# MPI Day-2:

Performance/Scaling
Non-blocking communication
Communicators/Groups/Topology
One-side communication

FEI MAO

SHARCNET

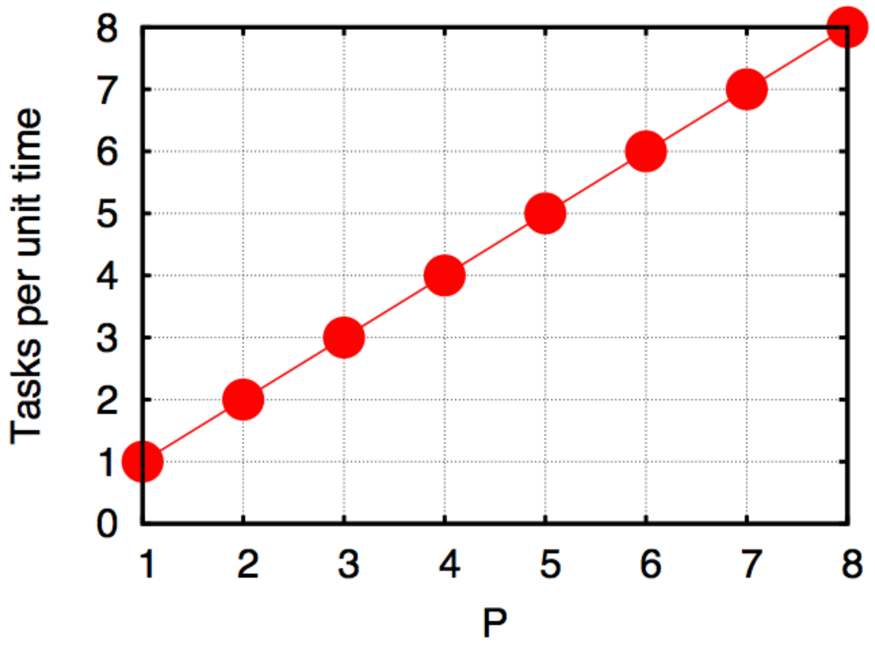
## Performance/Scaling

## Scaling — Throughput

- How a problem's throughput scales as processor number increases ("strong scaling").
- In this case, linear scaling:

$$\mathbf{H} \propto \mathbf{P}$$

- This is Perfect scaling.

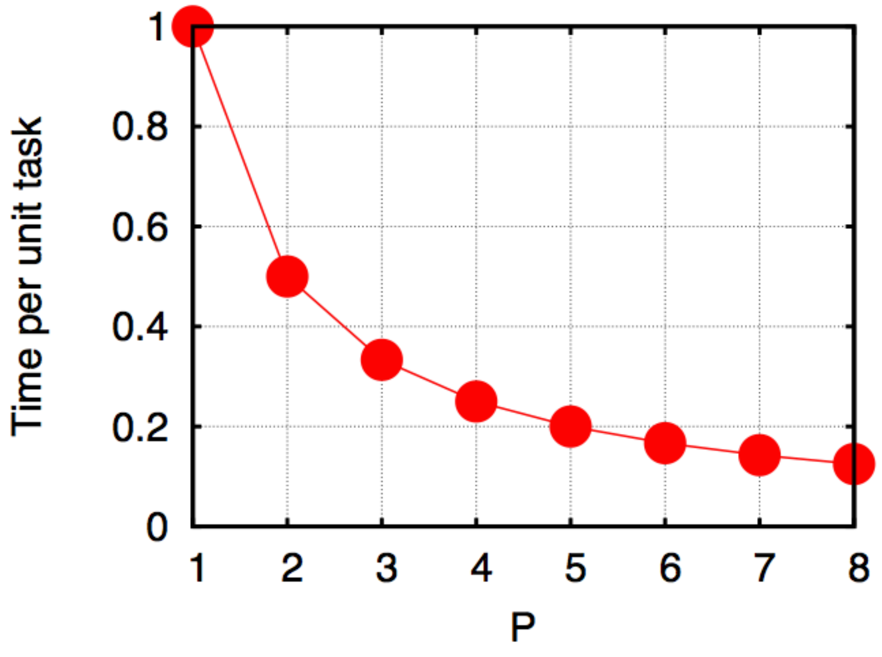


## Scaling – Time

- How a problem's timing scales as processor number increases.
- Measured by the time to do one unit. In this case, inverse linear scaling:

$$\mathbf{T \propto 1/P}$$

- Again this is the ideal case, or "embarrassingly parallel".

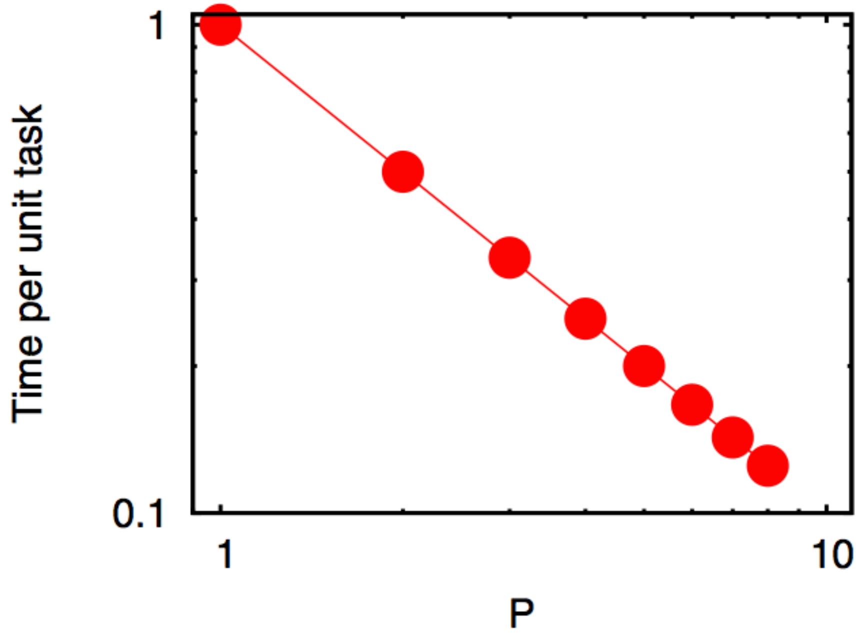


## Scaling – Time

- How a problem's timing scales as processor number increases.
- Measured by the time to do one unit. In this case, inverse linear scaling:

$$\mathbf{T \propto 1/P}$$

- Again this is the ideal case, or "embarrassingly parallel".

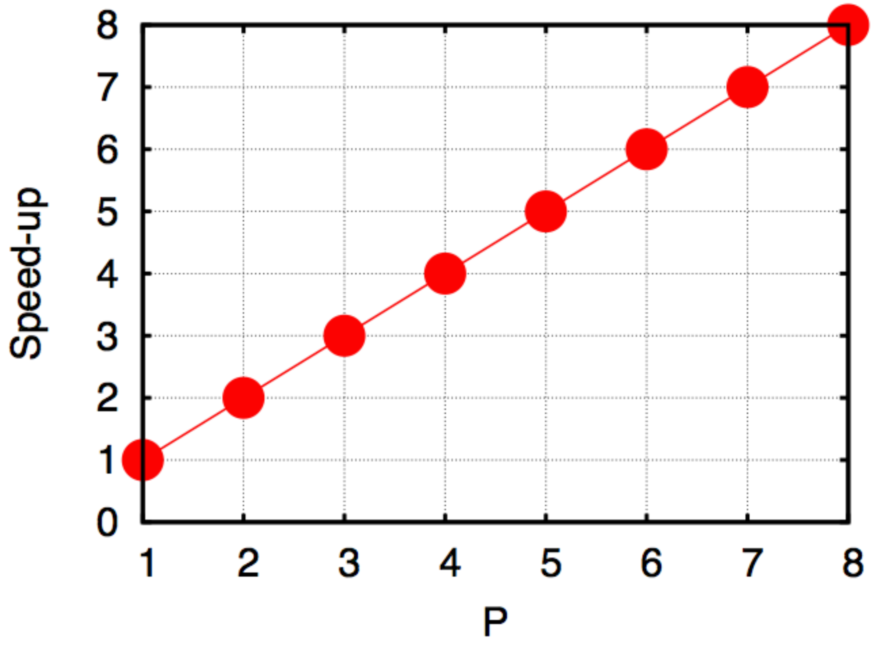


## Scaling – Speedup

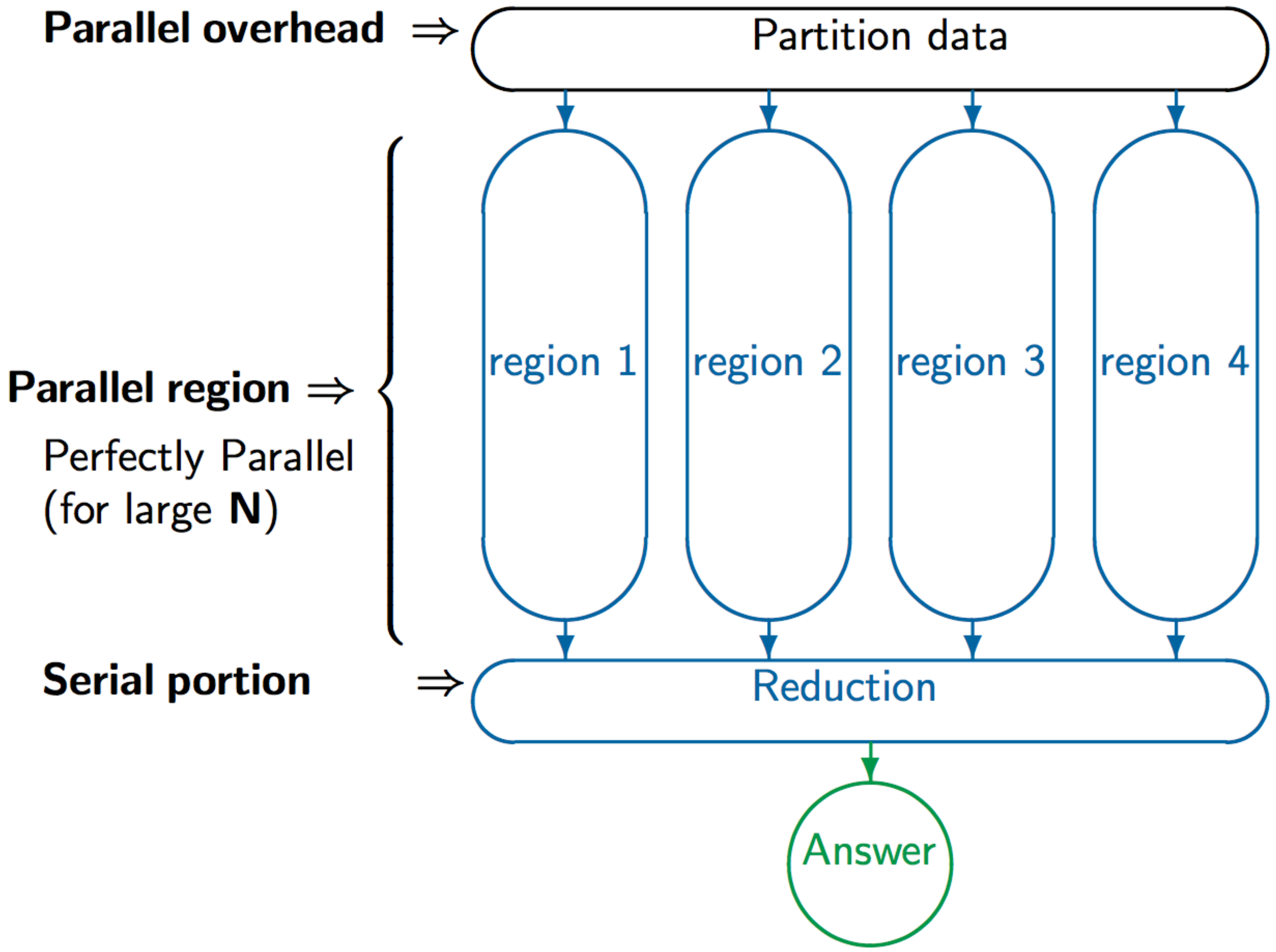
- How much faster the problem is solved as processor number increases.
- Measured by the serial time divided by the parallel time

$$S = \frac{T_{serial}}{T(P)} \propto P$$

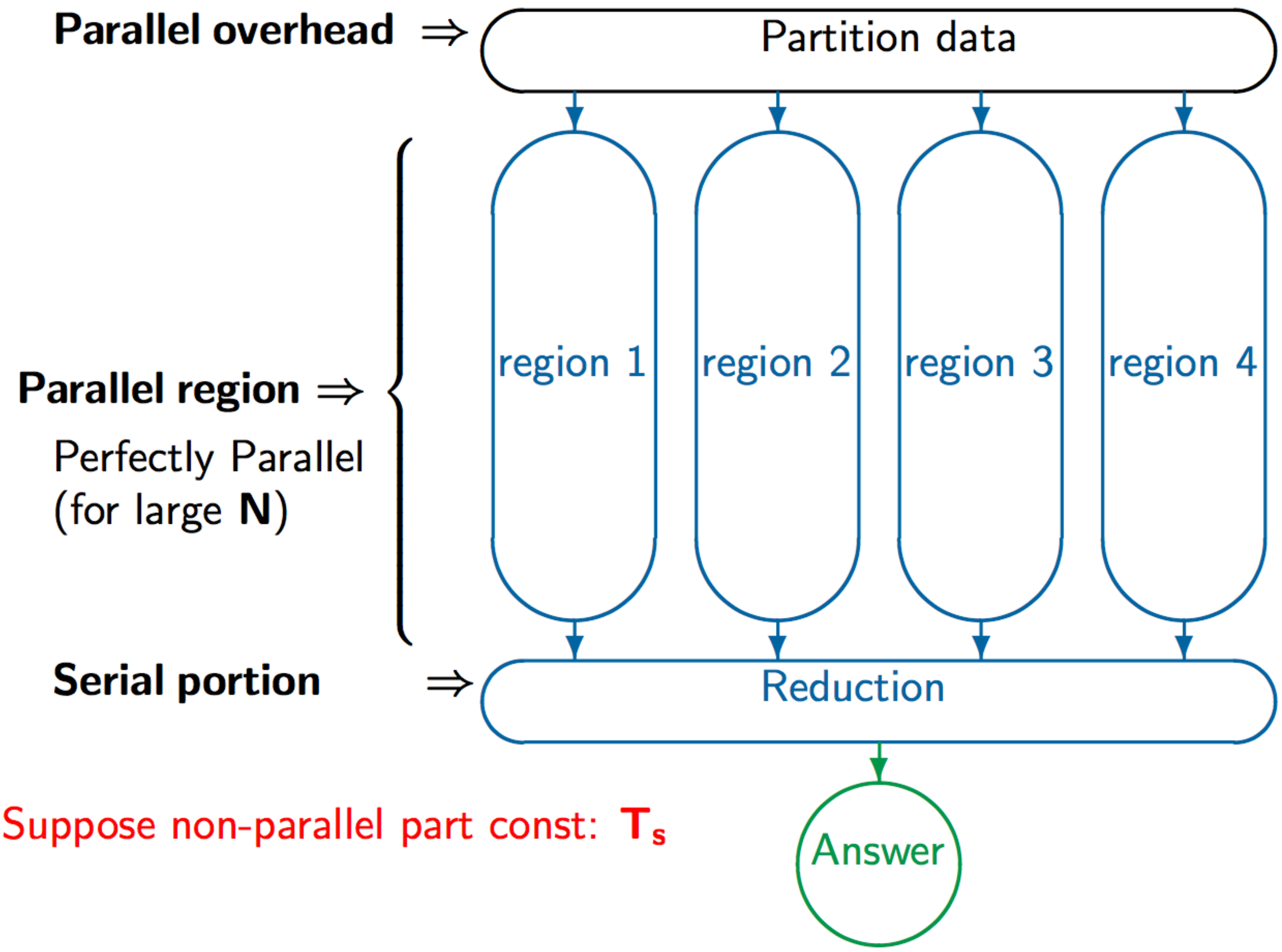
- For embarrassingly parallel applications: Linear speed up.



## Serial Overhead



## Serial Overhead



Suppose non-parallel part const: $\mathbf{T_s}$

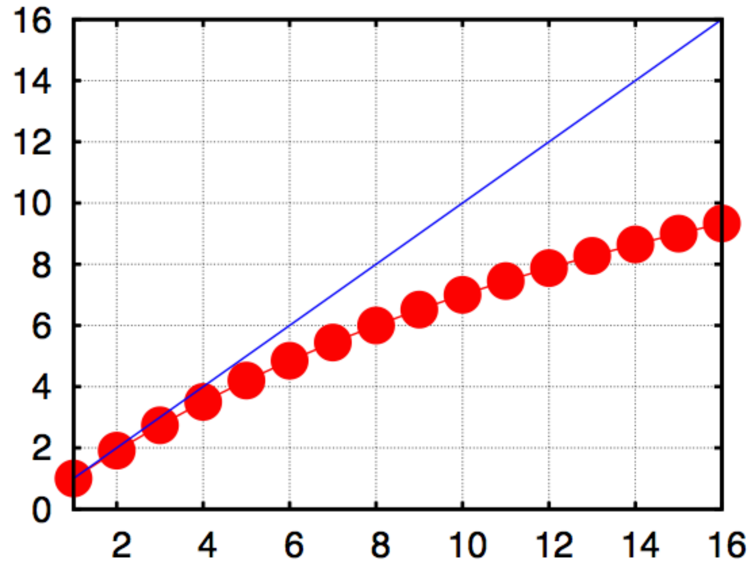
## Amdahl's law

Speed-up (without parallel overhead):

$$\mathbf{S} = \frac{\mathbf{NT_1 + T_s}}{\frac{\mathbf{NT_1}}{\mathbf{P}} + \mathbf{T_s}}$$

or, calling $\mathbf{f = T_s/(T_s + NT_1)}$ the serial fraction,

$$\mathbf{S} = \frac{\mathbf{1}}{\mathbf{f + (1 - f)/P}}$$



(for $\mathbf{f = 5\%}$)

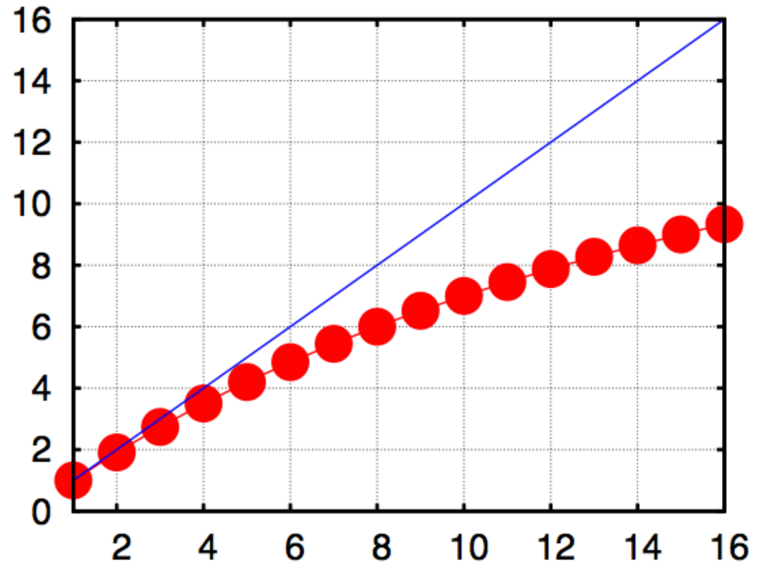
## Amdahl's law

Speed-up (without parallel overhead):

$$\mathbf{S} = \frac{\mathbf{NT_1 + T_s}}{\frac{\mathbf{NT_1}}{\mathbf{P}} + \mathbf{T_s}}$$

or, calling $\mathbf{f = T_s/(T_s + NT_1)}$ the serial fraction,

$$\mathbf{S} = \frac{\mathbf{1}}{\mathbf{f + (1 - f)/P}} \quad \xrightarrow{\mathbf{P \to \infty}} \quad \frac{\mathbf{1}}{\mathbf{f}}$$



Serial part dominates asymptotically.

Speed-up limited, no matter size of **P**.

And this is the overly optimistic case!

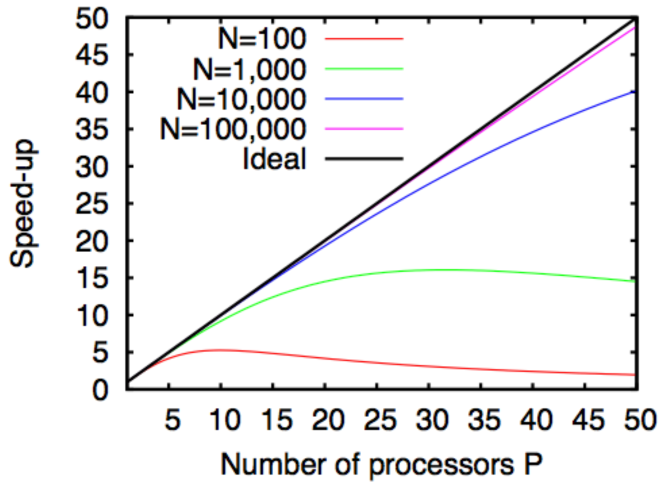
(for $\mathbf{f = 5\%}$)

## Trying to beat Amdahl's law

**Scale up!**

The larger **N**, the smaller the serial fraction:

$$\mathbf{f(P) = \frac{P}{N}}$$

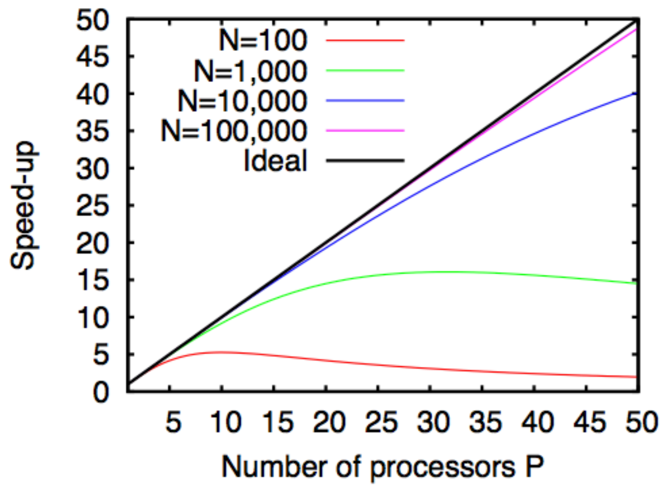


## Trying to beat Amdahl's law

**Scale up!**

The larger **N**, the smaller the serial fraction:

$$\mathbf{f(P) = \frac{P}{N}}$$



Weak scaling: Increase problem size while increasing **P**

$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

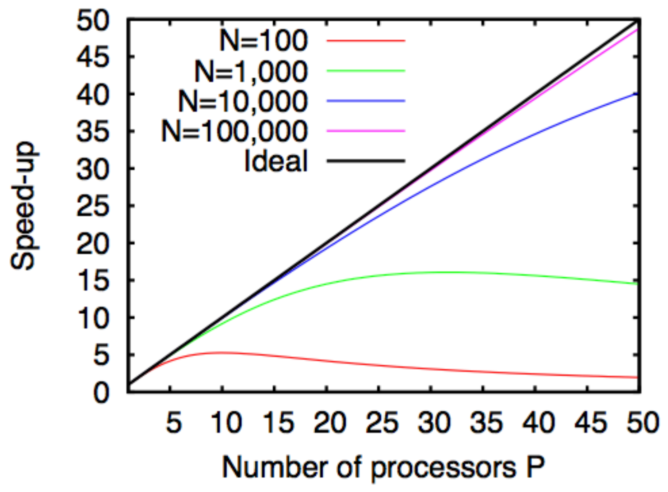
Good weak scaling means this time approaches a constant for large **P**.

# Trying to beat Amdahl's law

**Scale up!**

The larger **N**, the smaller the serial fraction:

$$f(P) = \frac{P}{N}$$



Weak scaling: Increase problem size while increasing **P**

$$\mathrm{Time}_{\mathrm{weak}}(P) = \mathrm{Time}(N = n \times P, P)$$

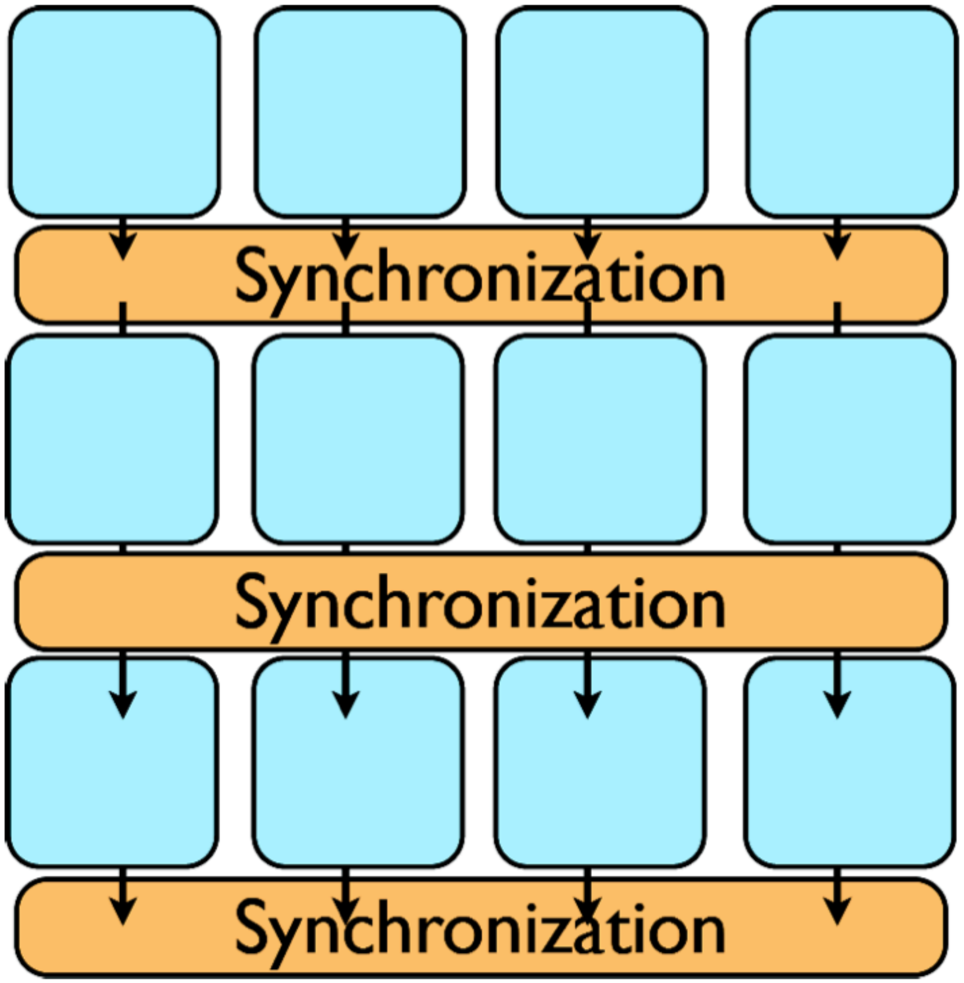
Good weak scaling means this time approaches a constant for large **P**.

Gustafson's Law

Any large enough problem can be efficiently parallelized (Efficiency$\longrightarrow$1).

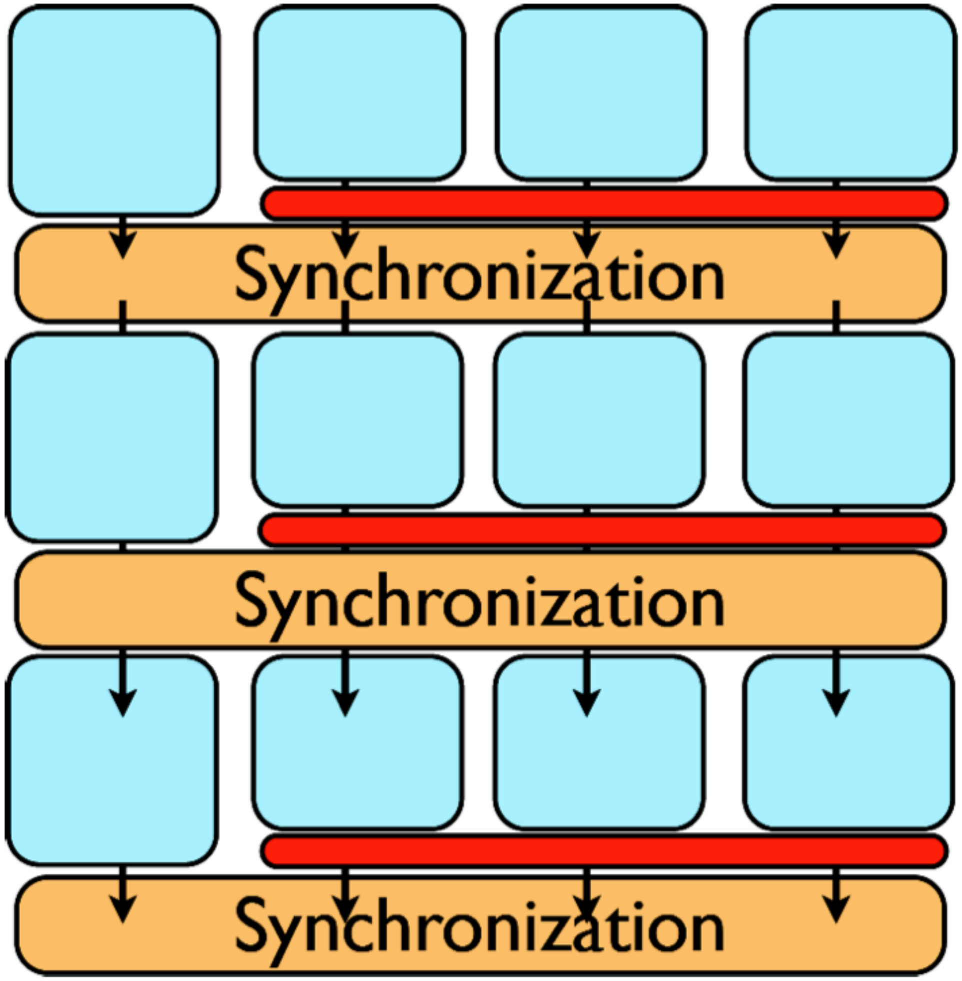
## Synchronization Overhead

- Most problems are not purely concurrent.
- Some level of synchronization or exchange of information is needed between tasks.
- While synchronizing, nothing else happens: increases Amdahl's **f**.
- And synchronizations are themselves costly.



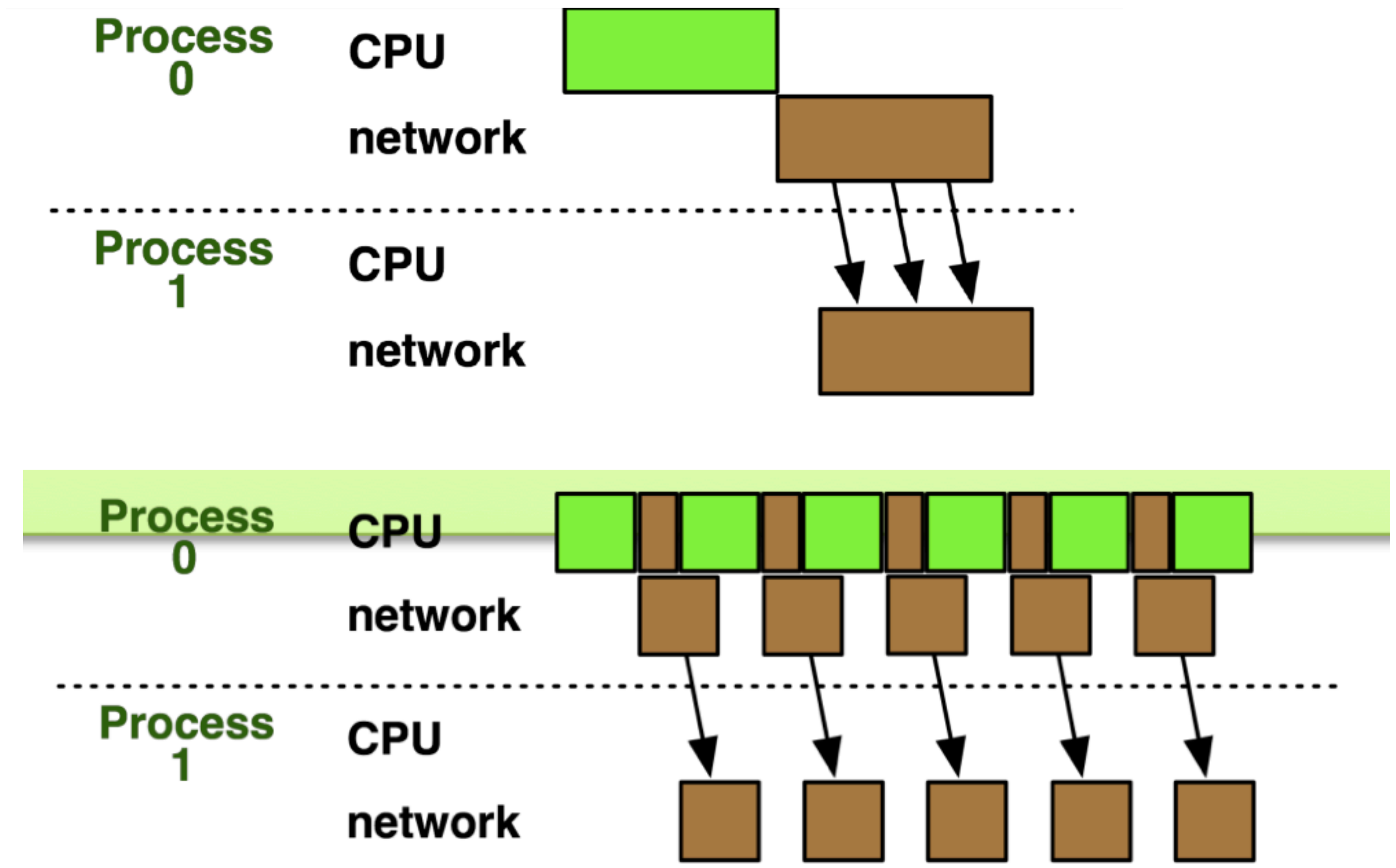
# Load Balancing

- The division of calculations among the processors may not be equal.
- Some processors would already be done, while others are still going.
- Effectively using less than **P** processors: This reduces the efficiency.
- Aim for load balanced algorithms.



## NONBLOCKING COMMUNICATION

# NONBLOCKING COMMUNICATION



## NON-BLOCKING COMMUNICATION

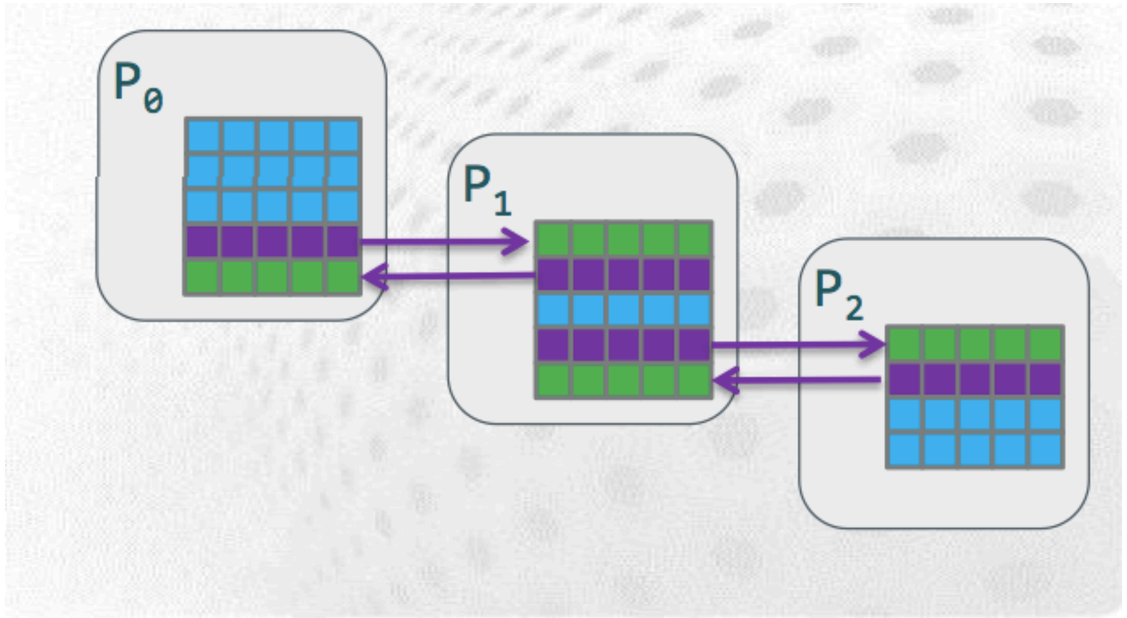
- Non-blocking sends and receives
  - MPI_Isend & MPI_Irecv
  - returns immediately and sends/receives in background
- Enables some computing concurrently with communication
- Avoids many common dead-lock situations
- Also non-blocking collective operations in MPI 3.0

## NON-BLOCKING COMMUNICATION

- Have to finalize send/receive operations
  - MPI_Wait, MPI_Waitall,...
    - Waits for the communication started with MPI_Isend or MPI_Irecv to finish (blocking)
  - MPI_Test,...
    - Tests if the communication has finished (non-blocking)
- You can mix non-blocking and blocking p2p routines
  - e.g.,receiveMPI_IsendwithMPI_Recv

## Typical usage pattern

- MPI_Irecv(ghost_data)
- MPI_Isend(border_data)
- Compute(ghost_independent_data)
- MPI_Waitall Compute(border_data)



## Non-blocking send

MPI_Isend(buf, count, datatype, dest, tag, comm, request)

- Parameters
  - Similar to MPI_Send but has an additional request parameter
- buf send buffer that must not be written to until one has checked that the operation is over
- request a handle that is used when checking if the operation has finished (integer in Fortran, MPI_Request in C)

## Non-blocking receive

MPI_Irecv(buf, count, datatype, source, tag, comm, request)

- parameters similar to MPI_Recv but has no status parameter
- buf receive buffer guaranteed to contain the data only after one has checked that the operation is over
- request a handle that is used when checking if the operation has finished

## Wait for non-blocking operation

MPI_Wait(request, status)

- Parameters
  - request handle of the non-blocking communication
  - status status of the completed communication
- A call to MPI_WAIT returns when the operation identified by request is complete

## Wait for non-blocking operation

MPI_Waitall(count, requests, status)

- Parameters
  - count number of requests
  - requests array of requests
  - status array of statuses for the operations that are waited for
- A call to MPI_Waitall returns when ***all*** operations identified by the array of requests are com

## Additional completion operations

other useful routines:

- **MPI_Waitany**
- **MPI_Waitsome**
- **MPI_Test**
- MPI_Testall
- MPI_Testany
- MPI_Testsome
- MPI_Probe

## Wait for non-blocking operations

MPI_Waitany(count, requests, index, status)

- Parameters
  - count number of requests
  - requests array of requests
  - index index of request that completed
  - status status for the completed operations
- A call to MPI_Waitany returns when one operation identified by the array of requests is complete

## Wait for non-blocking operations

MPI_Waitsome(count, requests, done, index, status)

- Parameters
  - count number of requests
  - requests array of requests
  - done number of completed requests
  - index array of indexes of completed requests
  - status array of statuses of completed requests
- A call to MPI_Waitsome returns when one or more operation identified by the array of requests is complete

## Non-blocking test for non-blocking operations

MPI_Test(request, flag, status)

- Parameters
  - request request
  - flag True if operation has completed status
  - status for the completed operations
- A call to MPI_Test is non-blocking. It allows one to schedule alternative activities while periodically checking for completion.

## Summary Non-blocking communication

- Non-blocking communication is usually the smarter way to do point-to-point communication in MPI
- Non-blocking communication realization
  - MPI_Isend
  - MPI_Irecv
  - MPI_Wait(all)
- MPI-3 contains also non-blocking collectives (MPI_Ibcast, MPI_Ireduce, etc)

## Exercise: A dead-lock situation

- Safe Code:

```
if (my_rank==0)
{
MPI_Send( &tosend, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
MPI_Recv( &toreceive, 1, MPI_INT, 1, 20, MPI_COMM_WORLD, &status);
}
else if(my_rank == 1)
{
MPI_Recv( &toreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
MPI_Send( &tosend, 1, MPI_INT, 0, 20, MPI_COMM_WORLD);
}
```

## Exercise: A dead-lock situation

- Deadlock Code:

```
if (my_rank==0)
{
MPI_Recv( &toreceive, 1, MPI_INT, 1, 20, MPI_COMM_WORLD, &status);
MPI_Send( &tosend, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
}
else if(my_rank == 1)
{
MPI_Recv( &toreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
MPI_Send( &tosend, 1, MPI_INT, 0, 20, MPI_COMM_WORLD);
}
```

- Task：
  - Modify code send_recv_2b.c with non-blocking send and recv functions.
  - Don’t forget to define “MPI_Request send_request,recv_request;” first.

# Exercise: A dead-lock situation

- Buffering dependent Code

```
if (my_rank==0)
{
MPI_Send( &tosend, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
MPI_Recv( &toreceive, 1, MPI_INT, 1, 20, MPI_COMM_WORLD, &status);
}
else if(my_rank == 1)
{
MPI_Send( &tosend, 1, MPI_INT, 0, 20, MPI_COMM_WORLD);
MPI_Recv( &toreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
}
```

**Success of this code is dependent on buffering. One of the send must buffer and return. Otherwise, deadlock occurs.**

## Message Buffering

- Definition of “completion” for MPI_Recv() is trivial – the data can now be used.
- Definition of “completion” for MPI_Send() is trickier. Completion implies that the data has been stored away such that the program is free to overwrite the send “message” buffer.
- -- **Non-local:** the data can be sent directly to the receive buffer.
- -- **Local (buffering)**: the data can be stored in a local buffer (system provided or user provided), in which case the send could return before the receive is initiated.

# More about Communication Modes

| Send Modes | MPI function | Completion Condition |
|---|---|---|
| Synchronous send | MPI_Ssend()<br>MPI_Issend() | A send will not complete until a matching receive has been posted and the matching receive has begun reception of the data. Completion of a synchronous send not only indicates that the send buffer can be reused, but also indicates that the receiver has reached a certain point in its execution |
| Buffered send | MPI_Bsend()<br>MPI_Ibsend() | Bsend() always completes (unless an error occurs) Completion is irrespective of the receiver. |
| Standard send | MPI_Send()<br>MPI_Isend() | message sent (no guarantee that the receive has started). It is up to MPI to decide what to do. |
| Ready send | MPI_Rsend()<br>MPI_Irsend() | may be used only when the a matching receive has already been posted |

"**Recommendations**: In general, use MPI_Send. If non-blocking routines are necessary, then try to use MPI_Isend or MPI_Irecv. Use MPI_Bsend only when it is too inconvenient to use MPI_Isend. The remaining routines, MPI_Rsend, MPI_Issend, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI."

## Write Safe Code

- A safe MPI program should not rely on system buffering for success.
- Any system will eventually run out of buffer space as message sizes are increased.
- User should design proper send/receive orders to avoid **deadlock**.
- The user **must NOT** overwrite the send buffer until the send (data transfer) is complete.
- The user **can NOT** use the receiving buffer before the receive is complete.

# Exercise: Non-blocking 1d diffusion

```c
initialize(uk, ukp1, numPoints, numProcs, myID);

for (step = 0; (step < maxsteps) && (maxdiff >= threshold); ++step) {

  double diff, maxdiff_local = 0.0;

  /* exchange boundary information */
  if (myID != 0)
    MPI_Send(&uk[1], 1, MPI_DOUBLE, leftNbr, 0, MPI_COMM_WORLD);
  if (myID != numProcs-1)
    MPI_Send(&uk[numPoints], 1, MPI_DOUBLE, rightNbr, 0, MPI_COMM_WORLD);
  if (myID != 0)
    MPI_Recv(&uk[0], 1, MPI_DOUBLE, leftNbr, 0, MPI_COMM_WORLD, &status);
  if (myID != numProcs-1)
    MPI_Recv(&uk[numPoints+1],1, MPI_DOUBLE, rightNbr, 0, MPI_COMM_WORLD, &status);

  /* compute new values for interior points */
  for (i = 2; i < numPoints; ++i) {
    ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
  }
  /* compute new values for boundary points
   * (no real need to do these separately, but it would allow us to
   * later overlap computation and communication with fewer code changes)
   */
  if (myID != 0) {
     int i=1;
     ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
  }
  if (myID != numProcs-1) {
     int i=numPoints;
     ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
  }

  /* check for convergence */
```

## Communicators and Groups

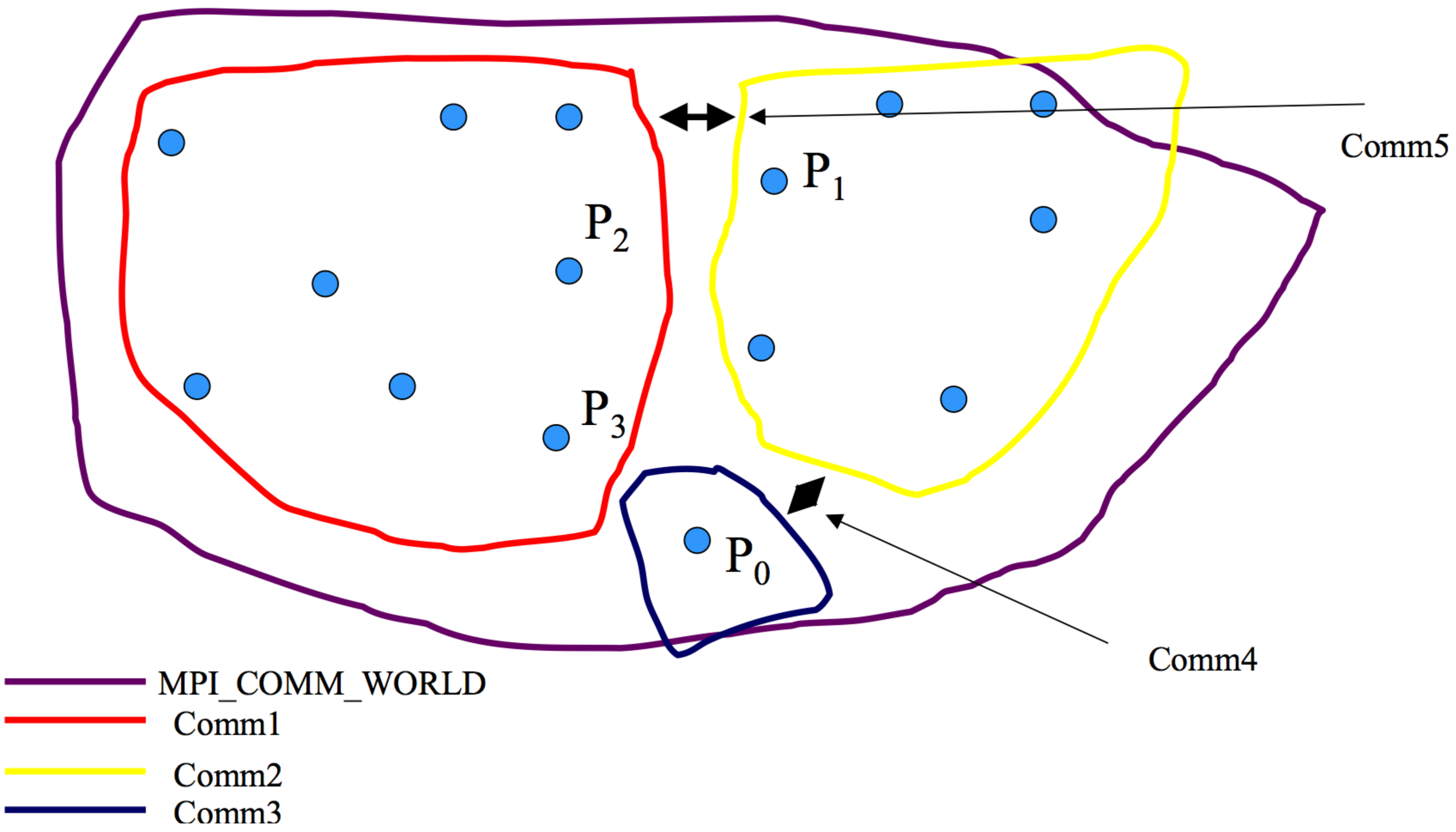
# Communicators and Groups

- Many MPI users are only familiar with the communicator MPI_COMM_WORLD
- A communicator can be thought of a handle to a group
- A group is an ordered set of processes
- Each process is associated with a rank
- Ranks are contiguous and start from zero
- For many applications (dual level parallelism) maintaining different groups is appropriate
- Groups allow collective operations to work on a subset of processes
- Information can be added onto communicators to be passed into routines

## Communicators and Groups

- While we think of a communicator as spanning processes, it is actually unique to a process
- A communicator can be thought of as a handle to an object (group attribute) that describes a group of processes
- An intracommunicator is used for communication within a single group
- An intercommunicator is used for communication between 2 disjoint groups

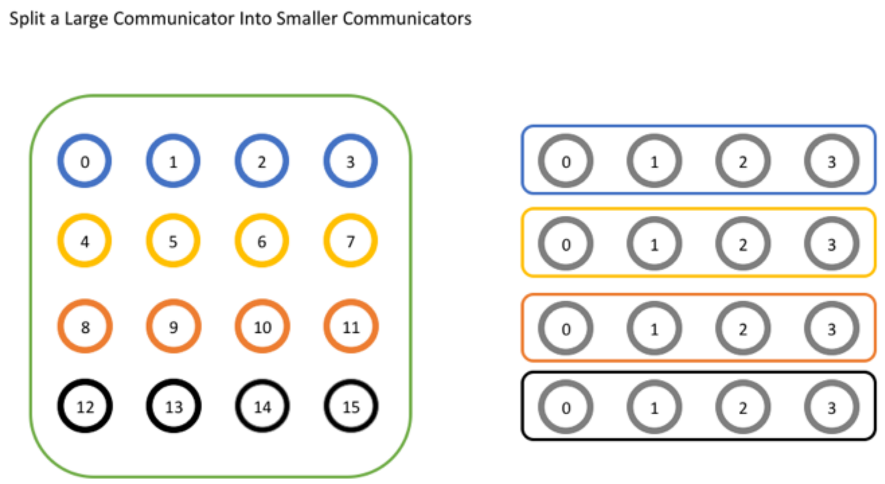
# Communicators and Groups



# Communicators and Groups

- Refer to previous slide
- There are 4 distinct groups
- These are associated with intracommunicators MPI_COMM_WORLD, comm1, and comm2, and comm3
- $P_3$ is a member of 2 groups and may have different ranks in each group(say 3 & 4)
- If $P_2$ wants to send a message to $P_1$ it must use MPI_COMM_WORLD (intracommunicator) or comm5 (intercommunicator)
- If $P_2$ wants to send a message to $P_3$ it can use MPI_COMM_WORLD (send to rank 3) or comm1 (send to rank 4)
- $P_0$ can broadcast a message to all processes associated with comm2 by using intercommunicator comm4

# Example of using multiple communicators



```
MPI_Comm_split(
   MPI_Comm comm,
   int color,
   int key,
   MPI_Comm* newcomm)
```

- As the name implies, MPI_Comm_split creates new communicators by "splitting" a communicator into a group of sub-communicators based on the input values color and key.
- The first argument, comm, is the communicator that will be used as the basis for the new communicators. This could be MPI_COMM_WORLD, but it could be any other communicator as well.
- The second argument, color, determines to which new communicator each processes will belong. All processes which pass in the same value for color are assigned to the same communicator.
- The third argument, key, determines the ordering (rank) within each new communicator. The process which passes in the smallest value for color will be rank 0, the next smallest will be rank 1, and so on.

# Example of using multiple communicators

```c
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
    world_rank, world_size, row_rank, row_size);
```

## Other communicator creation functions

- MPI_Comm_split is the most common communicator creation function
- MPI_Comm_dup is the most basic and creates a duplicate of a communicator.
- MPI_Comm_create creates a new communicator with communication group
- A more flexible way to create communicators using a new kind of MPI object, MPI_Group

## Overview of groups

- **Groups vs. Communicators**
  - A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. A group is always associated with a communicator object.
  - A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. In the simplest sense, the communicator is an extra "tag" that must be included with MPI calls.
  - From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

# Overview of groups

- Groups are dynamic - they can be created and destroyed during program execution.
- Processes may be in more than one group. They will have a unique rank within each group.
- MPI provides over 40 routines related to groups, communicators, and virtual topologies!
- Typical usage:
  1. Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
  2. Form new group as a subset of global group using MPI_Group_incl
  3. Create new communicator for new group using MPI_Comm_create
  4. Determine new rank in new communicator using MPI_Comm_rank
  5. Conduct communications using any MPI message passing routine
  6. When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free

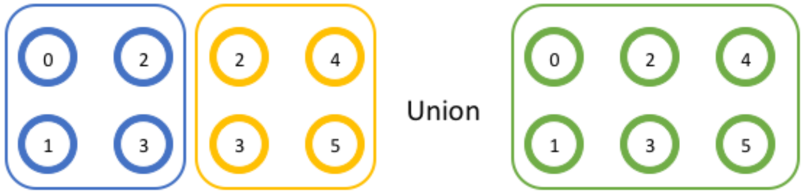
# Example of using groups

```c
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the group of processes in MPI_COMM_WORLD
MPI_Group world_group;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);

int n = 7;
const int ranks[7] = {1, 2, 3, 5, 7, 11, 13};

// Construct a group containing all of the prime ranks in world_group
MPI_Group prime_group;
MPI_Group_incl(world_group, 7, ranks, &prime_group);

// Create a new communicator based on the group
MPI_Comm prime_comm;
MPI_Comm_create_group(MPI_COMM_WORLD, prime_group, 0, &prime_comm);

int prime_rank = -1, prime_size = -1;
// If this rank isn't in the new communicator, it will be
// MPI_COMM_NULL. Using MPI_COMM_NULL for MPI_Comm_rank or
// MPI_Comm_size is erroneous
if (MPI_COMM_NULL != prime_comm) {
    MPI_Comm_rank(prime_comm, &prime_rank);
    MPI_Comm_size(prime_comm, &prime_size);
}

printf("WORLD RANK/SIZE: %d/%d \t PRIME RANK/SIZE: %d/%d\n",
    world_rank, world_size, prime_rank, prime_size);
```

# Using MPI groups

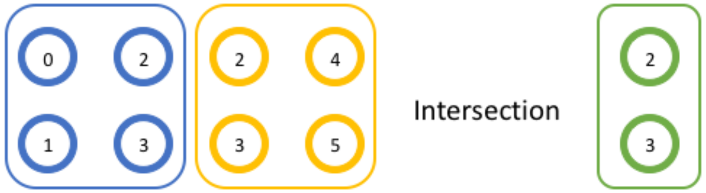
Once you have a group or two, performing operations on them is straightforward. Getting the union looks like this:

```
MPI_Group_union(
    MPI_Group group1,
    MPI_Group group2,
    MPI_Group* newgroup)
```



And you can probably guess that the intersection looks like this:

```
MPI_Group_intersection(
    MPI_Group group1,
    MPI_Group group2,
    MPI_Group* newgroup)
```



In both cases, the operation is performed on `group1` and `group2` and the result is stored in `newgroup`.

## Virtual and Physical Topologies

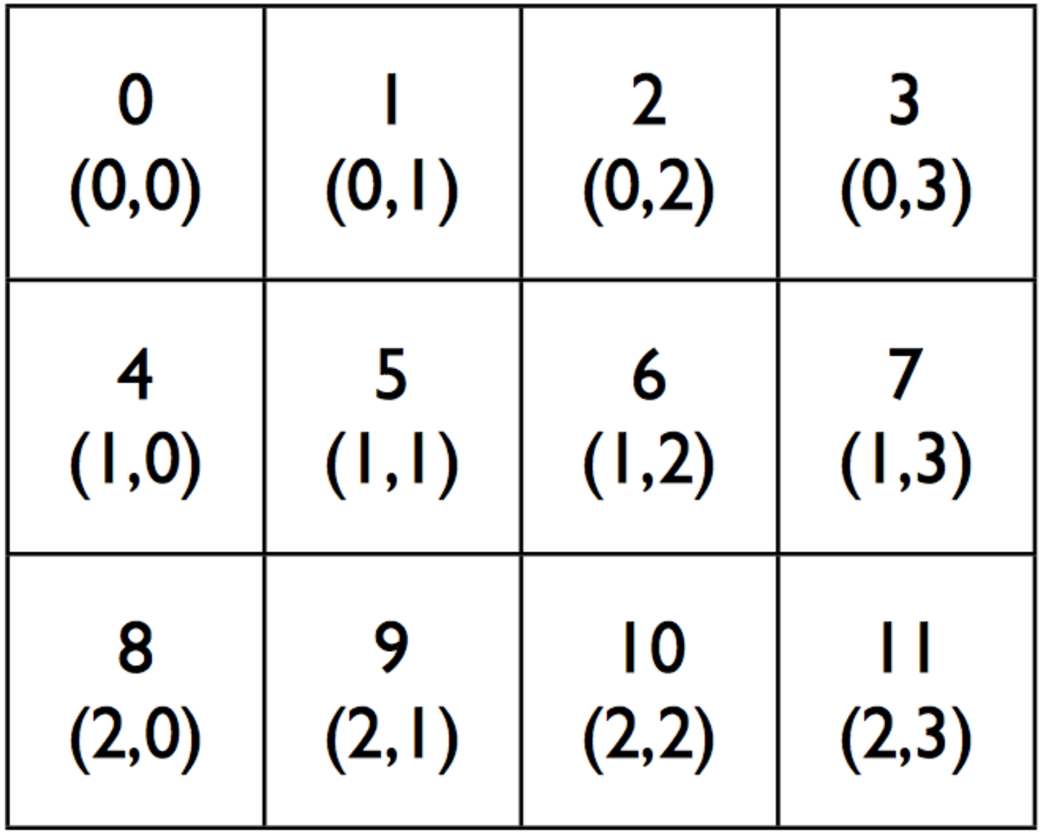
- A *virtual topology* represents the way that MPI processes communicate
  - Nearest neighbor exchange in a mesh
  - recursive doubling in an all-to-all exchange
- A *physical topology* represents that connections between the cores, chips, and nodes in the hardware

## MPI’s Topology Routines

- MPI provides routines to create new intra-communicators that order the process ranks in a way that *may* be a better match for the *physical topology*
- Two types of virtual topology supported:
  - Cartesian (regular mesh)
  - Graph (several ways to define in MPI)
- Additional routines provide access to the defined virtual topology
- (Virtual) topologies are properties of a communicator
  - Topology routines all create a *new* communicator with properties of the specified virtual topology

# MPI Cartesian Topology

- Example: 12 processes arranged on a 3 x 4 grid



| | | | |
|---|---|---|---|
| 0 (0,0) | 1 (0,1) | 2 (0,2) | 3 (0,3) |
| 4 (1,0) | 5 (1,1) | 6 (1,2) | 7 (1,3) |
| 8 (2,0) | 9 (2,1) | 10 (2,2) | 11 (2,3) |

## MPI Cartesian Topology

- Create a new virtual topology using
  - MPI_Cart_create
- Determine “good” sizes of mesh with
  - MPI_Dims_create

## MPI Cartesian Topology

int MPI_Cart_create (MPI_Comm comm_old , int ndims , int *dims , int *periods , int reorder , MPI_Comm *comm_cart)

- Creates a new communicator comm_cart from comm_old, that represents an ndims dimensional mesh with sizes dims.
- The mesh is periodic in coordinate direction if periods[i] is true.
- The ranks in the new communicator are reordered (to better match the physical topology) if reorder is true (Set to false, rank in *comm_cart* must be the same as in *comm_old*)

## MPI Cartesian Topology

- Creates logical 2-d Mesh of size 3x4

```
int dims[2] = {3,4};
int periods[2] = {0,0};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0,
&topocomm);
```

- But we're starting MPI processes with a one-dimensional argument (-p X)
  - User has to determine size of each dimension
  - Often as "square" as possible, MPI can help!

# MPI Cartesian Topology

- MPI_Dims_create(int nnodes, int ndims, int *dims)
  - Fill in the dims array such that the product of dims[i] for i=0 to ndims-1 equals nnodes.
  - Any value of dims[i] that is 0 on input will be replaced; Non-zero entries in dims will not be changed
  - MPI_Dims_create(12, 2, dims) == (3,4) or (4,3)?

## CARTESIAN QUERY FUNCTIONS

- MPI_Cartdim_get()
  - Gets dimensions of a Cartesian communicator
- MPI_Cart_get()
  - Gets size of dimensions
- MPI_Cart_rank()
  - Translate coordinates to rank
- MPI_Cart_coords()
  - Translate rank to coordinates

## CARTESIAN COMMUNICATION HELPERS

MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
int *rank_source, int *rank_dest)

- Returns the ranks of the processes that are a shift of disp steps in coordinate direction
- Useful for nearest neighbor communication in the coordinate directions
- May return MPI_PROC_NULL

## MPI Graph Topology

- MPI provides routines to specify a general graph virtual topology
  - Graph vertices represent MPI processes (usually one per process)
  - Graph edges indicate important connections (e.g., nontrivial communication between the connected processes)
  - Edge weights provide more information (e.g., amount of communication)

## MPI Graph Topology

int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int ∗index, int ∗edges, int reorder , MPI_Comm ∗comm graph)

- nnodes is the total number of nodes
- index i stores the total number of neighbors for the first i nodes (sum)
  - Acts as offset into edges array
- edges stores the edge list for all processes
  - Edge list for process j starts at index[j] in edges
  - Process j has index[j+1]-index[j] edges

# DISTRIBUTED GRAPH CONSTRUCTOR

- MPI_Graph_create is discouraged
  - Not scalable
  - Not deprecated yet but hopefully soon
- New distributed interface:
  - Scalable, allows distributed graph specification
    - Either local neighbors or any edge in the graph
  - Specify edge weights
    - Meaning undefined but optimization opportunity for vendors!
- Info arguments
  - Communicate assertions of semantics to the MPI library
  - E.g., semantics of edge weights

## DISTRIBUTED GRAPH CONSTRUCTOR

MPI_Dist_graph_create_adjacent(MPI_Comm oldcomm, int indegree, int sources[], int sourceweights[],int outdegree, int dests[], int destweights[], MPI_Info info, int qreorder, MPI_Comm *newcomm)

- Describe *only* the graph vertex corresponding to the calling process
  - Hence "Dist_graph" – distributed description of graph
- Graph is directed – separate in and out edges
- info allows additional, implementation-specific information
- qreorder if true lets MPI implementation reorder ranks for a better mapping to physical topology
- **MPI_UNWEIGHTED** may be used for weights *arrays*

## Other Graph Routines

- MPI_Dist_graph_create
  - More general, allows multiple graph vertices per process
- Information on graph
  - MPI_Dist_graph_neighbors_count
    - Query the number of neighbors of calling process
  - MPI_Dist_graph_neighbors
    - Query the neighbor list of calling process
- Collective Communication along arbitrary neighborhoods
  - MPI_NEIGHBOR_ALLGATHER
  - MPI_NEIGHBOR_ALLTOALL

## Topology: good and bad

- A common virtual topology is *nearest neighbor in a mesh*
  - Matrix computations
  - PDE Simulations on regular computational grids
- Many Large Scale Systems use a mesh as the physical topology
  - IBM Blue Gene series; Cray through XE6/XK7
- Performance can depend on how well the virtual topology is mapped onto the physical topology

- Bisection bandwidth does *not* scale with network size of a mesh/ torus network
- Non-nearest neighbor communication suffers from contention
- Graph topology: Complex to implement. No good implementations in general use; research work limited

## ONE-SIDED COMMUNICATION

## ONE-SIDED COMMUNICATION

- Two components of message-passing: sending and receiving
- One-sided communication
  - Only single process calls data movement functions (put or get) – remote memory access (RMA)
  - Communication patterns specified by only a single process – Always non-blocking

## Why one-sided communication?

- Certain algorithms featuring unstrucutred communication easier to implement
- Potentially reduced overhead and improved scalability
- Hardware support for remote memory access has been restored in most current-generation architectures

## Origin and target

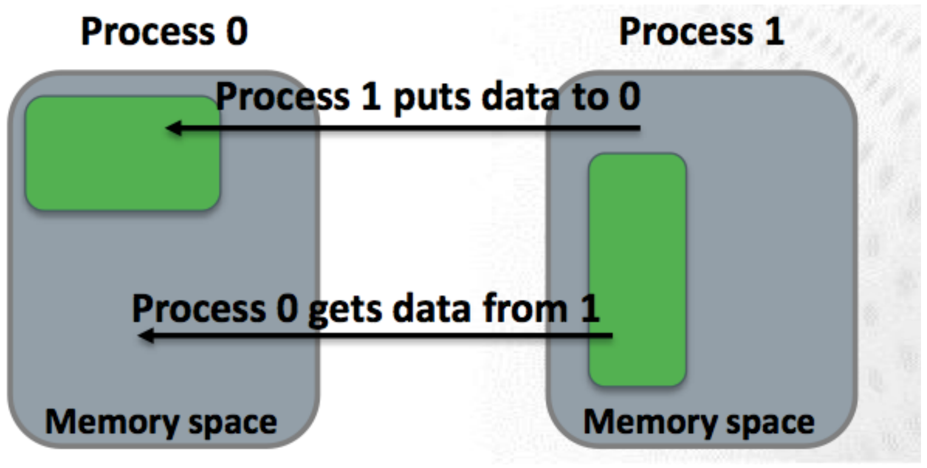
- Origin process: a process which calls data movement function
- Target process: a process whose memory is accessed

## Origin and target

- Origin process: a process which calls data movement function
- Target process: a process whose memory is accessed

## Remote memory access window

- *Window* is a region in process's memory which is made available for remote operations
- Windows are created by collective calls
- Windows may be different in different processes



## Data movement operations in MPI

- PUT data to the memory in target process
  - From local buffer in origin to the window in target
- GET data from the memory of target proces
  - From the window in target to the local buffer in origin
- ACCUMULATE data in target process
  - Use local buffer in origin and update the data (e.g. add the data from origin) in the window in target
  - One-sided reduction

## Synchronization

- Communication takes place within epochs
  - Synchronization calls start and end an epoch
  - There can be multiple data movement calls within an epoch
  - An epoch is specific to a particular window
- Active synchronization:
  - Both origin and target perform synchronization calls
- Passive synchronization:
  - No MPI calls at target process

## One-sided communication in a nutshell

- Define a memory window Start an epoch
  - Target: exposure epoch
  - Origin: access epoch
- GET, PUT, and/or ACCUMULATE data
- Complete the communications by ending the epoch

## Creating a window

MPI_Win_create(base, size, disp_unit, info, comm, win)

– base (pointer to) local memory to expose for RMA
– size size of a window in bytes
– disp_unit local unit size for displacements in bytes
– info hints for implementation
– comm communicator
– win handle to window

## Starting and ending an epoch

MPI_Win_fence(assert, win)

- assert optimize for specific usage. Valid values are ”0”, MPI_MODE_NOSTORE, MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE, MPI_MODE_NOSUCCEED
- win window handle

- Used both for starting and ending an epoch
  - Should both precede and follow data movement calls
- Collective, barrier-like operation

## Data movement: Put

MPI_Put(origin, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

- origin (pointer to) local data to be send  to target
- origin_count number of elements to put
- origin_datatype MPI datatype for local data
- target_rank rank of the target task
- target_disp starting point in target window
- target_count number of elements in target
- target_datatype MPI datatype for remote data
- win RMA window

## Simple example: Put

```
...
int data;
MPI_Win window;
data = rank;

// Create window
MPI_Win_create(&data, sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD,
&window);
 ...
MPI_Win_fence(0, window);

if (rank == 0)
        MPI_Put(&data, 1, MPI_INT, 1, 0, 1, MPI_INT, window);

MPI_Win_fence(0, window);
...
MPI_Win_free(&window);
```

## Performance considerations/summary

- Performance of the one-sided approach is highly implementation-dependent
- Maximize the amount of operations within an epoch
- Provide the assert parameters for MPI_Win_fence
- One-sided communication allows communication patterns to be specified from a single process
- Can reduce synchronization overheads and provide better performance especially on recent hardware
- Basic concepts:
  – Creation of the memory window
  – Communication epoch
  – Data movement operations (MPI_Put, MPI_Get etc)