MPI Programming Model

Customer Documentation and Training

What is MPI?

- MPI (Message Passing Interface) is a portable message passing style of parallel programming
  - Available on all HPC vendor platforms today
  - Most widely used HPC parallel programming style
  - Contains a rich set of routines, yet most programs use only a few of the routines
- Cray XT system uses standard MPICH-2 developed at Argonne National Laboratory
  - Full implementation of the MPI-2 standard, with the exception of the spawn functions
- Bindings for Fortran, C, and C++

General MPI Model

- Execution model allows each task to operate separately
  - Tasks generally are created at startup and continue throughout the entire execution
  - Synchronization is implicit in each point-to-point or collective data movement
- Memory model assumes that memory is private to each task
  - Allows mapping to single-address-space systems
  - Either distributed memory or shared memory systems
- Implemented as users’ calls to library functions
  - Move data point-to-point between tasks
  - Perform some collective computations

MPI Processes

- An MPI program consists of autonomous processes
  - The processes may run either the same code (SPMD style) or different codes (heterogeneous)
- Processes communicate with each other via calls to MPI functions
- A process can be sequential or multithreaded
  - MPI does not specify the initial allocation of processes
  - Cray XT systems that run Catamount on the compute nodes do not support multithreaded applications
  - Cray XT systems that run CNL on the compute nodes support multithreaded applications, such as applications using OpenMP

MPI Basics

- Communicator
  - An ordered set of processes, either system- or user-defined
  - The default communicator is: MPI_COMM_WORLD
  - The MPI_Comm_size function returns the number of processes in the communicator
- Rank
  - Your process number within a communicator
  - Used for actual sends and receives
  - The MPI_Comm_rank function returns the process rank within a communicator

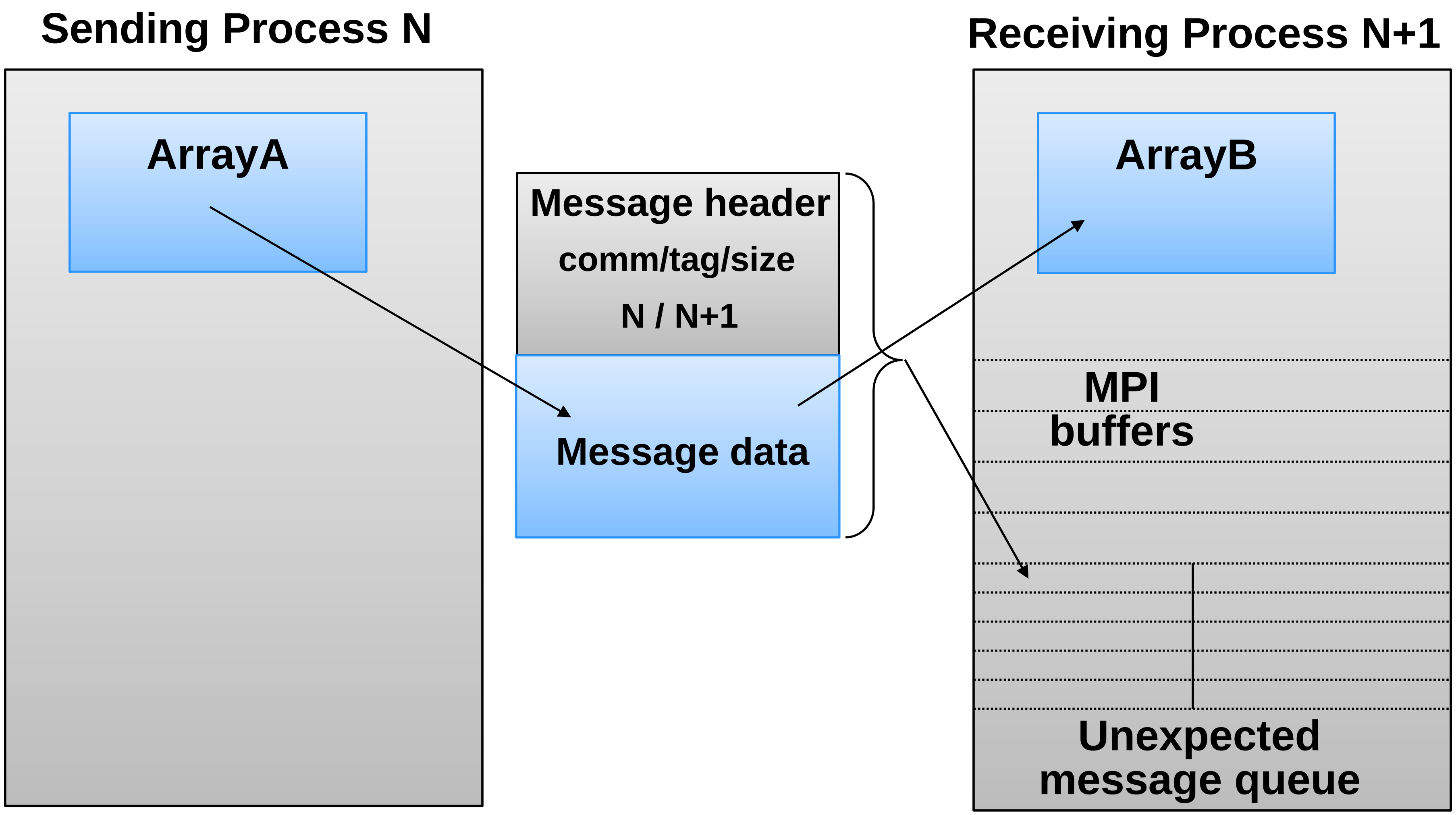
MPI Message Matching

- MPI enables an operation to control which messages it receives
  – MPI uses the source and tag argument to perform this matching
    - Source
      - The source specifier in the MPI_Recv function allows the programmer to specify that a message will be received either from a single named process (specified by its integer process identifier) or from any process (specified by the special value MPI_ANY_SOURCE)
    - Tag
      - Message tags provide another way to distinguish between different messages: a sending process must associate an integer tag with a message via the tag field in the MPI_Send call; a receiving process can then specify that it will receive messages either with a specified tag or with any tag (MPI_ANY_TAG)

MPI Message

- A message consists of:
  - An *envelope* portion
    - The exact definition depends on the implementation
    - Typically consists of the message tag, communicator, source, destination, and possibly the message length
  - A *data* portion
    - Contains the information to be passed
    - The exact definition depends on the implementation
      - Using standard or derived datatypes
- A message exists within a communicator
  - For example: MPI_COMM_WORLD

MPI Messages



Determinism

- Message-passing programming models are nondeterministic by default: the order of arrival of messages from two processes, A and B, to a third process, C, is not defined
  - The programmer must ensure that a communication is deterministic when this is required (as is usually the case)
  - However, MPI does guarantee that two messages from one process, A, to another process, B, will arrive in the order they were sent

MPI in Fortran

- **Function names are in uppercase; e.g., MPI_RECV**

  ```
  CALL MPI_XXXX(parameter, ... , IERROR)
  ```

- **Function return codes are represented by an additional integer argument. The return code for successful completion is MPI_SUCCESS; a set of error codes is also defined**

- **Compile-time constants are in uppercase and are defined in the mpif.h file, which must be included in any program that makes MPI calls**
  - **MPI Fortran header files:**

    ```
    INCLUDE 'mpif.h'
    ```

MPI in Fortran

- **An MPI datatype is defined for each Fortran datatype: `MPI_INTEGER`, `MPI_REAL`, `MPI_DOUBLE_PRECISION`, `MPI_COMPLEX`, `MPI_LOGICAL`, `MPI_CHARACTER`, etc.**
- **A status variable is an array of integers of size `MPI_STATUS_SIZE`; the constants `MPI_SOURCE` and `MPI_TAG` index the source and tag fields, respectively**
  - **All handles have type `INTEGER`**

MPI in C and C++

- **Function names have the MPI prefix and the first letter of the function name in upper case; e.g. `MPI_Recv`**

  ```
  error = MPI_Xxxxx(parameter, ...);
  ```

  - **Compile-time constants are defined in the `mpi.h` file, which must be included in any program that makes MPI calls**
    - **MPI C / C++ header file**

      ```
      #include <mpi.h>
      ```

  - **An MPI datatype is defined for each C datatype:**
    `MPI_CHAR`, `MPI_INT`, `MPI_LONG`, `MPI_UNSIGNED_CHAR`, `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG_DOUBLE`, etc.
  - **Function parameters with type IN are passed by value; parameters with type OUT and INOUT are passed by reference (that is, as pointers)**

MPI in C and C++

- Status values are returned as integer return codes. The return code for successful completion is MPI_SUCCESS; a set of error codes is also defined
  - A status variable has type MPI_Status and is a structure with fields, status.MPI_SOURCE and status.MPI_TAG, that contain source and tag information
  - Handles are represented by special defined types, which are defined in mpi.h

Basic Functions

- **MPI can be very simple. These six functions enable you to write many programs:**

```
MPI_Init
MPI_Comm_size
MPI_Comm_rank
MPI_Send
MPI_Recv
MPI_Finalize
```

MPI Initialization

- MPI processes launch during program startup, before user MAIN
  - MPI rank 0 is the root process
  - All processes in `MPI_Init`:
    - Read environment variables
    - Initialize local data structures
    - Acquire addresses for remote data structures
    - Initialize I/O and buffers
  - The `MPI_init` function must be the first MPI call
    - Fortran

      ```
      CALL MPI_INIT(IERROR)
      ```
    - C / C++

      ```
      int MPI_Init(int *argc, char ***argv);
      ```
  - It may be called only once
    - Subsequent calls are erroneous

MPI Send

- **SEND – Standard send: a blocking send operation**
  - **Fortran**

```
INTEGER COUNT,DATATYPE,DEST,TAG,COMM,IERROR
CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
  IERROR)
```

  - **C / C++**

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

- **Processes might deadlock if all are trying to send at the same time because a send may require that the message be received before the process can continue (this depends on the implementation).**

MPI Receive

- **Standard receive: a blocking receive operation**
  - **Fortran**

```
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, IERROR
INTEGER STATUS(MPI_STATUS_SIZE)
CALL MPI_RECV (BUF, COUNT, DATATYPE, SOURCE, TAG,
 COMM, STATUS, IERROR)
```

  - **C / C++**

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

MPI Finalize

- **Fortran**

  ```
  CALL MPI_FINALIZE(IERROR)
  ```

- **C / C++**

  ```
  int MPI_Finalize();
  ```

- **Cleans up all MPI state**
  - **All processes must call `MPI_Finalize()`**
    - **An implicit barrier permits proper exit sequence**
    - **Barrier ensures that all communications are complete**
    - **No MPI functions (including `MPI_Init`) can occur after `MPI_Finalize()`**

Fortran Example #1

```
PROGRAM SIMPLE ! SAMPLE 2-PE MPI CODE
INCLUDE 'mpif.h‘
INTEGER, PARAMETER :: N = 1000
INTEGER OTHER_PE
INTEGER SEND, RECV
INTEGER STATUS(MPI_STATUS_SIZE)
REAL, DIMENSION(N) :: RBUF, SBUF
CALL MPI_INIT(IERR)
IF (IERR /= 0) STOP 'BAD INIT‘
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPES, IERR)
IF (IERR /= MPI_SUCCESS) STOP 'BAD SIZE‘
CALL MPI_COMM_RANK(MPI_COMM_WORLD, ME, JERR)
IF (JERR /= MPI_SUCCESS) STOP 'BAD RANK‘
IF (NPES /=2) THEN
    PRINT*,'MUST RUN WITH 2 PES- EXITING‘
    CALL EXIT(2)
ENDIF
```

Fortran Example #1

```
IF (ME == 0) OTHER_PE = 1
IF (ME == 1) OTHER_PE = 0
DO J = 1, N
   SBUF(J) = J
ENDDO
IF (ME == 0) THEN
   CALL MPI_SEND(SBUF, N, MPI_REAL, OTHER_PE, 99, &
       MPI_COMM_WORLD, SEND)
   IF (SEND /= MPI_SUCCESS) STOP 'BAD SEND ON 0‘
     CALL MPI_RECV(RBUF, N, MPI_REAL, OTHER_PE, 99, &
         MPI_COMM_WORLD, STATUS, RECV)
   IF (RECV /= MPI_SUCCESS) STOP 'BAD RECV ON 0‘
ELSE ! PE 1
   CALL MPI_RECV(RBUF, N, MPI_REAL, OTHER_PE, 99, &
       MPI_COMM_WORLD, STATUS, RECV)
   IF (RECV /= MPI_SUCCESS) STOP 'BAD RECV ON 1‘
     CALL MPI_SEND(SBUF, N, MPI_REAL, OTHER_PE, 99, &
        MPI_COMM_WORLD, SEND)
   IF (SEND /= MPI_SUCCESS) STOP 'BAD SEND ON 1‘
ENDIF
```

Fortran Example #1

```
CALL MPI_FINALIZE(IERR)
IF (IERR /= MPI_SUCCESS) STOP 'BAD FINALIZE‘
IFLAG = 1
DO I = 1, N
   IF (RBUF(I) /= SBUF(I)) THEN
       IFLAG = 0
      PRINT*,'PE ', ME,': RBUF(',I,')=',RBUF(I), &
        ' SHOULD BE ', SBUF(I)
   ENDIF
ENDDO
IF (IFLAG == 1) THEN
   PRINT*,'TEST PASSED ON PE ', ME
ELSE
   PRINT*,'TEST FAILED ON PE ', ME
ENDIF
END PROGRAM SIMPLE
```

C Example #1

```
#include <mpi.h> /* sample 2-PE MPI code */
#define N 1000
 main(argc, argv)
 int argc;
 char *argv[];{
 int num_procs;
 int my_proc;
 int init, size, rank, send, recv, final;
 int i, j, other_proc, flag = 1;
 double sbuf[N], rbuf[N];
 MPI_Status recv_status;
/* Initialize MPI */
 if ((init = MPI_Init(&argc, &argv)) != MPI_SUCCESS) {
     printf("bad init\n");
     exit(-2); }
/* Determine the size of the communicator */
 if ((size = MPI_Comm_size(MPI_COMM_WORLD, &num_procs))
     != MPI_SUCCESS) {
     printf("bad size\n");
     exit(2);}
```

C Example #1

```
/* Make sure we run with only 2 processes */
if (num_procs != 2) {
  printf("must run with 2 processes\n");
  exit(1);
}
/* Determine process number */
if ((rank = MPI_Comm_rank(MPI_COMM_WORLD, &my_proc))
  != MPI_SUCCESS) {
  printf("bad rank\n");
  exit(1);
}
if (my_proc == 0) other_proc = 1;
if (my_proc == 1) other_proc = 0;
for (i = 0; i < N; i++)
  sbuf[i] = i;
```

C Example #1

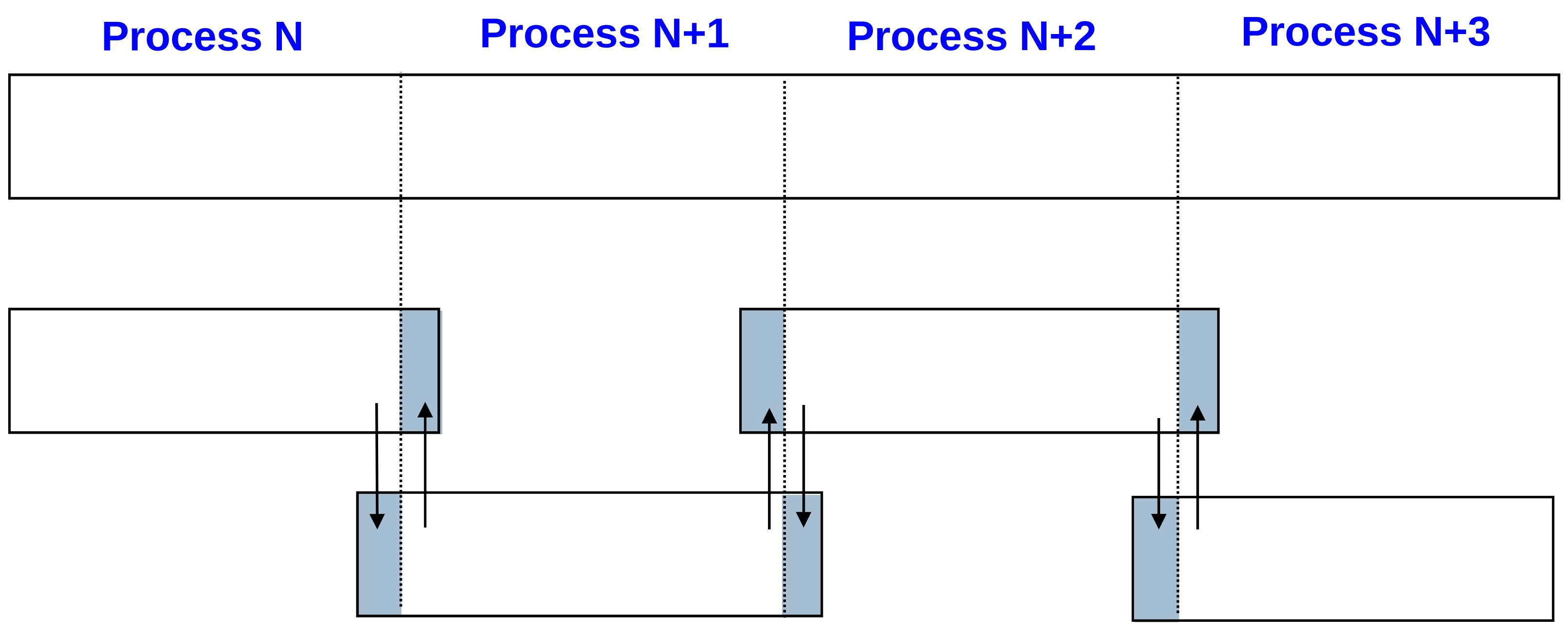
```
/* Both processes send and receive data */
if (my_proc == 0) {
  if ((send = MPI_Send(sbuf, N, MPI_DOUBLE, other_proc,
    99, MPI_COMM_WORLD)) != MPI_SUCCESS) {
    printf("bad send on %d\n",my_proc);
    exit(1); }
  if ((recv = MPI_Recv(rbuf, N, MPI_DOUBLE, other_proc,
    98, MPI_COMM_WORLD, &recv_status)) != MPI_SUCCESS){
    printf("bad recv on %d\n", my_proc);
    exit(1); }
}
else if (my_proc == 1) {
  if ((recv = MPI_Recv(rbuf, N, MPI_DOUBLE, other_proc,
    99, MPI_COMM_WORLD, &recv_status)) != MPI_SUCCESS){
    printf("bad recv on %d\n", my_proc); exit(1); }
  if ((send = MPI_Send(sbuf, N, MPI_DOUBLE, other_proc,
    98,MPI_COMM_WORLD)) != MPI_SUCCESS) {
    printf("bad send on %d\n",my_proc);  exit(1); }
}
```

C Example #1

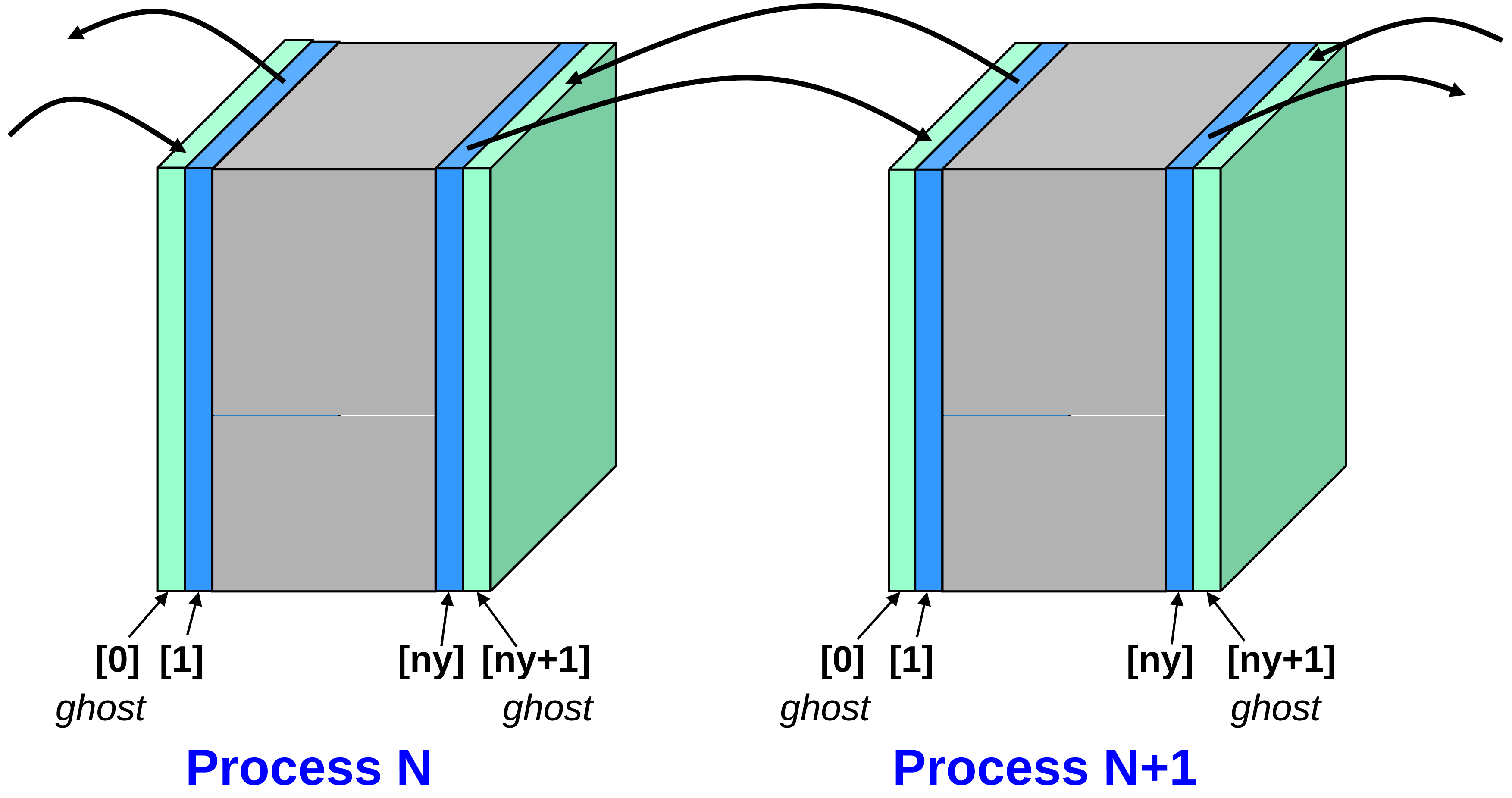
```
/* Terminate MPI */
if ((final = MPI_Finalize()) != MPI_SUCCESS) {
  printf("bad finalize \n");
  exit(1);
}
/* Making sure clean data has been transferred */
for(j = 0; j < N; j++) {
  if (rbuf[j] != sbuf[j]) {
  flag = 0;
  printf("process %d: rbuf[%d]=%f. Should be %f\n",
    my_proc, j, rbuf[j], sbuf[j]);
  }
}
if (flag == 1)
  printf("Test passed on process %d\n", my_proc);
else
  printf("Test failed on process %d\n", my_proc);
  exit(0);
}
```

Ghost Planes

- **When the members of a parallel application share a global virtual array, the shared edges or *ghost planes*, can be visualized this way:**



Ghost Planes



Collective Operations

- Communication that involves a group of processes
  - Barrier synchronization
  - Broadcast
  - Global reduction operations (e.g., sum, min, max, user-defined)
  - Gather/scatter operations and their variants
  - Combined reduction and scatter
  - Scan (prefix) operations

Collective Operations

- May be implemented with MPI point-to-point
  - Implementations can optimize for small transfers (latency), large (bandwidth), or both
  - Generality of some MPI collective operations can limit performance
    - Routines must assume that datatypes are general and discontiguous
    - Time/memory tradeoffs occur (for internal temporary buffers)

Barrier Synchronization

- **The calling process blocks until all group members call the barrier**
  - **Useful for synchronization among processes**

  - **Fortran**

```
INTEGER::COMM,IERROR
CALL MPI_BARRIER (COMM,IERROR)
```

  - **C/C++**

```
int MPI_Barrier (MPI_Comm comm)
```

Broadcast a Message

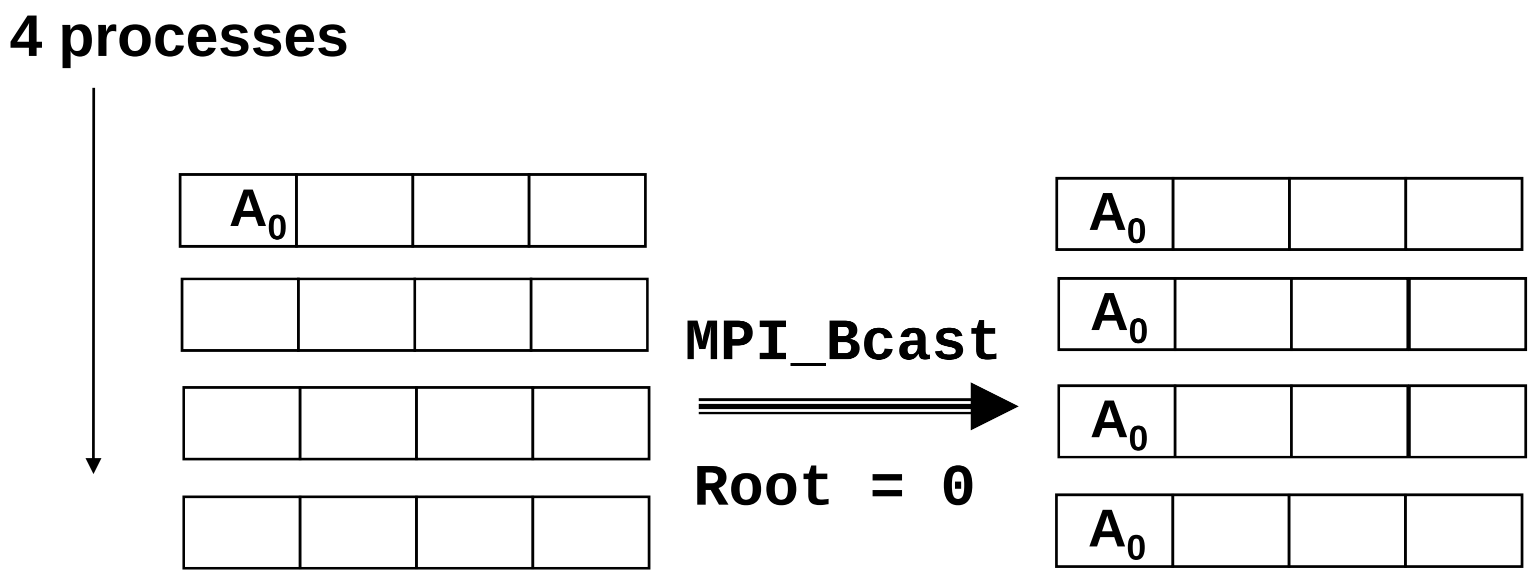
- **Broadcasts a message from one process (with rank ROOT) to all processes of the group**
  - **Fortran**

```
INTEGER::COUNT,DATATYPE,ROOT,COMM,IERROR
 <type>::BUF(*)
 CALL MPI_BCAST(BUF, COUNT, DATATYPE, \
 ROOT, COMM, IERROR)
```

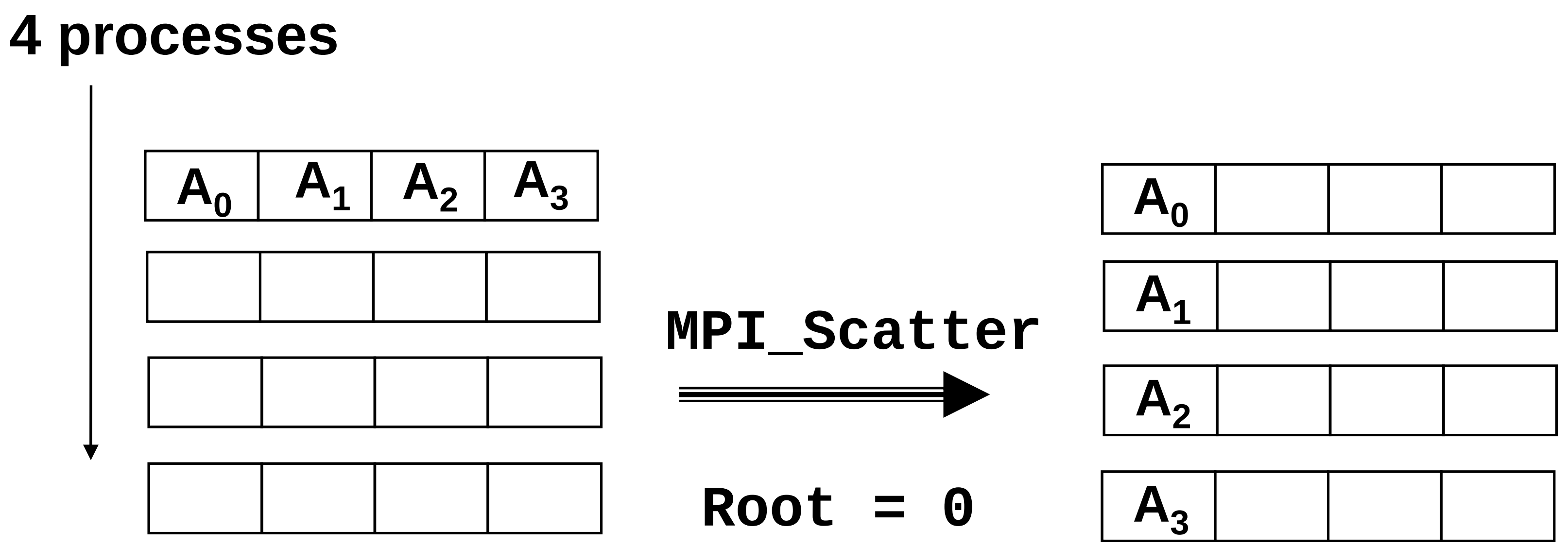
  - **C/C++**

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

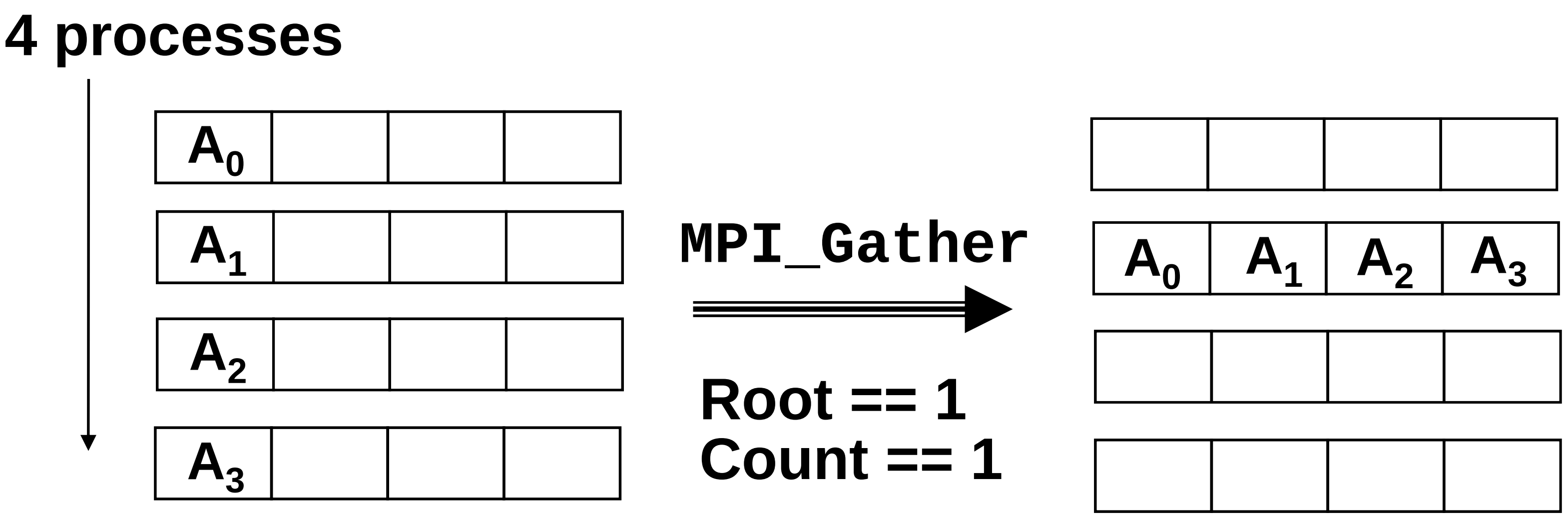
MPI_Bcast

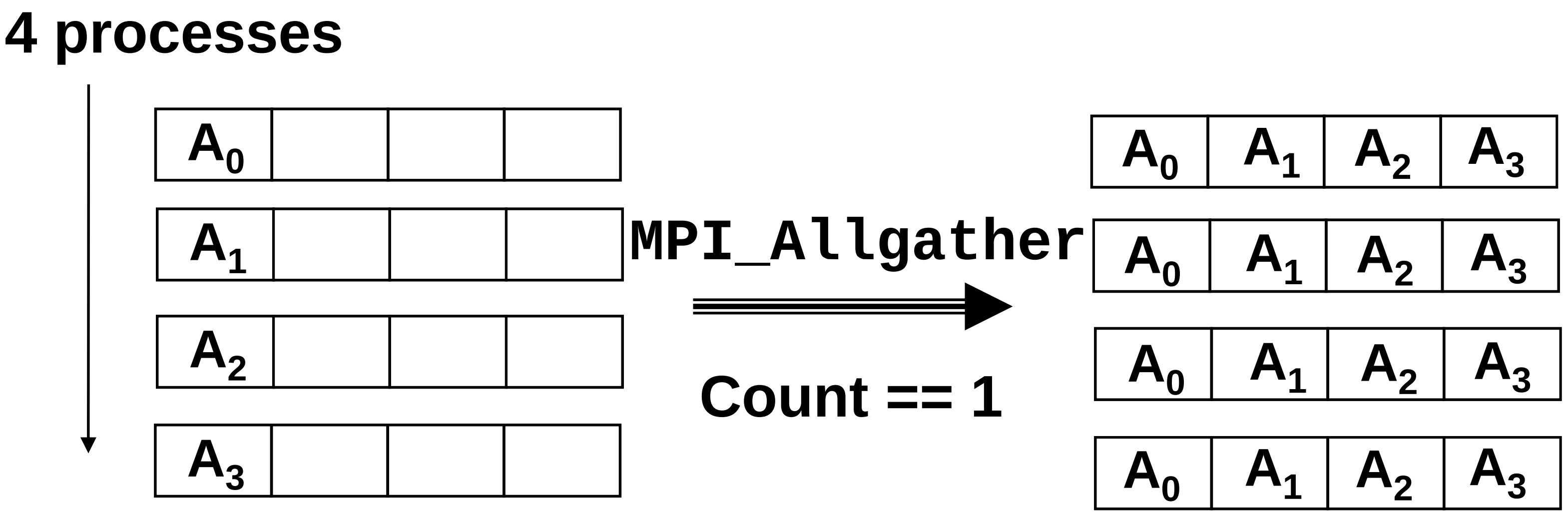


MPI_Scatter



MPI_Gather and MPI_Allgather





Global Reduction Operations

- Perform a global reduce operation
  - Predefined or user-defined
- Fortran

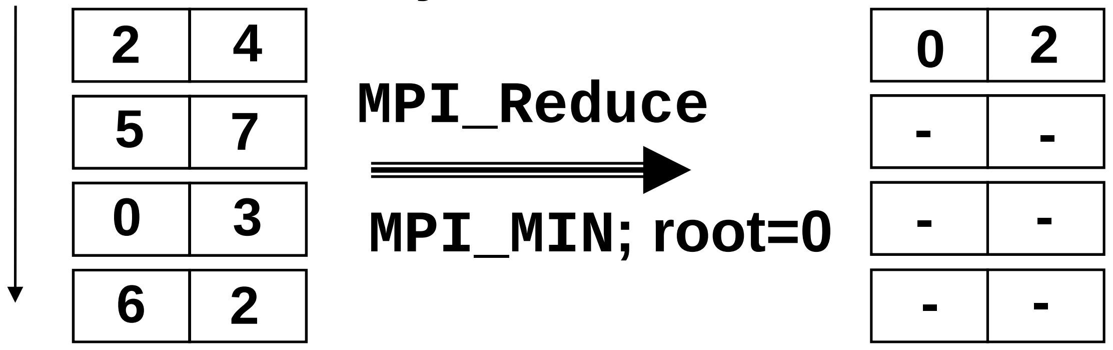
```
INTEGER::COUNT, DATATYPE, OP ROOT, COMM, IERROR
<type>::SENDBUF(*), RECVBUF(*)
CALL MPI_REDUCE (SENDBUF, RECBUF, COUNT, DATATYPE, \
                 OP, ROOT, COMM, IERROR)
```

- C/C++

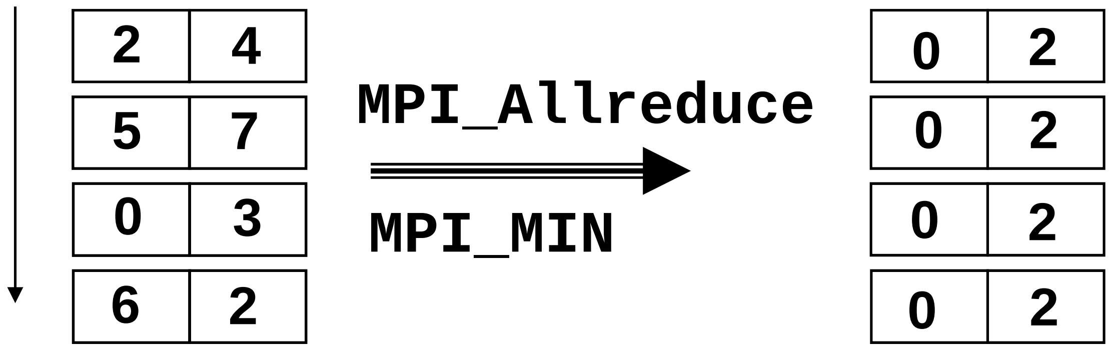
```
int MPI_Reduce (void* sendbuf, void* recvbuf, int count, \
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

MPI_Reduce and MPI_Allreduce

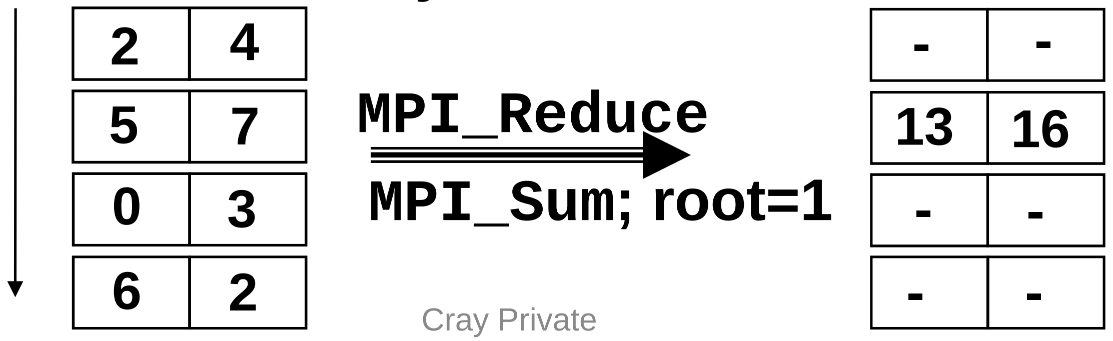
4 processes, array of 2 elements



4 processes, array of 2 elements



4 processes, array of 2 elements



MPI Terminology

- **Nonblocking – the function may return before the operation completes**
  - **The user must verify the resources specified in the call are available before using them again**
- **Blocking – a return from the function indicates that resources specified in the call are available**
  - **Send buffer is empty or receive buffer is full**
- **Local – completion of the function depends only on the local process that is executing**
- **Nonlocal – completion of the operation may require execution of some MPI function on another process**

MPI Communications

- Synchronous - operations complete only after the buffer becomes available for reuse (blocking operations)
- Asynchronous - the process continues while the communication is processing (nonblocking operations)
  - Requires that the program test or wait for operations to complete

MPI Blocking Operations

- `MPI_Send`
  - Starts a blocking send
    - Blocks until the buffer (array) is available for reuse
      - Depending on implementation, may wait for a matching receive
- `MPI_Rsend`
  - In addition, expects a matching receive to be posted
- `MPI_Ssend`
  - In addition, waits for the receive to start receiving data

- `MPI_Recv`
  - Starts a blocking receive

MPI Nonblocking Operations

- `MPI_Isend`
  - Starts a nonblocking send
- `MPI_Irsend`
  - In addition, expects a matching receive to be posted
- `MPI_Issend`
  - In addition, waits for the receive to start receiving data

- `MPI_Irecv`
  - Starts a nonblocking receive

Completion of Nonblocking Operations

- **`MPI_Test`**
  - Nonblocking test for the completion of a nonblocking operation
- **`MPI_Wait`**
  - Blocking test for the completion of a nonblocking operation
- **`MPI_Testall`, `MPI_Waitall`**
  - For all in a collection of requests
- **`MPI_Testany`, `MPI_Waitany`**
- **`MPI_Testsome`, `MPI_Waitsome`**

Testing for Arrived Messages

- `MPI_Probe`
  - Blocking test for an incoming message
- `MPI_Iprobe`
  - Nonblocking test for an incoming message

Fortran Nonblocking Example

- **Add a few variable declarations**

```
INTEGER:: REQUEST
INTEGER, DIMENSION(MPI_STATUS_SIZE)::STATUS
```

- **Change the main loop**

```
 DO I=1, NPES
 CALL MPI_ISEND(TOKEN, 1 , MPI_INTEGER, RIGHT, LFLAG, &
                   MPI_COMM_WORLD, REQUEST, IERROR)
 CALL MPI_RECV(OTHER, 1, MPI_INTEGER, LEFT, LFLAG, &
                   MPI_COMM_WORLD, STATUS, IERROR)
 CALL MPI_WAIT(REQUEST, STATUS, IERROR)
```

C Nonblocking Example

- **Use the nonblocking send to modify the previous C language example:**
  - **Add a few variable declarations**

```
MPI_Status send_status;
MPI_Request request;
```

  - **Change the main loop**

```
for(i = 0; i < size; i++) {
  MPI_Isend(&token, 1, MPI_INT, right, tag,
                MPI_COMM_WORLD, &request);
  MPI_Recv(&other, 1, MPI_INT, left, tag,
                MPI_COMM_WORLD, &recv_status);
  MPI_Wait(&request, &send_status);
```

MPI Buffers

- **Application buffer**
  - **User defined space that holds the data that will be sent or received**
  - **Usually an array of objects**
- **MPI library buffers**
  - **Not visible to the programmer**
  - **Data in the application buffer may need to be copied to or from library buffer space**
    - **Messages that are sent with `MPI_Send()`, `MPI_Isend()`, or `MPI_Ssend()` may be buffered, according to the MPI standard**
    - **The primary purpose of system buffer space is to enable asynchronous communications**

Application Buffers

- **Buffer space defined by the user and passed to MPI to use for buffering**
- **`MPI_Bsend`**
  - **Uses a user-defined buffer**
- **`MPI_Buffer_attach`**
  - **Defines the buffer for all buffered sends**
- **`MPI_Buffer_detach`**
  - **Completes all pending buffered sends and releases the buffer**
- **`MPI_Ibsend`**
  - **Nonblocking version of `MPI_Bsend`**

Persistent Communications

- `MPI_Send_init`
  - Creates a request (like `MPI_Isend`) but does not start it
  - Persistent ready, sync, and buffered sends:
    - `MPI_Rsend_init`, `MPI_Ssend_init`, `MPI_Bsend_init`
- `MPI_Start`
  - Actually begins an operation
- `MPI_Startall`
  - Starts all in a collection
- `MPI_Recv_init`
  - Persistent receive request
- Potential saving:
  - Allocation of `MPI_Request`
  - Validating and storing arguments

MPI-2 MPI_Get and MPI_Put

- One-sided access from/to remote memory
  - Remote Memory Access
  - Similar to `SHMEM`
- Establish a “window” to the remote memory with `MPI_Create_window`
  - Call `MPI_Win_free` to release the window
  - Window can be to any memory, without “symmetric” restrictions
- Use `MPI Win fence` to synchronize all communication from/to a window

Basic and Derived Datatypes

- **The type of data that a function sends or receives is specified as a datatype**
  - **MPI datatypes are either basic or derived**
    - **Basic datatypes correspond to the datatypes in the host programming language - integers, floating-point numbers, and so forth**
    - **Derived datatypes are created by a datatype constructor in MPI**
      - **Derived datatypes consist of multiple basic datatypes whether contiguous or discontiguous (sequential or random)**

Basic MPI Datatypes in Fortran

`MPI_INTEGER`	`INTEGER`
`MPI_LOGICAL`	`LOGICAL`
`MPI_REAL`	`REAL`
`MPI_DOUBLE_PRECISION`	`DOUBLE PRECISION`
`MPI_COMPLEX`	`COMPLEX`
`MPI_DOUBLE_COMPLEX`	`COMPLEX*16 (or COMPLEX*32)`
`MPI_INTEGER8`	`INTEGER*8`
`MPI_REAL8`	`REAL*8`

Basic MPI Datatypes in C

`MPI_CHAR`	`char`
`MPI_BYTE`	`unsigned char (see the standard)`
`MPI_SHORT`	`short`
`MPI_INT`	`int`
`MPI_LONG`	`long`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`

Derived Datatypes

- Any datatype created by a datatype constructor can be used as input to another datatype constructor
  - Therefore any discontiguous data layout can be represented in terms of a derived datatype
  - MPI has the following kinds of datatype constructors :
    - `contiguous`
    - `vector/hvector`
    - `indexed/hindexed/indexed_block`
    - `struct`
    - `subarray`
    - `darray`

Processor Grids

- **Some tools enable the programmer to view the processor grid as a Cartesian plane and use (X , Y) coordinates or column or row operations**
  - **MPI**

    `MPI_CARTE_CREATE` defines the size and shape of the processor grid

    `MPI_CART_COORDS` returns the coordinates of a processor

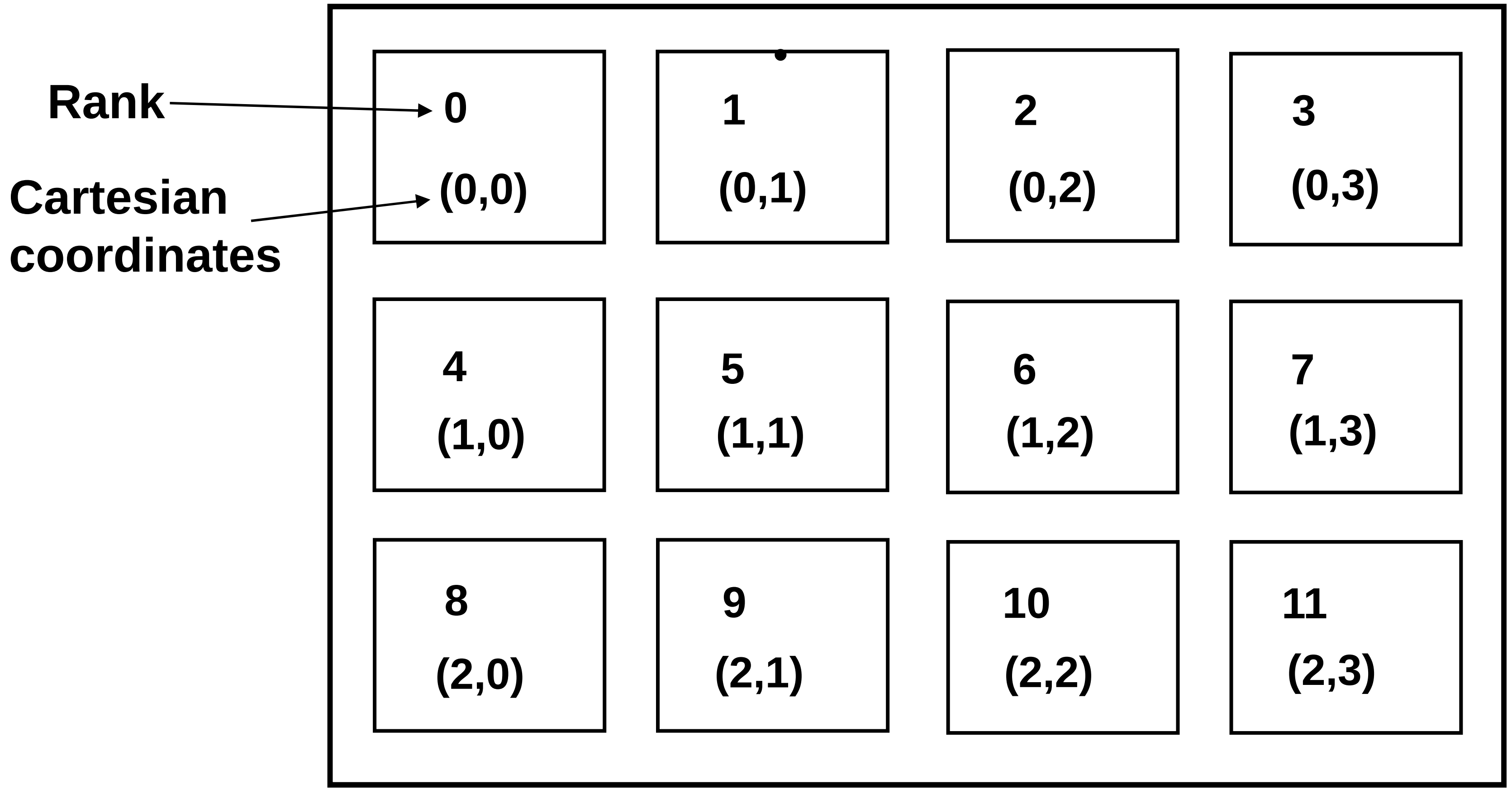
    `MPI_CART_SHIFT` returns the rank of the neighbors in any given dimension and distance
  - **BLACS**

    `BLACS_GRIDINIT` enables the user to define the size and shape of the processor grid

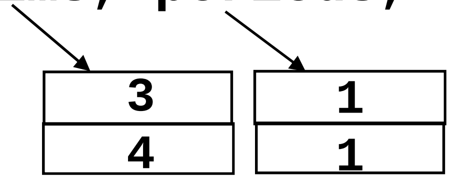
    `BLACS_GRIDINFO` returns the calling processor’s grid coordinates

    - **Matrices are sent and received by grid coordinates**

Processor Grids



```
MPI_Carte_create (old, 2, dims, periods, 0, new)
```

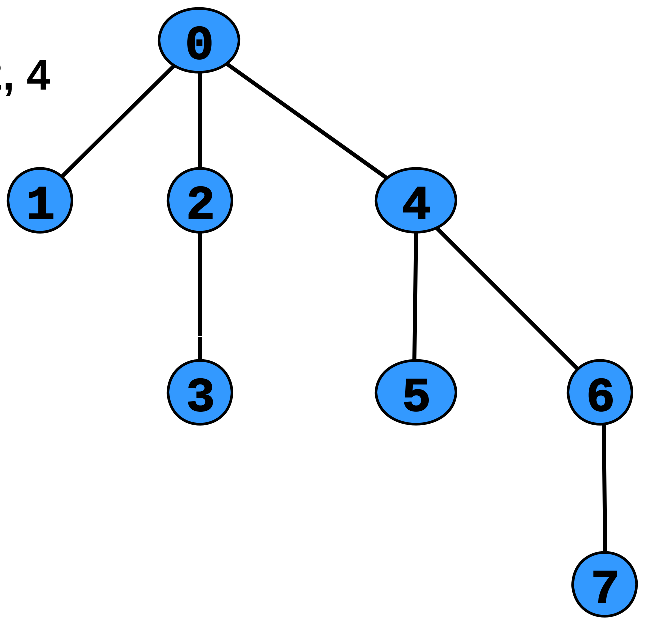


General Graph Topology

- `MPI_Graph_create (comm_old, 8, index, edges, 0, comm_graph)`

Node	
[0]	3
[1]	4
[2]	6
[3]	7
[4]	10
[5]	11
[6]	13
[7]	14

1
2 Node 0 connects to 1, 2, 4
4
0 Node 1 connects to 0
0
3
2
0
5
6
4
4
7 Node 6 connects to 4, 7
6



Heterogeneous Applications

- A heterogeneous application consists of multiple binaries that run as one group and share communicator `MPI_COMM_WORLD`
  - Cray XT/XE Systems do not support any form of MPI process creation (fork(), exec(), popen(), system()) so `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple` generate runtime errors
- Starting a heterogeneous application:
  - prog1 and prog2 start up with `MPI_COMM_WORLD` as an intracommunicator between the two programs

Aggregation to Reduce Latency

- Very small messages impose a large latency overhead per byte of information
  - Latency overhead increases very little as the size of the message increases
- Collect many small messages into a single large message
  - Latency (usually) outweighs packing cost
- Save several intermediate local computational results for a larger block exchange at the end
- Avoid unnecessary buffering
  - Creates extra copying of large amounts of data

Aggregation with Derived Data Types

- **Use derived data types to describe a regular pattern of data elements that can be moved at one time rather than word-by-word**
  - **Not optimized on Cray XT/XE systems**
  - **`MPI_Type_vector`**
    - **Equally spaced instances of another datatype**
    - **MPICH optimizes**
  - **`MPI_Type_struct` with `MPI_UB` entry**
    - **Irregularly spaced instances of other data types**
    - **MPI does the pack/unpack of a single instance of the structure**
    - **`MPI_UB` is the type's upper bound; it is set to describe the "extent" size of the structure**

Aggregation in Collective Operations

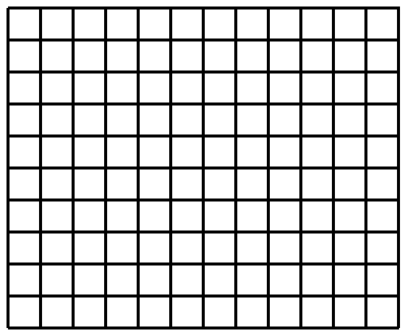
- Use collective routines to broadcast or gather many copies
- Use the collective functions instead of the equivalent point-to-point functions
  - gather, scatter, broadcast, reduce, scan
- Combine collective operations
  - Much cheaper to do one 2-element `allreduce` than two 1-element `allreduces`

Issues in Choosing a Decomposition

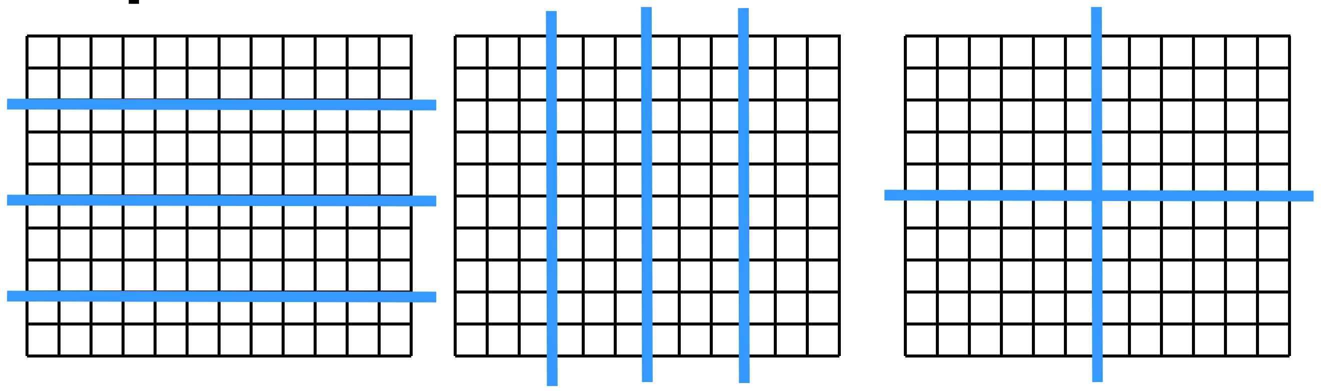
- One, versus two, versus three dimensions
- Minimize surface-to-volume ratio
  - Horizontal edges of 10x1000 array: 10 elements
  - Horizontal edges of 1000x10 array: 1000 elements
- More complex decompositions (e.g., hexagons in 2D) are possible, but *usually* not worthwhile
- Relatively small problems may not be worthwhile to parallelize; latency may dominate

Decomposition of Regular Meshes

- **A regular mesh**



Decomposition in coordinate directions



MPI Support for Regular Decompositions

- Using topology routines

  `MPI_Cart_create`

  `MPI_Cart_shift` / `MPI_Cart_coords`

  – Why you should use the topology routines
    - Simple to use
    - Allow MPI implementation to provide low expected contention layout of processes (if implementation is aware of nearest neighbors; the Cray XT/XE implementation is not).

Performance Issues of Decompositions

- Use of application's scaling behavior to identify problems
  - Fixed execution time suggests a poor decomposition
    - Noncontiguous data may be the cause
  - Actual choice of decomposition is complex
    - Spectral bisection
    - Coordinate based
    - Graph cutting

Load Balancing

- Small amounts of work imbalance lead to large losses in performance
  - Is load balancing central to the algorithm or part of performance tuning?
    - Central to the algorithm: in master/slave models, multilevel work masters (functional parallelism)
    - Part of performance tuning: load is balanced by decomposition tuning

Identifying Load Imbalances

- Identifying (distinguishing from latency/synchronization overhead)
  - Poor load balance focuses attention on collective operations because the implicit synchronization of the collective operation “equalizes” the time for each process
    - Can generate the appearance of a good load balance if not timed correctly

Load Balancing Functional Parallelism

- Post receives before sends; otherwise, you may have to handle unexpected messages.
- Multilevel masters
  - Work stealing
- Using `MPI_Ssend` (or `MPI_Issend`) to manage message flow
  - Avoids overwhelming buffer operations
- Fairness in message-passing
  - Ensure that no slave is starved for the attention of its master

Implementing Fairness

- Use **`MPI_Waitsome`** to poll for replies
  - **Master's code is:**

```
for (i=0;i<n;i++)
     MPI_Irecv( …, &r[i]);
while (not done) {
  MPI_Waitsome( n, r, &nready, i_ready,
                statuses );
  … Process r[i_ready] and repost
}
```

  - **Can double buffer requests/replies with `MPI_Issend` to control buffer use and allow slaves to overlap synchronization delays**

Load Balance by Tuning Decomposition

- **Static data decomposition**
  - **Different boundary behavior means you cannot simply count “mesh points” that belong to each node**
  - **Rule of thumb for a matrix: equalize the number of elements without breaking rows (this is a good compromise between perfection and workability)**

Changing the Algorithm

- Some algorithms are simply not good candidates for parallelization
  - If an algorithm is an approximation, another approximation may be a better choice (a different physical model)
  - If an algorithm is part of an iterative method, another iteration may be better (a different numerical model)

Trade Communication for Computation

- Example: Solving a small linear system when all processes need the results
  - Parallel solution is latency dominated – not worthwhile for small work; single solution using gather/bcast leaves processes idle
  - All-solve solution uses single gather (but has duplicate computational work)
- – For slowly converging algorithms, another form of blocking: take a number of steps and then check convergence (rather than checking at each iteration)
- – You can trade bandwidth/computation for latency (unroll a compute loop once, do a single send of more data, do duplicate computation)

Changing the Algorithm: Loop Unrolling

- **Classic algorithm change technique for improving performance:**

```
do I=1,10
  call f(I)
 exchange data for step I
```

- **Changed to:**

```
do I=1,10,2
 call f(I)
 call f(I+1)
 exchange data for steps I and I+1
```

Synchronization

- The wait for other processes is one of the largest consumers of wall clock time in very asynchronous parallel programs
  - Use the `MPI_barrier` function only when necessary
    - Unnecessary `MPI_barrier` functions inserted as “insurance” can decrease performance substantially
  - Use nonblocking sends/receives where useful local work can be performed while polling for message completion
    - Finding enough local-only work to hide most messaging latency can be difficult

Overlap Communications/Computation

- **Use nonblocking operations:**
- **`MPI_Isend`, `MPI_Irecv`, `MPI_Waitall`**

```
void ExchangeStart(Mesh *mesh)
{
   /* send up, then receive from below */
   MPI_Irecv(xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm,
        &mesh->rq[0]);
   MPI_Irecv(xlocal + maxm * (lrow+1), maxm, MPI_DOUBLE, up_nbr, 1,
        ring_comm, &mesh->rq[1]);
   MPI_Isend(xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
        ring_comm, &mesh->rq[2]);
   MPI_Isend(xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm,
        &mesh->rq[3]);
}
void ExchangeEnd(Mesh *mesh)
{
   MPI_Status statuses[4];
   MPI_Waitall (4, mesh->rq, statuses);
}
```

Start Receives Before Sends

- **`MPI_Irecv`, `MPI_Isend`, `MPI_Waitall`**

```
MPI_Status statuses[4];
MPI_Comm   ring_comm;
MPI_Request r[4];
/* send up, then receive from below */
  MPI_Irecv(xlocal, maxm, MPI_DOUBLE, down_nbr, 0,
            ring_comm, &r[1]);
  MPI_Irecv(xlocal + maxm * (lrow+1), maxm, MPI_DOUBLE,
            up_nbr, 1, ring_comm, &r[3]);
  MPI_Isend(xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
            ring_comm, &r[0]);
/* send down, then receive from above */
  MPI_Isend(xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1,
            ring_comm,&r[2]);
MPI_Waitall (4, r, statuses);
}
```

Start Receives Before Sends

```
void ExchangeInit(Mesh *mesh){
  MPI_Irecv(xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm,
            &mesh->rq[0]);
  MPI_Irecv(xlocal + maxm * (lrow+1), maxm, MPI_DOUBLE,
            up_nbr, 1, ring_comm, &mesh->rq[1]);
}
void Exchange(Mesh *mesh){
  MPI_Status statuses[4];
    /* send up and down, then receive */
  MPI_Send(xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,
           ring_comm);
  MPI_Send(xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1,
           ring_comm);
  MPI_Waitall (2, mesh->rq, statuses);
}
void ExchangeEnd(Mesh *mesh){
  MPI_Cancel(&mesh->rq[0]);
  MPI_Cancel(&mesh->rq[1]);
}
```

Use of MPI_Ssend

```
void Exchange(Mesh *mesh)
{
  MPI_Status status;
/* send up, then from below */
  MPI_Irecv(xlocal, maxm, MPI_DOUBLE, down_nbr, 0,
          ring_comm, &rq);
  MPI_Ssend(xlocal + maxm*lrow, maxm, MPI_DOUBLE, up_nbr, 0,
          ring_comm);
  MPI_Wait (&rq, &status);
/* send down, then receive from above */
  MPI_Irecv(xlocal + maxm * (lrow+1), maxm, MPI_DOUBLE,
          up_nbr, 0, ring_comm);
  MPI_Ssend(xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1,
          ring_comm);
  MPI_Wait (&rq, &status);
}
```

Timing With `MPI_Wtime`

- Using `MPI_WTIME`
  - You can compute the elapsed time between two points in an MPI program by using `MPI_Wtime`
  - `MPI_Wtime` granularity is 0.000001 sec. (see `MPI_Wtick`). You cannot time any period that is smaller than a microsecond with it.
  - The clock in each node is independent of the clocks in other nodes
  - `MPI_WTIME_IS_GLOBAL` has value=1 if `MPI_WTIME` is globally synchronized
    - Default is 0

MPI-IO

- **A key feature of MPI-IO is its ability to access noncontiguous data with a single I/O function call**
  - **Using MPI's basic or derived datatypes to describe:**
    - **The data layout in the user's buffer in memory**
      - **This can be used, for example, when the user's buffer represents a local array with a “ghost area” that will not be written to the file.**
    - **The data layout in a file**
      - **This can be used to describe the portion of a file the process must access (also called a file view).**
      - **Allowing any general, noncontiguous access pattern to be compactly represented.**
  - **NERSC support staff recommends using higher level libraries such as `HDF5` or `pnetCDF` rather than MPI-IO**

Parallel HDF and NetCDF

- Higher-level, open source APIs are available:
  - Parallel HDF – Hierarchical Data Format
    - From the National Center for Supercomputing Applications (NCSA)
      - http://hdf.ncsa.uiuc.edu/Parallel_HDF/
  - Parallel NetCDF - Network Common Data Form
    - From the Unidata Program Center in Boulder, CO
      - my.unidata.ucar.edu/content/software/netcdf/index.html
      - www-unix.mcs.anl.gov/parallel-netcdf/sc03_present.pdf (relationship to MPI)