Advanced Topics in MPI

William Gropp and Rajeev Thakur

Mathematics and Computer Science Division

A U.S. Department of Energy laboratory managed by The University of Chicago

Outline

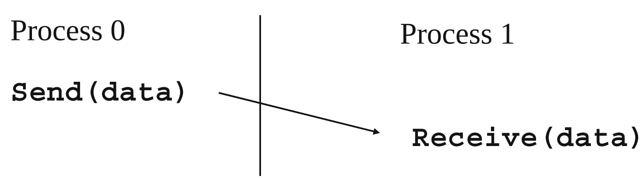
- MPI review
- MPI performance issues and tuning
- Advanced topics (datatypes etc)
- Parallel I/O and MPI-IO
- High-level I/O libraries (PnetCDF, HDF-5)

The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's.

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

Datatypes and Tags

- The data in a message to sent or received is described by a triple (address, count, datatype)
- The datatype describes the type of data to be sent (INT, FLOAT, or more complex noncontiguous types)
- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive.

MPI Basic (Blocking) Send

MPI_SEND (buf, count, datatype, dest, tag, comm)

- The message buffer is described by (buf**, count, datatype**).
- The target process is specified by **dest**, which is the rank of the target process in the communicator specified by **comm**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

MPI_RECV(buf, count, datatype, source, tag, comm, status)

- Waits until a matching (on **source** and **tag**) message is received from the system, and the buffer can be used.
- **source** is rank in communicator specified by **comm**, or **MPI_ANY_SOURCE**.
- **status** contains further information
- receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error.

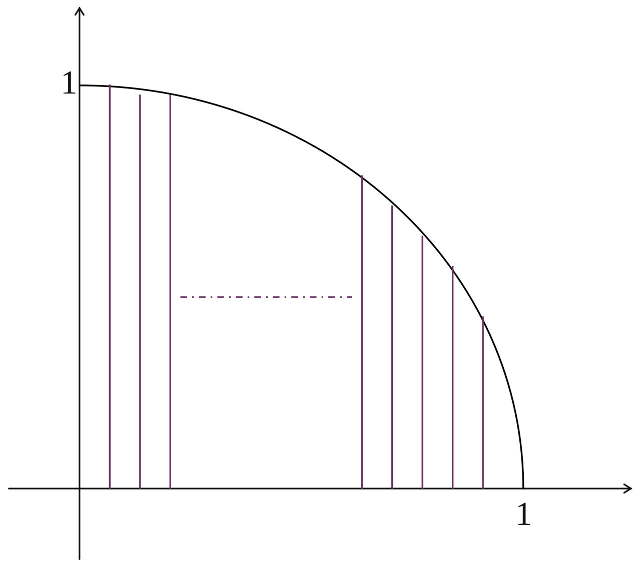
MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECV`**
- But, for performance, you need to use other features

Collective Communication

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

Example: Calculating Pi



Example: PI in C

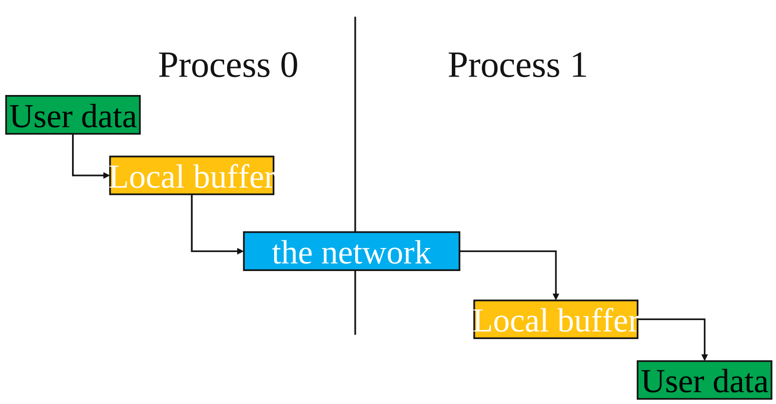
```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  while (!done)  {
    if (myid == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;
```

Example: PI in C

```
 h   = 1.0 / (double) n;
  sum = 0.0;
  for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  mypi = h * sum;
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);
  if (myid == 0)
    printf("pi is approximately %.16f, Error is %.16f\n",
           pi, fabs(pi - PI25DT));
}
MPI_Finalize();
return 0;
}
```

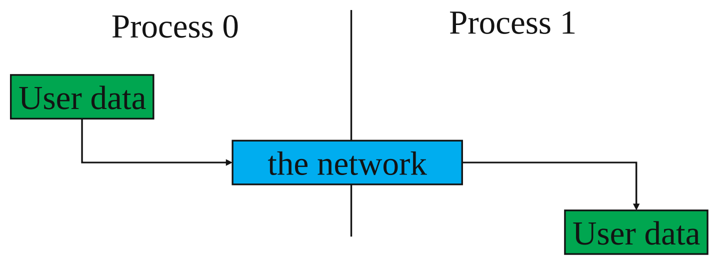

Buffering

- When you send data, where does it go? One possibility is:



Avoiding Buffering

- It is better to avoid copies:



This requires that **MPI_Send** wait on delivery, or that **MPI_Send** return before transfer is complete, and we wait later.

Blocking and Non-blocking Communication

- So far we have been using *blocking* communication:
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
`Send(1)`	**`Send(0)`**
`Recv(1)`	**`Recv(0)`**

This is called “unsafe” because it depends on the availability of system buffers

Solutions to the “safety” Problem

- Order the operations more carefully
- Supply receive buffer at same time as send (**`MPI_Sendrecv`**)
- Supply own buffer space (**`MPI_Bsend`**)
- Use non-blocking operations
  - safe
  - not necessarily asynchronous
  - not necessarily concurrent
  - not necessarily faster

MPI’s Non-blocking Operations

- Non-blocking operations return (immediately) “request handles” that can be tested and waited on.

```
MPI_Isend(start, count, datatype,
          dest, tag, comm, request)
MPI_Irecv(start, count, datatype,
          dest, tag, comm, request)
MPI_Wait(&request, &status)
```

- One can also test without waiting:

```
MPI_Test(&request, &flag, status)
```


Multiple Completions

- It is sometimes desirable to wait on multiple requests:

```
MPI_Waitall(count, array_of_requests,
            array_of_statuses)
MPI_Waitany(count, array_of_requests,
            &index, &status)
MPI_Waitsome(count, array_of_requests,
             array_of indices, array_of_statuses)
```

- There are corresponding versions of **Test** for each of these.

Communication Modes

- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**MPI_Ssend**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  - Buffered mode (**MPI_Bsend**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
  - Ready mode (**MPI_Rsend**): user guarantees that a matching receive has been posted.
    - *Allows access to fast protocols*
    - *undefined behavior if matching receive not posted*
- Non-blocking versions (**MPI_Issend**, etc.)
- **MPI_Recv** receives messages sent in any mode.

Timing MPI Programs

- The elapsed (wall-clock) time between two points in an MPI program can be computed using **MPI_Wtime**:

```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf( "time is %d\n", t2 - t1 );
```

- The value returned by a single call to **MPI_Wtime** has little value.
- Times in general are local, but an implementation might offer synchronized times. See attribute **MPI_WTIME_IS_GLOBAL**.

Measuring Performance

- Using MPI_Wtime
  - timers are *not* continuous — MPI_Wtick
- MPI_Wtime is local unless the MPI_WTIME_IS_GLOBAL attribute is true
- MPI Profiling interface provides a way to easily instrument the MPI calls in an application
- Performance measurement tools for MPI

Sample Timing Harness

- Average times, make several trials

```
for (k<nloop) {
     t1 = MPI_Wtime();
     for (I<maxloop) {
         <operation to be timed>
     }
     time = MPI_Wtime() - t1;
     if (time < tfinal) tfinal = time;
}
```

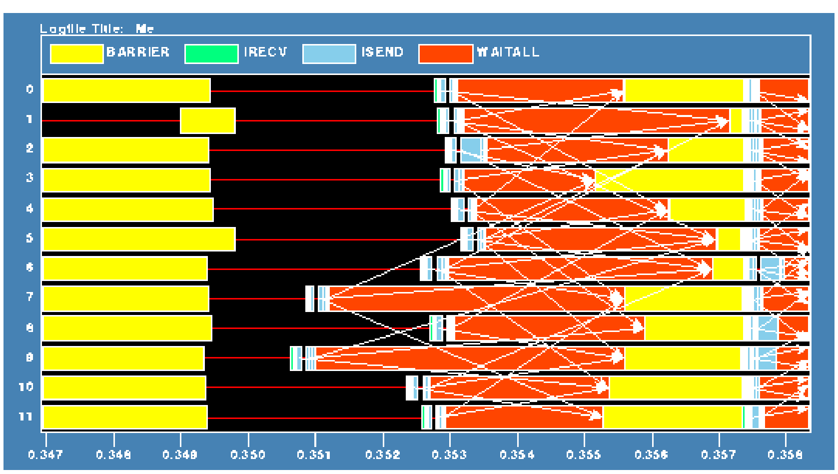
- Use MPI_Wtick to discover clock resolution
- Use getrusage to get other effects (e.g., context switches, paging)

Pitfalls in timing

- Time too short:
  ```
  t = MPI_Wtime();
  MPI_Send(…);
  time = MPI_Wtime() - t;
  ```
- Underestimates by MPI_Wtick, over by cost of calling MPI_Wtime
- “Correcting” MPI_Wtime by subtracting average of MPI_Wtime calls overestimates MPI_Wtime
- Code not paged in (always run at least twice)
- Minimums not what users see
- Tests with 2 processors may not be representative
  - T3D had processors in pairs, pingpong gave 130 MB/sec for 2 but 75 MB/sec for 4 (for MPI_Ssend)

Example of Paging Problem

- Black area is *identical* setup computation



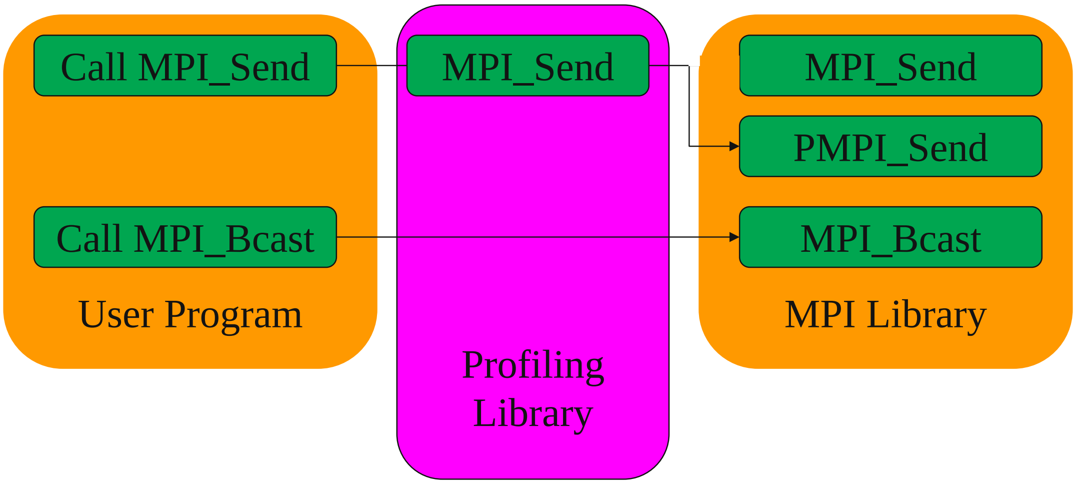
Exercise: Timing MPI Operations

- Estimate the latency and bandwidth for some MPI operation (e.g., Send/Recv, Bcast, Ssend/Irecv-Wait)
  - Make sure all processes are ready before starting the test
  - How repeatable are your measurements?
  - How does the performance compare to the performance of other operations (e.g., memcpy, floating multiply)?

MPI's Profiling Interface

- Every MPI function also exists in the library under the name PMPI_
- PMPI allows selective replacement of MPI routines at link time (no need to recompile)
- This feature can be used by profiling tools to instrument MPI calls

Profiling Interface



Using the Profiling Interface

```
static int nsend = 0;

int MPI_Send( void *start, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm )
{
    nsend++;
    return PMPI_send(start, count, datatype,
                     dest, tag, comm);
}
```

MPI Performance Tools on Jazz

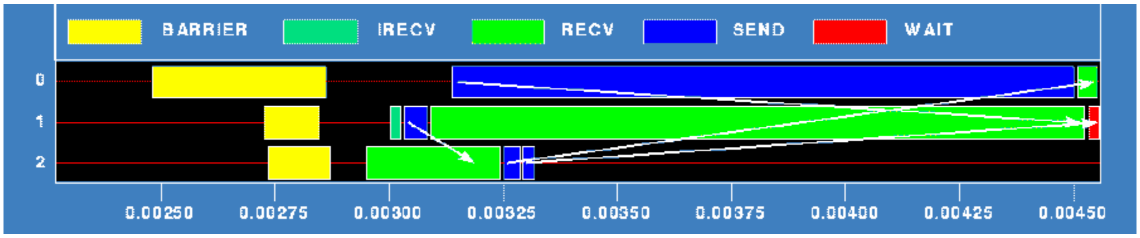
- These softenv keys will give you access to MPI performance tools on Jazz:
  - +fpmpi-2.0 MPI profiling library that provides summary information about the use of MPI within an application
  - +jumpshot Jumpshot is a performance visualization tool that gives you a detailed picture of the MPI operations within an application.
  - +jumpshot-1.2 jumpshot -- jumpshot-1.2
  - +jumpshot-1.2b jumpshot -- jumpshot-1.2b
  - +jumpshot-4 jumpshot -- jumpshot-4
- These require that the application be relinked but not recompiled. They are developed in MCS, so you can expect quick feedback on problems and feature requests

Synchronization Delays

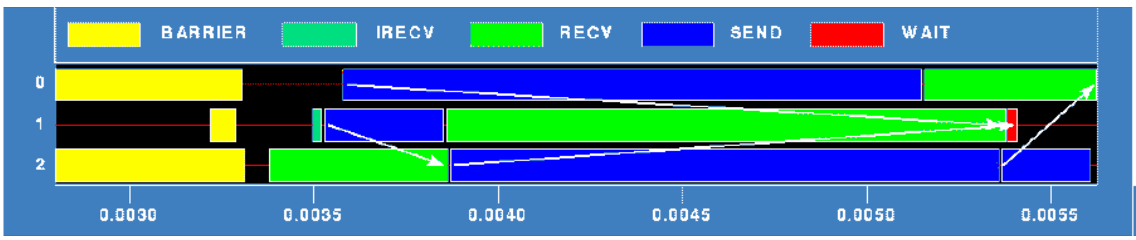
- Message passing is a cooperative method — if the partner doesn't react quickly, a delay results
- There is a performance tradeoff caused by reacting quickly — it requires devoting resources to checking for things to do

Observing Synchronization Delays

- Three processors sending data, with one sending a short message and another sending a long message to the same process:



Eager



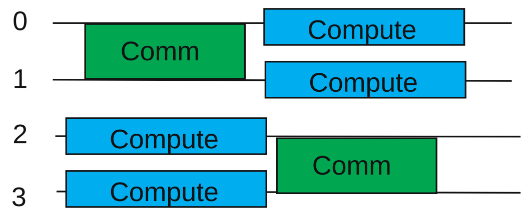
Rendezvous

Contention

- Point-to-point analysis ignores fact that communication links (usually) are shared
- Easiest model is to equally share bandwidth (if K can shared at one time, give each 1/K of the bandwidth).
- “Topology doesn’t matter anymore” is *not* true, but there is less you can do about it (just like cache memory)
- MPI has processor topology routines, though these are only useful on some MPI implementation. It is good to use them for best portability (e.g., they won’t make any difference on Jazz but can be inportant on BG/L).

Scheduling for Contention

- Many programs alternate between communication and computation phases
- Contention can reduce effective bandwidth
- Consider restructuring program so that some nodes communicate while others compute:



Effect of Contention

- IBM SP2 has a multistage switch. This test shows the point-to-point bandwidth with half the nodes sending and half receiving

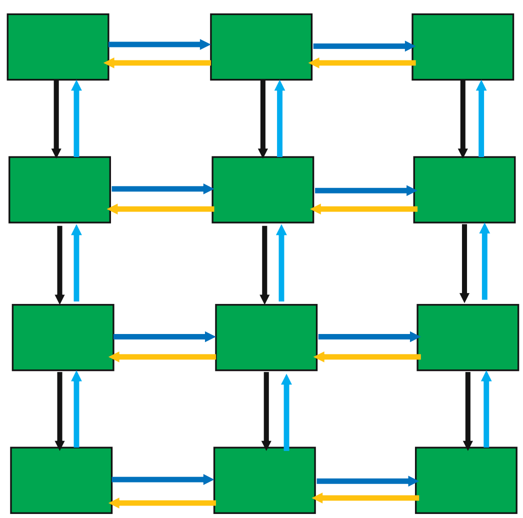
Processors	Bandwidth (MB/sec)
2	34
4	34
8	34
16	31
32	25
64	22

Unexpected Hot Spots

- Even simple operations can give surprising performance behavior.
- Examples arise even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - Blocking operations may cause unavoidable stalls

Mesh Exchange

- Exchange data on a mesh



Sample Code

- Do i=1,n_neighbors
  Call MPI_Send(edge(1,i), len, MPI_REAL, nbr(i), tag,comm, ierr)
  Enddo

  Do i=1,n_neighbors
  Call MPI_Recv(edge(1,i), len, MPI_REAL, nbr(i), tag, comm, status, ierr)
  Enddo

Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of
  if (has down nbr) then
  Call MPI_Send(… down …)
  endif
  if (has up nbr) then
  Call MPI_Recv(… up …)
  endif
  …
  sequentializes (all except the bottom process blocks)

Sequentialization

Start	Start	Start	Start	Start	Start	Send	Recv
Send	Send	Send	Send	Send	Send		
					Send	Recv	
				Send	Recv		
			Send	Recv			
		Send	Recv				
	Send	Recv					
Send	Recv						

Fix 1: Use Irecv

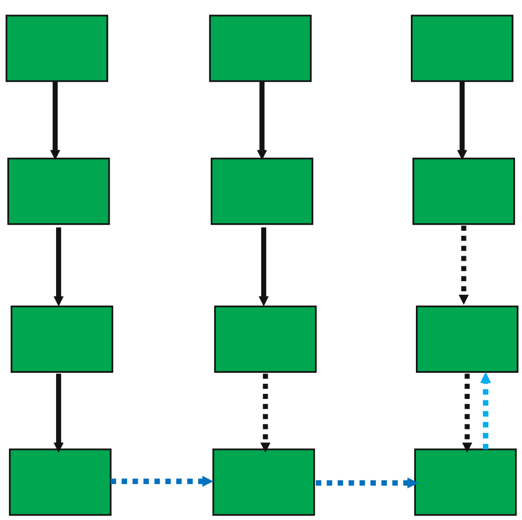
- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(i), ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Send(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, ierr)
  Enddo
  Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
- Does not perform well in practice. Why?

Understanding the Behavior: Timing Model

- Sends interleave
- Sends block (data larger than buffering will allow)
- Sends control timing
- Receives do not interfere with Sends
- Exchange can be done in 4 steps (down, right, up, left)

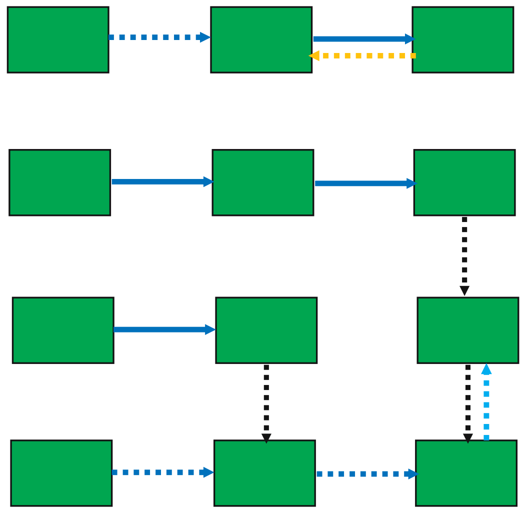
Mesh Exchange - Step 1

- Exchange data on a mesh



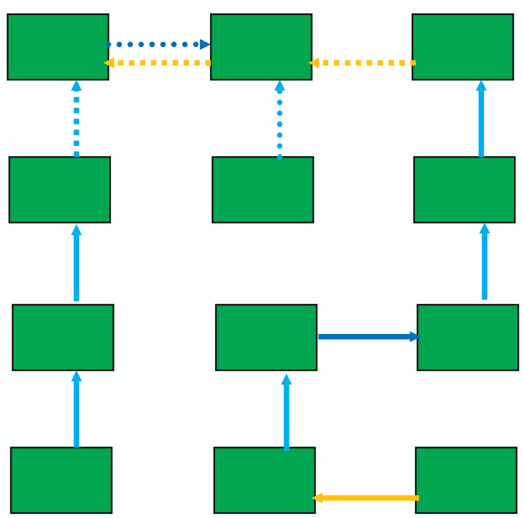
Mesh Exchange - Step 2

- Exchange data on a mesh



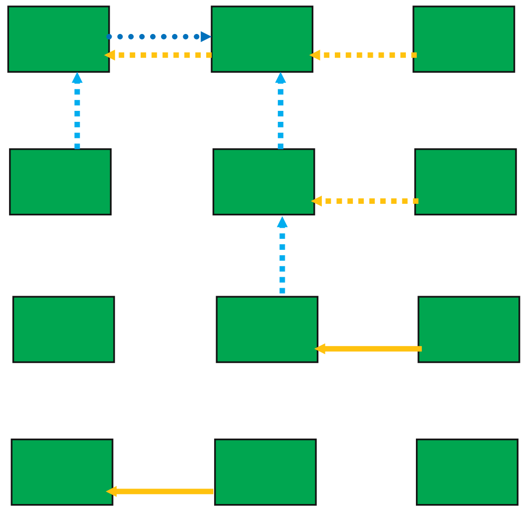
Mesh Exchange - Step 3

- Exchange data on a mesh



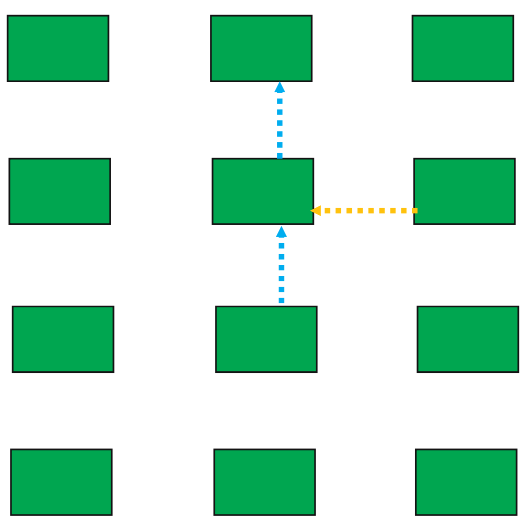
Mesh Exchange - Step 4

- Exchange data on a mesh



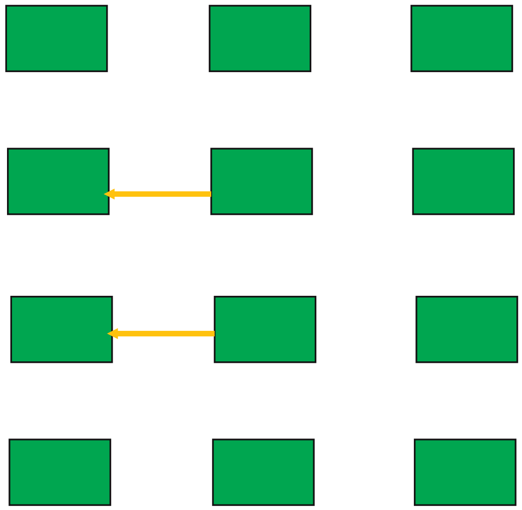
Mesh Exchange - Step 5

- Exchange data on a mesh

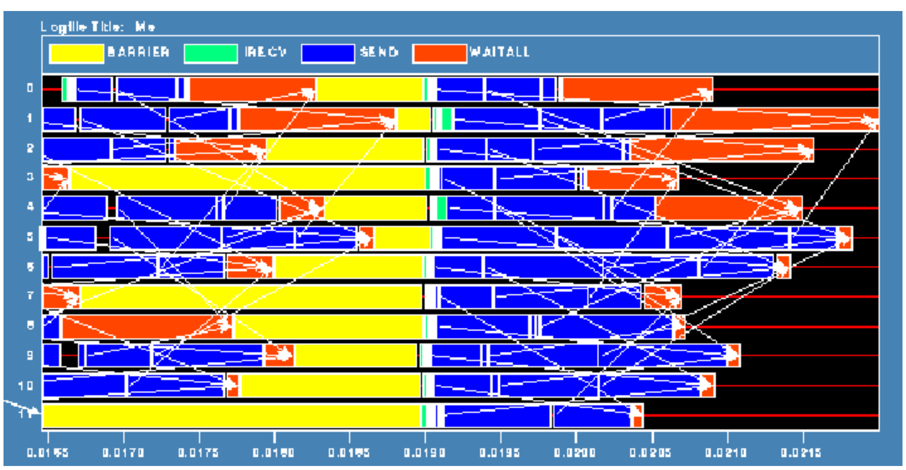


Mesh Exchange - Step 6

- Exchange data on a mesh

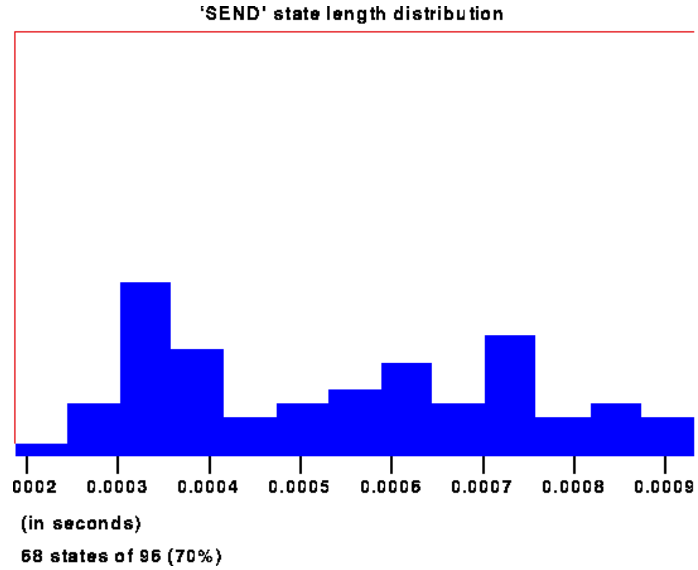


Timeline from IBM SP



- Note that process 1 finishes last, as predicted

Distribution of Sends



Why Six Steps?

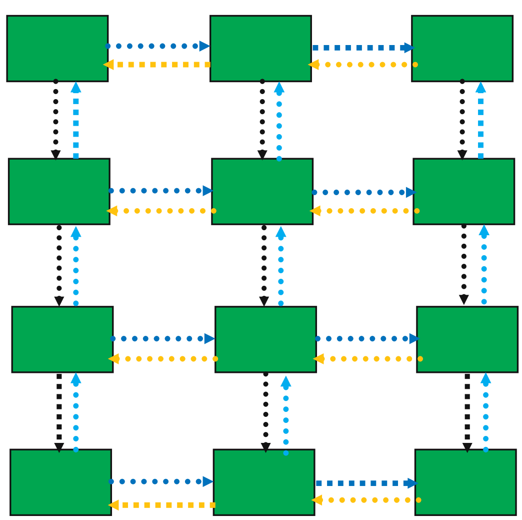
- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory

Fix 2: Use Isend and Irecv

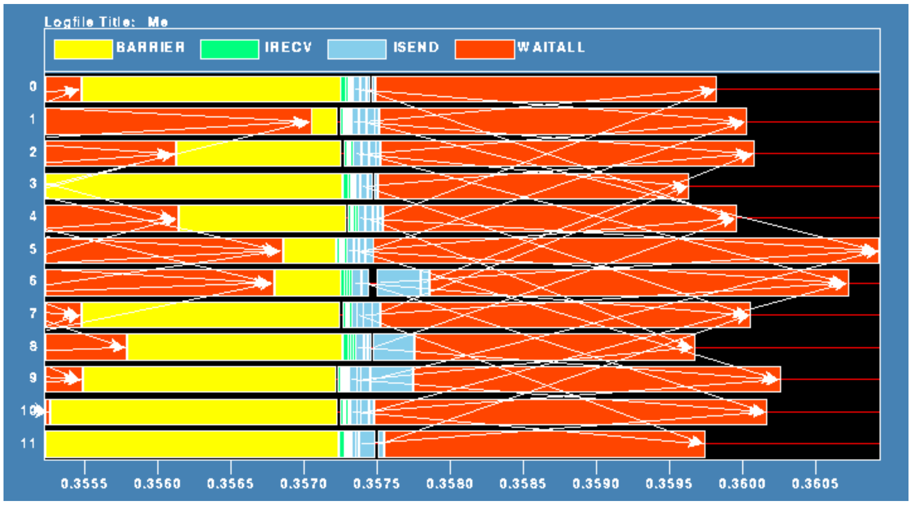
- Do i=1,n_neighbors
  Call MPI_Irecv(inedge(1,i),len,MPI_REAL,nbr(i),tag,&
  comm, requests(i),ierr)
  Enddo
  Do i=1,n_neighbors
  Call MPI_Isend(edge(1,i), len, MPI_REAL, nbr(i), tag,&
  comm, requests(n_neighbors+i), ierr)
  Enddo
  Call MPI_Waitall(2*n_neighbors, requests, statuses, ierr)

Mesh Exchange - Steps 1-4

- Four interleaved steps



Timeline from IBM SP



Note processes 5 and 6 are the only interior processors; these perform more communication than the other processors

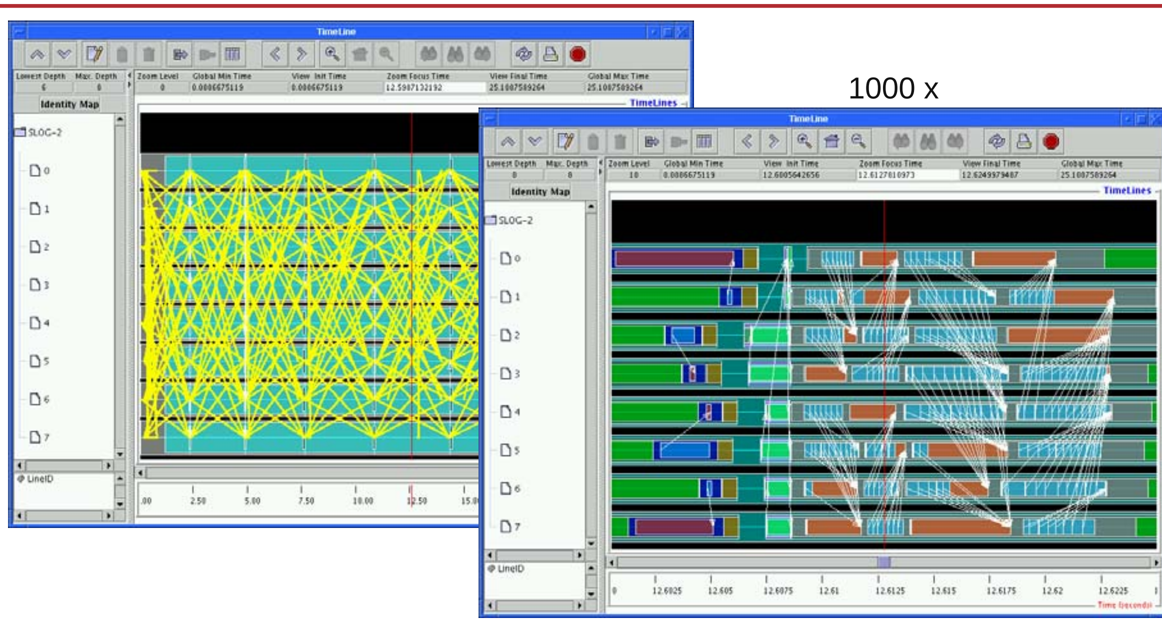
Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization
- In many cases, there is more synchronization than required
- Use nonblocking operations and MPI_Waitall to defer synchronization

Logging and Visualization Tools

- Jumpshot and MPE tools
  - http://www.mcs.anl.gov/perfvis
- TAU
  - http://www.cs.uoregon.edu/research/tau/home.php
- Intel Trace Analyzer and Collector (formerly Pallas Vampir)
  - http://www.intel.com/cd/software/products/asmo-na/eng/cluster/tanalyzer/index.htm
- Paradyn
  - http://www.cs.wisc.edu/~paradyn
- Pablo
  - http://www.renci.org/software/
- Many other vendor tools exist
  - e.g., xmpi (SGI and HP)

Viewing at Multiple Scales with Jumpshot



Detailed view shows opportunities for optimization

MPI Datatypes

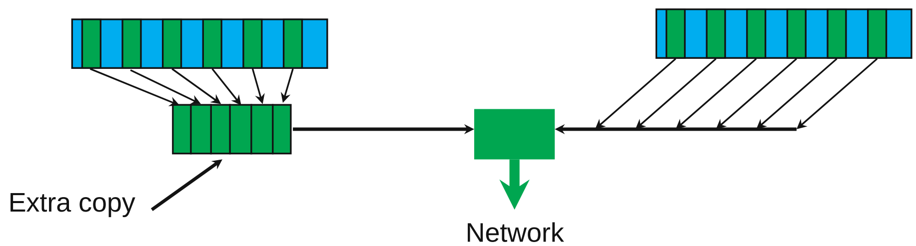
- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

Why Datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication).
- Specifying application-oriented layout of data in memory
  - can reduce memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available
- Specifying application-oriented layout of data on a file
  - can reduce system calls and physical disk I/O

Non-contiguous Datatypes

- Provided to *allow* MPI implementations to avoid copy



  - Not widely optimized yet
- Some MPI implementations can handle important special cases
  - Constant stride
  - Contiguous structures

Potential Performance Advantage in MPI Datatypes

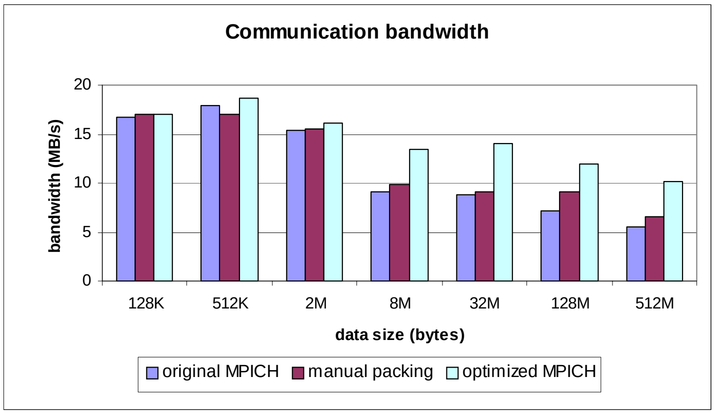
- Handling non-contiguous data
- Assume must pack/unpack on each end
  - $cn + (s + r\,n) + cn = s + (2c + r)n$
- Can move directly
  - $s + r'\,n$
  - r' probably > r but < $(2c+r)$
- MPI implementation must copy data anyway (into network buffer or shared memory); having the datatype permits removing 2 copies

Derived Datatype Performance

Test	Manual (MB/sec)	MPICH2 (%)	MPICH (%)	LAM (%)
Contig	1,156.40	97.2	98.3	86.7
Struct Array	1,055.00	107.0	107.0	48.6
Vector	754.37	99.9	98.7	65.1
Struct Vector	746.04	100.0	4.9	19.0
Indexed	654.35	61.3	12.7	18.8
3D Face, XY	1,807.91	99.5	97.0	63.0
3D Face, XZ	1,244.52	99.5	97.3	79.8
3D Face, YZ	111.85	100.0	100.0	57.4

- (without memory copying optimizations)

Memory Copy Optimizations for Derived Datatypes



- Experiments by Surendra Byna, IIT Chicago
- Matrix transpose example on SGI Origin 2000
- Using MPICH 1.2.5 code base
- Not yet integrated into MPICH2

Working With MPI Datatypes

- An MPI datatype defines a *type signature*:
  - sequence of pairs: (basic type,offset)
  - An integer at offset 0, followed by another integer at offset 8, followed by a double at offset 16 is
    - *(integer,0), (integer,4), (double,16)*
  - Offsets need not be increasing:
    - *(integer,64),(double,0)*
- An MPI datatype has an extent and a size
  - *size* is the number of bytes of the datatype
  - *extent* controls how a datatype is used with the *count* field in a send and similar MPI operations
  - extent is a misleading name

What does extent do?

- Consider MPI_Send(buf, count, datatype, …)
- What actually gets sent?
- MPI defines this as

```
do i=0,count-1
   MPI_Send(buf(1+i*extent(datatype)),1,
                 datatype,…)
```

  (buf is a byte type like integer*1)
- extent is used to decide where to send from (or where to receive to in MPI_Recv) for count > 1
- Normally, this is right after the last byte used for (i-1)

Changing the extent

- MPI-1 provides two special types, MPI_LB and MPI_UB, for changing the extent of a datatype
  - This doesn't change the *size*, just how MPI decides what addresses in memory to use in offseting one datatype from another.
- Use MPI_Type_struct to create a new datatype from an old one with a different extent
  - Use MPI_Type_create_resized in MPI-2

Sending Rows of a Matrix

- From Fortran, assume you want to send a row of the matrix
  A(n,m),
  that is, A(row,j), for j=1,…, m
- A(row,j) is not adjacent in memory to A(row,j+1)
- One solution: send each element separately:

```
Do j=1,m
   Call MPI_Send( A(row,j), 1, MPI_DOUBLE_PRECISION, …)
```

- Why not?

MPI Type vector

- Create a single datatype representing elements separated by a constant distance (*stride*) in memory
  - m items, separated by a stride of n:
  - call MPI_Type_vector(m, 1, n, &
    MPI_DOUBLE_PRECISION, newtype, ierr)
    call MPI_Type_commit(newtype, ierr)
  - Type_commit required before using a type in an MPI communication operation.
- Then send one instance of this type
  MPI_Send(a(row,1), 1, newtype, ….)

Test your understanding of Extent

- How do you send 2 rows of the matrix? Can you do this:
  MPI_Send(a(row,1),2,newtype,…)
- Hint: Extent(newtype) is distance from the first to last byte of the type
  - Last byte is a(row,m)
- Hint: What is the first location of A that is sent after the first row?

Sending with MPI_Vector

- Extent(newtype) is ((m-1)*n+1)*sizeof(double)
  - Last element sent is A(row,m)
- do i=0,1
  call MPI_Send(buf(1+i*extent(datatype)),1, datatype,…)
  becomes
- call MPI_Send(A(row,1:m),…)   (i=0)
  call MPI_Send(A(row+1,m:2m-1),…) (i=1)
- The second step is not
  call MPI_Send(A(row+1,1:m),…)
- Note: Do not use A(row,1:m) in MPI programs; it is used here as a shorthand for A(row,k) for k=1,m

Solutions for Vectors

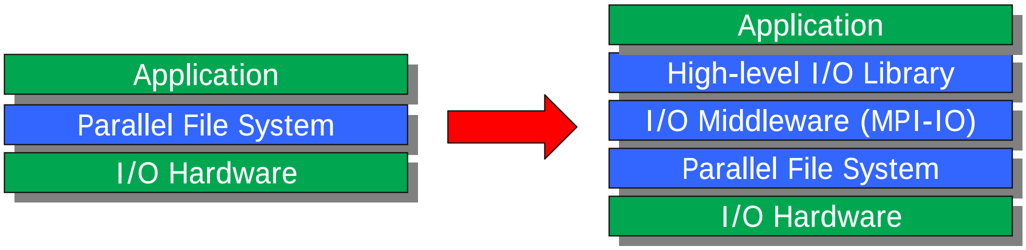
- MPI_Type_vector is for very specific uses
  - rarely makes sense to use count other than 1
- Two send two rows, simply change the blockcount:
  call MPI_Type_vector(m, 2, n, &
  MPI_DOUBLE_PRECISION, newtype, ierr)
- Stride is still relative to basic type

Top MPI Errors

- Fortran: missing ierr argument
- Fortran: missing MPI_STATUS_SIZE on status
- All: MPI_Bcast not called collectively (e.g., sender bcasts, receivers use MPI_Recv)
- All: Failure to wait on MPI_Request
- All: Reusing buffers on nonblocking operations
- All: Using a single process for all file I/O
- All: Using MPI_Pack/Unpack instead of Datatypes
- All: Unsafe use of blocking sends/receives
- All: Using MPI_COMM_WORLD instead of comm in libraries
- All: Not understanding implementation performance settings
- All: Failing to install and use the MPI implementation according to its documentation.

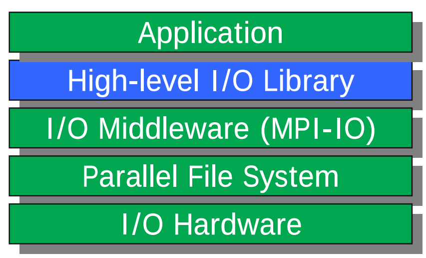
I/O

I/O for Computational Science



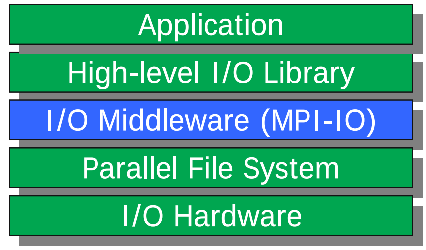
- Break up support into multiple layers with distinct roles:
  - High level I/O library maps app. abstractions to a structured, portable file format (e.g. HDF5, Parallel netCDF)
  - Middleware layer deals with organizing access by many processes (e.g. MPI-IO, UPC-IO)
  - Parallel file system maintains logical space, provides efficient access to data (e.g. PVFS, GPFS, Lustre)

High Level Libraries



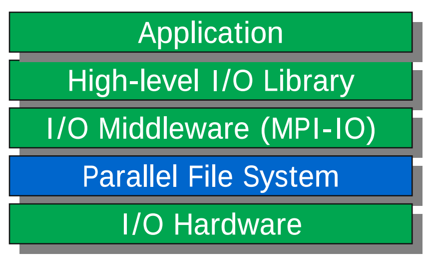
- Provide an appropriate abstraction for domain
  - Multidimensional datasets
  - Typed variables
  - Attributes
- Self-describing, structured file format
- Map to middleware interface
  - Encourage collective I/O
- Provide optimizations that middleware cannot
  - e.g. caching attributes of variables

I/O Middleware



- Facilitate concurrent access by groups of processes
  - Collective I/O
  - Atomicity rules
- Expose a generic interface
  - Good building block for high-level libraries
- Match the underlying programming model (e.g. MPI)
- Efficiently map middleware operations into PFS ones
  - Leverage any rich PFS access constructs

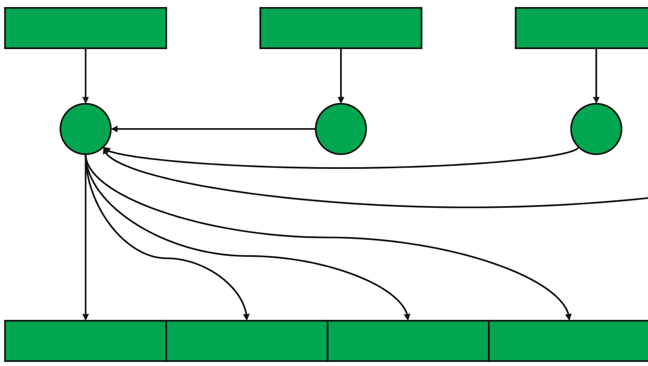
Parallel File System



- Manage storage hardware
  - Present single view
  - Focus on concurrent, independent access
  - Knowledge of collective I/O usually very limited
- In the context of computational science, publish an interface that middleware can use effectively
  - Rich I/O language
  - Relaxed but sufficient semantics

Common Ways of Doing I/O in Parallel Programs

- Sequential I/O:
  - All processes send data to rank 0, and 0 writes it to the file

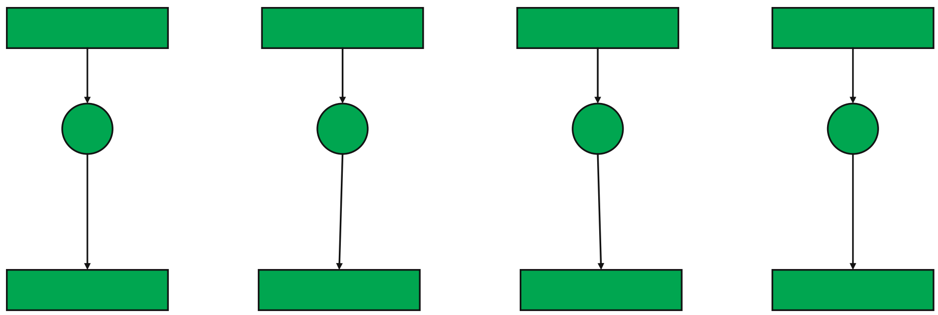


Pros and Cons of Sequential I/O

- Pros:
  - parallel machine may support I/O from only one process (e.g., no common file system)
  - Old versions of some I/O libraries (e.g. HDF-4, NetCDF) not parallel
  - resulting single file is handy for **`ftp`**, **`mv`**
  - big blocks improve performance
  - short distance from original, serial code
- Cons:
  - lack of parallelism limits scalability, performance (single node bottleneck)

Another Way

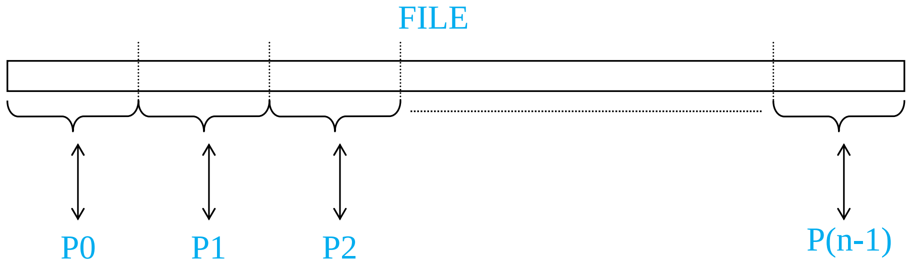
- Each process writes to a separate file



- Pros:
  - parallelism, high performance
- Cons:
  - lots of small files to manage
  - difficult to read back data from different number of processes

What is Parallel I/O?

- Multiple processes of a parallel program accessing data (reading or writing) from a *common* file



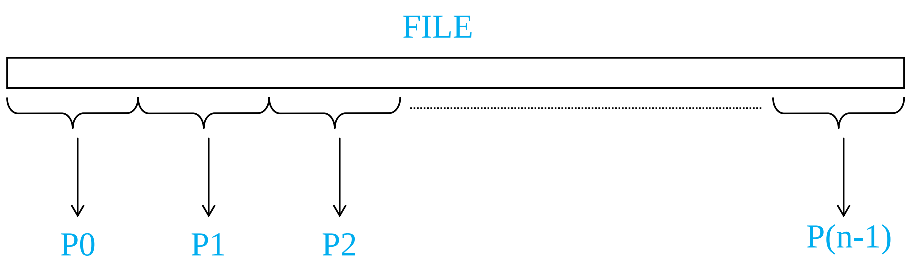
Why Parallel I/O?

- Non-parallel I/O is simple but
  - Poor performance (single process writes to one file) or
  - Awkward and not interoperable with other tools (each process writes a separate file)
- Parallel I/O
  - Provides high performance
  - Can provide a single file that can be used with other tools (such as visualization programs)

Why is MPI a Good Setting for Parallel I/O?

- Writing is like sending a message and reading is like receiving
- Any parallel I/O system will need a mechanism to
  - define collective operations (*MPI communicators*)
  - define noncontiguous data layout in memory and file (*MPI datatypes*)
  - Test completion of nonblocking operations (*MPI request objects*)
- i.e., lots of MPI-like machinery

Using MPI for Simple I/O



Each process needs to read a chunk of data from a common file

Using Individual File Pointers

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

Using Explicit Offsets

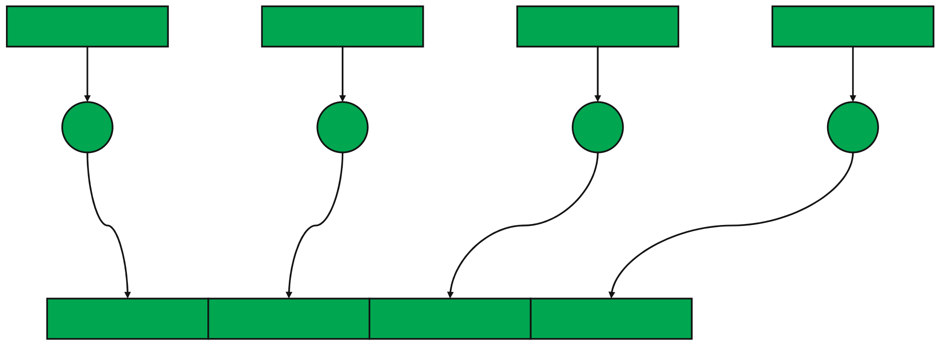
```
    include 'mpif.h'

    integer status(MPI_STATUS_SIZE)
    integer (kind=MPI_OFFSET_KIND) offset
C   in F77, see implementation notes (might be integer*8)

    call MPI_FILE_OPEN(MPI_COMM_WORLD, '/pfs/datafile', &
              MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
    nints = FILESIZE / (nprocs*INTSIZE)
    offset = rank * nints * INTSIZE
    call MPI_FILE_READ_AT(fh, offset, buf, nints,
                          MPI_INTEGER, status, ierr)
    call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
    print *, 'process ', rank, 'read ', count, 'integers'

    call MPI_FILE_CLOSE(fh, ierr)
```


Writing to a File

- Use **MPI_File_write** or **MPI_File_write_at**
- Use **MPI_MODE_WRONLY** or **MPI_MODE_RDWR** as the flags to **MPI_File_open**
- If the file doesn't exist previously, the flag **MPI_MODE_CREATE** must also be passed to **MPI_File_open**
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+" in Fortran

Using File Views

- Processes write to shared file



- **`MPI_File_set_view`** assigns regions of the file to separate processes

File Views

- Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **`MPI_File_set_view`**
- *displacement* = number of bytes to be skipped from the start of the file
- *etype* = basic unit of data access (can be any basic or derived datatype)
- *filetype* = specifies which portion of the file is visible to the process

File View Example

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",
               MPI_MODE_CREATE | MPI_MODE_WRONLY,
               MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile, myrank * BUFSIZE * sizeof(int),
                   MPI_INT, MPI_INT, "native",
                  MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
                MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```


MPI_File_set_view

- Describes that part of the file accessed by a single MPI process.
- Arguments to **MPI_File_set_view**:
  - **MPI_File file**
  - **MPI_Offset disp**
  - **MPI_Datatype etype**
  - **MPI_Datatype filetype**
  - **char *datarep**
  - **MPI_Info info**

Fortran Version

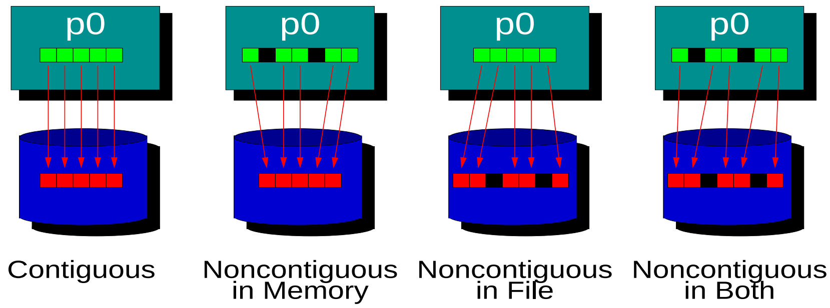
```
PROGRAM main

use mpi

integer ierr, i, myrank, BUFSIZE, thefile
parameter (BUFSIZE=100)
integer buf(BUFSIZE)
integer(kind=MPI_OFFSET_KIND) disp

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
do i = 0, BUFSIZE
    buf(i) = myrank * BUFSIZE + i
enddo
```

```
* in F77, see implementation notes (might be integer*8)
```


Fortran Version contd.

```
call MPI_FILE_OPEN(MPI_COMM_WORLD, 'testfile', &
                   MPI_MODE_WRONLY + MPI_MODE_CREATE, &
                   MPI_INFO_NULL, thefile, ierr)
call MPI_TYPE_SIZE(MPI_INTEGER, intsize)
disp = myrank * BUFSIZE * intsize
call MPI_FILE_SET_VIEW(thefile, disp, MPI_INTEGER, &
                       MPI_INTEGER, 'native', &
                       MPI_INFO_NULL, ierr)
call MPI_FILE_WRITE(thefile, buf, BUFSIZE, MPI_INTEGER, &
                       MPI_STATUS_IGNORE, ierr)
call MPI_FILE_CLOSE(thefile, ierr)
call MPI_FINALIZE(ierr)

END PROGRAM main
```

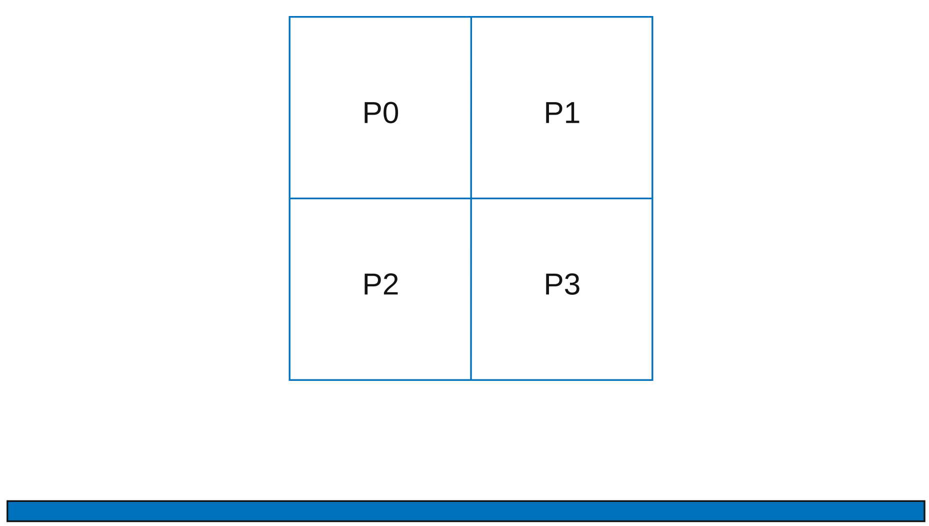
Noncontiguous I/O



- Contiguous I/O moves data from a single block in memory into a single region of storage
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O

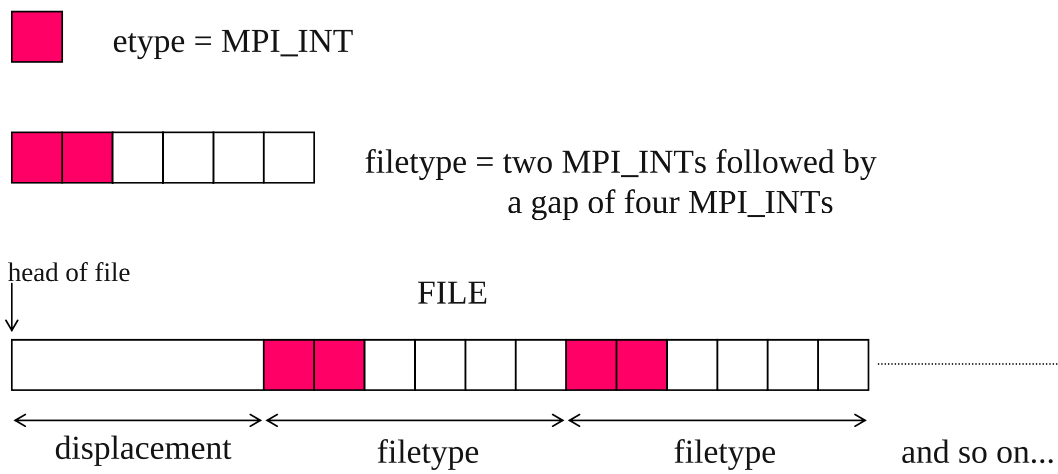
Example: Distributed Array Access

2D array distributed among four processes



File containing the global array in row-major order

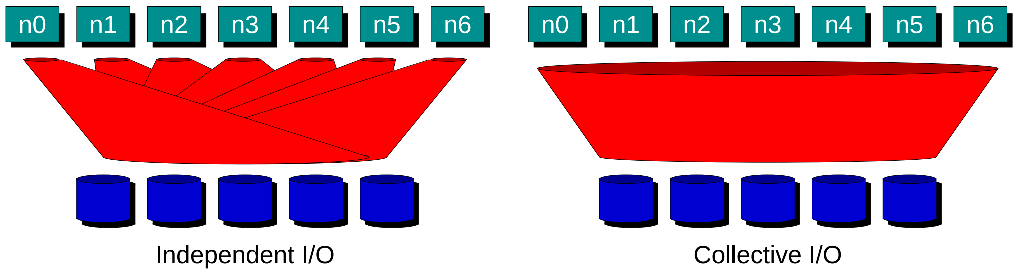
A Simple Noncontiguous File View Example



File View Code

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0; extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int); etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
     MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

Collective I/O (1)



- Many applications have phases of computation and I/O
- During I/O phases, all processes read/write data
  - We can say they are collectively accessing storage
- Collective I/O is coordinated access to storage by a group of processes
  - Collective I/O functions must be called by all processes participating in I/O
  - Allows I/O layers to know more about access as a whole
- Independent I/O is not organized in this way
- No apparent order or structure to accesses

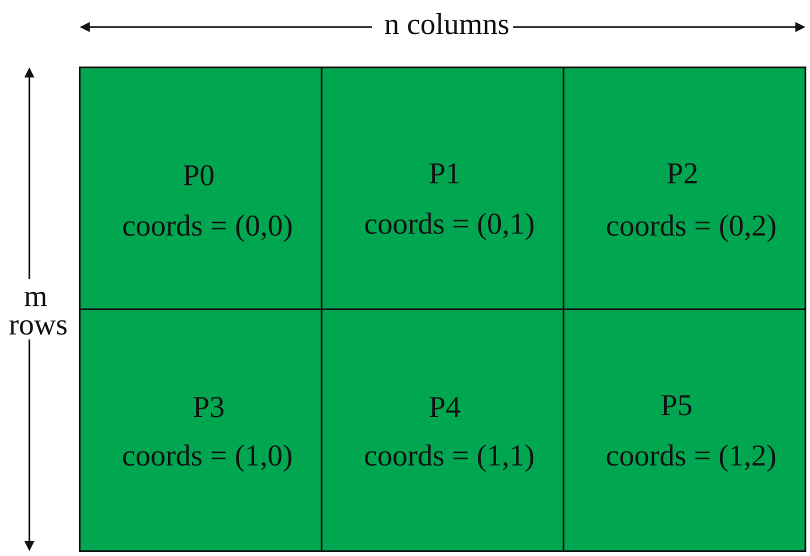
Collective I/O (2)

- **`MPI_File_read_all`**, **`MPI_File_read_at_all`**, etc
- **`_all`** indicates that all processes in the group specified by the communicator passed to **`MPI_File_open`** will call this function
- Each process specifies only its own access information -- the argument list is the same as for the non-collective functions

Under the Covers of MPI-IO

- MPI-IO implementation is given a lot of information in this case:
  - Collection of processes reading data
  - Structured description of the regions
- Implementation has some options for how to obtain this data
  - Noncontiguous data access optimizations
  - Collective I/O optimizations

Accessing Arrays Stored in Files



nproc(1) = 2, nproc(2) = 3

Using the Subarray Datatype

```
gsizes[0] = m;  /* no. of rows in global array */
gsizes[1] = n;  /* no. of columns in global array*/

psizes[0] = 2; /* no. of procs. in vertical dimension */
psizes[1] = 3; /* no. of procs. in horizontal dimension */

lsizes[0] = m/psizes[0]; /* no. of rows in local array */
lsizes[1] = n/psizes[1]; /* no. of columns in local array */

dims[0] = 2; dims[1] = 3;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);
```


Subarray Datatype contd.

```
/* global indices of first element of local array */
start_indices[0] = coords[0] * lsizes[0];
start_indices[1] = coords[1] * lsizes[1];

MPI_Type_create_subarray(2, gsizes, lsizes, start_indices,
                      MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
               MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native",
                MPI_INFO_NULL);
local_array_size = lsizes[0] * lsizes[1];
MPI_File_write_all(fh, local_array, local_array_size,
                 MPI_FLOAT, &status);
```

MPI-IO Hints

- MPI-IO hints may be passed via:
  - `MPI_File_open`
  - `MPI_File_set_info`
  - `MPI_File_set_view`
- Hints are optional - implementations are guaranteed to ignore ones they do not understand
  - Different implementations, even different underlying file systems, support different hints
- `MPI_File_get_info` used to get list of hints
- Next few slides cover only some hints

Examples of Hints (used in ROMIO)

MPI-2 predefined hints:
- **`striping_unit`**
- **`striping_factor`**
- **`cb_buffer_size`**
- **`cb_nodes`**

New Algorithm Parameters:
- **`ind_rd_buffer_size`**
- **`ind_wr_buffer_size`**

Platform-specific hints:
- **`start_iodevice`**
- **`pfs_svr_buf`**
- **`direct_read`**
- **`direct_write`**

Passing Hints to the Implementation

```
MPI_Info info;

MPI_Info_create(&info);

/* no. of I/O devices to be used for file striping */
MPI_Info_set(info, "striping_factor", "4");

/* the striping unit in bytes */
MPI_Info_set(info, "striping_unit", "65536");

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR, info, &fh);

MPI_Info_free(&info);
```

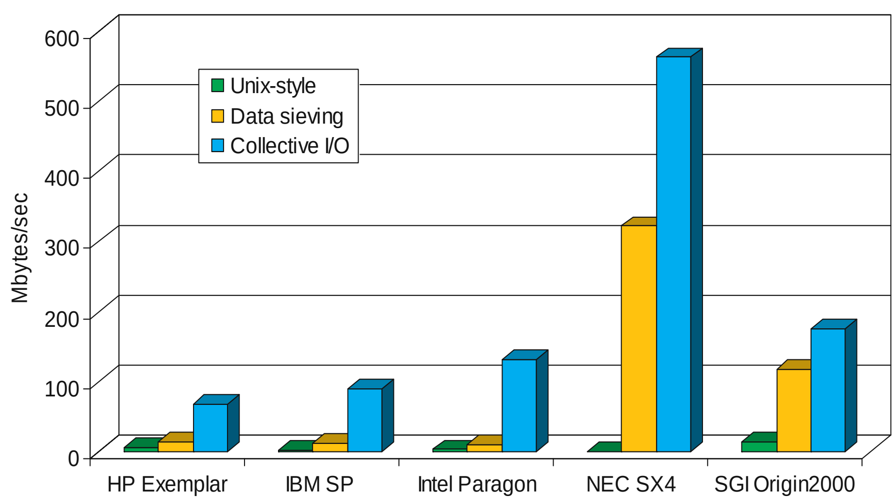

General Guidelines for Achieving High I/O Performance

- Buy sufficient I/O hardware for the machine
- Use fast parallel file systems, such as PVFS, not NFS-mounted home directories
- Do not perform I/O from one process only
- Make large requests wherever possible
- For noncontiguous requests, use derived datatypes and a single collective I/O call

Optimizations

- Given complete access information, an implementation can perform optimizations such as:
  - Data Sieving: Read large chunks and extract what is really needed
  - Collective I/O: Merge requests of different processes into larger requests
  - Improved prefetching and caching

Distributed Array Access: Read Bandwidth



64 procs 64 procs 256 procs 8 procs 32 procs

Array size: 512 x 512 x 512

MPI-IO Wrap-Up

- MPI-IO provides a rich interface allowing us to describe
  - Noncontiguous accesses in memory, file, or both
  - Collective I/O
- This allows implementations to perform many transformations in order to get better I/O performance
- Also forms solid basis for high-level I/O libraries
  - But they must take advantage of these features!

Higher Level I/O Interfaces

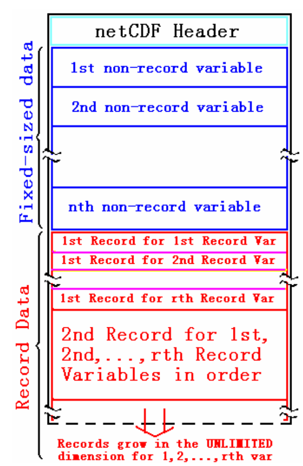
- Provide structure to files
  - Well-defined, portable formats
  - Self-describing
  - Organization of data
  - Interfaces for discovering contents
- Present APIs more appropriate for comp. science
  - Typed data
  - Noncontiguous regions in memory and file
  - Multidimensional arrays
- Both implemented on top of MPI-IO

Parallel netCDF (PnetCDF)

- Based on original “Network Common Data Format” (netCDF) work from Unidata
- Data Model:
  - Collection of variables in single file
  - Typed, multidimensional array variables
  - Attributes on file and variables
- Features:
  - C and Fortran interfaces
  - Portable data format (same as netCDF)
  - Noncontiguous I/O in memory using MPI datatypes
  - Noncontiguous I/O in file using subarrays
  - Collective I/O

netCDF/PnetCDF Files

- PnetCDF files consist of three regions
  - Header
  - Non-record variables (all dimensions specified)
  - Record variables (ones with an unlimited dimension)
- Record variables are interleaved, so using more than one in a file is likely to result in poor performance due to noncontiguous accesses
- Data is written in a big-endian format



Storing Data in PnetCDF

- Create a dataset (file)
  - Puts dataset in define mode
  - Allows us to describe the contents
    - *Define dimensions for variables*
    - *Define variables using dimensions*
    - *Store attributes if desired (for variable or dataset)*
- Switch from define mode to data mode to write variables
- Store variable data
- Close the dataset

Simple PnetCDF Examples

- Simplest possible PnetCDF version of “Hello World”
- First program creates a dataset with a single attribute
- Second program reads the attribute and prints it
- Shows very basic API use and error checking

Simple PnetCDF: Writing (1)

Integers used for references to datasets, variables, etc.

```
#include <mpi.h>
#include <pnetcdf.h>
int main(int argc, char **argv)
{
    int ncfile, ret, count;
    char buf[13] = "Hello World\n";
    MPI_Init(&argc, &argv);
    ret = ncmpi_create(MPI_COMM_WORLD, "myfile.nc",
      NC_CLOBBER, MPI_INFO_NULL, &ncfile);
    if (ret != NC_NOERR) return 1;

    /* continues on next slide */
```


Simple PnetCDF: Writing (2)

```
    ret = ncmpi_put_att_text(ncfile, NC_GLOBAL,
      "string", 13, buf);
    if (ret != NC_NOERR) return 1;
    ncmpi_enddef(ncfile);

    /* entered data mode – but nothing to do */
    /* ncmpi_put_vara_double_all(…); */

    ncmpi_close(ncfile);
    MPI_Finalize();
    return 0;
}
```

Storing value while in define mode as an attribute

Retrieving Data in PnetCDF

- Open a dataset in read-only mode (`NC_NOWRITE`)
- Obtain identifiers for dimensions
- Obtain identifiers for variables
- Read variable data
- Close the dataset

Simple PnetCDF: Reading (1)

```
#include <mpi.h>
#include <pnetcdf.h>
int main(int argc, char **argv)
{
    int ncfile, ret, count;
    char buf[13];
    MPI_Init(&argc, &argv);
    ret = ncmpi_open(MPI_COMM_WORLD, "myfile.nc",
      NC_NOWRITE, MPI_INFO_NULL, &ncfile);
    if (ret != NC_NOERR) return 1;

    /* continues on next slide */
```

Simple PnetCDF: Reading (2)

```
    /* verify attribute exists and is expected size */
    ret = ncmpi_inq_attlen(ncfile, NC_GLOBAL, "string",
      &count);
    if (ret != NC_NOERR || count != 13) return 1;

    /* retrieve stored attribute */
    ret = ncmpi_get_att_text(ncfile, NC_GLOBAL, "string", buf);
    if (ret != NC_NOERR) return 1;
    printf("%s", buf);

    ncmpi_close(ncfile);
    MPI_Finalize();
    return 0;
}
```


Compiling and Running

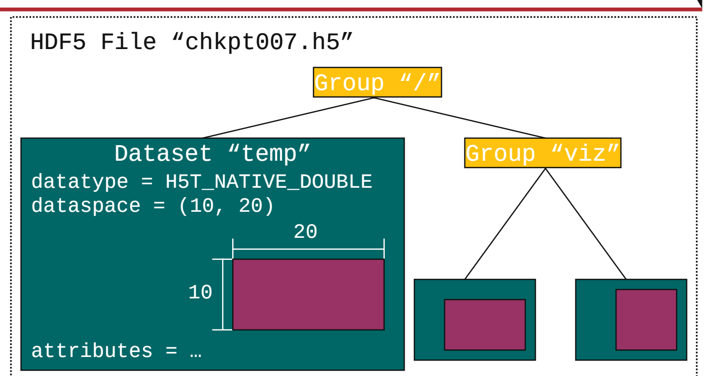
```
;mpicc pnetcdf-hello-write.c -I /usr/local/pnetcdf/include/ -L
   /usr/local/pnetcdf/lib -lpnetcdf -o pnetcdf-hello-write
;mpicc pnetcdf-hello-read.c -I /usr/local/pnetcdf/include/ -L
   /usr/local/pnetcdf/lib -lpnetcdf -o pnetcdf-hello-read
;mpiexec -n 1 pnetcdf-hello-write
;mpiexec -n 1 pnetcdf-hello-read
Hello World

;ls -l myfile.nc
-rw-r--r--    1 rross    rross        68 Mar 26 10:00 myfile.nc

;strings myfile.nc
string
Hello World
```

File size is 68 bytes; extra data (the header) in file.

HDF5

- Hierarchical Data Format, from NCSA
- Data Model:
  – Hierarchical data organization in single file
  – Typed, multidimensional array storage
  – Attributes on dataset, data
- Features:
  – C, C++, and Fortran interfaces
  – Portable data format
  – Optional compression (not in parallel I/O mode)
  – Data reordering (chunking)
  – Noncontiguous I/O (memory and file) with hyperslabs

HDF5 Files



- HDF5 files consist of groups, datasets, and attributes
  – Groups are like directories, holding other groups and datasets
  – Datasets hold array of typed data
    • *A datatype describes the type*
    • *A dataspace gives the dimensions of the array*
  – Attributes are small datasets associated with the file, a group, or another dataset
    • *Also have a datatype and dataspace*
    • *Can only be accessed as a unit*

HDF5 Data Chunking

- Apps often read subsets of arrays (subarrays)
- Performance of subarray access depends in part on how data is laid out in the file
  - e.g. column vs. row major
- Apps also sometimes store sparse data sets
- Chunking describes a reordering of array data
  - Subarray placement in file determined lazily
  - Can reduce worst-case performance for subarray access
  - Can lead to efficient storage of sparse data
- Coordination cost in this dynamic ordering

"Simple" HDF5 Examples

- HDF5 version of "Hello World"
- First program creates a character array, writes text into it
- Second program reads back the array and prints the contents
- Shows basic API use

"Simple" HDF5: Writing (1 of 3)

```
#include <mpi.h>
#include <hdf5.h>
int main(int argc, char **argv)
{
   hid_t file, string_dtype,
     string_dspace, string_dset;
   hsize_t dim = 13;
   herr_t status;
   char buf[13] = "Hello World\n";
   MPI_Init(&argc, &argv);
   file = H5Fcreate("myfile.h5", H5F_ACC_TRUNC,
     H5P_DEFAULT, H5P_DEFAULT);
   /* continued on next slide */
```

hid_t type used for references to files, datatypes, etc.

the "F" in H5Fcreate means that this a file operation

"Simple" HDF5: Writing (2 of 3)

```
/* To create the dataset we:
 * - make a simple, 1-D dataspace to describe shape of set
 * - get a copy of the "native" char type that we can use
 * - combine these two to create a dataset in the file
 */
string_dspace = H5Screate_simple(1, &dim, NULL);
string_dtype  = H5Tcopy(H5T_NATIVE_CHAR);
string_dset   = H5Dcreate(file, "string",
  string_dtype, string_dspace, H5P_DEFAULT);

status = H5Dwrite(string_dset,
                  H5T_NATIVE_CHAR,
                  H5S_ALL,
                  H5S_ALL,
                  H5P_DEFAULT,
                  buf);
```

Remember:
"S" is for dataspace,
"T" is for datatype,
"D" is for dataset!

dataset to write into (target)

memory datatype and dataspace

file dataspace (for subarray access)

pointer to buffer in memory

“Simple” HDF5: Writing (3 of 3)

```
    /* call appropriate close functions on all references */
    H5Sclose(string_dataspace);
    H5Tclose(string_datatype);
    H5Dclose(string_dataset);
    H5Fclose(file);

    MPI_Finalize();
    return 0;
}
```


Simple HDF5: Reading

```
#include <hdf5.h>
#include <stdio.h>
int main(int argc, char **argv) {
    hid_t file, string_dset;
    char buf[13];

    file = H5Fopen("myfile.h5", H5F_ACC_RDONLY,
      H5P_DEFAULT);
    string_dset = H5Dopen(file, "string");
    H5Dread(string_dset,
              H5T_NATIVE_CHAR, H5S_ALL,
              H5S_ALL,
              H5P_DEFAULT,
              buf);

    printf("%s", buf);
    H5Dclose(string_dset);  H5Fclose(file);
    return 0;
}
```

Remember:
“S” is for dataspace,
“T” is for datatype,
“D” is for dataset!

Compiling and Running

```
;mpicc hdf5-hello-write.c -I /usr/local/hdf5/include -L
  /usr/local/hdf5/lib/ -lhdf5 -o hdf5-hello-write
;mpicc hdf5-hello-read.c -I /usr/local/hdf5/include -L
  /usr/local/hdf5/lib/ -lhdf5 -o hdf5-hello-read
;mpiexec -n 1 hdf5-hello-write
;mpiexec -n 1 hdf5-hello-read
Hello World
;ls -l myfile.h5
-rw-r--r--    1 rross    rross       2061 Mar 27 23:06
  myfile.h5
;strings myfile.h5
HEAP
string
TREE
P]f@
SNOD
Hello World
```

File size is 2061 bytes; bigger header.

How do I choose an API?

- Your programming model will limit choices.
  - Domain might too (e.g. Climate, existing netCDF data)
- Find something that matches your data model.
- Avoid APIs with lots of features you won't use.
  - Potential for overhead costing performance is high.
- Maybe the right API isn't available?
  - Get I/O people interested, consider designing a new library

Summary of API Capabilities

	POSIX	MPI-IO	PnetCDF	HDF5
Noncontig. Memory	Yes	Yes	Yes	Yes
Noncontig. File	Sort-of	Yes	Yes	Yes
Coll. I/O		Yes	Yes	Yes
Portable Format		Yes	Yes	Yes
Self-Describing			Yes	Yes
Attributes			Yes	Yes
Chunking				Yes
Hierarchical File				Yes

Tuning Application I/O (1 of 2)

- Have realistic goals:
  - What is peak I/O rate?
  - What other testing has been done?
- Describe as much as possible to the I/O system:
  - Open with appropriate mode.
  - Use collective calls when available.
  - Describe data movement with fewest possible operations.
- Match file organization to process partitioning if possible
  - Order dimensions so relatively large blocks are contiguous with respect to data decomposition

Tuning Application I/O (2 of 2)

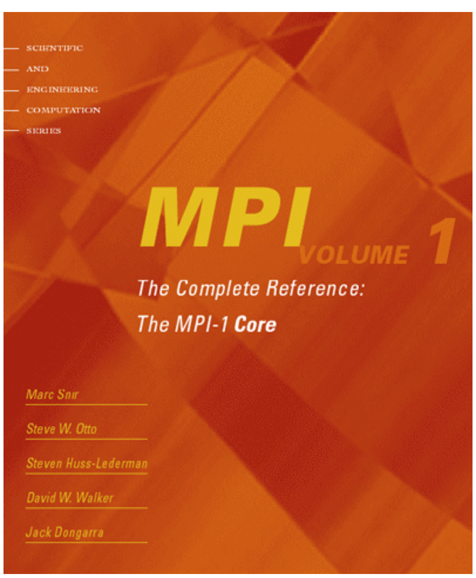
- Know what you can control:
  - What I/O components are in use?
  - What hints are accepted?
- Consider system architecture as a whole:
  - Is storage network faster than comm. network?
  - Do some nodes have better storage access than others?
- These guide our selection of hints

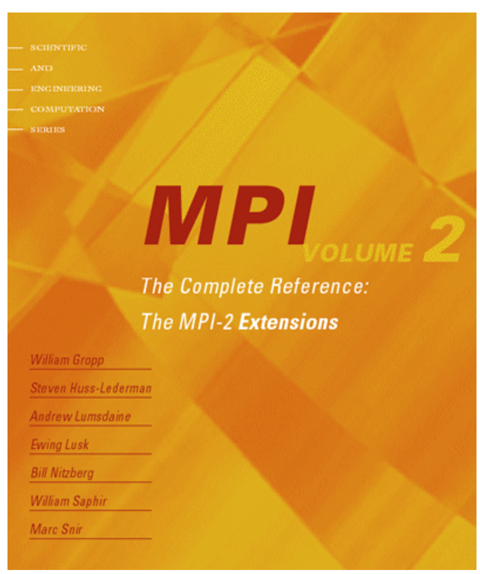
References

MPI Sources

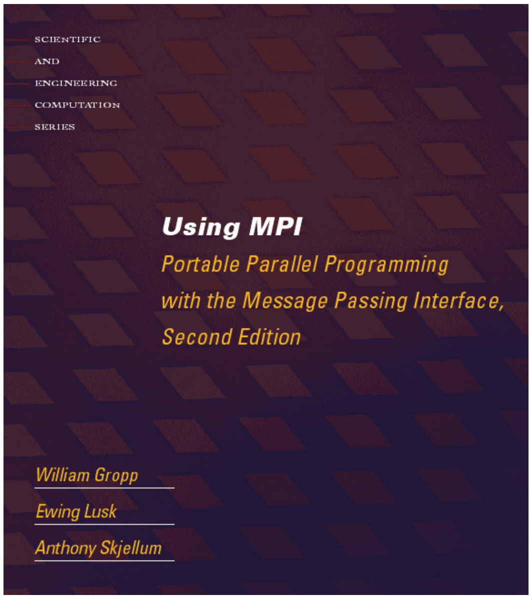
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd Edition, by Gropp, Lusk, and Skjellum, MIT Press, 1999. Also *Using MPI-2*, w. R. Thakur
  - *MPI: The Complete Reference,* 2 vols*,* MIT Press, 1999.
  - *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
  - *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

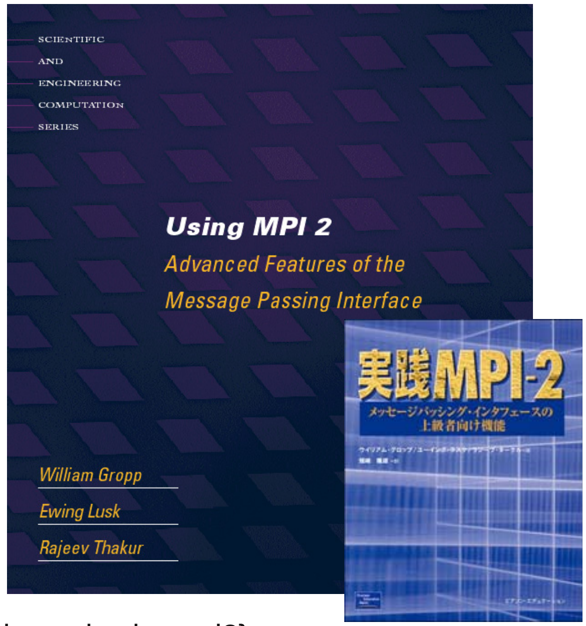
The MPI Standard (1 & 2)





Tutorial Material on MPI, MPI-2

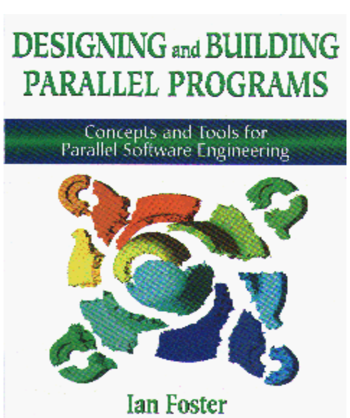


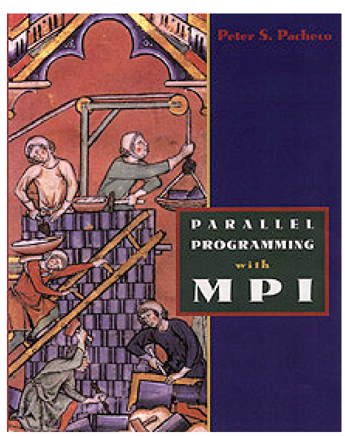


http://www.mcs.anl.gov/mpi/{usingmpi,usingmpi2}

Books on Programming with MPI

- *Designing and Building Parallel Programs*, by Ian Foster
- *Parallel Programming with MPI*, by Peter Pacheco
- *Using MPI,* by William Gropp, Ewing Lusk, Anthony Skjellum
- Practical MPI Programming, by Yukiya Aoyama and Jun Nakano (http://www.redbooks.com)





I/O References

- John May, Parallel I/O for High Performance Computing, Morgan Kaufmann, October 9, 2000.
  - Good coverage of basic concepts, some MPI-IO, HDF5, and serial netCDF
- netCDF

  http://www.unidata.ucar.edu/packages/netcdf/
- PnetCDF

  http://www.mcs.anl.gov/parallel-netcdf/
- ROMIO MPI-IO

  http://www.mcs.anl.gov/romio/
- HDF5 and HDF5 Tutorial

  http://hdf.ncsa.uiuc.edu/HDF5/

  http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor/index.html