Programming with MPI: Advanced Topics

Steve Lantz
Senior Research Associate
Cornell CAC

Workshop: Introduction to Parallel Computing on Ranger, July 14, 2010
Based on materials developed by Bill Barth at TACC
Goals

• To gain an awareness of specialized features in MPI that you may want to use right away in writing parallel applications
• To create a little mental catalog of MPI’s more advanced capabilities for future reference

At the end of each section, let’s ask:
• Why was this set of routines included? What might they be good for?
• Can we think of an example where they would be useful?
Introduction and Outline

1. Advanced point-to-point communication
2. Collective communication with non-contiguous data
3. Derived datatypes
4. Communicators and groups
5. Persistent communication
6. Parallel I/O (MPI-2)
7. Status of MPI-2
1. Advanced Point-to-Point Communication
Standard Send, Receive

*Standard-Mode Blocking Calls: MPI_Send, MPI_Recv*

- MPI_Send returns only when the buffer is safe to reuse:
  - the small message has been copied elsewhere, or
  - the large message has actually been transferred;
  - the small/large threshold is implementation dependent
- Rule of thumb: a send only completes if a matching receive is posted/executed
Synchronous and Buffered Modes

**Synchronous Mode**: MPI_Ssend
- Transfer is not initiated until matching receive is posted
- Non-local: handshake needed
- Returns after message is sent

**Buffered Mode**: MPI_Bsend
- Completes as soon as the message is copied into the user-provided buffer
- Buffer must be provided using MPI_Buffer_attach
- One buffer per process
Ready Mode and Deadlock

*Ready Mode*: MPI_Rsend
- Initiates transfer immediately
- Assumes that a matching receive has already been posted
- Error if receiver isn’t ready

*Deadlock*
- All tasks are waiting for events that yet haven’t been initiated
- Can be avoided by reordering calls, by using non-blocking calls, or with MPI_Sendrecv
Discussion of Send Modes

- Synchronous mode is portable and “safe”
  - does not depend on order (ready mode) or buffer space (buffered mode)
  - incurs substantial overhead
- Ready mode has least total overhead, but how can error be avoided?
  - sometimes the logic of the code implies the receiver must be ready
- Buffered mode decouples sender and receiver
  - sender doesn’t have to sync; receiver doesn’t have to be ready
  - time and memory overheads are incurred by copying to the buffer
  - sender can control size of message buffers and total amount of space
- Standard mode tries to strike a balance
  - small messages are buffered on receiver’s side (avoiding sync overhead)
  - large messages are sent synchronously (avoiding big buffer space)
**MPI_Sendrecv and MPI_Sendrecv_replace**

- **MPI_Sendrecv (blocking)**
  - send message A from one buffer; receive message B in another buffer
  - destination of A, source of B can be same or different
- **MPI_Sendrecv_replace (blocking)**
  - send message A from one buffer; receive message B in *SAME* buffer
  - again, destination of A, source of B can be same or different
  - system takes care of the extra internal buffering
- **Illustration 1**: data swap between processors
  - destination and source are identical
- **Illustration 2**: chain of processors
  - send result to `myrank+1`, receive next input from `myrank-1`
Non-Blocking Calls

- Calls return immediately
- System handles buffering
- Not “safe” to access message contents until action is known to be completed
- With MPI_Isend, message buffer is reusable right away if tag or receiver is different; otherwise, check status
- With MPI_Irecv, user must always check for data; only small messages are buffered
Use of Non-Blocking Communication

- Non-blocking calls permit overlap of computation and communication
- All send modes are available: MPI_Irsend, MPI_Ibsend, MPI_Issend
- Non-blocking calls must normally be resolved through a second call
  - main options: MPI_Wait, MPI_Test, MPI_Request_free
  - variants like MPI_Waitany help to resolve calls in arbitrary order
  - reason for doing this: avoid running out of request handles
- Outline for typical code:

```c
for (i=0;i<M;i++) MPI_Irecv( <declare receive buffers> );
for (i=0;i<N;i++) MPI_Isend( <mark data for sending> );
   /* Do local operations */
MPI_Waitall( <make sure all receives finish> )
   /* Operate on received data */
MPI_Waitall( <clear request handles for all sends> )
```
MPI_Wait and MPI_Test

- **MPI_Wait** halts progress until a *specific* non-blocking request (send or receive) is satisfied; the related message buffer is then safe to use
  - **MPI_Waitall** does the same thing for a *whole array* of requests
  - **MPI_Waitany** waits for *any one* request from an array
  - **MPI_Waitsome** waits for *one or more* requests from an array
- **MPI_Test** immediately returns the status (no waiting!) of a specific non-blocking operation, again identified by a request handle
  - returns **flag = true** only if the operation is complete
  - allows alternative instructions to be carried out if operation isn’t complete
  - has the same variants: MPI_Testall, MPI_Testany, MPI_Testsome

```c
MPI_Testany(int count, MPI_Request *array_of_reqs,
            int *index, int *flag, MPI_Status *status);
```
Other Ways to Gain Flexibility in Communication

- **MPI_ANY_SOURCE, MPI_ANY_TAG** are “wildcards” that may be used by receives (blocking and non-blocking) in situations where the source or tag of a message does not need to be known in advance
  - the *status* argument returns source, tag, and error status
  - a separate call to **MPI_Get_count** determines the size of the message
  - but… what if you need to know a message’s size *before* receiving it?

- **MPI_Iprobe** returns the properties of any message that has arrived without receiving it into a buffer (maybe you need to do a big malloc!)
  
  ```
  MPI_Iprobe(int source, int tag, MPI_Comm comm,
             int *flag, MPI_Status *status);
  ```

- **MPI_Probe** blocks until such a message arrives (no flag)
2. Collective Communication with Non-Contiguous Data
Review: Scatter and Gather

<table>
<thead>
<tr>
<th>p0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>A</td>
</tr>
<tr>
<td>p2</td>
<td>A</td>
</tr>
<tr>
<td>p3</td>
<td>A</td>
</tr>
</tbody>
</table>

broadcast

<table>
<thead>
<tr>
<th>p0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>B</td>
</tr>
<tr>
<td>p2</td>
<td>C</td>
</tr>
<tr>
<td>p3</td>
<td>D</td>
</tr>
</tbody>
</table>

scatter

<table>
<thead>
<tr>
<th>p0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>B</td>
</tr>
<tr>
<td>p2</td>
<td>C</td>
</tr>
<tr>
<td>p3</td>
<td>D</td>
</tr>
</tbody>
</table>

gather

<table>
<thead>
<tr>
<th>p0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>A</td>
</tr>
<tr>
<td>p2</td>
<td>A</td>
</tr>
<tr>
<td>p3</td>
<td>A</td>
</tr>
</tbody>
</table>

allgather
Introducing Scatterv, Gatherv

• MPI_{Scatter,Gather,Allgater}v
• What does v stand for?
  – varying size and relative location of messages
• Advantages
  – more flexibility
  – less need to copy data into temporary buffers
  – more compact
• Disadvantage
  – harder to program
CALL mpi_scatterv ( SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERR )

- \texttt{SENDCOUNTS(J)} is the number of items of type \texttt{SENDTYPE} to send from process \texttt{ROOT} to process \texttt{J}. Defined on \texttt{ROOT}.

- \texttt{DISPLS(J)} is the displacement from \texttt{SENDBUF} to the beginning of the \texttt{J}-th message, in units of \texttt{SENDTYPE}. Defined on \texttt{ROOT}. 

Scatter vs. Scatterv
Allgatherv Example

MPI_Comm_size(comm,&ntids);
sizes = (int*)malloc(ntids*sizeof(int));
MPI_Allgather(&n,1,MPI_INT,sizes,1,MPI_INT,comm);
offsets = (int*)malloc(ntids*sizeof(int));
s=0;
for (i=0; i<ntids; i++)
    {offsets[i]=s; s+=sizes[i];}
N = s;
result_array = (int*)malloc(N*sizeof(int));
MPI_Allgatherv
    ((void*)local_array,n,MPI_INT,(void*)result_array,
        sizes,offsets,MPI_INT,comm);
free(sizes); free(offsets);
3. Derived Datatypes
Derived Datatypes: Motivation

- MPI basic datatypes are predefined for contiguous data of single type
- What if an application needs to communicate data of mixed type or in non-contiguous locations?
  - solutions that involve making multiple MPI calls, copying data into a buffer and packing, etc., are slow, clumsy and wasteful of memory
  - better solution is to create/derive datatypes for these special needs from existing datatypes
- Derived datatypes can be created recursively at runtime
- Packing and unpacking is done automatically
MPI Datatypes

- **Elementary**: Language-defined types
- **Contiguous**: Vector with stride of one
- **Vector**: Elements separated by constant “stride”
- **Hvector**: Vector, with stride in bytes
- **Indexed**: Array of indices (for scatter/gather)
- **Hindexed**: Indexed, with indices in bytes
- **Struct**: General mixed types (for C structs etc.)
Picturing Some Derived Datatypes

Vector (strided)

- \( \text{blklen}=2 \)
- \( \text{count}=3 \) elements
- \( \text{stride}=5 \) (in elements)
- \( \text{v_blk_len}[0]=3 \)
- \( \text{v_blk_len}[1]=2 \)
- \( \text{v_blk_len}[2]=1 \)
- \( \text{v_disp}[0]=0 \)
- \( \text{v_disp}[1]=5 \) (in elements)
- \( \text{v_disp}[2]=12 \)

Indexed

- \( \text{count}=3 \) blocks
- \( \text{v_blk_len}[0]=3 \)
- \( \text{v_blk_len}[1]=2 \)
- \( \text{v_blk_len}[2]=1 \)
- \( \text{v_disp}[0]=0 \)
- \( \text{v_disp}[1]=5 \) (in elements)
- \( \text{v_disp}[2]=12 \)

“Struct”

- \( \text{count}=3 \) blocks
- \( \text{v_blk_len}[0]=2 \)
- \( \text{v_blk_len}[1]=3 \)
- \( \text{v_blk_len}[2]=4 \)
- \( \text{v_disp}[0] \)
- \( \text{v_disp}[1] \) (in bytes)
- \( \text{v_disp}[2] \)
Using MPI’s Vector Type

- Function `MPI_TYPE_VECTOR` allows creating non-contiguous vectors with constant stride. Where might one use it?

```
mpi_type_vector(count,blocklen,stride,oldtype,vtype,ierr)
```

Array A

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>12</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>13</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>14</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
</tr>
</tbody>
</table>

ncols = 4
nrows = 5

call MPI_Type_vector(ncols,1,nrows,MPI_DOUBLE_PRECISION,&vtype,ierr)
call MPI_Type_commit(vtype,ierr)
call MPI_Send(A(nrows,1),1,vtype...)
4. Communicators and Groups
Communicators and Groups: Definitions

- All MPI communication is relative to a *communicator* which contains a *context* and a *group*. The group is just a set of processes.

- Processes may have different ranks in different communicators.
Subdividing Communicators: Approach #1

- To subdivide a communicator into multiple non-overlapping communicators, one approach is to use `MPI_Comm_split`.

  ```c
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  myrow = (int)(rank/ncol);
  MPI_Comm_split(MPI_COMM_WORLD,myrow,rank,row_comm);
  ```

### Diagram

![Diagram of column and row distribution](image.png)

- **Column**
  - 0
  - 1
  - 2
  - 3
  - 4

- **Row**
  - 0
  - 1
  - 2
Arguments to MPI_Comm_split

```c
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
myrow = (int)(rank/ncol);
MPI_Comm_split(MPI_COMM_WORLD,myrow,rank,row_comm);
```

1. Communicator to split
2. Key – all processes with the same key go in the same communicator
3. Value to determine ordering in the result communicator (optional)
4. Result communicator
Subdividing Communicators: Approach #2

• The same goal can be accomplished using groups
• `MPI_Comm_group` – extract the group defined by a communicator
• `MPI_Group_incl` – make a new group from selected members of the existing group (e.g., members in the same row of a 2D layout)
• `MPI_Comm_create` – form a communicator based on this group
MPI_Group base_grp, grp; MPI_Comm row_comm, temp_comm; int row_list[NCOL], irow, myrank_in_world;

MPI_Comm_group(MPI_COMM_WORLD, &base_grp); //get base
MPI_Comm_rank(MPI_COMM_WORLD, &myrank_in_world);

irow = (myrank_in_world/NCOL);
for (i=0; i < NCOL; i++) row_list[i] = i;
for (i=0; i < NROW; i++){
    MPI_Group_incl(base_grp, NCOL, row_list, &grp);
    MPI_Comm_create(MPI_COMM_WORLD, grp, &temp_comm);
    if (irow == i) row_comm = temp_comm;
    for (j=0; j < NCOL; j++) row_list[j] += NCOL;
}
Communicators and Groups: Summary

- In **Approach #1**, we used `MPI_Comm_split` to split one communicator into multiple non-overlapping communicators.
- This approach is relatively compact and is suitable for regular decompositions.

- In **Approach #2**, we broke the communicator into (sub)groups and made these into new communicators to suit our needs.
- We did this using `MPI_Comm_group`, `MPI_Group_incl`, and `MPI_Comm_create`.
- This approach is quite flexible and is more generally applicable.
- A number of other group functions are available: union, intersection, difference, include, exclude, range-include, range-exclude.
5. Persistent Communication
Motivation: we’d like to save the argument list of an MPI call to reduce overhead for subsequent calls with the same arguments.

INIT takes the original argument list of a send or receive call and creates a persistent communication request from it:
- `MPI_Send_init` (for nonblocking send)
- `MPI_Bsend_init` (for buffered send – can do Rsend or Ssend as well)
- `MPI_Recv_init` (for nonblocking receive)

START starts an operation based on the communication request:
- `MPI_Start`
- `MPI_Startall`

REQUEST_FREE frees the persistent communication request:
- `MPI_Request_free`
Typical Situation Where Persistence Might Be Used

```c
MPI_Recv_init(buf1, count, type, src, tag, comm, &req[0]);
MPI_Send_init(buf2, count, type, src, tag, comm, &req[1]);

for (i=1; i < BIGNUM; i++)
{
    MPI_Start(&req[0]);
    MPI_Start(&req[1]);
    MPI_Waitall(2, req, status);
    do_work(buf1, buf2);
}

MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```
Performance Benefits from Using Persistence

Improvement in Wallclock Time (IBM SP2)
Persistent vs. Conventional Communication

<table>
<thead>
<tr>
<th>size, bytes</th>
<th>mode</th>
<th>improvement</th>
<th>mode</th>
<th>improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>async</td>
<td>19 %</td>
<td>sync</td>
<td>15 %</td>
</tr>
<tr>
<td>4096</td>
<td>async</td>
<td>11 %</td>
<td>sync</td>
<td>4.7 %</td>
</tr>
<tr>
<td>8192</td>
<td>async</td>
<td>5.9 %</td>
<td>sync</td>
<td>2.9 %</td>
</tr>
<tr>
<td>800,000</td>
<td>-</td>
<td>-</td>
<td>sync</td>
<td>0 %</td>
</tr>
<tr>
<td>8,000,000</td>
<td>-</td>
<td>-</td>
<td>sync</td>
<td>0 %</td>
</tr>
</tbody>
</table>

- **Takeaway**: it’s most effective when applied to lots of small messages
6. Parallel I/O (MPI-2)
What is Parallel I/O?

• HPC Parallel I/O occurs when:
  – multiple MPI tasks can read or write simultaneously,
  – from or to a single file,
  – in a parallel file system,
  – through the MPI-IO interface.

• A parallel file system works by:
  – appearing as a normal Unix file system, while
  – employing multiple I/O servers (usually) for high sustained throughput.

• Two common alternatives to parallel MPI-IO are:
  1. Rank 0 accesses a file; it gathers/scatters file data from/to other ranks.
  2. Each rank opens a separate file and does I/O to it independently.
Why Parallel I/O?

- I/O was lacking from the MPI-1 specification
- Due to need, it was defined independently, then subsumed into MPI-2
- HPC Parallel I/O requires some extra work, but it
  - potentially provides high throughput and
  - offers a single (unified) file for viz and pre/post processing.
- Alternative I/O schemes are simple enough to code, but have either
  - poor scalability (e.g., single task is a bottleneck) or
  - file management challenges (e.g., files must be collected from local disk).
- MPI-IO provides
  - mechanisms for performing synchronization,
  - syntax for data movement, and
  - means for defining noncontiguous data layout in a file (MPI datatypes).
Simple MPI-IO

Each MPI task reads/writes a single block:

P#  is a single processor with rank #.

P0  memory
P1  memory
P2  memory
   ...
P(n-1) memory
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);
include 'mpif.h'
integer status(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) offset

nints  = FILESIZE/(nprocs*INTSIZE)
offset = rank * nints * INTSIZE

call MPI_FILE_OPEN( MPI_COMM_WORLD, '/pfs/datafile', &
    MPI_MODE_RDONLY, &
    MPI_INFO_NULL, fh, ierr)
call MPI_FILE_READ_AT( fh, offset, buf, nints,
    MPI_INTEGER, status, ierr)
call MPI_FILE_CLOSE(fh, ierr)
Writing with Pointers and Offsets; Shared Pointers

• Use MPI_File_write or MPI_File_write_at
• MPI_File_open flags:
  – MPI_MODE_WRONLY (write only)
  – MPI_MODE_RDWR (read and write)
  – MPI_MODE_CREATE (create file if it doesn’t exist)
  – Use bitwise-or ‘|’ in C, or addition ‘+’ in Fortran, to combine multiple flags

Shared Pointers
• Create one implicitly-maintained pointer per collective file open
  – MPI_File_read_shared
  – MPI_File_write_shared
  – MPI_File_seek_shared
Noncontiguous Accesses

• Common in parallel applications
  – example: distributed arrays stored in files

• A big advantage of MPI I/O over Unix I/O is the ability to specify noncontiguous accesses in a file and a memory buffer
  – do this by using derived datatypes within a single MPI function call
  – allows implementation to optimize the access

• Collective I/O combined with noncontiguous accesses yields the highest performance
File Views

- A view is a triplet of arguments (displacement, etype, filetype) that is 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 layout of etypes within file
Example #1: File Views for a Four-Task Job

etype = MPI_DOUBLE_PRECISION  
  elementary datatype

filetype = myPattern  
  derived datatype, sees every 4\textsuperscript{th} DP

VIEW: each task repeats myPattern with different displacements

file

- task0
- task1
- task2
- task3

head of file
Example #2: File Views for a Four-Task Job

- 1 block from each task, written in task order

```
MPI_File_set_view assigns regions of the file to separate processes
```
Code for Example #2

```c
#define N 100
MPI_Datatype arraytype;
MPI_Offset disp;

disp = rank*sizeof(int)*N; etype = MPI_INT;
MPI_Type_contiguous(N, MPI_INT, &arraytype);
MPI_Type_commit(&arraytype);

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, arraytype,
                 "native", MPI_INFO_NULL);
MPI_File_write(fh, buf, N, etype, MPI_STATUS_IGNORE);
```
Example #3: File Views for a Four-Task Job

- 2 blocks from each task, written in round-robin fashion to a file

```c
MPI_File_set_view
```

assigns regions of the file to separate processes
int buf[NW*2];

MPI_File_open(MPI_COMM_WORLD, "/data2",
              MPI_MODE_RDWR, MPI_INFO_NULL, &fh);

/* want to see 2 blocks of NW ints, NW*npes apart */
MPI_Type_vector(2, NW, NW*npes, MPI_INT, &fileblk);
MPI_Type_commit(&fileblk);
disp = (MPI_Offset)rank*NW*sizeof(int);
MPI_File_set_view(fh, disp, MPI_INT, fileblk,
                  "native", MPI_INFO_NULL);

/* processor writes 2 'ablk', each with NW ints */
MPI_Type_contiguous(NW, MPI_INT, &ablk);
MPI_Type_commit(&ablk);
MPI_File_write(fh, (void *)buf, 2, ablk, &status);
Collective I/O in MPI

- A critical optimization in parallel I/O
- Allows communication of “big picture” to file system
- Framework for 2-phase I/O, in which communication precedes I/O (uses MPI machinery)
- Basic idea: build large blocks, so that reads/writes in I/O system will be large
MPI Routines for Collective I/O

• Typical routine names:
  - MPI_File_read_all
  - MPI_File_read_at_all, etc.

• The _all indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function.

• Each process provides nothing beyond its own access information; therefore, the argument list is the same as for the non-collective functions.
Advantages of Collective I/O

- By calling the collective I/O functions, the user allows an implementation to optimize the request based on the combined requests of all processes.

- The implementation can merge the requests of different processes and service the merged request efficiently.

- Particularly effective when the accesses of different processes are noncontiguous and interleaved.
Collective I/O: Memory Layout, Communication

Original memory layout on 4 processors

MPI collects in temporary buffers

then writes to File layout
More Advanced I/O

- Asynchronous I/O:
  - `iwrite/iread`
  - terminate with `MPI_Wait`

- Split operations:
  - `read_all_begin/end`
  - `write_all_begin/end`
  - give the system more chance to optimize
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);

Examples of Hints (Used in ROMIO)

- striping_unit
- striping_factor
- cb_buffer_size
- cb_nodes

- ind_rd_buffer_size
- ind_wr_buffer_size

- start_iodevice
- pfs_svr_buf
- direct_read
- direct_write

- MPI-2 predefined hints
- New algorithm parameters
- Platform-specific hints
Summary of Parallel I/O Issues

• MPI-IO has many features that can help users achieve high performance
• The most important of these features are:
  – the ability to specify noncontiguous accesses
  – the collective I/O functions
  – the ability to pass hints to the implementation
• Use is encouraged, because I/O is expensive!
• In particular, when accesses are noncontiguous, users must:
  – create derived datatypes
  – define file views
  – use the collective I/O functions
7. Status of MPI-2
Features of MPI-2

- Parallel I/O (MPI-IO) – probably the most popular
- One-sided communication (put / get)
- Dynamic process management (spawn)
- Expanded collective communication operations (e.g., non-blocking)
- Support for multithreading
- Additional support for programming languages
  - C++ interface
  - limited F90 support
  - interfaces for debuggers, profilers
MPI-2 Status Assessment

- Virtually all vendors offer MPI-1
  - Well-established free implementations (MPICH, OpenMPI) support networks of heterogeneous workstations, e.g.
  - The functionality of MPI-1 (or even a subset) is sufficient for most applications
- Partial MPI-2 implementations are available from most vendors
- MPI-2 implementations tend to appear piecemeal, with I/O first
  - MPI-IO now available in most MPI implementations
  - One-sided communication available in some
  - OpenMPI (aka LAM) and MPICH2 now becoming complete
  - Dynamic process management may not mesh well with batch systems
References

- **Index to the MPI 1.1 standard**
- **Index to the MPI 2 standard**
- **The I/O Stress Benchmark Codes**