Debugging MPI programs

If you are having trouble getting your MPI program to work (segmentation faults, errors, deadlocks, wrong answers), you will need to debug your program. There is no denying that debugging MPI programs is generally more challenging than debugging sequential programs.

There are many different ways (including expensive tools) in order to do so. The following overview does not cover all possibilities, but will give you a small overview some ways of debugging MPI programs using common command-line tools.

1 . Double check all MPI calls

MPI operations take simple (void*) pointers as input parameters. MPI does not care about what the pointers point to and does no type checking. It is thus of utmost importance, that you, the programmer, double check all calls to MPI (both send AND receive buffers). Make sure that the pointer you pass actually points to the data.

If you are working with std::vector, you have to pass the address of the data. You can do this in the following two ways:

```
std::vector<int> vec(n);
// ... fill vector
MPI_Send(&(vec[0]),....)  // pass the address to the first element using [0]
MPI_Send(&(vec.front())    // pass the address to the first element using .front()
// WRONG:
MPI_Send(&vec,...    // This will pass the address of the vector object.
```

Also make sure that the MPI_Datatype parameters are correct. Check that if you are passing 64 bit integers, that you are using the corresponding MPI_Datatype (which is NOT MPI_INT). Also make sure that both the sending and receiving side are using the same datatype.

Furthermore, make sure that you allocate space prior to receiving. If you forget to allocate enough space, MPI will simply overwrite other data or fail with a Segmentation Fault. If MPI is overwriting data it shouldnt, then an error might occur at some later time when another data structure is trying to read the overwritten memory. As an example, if you pass &vec instead of &vec[0] as a receive buffer, than a .push_back() operation or other vector operations might fail afterwards. (Also see part **(5)** about valgrind)

2. Local debugging

If a job on Jinx fails, it is often hard to tell what was going wrong. It is thus important that before you run your MPI program on Jinx, that you run and debug your code first on a local machine. I very much recommend using a Linux machine for this, since I am unable to help you out in any way if you use any other operating system. If you don't have any Linux distribution installed, you can always install a VM application such as VirtualBox (free) and run Ubuntu insude of it. For the rest of the instructions, I'm going to assume that you are using Ubuntu or an equivalent Linux distribution.

Running your code locally inside the terminal using mpirun usually means that all your output from different processes will appear in the same output stream. The output will appear mangled up, where the output from different processors is interleaved into each other. This might make it harder to see what exactly is going on.

You can run your MPI program, with each process in a single terminal window like the following:

```
mpirun -np 4 xterm -hold -e <mpi-program>
```

Note that this will spawn 4 xterm terminals, each running one instance of the MPI program. This way you can see the output from different processes in different terminal windows.

If you are already familiar with the terminal multiplexer tmux, then the script tmpi (on github here: tmpi) can do the same inside a single big terminal window. You can use it to spawn 4 MPI processes, each inside a pane of the terminal by running this inside a tmux session:

```
tmpi 4 <mpi-program>
```

Note that the tmpi solution also allows you to synchronize keyboard input into all windows. This is especially helpful for debugging MPI applications, since all typed commands will be executed by in all windows/instances at the same time.

3. Print Statements

One way of trying to figure out what is going on is to add print outs to your code and observe what is happening. In order to figure out what is happening on each processor, you can use the technique described in **(2)**.

4. GDB

GDB is a very powerful debuger, which is used (among others) to debug C/C++ programs in Linux. You can run your MPI programs inside GDB as well. To do so, use the technique described in **(2)**. You can spawn each instance of your program inside gdb using the following:

```
mpirun -np 4 xterm -hold -e gdb --args <mpi-program> <arguments-to-mpi-program>
or
tmpi 4 gdb --args <mpi-program> <arguments-to-mpi-program>
```

you can then set break points and run your code with the run command. If your code is failing with segementation faults or other errrors, you can also immediately start running the code in the debugger. The debugger will then stop exactly where the error occured, enabling you to figure out what was going on. To immediately start running the program do:

```
mpirun -np 4 xterm -hold -e gdb -ex run --args <mpi-program> <arguments-to-mpi-program>
or
tmpi 4 gdb -ex run --args <mpi-program> <arguments-to-mpi-program>
```

5. Valgrind

Valgrind is a very powerful toolset. Among other functionality, it has a memory checker tool. This will keep track of all your allocations, and show errors when your program illegally accesses memory (like if your MPI receives into non allocated or wrong memory). This is very helpful to pinpoint where errors originate. Additonally to showing you where the accesses happen, valgrind can also start a gdb instance at the exact point that an error occurs.

(i) Staring without gdb support:

```
mpirun -np 4 xterm -hold -e valgrind <mpi-program> <arguments-to-mpi-program>
or
tmpi 4 valgrind <mpi-program> <arguments-to-mpi-program>
```

(ii) Starting with gdb support

```
mpirun -np 4 xterm -hold -e valgrind --db-attach=yes <mpi-program> <arguments-to-
mpi-program>
or
tmpi 4 valgrind --db-attach=yes <mpi-program> <arguments-to-mpi-program>
```