# Project Proposal

Victoria Som de Cerff
vs2606@columbia.edu

April 17, 2019

## 1 Revised Project Proposal

The overall goal and problem statement are the same but based off of the work done to define examples of the bugs I want to solve, it is likely that I will drill down into resolving issues for either race conditions or dead lock but not both. They both present unique challenges that will require different tactics to resolve. More details will be presented in the revision of the Research Paper.

Sections 1-4 is a recycle of midterm paper content, sections 5 onward is new project information (no changes since initial submission).

## 2 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.

## 3 Proposal

Develop a testing framework that would deliberately manipulate process behavior without modifying the actually functionality of the code to uncover false positives tests.

# 4 Failures of Interest

The framework is not intended to test "sequential" portions of the code, as those can be tested through typical unit test frameworks. For example, say there is a program to calculate a student's grade for a semester by taking the average of all of their test scores. Further, if there was a parallel MPI program that used 10 processes to calculate 10 students' grades and collect all of the scores together to a single root process. This test suite's goal would be to ensure that all 10 processes were able to send the final grade they calculated back to the root process successfully. But the test suite would not ensure that how the processes calculated the average was correct.

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.

## 4.1 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.

In this example the MPI_Isend function begins a nonblocking send. A nonblocking function requests the MPI library to perform an operation; but, the process does not wait for the operation to complete before moving to the next line of code. Below it is shown that after a nonblocking send the process modifies the data that the MPI_Isend data buffer is referencing. This means it is possible that the data the source process had intended to send gets modified before the destination process has read it. The MPI_Wait command forces the process to wait for the specified request to complete before the process can move on. But in this case the process modified the data before waiting to see if it was safe to proceed .

```
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?
```

### 4.2 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.

Here we give an example of deadlock. Each process will calculate some value independently then desires to share the result with the process to its "left", using the point to point calls MPI_Send and MPI_Recv. Both MPI_Send and MPI_Recv are blocking calls. This means if process 0 calls MPI_Send with the destination of process 1, process 0 will wait for process 1 to call MPI_Recv before moving onto the next line of code. As seen below, both process 0 and 1 use MPI_Send to share their value with each other; but now both will block waiting for the other to call MPI_Recv. Since neither can actually move forward to the MPI_Recv call, the program is deadlocked .

```
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);
```

## 5 Approach

Consider the race condition example from earlier where the source process 0 changes the value of the variable being referenced in the MPI_Isend message immediately after sending. It is unknown if the destination process was able to read the message before it was changed. A simple test might always yield a false positive success.

But what if the destination process was intentionally stalled before reading performing the MPI_Recv. Then a test might be more likely to uncover that the data had been altered before the destination process could read it. This approach would involve intercepting all MPI calls and purposely sleeping that process to uncover deadlock or race conditions.

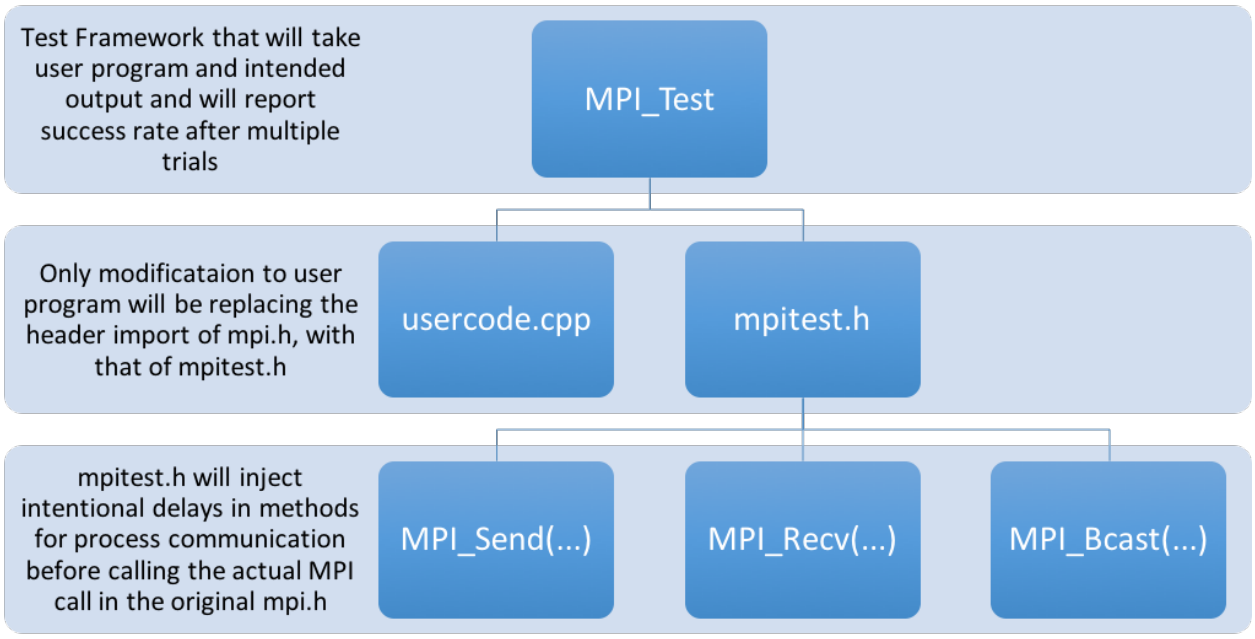
The framework would log the processes it manipulates and and report the order in which the processes arrived to the MPI calls to help developers understand why and where an error occurred. Other details gathered by the framework would be the timing of process execution during tests that run successfully. When errors occur it would capture the last message a process was able to successfully transmit/receive, and the MPI call where a process has

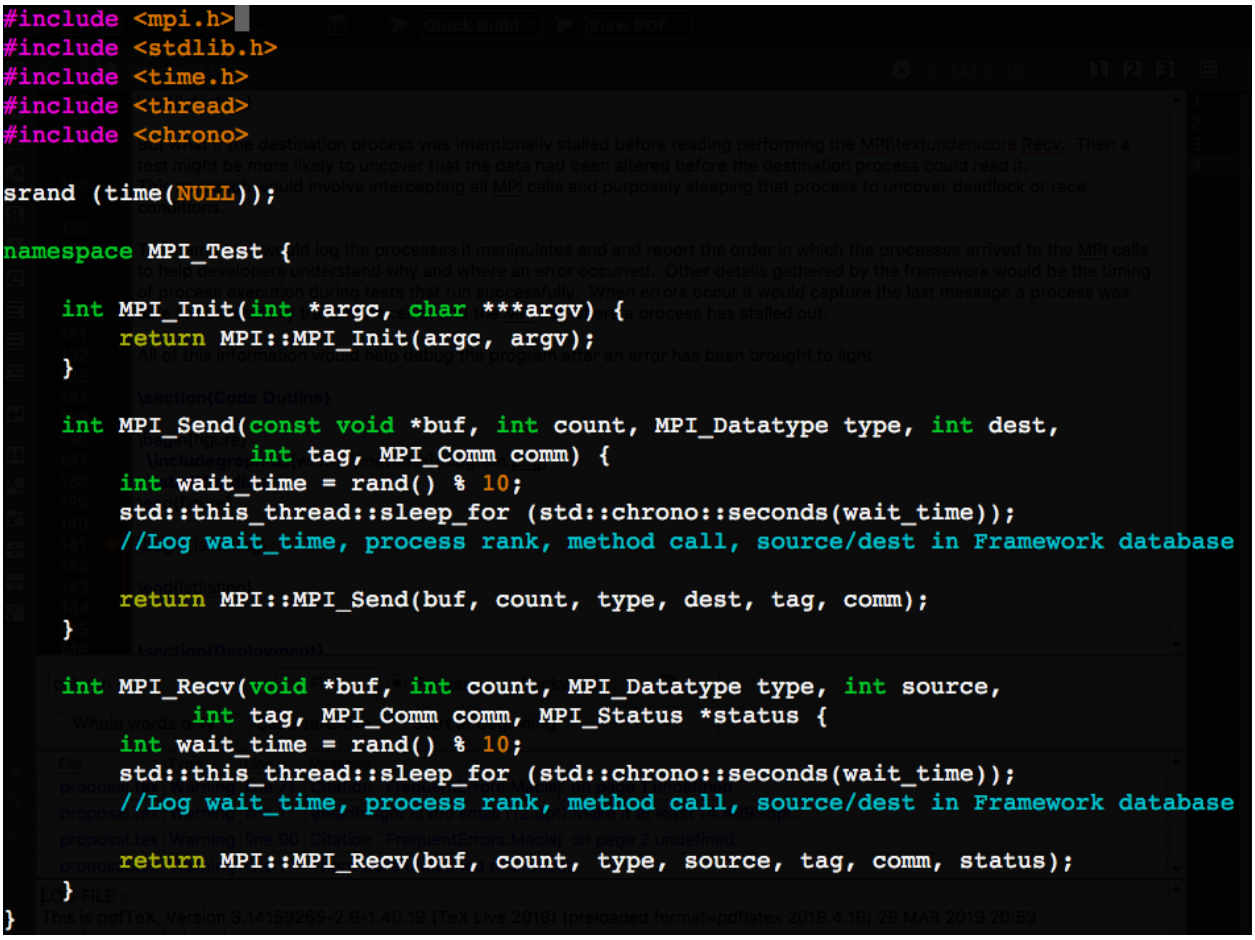
stalled out.

All of this information would help debug the program after an error has been brought to light.

# 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.


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(...)



```
#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 Evaluation

The entire goal is to find hidden errors. Thus the method of evaluation would be to test a parallel program with known errors by running it with no intervention and with intervention via the testing framework. Run it $n$ times both ways. Ideally the failure rate should go up with the process manipulation, showing that standard testing methods do not reveal errors as reliably since they do not give attention to the indeterminate nature of multiprocessing.

The other part of the evaluation would see if logging could be done to provide useful information in tracking down the source of the error. This however would have to be qualitatively analyzed.

# 8 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.

# 9 Concerns

The logs would have to be written to disk intermittently to ensure that the information is not lost if/when the error it is trying to catch occurs. Some sort of post processing would then need to be done with the saved information to generate a cohesive report for the user.

Recovering from the MPI errors within the program so that the framework can analyze what went wrong is critical for ease of use. The biggest concern is when it comes to deadlock. Firstly, identifying that deadlock occurred. After it has been identified, recovering all of the processes so that the framework can move onto the next test in the suite.