Project Update

Victoria Som de Cerff
vs2606@columbia.edu

April 18, 2019

1 Problem Statement

Generally, parallel codes are modeled and tested on small scales initially before transitioning to full scale production runs. This is because it is easier during development to conceptualize a simulation on small number of processes. And it is typically easier to acquire a smaller pool of resources when working in a test environment or shared computing system.

With MPI programs, its entirely possible that even though standard tests have passed that a bug still lies in wait because the program execution has not been thoroughly stressed. These false positives are often not uncovered until the software is in production and then difficult to track down without testing the program at production scale.

Further once a problem has been realized, reproducing and tracking down the issue brings about another level of complexity for the developer.

2 Proposal

Develop a testing framework that would deliberately manipulate process behavior with minimal modifications to the actually functionality of the code to uncover false positives tests. As well as producing useful information debugging information

3 Failures of Interest

The errors that have become the focus of the test suite are race conditions and deadlock. Pinpointing race conditions and deadlocks is of particular interest because as they become more visible at large scales, their root cause can still be difficult to assess as they are usually indicated only by the program hanging.

3.1 Race Conditions

A race condition occurs when a program attempts to perform two or more operations at the same time, but the operations must be done in a specific order to be done correctly.

```
int a = 13;

// Start immediate send
MPI_Isend (&a, 1, MPI_INT, dest, tag, comm, &req);
a = 0;

MPI_Wait (&req, &stat);

// Has the dest process received 13 or 0?
```

3.2 Deadlock

Deadlock is a situation where two or more processes are all waiting for each other to take action before continuing onto the next state. Since none of the processes will budge, the program comes to a halt.

```
int a = calculate_something (my_rank)
int b;

MPI_Send (&a, 1, MPI_INT, left, tag, comm);

// MPI does not guarantee that the execution won't block inside MPI_Send
MPI_Recv (&b, 1, MPI_INT, right, tag, comm, &stat);
```

4 Implementation

There are three main components to be implemented for this framework:

- Logging Communications. This is needed for the post-error debugging phase.
    1. Logging communications. Registering all of the MPI messages sent and received.
    2. Unifying communications. Each process is going to log its communications. How are all of these logs compiled to one useful report.

3. Saving report. In the case of failure or deadlock, how will the logs be saved and recovered.

- Inducing Failure. Stress non determinism and provoke failures.

  1. Stalling processes. Randomly stall process communication in hopes of discovering valid but unsuspecting runtime behavior.

- Recover from Failures.

  1. Recognizing deadlock. By definition the program comes to a halt, framework needs to be able realize that this has happened, record it, and return gracefully.

- Unified Error Reporting. Adapt Unit Test framework to collate success/failure across all processes.

  – Relies on recovering from failures.

5 Work done thus far

5.1 Test cases

A set of test cases for race condition and deadlock have been written. All have the ability to fail the unit test. At this moment some already do, some don't. Ideally I would like to see all of them fail as expected.

5.1.1 Concerns

After reading more about MPI, it might not be possible to induce failure in such simple programs due to the optimizations within the library. Rather than spend too much time writing increasingly complex test cases that won't play into the optimizations, the goal will be to attempt to increase the baseline rate of failure. As opposed to reaching 100% failures amongst all tests.

I will look at using slightly more complex but still common MPI algorithms in the tests too. I have found a basic MPI example for calculating the value of Pi.

5.2 Avoiding impact of deadlock on framework

Threading has implications for two parts of the implementation. One the ability log activity in the background and to recognize deadlock. MPI has four types of threading it supports:

- MPI THREAD SINGLE: Only one thread will execute.
- MPI THREAD FUNNELED: The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funneled to the main thread).
- MPI THREAD SERIALIZED: The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are serialized).
- MPI THREAD MULTIPLE: Multiple threads may call MPI, with no restrictions.

Before diving deep into those options, what I have learned from trying out threads with MPI is that all threads must rejoin with the main thread of execution in a process before it calls MPI_Finalize otherwise the code will crash catastrophically.

What I have spent my time planning is whether the MPI wrapper library that I will implement will use threads to detect deadlock and intervene to ignite the recovery process. Or an equally complicated task would be to convert all blocking calls to nonblocking calls and continuously check if nonblocking call completed before returning. In this scenario, if the call never completes then that would indicated deadlock.

With either option, deciding whether deadlock occurred is impossible to detect at runtime. At runtime there is no way to tell what the other processes are currently doing or expected to do in the future. This unfortunately will be an open plot hole in program. I will have to rudimentally require that the user input some sort of cutoff limit for the test. If the test or certain operations within the test do not complete in a predetermined set of time then I could consider the program deadlocked.

Back to threading. Threading *could* be used for unifying log messages in real time. Meaning there would be the foreground thread in each process communicating for the real purpose of the program, and background thread logging and sending log information back to a single master to merge and sort the information then ultimately save it out. This would utilize the MPI THREAD MULTIPLE feature of MPI. The wrapper around the MPI library would then signal from the foreground thread to the background thread when Finalize was called so that the background thread could rejoin into a single thread of execution and complete the program gracefully.

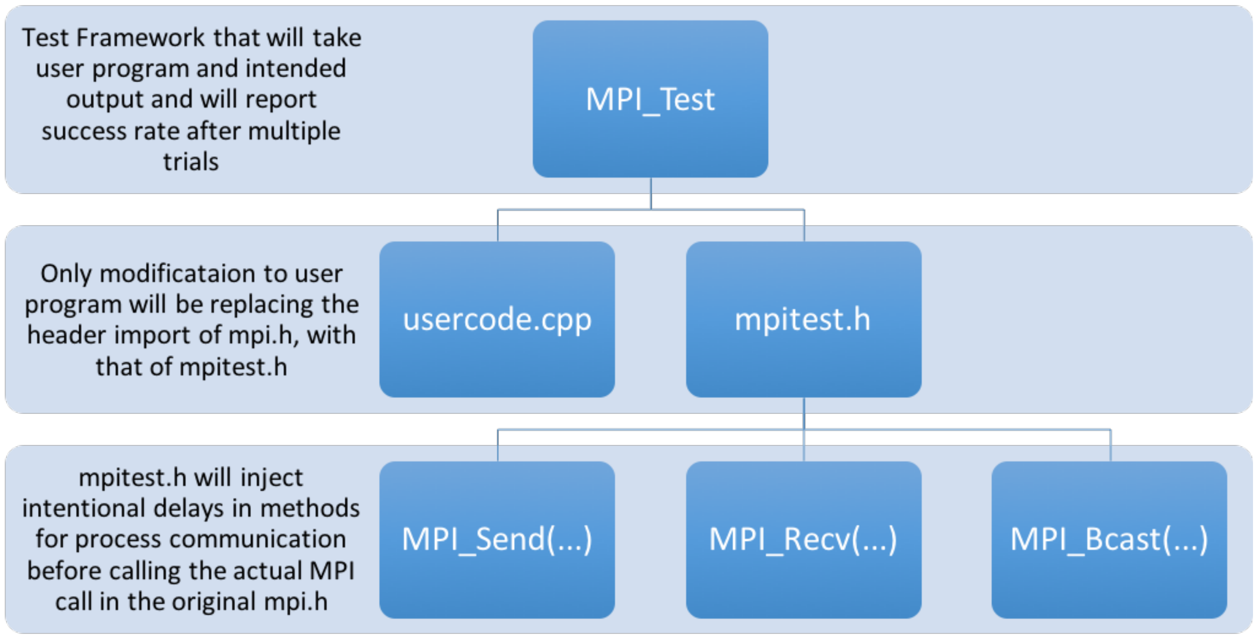
I am not sure yet if this level of communication level of complexity buys any value rather than having each process log their information in isolation and writing it out individually. Then having a separate process read the logs after the fact to unify them.

For this reason I am moving forward without threading and instead converting all blocking calls into non blocking. Now to keep this architecture change from affecting the original intention of the code. I will ensure that the wrapper simulates the block and does not return until the intended blocking call completes.

This is achieved in the following code.

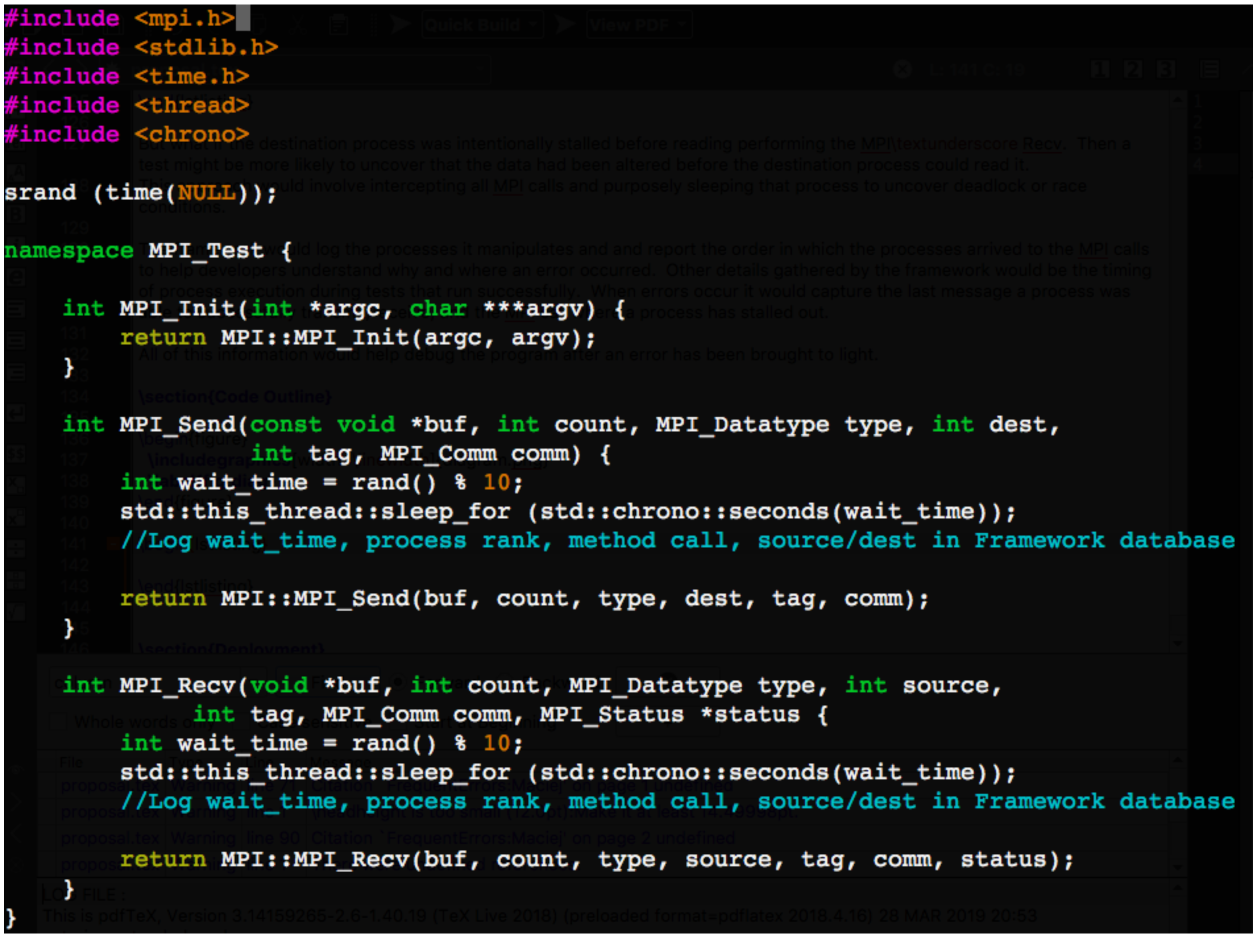
```
int MPI_Send(const void *buf, int count, MPI_Datatype type, int dest,
               int tag, MPI_Comm comm) {

       // Purposefully stall code
       int wait_time = rand() % 10;
       std::this_thread::sleep_for (std::chrono::seconds(wait_time));
       //Log wait_time, process rank, method call, source/dest in Framework
           database

       int ret_value;

       MPI_Request req;
       MPI_Status stat;

       ret_value = MPI::MPI_Isend (&buf, count, type, dest, 0, comm, &req);

      if (ret_value != MPI_SUCESS) { return ret_value; }

       bool flag = false
     while(flag) {
            ret_value = MPI_Test(request, flag, status)

            if (ret_value != MPI_SUCESS) { return ret_value; }

            // If testing loop exceeds timing threshold then deadlock
       }

      return ret_value;
   }
```

MPI_Test will check if a call has been completed. We will continue to check if the non blocking send has completed before actually returning. Thus to the user, it is still a non blocking call. But to the framework it won't actually get caught in deadlock preventing it from moving forward.


Test Framework that will take user program and intended output and will report success rate after multiple trials
MPI_Test
Only modificataion to user program will be replacing the header import of mpi.h, with that of mpitest.h
usercode.cpp
mpitest.h
mpitest.h will inject intentional delays in methods for process communication before calling the actual MPI call in the original mpi.h
MPI_Send(...)
MPI_Recv(...)
MPI_Bcast(...)

6 Code Outline

A diagram is provided to show a high level of how the framework operates. Additionally, an example of what the wrapper library that will intercept MPI calls and hold the process execution is included.



```
#include <mpi.h>
#include <stdlib.h>
#include <time.h>
#include <thread>
#include <chrono>

srand (time(NULL));

namespace MPI_Test {

    int MPI_Init(int *argc, char ***argv) {
        return MPI::MPI_Init(argc, argv);
    }

    int MPI_Send(const void *buf, int count, MPI_Datatype type, int dest,
                 int tag, MPI_Comm comm) {
        int wait_time = rand() % 10;
        std::this_thread::sleep_for (std::chrono::seconds(wait_time));
        //Log wait_time, process rank, method call, source/dest in Framework database

        return MPI::MPI_Send(buf, count, type, dest, tag, comm);
    }

    int MPI_Recv(void *buf, int count, MPI_Datatype type, int source,
                int tag, MPI_Comm comm, MPI_Status *status {
        int wait_time = rand() % 10;
        std::this_thread::sleep_for (std::chrono::seconds(wait_time));
        //Log wait_time, process rank, method call, source/dest in Framework database

        return MPI::MPI_Recv(buf, count, type, source, tag, comm, status);
    }
}
```

Figure 1: Example of code that would augment process behavior before making the real MPI call.

7 Deployment

The code will be developed on Github in C++. A Makefile will be provided to compile and launch examples of the framework in action.

https://github.com/vsomdecerff/coms6156

8 Demo

My demo will involve executing the ”vanilla” unit test cases to prove how they are deficient. Then execute my unit test framework to hopefully uncover increased failures. Then review the reported log information to show how that helps user debug.

9 Questions

So far the most interesting facet of the project, as you might be able to tell, is how to recover from deadlock in the unit test framework. The big question is can this be done with minimal impact to the original source code and no impact to the original source code's true functionality.

The other question is 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.

10 Systems

I am running this on my laptop. I might scale up on my work's super computer but preliminary tests have already shown that failures can be induced without needing such large scales.

11 Novel

I think this has been implied before but this is novel in that it is a unit test framework that purposely targets MPI behavior.