Strategies for Testing MPI Programs

Victoria Som de Cerff
vs2606@columbia.edu

April 11, 2019

Abstract

In this paper I investigate two major topics. First, a study as to what the major challenges and issues faced by developers writing multiprocessing parallel code using the MPI library. Second, what testing strategies would best assist programmers during the development stage in uncovering and debugging these errors. This research will help inform a testing framework architecture that could effectively operate in small scale environments but still locate bugs that might normally only be caught during the deployment into larger scale production.

1 Introduction

Software testing is integral to software engineering. Test frameworks and test automation have made employing good software testing practices into code bases of all sizes a lot easier. However, most these frameworks and ideologies are implicitly geared towards sequential code.

Testing and debugging multiprocess parallel codes has an added level of complexity as the same unit of code may have different behavior based on the number of processes. Further the effort is not simply multiplicative with the number of processes involved, while the processes are independent entities the coordination amongst them is user-defined and non-deterministic. Maintaining control and uniformity during test time or production time is not guaranteed. And thus tests should be developed to take that into account and leverage that fact to discover points of failure.

The purpose of this paper is to research what are the common run time issues of MPI programs, especially at scale. Furthermore, to study what testing and debugging frameworks are currently available for MPI programs and determine what work could be done to improve or augment these tools.

2 Background

2.1 High Performance Computing

As of November 2018, Summit at Oakridge National Laboratory claimed the number one spot as largest super computer in the world. It has 143.5 petaflops of compute power, which to put into perspective is almost 1.5 million times more powerful than your laptop. Summit has 2,397,824 cores and 2,801,664 GB of memory. It has 4,608 nodes composed of 2 IBM POWER9 processors per node and 27,648 NVIDIA V100 GPUS in total. The system draws 9,783 kW to power and cool the computer. [1]

With that many cores, a trivial way to utilize them would be to run something 2,397,824 times all at once. However, very few things, if any, require that level of redundancy. The real value of a machine like this is the ability to reduce the overall execution time of a program by putting, potentially, 2,397,824 cores to work in a cooperative manner. Imagine if you had a program that took 2,397,824 hours to run. You and the next ten generations of your family would have lived your lives before the program finishes. If you were to convert your algorithm to run in parallel, and have perfect scaling, with the full power of Summit your program would finish during lunch.

Super computers are used for a variety of different applications, like climate modeling, geologic subsurface modeling, high frequency trading, and film rendering. When they created the movie Monsters University in 2013, they had about 2,000 computers leveraging more than 24,000 cores in total. Even with this level of computing power it still took an entire days worth of time to render a single frame. It took over 100 million hours of pure CPU time to render the whole film. If you were to try to achieve this without supercomputers and multiprocessing it would have taken 10,000 years to produce. [2]

2.2 Code Parallelism

There are two main forms of code parallelism: mutliprocessing and multithreading. This paper focuses on multiprocessing. But it's important to understand the distinction. A process is program that executes on a computer. A thread is the unit of execution within a process. A process can have one or more threads. These threads share access to the memory and resources of a process. Note that all threads belong within a process. A process runs on a CPU. Multithreading is limited to the one CPU that process is assigned to. Multiprocessing, however, is where you have several processes running on multiple CPUs. This is how programs scale across supercomputers.

2.3 MPI: Message Passing Interface

Multiprocessing most simply is multiple processes running across multiple CPUs at the same time. It does not imply a cooperative work effort between these processes. As mentioned earlier though, being able to divide

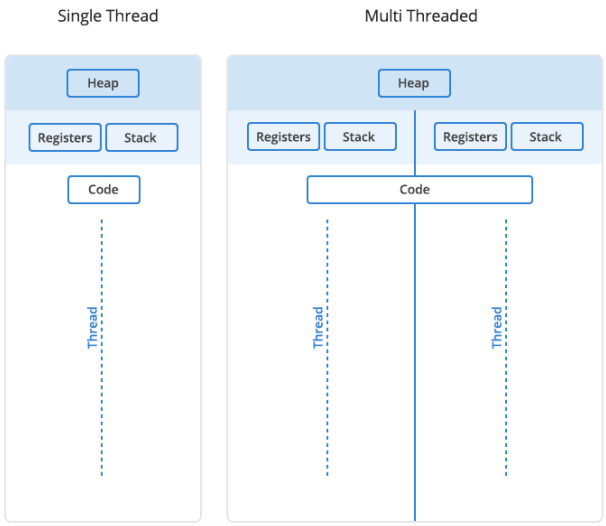

Figure 1: Example of a process with single thread, and a multithreaded process. [3]

and conquer a problem has extreme value in reducing computation time.

MPI stands for Message Passage Interface. MPI is a portable message-passing standard for implementing parallel codes on different architectures. MPI provides a means of communication for processes that are running the same code/executable in parallel. This allows for synchronization and direct message passing throughout the execution of the program [4].

The MPI library has C, C++, and Fortran implementations. mpi4py provides a python interface to the library. MPI includes a library, compiler and executable that allows a developer to specify the communication protocol between processes, and launch code with any number of processes that your hardware will support.

Note that MPI runs on any CPU machine. So you don't need a supercomputer of your own to use it. You can install and run on your MacBook even. MPI will scale to whatever number of cores it finds available on the hardware where it is installed. Lots of supercomputers are managed by schedulers. So the cores are considered a shared pool of resources and the scheduler will allocate them to users depending on how much they ask for. In this case it is common that users will request a smaller number of resources during development and testing. Then when they are ready for production runs request the full load of cores needed to run the program at intended scale.

Below is a simple Hello World example that shows how to incorporate MPI into a C++ code. This specific algorithm does not do any cooperative work. There are 4 main components [5] :

1. include mpi.h This imports all of the functions and types necessary to perform message passing

2. MPI_Init This initializes the MPI environment amongst the processes. It must be called before any other MPI calls are made. Every process must call it, but each process can only call it once.

3. After MPI_Init is called, any calls from the MPI library can be called, typically denoted by starting with "MPI_". There are point-to-point messages and group communication methods. The messages sent are arrays of fundamental types. The methods include the source or destination process. In addition to the

functions that facilitate communication, there are functions that allow the process to gain more knowledge about the environment it is operating in. For example, every process is given a rank. The rank is an integer in [0, number of processes - 1] that uniquely identifies it amongst all of the processes.

4. MPI_Finalize This is the complementary call to MPI_Init. This signals that the process is ready to terminate the MPI environment. Every process must call it before the environment will terminate.

In this example each process figures out the size of the default communicator group COMM_WORLD and what their rank is within this group [6].

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD,
        &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD,
        &world_rank);

    // Print off a hello world message
    printf("Hello world from processor
        rank %d out of %d\n",
```

```
Software Engineering $ mpicc -o hello.out hello.c
Software Engineering $ mpirun -n 3 hello.out
Hello world from processor rank 0 out of 3
Hello world from processor rank 1 out of 3
Hello world from processor rank 2 out of 3
```

Figure 2: Hello World execution with 3 processes

```
        world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

MPI_Init(int *argc, char ***argv). If NULL is passed for both argc and argv then nothing occurs. If argc and argv from main are passed into the method it will strip out the command line arguments related to MPI so that argc and argv only contain command line arguments for the actual C++ program. So in the send receive example, if argc and argv were passed then the command line arguments ”mpirun” ”-m” ”3” would be removed when the init returns.

In this execution the processes happened to print in rank order. That is not necessarily guaranteed. The processes run completely independent of each other when not making MPI calls. Who gets to the print statement first and who is given control of standard out buffer is unknown and not in contol

In more complex MPI programs the processes can send arrays of data to each other to share information that they have individually computed. A common parallel framework might be master-slave paradigm. A simplified version is shown below [6]:

```
#include <mpi.h>
#include <stdio.h>

int main ()
{
    MPI_Init(NULL, NULL);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD,
        &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD,
        &world_size);

    int number;
    if (world_rank == 0) {
        number = -1;
        for (int i = 1; i < world_size;
            ++i)
           MPI_Send(&number, 1, MPI_INT,
              i, 0, MPI_COMM_WORLD);
    } else if (world_rank > 0) {
        MPI_Recv(&number, 1, MPI_INT,
            0, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
        printf("Process %d received
            number %d from process 0\n",
                   world_rank, number);
    }

    MPI_Finalize();

    return 0;
}
```

Note that in these examples all of the processes are running the same program. However, we are able to enforce their individual behavior programatically; as seen in the Send Recv example where the control flow was dictated by the process's rank.

```
mpi $ mpicc -o send.o send.c
mpi $ mpirun -n 3 send.o
Process 1 received number -1 from process 0
Process 2 received number -1 from process 0
```

Figure 3: Send Recv execution with 2 processes

3 Problem Statement

Common goals of test suites for code bases that include MPI are the following [7] :

1. Whether the MPI can be successfully installed or not
2. Whether MPI test programs can be compiled and linked against the MPI installation
3. Whether MPI test programs run successfully and/or generate valid performance results

Item 3 captures two important and considerable tasks. First, does the parallel implementation produce the correct result. We would expect that parallel and sequential versions of a code still produce the same answer. Secondly, does the parallel implementation improve performance. Writing a parallel program often introduces additional complexity to the original algorithm. The trade off being that the increased performance yields better performance. Whether that be the ability to run a problem of equivalent size in less time or being able to evaluate problems of a larger size in without increasing the time.

As communication protocols implemented using the MPI library become more complex there is increased chance for run time failure as the code starts to scale. The 3rd goal is harder to validate without offering specific attention the MPI strategy used in the program.

Generally, parallel codes are modeled and tested on small scales initially before transitioning to full scale production runs [8]. 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. Even though synchronization points can be dispersed in the code, the behavior between those points is not uniform across all of the processes.

Thus in basic tests that are not focused on the parallelism, the execution of the code may have gotten lucky. All of the processes may have managed to arrive at communication points in the code in the right order to not trigger unrealized bad behavior. This does not exhaustively ensure that the code is correct, especially as codes start to scale and the workload per process increases. To really ensure that the third goal of correctness is met, the approach must take into account the nondeterminism of MPI programs.

4 Expectations

4.1 Personal

Overall, I want to determine a specific set of MPI failures that could be prevented or caught via static code evaluation or run time manipulation that might not have ordinarily been recognized with conventional testing.

4.2 Explicit Questions

1. What testing frameworks are available for multiprocessing?

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

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

4.3 Audience

1. Understanding of multiprocessing as a collaborative execution of multiple processes.

2. Basic MPI API usage.

3. Common pitfalls in parallel programming. Implications of pitfalls and why mitigating them is important

5 Related Works

Here is a brief summary of some research papers that have influenced the initial direction of my paper. More detail will be provided through the rest of the document.

As I began researching MPI testing, I found a lot of work on debugging tools. The goal of this research is to develop more preventative measures for locating bugs in multiprocess codes. These papers provided additional detail about the errors most often encountered in multiprocess codes and how they are typically found then subsequently resolved. I want to develop a more automated way of locating such errors before production.

Test frameworks, such as MPI Testing Tool, have been developed to test MPI library itself on different computing architectures [7]. This is out of scope for this research. The assumption here is that the MPI library is correct. The errors we are trying to detect are introduced by the developer using the MPI library and are due to how the developer chose to implement their parallel algorithm.

As learned through the semester lecture topics, sometimes simplicity is all that is necessary to meet expectations. There are advanced testing tools for MPI program like MUST. MUST is an application that incorporates asynchronous code and a shared memory model to deliver a scalable and crash safe approach to testing MPI programs [9]. Note that the shared memory model was implemented on top of MPI and was not a feature of MPI natively. But considering the timeline of this course, a key criteria in developing a new test framework was an implementation that would be overall simple to architect and require minimal changes to the code being tested.

6 Study

There are two components to the study in this paper. The first is self study, where I planned to do an evaluation on current parallel testing techniques and formulate criteria of my own for a testing framework. During this research I found "Classification of Programming Errors in Parallel Message Passing Systems" by Jan B. Pedersen [10]. Pederson conducted a semester long study of graduate students in a parallel message passing programming class that if I had the time and access to that kind of population I would have done something similar for this project. I will leverage many of his results in the later sections.

6.1 Self Study

The study I conducted myself involved research MPI programs to identify common failures related to the utilization of MPI; then differentiate failures in the following categories:

1. Identifiable during compilation or at runtime
2. Logical, protocol, timing errors
3. Blocking vs non blocking
4. Reproducibility

These categories I was able to devise even before I started the study based on my own experience writing MPI code. The errors that plague other people's implementations with MPI and even my own are very often deadlock. Three things often contribute to this: the scale of the number of processes, the complexity of process interaction, and the divergence of process activity.

The underlying hypothesis of this whole paper is that MPI errors are hard to notice and hard to find. And the purpose of the final project is to create a test framework that will notice them and help find them. This phase of the study was to implement and execute several variations of the errors identified during my research. You can visit my project github to see the list of test cases I prepared for deadlock and race conditions.
https://github.com/vsomdecerff/coms6156/
The tests are implemented using Catch, a simple unit testing framework. It has been slightly modified to run the several test cases.

It is my learnings from theses test cases that have helped shape a more concrete direction for the final project.

The tests are variations of the examples shown in this paper already that were provided by the source: [12] In addition to noticing failures that don't always readily fail, I also want to address what to do once you have found failure. As the next section will show noticing the bug is only part of the problem. Finding the source of a bug is also a critical issue for developers. I want this test framework to be able to deliver reports and information about the runtime of these parallel programs to help developers find the source of the issue with minimal code changes to original program's source code.

Something else that the test cases revealed is that some processes reported success while others reported failure. I would want to ensure that the test framework collated this information from all of the processes and reported the differences in behavior to again further help fix bugs across the entire implementation of the program.

Pasting the code snippets an screen shots for all of the tests make this paper seem really clunky so I will describe the components of the test and point to github for the source code and screen grabs.

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

src/parallel/race_condition.cpp: contains different race conditions that should all cause a failure. Tests that require a certain number of processes will fail if there are too many or to few.

src/parallel/deadlock.cpp: contains different deadlocks that should all cause a failure. Tests that require a certain number of processes will fail if there are too many or to few.

src/tests/catch_mpi_main.hpp: initializes and finalizes he MPI environment and kicks off all of the unit tests

src/tests/mpi_test.cpp: defines all of the test cases. The test cases are tagged using the square braces to group together tests that require a certain number of processes to run. [np2] means that the test needs exactly two processes. [npany] can run with any number of processes. For example: TEST_CASE("RC: Isend Irecv, array, 2

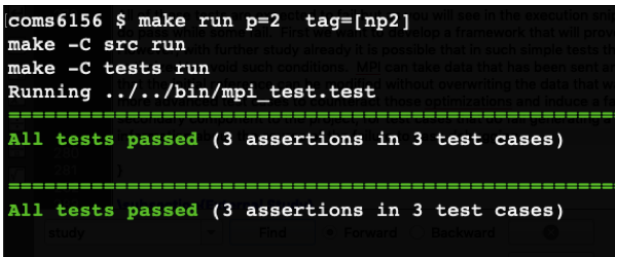

Figure 4: Same set of test cases and number of processes, all pass

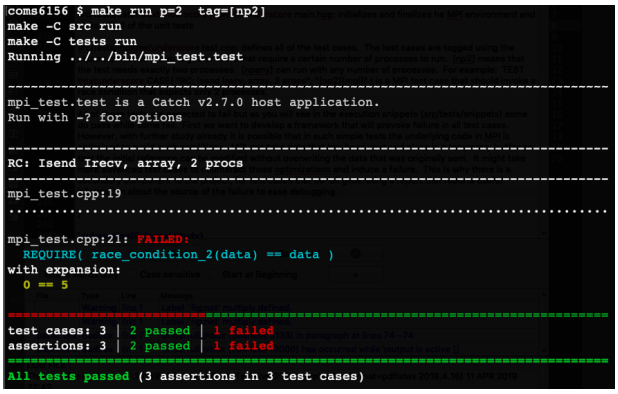

Figure 5: Same set of tests cases and number of processes, one fails

procs", "[np2][mpi") is a MPI test case that should invoke a race condition that expects only 2 processes.

-All of these tests are expected to fail but as you will see in the execution snippets (src/tests/snippets) some do pass while some fail. First we want to develop a framework that will provoke failure in all test cases. However, with further study already it is possible that in such simple tests the underlying code in MPI is optimized to avoid such conditions. MPI can take data that has been sent and place in a temporary buffer so that the initial reference can be modified without overwriting the data that was originally sent. It might take more advanced test cases to counteract those optimizations and induce a failure. This is why there is a secondary component to the project. For test cases that do fail have the application generate a report that provides useful information about the source of the failure to ease debugging.

6.2 External Study

Jan B Pedersen of the University of Nevada performed a study to aid the development, deployment, and debugging of parallel message passing systems by surveying graduate students in a parallel message passing programming class and analyzing the results [10]. Students were asked to fill out a questionaire every time they had to debug an issue for an assignment in the course. The questions were:

1. Describe the bug.
2. How did you find/fix it?
3. How long did it take?

From the results of the survey he categorized the bugs and did a deeper analysis on how these bugs were debugged. This study is of interest because my hypothesis is that a more MPI-centric testing framework can help prevent and resolve difficult bugs.

Ian Foster described a four stage model, PCAM, for developing a parallel program [11]. Partitioning, Communication, Agglomeration, and Mapping. This design was also mentioned in Pedersen's study. We focus the first two stages as these involve how the MPI library is used to develop a parallel algorithm.

Partitioning: The computation that is to be performed and the data operated on by this computation are decomposed into small tasks.

Communication. The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.

Pederson broke down these two stages of further into: algorithmic changes, data decomposition, data exchange, and protocol specification.

Algorithm changes: If parallel algorithms are based on, or derived from, existing algorithms and/or programs, then a transformation from the sequential to the parallel domain must occur.

Data decomposition: When a program is re-implemented, the data is distributed according to the algorithm being implemented.

Data exchange: As parallel programs consist of a number of concurrently executing processes that each exclusively own a partition of a problem, the need to explicitly exchange data inevitably arises. This problem does not exist in the sequential programs where all of the data exists within the scope of a single program.

An example of data exchange is imagine you were solving a jigsaw puzzle by yourself. You have a strategy for solving the puzzle and have all of the pieces to put together yourself; and it takes a day for you to complete. What if you had a friend come to help? You randomly take half of the pieces and your friend takes the other half and you both work on your respective halves of the puzzle image using the same strategy. Eventually, you will come to a point where you need some of the pieces your friend has and vice versa. At some agreed upon time you will swap pieces and finish solving your respective halves. Once you both have run out of pieces you will connect your two halves of the puzzle and it will be solved in hopefully half of the time. There were three instances of data exchange here. 1. Dividing up the initial puzzle pieces. 2. Exchanging pieces while you were solving. 3. Connecting your two solves halves together. And if you are interested in seeing this puzzle strategy happen with MPI, let me know cause I actually have a naive implementation written in python.

Protocol specification: The protocol for a parallel system is defined as the content, order, and overall structure of the message passing between communicating processes.

With my intent of developing a testing suite, trying to consider the correctness of the algorithm seems too broad a task. Data decomposition is also heavily dependent on the algorithm as well. It seems unreasonable for a test suite to flag the correctness of a parallel algorithm. A sequential algorithm may have multiple valid parallel transformations. The tasks of data exchange and protocol specification rely on the proper use of the MPI library. These elements seem best suited for a MPI-

centric test framework to effectively evaluate.

The types of bugs reported in the questionnaire were classified as one of these seven types: Data decomposition, functional decomposition, API usage, sequential error, message problem, protocol problem, and other. API usage, message problem, and protocol problem are the types of errors that fall into our target scope. They made up 20.0%, 10.32%, and 24.52% of all errors reported, respectively. Students reported an average of 52 minutes to debug each issue they encountered related to MPI. Compared to 37 minutes on average spent on bugs related to sequential code errors. Students also rated debugging as more difficult than data decomposition and function decomposition. Students weren't having challenges coming up with the parallel algorithm, but more so implementing it. This goes to show that a testing framework that could test for these kinds of errors would have a great impact on development of MPI programs.

Pedersen also gathered that print statements were used to debug 47.10% of the issues. We hope that the proposed test framework can not only catch errors but help reveal the source in a more user friendly manner.

7 MPI Errors

Presented here are three types of errors that fall under the category of data exchange and protocol specification.

As noted previously, errors in the MPI library itself are out of scope for study. Errors in the core sequential tasks performed by the processes are excluded as well. Those kinds of errors can be tested using "traditional" test frameworks.

The errors that have become the focus of the study are race conditions, deadlock, and different results [12]. 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 [8].

7.1 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 [12].

```
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?
```

Here is an example of one way the code can be modified to eliminate the race condition and produce the desired result. The source process waits to see that the message in the non blocking send was successfully transmitted before modifying the data at that location.

```
int a = 13;

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

// It is safe here to modify memory
    locations other than where a is
    stored
MPI_Wait (&req, &stat);

// It is safe to modify a only after
    all operations using it as message
    buffer have completed
a = 0;
```

7.2 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 [12].

```
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);
```

A simple fix is to use MPI_Sendrecv. This call is specifically provided by MPI for these situations and will guarantee no deadlock.

```
int a = calculate_something (my_rank)
int b;
MPI_Sendrecv (&a, 1, MPI_INT, left,
    tag, &b, 1, MPI_INT, right, tag,
```

```
    comm, &stat);
// send and receive completed; safe to
    re-use a
```

Another option would be to use MPI_Isend as seen previously. This call will not block so both processes are free to move onto the MPI_Recv call.

```
int a = calculate_something (my_rank)
int b;
MPI_Isend (&a, 1, MPI_INT, left, tag,
    comm, &req);
// MPI_Isend never blocks

MPI_Recv (&b, 1, MPI_INT, right, tag,
    comm, &stat);
// b received already

// isend not yet completed even though
    the message may have been already
    transferred
// to other_rank and the matching
    receive already completed

MPI_Wait (&req, &stat)
// Isend finally completed
```

There is also a non blocking version to receive messages, MPI_Irecv.

```
int a = calculate_something (my_rank)
int b;
MPI_Irecv (&b, 1, MPI_INT, right, tag,
    comm, &req);
// MPI_Irecv never blocks

MPI_Send (&a, 1, MPI_INT, left, tag,
    comm);
// send completed, safe to re-use a

// irecv not completed yet
MPI_Wait (&req, &stat)
// Irecv completed only now
```

7.3 Different Results/ Reproducibility

Data reduction is a concept that involves reducing a set of numbers into a smaller set of numbers via a function [6]. For example, the call MPI_Reduce using the MPI_SUM operation on the set of values will add them all together. If a set of three processes made this call and passed the value equal to their rank number, the function would yield 3. Process 0 would contribute the value 0, process 1 would contribute the value 1, and process 2 would contribute 2. 0+1+2=3.

The MPI standard does not specify the exact order in which the reduction operation is performed. Meaning the function could be 2+1+0=3 or 1+0+2=3. All the same function and result, just the operations are performed in different orders. This can depend on optimization in the computer network or the order in which the processes each called MPI_Reduce. When it comes to floating point arithmetic, order plays a role in the outcome of the equation. Since the order is not guaranteed in MPI, it is possible to see different results in reduction calls on floating point values.

In the code below, each process will calculate a floating point value and then share that value to be summed with every other processes value. The same result is likely but

not guaranteed [12].

```
float tsum = 0.0;
float inp;

inp = some_interesting_calculations();

MPI_Reduce (&inp, &tsum, 1, MPI_FLOAT,
    MPI_SUM, root, comm)
```

As a remedy, every process sends it result back to a predefined root process, typically process 0. The root process receives the values in order of process rank into an array. The root can do the summation itself in order to guarantee the same result. In this case however, now only the root process knows the result. If the other processes need to know the result then the root could call MPI_Bcast to broadcast the value to all other processes.

```
float *globinp;
float tsum = 0.0;
int np, rank, i;

MPI_Comm_rank (comm, &rank);
MPI_Comm_size (comm, &np);

globinp = (float*)malloc (np * sizeof
    (float));
MPI_Gather (&input, 1, MPI_FLOAT,
    globinp, 1, MPI_FLOAT, 0, comm);
if (rank == root) {
   for (i = 0; i < np; i++) {
      tsum = tsum + globinp[i];
   }
}
free (globinp);
```

8 Potential MPI Testing Approaches

Now that we have identified the types of MPI errors we wish to prevent. The next part of the research is how a test framework would be developed to identify them. Both of these testing methods address the API usage, message problem, and protocol problem errors noted in Pedersen's study into data exchange and protocol specification of MPI program development.

8.1 Static Analysis

Two strategies immediately come to mind. The first is static code analysis where the MPI program would be evaluated before run time against a set of coding rules. The code snippet where floating point values are operated on in a reduction call might be easily caught. But errors involving race conditions and deadlock would be much harder to detect as the communication scheme implemented by the developer becomes more complex. Looking back at the previous example of deadlock. That code snippet does not directly mean deadlock will occur.

In this example the same slice of code is used; however, by adding code in the form of a control flow statement we can avoid deadlock. Here all of the even processes will send a piece of data and block. The odd processes will block till they can read the incoming data. Every Send and Recv is matched. Then the roles reverse and the odd processes will have their turn to send a piece of data,

while the even processes receive.

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

if (my_rank % 2 == 0)
{
   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);
}
else
{
   MPI_Recv (&b, 1, MPI_INT, right,
       tag, comm, &stat);
   MPI_Send (&a, 1, MPI_INT, left, tag,
       comm);
}
```

Even in this seemingly simple example it is hard to tell if a code will deadlock without having a global view of the communication scheme. The analysis complexity will only increase as the communication between processes increases. The MPI calls and all of its arguments would have to be mapped into some sort of graph structure to track the matching source and destination calls. Then an analysis would be done on the connectedness of the graph to determine if there was any deadlock. This graph could also recognize where nonblocking calls are used to highlight areas of possible race condition.

8.2 Dynamic Analysis

This approach is focused on stress testing the MPI code to illuminate race conditions and deadlocks that might otherwise go unnoticed in small scale tests.

Now running an MPI program with more processes is one obvious way of stress testing the program. But that hardly requires a whole test framework to execute. You would simply change how you execute your test and await the result.

Change from:

$ mpirun -n 4 my_mpi_test.o

to:

$ mpirun -n 4000 my_mpi_test.o

This can be a non-starter if you don't have that many resources to begin with, and also some programs just weren't designed to run at that scale anyways. It would be more relevant to users to test a program at the scale it was intended to run at. Thus an alternative approach would be to keep the number of processes fixed but to deliberately manipulate the process execution between MPI calls.

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.

```
if (my_rank == 0)
{
   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?
}
else
{
   int b;
   MPI_Recv (&b, 1, MPI_INT, source,
       tag, comm, &stat);
}
```

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. Note that this may not be suitable in a program where timing played a role in the control logic.

9 Random Testing

The research here shifts gears into understanding how random testing can be valuable in accomplishing the proposed MPI testing strategies. Previously we proposed intentionally holding back the destination process to uncover the race condition error. In that same scenario, if we held back the source process then the error might never occur because the destination process will have ample time to call MPI_Recv and read the message. Which means our error goes undetected and we have learned nothing new. So how do we expect a test framework to know which process to stall to uncover any underlying errors that may exist in the code?

A naive approach is to try all of them, re run the program each time holding a different every process back. This can get costly as the code scales. Research done to optimize this approach by pairing down all possible combinations to "all relevant interleavings" [13]. The second most naive approach is to random pick a process. While random testing cannot ensure 100% correctness, it could provide a satisfactory amount of coverage. Random testing has proven effective in real world software [14].

One might think, if the order of processes is going to be manipulated manually then why not let them run unencumbered naturally. The speed/order of execution for MPI calls isn't guaranteed and therefore inherently random. Here we note the additional value to be gained from an MPI focused test framework. Although the execution time would still be random, the framework could 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 in both the controlled setting and where the program runs normally. 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.

This is certainly not an exhaustive strategy, as it is hard to be exhaustive when you consider the thousands of processes these programs can scale up to. However, the goal would be to reduce false positives.

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

9.1 Automated Test Generation

Some consideration was given to the idea of automated test generation. A strategy is inspired by the research done for Sapienz [15]. Sapienz is a code for automated testing of Android applications. One feature of Sapienz is that that extracts statically defined string constants from the APK. Further, it uses those strings a seeds for input into the test sequences. The initial idea was that the code would be statically analyzed to extract source and destination tags from the MPI calls and possibly locate control flow statements that are based on rank. Then consider those processes "of interest" and target those processes for manipulation. See the comments in the code below.

```
int number;
 if (my_rank == 0) { // process 0 is
     considered "of interest" from
     this in control statement
     number = -1;
     for (int i = 1; i < world_size;
         i+=2)
        MPI_Send(&number, 1, MPI_INT,
            i, 0, MPI_COMM_WORLD); //
            odd numbered processes
            are the destination and
            thus "of interest"
}
else if (my_rank%2 == 1) {
    MPI_Recv(&number, 1, MPI_INT,
        0, 0,
        MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        // process 0 is the source
        of this message and thus "of
        interest"
}
else { }
```

However, after further analysis this idea seemed to only increase the complexity of the testing framework and not add any additional benefit beyond the idea of randomly sampling the processes to manipulate. Developing a tool to extract the "of interest" processes statically could be just as effectively done by having an intermediate library that would a wrap around all of the MPI communication calls. The intermediate library could intercept the messages and note the source and destination processes. This effectively boils down the Dynamic Analysis approach already described.

9.2 Debuggers

There are commercial tools for debugging MPI programmers, like TotalView. However, these commercial tools are rather expensive. For example the TotalView license charges you per process you want to scale up to when running the debugger. This test framework firstly intends to be free. Also it is not meant

to be a "debugger" per se, but provide useful information during the act of debugging.

10 Conclusion

When it comes to writing multiprocessing code, developing the parallel communication strategy is only half the battle. This is supported by the study where students indicated that debugging was the most complex aspect of engineering a program using MPI, spending almost an hour on average per bug. An MPI specific testing framework would firstly provide information useful to aid developers during debugging. But I believe an equally important benefit is that it would be able to trigger and document conditions that code might not have been experienced due to the scale of performance during normal testing procedures.

10.1 Future Work: Final Project

In the final project I would want to take the examples of race condition and deadlock and write a suite of tests that actually demonstrate the unstable nature of using standard testing tools that do not take MPI into consideration. The hypothesis here would be that this control group would yield a high rate of false positive of test case passing. Once this control group has been established, I would write an intermediate library that wraps around the MPI library to allow the test framework to control when a process actually reached the message passing communications. The goal of the framework would be to decrease the rate of false positives and highlight the failure in the protocol specified by the programmer. The framework would then generate a report to augment the stack trace that would help the user locate the root of the error.

References

[1] TOP500.org, "Top500 list," tech. rep., Nov 2018.

[2] chabowski, "How hpc impacts our lives ii: Hpc (and linux) in the movies." `https://www.suse.com/c/hpc-impacts-lives-ii-hpc-linux-movies/`, Nov 2016.

[3] R. Bauer, "What's the diff: Programs, processes, and threads." `https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/`, Nov 2017.

[4] M. P. I. Forum, *MPI: A Message-Passing Interface Standard*, 3.1 ed., June 2015.

[5] A. I. El-Nashar and N. Masaki, "To parallelize or not to parallelize, bugs issue," *International Journal of Intelligent Computing and Information Science, Egypt*, vol. 10, July 2010.

[6] W. Kendall, "Mpi tutorial." `http://mpitutorial.com/tutorials/`, 2019.

[7] J. Hursey, E. Mallove, J. M. Squyres, and A. Lumsdaine, "An extensible framework for distributed testing of mpi implementations," in *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'07, (Berlin, Heidelberg), pp. 64–72, Springer-Verlag, 2007.

[8] I. Laguna, D. H. Ahn, B. R. de Supinski, T. Gamblin, G. L. Lee, M. Schulz, S. Bagchi, M. Kulkarni, B. Zhou, Z. Chen, and F. Qin, "Debugging high-performance computing applications at massive scales," *Commun. ACM*, vol. 58, pp. 72–81, Aug. 2015.

[9] J. Protze, T. Hilbrich, M. Schulz, B. R. d. Supinski, W. E. Nagel, and M. S. Mueller, "Mpi runtime error detection with must: A scalable and crash-safe approach," in *2014 43rd International Conference on Parallel Processing Workshops*, pp. 206–215, Sep. 2014.

[10] J. B. Pedersen, "Classification of programming errors in parallel message passing systems," in *CPA*, 2006.

[11] I. Foster in *Designing and Building Parallel Programs*, ch. 2.1, Addison-Wesley Inc., Argonne National Laboratory, and the NSF Center for Research on Parallel Computation, 1995.

[12] C. Golebiewski, Maciej (IM&T, "Frequent parallel programming errors." `https://confluence.csiro.au/display/SC/Frequent+parallel+programming+errors`, Nov 2012.

[13] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Palmer, R. Thakur, and W. Gropp, "Practical model-checking method for verifying correctness of mpi programs," in *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*,

PVM/MPI'07, (Berlin, Heidelberg), pp. 344–353, Springer-Verlag, 2007.

[14] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering*, vol. 38, pp. 258–277, March 2012.

[15] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), pp. 94–105, ACM, 2016.

[16] G. Gopalakrishnan and R. M. Kirby, "Formal methods for mpi programs," *Electronic Notes in Theoretical Computer Science*, vol. 193, pp. 19 – 27, 2007. Festschrift honoring Gary Lindstrom on his retirement from the University of Utah after 30 years of service.

[17] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "Mpicheck: a tool for checking fortran 90 mpi programs," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 93–100, 02 2003.

[18] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 201–211, Nov 2015.

[19] P. García-Risueño and P. E. Ibáñez, "A review of high performance computing foundations for scientists," *International Journal of Modern Physics C*, vol. 23, no. 7, 2012.

[20] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. J. Li, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978 – 2001, 2013.