Preventing and Finding Bugs in Parallel Programs

Charlie Peck, Samuel Leeman-Munk

Intermediate Parallel Programming and Cluster Computing

@ University of Oklahoma OSCER

August, 2010

How Did We Get Here?

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

This material is as much about software engineering and debugging generally as it is about techniques unique to debugging parallel programs. This reflects the nature of the work at–hand.

Strategies for Preventing Bugs in Parallel Programs

- Enbug rather than debug.
- Practice defensive programming:

  1. Check all return codes
  2. Check all function arguments (`--pedantic`)
  3. Use layout to infer structure
  4. Choose meaningful variable names

- Think carefully about how shared data elements are read and written. All parallel programs are strong candidates to exhibit race conditions.

- 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 (and it most likely will).
- Code deleted is code debugged, or put another way less is often more in software engineering.
- Make the program correct, then make the program fast.

Strategies for Finding Bugs in Parallel Programs

- The basic process:

  1. Characterize the bug

     (a) If possible run the program serially, make sure it works correctly in that mode.
     (b) Run the program with 2–4 processes on a *single* core, make sure it works correctly in that mode. In general the fewer processes the easier it is to debug and on one core many race conditions are prevented.
     (c) Run the program with 2–4 processes on 2–4 cores. This configuration begins to expose potential race conditions and allows you to verify synchronization and timing in a simple case.

  2. Fix the bug
  3. Test the fix

- Change the input and then study the output to characterize the bug. *Do not* keep the input constant, change the code and then study the output to characterize the bug. An exception to this is the addition of statements to examine variables at runtime (see below).
- 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.
- Suspect that it's a race condition, setup test cases to prove that the software doesn't have any.
- The non–deterministic nature of parallel programs can lead to a false impression of what's actually going on.

- Use guarded print statements (*e.g.* `#ifdef DEBUG printf("var = ...)` and leave them in the program when you are done, you'll probably need them again. Most bugs can found using this approach.
- Make `DEBUG` a symbol that can easily be set from the command line of your program at runtime.
- Beware of lost output when a program terminates abnormally, this leads to false impressions about what is actually going on. Use `fflush(stdout)`, or write to `stderr` (which isn't buffered) or use `setbuf(STREAM, 0)` to prevent messages from being lost in a buffer when the program crashes or deadlocks.
- Learn about the C constructs `__FILE__`, `__LINE__`, `__FUNCTION__` and use them with an error handling routing to improve the quality of your debugging output.

- Learn how to use `gdb`, it's a powerful tool that can help find some types of bugs.
- When you do start changing the code make one logical set of changes at a time and then re–test. Keep your focus, don't wander off and start futzing with unrelated code while working on a particular bug.
- Each time you go on a debugging tour document and preserve the test script(s) that you develop. Add these to the regression testing suite for that program.

Strategies for Preventing Bugs in MPI Programs

- Synchronization
  - 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.
- NOTE - This issue will cause a hangup only in some bindings of MPI, such as MPICH. OpenMPI will not fail.

- Incorrect Results
  - Problem - Data 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 with collective communications.

Strategies for Finding Bugs in MPI Programs

- 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 when they come from more than one process.
- Use the MPI functions `MPI_<type>_set_name` and `MPI_<type>_get_name`, where `<type>` can be: `Comm`, `Win`, or `Type`. These give human readable names to MPI's structures which makes debugging much easier.
- Always use `fprintf(stderr, "rank=%d, ...", my_rank, ...)` or `cout` statements, guarded with conditionals (see *Debugging Parallel Programs*), so that it's easy to identify where particular output is coming from.

- Test your code with every MPI binding you can! You want your code to work in as many different environments as possible!

Resources

1. Appendix C of Quinn's *Parallel Programming in C with MPI and OpenMP*
2. Chapter 5 of Kernighan and Pike's *The Practice of Programming*
3. `http://www.open-mpi.org/faq/?category=debugging`
4. `http://www.hlrs.de/organization/av/amt/research/marmot` - Marmot, an MPI call syntax and behavior analyzer
5. Geist et al., *Debugging Parallel Programs*, 1994
6. Electric Fence
7. Valgrind