Final Report

Victoria Som de Cerff
vs2606@columbia.edu

May 5, 2019

1 Proposal

Develop a testing framework that would deliberately manipulate process behavior with minimal modifications to the actually functionality of the code to uncover false negative tests. Tests that fail for whatever reason should fail smoothly without hanging and preventing further tests from executing. The framework will collect useful information for debugging test execution afterwards.

2 Novel

This is novel in that it is a unit test framework that purposely targets MPI programs. I am leveraging Catch which is a unit test framework that is synonymous with many other available unit test frameworks. And what these unit test frameworks have in common is that they assume sequential single process code.

As you will see in my implementation, additional structure can be added to provide functionality suitable for multiprocess codes but it is not out of the box.

2.1 User Community and Value

Hopefully this framework can be a use add on to the Catch unit test framework such that other developer who work on high performance computing and large scale parallel codes can have a tool that enables them to write sound code without additional burden of having to alter their original code or have to develop highly specialized test code.

Debugging is a natural process in software development; and testing is key software engineering practice. This framework drives to make debugging of parallel codes easier; and promote thorough testing of parallel strategies.

3 Research Questions

3.1 Paper/Proposal Questions

Here are the answers I uncovered from the questions posed during my original proposal:

1. What testing frameworks are available for multiprocessing?

   None. I could not find any out of the box test frameworks for MPI programs. There were test frameworks for the MPI library itself. But not codes that actually use this library. All other examples I saw were bootstrapping off of regular test frameworks, similar to what I have done for this project.

2. What level of control can be enforced when testing and manipulating specific processes, and how much effort level of control requires?

   It is possible to right a wrapper around MPI calls and insert sleep calls in hopes of manipulating process execution behavior. This is done rather simply. However, I have found that the MPI library performs many optimizations under the hood so even putting in these sleep calls is not necessarily enough to change the outcome of certain MPI programs. To gain explicit control of the processes you would have to make changes within the MPI library which may have catastrophic effects to the API without a deeper knowledge of the protocol.

3. Where multiprocessing test suite can add the most value to developers and end users?

   For me personally and what I feel has been echoed by other developers during my research, is having a more effective debugging method would be most helpful in testing MPI programs. Which is why my focus shifted to creating a logger of all MPI calls and processing them to highlight potential causes of failure. Along with having a test strategy that gracefully recovers from failure so that information can be collected.

3.2 Updated Questions

At the progress update milestone, new questions were posed.

1. Can recovery from deadlock be done with minimal impact to the original source code and no impact to the original source code's true functionality?

   Yes! I had issues with conflicting namespaces. Thus my wrapper calls did have to have different names than the function names in the MPI library. Therefore in my programs I had to rename all of the MPI calls to my wrapper calls. However, that was the only change necessary. The function arguments and return values remained the same.

   Additionally none of the programs for the test could call MPI_Init or MPI_Finalize. These calls initialize and destroy the MPI environment and should only be called once throughout the lifetime of a program. These calls were handled by the Catch test framework instead since that is where the overall program started and ended. If MPI_Finalize were to be called in any of the individual tests then none of the remaining tests would be able to operate in the MPI environment.

2. Can stalling processes really increase failure rate. This has been a presumed hypothesis from real life experience and research from my paper but it might be hard to recreate with such trivial examples?

   Maybe? Early on it became clear that this could have an impact but not a strong reproducible one. As mentioned earlier more in depth manipulation would need to occur within the MPI library to have a true consistent effect on process execution.

   I did see failure rates increase but not at any higher rate than just running the program without my manipulations.

3.3 Further Questions

Here are some questions still left open that have come up as I have continued to work on this project.

1. MPI has a really simple call to signal something has gone wrong. MPI_Abort. Once any process issues this, all other processes will get interrupted. This would have be clean way for one process to signal to the other processes that they should all give up and move onto the next test. However, the abort also destroys the MPI environment, meaning these processes can never communicate again afterwards so no other tests would run effectively after that call.

   If I were going to attempt to manipulate the underlying MPI library, I would want to see if there would be a way to send a signal, not so severe as the abort, but something that would interrupt the processes and let them jump to some known safe checkpoint in code.

4 Starting Point

I started with Catch as a unit test framework so that I wouldn't have to create that myself. From there I wrote a series of small parallel programs that should exhibit race condition and deadlocks. I used Catch to write a test for each of the parallel programs. I'll go more in depth about the test_setup and test_closeout methods you see below further into the report. See src/tests/mpi_test.cpp

```
TEST_CASE( "DL: Send Recv, single value, 2 procs", "[np2][mpi][dead][dead2]" ) {
   int rank, mpi_size;
    test_setup(&rank, &mpi_size, "DL2");
    REQUIRE(mpi_size == 2);

    unsigned int data = 5;
    int desired_value = data;
    int ret_value = deadlock_2(data);

    check_test_results(rank, mpi_size, desired_value, ret_value);

    test_closeout();
}
```

Getting these tests to actually run was the next challenge. First I had to centralize where the MPI environment was created before the test ran and make sure that it was destroyed afterwards. See src/tests/catch_mpi_main.hpp

```
#define CATCH_CONFIG_RUNNER
#include <catch2/catch.hpp>
#include <mpi.h>

int main(int argc, char * argv[]) {
    MPI_Init(&argc, &argv);
    int result = Catch::Session().run(argc, argv);
    MPI_Finalize();
    return result;
}
```

After that it was working out my own bugs of getting the processes to complete these tests together in order.

5 Project Details

There were four main goals for this framework and I will share details about how I achieved them in the following sections.

5.1 Effective MPI unit test environment

After making the changes to catch_mpi_main.hpp the MPI environment existed for the test cases. I included a test_setup that I called at the beginning of each test. This was key for the other Logger which is talked about further in the report. At the end of each test I called test_closeout. Initially it simply called MPI_Barrier, which meant every process must get to this point before anyone can continue. This is important so that every process agrees that they've finished the test before moving onto the next test case. But what if one process was deadlocked somewhere else? Then all of the remaining processes would hang here waiting for someone who will never come. That is where the all_here method was critical change.

To achieve this I issue and Ibarrier, which is a nonblocking form of Barrie. So all processes still need to reach this point however, it does not force the processes to wait until everyone gets there. This frees up the processes to do other things, such as MPI_Test if everyone has reached that call. Or see if a deadlock message has been emitted.

```
void all_here() {
    MPI_Request req;
    MPI_Status stat;
    MPI_Ibarrier(MPI_COMM_WORLD, &req);

    int all_here = false;
    while (!all_here) {
        MPI_Test(&req, &all_here, &stat); // Test if everyone reached this
            Ibarrier request
        MPI_Wrap::check_deadlock_message();
    }
}
```

The check_test_results centralizes all of the results for each process to rank 0, allowing rank 0 to call the Catch method CHECK to verify if a test assertion was true. This was important so that the output of Catch would show much more simply the results of that test case from one process, rather than scrambling through the output of all the processes.

```
void check_test_results(int rank, int mpi_size, int desired_value, int
    ret_value) {
```

```
  all_here(); // Check for deadlock first otherwise proceed

  int all_desired[mpi_size];
  int all_returned[mpi_size];
  //Send desired_value back into array all_desired array owned by process 0
  MPI_Gather(&desired_value, 1, MPI_INT, &all_desired, 1, MPI_INT, 0,
      MPI_COMM_WORLD );
  //Send returned_value back into array all_returned array owned by process 0
  MPI_Gather(&ret_value, 1, MPI_INT, &all_returned, 1, MPI_INT, 0,
      MPI_COMM_WORLD );
  if (rank == 0) {
       for ( int p = 0; p < mpi_size; p++) {
        if (all_returned[p] != all_desired[p] ) {
           printf("Undesireable value from process %d", p);
        }
           CHECK( all_returned[p] == all_desired[p] );
       }
   }
}
```

5.2 Provide useful information for debugging

This was achieved through a Logger. Each process had its own logger to track all of the MPI calls it issued. Every time a call was made through my MPI Wrapper library a log entry was stored in a vector containing.

- message id: index of message in vector
- call: enumeration indicating which MPI call was issued. This did not cover all possible MPI messages only: MPI_SEND, MPI_ISEND, MPI_RECV, MPI_IRECV, MPI_BCAST, MPI_IBCAST
- source: source of the message
- destination: receiver of the message
- tag: tag of the message (integer, unique way of grouping messages, user specified)
- wait time: how long the process was purposely stalled before actually issuing the real MPI call

At the beginning of each test, this list was cleared. At the end a test or at the time of deadlock it was written to disk by each individual process.

There is a python post processing that would read in all of the files for a given test and match every Send to a Recv and check if every Bcast was called by all processes. Any unmatched calls would be displayed to users as a likely cause of failure for the test.

5.3 MPI Wrapper

This namespace declared the static Logger that every process had on its own. It also defined the wrapper functions to MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv, MPI_Bcast, MPI_Ibcast. See src/util/mpitest.cpp

5.3.1 Reduce false negatives

Each of these wrappers would sleep the process for a random interval. Make a log entry then make a real MPI call. The sleeping was in hopes to induces false negatives by simulating different execution variants that might not be achieved in small scales. As mentioned before this didn't seem very fruitful so not much more interest was put into it.

I had enough failing test cases that didn't need direct manipulation so I focused on improving the test experience for those cases.

5.3.2 Recovery from Deadlock

This was the most complex part of this framework. Firstly, how to detect if deadlock has occurred. Secondly, how to get out of without having to hit Ctrl-C and just kill the whole thing.

The most non-specific way to detect whether or not deadlock had occurred was to set a time out. This obviously forces the developer to know upfront some expected duration for each communication throughout the program. But its better than nothing if the program can honor it and recover from it. And the developer can know to adjust the time if they low ball it and see too many unintended deadlock cases.

Now to make this work all blocking calls had to be converted to a nonblocking call. Otherwise, well deadlock. But I didn't want my wrapper to change how the original function intended to run. So even if I make the call nonblocking, it needs to appear blocking to the end user of the wrapper. Thus after the nonblocking call, I continuously loop checking to see if the nonblocking call actually completed. If it did, great, return as normal to the user. If not, we keep looping. During this loop we check if any other process has indicated deadlock. If not, we keep looping. Eventually if the processes passes the timeout period, the process

will indicate deadlock to everyone else by sending an MPI message with our deadlock tag identifier.

After deadlock occurs we want to the test to end cause clearly something has gone wrong. As mentioned before, we can't use MPI_Abort cause that kill this test and all future tests. Here we rely on Catch. Catch has two assertion types. CHECK which expects a true assertion but will let the test continue. And REQUIRE, which if the assertion resolves to false then the test ends there. Which is exactly what we need. It's important that no process quits the test before receiving the deadlock message or else the process that sent it will deadlock too. And that puts us back in the situation we are trying to avoid.

How do we receive a message that may or may not be there? MPI_Iprobe. It is a non blocking probe, which checks to see if there is any message designated for that process that matches a certain criteria. So the process can check if a message has been sent for it with our deadlock tag identifier.

```
bool Logger::is_deadlock_detected()
{
   int detected = false;
   MPI_Iprobe(MPI_ANY_SOURCE, DEADLOCK_TAG, MPI_COMM_WORLD, &detected,
       MPI_STATUS_IGNORE);
   if(detected) {
      int noop = 0;
      MPI_Recv(&noop, 1, MPI_INT, MPI_ANY_SOURCE, DEADLOCK_TAG, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
      REQUIRE(false);
   }
   save(test_name + "_" + std::to_string(rank) + ".csv");
   return detected;
}
```

```
void Logger::emit_deadlock_detected(std::string method_name)
{
   printf("DEADLOCK DETECTED by %d in %s \n", rank, method_name.c_str());
   int noop = 0;
   for( int p = 0; p < size; p++ ) {
      if (p != rank) {
         MPI_Send(&noop, 1, MPI_INT, p, DEADLOCK_TAG, MPI_COMM_WORLD);
      }
   }
   save(test_name + "_" + std::to_string(rank) + ".csv");
```

```
  REQUIRE(false);
}
```

Getting to this strategy is where I definitely wish I had an MPI testing framework cause I hit a lot of deadlocks getting this to function properly.

6 How to access

https://github.com/vsomdecerff/coms6156/

See README.md for usage.

7 Challenges

Getting the recovery from deadlock strategy is where I definitely wish I had an MPI testing framework cause I hit a lot of deadlocks getting this to function properly.

I would have liked to integrate this into the Catch code itself but everything I tried to put into that code base broke it. So with more experience I would like to include this as a true feature instead of an add-on.

Not having a deep understanding in how MPI communicates over the different network fabrics also posed challenges early on. I got a better idea as I continued my project but I never dug deep into how it is performing the optimizations when copying the message buffers back and forth. I found that the recv calls are the only things you can count on to really block since it is waiting for data. But blocking send calls might be able to move forward if MPI can store the message away in a temporary buffer somewhere in memory for the receiving process to access when it gets around to it. I probably could do an entire research paper in understanding how MPI is implemented. Note I am using OpenMPI but there are other implementations. Such as IMPI which is Intel's implementation that has its own optimizations for the MPI standard.

8 Future Work

This only works in MPI_COMM_WORLD This is default communicator group where all processes exist together but has not been implemented for subgroups.

Also I would want to resolve the namespace issue so that absolutely zero change would have to be made to the original developer's source code to use the wrapper.

Then the next step would be to extend the wrapper to a broader set of the MPI library communication calls.