Preventing and Finding Bugs in Parallel Programs

Charlie Peck
Earlham College

Introduction to Parallel Programming and Distributed Computing
UW and ISU
NCSI/SC Education Program
June, 2011

How Did We Get Here?

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

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

- Practice defensive programming:

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

- 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.
- Learn and use the features of the language that help prevent bugs, *e.g.* in C use `const` variable declarations rather than the pre-processor's `#define` mechanism.

- Think carefully about how shared data elements are read and written. All parallel programs are strong candidates to exhibit race conditions.
- Make the program clear and correct, then make the program fast. This does not apply to the design of the algorithm but rather to the details of the implementation.
- Days of debugging can save you hours of design and planning.

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. Develop a script that automates the process of "activating" the bug. Automation is key to making the debugging process as effective and inoffensive as possible.

3. Fix the bug.
4. Test the fix using the script developed earlier.

- To characterize the bug change the input and then study the output. *Do not* keep the input constant, change the code, and then study the output. Changing the input exercises the whole path, whereas changing the code only effects a limited portion, and may introduce new bugs. An exception to this is the addition of print 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 scripts to prove that the program doesn't have any.
- The non-deterministic nature of parallel programs can lead to a false impression of what's actually going on. Remove as much of the non-determinism as you can.

- If you are developing code on a 64 bit platform build and run it on a 32 bit platform, or the obverse. This illuminates bugs related to word size.
- Use guarded print statements (*e.g.* `#ifdef DEBUG fprintf(stderr, "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 a custom error handling routine (see below) to improve the quality of your debugging output.

- Learn how to use `gdb`, it's a powerful tool that can help find many 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.
- Don't under-estimate the value of a second set of eyeballs.

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 - Only some MPI bindings will hang in this case, as of the most recent version of this presentation MPICH will hang but OpenMPI will not.

- Problem - Deadlock.
- Solution - Perform operations in the same order in each place they are done, *e.g.* send/receive pairs, collective calls, and locking.

- 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 or incorrect 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 to 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.
- Explore MPI's support for custom error handlers. A ready-to-use example of one can be found in `mpi-error-handler-example.c`.
- 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 can make debugging much easier.
- Always use `fprintf(stderr, "rank=%d, ...", my_rank, ...)` or `cout` statements, guarded with conditionals (see above), so that it's easy to identify where particular output is coming from.

- Build and run your code with a different MPI binding. This often illuminates bugs, and you want your code to work in as many different environments as possible.
- Explore the debugging options supported by the MPI binding you are using. Many bindings have linkages between `mpirun` and debuggers that automagically invoke the debugger when an error is encountered.

Strategies for Preventing Bugs in OpenMP Programs

- Avoid loop carried dependencies and other constructs which lead to race conditions. Shared memory parallel programs like those built with OpenMP are particularly vulnerable to race conditions. Missing `private` and `critical` clauses can also lead to race conditions.

- Use `critical`, `master`, `single` sections as appropriate to ensure deterministic behavior when necessary.

- Variables not explicitly made `private` are shared among all threads, check to make sure this is reasonable.

- Make sure that any `barrier` calls are executed by all threads.

- Make sure that any library and function calls made inside parallel regions are thread safe.

- Beware global variables, they are not thread safe.

- Carefully examine all locking constructs to prevent deadlock. Always lock and release objects in the same order.

Strategies for Finding Bugs in OpenMP Programs

- Vary the number of threads (using `OMP_NUM_THREADS` so you don't have to recompile) and observe the results for changes in behavior. Start with `OMP_NUM_THREADS == 1` to ensure that the serial version works correctly.

- Learn and use GDB's thread capabilities:

  1. Automatic notification when threads are created and destroyed.
  2. `info threads`
  3. Thread specific breakpoints, `break linespec thread threadnum`
  4. Conditional breakpoints, `break buffer.c:15 thread 3 if bob == green`
  5. Switch between threads, `thread threadnum`

- Remove any compiler optimizations, with gcc use `-O0`.

- Selectively enable and disable parallel directives on specific sections of code to isolate where the error is happening, think binary searching.

- If you suspect a race condition increase the number of threads greatly. More threads increases the likelyhood of illuminating race conditions.

Strategies for Preventing Bugs in CUDA Programs

- TBD

Strategies for Finding Bugs in CUDA Programs

- Learn and use CUDA-gdb.

Strategies for Preventing Bugs in Hybrid Programs

- Architect the code so that you can disable each of the different parallel paradigms, this will make it easier to isolate where the bug is.

Strategies for Finding Bugs in Hybrid Programs

- Disable each of the parallel paradigms one at a time to isolate where the bug is.
- TBD

Resources

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