Data Partitioning

Isoefficiency Metric

- Parallel system: parallel program executing on a parallel computer
- Scalability of a parallel system: measure of its ability to increase performance as number of processors increases
- A scalable system maintains efficiency as processors are added
- Isoefficiency: way to measure scalability

Isoefficiency Metric

- Establish a relationship between the amount of work, W, to be accomplished and the number of processors, p, such that E remains constant as p increases

$$T_p = (W + T_O(W,p)/p, \quad W = T_1$$
$$S = W/T_p = Wp/(W + T_O)$$
$$E = S/p = W/(W + T_O(W,p)$$

- E will remain constant if $T_O(W,p)/W$ is constant

$$W = K(N)T_O(W,p)$$ K = isoefficiency function

Isoefficiency Relation Usage

- Used to determine the range of processors for which a given level of efficiency can be maintained
- The way to maintain a given efficiency is to increase the problem size when the number of processors increase.
- The maximum problem size we can solve is limited by the amount of memory available
- The memory size is a constant multiple of the number of processors for most parallel systems

Scalability Function

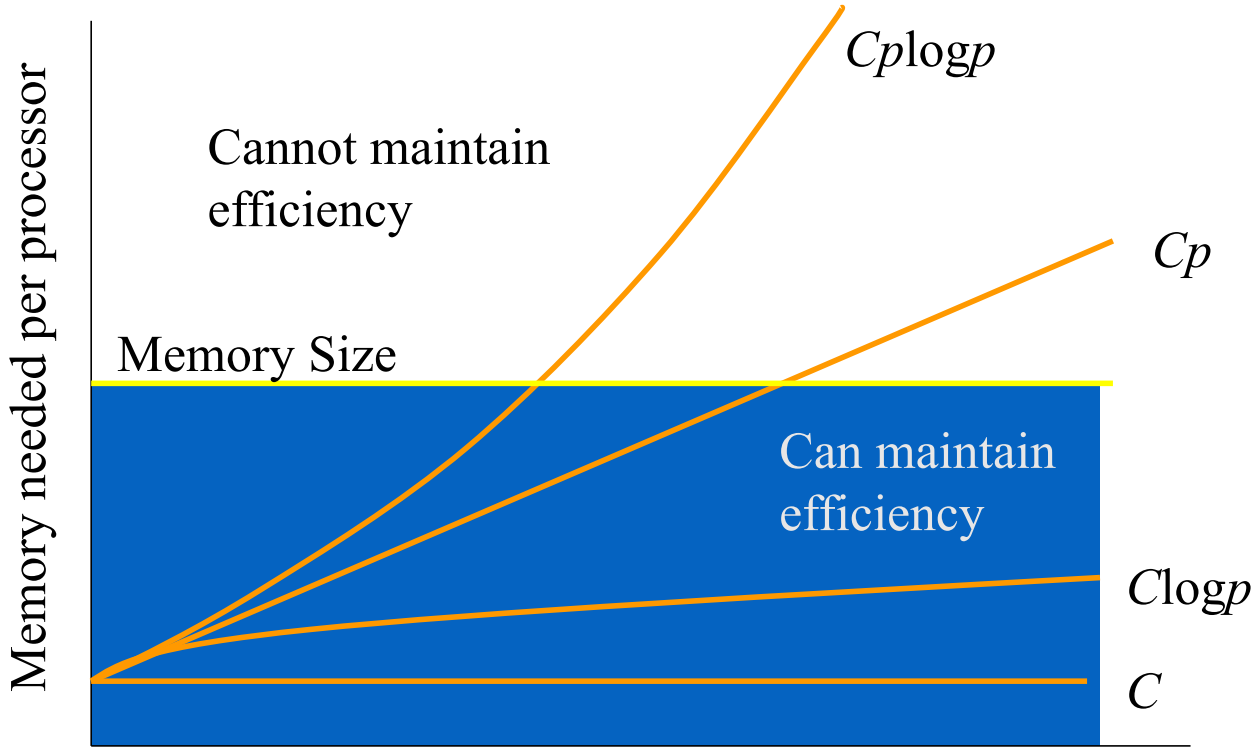
- Isoefficiency relation: $W = K(N)T_O(W, p)$

- Suppose isoefficiency relation is $N = f(p)$
- Let *M(N)* denote memory required for problem of size *N*
- *M(f(p))/p* shows how memory usage **per processor** must increase to maintain same efficiency
- We call *M(f(p))/p* the scalability function

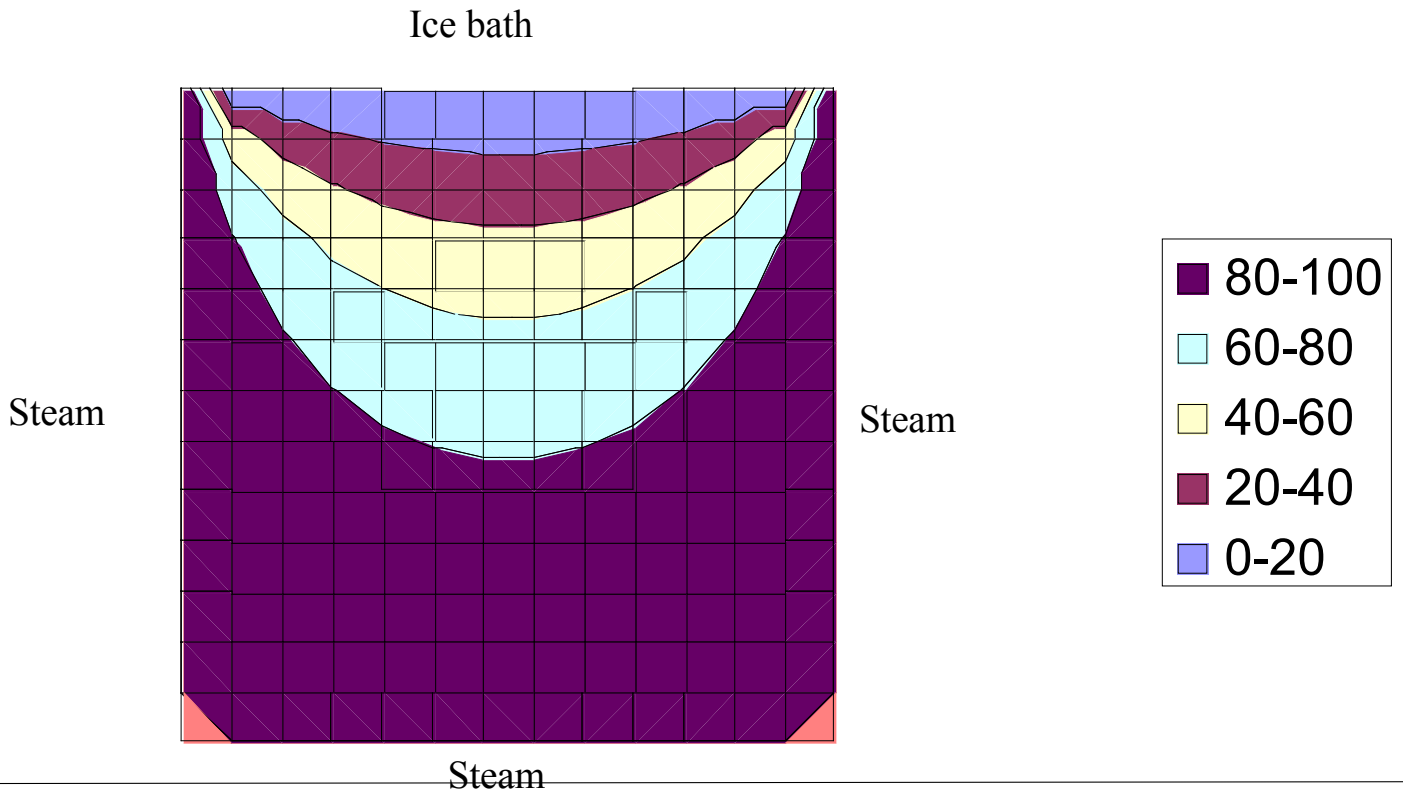
Meaning of Scalability Function

- To maintain efficiency when increasing *p*, we must increase *n*
- Maximum problem size is limited by available memory, which increases linearly with *p*
- Scalability function shows how memory usage per processor must grow to maintain efficiency
- If the scalability function is a constant this means the parallel system is perfectly scalable

Interpreting Scalability Function



Steady State Heat Distribution Problem



Solving the Problem

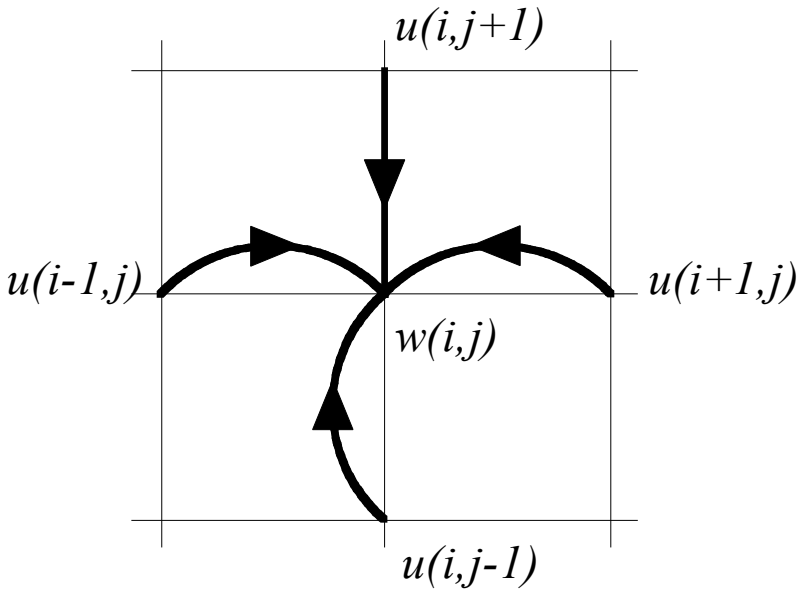
- Underlying PDE is the Poisson equation

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

- 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

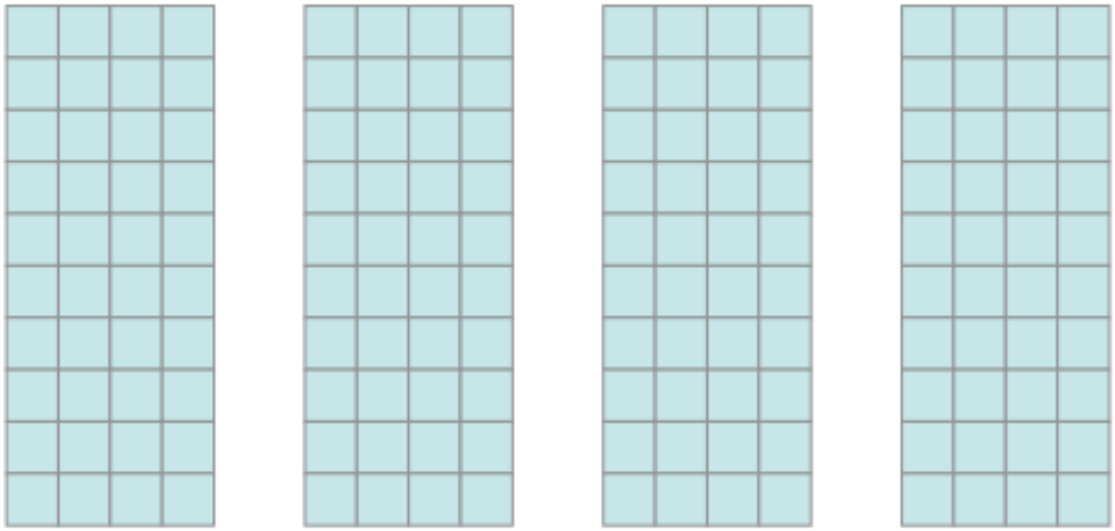
```
w[i][j] = (u[i-1][j] + u[i+1][j] +
           u[i][j-1] + u[i][j+1]) / 4.0;
```



Parallel Algorithm 1

- Associate primitive task with each matrix element
- Agglomerate tasks in contiguous rows (rowwise block striped decomposition)
- Add rows of ghost points above and below rectangular region controlled by process

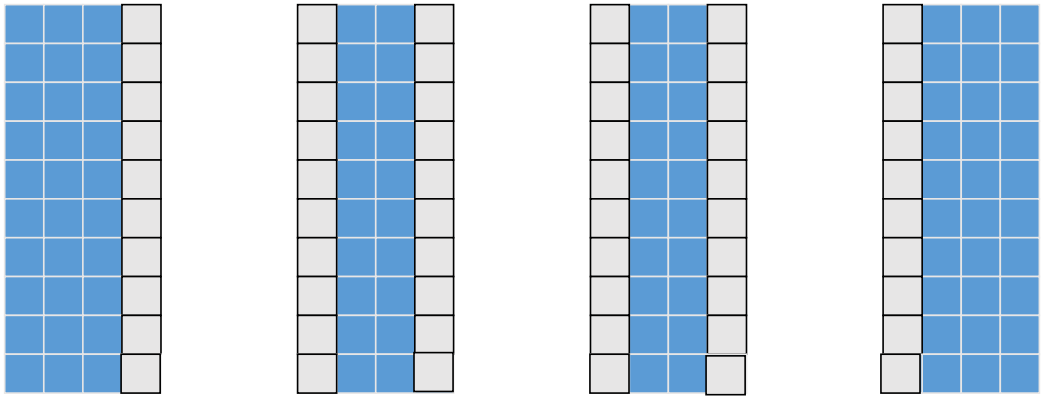
Agglomerate and map



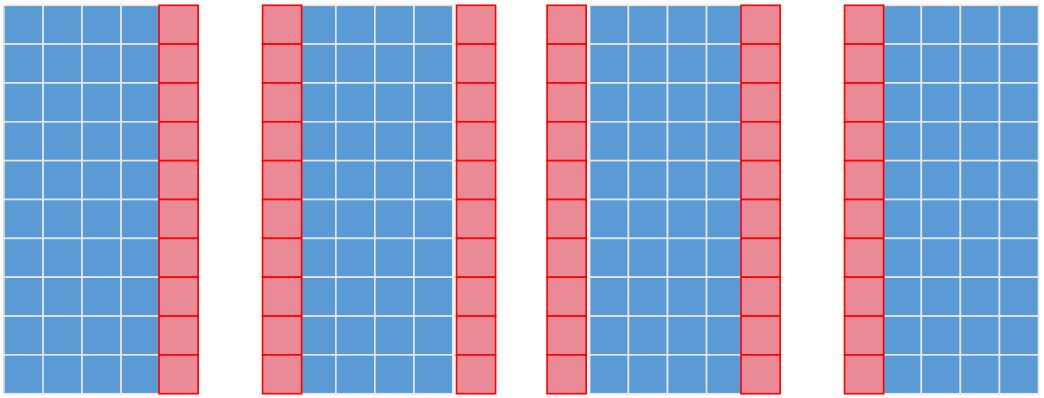
How does the communication work?

Communication Still Needed

- Exchange between columns
- Values in black cells cannot be computed without access to values held by other tasks



Matrices Augmented with Ghost Points

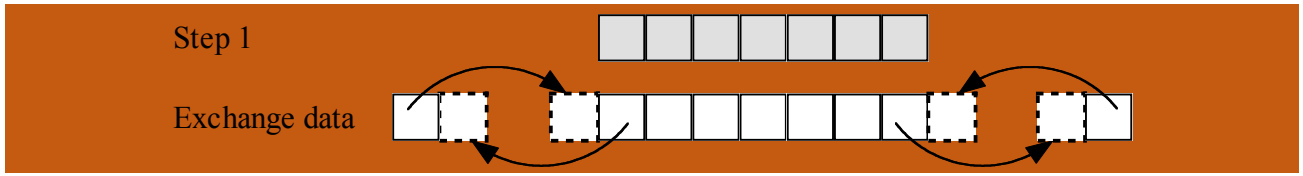


Red cells are the ghost points.

Ghost Points

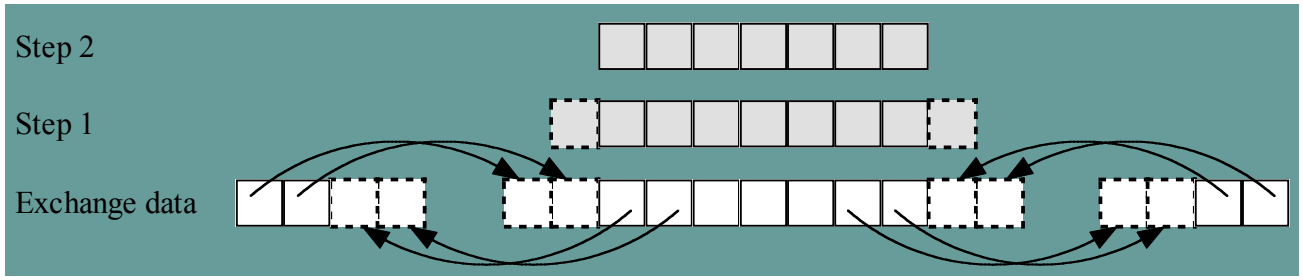
- Ghost points: memory locations used to store redundant copies of data held by neighboring processes
- Allocating ghost points as extra columns simplifies parallel algorithm by allowing same loop to update all cells

Communication to put ghost points in place

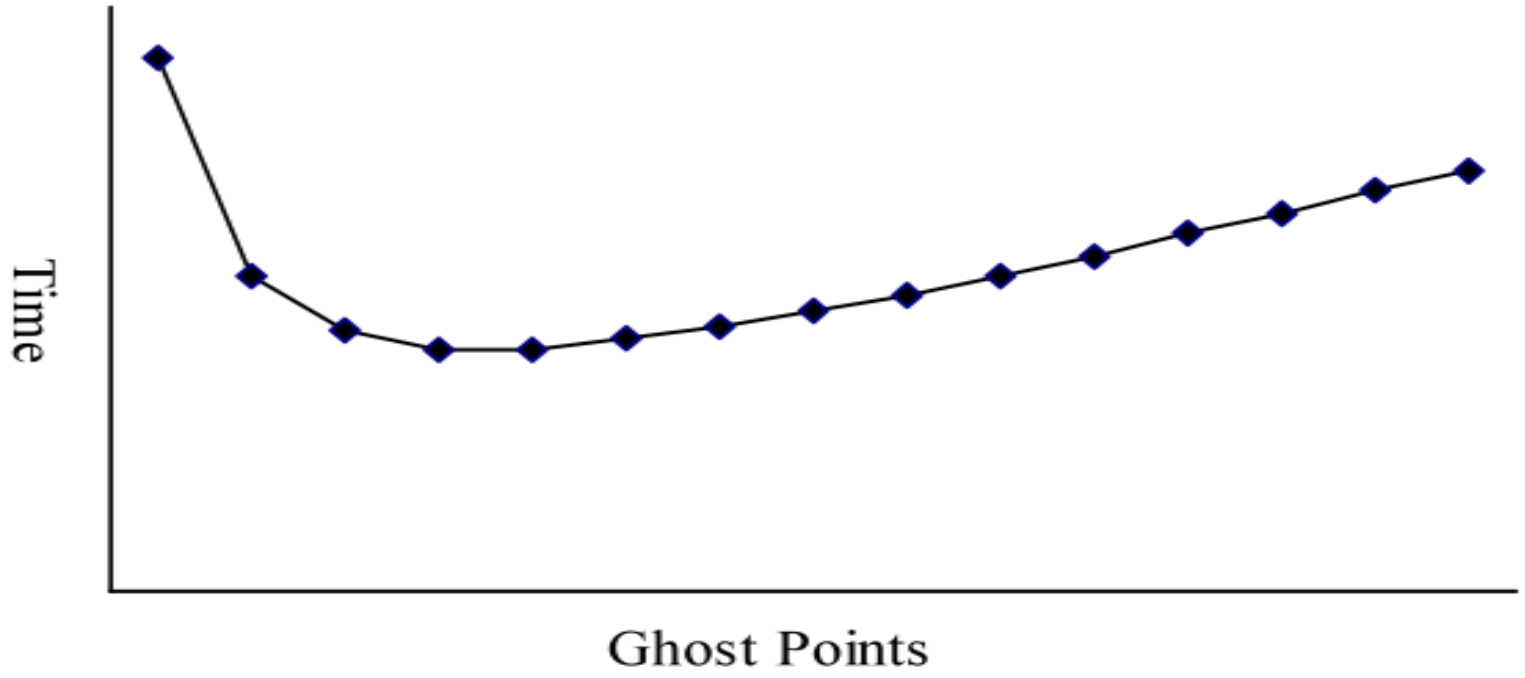


Improve communication efficiency

- Add more ghost points (second ghost column)
- Replicate data to reduce number of messages per computation



Communication time vs. number of ghost columns



Complexity Analysis

- Sequential time complexity:
  $\Theta(n^2)$ each iteration
- Parallel computational complexity:
  $\Theta(n^2 / p)$ each iteration
- Parallel communication complexity:
  $\Theta(n)$ each iteration (two sends and two receives of n elements)

Isoefficiency Analysis

$$W = K(N)T_O(W, p)$$

- Sequential time complexity: $\Theta(n^2)$
- Parallel overhead: $\Theta(pn)$
- Isoefficiency relation: $n^2 = Cnp \Rightarrow n = Cp$
- *M(f(p))/p*

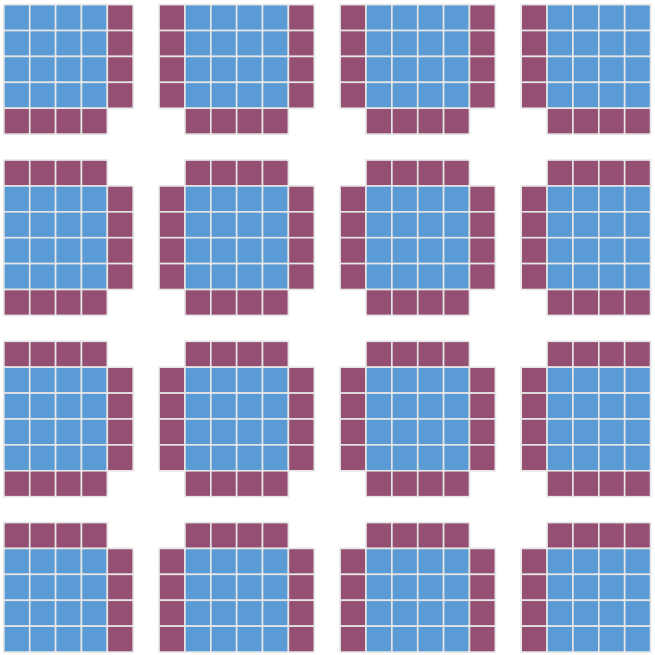
$$\boxed{M(Cp) / p = C^2 p^2 / p = C^2 p}$$

- This implementation has poor scalability

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

Complexity Analysis

- Sequential time complexity:
  $\Theta(n^2)$ each iteration
- Parallel computational complexity:
  $\Theta(n^2 / p)$ each iteration
- Parallel communication complexity:
  $\Theta(n / \sqrt{p})$ each iteration (four sends and four receives of $n / \sqrt{p}$ elements each)

Isoefficiency Analysis

- Sequential time complexity: $\Theta(n^2)$
- Parallel overhead: $\Theta(n\sqrt{p})$
- Isoefficiency relation:
  $n^2 \geq Cn\sqrt{p} \Rightarrow n \geq C\sqrt{p}$

$$M(C\sqrt{p})/p = C^2 p/p = C^2$$

- This system is perfectly scalable

Data Decomposition Options

- Have *n* elements and *p* processors
- How do we best distribute the n elements to p processors?
- Interleaved (cyclic)
  - Easy to determine “owner” of each index
- Block
  - Balances loads
  - More complicated to determine owner if *n* not a multiple of *p*

Block Decomposition Options

- Want to balance workload when n not a multiple of p
- Each process gets either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ elements
  - $\lceil x \rceil$ ceiling: smallest integer not less than x
  - $\lfloor x \rfloor$ floor: largest integer not greater than x

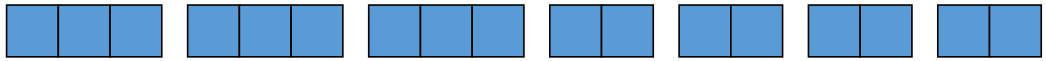
- Seek simple expressions
  - Find low, high indices given an owner
  - Find owner given an index

Method #1

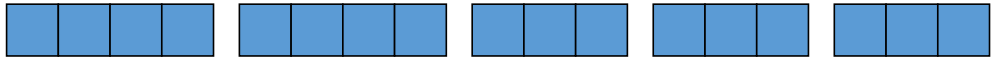
- Let $r = n \bmod p$
- If $r = 0$, all blocks have same size
- Else
  - First r blocks have size $\lceil n/p \rceil$
  - Remaining $p-r$ blocks have size $\lfloor n/p \rfloor$

Examples

17 elements divided among 7 processes



17 elements divided among 5 processes



17 elements divided among 3 processes

Let $r = n \bmod p$

If $r = 0$, all blocks have same size

Else

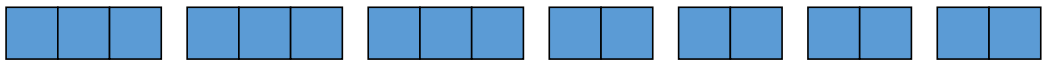
First r blocks have size $\lceil n/p \rceil$

Remaining p-r blocks have size $\lfloor n/p \rfloor$

Method #1 Calculations

- Indexing starts with 0
- First element controlled by process *i* $i\lfloor n/p \rfloor + \min(i,r)$
- Last element controlled by process *i* $(i+1)\lfloor n/p \rfloor + \min(i+1,r) - 1$
- Process controlling element *j* $\min(\lfloor j/(\lfloor n/p \rfloor + 1) \rfloor, \lfloor (j-r)/\lfloor n/p \rfloor \rfloor)$

17 elements divided among 7 processes



Method #2

- Scatters larger blocks among processes
- First element controlled by process *i* $\lfloor in / p \rfloor$
- Last element controlled by process *i* $\lfloor (i+1)n / p \rfloor - 1$
- Process controlling element *j* $\lfloor p(j+1) - 1) / n \rfloor$

Examples

First element controlled by process *i*

$$\lfloor in / p \rfloor$$

17 elements divided among 7 processes

17 elements divided among 5 processes

17 elements divided among 3 processes

Comparing Methods

Our choice

Operations	Method 1	Method 2
Low index	4	2
High index	6	4
Owner	7	4

Assuming no operations for "floor" function

Pop Quiz

- Illustrate how block decomposition method #2 would divide 13 elements among 5 processes.

First element controlled by process *i*

$$\lfloor in / p \rfloor$$

$13(0)/ 5 = 0$ $13(2)/ 5 = 5$ $13(4)/ 5 = 10$

$13(1)/5 = 2$ $13(3)/ 5 = 7$

Block Decomposition Macros

```
#define BLOCK_LOW(id,p,n)   ((i)*(n)/(p))

#define BLOCK_HIGH(id,p,n) \
        (BLOCK_LOW((id)+1,p,n)-1)

#define BLOCK_SIZE(id,p,n) \
        (BLOCK_LOW((id)+1)-BLOCK_LOW(id))

#define BLOCK_OWNER(index,p,n) \
        (((p)*(index)+1)-1)/(n))
```

Implementation

- Grid_decomposition.f90
- Make use of MPI virtual topologies

MPI – Virtual Topologies

- In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape”
- The two main types of topologies supported by MPI are Cartesian (grid) and Graph.
- MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology
- Virtual topologies are built upon MPI communicators and groups
- Must be "programmed" by the application developer.

Why use MPI Topologies

- Convenience
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.

- Communication Efficiency
  - Some hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

How to use a Virtual Topology

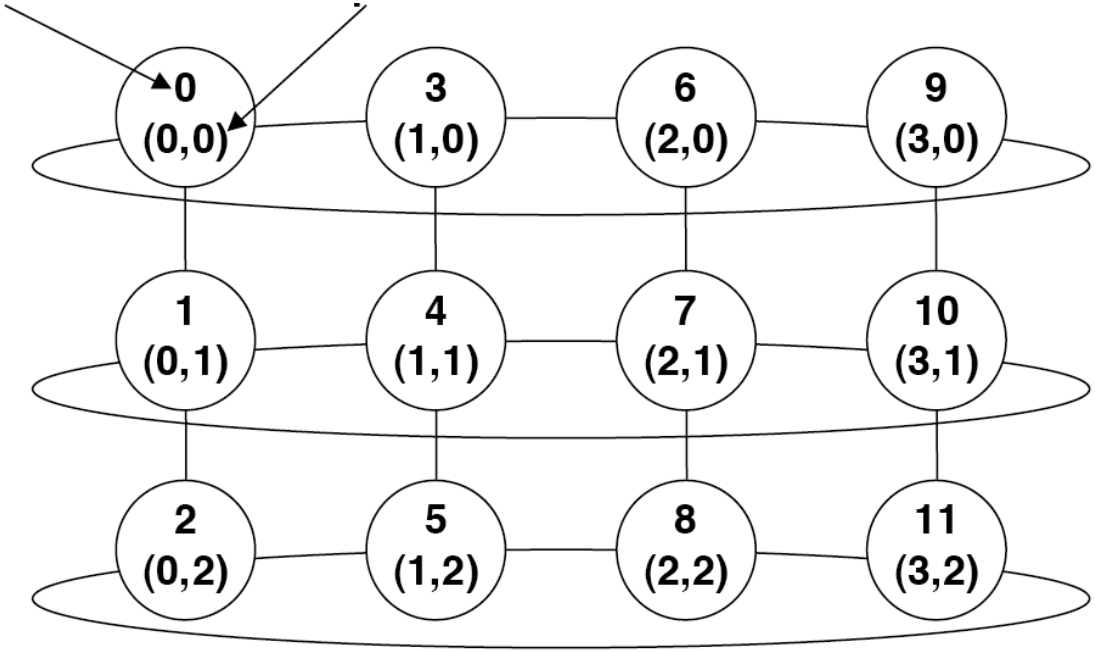
- Creating a topology produces a new communicator.
- MPI provides mapping functions:
  - to compute process ranks, based on the topology naming scheme,
  - and vice versa.

Topology Types

- Cartesian Topologies
  - each process is connected to its neighbor in a virtual grid,
  - boundaries can be cyclic, or not,
  - processes are identified by Cartesian coordinates,
  - of course, communication between any two processes is still allowed.
- Graph Topologies
  - general graphs,
  - not covered here.

Example - 2d Cylinder

- Ranks and Cartesian process coordinates



Creating 2-D Virtual Grid of Processes

- MPI_Dims_create
  - Input parameters
    - Total number of processes in desired grid
    - Number of grid dimensions
  - Returns number of processes in each dim
- MPI_Cart_create
  - Creates communicator with Cartesian topology

MPI_Cart_create

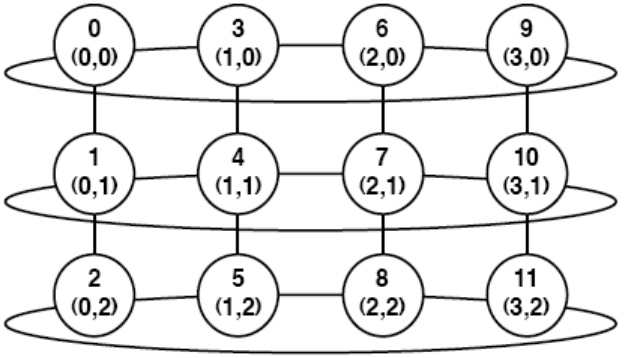
- Fortran: MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, *COMM_CART, IERROR*)

  INTEGER :: COMM_OLD, NDIMS, DIMS(*)

  LOGICAL :: PERIODS(*), REORDER

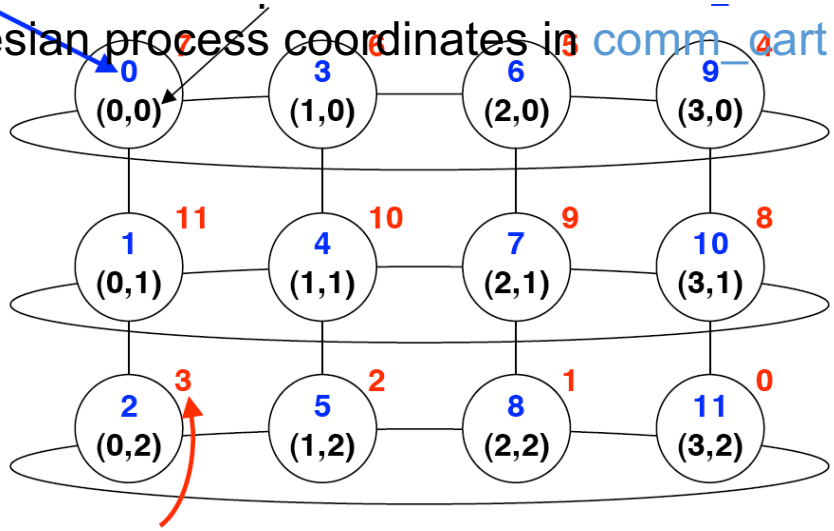
  INTEGER :: COMM_CART, IERROR

```
comm_old  =  MPI_COMM_WORLD
    ndims =  2
     dims = ( 4,        3        )
  periods = ( 1/.true., 0/.false. )
  reorder =  see next slide
```



Example - 2d Cylinder

- Ranks and Cartesian process coordinates in comm_cart



- Ranks in comm and comm_cart my differ, if reorder =1 or .TRUE.
- This reordering can allow MPI to optimize communications

Using MPI_Dims_create and MPI_Cart_create

```
MPI_Comm cart_comm;
int p;
int periodic[2];
int size[2];
...
size[0] = size[1] = 0;
MPI_Dims_create (p, 2, size);
periodic[0] = periodic[1] = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, size,
                 1, &cart_comm);
```

Useful Grid-related Functions

- MPI_Cart_rank
  - Given coordinates of process in Cartesian communicator, returns process rank
- MPI_Cart_coords
  - Given rank of process in Cartesian communicator, returns process' coordinates

MPI_CART_SHIFT

- Subroutine MPI_Cart_shift(comm, direction, displ, source, dest, ierr)

```
!**create cartesian topology for processes
      dims(1) = nrow      ! number of rows
      dims(2) = mcol      ! number of columns
      period(0) = .true.  ! cyclic in this direction
      period(1) = .false. ! no cyclic in this direction
      call MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder,
comm2D, ierr)
      call MPI_Comm_rank(comm2D, me, ierr)
      call MPI_Cart_coords(comm2D, me, ndim, coords, ierr)

      direction =  0   ! shift along the 1st index (0 or 1)
      displ =  1       ! shift by  1

      call MPI_Cart_shift(comm2D, direction, displ, source, dest, ierr)
```

Mapping functions

- Fortran: MPI_CART_COORDS(COMM_CART, RANK, MAXDIMS, *COORDS, IERROR*)

  INTEGER :: COMM_CART, RANK

  INTEGER :: MAXDIMS, *COORDS(*), IERROR*
- Fortran: MPI_CART_RANK(COMM_CART, COORDS, *RANK, IERROR*)

  INTEGER :: COMM_CART, COORDS(*)

  INTEGER :: *RANK, IERROR*
- Each process gets its own coordinates with
  - MPI_Comm_rank(comm_cart, my_rank, ierror)
  - MPI_Cart_coords(comm_cart, my_rank, maxdims, my_coords, ierror)

Data Exchange

- Using non-blocking send and receive
- Copy data in buffers
- Exchange data
- Copy data into blocks
- Optimization
  - Don’t copy use MPI types

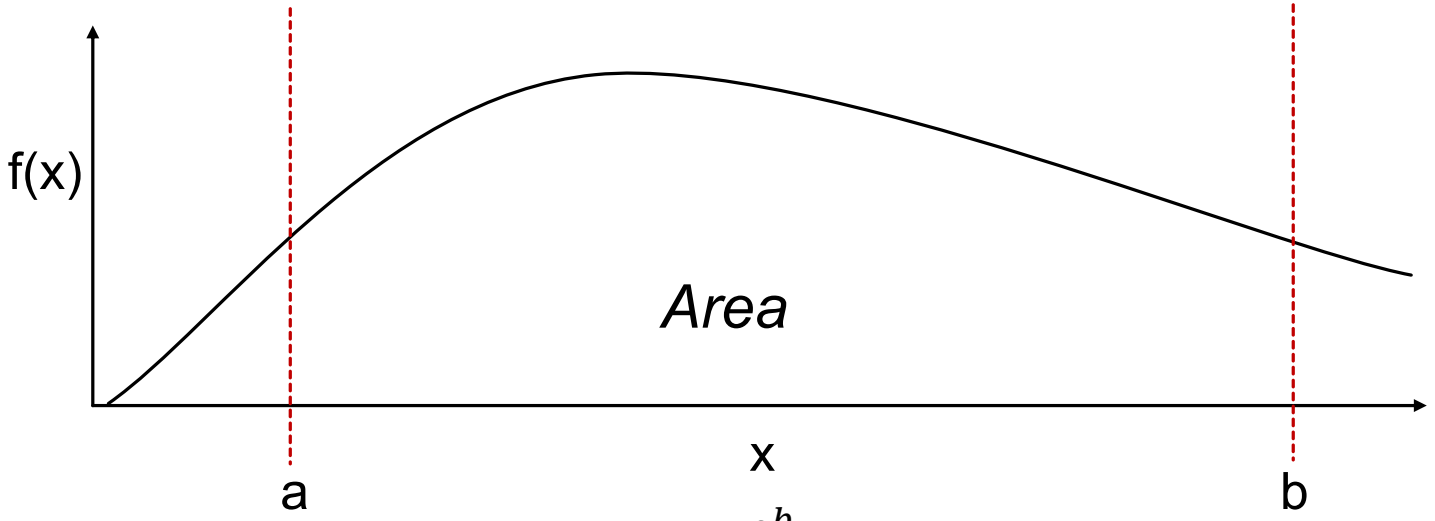
Sending Rows of a Matrix

- A(n,m)
- A(row, j) for j=1,…ny is not adjacent in memory
  - Copy into a 1D array and Send
  - Implemented in grid_decomposition.f90

MPI Type vector

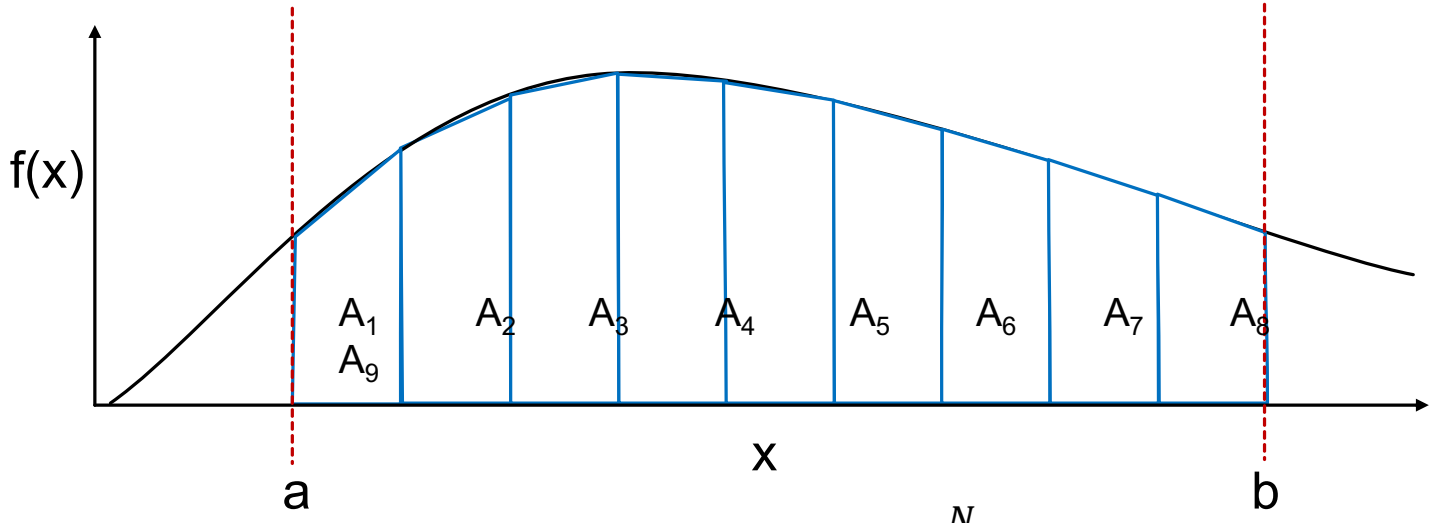
- Create a single datatype representing elements separated by a constant stride in memory
  - m items separated by a stride of n
  - MPI_Type_vector(m, 1, n, MPI_DOUBLE_PRECISION, newtype, ierr)
  - MPI_Type_commit(newtype, ierr)
- Sending new becomes
  - MPI_SEND(a(row, 1), 1, newtype …)
  - See grid_type.f90

Integration: Calculus



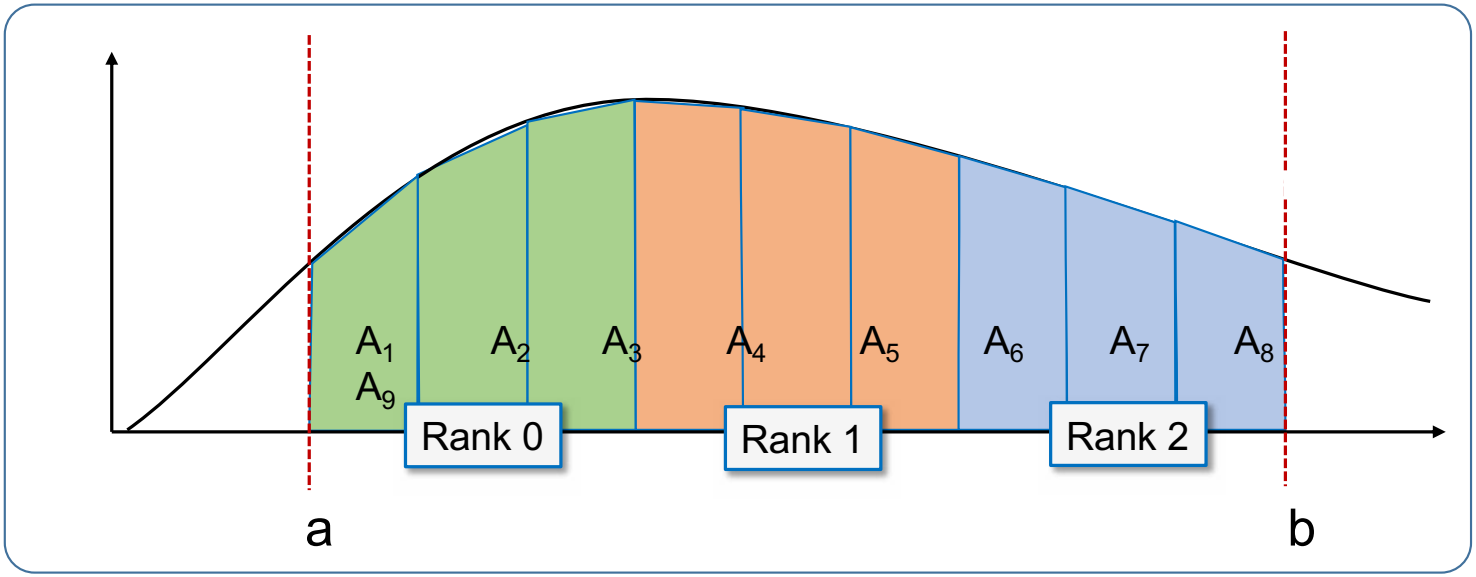
$$Area = \int_a^b f(x)dx$$

Integration: Trapezoidal Rule



$$Area = \int_a^b f(x)dx \approx \sum_{i=1}^{N} A_i$$

Work Distribution (Load-Balancing)



- Idea: Assign each MPI rank a different range in x
- Sum over areas at the end via Allreduce

Exercise 1: Integration

- We've started the problem setup already:

```
ntrap = 1000000/num_proc
```

- Local limits of integration : myxone, myxtwo
- Use my_rank and num_proc to modify these limits appropriately…

```
xone = 1.0
xtwo = 2.0

deltax = (xtwo-xone)
myxone = xone
myxtwo = myxone+deltax
```

… then run the code!

Exercise 2: Collatz Sequences

Edit this file: CSDM-HPC_clinic_2017/
MPI / session1 /exercises/
mpi_collatz.{f90,cpp}

Generating a Collatz Sequence

1. Pick any integer N
2. If N =1, stop
3. If N > 1:
   - If N is even; N -> N/2
   - If N is odd ; N -> 3N+1
4. GOTO 2

Collatz Conjecture:

Such a sequence initiated for any integer N will eventually terminate at 1. i.e., there are no infinite sequences

Collatz Sequencing

8: 8, 4, 2, 1
(length 4)
16: 16, 8, 4, 2, 1
(length 5)
3: 3, 10, 5, 16, 8, 4, 2, 1
(length 8)

Exercise:

- Your code template is set up so that each of your *P* MPI ranks computes the maximum sequence length occurring for *N*'s in the interval [1, *P* million]
- Parallelize this code so that
  - Each rank examines a unique subinterval within [1, *P* million]
  - The maximum length found locally is reduced globally using Allreduce
  - The integer *N* whose sequence is the longest is also reduced/communicated globally using Allreduce (slightly tricky…)