MPI C++ on the HPC

Andrew Peterson

Big Data in Finance
Baruch Master in Financial Engineering

February 10, 2016

Outline

- Quick Overview of MPI
- MPI on the CUNY cluster
- Hands-on Practice
- (if time) additional MPI functions, examples

Materials

Materials for this tutorial are at:

`https://github.com/aristotle-tek/cuny-bdif/MPI`

You can clone the files using the pbs script '`gitclone.pbs`'.

Different ways to parallelize

- HPC batch jobs (manual data split & recombine)
- Threading
- MPI (flexible & powerful)

Multithread vs Multiprocessor

- Threads - shared memory
- MPI - better if multiple data (SIMD/MIMD)
- (Can also use threading within an MPI job)

Pitfalls of Parallel Computing

- data racing (to update the same data)
- indefinite postponement (waiting on possibly failed task)
- deadlock (both waiting on same locked resource)
- communication problems (diff operating systems, etc)

(Hughes & Hughes, 2004, ch.2)

MPI

- Message passing standard, with C, C++*, Fortran bindings.
- Support for parallel I/O, remote memory, threads, etc.

Compiling MPI (GNU/Intel)

Link & compile C:

```
mpicc -o foo foo.c
```

Link & compile C++ :

```
mpic++ -o foo foo.cpp
```

Execute with 16 processors:

```
mpirun -np 16 foo
```

HPC: Access

- ssh <user.name>@chizen.csi.cuny.edu (gateway)
- ssh <user.name>@penzias.csi.cuny.edu
- windows: WinSCP or PuTTY
- transferring files: scp / sftp

HPC: Workspaces

- /global/u/ (backed up, 50GB)
- /scratch/ (temporary)
- SR1 (long-tern storage resource, iRODS commands)

HPC: Submitting PBS jobs

- 3rd party software: module (avail | list | purge | load)
- Submit jobs to the queue using PBS (bash) scripts

PBS scripts

- specify hardware requirements, walltime
- normal bash scripting

PBS script example

```
#!/bin/bash
#
# Run a distributed memory MPI job in the
production queue requesting 16 chunks each with one 1 cpu.
#
#PBS -q production
#PBS -N <job_name>
#PBS -l select=16:ncpus=1
#PBS -l place=free
#PBS -V

# Change to working directory (The PBS_O_WORKDIR var is automatically
# the path to the directory you submit your job from)
cd $PBS_O_WORKDIR

# Run my 16-core MPI job
mpirun -np 16 </path/to/your_binary> > <my_output> 2>&1
```

(http://www.csi.cuny.edu/cunyhpc/pdf/User_Manual.pdf)

Allocating nodes for MPI jobs

- latency versus quick pbs queuing tradeoff:
- requesting more cpus together ("ncpus=8") makes for lower latency but slower to be queued, since pbs has to wait to find all 8 free together. So generally better to do, e.g.:

```
#PBS -l place=free
#PBS -l select=32:ncpus=2
```

to get 32 chunks of 2 cores each = 64 total.

PBS Management

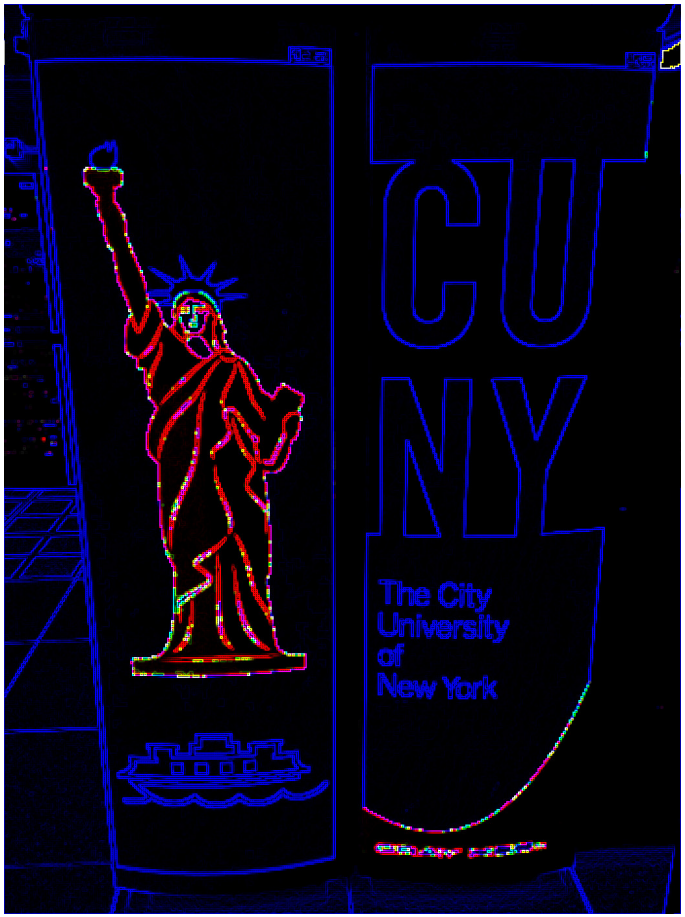
- (from the login node)
- submit to queue: `qsub foo.pbs`
- status: `qstat -u <username>`
- delete job: `qdel <processid>`

PBS Queues

- `production` - normal
- `development` - higher priority, only for testing.
  max 8 cores, 1hr CPU time.
- `interactive` - quick interactive tests.
  max 4 cores, 15 min CPU time

  `(or: qsub -I -l nodes=1:ppn=2 -l mem=1GB -l walltime=0:05:00)`

Hands on practice: hello world



MPI: Basic Functions

1. `MPI_Init()`
2. `MPI_Finalize()`
3. `MPI_Get_size()` –number of processes
4. `MPI_Get_rank()` –ID for that specific process

C version

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

int main(int argc, char **argv)
{
   int id;
   int worldsize;
   MPI_Init (&argc, &argv); /* starts MPI */
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &worldsize);

   if ( id == 0 )
   {
      printf("This will only be printed by process 0.\n");
   }

   printf( "Hello from process %d of %d\n", id, worldsize);
   MPI_Finalize();
   return 0;
}
```

C++ version

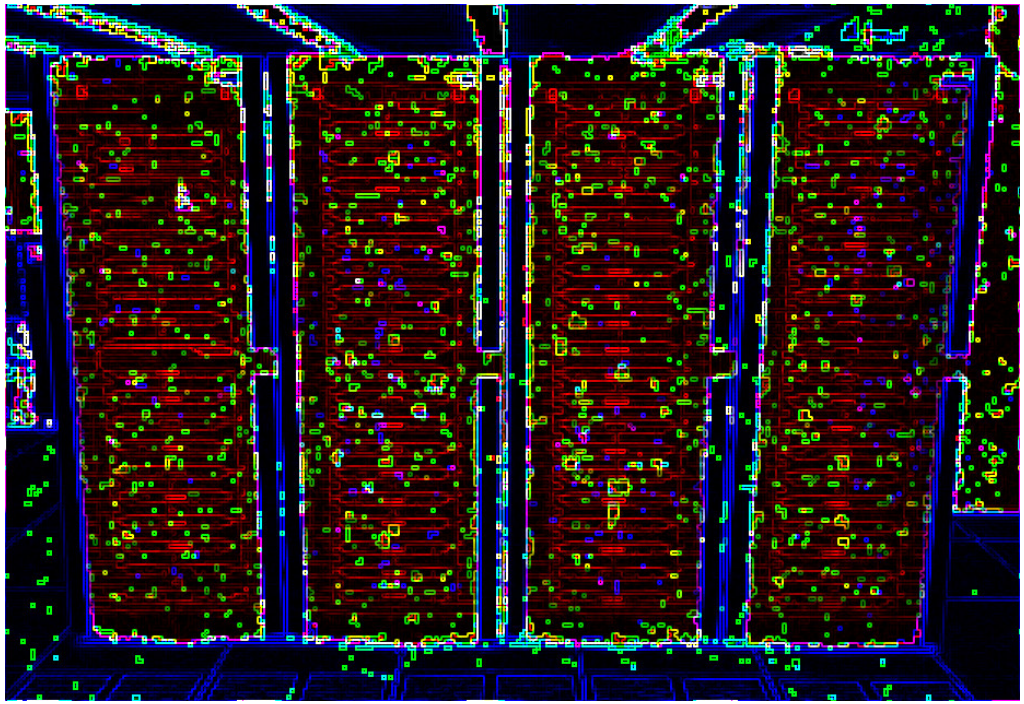
```
# include <cstdlib>
# include "mpi.h"
using namespace std;

int main ( int argc, char *argv[] )
{
   int id;
   int worldsize;

   MPI::Init ( argc, argv );

   worldsize = MPI::COMM_WORLD.Get_size ( );
   id = MPI::COMM_WORLD.Get_rank ( );

   if ( id == 0 )
   {
      cout << "This will only be printed by process 0.\n";
      cout << "The number of processes involved is: " << worldsize << "\n";
      cout << "\n";
   }

   cout << "  Hello from process " << id << ". \n";

   MPI::Finalize ( );
   return 0;
}
```

Compiling

Can compile from the login node:

```
mpicc -o hello hellos.c
or
mpic++ -o hello hellos.cpp
```

MPI: Additional Info & Examples



MPI: Additional Info & Examples

1. sending & receiving messages
2. parallel data I/O
3. additional resources

Sending & Receiving Messages

- `MPI_Send()` & `MPI_Recv()`
- **blocking** – don't return until finished
  (non blocking versions: `MPI_Isend`, `MPI_Irecv`)
- **message buffer** – the initial address of send buffer
- **count** – the number of elements send (nonneg int)
- target process
- communicator

Sending & Receiving Messages

```
› MPI_Send(const void *buf,
           int count,
           MPI_Datatype datatype,
           int dest,
           int tag,
           MPI_Comm communicator)
```

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

MPI Datatypes

```
MPI_SHORT, MPI_INT, MPI_LONG,
MPI_LONG_LONG, MPI_UNSIGNED_CHAR,
MPI_UNSIGNED_SHORT, MPI_UNSIGNED,
MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG,
MPI_FLOAT,
MPI_DOUBLE, MPI_LONG_DOUBLE ,
MPI_BYTE
```

Ping-Pong example

```
[...]
int ping_pong_count = 0;
int partner_rank = (world_rank + 1) % 2;
while (ping_pong_count < PING_PONG_LIMIT) {
  if (world_rank == ping_pong_count % 2) {
  // Increment the ping pong count before you send it
  ping_pong_count++;
  MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD);
  printf("%d sent and incremented ping_pong_count %d to %d\n",
     world_rank, ping_pong_count, partner_rank);
  } else {
    MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
    printf("%d received ping_pong_count %d from %d\n",
      world_rank, ping_pong_count, partner_rank);
}
```

Source: Wes Kendall www.mpitutorial.com

Ping-Pong example

output:

```
0 sent and incremented ping_pong_count 1 to 1
0 received ping_pong_count 2 from 1
0 sent and incremented ping_pong_count 3 to 1
0 received ping_pong_count 4 from 1
[...]
1 received ping_pong_count 1 from 0
1 sent and incremented ping_pong_count 2 to 0
1 received ping_pong_count 3 from 0
1 sent and incremented ping_pong_count 4 to 0
1 received ping_pong_count 5 from 0
[...]
```

MPI & Parallel I/O

- concurrent reads/writes to a common file
  (MPI + Lustre common file system)

MPI & Parallel I/O

Different methods for positioning read/write offsets:

- Individual file pointers (`MPI_file_seek`/ `MPI_file_read`)
- Calculate byte offsets (`MPI_File_read_at`)
- Access a shared pointer (`MPI_file_seek_shared`...)

Parallel read process simplified

1. Determine chunk size from number of processes
   `MPI_Comm_size()`
2. open the file into MPI communicator
   `MPI_File_open()`
3. each process will read a chunk, differentiated by rank
   ```
   MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf,
    int count, MPI_Datatype datatype, MPI_Status *status)
   ```
4. close file
   `MPI_File_close()`

Parallel read example

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 3200

int main(int argc, char **argv){
    int rank, size, bufsize, nints;
    MPI_File fh;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    bufsize = FILESIZE/size;
    nints = bufsize/sizeof(int);
    int buf[nints];
    MPI_File_open(MPI_COMM_WORLD,"binaryfile",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh)
    MPI_File_read_at(fh, rank*bufsize, buf, nints, MPI_INT, &status);
    printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
    MPI_File_close(&fh);
    MPI_Finalize();
    return 0;
}
```

MPI & Parallel I/O

- `MPI_File_write_at_all()`
- "all" $\rightarrow$ collective function - all processes will call this
- forces simultaneous read/write

Construct a view

› `MPI_File_set_view` – specifies how different processes will use the same file, so as to have different views & not overlap.

```
(MPI_File fh, MPI_Offset displacement,
 MPI_Datatype etype, MPI_Datatype filetype,
 char *datarep, MPI_Info info);
```

Data representations

- “native" – written as it is stored in memory
- “internal" – written in an MPI-implementation-dependent (OS, architecture-independent)
- “external32" – written in the “big-endian IEEE” format (OS, architecture-independent)

Reminder

- CUNY HPC should not be used for commercial purposes
- Research using the HPC should provide citation as indicated in the user guide

Additional Resources

- CUNY HPC user guide
- MPI documentation: `http://www.mpi-forum.org/`
- John Burkhardt – slides, examples:
  `http://people.sc.fsu.edu/~jburkardt/`
- Data I/O slides:
  `https://www.tacc.utexas.edu/documents/13601/900558/MPI-IO-Final.pdf`
- Hughes & Hughes. 2004. Parallel and Distributed Programming Using C++.