ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

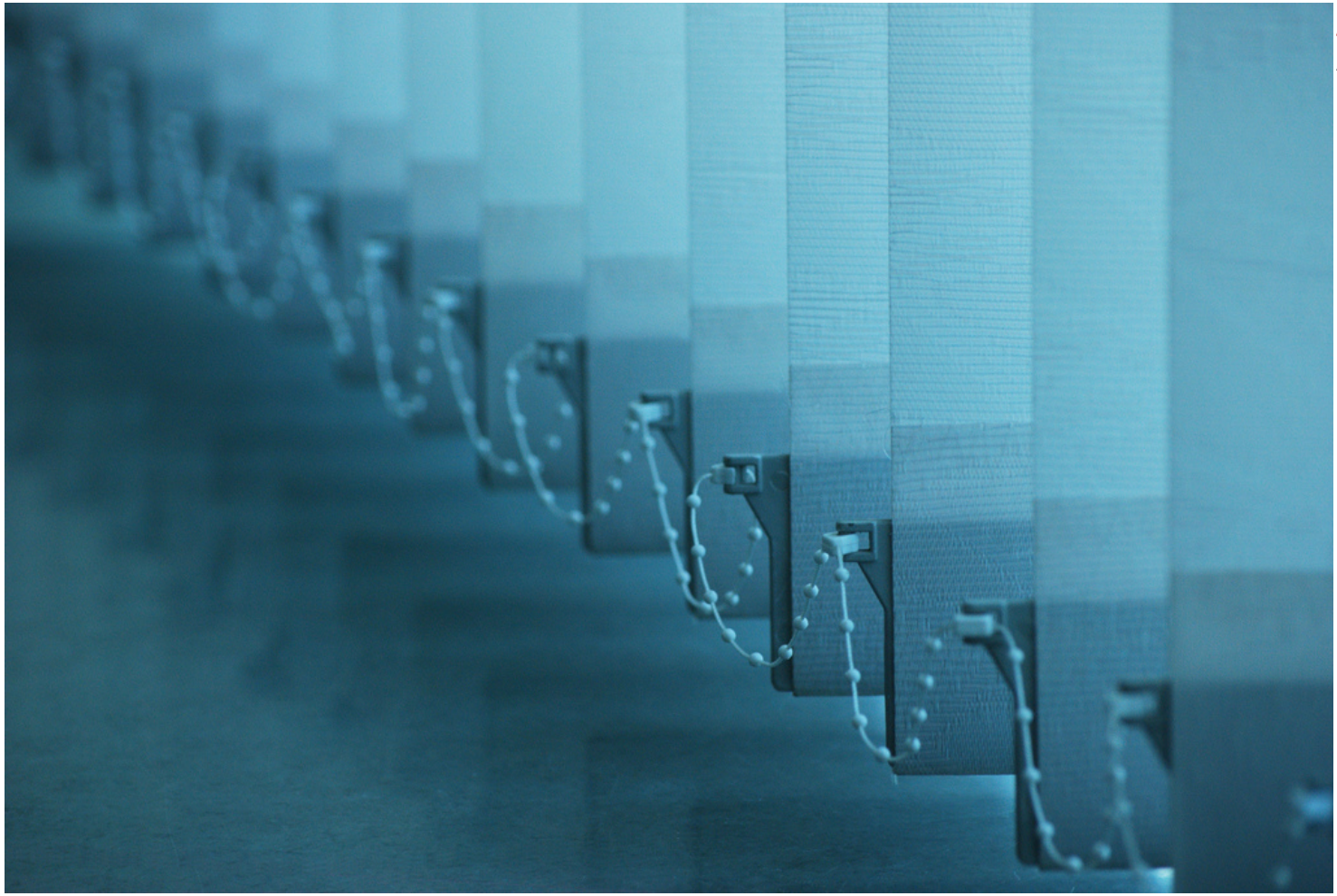
Introduction to Programming with MPI

Course Outline

ASTRONOMY DATA AND COMPUTING SERVICES

1. Introduction
2. Exercise 1: “Hello World”
3. Point-to-point communication
4. Exercise 2: "Ping-pong"
5. Non-blocking communication
6. Exercise 3: "Pass a message around a ring"
7. Communicators
8. Exercise 4: "Using a Cartesian communicator"
9. Collective Communication
10. Exercise 5: "Using collective communication"
11. Break
12. Game of Life mini-code introduction
13. Exercise 6: "Game of life"
14. Overview of other topics and Summary
15. Close

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES



"Parallel Worlds" by aloshbennett from

http://www.flickr.com/photos/aloshbennett/3209564747/sizes/l/in/photostream/

I. Introduction

Supercomputing – bit of history

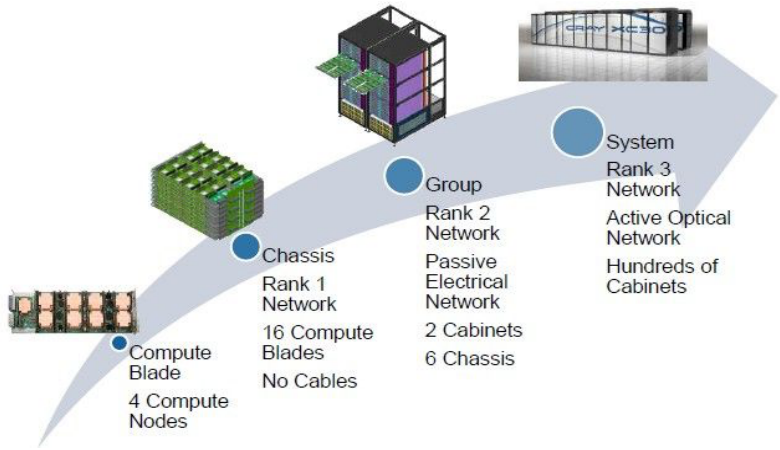
ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

Cray 1 (1976)

IBM Blue Gene (2005)

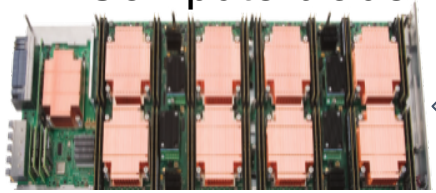
Cray XT5 (2009)

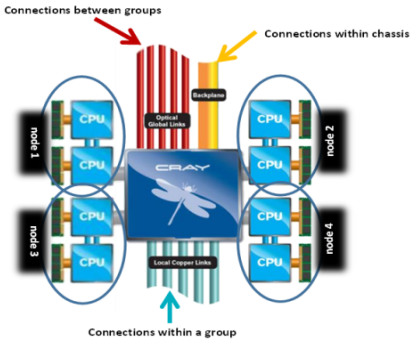
Architecture of Cray XC 40 -- Magnus



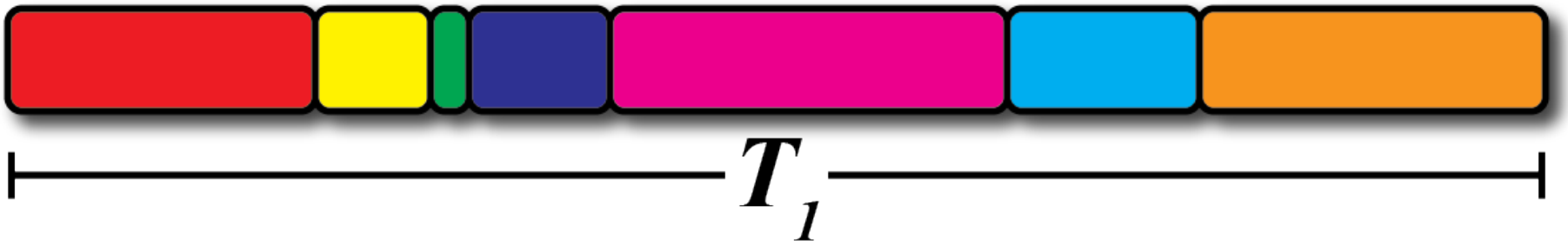
Compute node

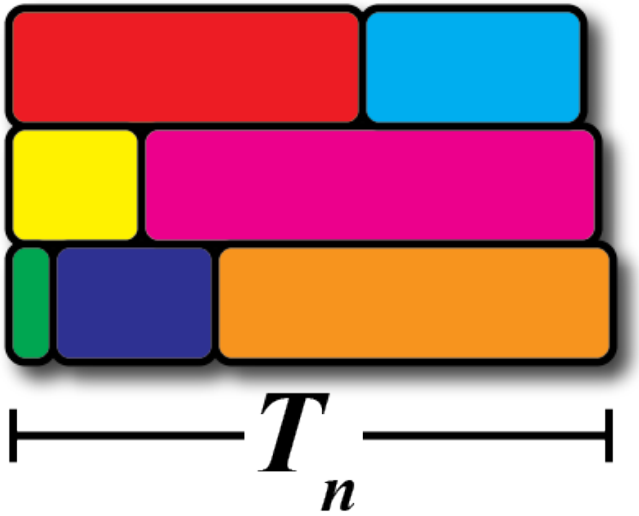
Compute blade





Serial vs. Parallel



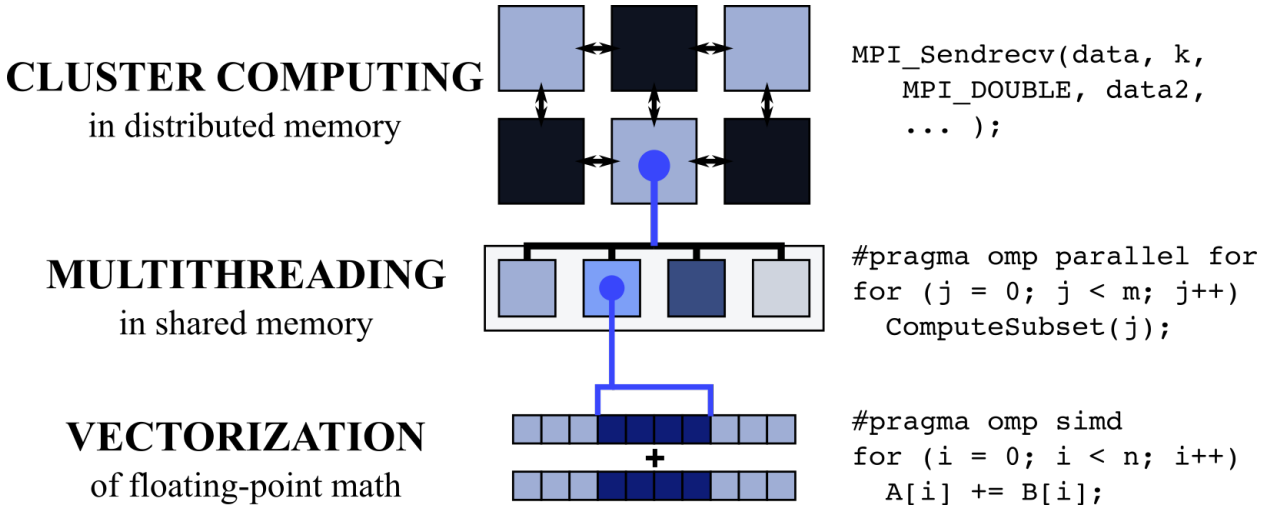


- 1 processer can do the work in time T_1
- n processors can do the same work in time T_1 / n (in theory)

Parallel programming layers

How to get the maximum out of the modern microprocessor architecture?

- MPI across the nodes
- Multithreading on the node -- OpenMP
- Vectorization employed by each thread



Parallelization Strategies

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- How do we divide the work amongst processors?
  - Define “work”
- Data Parallelism
- Task Parallelism

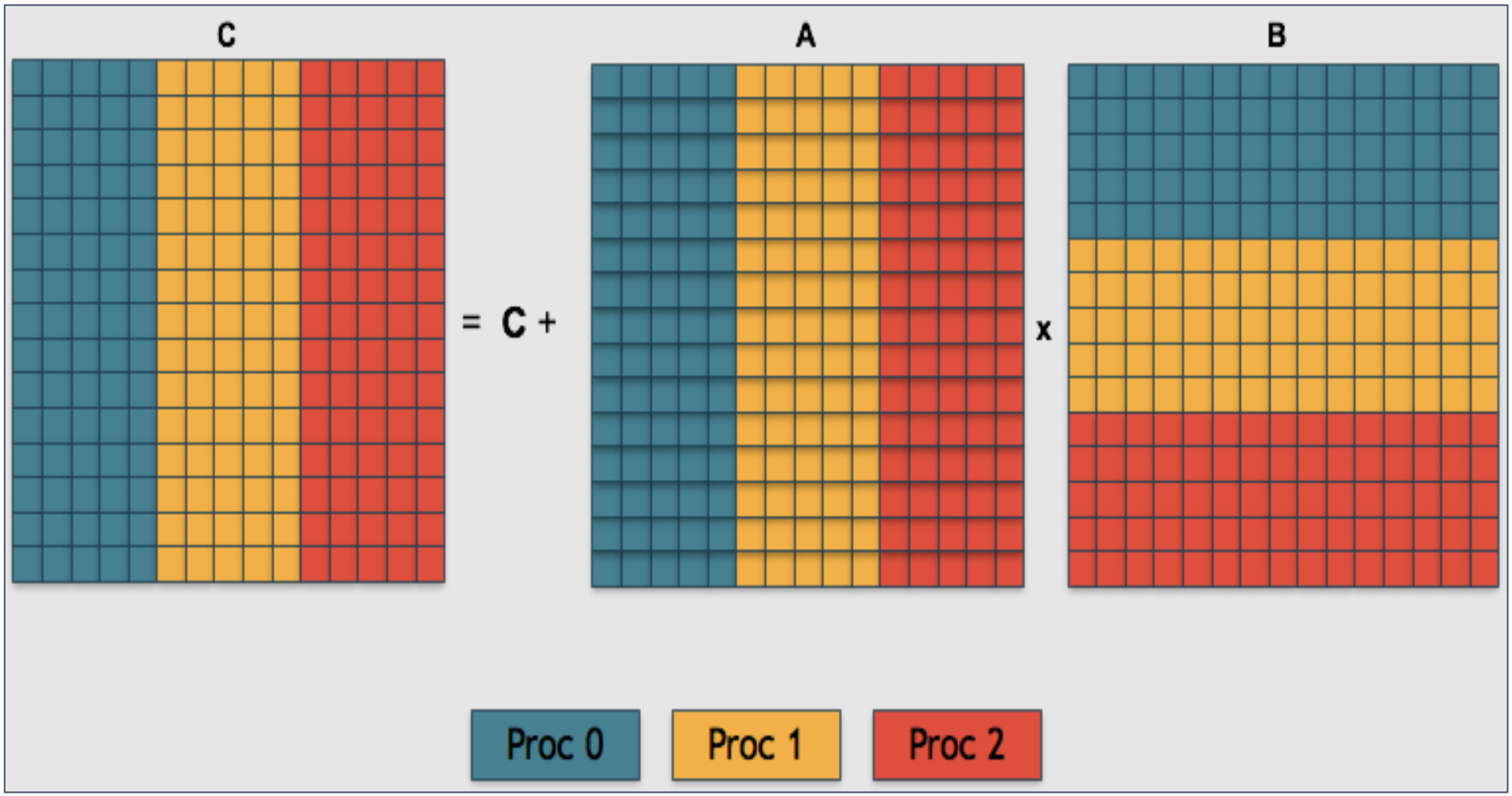
Data Parallelism

- Same instruction (program) runs on each processor
- Unique portion of data assigned to each processor
- Example: for loop

```
for k=1:N
 for j=1:N
  for i=1:N
   C[i,j] += A[i,k]*B[k,j]
```

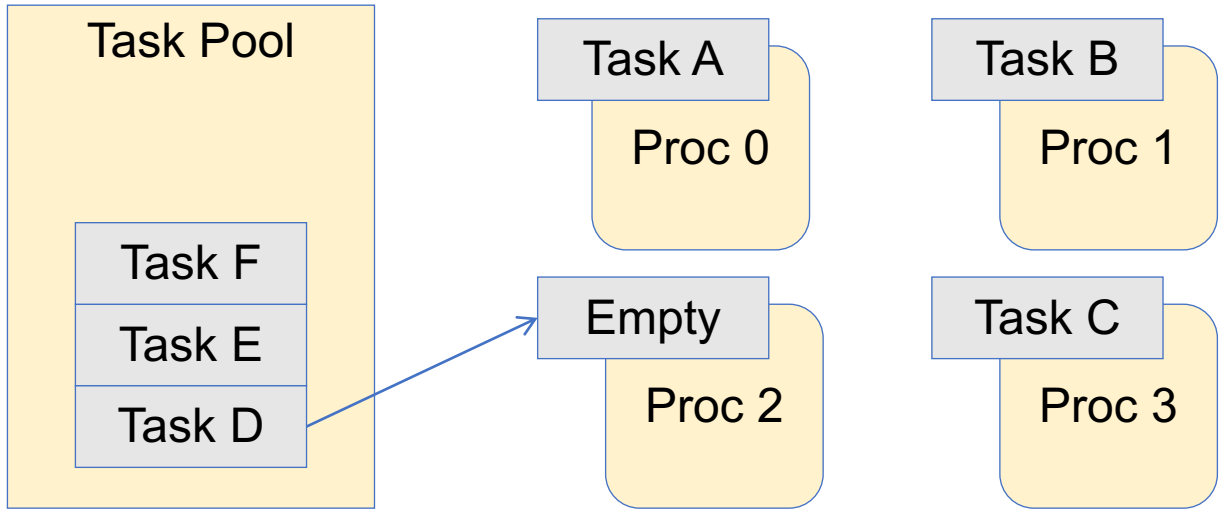
Processor	Iterations	Data
Proc 0	K = 1:5	A[i, 1:5] B[1:5, j]
Proc 1	K = 6:10	A[i, 6:10] B[6:10, j]
Proc 2	K = 11:15	A[i, 11:15] B[11:15, j]

Data Parallelism



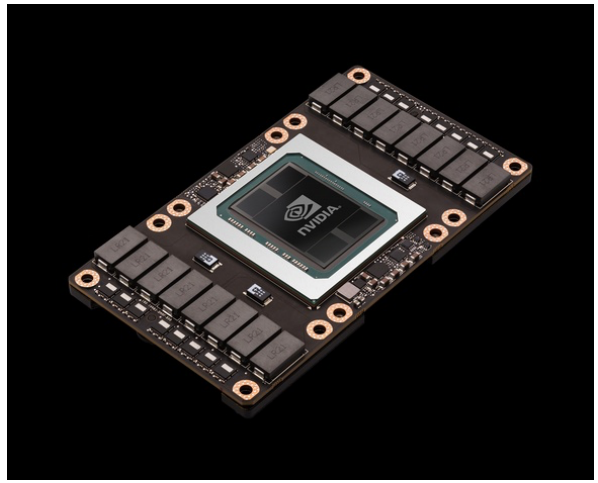
Task Parallelism

- Simultaneous execution of different functions/tasks
  - Can be on same or different data sets
- Example: Instruction pool
  - Processes pull task from a pool
  - Tasks may be non-uniform in size – Load Balancing issues
  - Implication -> Scaling may be limited

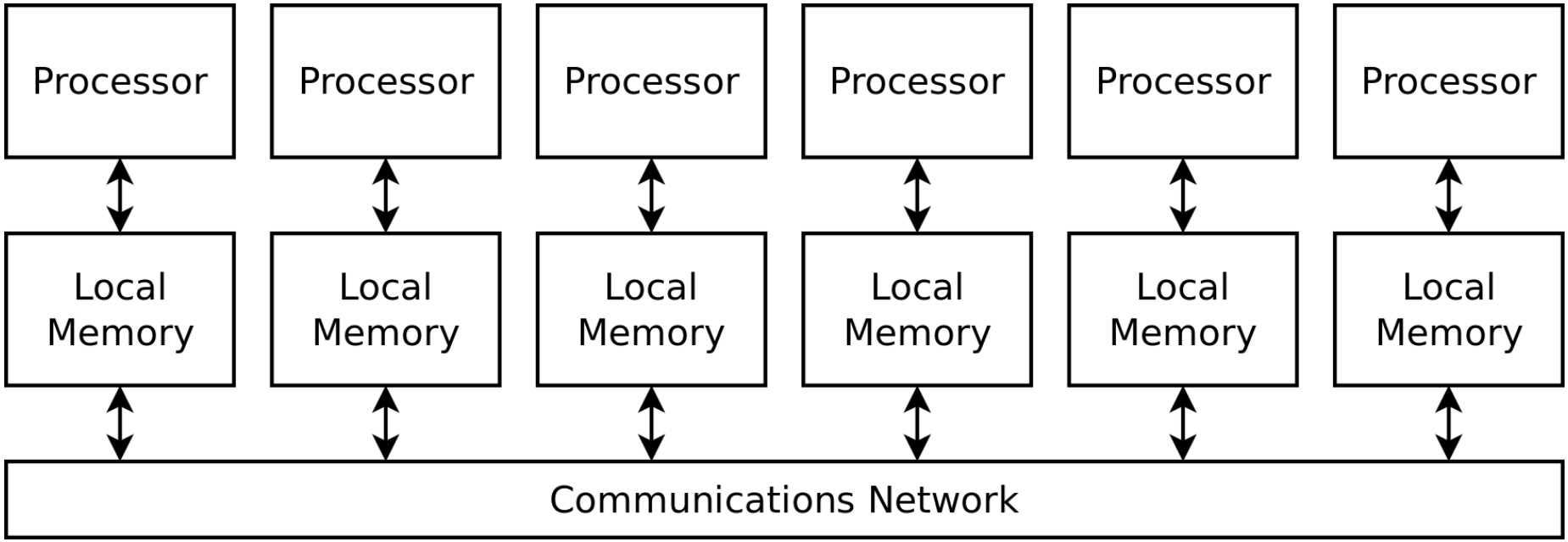


Hardware/Software Relation

- Different architectures dictate different programming models
  - Benefits/downsides to each
- Two main categories
  - Shared Memory
    - Covered in OpenMP course
  - **Distributed Memory**
    - Focus of today's course



Distributed Memory Model



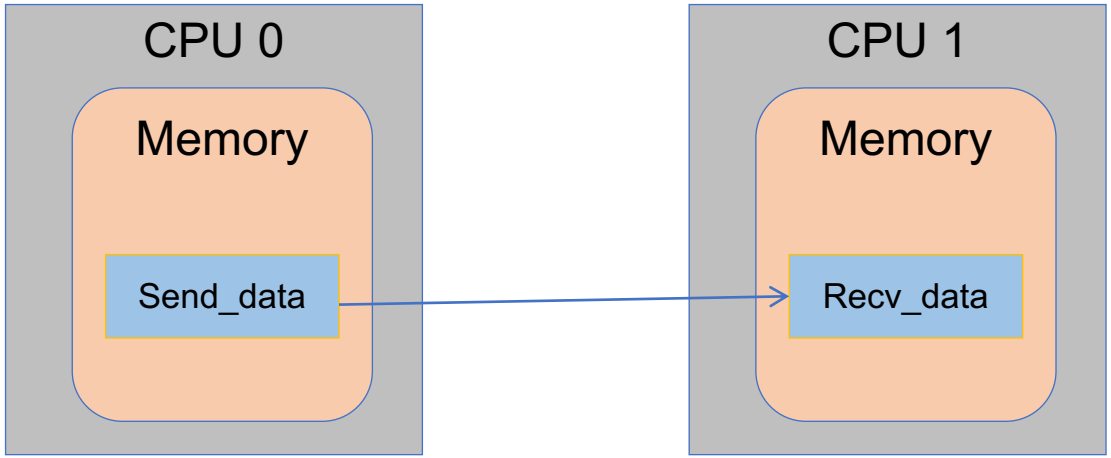
- Each processor has its own local memory, with its own address space
- Data is shared via a communications network
- May require additional memory for multiple copies of data

Programming for Distributed Memory

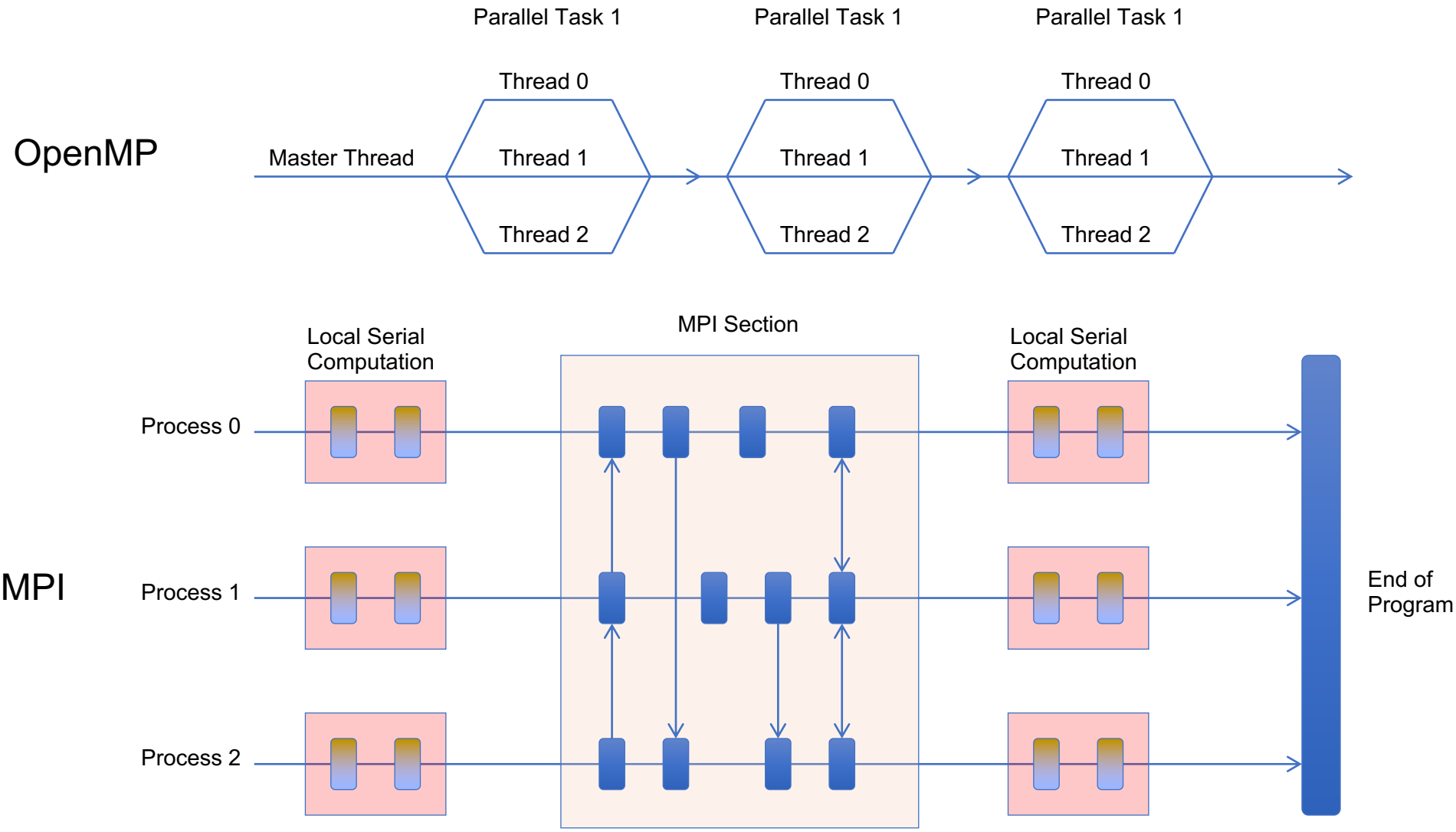
- Each CPU
  - Has its own memory (not accessible to other processors)
  - Executes the same program (independent of other CPUs)
  - Works on a unique portion of the whole problem
  - CPUs require synchronization points to exchange data
- Challenge – How do you write a code to do this?

MPI

- Message Passing Interface
  - Standard defining how CPUs send and receive data
  - Vendor specific implementation adhering to the standard
  - Allows CPUs to “talk” to each other
    - i.e. read and write memory



Distributed vs. Shared: Program Execution



Distributed vs. Shared: Pros and Cons

API	Pros	Cons
Shared (OpenMP)	• Debugging • Compiler handles much of the parallelization (directives) • Less development in adding OpenMP to existing serial application (incrementally add in parallelism)	• Require compiler support • Limited to shared-memory environment (e.g. single node of Magnus) • Scalability issues • Parallelization tends to focus on loops, leaving large portions of the code still serial • I/O is recommended to be handled by a single thread
Distributed (MPI)	• Can run on either shared or distributed memory architectures • Distributed memory systems cheaper and more prevalent • Scalable	• Debugging • Development process (incremental parallelism is difficult) • Performance can be limited by communication network

Source: http://www.cnhindustrial.com/en-us/our_brands/new_holland_agriculture/Pages/default.aspx

Message Passing Interface

Introduction to MPI

ASTRONOMY DATA AND COMPUTING SERVICES

- Message Passing Interface
- Standard defining API
  - Names
  - Syntax
  - Results
- Not a library (although a library is used)
  - Standard defines what should be in the library
  - Vendor specific version and Open versions exist of the libraries made on these standards

MPI Standard

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- Multiple MPI standard versions
- MPI-1 (1994)
  - Core functionality, Fortran and C bindings
- MPI-2 (1998)
  - Superset of MPI-1, parallel I/O features, one-sided communication, C++ bindings (deprecated in later versions)
- MPI-3 (2012)
  - Expanded one-sided communication, non-blocking collectives

MPI Routines

- Focus for today:
  - Environment
  - Point-to-point communication
  - Simple use of communicators
  - Collective operations
  - … can be combined to perform useful work
- Many, many other routines
  - Not covered

6 Primary MPI Routines - C

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- `int MPI_Init(int * argc, char *** argv);`
- `int MPI_Finalize(void);`
- `int MPI_Comm_size(MPI_Comm comm, int * size);`
- `int MPI_Comm_rank(MPI_Comm comm, int * rank);`
- `int MPI_Send(void * buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);`
- `int MPI_Recv(void * buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status * status);`

Initiation and Termination

ASTRONOMY DATA AND COMPUTING SERVICES

- **`MPI_Init`**`(int * argc, char *** argv)`
  - Initialises the MPI environment
  - 1st MPI call (usually)
  - May be called only once
  - Place in body of code after variable declarations and before any MPI commands
- **`MPI_Finalize`**`(void)`
  - Terminates MPI environment
  - Last MPI command

Environment Inquiry

- **`MPI_Comm_size`**`(MPI_Comm comm, int *size)`
  - Find out number of MPI tasks (ranks)
  - Allows flexibility in number of processes used in program
- **`MPI_Comm_rank`**`(MPI_Comm comm, int *rank)`
  - Find out rank of current process
  - $0 \leq$ **`rank`** $\leq$ **`size`**-1

The Communicator

ASTRONOMY DATA AND COMPUTING SERVICES

- **`MPI_Comm_size`**`(MPI_Comm comm, int *size)`
  - Requires an argument which is the communicator
- The context within which communication takes place
  - A default communicator is provided
  - **`MPI_COMM_WORLD`**

Basic Program Structure - C

ASTRONOMY DATA AND COMPUTING SERVICES

```
#include <stdio.h>
#include “mpi.h”
int main(int argc, char *argv[]) {
      int ntasks, rank;
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      /* Do some work
       * More MPI calls (MPI_SSend, MPI_Recv, etc.)
       * Do more work */
      MPI_Finalize();
}
```

Accessing exercises

All material in Github. Clone it if you don't have it from previous tutorials.

```
git clone https://github.com/ADACS-Australia/HPC-Workshop.git
```

OpenMP Exercises, Solutions and Demos in:

```
cd HPC-Workshop/Introduction_to_MPI
```

Exercises and Solution directories contain:

- C and Fortran source codes
- Makefile and job submission scripts.

Exercise 1: Hello World

ASTRONOMY DATA AND COMPUTING SERVICES

```
> cd HPC_Workshop/Exercises/c/ex1-hello
```

- Write an MPI program that
  - Sets up the MPI environment
  - Prints (for each MPI rank) the rank and the total number of MPI tasks
  - Prints the rank of each MPI task to standard out

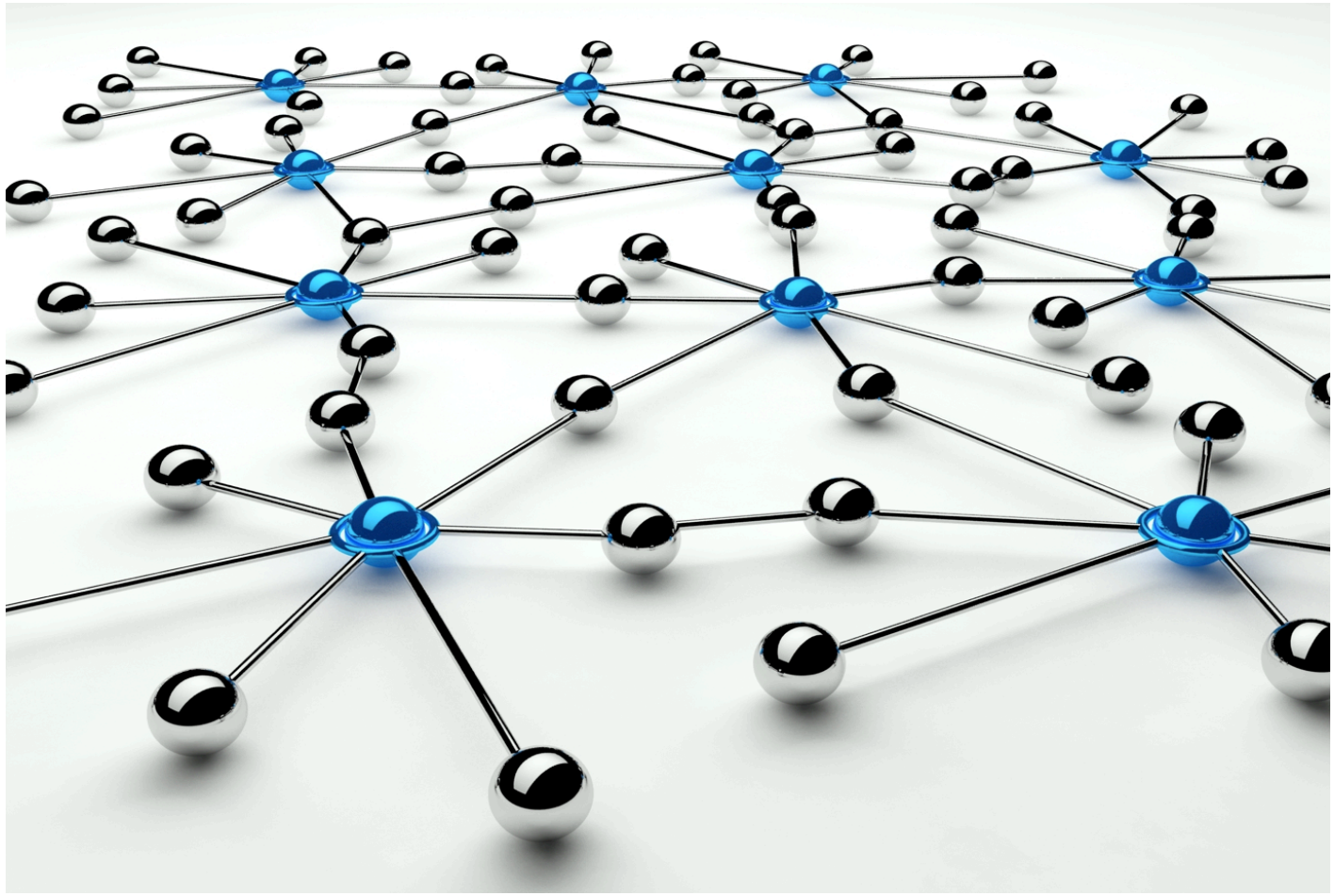
  Compile and link it:

  ```
  > make
  ```

  Submit to the scheduler to run:

  ```
  > sbatch hello-mpi.slurm
  ```

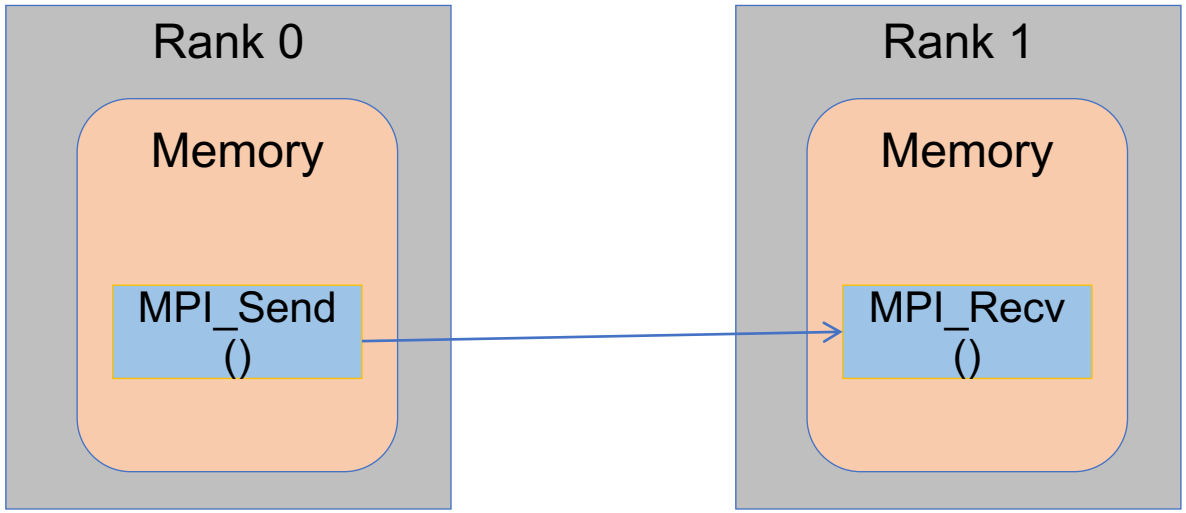
ADACS
ASTRONOMY DATA AND COMPUTING SERVICES



Point-to-point communication

Point-to-Point Communication

- Fundamental operation in MPI
  - One MPI task sends data and another receives
- A two-sided operation
  - Both ranks must participate (explicit send and receive calls)



MPI Messages

ASTRONOMY DATA AND COMPUTING SERVICES

- Message consists of 2 parts
  - **Envelope**
  - **Message Body (actual data)**
- Envelope components
  - **Source** – rank that is sending
  - **Destination** – rank that is receiving
  - **Communicator** – source and destination must be in same communicator
  - **Tag** – identifies a message

MPI Messages

ASTRONOMY DATA AND COMPUTING SERVICES

- 3 Components to a message
- **Buffer**
  - Starting location in memory of data to be sent
  - C – void *
  - Fortran – reference
- **Datatype**
  - Elementary type: integer, real, float, etc.
- **Count**
  - Number of items of given datatype to be sent

MPI Datatypes

MPI Datatype	C Type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	(none)
MPI_PACKED	(none)

Message Passing: Send

ASTRONOMY DATA AND COMPUTING SERVICES

- **`MPI_Send`**`(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Post a send of length **`count`** items of datatype **`datatype`** contained in **`buf`** with tag **`tag`** to process number **`dest`** in communicator **`comm`**
- Example

```
MPI_Send(&x, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD)
```

Message Passing: Receive

ASTRONOMY DATA AND COMPUTING SERVICES

- **`MPI_Recv`**`(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - Post a receive of length **`count`** items of datatype **`datatype`** with tag **`tag`** in buffer **`buf`** from process number **`source`** in communicator **`comm`** and record status **`status`**
- Example

```
MPI_Recv(&x, 1, MPI_DOUBLE, source, tag,
MPI_COMM_WORLD, &status)
```

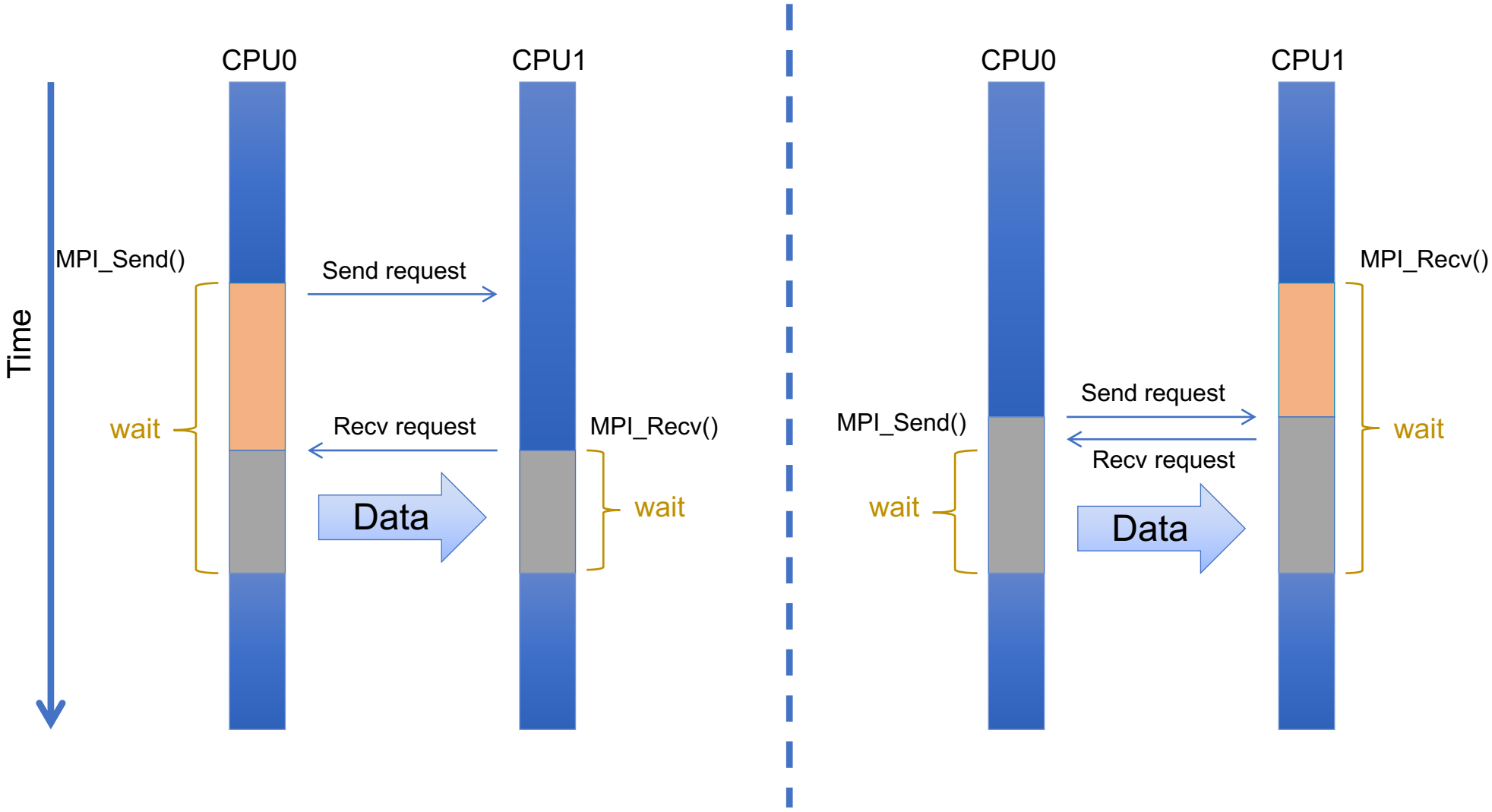
MPI_Status

- Holds the source, tag, and size of the message that was received

```
MPI_Status status;
```

- Can be useful in some contexts but often ignored

Blocking Call (Synchronous send and receive)



MPI Send Modes

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- `MPI_Send()`
  - Basic blocking send call
  - Will not return until the send buffer can be used
- `MPI_Ssend()`
  - Synchronous blocking send
  - Blocks until application buffer in sending task is free **AND** destination task has started to receive
  - Likely to give good performance
    - MPI implementation avoids having to buffer data

MPI Send Modes

- `MPI_Bsend()`
  - User should supply buffer space for message
  - Only use if necessary
- Recommendation:
  - Use `MPI_Ssend()` if in doubt
- Only one type of `MPI_Recv()`

Exercise 2: Ping Pong

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- Write an MPI program to do the following
  - For 2 MPI tasks, have each send a message back and forth
  - Set a limit of the number of “ping-pongs” (e.g., 10)
  - Hints:
    - `MPI_Ssend()`
    - `MPI_Recv()`
    - Message passing in a loop
    - Need to create variables for processor IDs
    - Can use `MPI_Wtime()` if you want to try timing
    - Solution in Develop git repository

Message Passing

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- `MPI_Send()` and `MPI_Recv()` are *blocking*
- **`MPI_Recv`** returns only after receive buffer contains requested message
- **`MPI_Send`** will block until message data copied out of the send buffer
- Must avoid deadlock

Deadlocking Example (Always)

```
if (myRank%2==1) {
  dest= myRank - 1;
  src = myRank - 1;
}
else {
  dest= myRank + 1;
  src = myRank + 1;
}
MPI_Recv(&recv, 1, MPI_INT, src, tag, MPI_COMM_WORLD, &status);
MPI_Send(&myRank, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
printf("Sent %d to proc %d, received %d from
       proc %d\n", myRank, dest, recv, src);
```

Deadlocking Example (Safe)

```
if (myRank%2==1) {

  dest= myRank - 1;

  src = myRank - 1;

}

else {

  dest= myRank + 1;

  src = myRank + 1;

}

if (myRank%2 == 0) {

MPI_Send(&myRank,1,MPI_INT,dest,myR
ank,MPI_COMM_WORLD);

MPI_Recv(&recv,1,MPI_INT,src,src,MP
I_COMM_WORLD, &status);

}
```

```
else {

MPI_Recv(&recv,1,MPI_INT,src,src,MP
I_COMM_WORLD, &status);

MPI_Send(&myRank,1,MPI_INT,dest,myR
ank,MPI_COMM_WORLD);

}

printf("Sent %d to proc %d,
received %d from proc %d\n",
myRank,dest,recv,src);
```

Explanation:
Always Deadlock Example

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- Logically incorrect
- Deadlock caused by blocking **`MPI_Recvs`**
- All processes wait for corresponding **`MPI_Sends`** to begin, which never happens

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

Non-Blocking communication

Blocking vs. Non-blocking

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- Blocking communication
  - Function call will not return until send/receive is complete
- Blocking send
  - Guarantee that variables can safely be overwritten
- Blocking receive
  - Data can safely be used

Blocking vs. Non-blocking

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- Non-blocking communication
  - Function call returns immediately
  - State of data (send or receive) is unknown
- Programmer must check status of request
- Advantage
  - Avoids additional logic
  - Overlap communication and computation
  - Performance improvement
- Disadvantage
  - Complexity; all requests must be handled

Non-blocking Send/Receive

- **MPI_Isend**(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
- **MPI_Issend**(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
- **MPI_Irecv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

- Similar to blocking calls, but with **MPI_Request**
  - Used to test if receive has completed
  - How to test?

MPI_Wait

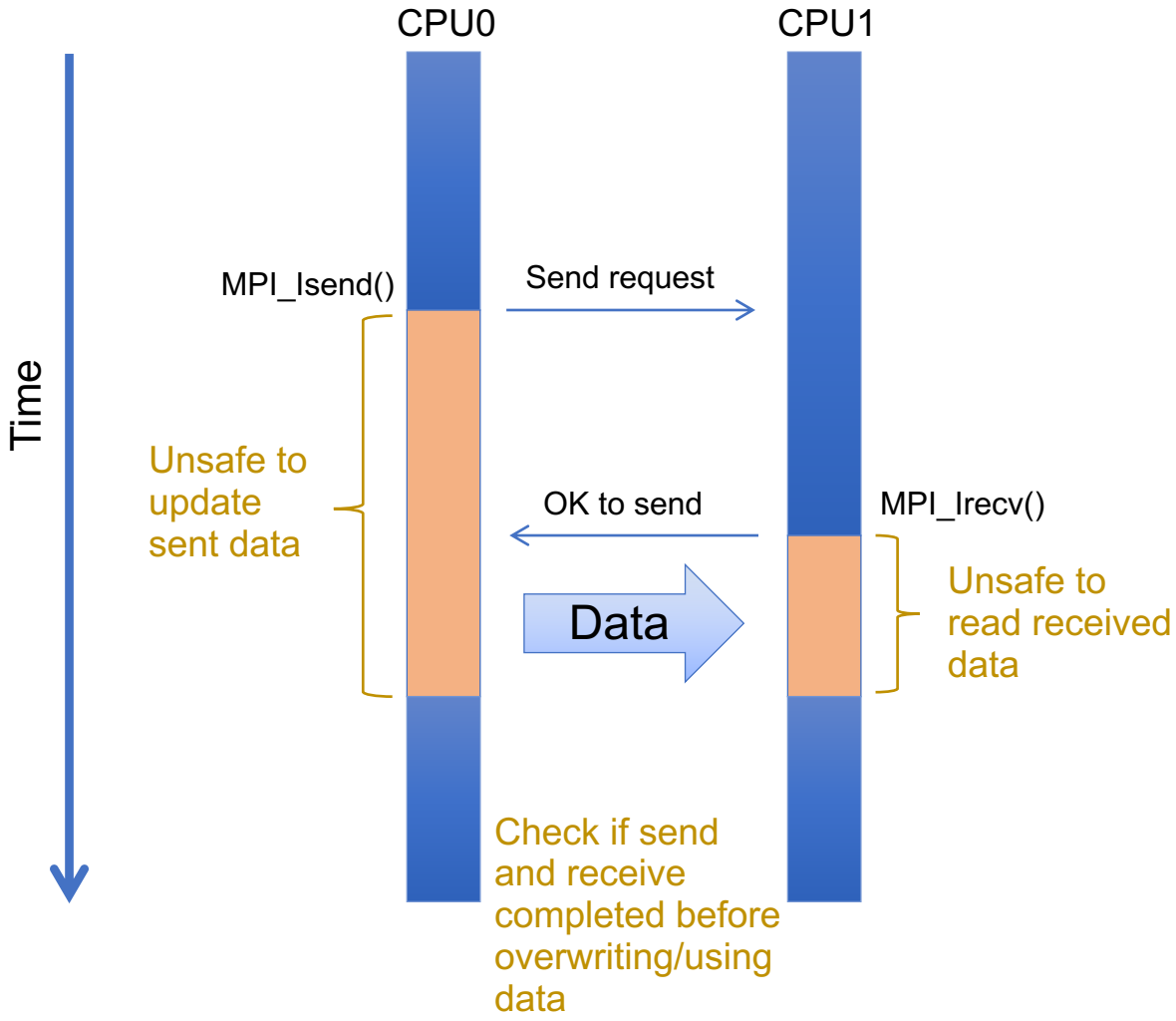
- Function used to test if a request has completed
- **`MPI_Wait`**`(MPI_Request *request, MPI_Status *status)`
- Calling **`MPI_Wait()`** immediately after **`MPI_Isend()`** is equivalent to **`MPI_Send()`**
- How to use **`MPI_Wait()`**

```
MPI_Isend(…);
// do other computation, write to disk, read
// from disk, etc.
MPI_Wait(…);
```

- Array version:

`MPI_Waitall``(int count, MPI_Request requests[], MPI_Status status[]`

Non-blocking Call



Exercise 3: Ring

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

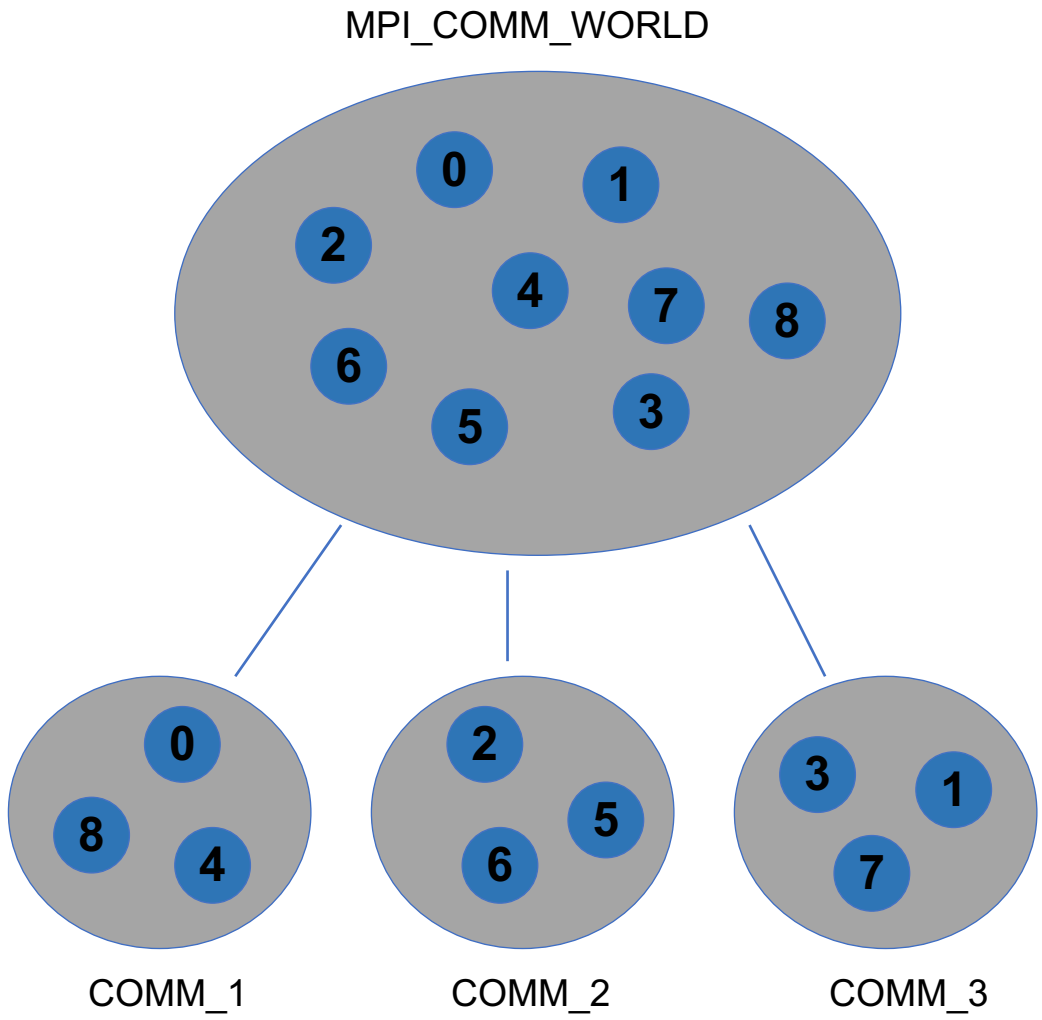
- Write an MPI program to do the following
  - Accumulate the sum of MPI ranks by sending a message around a ring
  - Hints:
    - Need to figure out neighboring ranks (think about end points)
    - Non-blocking calls (`MPI_Irecv`, `MPI_Issend or MPI_Isend`)
    - Need to check if sends (and receives) have completed
    - Loop until left neighbor sends value same as my rank
    - Can use `MPI_Wtime()` if you want to try timing – optional
    - Everyone rank must have the sum of all the ranks
    - Solution available in Solutions directory for reference

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

MPI Communicators

Communicators

- MPI object that defines a group of processors
  - Only MPI tasks in a communicator can talk to each other
- Default communicator: **`MPI_COMM_WORLD`**
  - All processors



Communicators

- Can be created/destroyed during execution
- Processes can belong to multiple communicators
- Allows you to organize tasks domains
- Used in collective operations (more on that later)
- Used in virtual topologies

Virtual Topologies

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- Method for grouping MPI processes into a “shape”
  - Built upon MPI communicators
- May be mapped efficiently to the physical topology (connection between cores, chips/sockets and nodes)
- MPI-provided topologies
  - Graphs
  - Grid (Cartesian)

Cartesian Example

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)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Blue - Rank of process

Black – 2D grid location

Image credit: http://computing.llnl.gov/tutorials/mpi

MPI Cartesian Communicator Routines

ASTRONOMY DATA AND COMPUTING SERVICES

```
MPI_Cart_create(MPI_Comm old_comm, int ndims, int * dims, int
* periods, int reorder, MPI_Comm * new_comm)
```

IN::

old_comm – Input MPI communicator

ndims – Number of dimensions in grid

dims – Array of size ndims which holds decomposition

periods – Logical array specifying if the grid is periodic or not (for each dimension)

reorder – Specifies whether or not ranks may be reordered to better suit the physical topology

OUT::

new_comm – New communicator

MPI Cartesian Communicator Routines

```
MPI_Dims_create(int nnodes, int ndims, int * dims)
```

IN

nnodes – Number nodes in a grid

ndims – Number of dimensions in a grid

OUT

dims – Array specifying the number of nodes in each direction

Convenience function to help the user select a balanced number of processors in each direction

MPI Cartesian Communicator Routines

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

IN

comm –MPI communicator

direction – Coordinate dimension of shift

disp – Displacement (length of the hops in direction)

OUT

rank_source – Rank of source process

rank_dest – Rank of destination process

Returns the shifted source and destination ranks for a given shift direction and amount

MPI_Cart_shift

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)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Convenience function used for sending data in a direction (e.g., halo swap)

Example:

```
MPI_Cart_shift(comm, 0, 1, &source, &dest)
```

comm – global communicator

0 – indicates column direction

1 – specifies we want processes 1 step away

For process 9, `MPI_Cart_shift` will return

- process 5 as the destination
- process 13 as the source

Rank to coordinates mapping

ASTRONOMY DATA AND COMPUTING SERVICES

You can retrieve what coordinates a rank has in a Cartesian grid and vice versa

```
MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

IN:

maxdim -- Maximum number of dimensions

rank

comm

OUT:

coords -- an array of size maxdims to hold return values of coordinates for in each dimension

```
MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

IN:

comm

coords - an array of size ndims holding the queried coordinates

OUT:

rank – the return value is the rank of the process having these coordinates.

Exercise 4: Ring

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- Write an MPI program to do the following
  - Pass a message (sum of all ranks) around a ring in using a 1D Cartesian communicator
  - Use communication as before, but in new Cartesian communicator
  - Hints:
    - MPI_Dims_create()
    - MPI_Cart_create()
    - MPI_Cart_shift()
    - Use perioidic boundary conditions

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

MPI Collectives

"The First Tractor" by Vladimir Krikhatsky (socialist realist, 1877-1942). Source: http://en.wikipedia.org/wiki/File:Wladimir_Gawriilowitsch_Krikhatzkij_-_The_First_Tractor.jpg

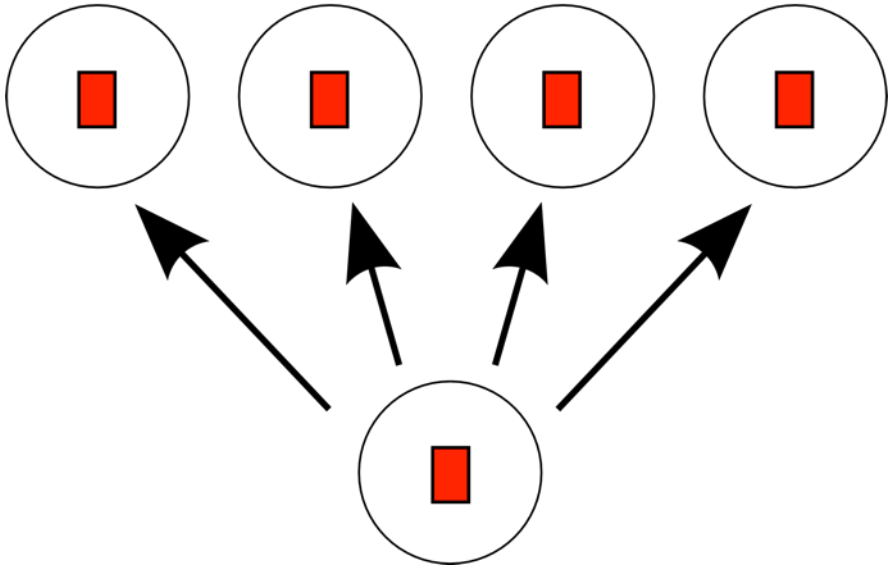
MPI Collectives

ASTRONOMY DATA AND COMPUTING SERVICES

- Communication involving group of processes
- Collective operations
  - Broadcast
  - Gather
  - Scatter
  - Reduce
  - All- {Gather, Scatter, Reduce}
  - Barrier
- Called by all processes in a communicator

Broadcast

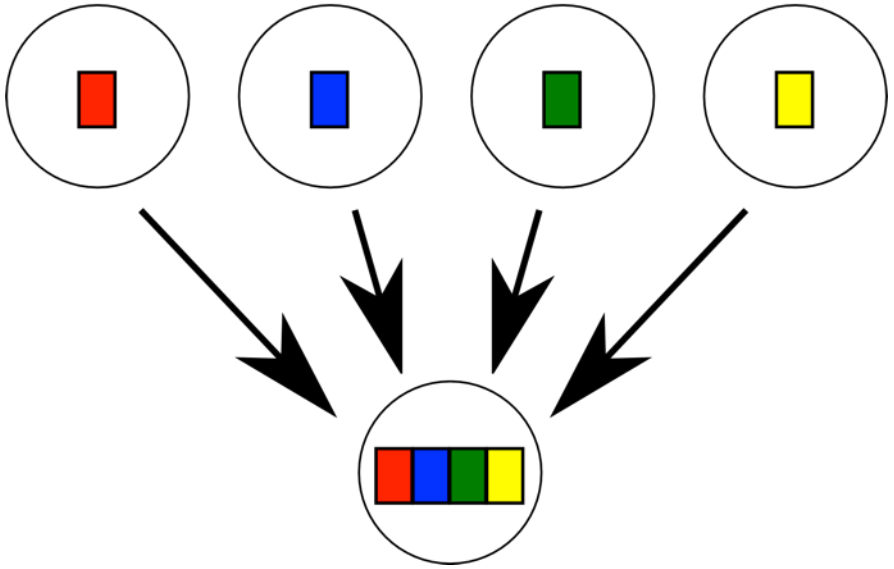
- One processor distributes a copy of data to all other processors in a communicator
- More efficient than multiple individual messages



```
int MPI_Bcast(void* buffer, int count,
MPI_Datatype datatype, int root, MPI_Comm comm)
```

Gather

- Data (of same type) on multiple processors collected onto single processor
- Messages concatenated in rank order

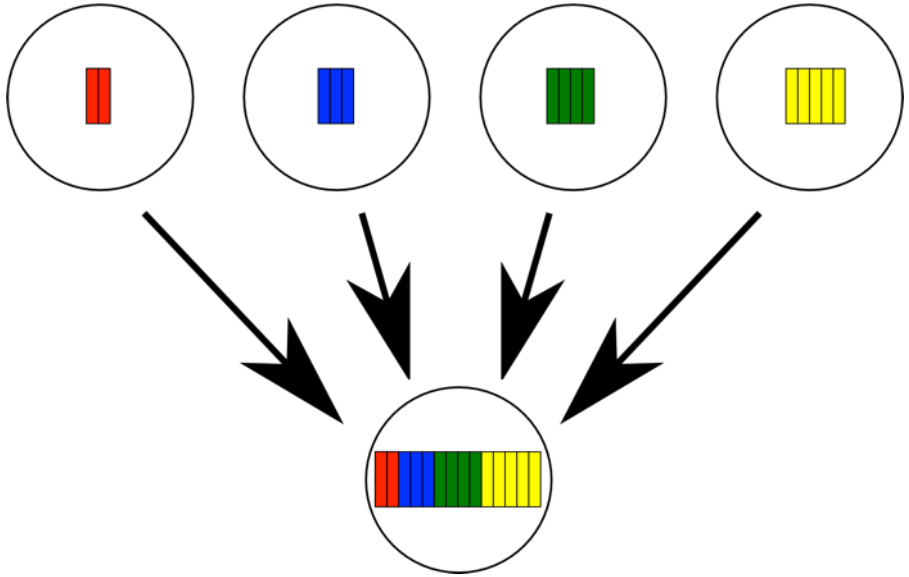


```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

Note: **`recvcount`** = number of items received from each process, not total

Gatherv

- Same as gather operations, but allows varying data count from each process with **`MPI_Gatherv(…)`**

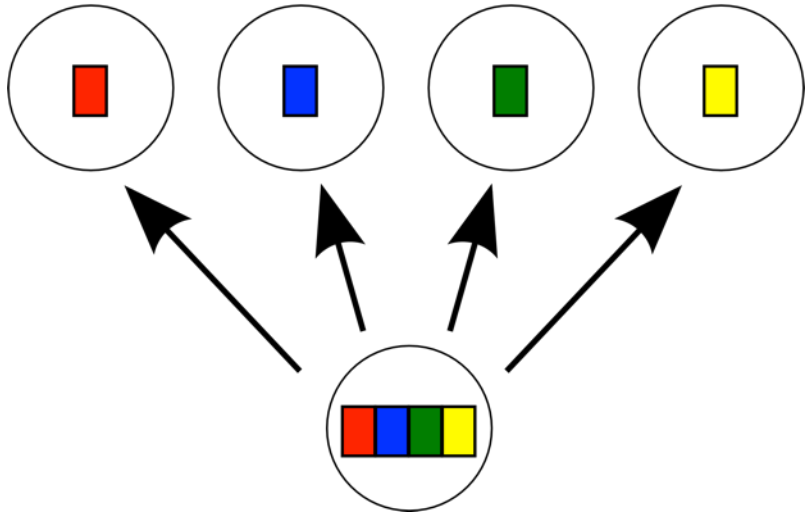


```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int *recvcounts, int *displs,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

`recvcounts` is array; ith entry of **displs** array specifies displacement relative to **recvbuf[0]** at which to place data from receiving process number

Scatter

- Inverse of gather
- Split message into **np** equal pieces, with **i**th segment sent to **i**th process in group

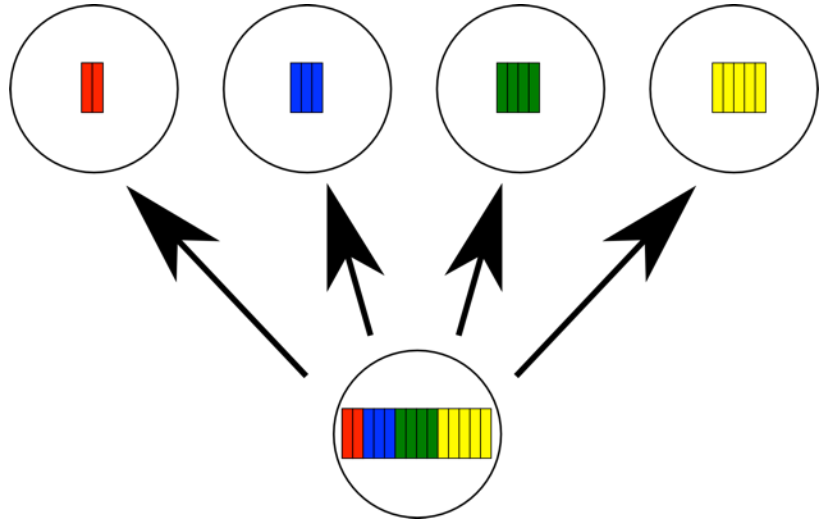


```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

Scatterv

- Send messages of varying sizes across processes in group:

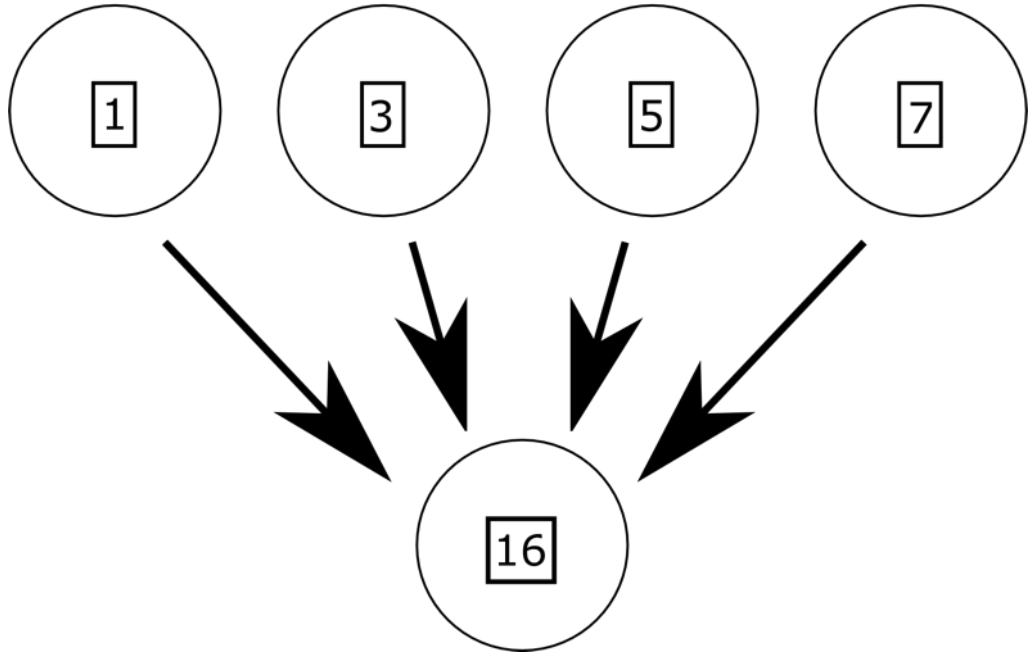
  `MPI_Scatterv(…)`



```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int
*displs, MPI_datatype sendtype, void* recvbuf, int
recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Reduction

- Data is collected onto a single processor
- Operation performed on all data, producing a single value
- Example: Sum all processor values




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

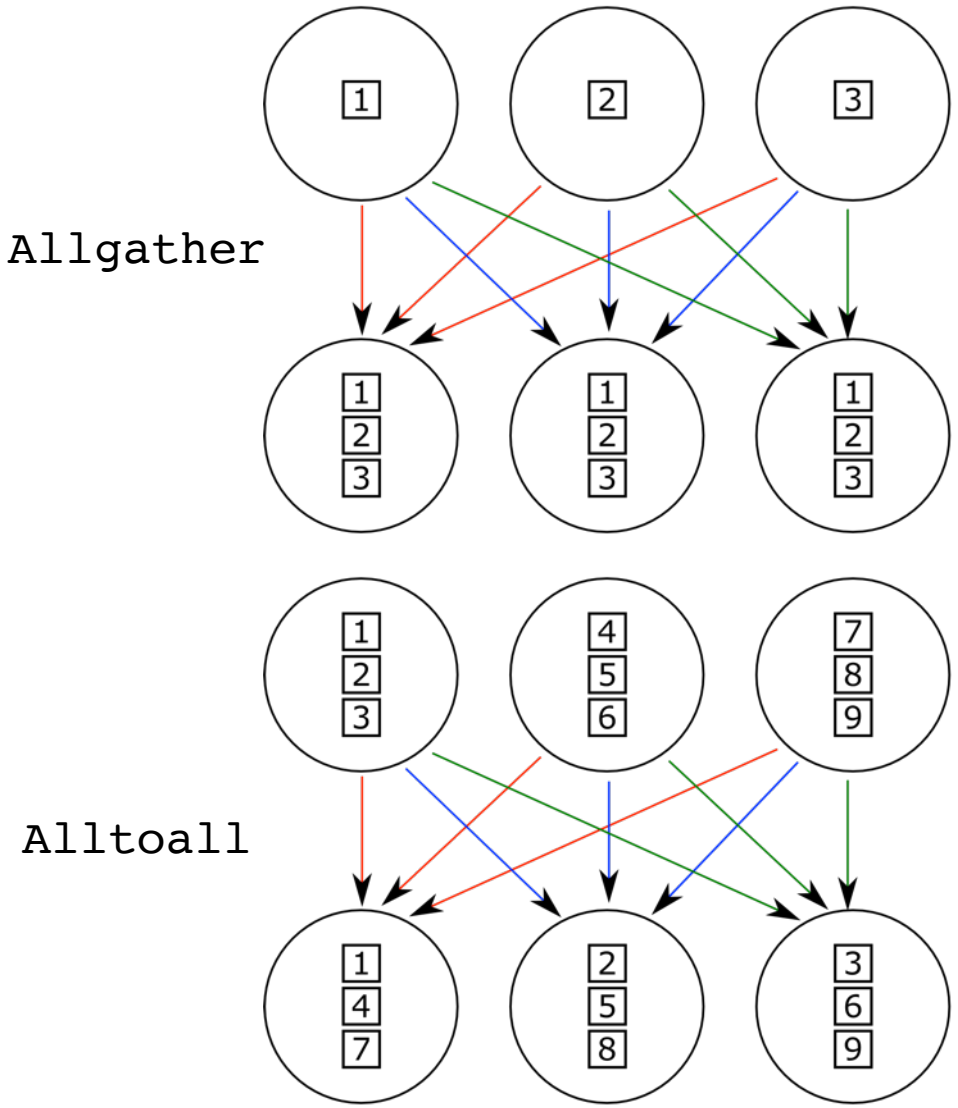
Reduce: Predefined Operations

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

MPI_Op	Meaning	Allowed Types
MPI_MAX	Maximum	Integer, floating point
MPI_MIN	Minimum	Integer, floating point
MPI_SUM	Sum	Integer, floating point, complex
MPI_PROD	Product	Integer, floating point, complex
MPI_LAND	Logical and	Integer, logical
MPI_BAND	Bitwise and	Integer, logical
MPI_LOR	Logical or	Integer, logical
MPI_BOR	Bitwise or	Integer, logical
MPI_LXOR	Logical xor	Integer, logical
MPI_BXOR	Bitwise xor	Integer, logical
MPI_MAXLOC	Maximum value & location	*
MPI_MINLOC	Minimum value & location	*

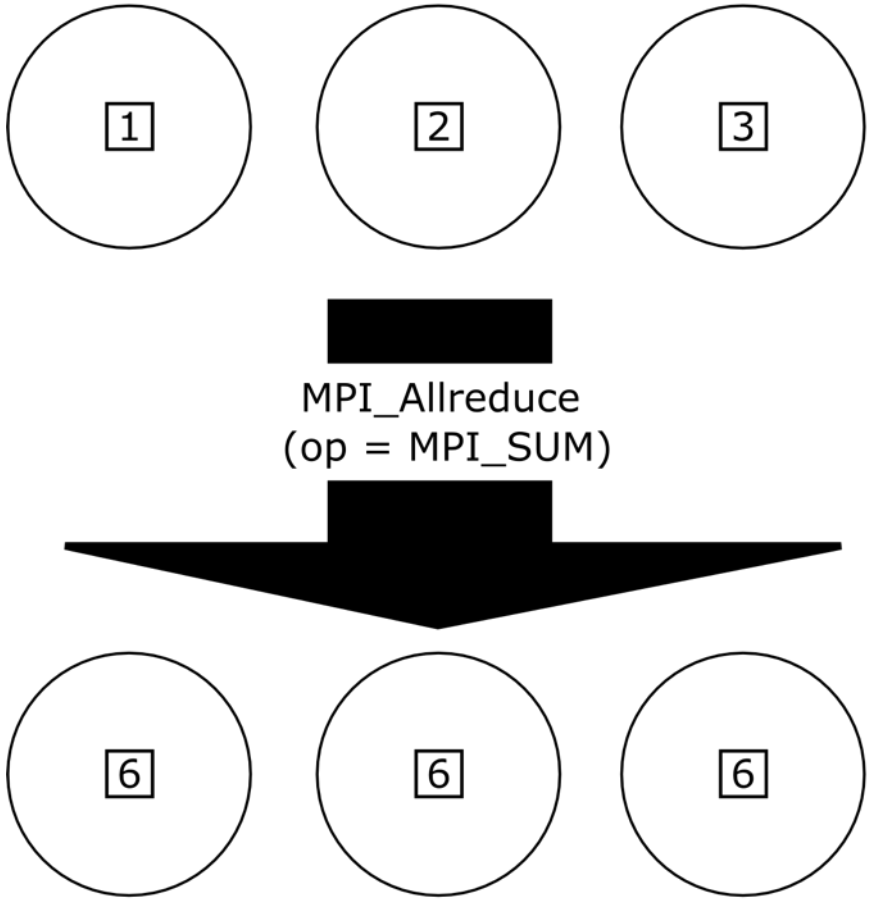
All- Operations

Typically expensive operations – perform well on Magnus at scale – performance is interconnect dependent.



All-Reduce

- Same as **`MPI_Reduce`** except result appears on all processes




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

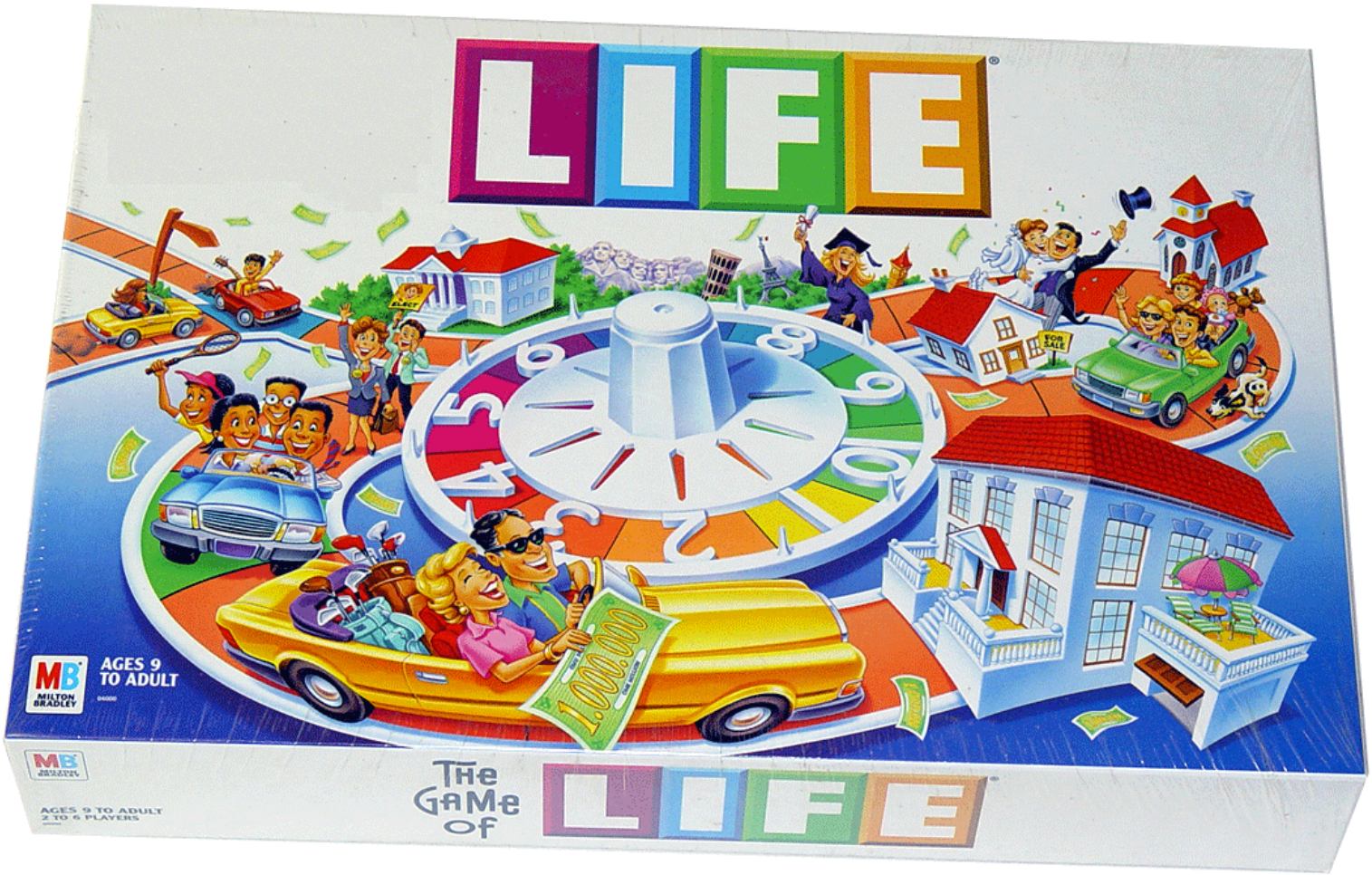
Barrier

- In algorithm, may need to synchronize processes
- Barrier blocks until all group members have called it
- `int` **`MPI_Barrier`**`(MPI_Comm comm)`

Exercise 5: Collective

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- Write an MPI program to do the following
  - Add the integer ranks in a communicator
  - Have each MPI task print the sum
  - Hints
    - Try `MPI_Reduce()`
    - Then `MPI_Allreduce()`



Game of life example

Conway's Game of Life

ASTRONOMY DATA AND COMPUTING SERVICES

- Cellular Automata model
- Grid begins with an initial configuration
  - Cells, either alive or dead
- Rules governing how system evovles
  - Any live cell with fewer than 2 lives neighbors dies
  - Any live cell with 2 or 3 living neighbors lives
  - Any live cell with more than 3 living neighbors dies
  - Any dead cell with exactly 3 live neighbors becomes live

https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

Exercise 6: Game of Life

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- Write an MPI program to carry out Conway's Game of Life
- 1D domain decomposition
- Boundary conditions may be periodic or closed
- Let the system evolve for 12 steps/iterations
- Print the total number of live cells at each step (only process 0 should print)

Suggestions/Hints

ASTRONOMY DATA AND COMPUTING SERVICES

- Use a 1D Cartesian communicator
  - MPI_Cart_shift can be used to figure out left and right neighbors
- Need a way to store both the current state of the grid, and the state of the grid in the next iteration
  - Probably need to allocate local arrays for each MPI process
- Need to set up an initial state for the grid
- Look at collective operations for the printing step

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

Other topics

Parallel I/O

ASTRONOMY DATA AND COMPUTING SERVICES

- Typically one processor writes out data
- Possible choke point if problem has been decomposed over a large number of processors.
  - 100,000 CPUs have to send data to a single CPU to write out data
- Reading and writing data in parallel similar to sending and receiving messages

One-Sided Communication

ADACS
ASTRONOMY DATA AND COMPUTING SERVICES

- Allows 1 process to define communication
  - For both send and receive
- RMA – Remote Memory Access
  - Each process allocates a portion of its memory to a global “window”
- Processes “put” and “get” data
- Advantage:
  - Can be significantly faster than point-to-point
  - Ease of programming (similar to shared-memory programming)
- Disadvantage
  - Implementation-dependent (MPICH vs OpenMPI vs Cray etc.)
  - Hardware-dependent
  - Synchronization issues
- Present in MPI-2, but expanded in MPI-3

Blocking/Non-Blocking Collectives

ASTRONOMY DATA AND COMPUTING SERVICES

- All collective operations are blocking
  - Except in MPI-3
- MPI-3 introduced non-blocking collectives
  - Still vendor dependent

Bibliography/Resources: MPI/ MPI Collectives

ASTRONOMY DATA AND COMPUTING SERVICES

- Snir, Marc, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. (1996) *MPI:The Complete Reference.* Cambridge, MA: MIT Press. (also available at http://www.netlib.org/utk/papers/mpi-book/mpi-book.html)
- MPICH Documentation http://www-unix.mcs.anl.gov/mpi/mpich/
- C, C++, and FORTRAN bindings for MPI-1.2 http://www.lam-mpi.org/tutorials/bindings/

Bibliography/Resources: MPI/ MPI Collectives

- Message Passing Interface (MPI) Tutorial
  https://computing.llnl.gov/tutorials/mpi/
- MPI Standard at MPI Forum
  - MPI 1.1: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
  - MPI-2.2: http://www.mpi-forum.org/docs/mpi22-report/mpi22-report.htm
  - MPI 3.0: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf