Parallel Concepts and MPI

Rebecca Hartman-Baker

Oak Ridge National Laboratory

hartmanbakrj@ornl.gov

Outline

I. Parallelism

II. Supercomputer Architecture

III. Basic MPI

IV. MPI Collectives

V. Advanced Point-to-Point Communication

VI. Communicators

I. PARALLELISM

Parallel Lines **by Blondie. Source:**
http://xponentialmusic.org/blogs/885mmmm/2007/10/09/403-blondie-hits-1-with-heart-of-glass/

I. Parallelism

- Concepts of parallelization
- Serial vs. parallel
- Parallelization strategies

Parallelization Concepts

- When performing task, some subtasks depend on one another, while others do not
- Example: Preparing dinner
  - Salad prep independent of lasagna baking
  - Lasagna must be assembled before baking
- Likewise, in solving scientific problems, some tasks independent of one another

Serial vs. Parallel

- Serial: tasks must be performed in sequence
- Parallel: tasks can be performed independently in any order

Serial vs. Parallel: Example

- Example: Preparing dinner
  - Serial tasks: making sauce, assembling lasagna, baking lasagna; washing lettuce, cutting vegetables, assembling salad
  - Parallel tasks: making lasagna, making salad, setting table

Serial vs. Parallel: Example

- Could have several chefs, each performing one parallel task
- This is concept behind parallel computing

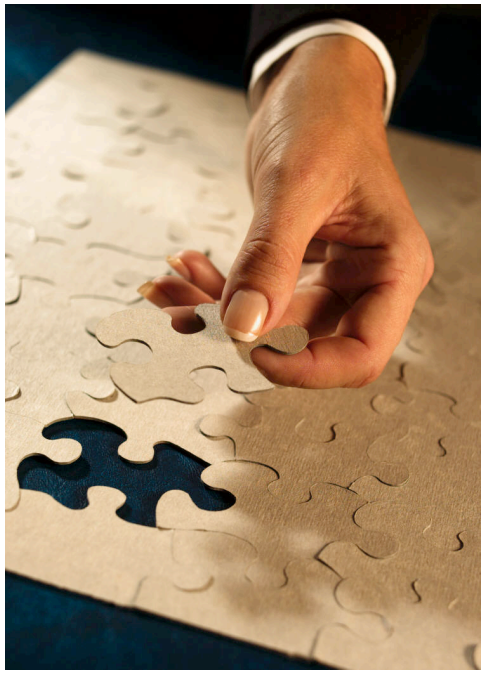
Parallel Algorithm Design: PCAM

- ***P**artition:* **Decompose problem into fine-grained tasks to maximize potential parallelism**
- ***C**ommunication:* **Determine communication pattern among tasks**
- ***A**gglomeration:* **Combine into coarser-grained tasks, if necessary, to reduce communication requirements or other costs**
- ***M**apping:* **Assign tasks to processors, subject to tradeoff between communication cost and concurrency**

(taken from *Heath: Parallel Numerical Algorithms*)

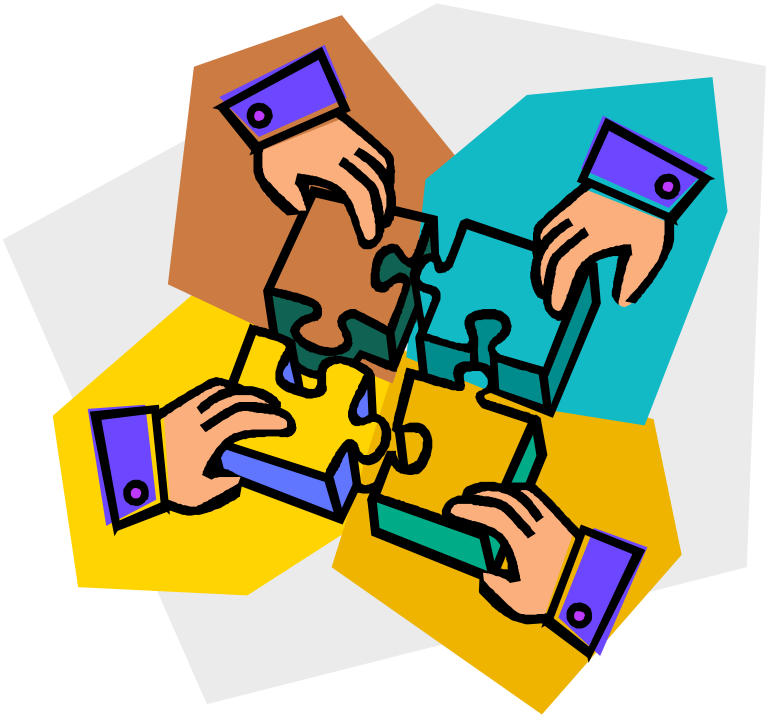
Discussion: Jigsaw Puzzle*

- Suppose we want to do 5000 piece jigsaw puzzle
- Time for one person to complete puzzle: n hours
- How can we decrease walltime to completion?



* Thanks to Henry Neeman

Discussion: Jigsaw Puzzle



- Add another person at the table
  - Effect on wall time
  - Communication
  - Resource contention
- Add p people at the table
  - Effect on wall time
  - Communication
  - Resource contention

Discussion: Jigsaw Puzzle

- What about: p people, p tables, $5000/p$ pieces each?
- What about: one person works on river, one works on sky, one works on mountain, etc.?

II. ARCHITECTURE

Image: Louvre Abu Dhabi – Abu Dhabi, UAE, designed by Jean Nouvel, from
http://www.inhabitat.com/2008/03/31/jean-nouvel-named-2008-pritzker-architecture-laureate/

II. Supercomputer Architecture

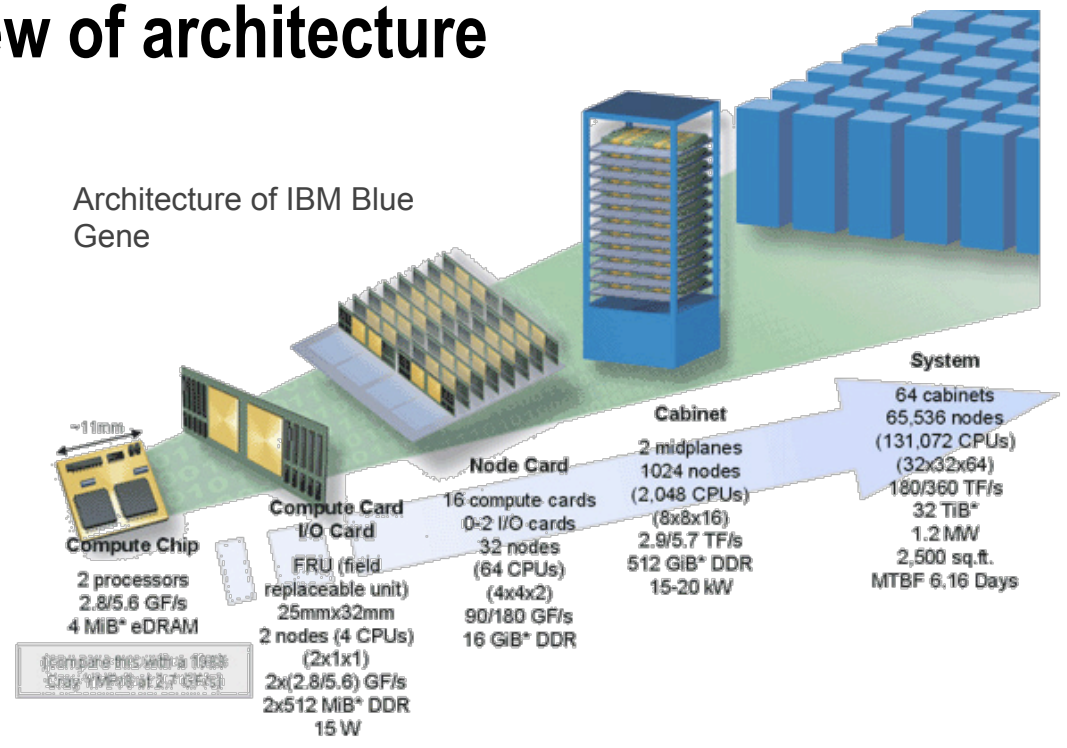
- **What is a supercomputer?**
- **Conceptual overview of architecture**

Cray 1
(1976)

IBM Blue
Gene
(2005)

Cray XT5
(2009)

Architecture of IBM Blue Gene



What Is a Supercomputer?

- "The biggest, fastest computer right this minute." -- Henry Neeman
- Generally 100-10,000 times more powerful than PC
- This field of study known as *supercomputing, high-performance computing (HPC),* or *scientific computing*
- Scientists use really big computers to solve really hard problems

SMP Architecture

- Massive memory, shared by multiple processors
- Any processor can work on any task, no matter its location in memory
- Ideal for parallelization of sums, loops, etc.

Cluster Architecture

- CPUs on racks, do computations (fast)
- Communicate through myrinet connections (slow)
- Want to write programs that divide computations evenly but minimize communication

State-of-the-Art Architectures

- Today, hybrid architectures gaining acceptance
- Multiple {quad, 8, 12}-core nodes, connected to other nodes by (slow) interconnect
- Cores in node share memory (like small SMP machines)
- Machine appears to follow cluster architecture (with multi-core nodes rather than single processors)
- To take advantage of all parallelism, use MPI (cluster) and OpenMP (SMP) hybrid programming

III. MPI

MPI also stands for Max Planck Institute for Psycholinguistics. Source: http://www.mpi.nl/WhatWeDo/istitute-pictures/building

III. Basic MPI

- Introduction to MPI
- Parallel programming concepts
- The Six Necessary MPI Commands
- Example program

Introduction to MPI

- Stands for *M*essage *P*assing *I*nterface
- Industry standard for parallel programming (200+ page document)
- MPI implemented by many vendors; open source implementations available too
  - ChaMPIon-PRO, IBM, HP, Cray vendor implementations
  - MPICH, LAM-MPI, OpenMPI (open source)
- MPI function library is used in writing C, C++, or Fortran programs in HPC
- MPI-1 vs. MPI-2: MPI-2 has additional advanced functionality and C++ bindings, but everything learned today applies to both standards

Parallelization Concepts

- Two primary programming paradigms:
  - SPMD (single program, multiple data)
  - MPMD (multiple programs, multiple data)
- MPI can be used for either paradigm

SPMD vs. MPMD

- SPMD: Write single program that will perform same operation on multiple sets of data
  - Multiple chefs baking many lasagnas
  - Rendering different frames of movie
- MPMD: Write different programs to perform different operations on multiple sets of data
  - Multiple chefs preparing four-course dinner
  - Rendering different parts of movie frame
- Can also write hybrid program in which some processes perform same task

The Six Necessary MPI Commands

- `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 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)`

Initiation and Termination

- `MPI_Init(int *argc, char **argv)` initiates MPI
  - Place in body of code after variable declarations and before any MPI commands
- `MPI_Finalize(void)` shuts down MPI
  - Place near end of code, after last MPI command

Environmental Inquiry

- `MPI_Comm_size(MPI_Comm comm, int *size)`
  - Find out number of processes
  - Allows flexibility in number of processes used in program
- `MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Find out identifier of current process
  - 0 ≤ `rank` ≤ `size`-1

Message Passing: Send

- **`MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`**
  - Send message of length **`count`** bytes and datatype **`datatype`** contained in **`buf`** with tag **`tag`** to process number **`dest`** in communicator **`comm`**
  - E.g. **`MPI_Send(&x, 1, MPI_DOUBLE, manager, me, MPI_COMM_WORLD)`**

Message Passing: Receive

- `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - Receive message of length `count` bytes and datatype `datatype` with tag `tag` in buffer `buf` from process number `source` in communicator `comm` and record status `status`
  - E.g. `MPI_Recv(&x, 1, MPI_DOUBLE, source, source, MPI_COMM_WORLD, &status)`

Message Passing

- WARNING! Both standard send and receive functions are *blocking*
- `MPI_Recv` returns only after receive buffer contains requested message
- `MPI_Send` may or may not block until message received (usually blocks)
- Must watch out for deadlock

Deadlocking Example (Always)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
  int me, np, q, sendto;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  if (np%2==1) return 0;
  if (me%2==1) {sendto = me-1;}
  else {sendto = me+1;}
  MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD,
   &status);
  MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
  printf("Sent %d to proc %d, received %d from proc %d\n", me,
   sendto, q, sendto);
  MPI_Finalize();
  return 0;
}
```

Deadlocking Example (Sometimes)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
  int me, np, q, sendto;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  if (np%2==1) return 0;
  if (me%2==1) {sendto = me-1;}
  else {sendto = me+1;}
  MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
  MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD,
   &status);
  printf("Sent %d to proc %d, received %d from proc %d\n", me,
   sendto, q, sendto);
  MPI_Finalize();
  return 0;
}
```

Deadlocking Example (Safe)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
  int me, np, q, sendto;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  if (np%2==1) return 0;
  if (me%2==1) {sendto = me-1;}
  else {sendto = me+1;}
  if (me%2 == 0) {
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD,
   &status);
  } else {
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD,
   &status);
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
  }
  printf("Sent %d to proc %d, received %d from proc %d\n", me,
   sendto, q, sendto);
  MPI_Finalize();
  return 0;
}
```

Explanation: Always Deadlock Example

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

Explanation: Sometimes Deadlock Example

- Logically correct
- Deadlock could be caused by `MPI_Sends` competing for buffer space
- Unsafe because depends on system resources
- Solutions:
  - Reorder sends and receives, like safe example, having evens send first and odds send second
  - Use non-blocking sends and receives or other advanced functions from MPI library (see MPI standard for details)

IV. MPI COLLECTIVES

“Collective Farm Harvest Festival” (1937) by Sergei Gerasimov. Source:
http://max.mmlc.northwestern.edu/~mdenner/Drama/visualarts/neorealism/34harvest.html

MPI Collectives

- Communication involving group of processes
- Collective operations
  - Broadcast
  - Gather
  - Scatter
  - Reduce
  - All-
  - Barrier

Broadcast

- Perhaps one message needs to be sent from manager to all worker processes
- Could send individual messages
- Instead, use broadcast – more efficient, faster
- `int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`

Gather

- All processes need to send same (similar) message to manager
- Could implement with each process calling `MPI_Send(…)` and manager looping through `MPI_Recv(…)`
- Instead, use gather operation – more efficient, faster
- 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

Gather

- Maybe some processes need to send longer messages than others
- Allow 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; entry `i` in `displs` array specifies displacement relative to `recvbuf[0]` at which to place data from corresponding 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)`
- 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)`

Reduce

- Perhaps we need to do sum of many subsums owned by all processors
- Perhaps we need to find maximum value of variable across all processors
- Perform global reduce operation across all group members
- `int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`

Reduce: Predefined Operations

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 and location	*
MPI_MINLOC	Minimum value and location	*

Reduce: Operations

- **`MPI_MAXLOC`** and **`MPI_MINLOC`**
  - Returns {max, min} and rank of first process with that value
  - Use with special MPI pair datatype arguments:
    - **`MPI_FLOAT_INT`** (`float` and `int`)
    - **`MPI_DOUBLE_INT`** (`double` and `int`)
    - **`MPI_LONG_INT`** (`long` and `int`)
    - **`MPI_2INT`** (pair of `int`)
  - See MPI standard for more details
- User-defined operations
  - Use **`MPI_Op_create(…)`** to create new operations
  - See MPI standard for more details

All- Operations

- Sometimes, may want to have result of gather, scatter, or reduce on all processes
- Gather operations
  - `int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
  - `int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)`

All-to-All Scatter/Gather

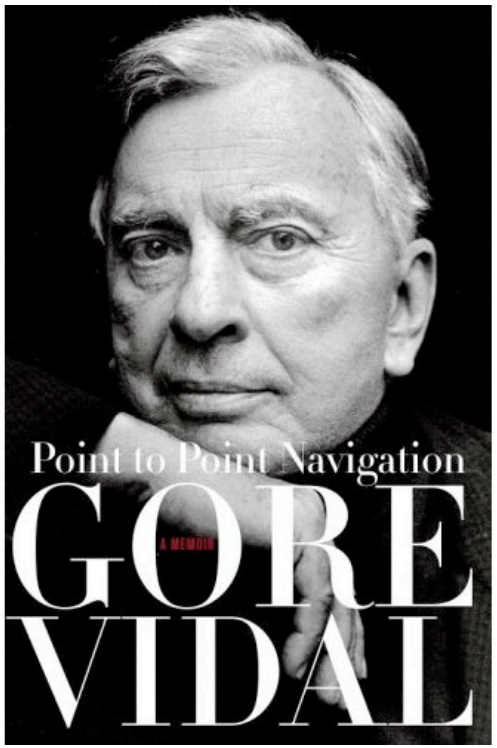
- Extension of **`Allgather`** in which each process sends distinct data to each receiver
- Block **`j`** from process **`i`** is received by process **`j`** into **`i`**th block of **`recvbuf`**
- **`int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`**
- Also corresponding **`AlltoAllv`** function available

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)`



V. POINT-TO-POINT OPERATIONS

Point to Point Navigation: **Gore Vidal's Autobiography (ISBN 0307275019)**

Point-to-Point Operations

- Message passing overview
- Types of operations
  - Blocking
  - Buffered
  - Synchronous
  - Ready
- Communication completion
- Combined operations

Definitions

- **Blocking**
  - Returns only after message data safely stored away, so sender is free to access and overwrite send buffer
- **Buffered**
  - Create a location in memory to store message
  - Operation can complete before matching receive posted
- **Synchronous**
  - Start with or without matching receive
  - Cannot complete without matching receive posted
- **Ready**
  - Start only with matching receive already posted

Message Passing Overview

- How messages are passed
  - Message data placed with message “envelope,” consisting of source, destination, tag, and communicator
- Entire envelope transmitted to other processor
- Count (second argument in Send or Recv) is upper bound on size of message
  - Overflow error occurs if incoming data too large for buffer

Blocking

- Returns after message data and envelope safely stored away so sender is free to access and overwrite send buffer
- Could be copied into temporary system buffer or matching receive buffer – standard does not specify
- Non-local: successful completion of send operation depends on matching receive

Buffered

- Can begin and complete before matching receive posted
- Local: completion does not depend on occurrence of matching receive
- Use with `MPI_Buffer_attach(…)`

Synchronous

- Can be started before matching receive posted
- Completes only after matching receive is posted and begins to receive message
- Non-local: communication does not complete without matching receive posted

Ready

- May be started only if matching receive already posted
- If no matching receive posted, outcome is undefined
- May improve performance by removing hand-shake operation
- Completion does not depend on status of matching receive; merely indicates that send buffer is reusable
- Replacing ready send with standard send in correct program changes only performance

The (Secret) Code

Abbreviation	Meaning	Example
(empty)	Blocking	`MPI_Send(…), MPI_Recv(…)`
S	Synchronous	`MPI_Ssend(…), MPI_Srecv(…)`
B	Buffered	`MPI_Bsend(…), MPI_Brecv(…)`
R	Ready	`MPI_Rsend(…), MPI_Rrecv(…)`
I	Non-Blocking (immediate)	`MPI_Isend(…), MPI_Irecv(…), MPI_Issend(…), MPI_Ibsend(…), MPI_Irsend(…)`

Completion

- Nonblocking communications use `MPI_Wait(…)` and `MPI_Test(…)` to complete
- `MPI_Wait(MPI_Request *request, MPI_Status *status)`
  - Returns when request is complete
- `MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)`
  - Returns `flag = true` if request is complete; `flag = false` otherwise

Completion: Example

```
MPI_Request *request;
MPI_Comm_rank(MPI_COMM_WORLD, &me);
if (me == 0) {
  MPI_Isend(my_array, array_size,
  MPI_DOUBLE, 1, tag, MPI_COMM_WORLD,
  request);
  // do some work
  MPI_Wait(request, status);
} else {
  MPI_Irecv(my_array, array_size,
  MPI_DOUBLE, 0, tag, MPI_COMM_WORLD,
  request);
  // do some work until I need my_array
  MPI_Wait(request, status);
}
```

Multiple Completions

- Want to await completion of any, some, or all communications, instead of specific message
- Use `MPI_{Wait,Test}{any,all,some}` for this purpose
  - `any`: `Waits` or `Tests` for any one option in array of requests to complete
  - `all`: `Waits` or `Tests` for all options in array of requests to complete
  - `some`: `Waits` or `Tests` for all enabled operations in array of requests to complete

Multiple Completions: Syntax (1)

- `int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)`
  - Returns index of request that completed in `index` (or `MPI_UNDEFINED` if array empty or contains no incomplete requests)
- `int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses)`
  - Returns `flag = true` if all communications associated with active handles in array have completed; each status entry corresponding to null or inactive handles set to empty

Multiple Completions: Syntax (2)

- `int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)`
  - Waits until at least one operation associated with active handles has completed; returns in first `outcount` locations within `array_of_status` the status for completed operations

Send-Receive

- Combine into one call sending of message to one destination and receipt of message from another process – not necessarily same one, but within same communicator
- Message sent by send-receive can be received by regular receive or probe
- Send-receive can receive message sent by regular send operation
- Send-receive is blocking
- ```
  int MPI_Sendrecv(void *sendbuf, int
  sendcount, MPI_Datatype sendtype, int
  dest, int sendtag, void *recvbuf, int
  recvcount, MPI_Datatype recvtype, int
  source, int recvtag, MPI_Comm comm,
  MPI_Status *status)
  ```
  ```

VI. COMMUNICATORS

Star Trek Communicators, available for sale at Mark Bergin Toys:
http://www.bergintoys.com/sp_guns/2005-Feb/index.html

VI. Communicators

- Motivation
- Definitions
- Communicators
- Topologies

Motivation

- Perhaps you created hierarchical algorithm in which there are manager, middle-manager, and worker groups
  - Workers communicate with their middle-manager
  - Middle-managers communicate with each other and manager
- Perhaps subdivide work into chunks and associate subset of processors with each chunk
- Perhaps subdivide problem domain and need to associate problem topology with process topology
- Defining subsets of processors would make these algorithms easier to implement

Definitions

- **Group**
  - Ordered set of processes (0 → N-1)
  - One process can belong to multiple groups
  - Used within communicator to identify members
- **Communicator**
  - Determines "communication world" in which communication occurs
  - Contains a group; source and destination of messages defined by rank within group
  - *Intracommunicator:* used for communication within single group of processes
  - *Intercommunicator:* point-to-point communication between 2 disjoint groups of processes (MPI-1) and collective communication within 2 or more groups of processes (MPI-2 only)

Universal Communicator

- **`MPI_COMM_WORLD`**
  - Default communicator
  - Defined at **`MPI_Init(…)`**
  - Static in MPI-1; in MPI-2, processes can dynamically join, so **`MPI_COMM_WORLD`** may differ on different processes

Creating and Using Communicators

- First, create Group that will form communicator
  - Extract global group handle from `MPI_COMM_WORLD` with `MPI_Comm_group(…)`
  - Use `MPI_Group_incl(…)` to form group from subset of global group
- Create new communicator using `MPI_Comm_create(…)`
- Find new rank using `MPI_Comm_rank(…)`
- Do communications
- Free communicator and group using `MPI_Comm_free(…)` and `MPI_Group_free(…)`

Example: Separate Collectives

```
/* Assume 8 processes */
#include <mpi.h>
#include <stdio.h>
int main (int argc, char
   **argv) {
int me, rank, sbuf, rbuf;
int ranksg1[4] = {0,1,2,3},
   ranksg2[4] = {4,5,6,7};
MPI_Group world, newgroup;
MPI_Comm newcomm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
   &me);
sbuf = me;
/* Extract original group
   handle */
MPI_Comm_group(MPI_COMM_WORLD,
   &world);
/* Divide tasks into 2 groups
   based on rank */
if (me < 4) {
  MPI_Group_incl(world, 4,
   ranksg1, &newgroup);
} else {
  MPI_Group_incl(world, 4,
   ranksg2, &newgroup);
}
/* Create newcomm and do work
   */
MPI_Comm_create
   (MPI_COMM_WORLD, newgroup,
   &newcomm);
MPI_Allreduce(&sbuf, &rbuf, 1,
   MPI_INT, MPI_SUM, newcomm);
MPI_Group_rank(newgroup,
   rank);
printf("me=%d, rank=%d, rbuf=%
   d\n", me, rank, rbuf);
MPI_Finalize();
}
```

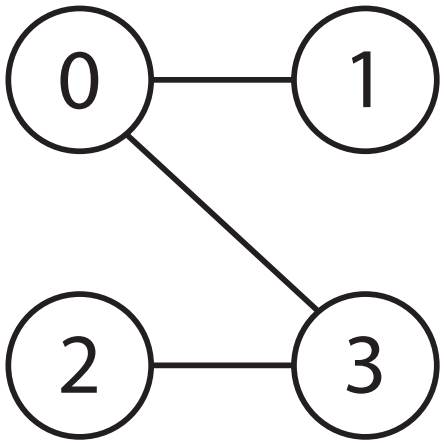
Virtual Topologies

- Mapping of MPI processes into geometric shape, e.g., Cartesian, Graph
- Topology is virtual – “neighbors” in topology not necessarily “neighbors” in machine
- Virtual topologies can be useful:
  - Algorithmic communication patterns might follow structure of topology

Virtual Topology Constructors

- `int MPI_Cart_create(MPI_Comm comm_old, int ndims int *dims, int *periods, int reorder, MPI_Comm *comm_cart)`
  - Returns in `comm_cart` new communicator with cartesian structure
  - If `reorder == false`, then ranks in new communicator remain identical to ranks in old communicator
  - Can use `MPI_Dims_create(int nnodes, int ndims, int *dims)` to create `dims` array
- `int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges, int reorder, MPI_Comm *comm_graph)`
  - Returns in `comm_graph` new communicator with graph structure
  - `nnodes` = # nodes in graph
  - `index` = array storing cumulative number of neighbors
  - `edges` = flattened representation of edge lists

Example: Graph Constructor



- **nnodes = 4**
- **index = {2,3,4,6}**
- **edges = {1,3,0,3,0,2}**

Topology Inquiry Functions

- `int MPI_Topo_test(MPI_Comm comm, int *status)`
  - Returns type of topology in output value status: **`MPI_GRAPH`** (graph topology), **`MPI_CART`** (cartesian topology), or **`MPI_UNDEFINED`** (no topology)
- `int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)`
  - Input integer array (of size **`ndims`**) specifying cartesian coordinates
  - Returns rank of process represented by coords in **`rank`**

Example: 2-D Parallel Poisson Solver

```
/* Assume variables
   predefined as appropriate
   */
int ND = 2, NNB = 4;
int dims[ND], my_pos[ND],
   nbr[ND], my_nbrs[NNB];
/* Set grid size and
   periodicity */
MPI_Dims_create(comm, ND,
   dims);
int periods = {1,1};
/* Create grid and inquire
   about own position */
MPI_Cart_create(comm, ND,
   dims, periods, reorder,
   comm_cart);
MPI_Cart_get(comm_cart, ND,
   dims, periods, my_pos);
/* Look up my neighbors */
nbr[0] = my_pos[0]-1;
nbr[1] = my_pos[1];
MPI_Cart_rank(comm_cart, nbr,
   my_nbrs[0]);
nbr[0] = my_pos[0]+1;
MPI_Cart_rank(comm_cart, nbr,
   my_nbrs[1]);
nbr[0] = my_pos[0];
nbr[1] = my_pos[1]-1;
MPI_Cart_rank(comm_cart, nbr,
   my_nbrs[2]);
nbr[1] = my_pos[1]+1;
MPI_Cart_rank(comm_cart, nbr,
   my_nbrs[3]);
/* Now do work */
initialize(u, f);
for (i=0; i<100; i++) {
  relax(u,f);
/* Exchange data */
  exchange(u, comm_cart,
   my_nbrs, NNB);
}
output(u);
```

Bibliography/Resources: Programming Concepts

- Heath, Michael T. (2006) *Notes for CS554: Parallel Numerical Algorithms,* http://www.cse.uiuc.edu/cs554/notes/index.html
- MPI Deadlock and Suggestions http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/CommonDoc/MessPass/MPIDeadlock.html

Bibliography/Resources: MPI/ MPI Collectives

- 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.gov/docs/mpi-11-html/mpi-report.html#Node0
  - MPI-2:
    http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.htm#Node0