Introduction to MPI

SHARCNET MPI Lecture Series: Part I of II

Paul Preney, OCT, M.Sc., B.Ed., B.Sc.

preney@sharcnet.ca

School of Computer Science
University of Windsor
Windsor, Ontario, Canada

November 11, 2015

SHARCNET™

Presentation Overview

1. Overview
2. An MPI “Hello World!”
3. Sending and Receiving Messages
4. Demonstrations & Questions
5. References

SHARCNET™

Table of Contents

1. Overview
   - What is MPI?
   - Choices Other Than MPI?
   - Some Terms
   - The MPI Parallel Programming Model
   - Important Parallel Programming Goals

SHARCNET™

What is MPI?

Message-**P**assing **I**nterface:

- is a *de facto* standard dating back to 1994. [1]
- is one of many ways to write code that executes in parallel
- has language bindings for Fortran and C.
  - **NOTE:** The C++ language bindings were removed in MPI v3.0. [2, §16.2, p.596]
- enables compute nodes to efficiently pass "messages" to one another.

SHARCNET™

Choices Other Than MPI?

If you don't need to use MPI you may want to also consider (each with its own pros and cons):

- OpenCL (CPUs and/or GPUs)
  - Devices can be CPUs or GPUs, OpenCL C code is deployed to each device.
  - Can achieve high performance but more difficult to program in.
  - OpenCL C code is not the same as traditional C, C++, or Fortran code.
- GPU-specific
  - OpenACC (open), CUDA (proprietary to NVIDIA)
  - More difficult to program.
  - Code is not the same as traditional C, C++, or Fortran code.

SHARCNET™

Choices Other Than MPI? (con't)

- OpenMP (CPUs)
  - Already have C, C++, Fortran code? OpenMP makes it relatively easily to convert specific constructs (e.g., for loops) to use threads.
  - Normally run on CPUs.
- pthreads, C11, or C++11 threads (CPUs)
  - Traditional C, C++, or Fortran code using threads.
  - Code will be portable, but, is only deployed on CPUs.

If you have have questions or require assistance choosing an appropriate technology, please do ask! :-)

SHARCNET™

Some Terms

distributed	Resources that are spread out over a number of compute nodes (which may or may not be separate computers).
message	A POD or an array of PODs.
POD	"plain-old data", i.e., C-style data types but avoid pointers. No data or function pointers, object-oriented types using inheritance or containing virtual functions (e.g., roughly in C++ terms nothing can be virtual).
SIMD	Single-Instruction, Multiple Data (streams).
UE	Unit of execution (e.g., an MPI process).

SHARCNET™

The MPI Parallel Programming Model

- Memory in MPI is assumed to be distributed and not shared.
  - This means you cannot access data in a UE without the UE sending it as a "message" back to you.
- UEs in MPI are processes and should be assumed to be such.
  - MPI v2 has the ability to run each MPI process as a thread —but the MPI software must be configured to run this way by the sysadmin. So you must assume all memory is not shared.
- Sending messages to/from MPI processes is not free.
  - If you need to send 10 MB messages to 10 nodes from one node, the sender will need to transmit 100 MB of data. No matter how fast the interconnects are, it takes time to transmit and receive these messages!

SHARCNET™

Important Parallel Programming Goals

Once you become comfortable with parallel programming, you will find yourself adopting these goals:

- Creating all data structures / objects so that they are PODs or easily converted into a POD.
  - Translation: In C++ terms, avoid using object-oriented virtual inheritance or virtual functions.
- The preferred "container" data structure to hold multiple PODs is an string, array, `std::string` (C++), `std::array` (C++), or `std::vector` (C++).
  - If you use almost anything else, you will need to convert it to a POD / array of PODs, or, you will need to write advanced code to map your object so that it can be sent/received as a message.

SHARCNET™

Important Parallel Programming Goals (con't)

- **Think** about what you want to do and **how** you will go about doing it **before** writing any code!
  - It is **much harder** to write a correct parallel program than to write a correct sequential ("serial") program.
  - It is **much harder** to debug an incorrect parallel program than to debug an incorrect sequential program.

Adopt the noble goal of trying to **design** a correct solution before writing any code, then write it. This will greatly **reduce** the difficulty of writing the code considerably.

SHARCNET™

Table of Contents

2 An MPI "Hello World!"

- Hello Word: MPI Calls
- Hello World: Sample Output
- Hello World: The Programs
- Hello World: Compiling
- Hello World: Running
- Hello World: Using SHARCNET's sqsub

Hello Word: MPI Calls

MPI Call	Purpose
MPI_INIT	Required to start up and use MPI functionality.
MPI_COMM_RANK	The ID of the current MPI process running.
MPI_COMM_SIZE	The total number of MPI processes running.
MPI_FINALIZE	Required to stop using and shut down MPI.

SHARCNET™

Hello World: Sample Output

Sample output from the "Hello World!" programs that will be presented:

```
1  $ mpirun -n 4 ./helloworld2-cxx.exe
2  Hello world! I am process 1 out of 4!
3  Hello world! I am process 0 out of 4!
4  Hello world! I am process 2 out of 4!
5  Hello world! I am process 3 out of 4!
6  $ mpirun -n 4 ./helloworld2-cxx.exe
7  Hello world! I am process 1 out of 4!
8  Hello world! I am process 2 out of 4!
9  Hello world! I am process 3 out of 4!
10 Hello world! I am process 0 out of 4!
11 $
```

NOTE: The lines output are not in the same order!

SHARCNET™

Hello World: The Programs

An MPI "Hello World!" program in Fortran-90:

```
1  program helloworld
2  use mpi
3  integer ierr, numprocs, procid
4
5  call MPI_INIT(ierr)
6
7  call MPI_COMM_RANK(MPI_COMM_WORLD, procid, ierr)
8  call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
9
10 print *, "Hello world! I am process ", procid, " out of ",
       numprocs, "!"
11
12 call MPI_FINALIZE(ierr)
13
14 stop
15 end
```

SHARCNET™

Hello World: The Programs (con't)

An MPI "Hello World!" program in C:

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

int main(int argc, char *argv[])
{
  int ierr, procid, numprocs;

  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &procid);
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  printf("Hello␣world!␣I␣am␣process␣%d␣out␣of␣%d!\n", procid,
       numprocs);

  ierr = MPI_Finalize();
}
```

SHARCNET™

Hello World: The Programs (con't)

An MPI "Hello World!" program in C++ using `std::printf()`:

```
1  #include <cstdio>
2  #include <mpi.h>
3
4  int main(int argc, char *argv[])
5  {
6    using namespace std;
7
8    int ierr = MPI_Init(&argc, &argv);
9    int procid, numprocs;
10   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &procid);
11   ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
12
13   printf("Hello␣world!␣I␣am␣process␣%d␣out␣of␣%d!\n", procid,
         numprocs);
14
15   ierr = MPI_Finalize();
16 }
```

SHARCNET™

Hello World: The Programs (con't)

An MPI "Hello World!" program in C++ using `std::cout`:

```
1  #include <iostream>
2  #include <mpi.h>
3
4  int main(int argc, char *argv[])
5  {
6    using namespace std;
7
8    int ierr = MPI_Init(&argc, &argv);
9    int procid, numprocs;
10   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &procid);
11   ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
12
13   cout << "Hello␣world!␣I␣am␣process␣" << procid << "␣out␣of␣" <<
           numprocs << "!\n";
14
15   ierr = MPI_Finalize();
16 }
```

Hello World: Compiling

To compile the code:

- C: `mpicc -o prog.exe helloworld.c`
- C++: `mpic++ -o prog.exe helloworld.cxx`
- Fortran: `mpifort -o prog.exe helloworld.f90`

With final programs, remember to add compile with optimizations:

- C: `mpicc -o prog.exe -O2 helloworld.c`
- C++: `mpic++ -o prog.exe -O2 helloworld.cxx`
- Fortran: `mpifort -o prog.exe -O2 helloworld.f90`

SHARCNET™

Hello World: Running

To run an MPI program locally on **your own computer** or a SHARCNET **development node** run:

```
mpirun -n NCPUS ./prog.exe
```

where `NCPUS` is the number of available CPUs/cores.
e.g.,

- 4 cores: `mpirun -n 4 ./prog.exe`
- 16 cores: `mpirun -n 16 ./prog.exe`

SHARCNET™

Hello World: Using SHARCNET's sqsub

- SHARCNET collections of computers (i.e., nodes) placed in groups called clusters.
  - e.g., kraken.sharcnet.ca, orca.sharcnet.ca, saw.sharcnet.ca
- When you want to run a job on a cluster, you will:
  - Log in to that cluster's login node.
  - Submit the job to the **mpi** queue.
  - You job will normally not run right away so you'll need to wait until it has finished running.

SHARCNET™

Hello World: Using SHARCNET's sqsub (con't)

The syntax required to submit `prog.exe` to `sqsub` is:

```
sqsub -q mpi -n NCPUS -o output.log -r TIME --mpp RAM ./prog.exe
```

where:

- `NCPUS` is the number of cores
- `output.log` is where all screen (stdout, stderr) output is written
- `TIME` is a number (representing the maximum time to run) followed by a time unit ('m' is minutes, 'h' is hours, and 'd' is days)
- `RAM` is a number (representing the maximum amount of RAM per MPI process) followed by a unit ('M' is megabytes, 'G' is gigabytes)

SHARCNET™

Hello World: Using SHARCNET's sqsub (con't)

For example, to place "Hello World!" in a queue to run on SHARCNET one would use:

```
sqsub -q mpi -n 4 -o out.log -r 5m --mpp 200M ./helloworld.exe
```

SHARCNET™

Table of Contents

3 Sending and Receiving Messages
- Overview
- MPI_Send
- MPI_Recv
- MPI_Datatype
- Example 1: MPI_SEND & MPI_RECV
- Example 2: MPI_SEND & MPI_RECV
- Example 3: MPI_SEND & MPI_RECV

SHARCNET™

Overview

The simplest way to communicate point-to-point messages between two MPI processes is to use:

- `MPI_Send()` to send messages.
- `MPI_Recv()` to receive messages.

SHARCNET™

Overview (con't)

One must specify:

- the datatype being sent/received,
- the receiver's process ID when sending,
- the sender's process ID (or `MPI_ANY_SOURCE`) when receiving, and,
- the sender's tag ID (or `MPI_ANY_TAG`) when receiving.

In order to receive a message, MPI requires the type, process id, and the tag match.

If they don't match, it the receive call will wait forever —hanging your program.

SHARCNET™

MPI_Send

`MPI_Send(buf,count,type,dest,tag,comm)` is a blocking send operation whose arguments are defined as follows: [2, §3]

Argument	In/Out	Description
buf	IN	starting address of send buffer
count	IN	number of elements in send buffer
type	IN	`MPI_Datatype` of each send buffer element
dest	IN	node rank id to send the buffer to
tag	IN	message tag
comm	IN	communicator

When called, `MPI_Send` transmits `count` elements in `buf` all of type `type` to node `dest` with the label `tag`.

The buffer is assumed to have been sent after the call returns.

SHARCNET™

MPI_Recv

`MPI_Recv(buf,count,type,src,tag,comm,status)` is a blocking receive operation whose arguments are defined as follows: [2, §3]

Argument	In/Out	Description
buf	OUT	starting address of receive buffer
count	IN	number of elements in receive buffer
type	IN	`MPI_Datatype` of each buffer element
src	IN	node rank id to receive the buffer from
tag	IN	message tag
comm	IN	communicator
status	OUT	status object

When called, `MPI_Recv` receives up to `count` elements in `buf` all of type `type` from node `src` with the label `tag`.

Up to `count` buffer elements can be stored.

SHARCNET™

MPI_Datatype

MPI uses instances of a special type called `MPI_Datatype` to represent the types of messages being sent or received.

The MPI standard defines a set of predefined `MPI_Datatypes` that map to C's fundamental types. Some of these mappings are: [2, §3.2]

MPI_Datatype Name	C Type
`MPI_C_BOOL`	`_Bool`
`MPI_CHAR`	`char` (text)
`MPI_UNSIGNED_CHAR`	`unsigned char` (integer)
`MPI_SIGNED_CHAR`	`signed char` (integer)
`MPI_INT`	`signed int`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_C_DOUBLE_COMPLEX`	`double _Complex`

SHARCNET™

MPI_Datatype (con't)

Mappings to Fortran types are defined as follows: [2, §3.2]

MPI_Datatype Name	Fortran Type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)

SHARCNET™

Example 1: MPI_SEND & MPI_RECV

Let's look at a program that has Process 0 sending 3.14 to Process 1.

Sample runs of this program are:

```
1 $ mpirun -np 2 ./sendrecv1-c.exe
2 ProcID 0 sent value 3.140000 to ProcID 1.
3 ProcID 1 received value 3.140000.
4 $ mpirun -np 3 ./sendrecv1-c.exe
5 ------------------------------------------------------------------------

6 MPI_ABORT was invoked on rank 1 in communicator MPI_COMM_WORLD
7 with errorcode 1.
8
9 [snip]
```

Example 1: MPI_SEND & MPI_RECV (con't)

The `sendrecv1.c` program is:

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char *argv[])
5  {
6    int ierr, procid, numprocs;
7
8    ierr = MPI_Init(&argc, &argv);
9    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &procid);
10   ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
11
12   if (numprocs != 2)
13   {
14     printf("ERROR: Number of processes is not 2!\n");
15     return MPI_Abort(MPI_COMM_WORLD, 1);
16   }
```

SHARCNET™

Example 1: MPI_SEND & MPI_RECV (con't)

```
17  if (procid == 0)
18  {
19    // procid 0 will send the number 3.14 to procid 1...
20    double pi = 3.14;
21    MPI_Send(&pi, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
22    printf("ProcID␣%d␣sent␣value␣%lf␣to␣ProcID␣1.\n", procid, pi)
          ;
23  }
```

SHARCNET™

Example 1: MPI_SEND & MPI_RECV (con't)

```
24   else
25   {
26     // procid 1 will wait to receive a double from procid 0...
27     double value;
28     MPI_Status status;
29     ierr = MPI_Recv(&value, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &
           status);
30     if (ierr == MPI_SUCCESS)
31       printf("ProcID␣%d␣received␣value␣%lf.\n", procid, value);
32     else
33       printf("ProcID␣%d␣did␣not␣successfully␣receive␣a␣value!\n",
           procid);
34   }
35
36   ierr = MPI_Finalize();
37 }
```

Example 2: MPI_SEND & MPI_RECV

Let's look at a program that has Process i sending the results of $3.14 + i$ to Process $i + 1$.

A sample run of this program is:

```
1 $ mpirun -np 6 ./sendrecv2-c.exe
2 ProcID 0 sent value 3.140000 to ProcID 1.
3 ProcID 1 received value 3.140000.
4 ProcID 4 sent value 7.140000 to ProcID 5.
5 ProcID 2 sent value 5.140000 to ProcID 3.
6 ProcID 3 received value 5.140000.
7 ProcID 5 received value 7.140000.
8 $
```

SHARCNET™

Example 2: MPI_SEND & MPI_RECV (con't)

The `sendrecv2.c` program is:

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char *argv[])
5  {
6    int ierr, procid, numprocs;
7
8    ierr = MPI_Init(&argc, &argv);
9    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &procid);
10   ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
11
12   if (numprocs % 2 != 0)
13   {
14     printf("ERROR: Number of processes is not even!\n");
15     return MPI_Abort(MPI_COMM_WORLD, 1);
16   }
```

SHARCNET™

Example 2: MPI_SEND & MPI_RECV (con't)

```
17  if (procid % 2 == 0)
18  {
19    // even procid i will send the number 3.14+i to procid i+1...
20    double val = 3.14+procid;
21    MPI_Send(&val, 1, MPI_DOUBLE, procid+1, 0, MPI_COMM_WORLD);
22    printf("ProcID␣%d␣sent␣value␣%lf␣to␣ProcID␣%d.\n", procid,
          val, procid+1);
23  }
```

SHARCNET™

Example 2: MPI_SEND & MPI_RECV (con't)

```
24   else
25   {
26     // odd procid i will wait to receive a value from any procid
27     double val;
28     MPI_Status status;
29     ierr = MPI_Recv(&val, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
           MPI_COMM_WORLD, &status);
30     if (ierr == MPI_SUCCESS)
31       printf("ProcID␣%d␣received␣value␣%lf.\n", procid, val);
32     else
33       printf("ProcID␣%d␣did␣not␣successfully␣receive␣a␣value!\n",
             procid);
34   }
35
36   ierr = MPI_Finalize();
37 }
```

Example 3: MPI_SEND & MPI_RECV

Let's look at a program that has Process i sending the results of $-i$ to Process 0. Process 0 will sum up all i values received.

A sample run of this program is:

```
1  $ mpirun -np 4 sendrecv3-c.exe
2  ProcID 1 sent value -1.000000 to ProcID 0.
3  ProcID 3 sent value -3.000000 to ProcID 0.
4  ProcID 2 sent value -2.000000 to ProcID 0.
5  ProcID 0 sent value -0.000000 to ProcID 0.
6  ProcID 0 received value -0.000000.
7  ProcID 0 received value -1.000000.
8  ProcID 0 received value -3.000000.
9  ProcID 0 received value -2.000000.
10 The total is -6.000000.
11 $
```

SHARCNET™

Example 3: MPI_SEND & MPI_RECV (con't)

The `sendrecv3.c` program is:

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char *argv[])
5  {
6    int ierr, procid, numprocs;
7
8    ierr = MPI_Init(&argc, &argv);
9    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &procid);
10   ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
11
12   // All procids send the value -procid to Process 0...
13   double val = -1.0 * procid;
14   MPI_Send(&val, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
15   printf("ProcID␣%d␣sent␣value␣%lf␣to␣ProcID␣0.\n", procid, val);
```

SHARCNET™

Example 3: MPI_SEND & MPI_RECV (con't)

```
16   if (procid == 0) {
17     // Process 0 must receive numprocs values...
18     int i; double val, sum = 0; MPI_Status status;
19
20     for (i = 0; i != numprocs; ++i) {
21       ierr = MPI_Recv(&val, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
              MPI_COMM_WORLD, &status);
22       if (ierr == MPI_SUCCESS) {
23         printf("ProcID␣%d␣received␣value␣%lf.\n", procid, val);
24         sum += val;
25       }
26       else MPI_Abort(MPI_COMM_WORLD, 1);
27     }
28     printf("The␣total␣is␣%lf.\n", sum);
29   }
30   ierr = MPI_Finalize();
31 }
```

Demonstrations & Questions

Demonstrations and questions.

SHARCNET™

References

[1] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 1.0. 1994-05-05. URL: http://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps (cit. on p. 4).

[2] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Version 3.0. 2012-09-21. URL: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (cit. on pp. 4, 26–29).

SHARCNET™