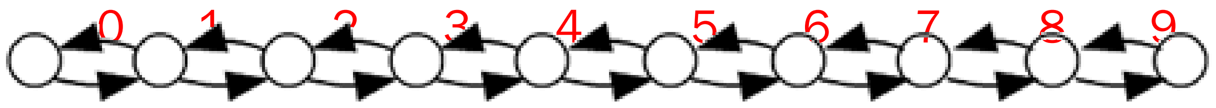
- Imagine we have a number of processes communicating with those next to them



  - The rod in an ice bath example
    - Each task is one spot of the rod in time
  - Or the heated plate example
    - Each task is one ‘column’ of the plate in time
- How would they communicate?

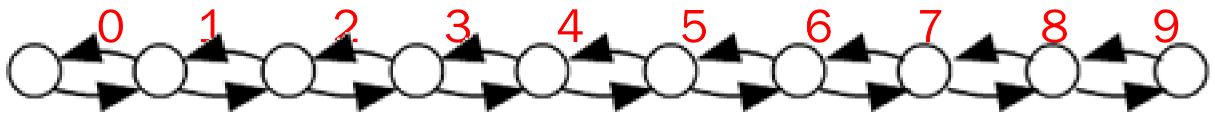
COMMUNICATION SYNCHRONIZATION

- Each task has a rank:



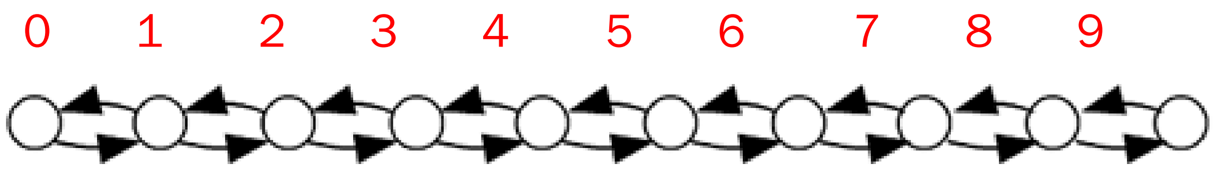
- Note that ranks 0 and 9 only have to do one communication
- All the other ranks have to communicate with the rank one to the left and the rank one to the right

COMMUNICATION STRATEGY 1



- First ‘round’ of the communication: each rank sends to the rank one to the left, and receives from the one to the right
  - On the second ‘round’, it does the reverse
- Will this work? Why or why not?

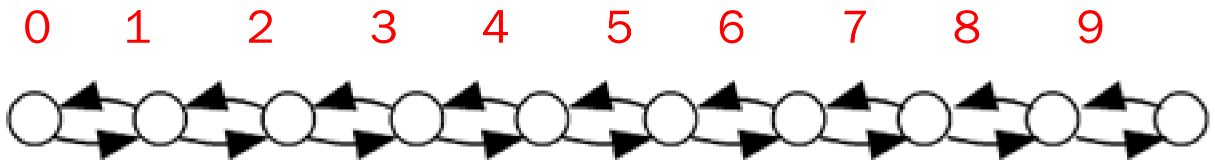
COMMUNICATION STRATEGY 2



- Assume an even number of nodes (can be enforced in the code)
- First 'round' of the communication: each even rank sends to the odd rank one to the left, and each odd rank receives from the one to the right
  - On the second 'round', it does the reverse
  - This needs two more rounds for a total of 4!
- Will this work? Why or why not?

University of Virginia | Research Computing

COMMUNICATION STRATEGY 3



- Assume an even number of nodes
- First 'round' of the communication: each even rank and the odd rank to the left do a MPI_SendRecv()
  - On the second 'round', each odd rank and the even to the right do a MPI_SendRecv()
- Will this work? Why or why not?

SPECIAL CONSIDERATIONS

- Need to handle special cases of ranks 0 and *p*-1
  - Probably easiest to have them send and receive dummy values
  - MPI_PROC_NULL is a predefined "no op"

WHAT WILL THIS LOOK LIKE IN MPI?

- int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

THE EVEN EASIER WAY

```
if rank==0
    up=MPI_PROC_NULL
 or for Python MPI.PROC_NULL
    down=rank+1
else if rank==npes-1
    up=rank-1
    down=MPI_PROC_NULL
else
    up=rank-1
    down=rank+1
end
```

THEN WE CAN USE SENDRECV

- First send up and receive down

```
comm.Sendrecv([w[1,1:nc+1],MPI.DOUBLE],up,tag,[w[nrl+1,1:nc
+1],MPI.DOUBLE],down)
```

- Then send down and receive up

```
comm.Sendrecv([w[nrl,1:nc+1],MPI.DOUBLE],down,tag,[w[0,1:nc+1],
MPI.DOUBLE],up)
```

C uses

(upperlimit=500,ghost_rows=1)

Following is all one line

```
MPI_Sendrecv (u[upperlimit], N, MPI_DOUBLE, rank+1,
0,u[upperlimit+ghost_rows], N, MPI_DOUBLE, rank+1, 0,
MPI_COMM_WORLD, &status);
```

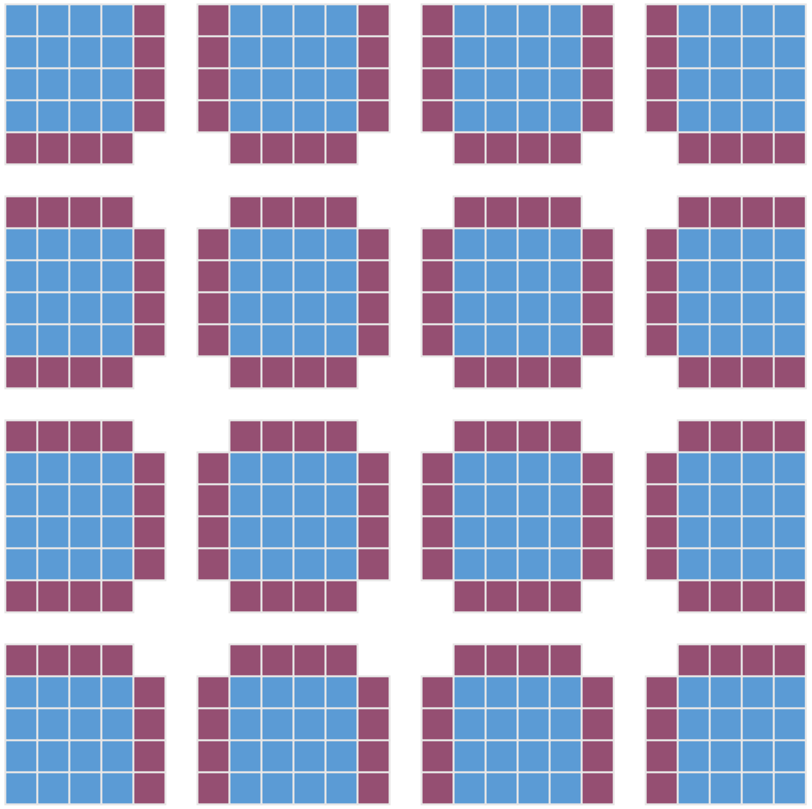
FORTRAN USES

```
call MPI_SENDRECV(w(1:nr,1), nr,MPI_DOUBLE_PRECISION,left,tag, &
       w(1:nr,ncl+1),nr,MPI_DOUBLE_PRECISION,right,tag,         &
                                            MPI_COMM_WORLD,status,ierr)

call MPI_SENDRECV(w(1:nr,ncl),nr,MPI_DOUBLE_PRECISION,right,tag,&
       w(1:nr,0),nr,MPI_DOUBLE_PRECISION,left,tag,                &
                                            MPI_COMM_WORLD,status,ierr)
```

PARALLEL ALGORITHM 2

- Associate primitive task with each matrix element
- Agglomerate tasks into blocks that are as square as possible (checkerboard block decomposition)
- Add rows of ghost points to all four sides of rectangular region controlled by process

EXAMPLE DECOMPOSITION



IMPLEMENTATION DETAILS

- Using ghost points around 2-D blocks requires extra copying steps
- Ghost points for left and right sides are not in contiguous memory locations
- An auxiliary buffer must be used when receiving these ghost point values
- Similarly, buffer must be used when sending column of values to a neighboring process

MPI TYPES

- MPI provides derived datatypes which can simplify the creation of columns (for C) or rows (for Fortran)
- A little beyond our scope but not hard to use.
- Create an MPI_Type_vector
  - C
  - `MPI_Datatype columntype;`
  - `MPI_Type_vector(nrows,1,ncols,MPI_FLOAT,&columntype);`
  - `MPI_Commit(&columntype);`
  - Fortran
  - `integer   :: rowtype`
  - `Call MPI_Type_vector(ncols,1,nrows,MPI_REAL,rowtype,ierr)`
  - `Call MPI_Commit(rowtype,ierr)`

PGAS

PARTITIONED GLOBAL ADDRESS SPACE

- PGAS abstracts the data decomposition problem
- SPMD (Single Program Multiple Data) model
- Arrays are declared as global entities and are automatically decomposed and distributed among processing elements (PEs).
- Local hardware models are utilized to maximize efficiency
- Often implemented through *coarrays*
- Typically uses a message-passing communications layer

PGAS LANGUAGES

- Unified Parallel C (UPC)/Unified Parallel C++ (UPC++)
- Co-Array Fortran
  - Part of the 2008 standard
- Chapel

CO-ARRAY FORTRAN

- To see how this works we'll look at Co-Array Fortran
- Each copy of the program running as a process is called an *image*.
- Each image runs as a normal Fortran program.
- Example declaration:
  - Real, dimension(1000), codimension[*] :: x,y
  - Real, codimension[*] :: z
- Then

  ```
  x(:)=y(:)[q]
  ```

  copies the version of coarray y on image q to coarray x on the executing image (which could be all of them).

COARRAYS

- Coarrays always exist on each image
- Number of images is returned by an intrinsic function `num_images()`
- Intrinsic function `this_image()` returns the image index (counting from 1 as usual for Fortran)
- With no square brackets the array is only that on the image (the local copy)

COARRAY DECLARATIONS

- Coarrays are declared much like any Fortran array and can have rank higher than 1
- The upper bound for the codimension is never specified, so that any number of images can be instantiated
- The total number of subscripts (dimensions) local+codimension is limited to 15
- Example
  - real :: array(10,20)[10,-1:8,0:*]
    - Shape is 10,20. If we set up 128 images the lower cobounds are 1,-1,0 and the upper cobounds are 10,8,1

MORE ABOUT COARRAYS

- Coarrays may be allocatable
- Coarrays may contain derived types
- Coarrays may not be pointers (either Fortran style or c_ptr style)
- Codimension bounds are column-oriented as for regular bounds
- Must be allocated over all images (no support for subsetting processes yet)
- Only a single image can be addressed at a time (as of Fortran 2008)

BARRIERS

- The only barrier implemented now is SYNC
- `sync_all`
- `sync_images(integer, integer array, or *)`
- `sync_memory`

EXAMPLE

- From gfortran wiki

```
! Created by Tobias Burnus 2010.
program Hello_World
implicit none
 integer :: i
! Local variable
character(len=20) :: name[*] ! scalar coarray
 ! Note: "name" is the local variable while "name[<index>]“
 ! accesses the variable on a remote image

 ! Interact with the user on Image 1
if (this_image() == 1) then
write(*,'(a)',advance='no') 'Enter your name: '
read(*,'(a)') name
! Distribute inormation to other images
do i = 2, num_images() name[i] = name
end do

end if
sync all ! Barrier to make sure the data has arrived
! I/O from all nodes
write(*,'(3a,i0)') 'Hello ',trim(name),' from image ', this_image()
 end program Hello_world
```

- Scales poorly due to serialization of broadcast
- 2015 standard introduces many more collective communications

CRITICAL SECTIONS

- Like OpenMP a critical section can be defined

```
critical
  Code executed on one image at a time
end critical
```

UPC/C++

```
#include <upc.h>
printf("Thread %d of %d: hello UPC world\n",
MYTHREAD, THREADS);
```

- Looks more like OpenMP than CAF does.
- UPC++ looks more like CAF

```
shared_var<int> s; //shared  ints  in  UPC
shared_array<int> sa(8); //shared  int sa[8]
```

RESOURCES

- http://www.opencoarrays.org/
- https://crd.lbl.gov/departments/computer-science/CLaSS/research/DEGAS/degas-software-releases/