Preventing Bugs and Finding Bugs in MPI Programs

Charlie Peck
LSU/PSC Parallel and Cluster Computing
SC Education Program Workshop
July, 2009

```
#include "fancy logos and distracting graphics"
```

How Did We Get Here?

Debugging serial programs can be hard; debugging parallel programs is usually about `-np X` times harder.

Strategies for Preventing Bugs in MPI Programs

- Deadlock
  - Problem - Only a single process calls a collective communication function, *e.g.* MPI_Reduce or MPI_Bcast
  - Solution - Do not put collective calls inside conditionally executed code.
  - Problem - Two or more processes are trying to exchange data but all call a blocking receive function before any calls a send function.
  - Solution - Always call send before you call receive; use MPI_Sendrecv; use non-blocking send and receive calls.

- Problem - A process tries to receive data from a process that will never send it, or send it to a process that will never receive it.
- Solution - Use collective communications functions whenever possible; if you need point-to-point communications keep the communication pattern as simple as possible.

- Problem - A process tries to receive data from itself.
- Solution - Carefully examine your source code.

- Incorrect Results

  - Problem - Type mismatch between send and receive, e.g. MPI_INT on the send and MPI_CHAR on the receive.
  - Solution - Make it easy to match-up your sends and receives, check the message length and type.

  - Problem - Mis-ordered parameters to MPI function calls.
  - Solution - Check them closely and use a `man` page or another MPI reference when coding.

- In general there are more opportunities for bugs with point-to-point communications than collective communications.
- Practice defensive programming: check all return codes, check MPI function arguments, code deleted is code debugged.
- Make the program correct, then make the program fast.

Strategies for Finding Bugs in MPI Programs

- If the program will run as a single process use this form to debug it as much as possible.
- If the program will run with 2 processes on a single node/core and exercise all of the functionality use this form to debug it next. In general the fewer processes the easier it is to debug and on one node/core many race conditions are prevented.
- If the program will run with 2 processes on 2 nodes/cores and exercise all of the functionality use this form to debug it next. This configuration begins to expose potential race conditions and allows you to verify synchronization and timing in the simple case.

- Work with the smallest problem size possible which exercises all of the functionality. This allows you to examine entire data structures, all loop iterations, etc.
- Use `fflush(stderr);` after each `fprintf(stderr, "rank=%d, ...", my_rank, ...);` call, this prevents messages from being lost in a buffer when the program crashes or deadlocks.
- For point-to-point messages print the data elements before the send and after the receive, make sure you are sending and receiving what you think you are.
- Don't assume the order of received messages from more than one process.
- Guard your debugging statements with something like `#ifdef DEBUG ... #endif` so that you can easily enable and disable them as need be.
- Build, run, and test your program incrementally as you go, this usually reduces the amount of code you have to examine when something does go belly-up.

Tools, Techniques and Resources

- `gdb`, `ddd`, `XMPI` (LAM MPI only), other MPI binding specific debuggers.
  - `while (1) do {}` just after main and then attach with `gdb` or
  - A shell script with `export DISPLAY=...; mpirun...`
- `__FILE__`, `__LINE__`, `__FUNCTION__` with a C preprocessor macro.
- `fprintf(stderr, "rank=%d, ...", my_rank, ...)` or `cout` statements, bracketed with conditionals so they can be easily enabled and disabled.
- Appendix C of Quinn's *Parallel Programming in C with MPI and OpenMP*
- Chapter 5 of Kernighan and Pike's *The Practice of Programming*
- `http://www.open-mpi.org/faq/?category=debugging`