An Introduction to MPI

Brock Palen
brockp@umich.edu

May 24, 2011

Outline

1. **Overview**
   - What is MPI?
   - Alternatives

2. **Hardware**
   - Networks

3. **Code**
   - Mechanics
   - Utility Functions
   - Point to Point
   - Collectives

4. **Summary**
   - Advanced MPI
   - Documentation
   - Contact

MPI

- MPI - Message Passing Interface
- C/C++/F77/F90 Bindings
- Distributed Memory Programming
- Series of Functions

Assumptions

- All examples work on nyx-login.engin.umich.edu
- Used with OpenMPI built with the PGI Compilers

3. man MPI_Send for prototypes
**MPI’s Flow**

MPI-All processes (normally one CPU per process but could be more) execute the same code. These processes only know ‘who they are’ (their Rank). These ranks then have to explicitly pass data. This passing follows the form:

1. Process 1 ‘Send to process 2’
2. Process 2 ‘Recv from process 1’

Others options Collectives:

- All process in a single Communicator, 'ALL DO THIS'

---

**OpenMP**

OpenMP http://www.openmp.org (Not to be confused with OpenMPI) is implemented in the compiler. Each compiler has its own issues with their own implementation of OpenMP but most users should find no problems with the basics.

- Anything that usesOMP_NUM_THREADS uses OpenMP
- PGI Compilers: pgf90 -mp openmp.f90
- Intel Compilers: ifort -openmp openmp.f90
Other Options

- HPF-High Performance Fortran – Use OpenMP/MPI
- pthreads – Use OpenMP
- PVM – Dead, Not installed, Use MPI
- CoArray
- Shmem – Use MPI_Get() and MPI_Put()

Network Types

Networks Matter

Networks have a huge impact on performance of MPI functions. Most MPI libraries abstract networks away and chose the fastest network found for a given communication.

Network Examples

- Ethernet 1Gbps 40 µsec
- Shared Memory varies 200nanosec-1µsec
- Myricom MX 10Gbps 5 µsec
- QDR Infiniband 40Gbps 1.5 µsec
- Comcast 0.006Gbps

MPI abstracts out how the data moves, understanding over what your data is moved is highly important. Examples while every machine comes with an Ethernet network port the performance is poor. Infiniband and Myricom have high costs, but provide huge performance gains. Significant gains using fast networks can be made in the number of cores an MPI application can scale to. The better networks also support Operating System Bypass, a method of using pinned memory to copy MPI data directly into the receive buffer of an MPI call. Ethernet requires the TCP stack of the OS Kernel, and thus requires extra buffer copies, and consumes extra CPU resources in this copy.
MPI Mechanics

Parts of an MPI Message
- Address of data, called a buffer
- MPI Datatype
- Count, or number of MPI datatypes to message
- Tag, Used to separate messages from the same processor
- Communicator
- Rank of target

MPI uses the buffer address, datatype and count to find how much raw data to send. For example, if MPI_DOUBLE is 8 bytes, and count is 10, MPI 'knows' to send 80 bytes (10*8 bytes) to the target process.

Note that std::vector is not a valid buffer for MPI! (or any STL class). The STL does not make sure that all memory in continuous. This breaks MPI, remember all buffers must be well defined.

A few other advanced methods are around that can take the place of MPI_Init() but most users will never used them, nor should they use them.
These functions can be called as many times as needed. Ranks that belong to more than one communicator will have a unique rank in each communicator, thus two calls to MPI_Comm_rank() would be needed for each communicator to find its rank in each. Repeat calls should not be needed.

A process’s rank in a given communicator will never change. Also the user is not able to adjust a process’s rank. There are methods for control over the placement of ranks on cpus on the cluster that are given to `mpirun`.

---

### Communicators and Tags

**Communicators**
- Specifies a group of processes that can communicate together
- `OMPI_COMM_WORLD`
- Created at startup includes all processes started by the launcher

**Tags**
- Adds distinct meaning to a message
- Very Useful

---

- **Communicators**
- There can be more than one communicator
- They can be duplicated `MPI_Comm_dup()`
- They can be created `MPI_Comm_create()`
- They can have physical meaning `MPI_Comm_cart_create()`

- **Tags**
- `MPI_ANY_TAG`
- Information on tag available in `MPI_Status`
**MPI_DATATYPE**

**C**
- MPI_CHAR char
- MPI_INT int
- MPI_FLOAT float
- MPI_DOUBLE double
- MPI_Pack() struct

**Fortran**
- MPI_CHARACTER CHARACTER
- MPI_INTEGER INTEGER
- MPI_REAL REAL
- MPI_DOUBLE PRECISION DOUBLE
- MPI_COMPLEX COMPLEX
- MPI_DOUBLE_COMPLEX DOUBLE_COMPLEX

- Unsigned and shorts also available
- Fortran can use MPI_REAL8 MPI_INTEGER2 etc.
- C structs must be 'packed' with MPI_Pack()

---

**MPI_Send()**

**C**
- MPI_Send( void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)

**Fortran**
- MPI_SEND( <type> BUF, INTEGER COUNT, MPI_Datatype TYPE, INTEGER DEST, INTEGER TAG, MPI_Comm COMM, INTEGER IERROR)

These are called 'blocking' sends. Note that MPI does not require that `MPI_Send()` block, just that it not return until buf is safe to use again. This can cause deadlocks to appear in code when scaled up. See the example included with this.
MPI_Recv()

**C**

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

**Fortran**

```fortran
MPI_RECV( <type> BUF,
          INTEGER COUNT,
          MPI_Datatype TYPE,
          INTEGER SOURCE,
          INTEGER TAG,
          MPI_Comm COMM,
          INTEGER STATUS,
          INTEGER IERROR)
```

In Fortran STATUS must be: integer status(MPI_STATUS_SIZE) The same issues apply here as MPI send. For every MPI_Send() there must be a MPI_Recv(). Most codes will put most of their communication time waiting for Recv's to actually get their data. If there is no Send, Recv will wait forever. See MPI_Irecv() for non-blocking.

First MPI Program

cac-mpi-code.tar.gz includes a Makefile all examples can be built by running make or built one at a time, make chello will build the helloworld.c example. While make fapps will build all Fortran examples. See README included for information.
Deadlocks

### Deadlock

- Every MPI\_Send() must have a matching MPI\_Recv()
- A call that does not have a matching Send or Recv is deadlocked
- Calls do not return unless buffer is safe to reuse, not that it was received
- Some MPI libraries will let deadlocked code run until messages reach a given size

### cdeadlock

- make cdeadlock no-cdeadlock
- mpirun -np 4 cdeadlock
- mpirun -np 4 no-cdeadlock

---

This example will demonstrate the effects of deadlock, and eager messages.

cdeadlock and no-cdeadlock is the same code, the change is in the size of buffer when buffer falls below some value eager messaging takes place. In this case the MPI library says the message is small enough, lets allocate memory copy the buffer to it and return right away, so the buffer is now safe to reuse allowing the code to continue. In the cdeadlock/fdeadlock case these buffers are to large and MPI will not copy the code will block at MPI\_Send() for forever.

---

### Deadlock Example

- Each process does a MPI\_Send() to rank+1
- Each process then calls MPI\_Recv() from rank-1
- make cdeadlock or make fdeadlock
- mpirun -np 4 cdeadlock
- make no-cdeadlock or make no-fdeadlock

As of openmpi/1.3.2-pgi the ompi\_info --param btl sm btl\_sm\_eager\_limit is 4096 bytes. The MPI envilope is 64bytes, or the same size as 8 doubles, this lets 503 doubles be the max that can be sent over the sm btl before locking up.

504doubles*8bytes/double+64bytes=4096bytes.
**MPI_Sendrecv()**

- Swaps data between two ranks.
- Much simpler than Non-blocking functions
- Much simpler than trying to inter-mix Sends and Recvs

**Non-Blocking Functions**

- Example non-blocking.c/non-blocking.f90
- Can mix blocking with non-blocking
- MPI_Isend() -> MPI_Recv()
- Uses MPI_Request objects
- MPI_Test() MPI_Wait() etc.

**MPI_Isend() MPI_Irecv()**

- Example non-blocking.c/non-blocking.f90
- Can mix blocking with non-blocking
- MPI_Isend() -> MPI_Recv()
- Uses MPI_Request objects
- MPI_Test() MPI_Wait() etc.

**MPI_Sendrecv()**

- Swaps data between two ranks.
- Much simpler than Non-blocking functions
- Much simpler than trying to inter-mix Sends and Recvs

**Non-Blocking Functions**

- Example non-blocking.c/non-blocking.f90
- Can mix blocking with non-blocking
- MPI_Isend() -> MPI_Recv()
- Uses MPI_Request objects
- MPI_Test() MPI_Wait() etc.

**MPI_Wait()** will block waiting for the send or recv to complete.
**MPI_Test()** returns if a message has completed. You can not touch the buffer once it is used in a request. Once the request has been completed by checking with **MPI_Wait()** or **MPI_Test()** the buffer may be reused.

- **MPI_Testany() MPI_Waitany()**
- These calls have overhead the blocking calls don’t
- The overhead of complicated programming
- Overhead for more complicated communication may outweigh benefits
Collectives

Collectives work on an entire communicator of processes
- MPI_Barrier() Blocks till all ranks reach a single point
- MPI_Bcast() Send data from one rank to all others
- MPI_Reduce() Do an operation on data on all processors and store result on a single rank
- MPI_Scatter() Scatter data from one CPU to many
- MPI_Gather() Gather data and put on a single rank

All these collective functions have a 'All' version (MPI_Allreduce()) which has the results end on all CPUs. For example MPI_Allreduce() puts the results of the reducing operation on all ranks not just a single rank.

Other functions to look at:
- MPI_Allreduce()
- MPI_Allscatter()
- MPI_Allgather()
- MPI_Alltoall()

root is the rank you wish to broadcast data from. Root must be specified and the same on all ranks in comm.
If you need every rank to broadcast its data to every other process (A very inefficient operation) the function is MPI_Alltoall()
**MPIReduce()**

C

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

Fortran

```fortran
MPI_REDUCE(<type> SENDBUF,
            <type> RECVBUF,
            INTEGER COUNT,
            MPI_Datatype TYPE,
            MPI_Op OP,
            INTEGER ROOT,
            MPI_Comm COMM,
            INTEGER IERROR)
```

- **root** is the rank that the results of **op** should go to
- **recvbuf** does need to be defined on all ranks

**MPI Operations**

- **LAND**, **LOR**, **LXOR**: Logical AND, Logical OR, Logical XOR.
- **BAND**, **BOR**, **BXOR**: Bitwise AND, Bitwise OR, Bitwise XOR.

Use `MPI_Reduce_scatter()` to reduce the buffers with **MPI_Op** and then scatter (to be covered) the results to all ranks. Rather than using two calls.
Overview

Hardware

Code

Summary

---

**fpi.f90**

**Calculate pi**

\[ \int_0^1 \frac{4}{1 + x^2} \, dx = \pi \]

- Break into \( n \) rectangles
- Have each rank work on a subset of these
- Find a global sum from partial sums

**fpi.f90**

make fpi

mpirun -np 4 fpi

---

**MPI_Scatter()** **MPI_Gather()**

- MPI_Scatter()
- MPI_Gather() is the reverse

MPI_Scatter() takes data on a single rank breaks it evenly among all the ranks leaving each rank with a small portion of the entire set of data in its output buffer.

It is commonly useful to combine MPI_Scatter() with MPI_Reduce(). MPI defines a function that does this for you: MPI_Reduce_scatter(). This is different than MPI_Allreduce() because the reduced data is not cloned across all ranks but broken in pieces across all ranks.

Because sometimes each rank is not to get the same amount of data from a Scatter or Gather MPI defines: MPI_Scatterv(), MPI_Gatherv() and MPI_Allgatherv(). These calls take an extra displs option that describes how much data each rank should receive.

See the man pages for these calls.
MPI::Allgather() collects distributed data like MPI_Gather() but then distributes this data to all ranks in the communicator. This behavior could be simulated with MPI_Gather() followed by MPI_Bcast(). As always use the single collective not the two latter together.

Warnings with Collectives

While I point out to avoid these calls its because of the communication complexity. For example MPI_Alltoall() in its worse form is:

\[ O(n^2 - n) \]

You should not try to mimic its behavior by mixing other calls. What should be done is find a algorithm that does not require such a exchange. If this is not the case do make sure to use MPI_Alltoall(). This applies to all other mentioned calls.

MPI_Barrier() gets abused. Its use normally is not needed. Because most other MPI calls block such synchronization is implied. Using a barrier is really only good for timing with MPI_Wtime().
**Advanced Calls**

**Topology**
- MPI_Cart_create() creates a communicator with Cartesian topology.

**Timing**
- MPI_Wtime() Time in seconds from arbitrary time in the past.
- MPI_Wtick() Returns resolution of MPI_Wtime()

---

**Documentation**

Use the man pages on the CAC systems.

- http://ci-tutor.ncsa.uiuc.edu
- Google/Yahoo Etc.
- CoE EECS 587
- "Using MPI" – Gropp, Lusk, and Skjellum
- "Parallel programming in C with MPI and OpenMP” – Quinn
- "Parallel scientific computing in C++ and MPI” – Karniadakis, Kirby

---

**Advanced Calls**

MPI_Wtime() does not start from zero. They start from some arbitrary time in the past. So two calls to MPI_Wtime() is normally needed to get a time result.

MPI_Barrier() needs to be used often around calls to MPI_Wtime() to get accurate times around single portions of code.

---

**Documentation**

Many books are available in Mirlyn:

- "Parallel programming in C with MPI and OpenMP" QA 76.73 .C15 Q551 2004
- "Parallel scientific computing in C++ and MPI" QA 76.58 .K371 2003
- “Best Online Tutorial: http://ci-tutor.ncsa.uiuc.edu/
Subscribe to our website’s RSS feed for listings of new software, courses and machine outages.

- FFT’s FFTW. http://www.fftw.org/