Security in High-Performance Computing Using MPI Calls

Brian Meginness

CS 4960

Professor Thomas

Abstract

High performance computing environments have challenging security requirements and as such need a creative security solution. One solution is to use machine learning to analyze the performance logs of the computers' Message Passing Interface (MPI) calls to determine what normal behavior looks like and with that identify any abnormal usage of the system. This paper explains the general history of MPI along with a selection of its commands — *MPI_Send, MPI_Recv, MPI_Gather, MPI_Waitall,* and *MPI_Testall* — and describes how DeMasi et al. use it for anomaly-detection in a way that could be employed for securing these systems.

1. Introduction

High performance computers (HPCs) are systems comprised of thousands of computers running in parallel, such as Lawrence Livermore National Labs' "Sierra", which is capable of achieving speeds up to 94.6 PFLOPS, meaning it can perform 94.6 quadrillion floating point operations per second[10]. Systems such as these present certain security complications that are not present in everyday systems, detailed below, which means that typical security measures cannot be employed. Because of this, new techniques and strategies must be developed to protect HPC systems. One such technique, developed by DeMasi, Samak, and Bailey[3], is to analyze the Message Passing Interface (MPI) performance logs using machine learning algorithms to identify

and flag aberrant programs running on HPCs. This paper intends to provide an overview of what MPI is and how DeMasi et al. utilize it in their approach.

2. What Makes HPC Security Different

2.1 Nonstandard Design

Before beginning, the abnormal security complications present in HPC systems need to be explained. HPCs vary from traditional computers in several ways, and with those differences come unique security needs. One significant way that HPCs are different from traditional is that, while most computers will be using one of a couple different operating systems — such as Windows, MacOS, or Unix — HPC operating systems are often customized to their specific needs and can vary wildly from system to system[4]. Because of this, the way programs for these systems are written are drastically different and making a single tool, or even a suite of tools that support all or most HPCs would be a daunting task at best and impossible at worst.

Operating Systems aren't the only thing that vary from system to system in HPC environments. Because each system tends to be developed in isolation from others, the scripts they run, such as one to check if a node is ready to run, are often written equally non-collaboratively[1]. This means that two scripts that do the exact same thing on two different systems may be entirely different under the hood with different algorithms and coding techniques that reflect the personnel that wrote them, not a community standard[1]. As a result, just as with a system's stack, there is no "one size fits all" for detecting malicious code on HPCs.

2.2 Unique Environments

In addition to system variance, the actual use cases for HPCs add yet another layer of difficulty to security. The sheer number of floating point operations run on HPC systems can reach

in the several hundreds of quadrillions every second[7], which creates a serious problem for most conventional security solutions[8]. Very few continuous monitoring tools have the capability of reporting to the degree of accuracy and validity necessary for HPC systems when working at these speeds, and while risk-based security is highly accurate, the lack of immediate response would mean that some security threats could be missed entirely[8].

Lastly, many HPC systems are not used exclusively, or even mainly, locally. With little to no verification process, some HPCs can be accessed by researchers from across the world who simply set up some configuration files and have the system perform their operations[4]. There do exist numerous HPCs that don't have this level of accessibility, often these are systems that work on highly confidential data such as those operated by the United States Department of Defense, but even in these cases the systems are mainly accessed remotely. Obviously, for the rest of HPCs, this openness causes great risk on its own, but additionally, roughly eighty percent of HPC vulnerabilities are a result of misconfiguration[8]. By allowing just anyone to configure an HPC system, the odds of such a misconfiguration increase greatly.

However, not all differences from standard computers make security harder. There are also certain aspects of HPC systems that allow for techniques that would be impossible, or at least impractical, on regular systems. Most significantly, HPCs operate using very few scripts, such as the National Energy Research Scientific Computing Center (NERSC) HPC that completed half of its workload in 2014 using only thirteen different scripts, and 80% of it with only 50[5].Compare that to a traditional computer used in a home or office, which has almost certainly utilized more than 50 scripts before even finishing its boot process. The result of this is that, despite each HPC possibly having its own 50 unique scripts, an individual system will only be running a handful that a security tool will need to track.

3. An Overview of MPI

3.1 What is MPI?

Now that this paper has explained the necessity of alternative security measures for HPCs, next the tools that will be used to realize them need to be discussed. Specifically, MPI — what it is and how it works.

MPI is a message passing specification that was initially written in 1994 by a group of computing researchers known as the MPI Forum, in conjunction with the wider high performance computing community, as a way to standardize messaging across parallel and distributed computing systems[2]. Since then, MPI has undergone several updates including new functionality, extensions, and incorporating past errata, and has recently been updated to MPI 4.0 on June 9, 2021[9].

MPI's main design goals are ease of use and portability[9]. In parallel and/or distributed environments, such as HPC environments, having a standard in place to determine how each computer sends information to another is vital. Additionally, as they are the most commonly used languages for the desired systems, MPI is designed specifically to be compatible with FORTRAN and C, and by extension C++[9]. However, many MPI implementations include bindings for other languages, such as Open MPI supporting Java, and even more third party language bindings exist beyond what is implemented out of the box[5].

3.2 MPI Commands

This paper has explained, briefly, what MPI is and what its purpose is, but how does it work? Because MPI is a specification, not a functional library, explaining its commands is a somewhat complicated task. For this purpose, this paper will be using examples from the C binding

of Open MPI, an open source MPI implementation. Also, because there are over three hundred commands in the MPI specification[6], only a selection from the subset of commands utilized by DeMasi et al. in their research will be overviewed[3]. These will be *MPI_Send*, *MPI_Recv*, *MPI_Gather*, *MPI_Waitany*, and *MPI_Test* in order to have at least one command from each subset of functionality: sending, receiving, collective, and other miscellaneous commands. These commands specifically were chosen because they represent the core functionality of MPI implementations, which is why, for example, *MPI_Send* was chosen instead of *MPI_Rsend*, which is a derivative thereof[9].

3.2.1 MPI_Send

First up is *MPI_Send*, which preforms a standard-mode blocking send[6]. In the MPI specification, the standard communication mode is when the program will automatically determine whether a send will be buffered — that is, whether the message is sent to the send/receive buffer so that the call can complete before being received or if the call must wait to complete until a matching receive is posted[9]. The purpose of the buffer in MPI is to combine multiple messages and send them all at once as one large message, thereby decreasing the number of signals that must be sent. A blocking procedure is one that is non-local, meaning it requires a related MPI procedure to be called in another process, or one that is completing, which requires at least one associated operation to reach completion before returning, or both. To contrast, and because it is an important part of MPI procedures that would otherwise go unmentioned, a nonblocking procedure is one that is both local and incomplete, the logical opposites of non-local and complete, respectively. Put it together and it means that *MPI_Send* will send a message, either to the buffer or directly to a matching receive, and will only return once the message has reached its desired destination[9].

For a more in-depth explanation of *MPI_Send*, the function prototype provides further details. Again, because MPI itself is just a specification, this paper will be using its C binding. While the syntax will vary, the data contained in parameters themselves will largely remain consistent across all bindings.

```
#include <mpi.h>
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)
```

[6] https://www.open-mpi.org/doc/v4.1/man3/MPI_Send.3.php

MPI_Send takes six parameters as input, **buf*, *count*, *datatype*, *dest*, *tag*, and *comm*. The first three comprise the data part of the message and the last make up what is called the "message envelope"[9]. Taking these in order, **buf* is the initial address in the send buffer[6]. As discussed previously, the send buffer where messages are collected before all being sent together. Next is *count*, which specifies how many elements the message contains, this parameter must be non-negative but can be zero, in which case the data part of the message is left blank[9]. The reason count specifies the number of elements in a message, not the number of bytes, is because MPI is designed to be machine independent and to function on the application level[9]. What determines the size of these elements is *datatype*, which itself has a unique datatype of *MPI_Datatype*. The exact possible values for this parameter will vary between implementations, but they will always correspond to the language's basic datatypes, such as *MPI_CHAR*, *MPI_INT*, *MPI_FLOAT*, and *MPI_C_BOOL* for C, among others. Uniquely, *MPI_BYTE* and *MPI_PACKED* do not correspond to a datatype in any implementation, rather, *MPI_BYTE* is an uninterpreted set of 8 bits and

MPI_PACKED is itself a contiguous buffer as one might use in various other communication libraries[9].

The next three parameters are what the MPI Forum refers to as the "message envelope" and contain identifying information about the message[9]. The destination is specified by *dest*, an integer that represents the target's rank within its group. The group can be of processes or of other systems if working in parallel or distributed environments, as specified by the communicator. More on that later. The *tag* argument is an identifying number for the message and can be set as any positive integer less than *UB*, where *UB* is an implementation dependent value no less than 32,767[9]. Lastly *comm*, as was briefly alluded to before, specified the message's communicator. The communicator specifies the context of the message: the set of processes that can receive the communication and what the nature of those processes are, even if the local process is not within the message's target context[9]. It is also a unique datatype, *MPI_Comm*, which can be either one of MPI's predefined communicators such as *MPI_COMM_WORLD*, or *MPI_COMM_SELF*, which are a communicator that include all processes the local process can communicate with and a communicator that contains only the local process itself, respectively, or other implementation specified communicators[9].

3.2.2 MPI_Recv

Of course, *MPI_Send* is worthless without an associated *MPI_Recv* to receive the message. *MPI_Recv* preforms a standard-mode blocking receive which, like *MPI_Send*, means it is a blocking process, but in this case, it being standard-mode means that the message it should expect

to receive may or may not have a buffer[6]. Put simply, as their names imply, *MPI_Send* sends the message from one process and *MPI_Recv* receives it on another.

```
#include <mpi.h>
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Status *status)
```

[6] https://www.open-mpi.org/doc/v4.1/man3/MPI_Recv.3.php

MPI_Recv takes seven parameters, **buf*, *count*, *datatype*, *source*, *tag*, *comm*, and **status*[6]. Unlike *MPI_Send*, which only had input parameters, **buf* and **status* are output parameters and should be passed by reference instead of value. Otherwise, the parameters shared with *MPI_Send*, **buf*, *count*, *datatype*, *tag*, and *comm*, serve the exact same purpose as before, representing the address of the receive buffer, the number of entries and their types, the unique identifier of the message, and the communicator. The only new input parameter is *source*, which is an integer representing the rank of the message's source in the same but opposite way as *MPI_Send*'s *dest*. The wildcard values *MPI_ANY_SOURCE* and *MPI_ANY_TAG* can be used in place of a specific source or tag to indicate that a message with any source or tag from the specified communicator is acceptable to be received[9]. Because the source or tag of a message may not be known due to wildcards, or in the event that multiple messages are received at a time that need to be uniquely identified **status* is returned containing the message's original source, tag, and an integer error code[9]. In C, **status* is a structure that contains these values as *MPI_SOURCE*, *MPI_TAG*, and *MPI_ERROR*.

3.2.3 MPI_Gather

In the event that you have multiple processes all with one or more messages in their send buffers, you'll likely want to have a way to collect those messages in one place. This is the purpose

of *MPI_Gather*. *MPI_Gather* works by simulating an *MPI_Send* call in all processes with a *dest* of the root process and an *MPI_Recv* in said root for each process[9].

```
#include <mpi.h>
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm)
```

[6] https://www.open-mpi.org/doc/v4.1/man3/MPI_Gather.3.php

MPI_Gather's prototype, as one might expect, looks largely like a combination of the *MPI_Send* and *MPI_Recv* parameters, just this time with prefixes to indicate which one they're for. As such, **sendbuf*, *sendcount*, *sendtype*, **recvbuf*, *recvcount*, *recvtype*, and *comm* all function at least almost identically to how they would if *MPI_Gather* was actually a true combination of the two calls[6]. One thing to note about *sendcount* and *recvcount* is that they are the counts for each individual send and receive, not the total, so if *sendcount* = 5 each process in the same context as the *MPI_Gather* call would be sending with a count of five[9]. The new parameter introduced by *MPI_Gather*, *root*, specifies the rank of the process that is the root — that is, what process is the one receiving the messages. This parameter behaves slightly different in parallel or distributed environments, in which case one group defines the root process and all but the root in that group pass *MPI_PROC_NULL*, a "dummy" process, the root process passes *MPI_ROOT*, a unique value specifically for this circumstance, and all processes of the other group(s) pass the rank of the root in the group that defined it[9].

3.2.4 MPI_Waitall

If you have a set of MPI calls that absolutely must be completed before another procedure can begin, *MPI_Waitall* is the tool for the job. As it says on the tin, *MPI_Waitall* waits for all of a

specified collection of procedures to finish before returning. As such, it is a blocking procedure and is frequently used along with nonblocking procedures to ensure a proper order of operation[9].

```
#include <mpi.h>
int MPI_Waitall(int count, MPI_Request array_of_requests[],
    MPI_Status *array_of_statuses)
```

[6] https://www.open-mpi.org/doc/v4.1/man3/MPI_Waitall.3.php

MPI_Waitall has only three parameters: *count*, *array_of_requests[]*, and **array_of_statuses*[6]. While fairly self-explanatory, *count* represents the number of requests to be specified. Almost equally as self-explanatory if not for the nature of requests themselves is *array_of_requests[]*, which, as the name implies, contains an array of *MPI_Requests*. An *MPI_Request* is a unique type generated by the return of nonblocking procedures such as *MPI_Isend* and *MPI_Irecv*, the nonblocking counterparts of *MPI_Send* and *MPI_Recv*[9]. The last parameter, **array_of_statuses*, is an array containing the same *status* structures, or whatever data structure the implementation uses, that were discussed before in the receiving procedures. The indices of *array_of_requests[]* and **array_of_statuses* are identical to each other for a given request — that is, *array_of_requests[3]* is the request that generates the *status* in **array_of_statuses[3]*[9]. Because send procedures don't return a status, if one or more of the requests is from a send the associated *status* will be undefined except in the case of an error[9].

3.2.5 MPI_Testall

Last on the list is *MPI_Testall*, which functions similarly to *MPI_Waitall*. The key difference is that *MPI_Testall* is a nonblocking procedure that only returns a Boolean value indicating whether the set of procedures have finished[6]. As such, where *MPI_Waitall* is used to control flow and execution order, *MPI_Testall* is instead a validation tool[9].

```
#include <mpi.h>
int MPI_Testall(int count, MPI_Request array_of_requests[],
    int *flag, MPI_Status array_of_statuses[])
```

[6] https://www.open-mpi.org/doc/v4.1/man3/MPI_Testall.3.php

Like *MPI_Waitall*, *MPI_Testall* has the parameters *count*, *array_of_request[]*, and *array_of_statuses[]*, which all function identically as before. In addition, **flag* is the Boolean value that indicates whether or not the requests are all completed at the time of the call[9].

4. DeMasi et al.'s Solution

Having now established at least the most basic concepts of MPI and HPC security, this paper can now conclude with how DeMasi, Samak, and Bailey approached the problem. In order to classify the procedures of a node, they looked at two main sets of data: how the node spent its time and the performance signature of the node's MPI calls[3]. How the node spends its time is determined by measuring the percentage of time it spends in user, system, and MPI calls. The performance signature of the MPI calls grouped the commands used by those that were blocking, nonblocking, sending, receiving, point to point, collective, or other miscellaneous calls. Performance was measured for each by the amount of time the node spent in a call of the associated group, the number of calls in that group, the percentage of MPI time those calls accounted for, and the percentage of wall time (real-time) those calls accounted for[3]. This also served as a way to track what calls were being made by a given node, such as whether it used *MPI_Gather* or not.

4.2 Results

Having compiled the dataset classifying each node, they created a supervised machine learning algorithm using a rule ensemble that predicts whether an observation is or is not within a certain set of parameters[3]. Each rule in the ensemble was used to provide a single Boolean value

for characteristics of the observation; one might determine whether it contains *MPI_Send*, another if it spends more than 50% of its time in system calls. By evaluating the final result of truth values regarding an observation's characteristics and checking them against the training data, the algorithm could classify what code was running[3]. However, due to the nature of the algorithm not allowing for uncertainty, any code that was not in the training data was misclassified as one that was and, as a result, the model had an error of approximately 5%. By allowing the algorithm to leave observations unclassified if it could not confidently place it, this number dropped under 3% with less than 5% of the observations remaining unclassified[3].

As a secondary result, the researchers also found that this information could be used to determine which of their considered datapoints had the highest, or lowest, deterministic power. It was found that the MPI time a node spent in *MPI_Waitall* was the most significant factor in classifying code, where the wall clock time spent in collective calls was the least deterministic[3]. Interestingly, four of the top nine attributes were regarding the percentage of time spent in various MPI calls despite six out of the eight least important attributes being based on the time spent in certain MPI calls[3]. While one might initially think to use this as a sign that certain MPI calls are more deterministic than others, metrics based on *MPI_Test* can be found in both lists.

5. Conclusions

Because HPCs tend to spend much of their time in only a handful of codes, anomaly-based detection is a far more viable option than in traditional environments. As such, using DeMasi et al.'s method of code classification, in which anomalous processes on HPCs can be automatically identified with great accuracy, could allow for greater efficiency than other security methods. Additionally, because it analyzes performance logs and not the processes themselves, this method

more is lightweight and efficient than conventional approaches, something of great importance in HPC environments.

5.2 Reflections

In writing this paper, the core aspect of what I learned was in regard to MPI. Notably, I learned everything I now know about MPI, much of which I have shared in the preceding sections. While most of what I wrote about HPCs is information I didn't have beforehand, it's largely the details specified, not the general concepts, that I learned in the course of this paper. I had at least minimal, vague knowledge of the core aspects of what I discussed, such that if someone were to have explained the details I wrote in this paper to me before I started this research I would have at least been able to follow along. An example of this is that, while I had never read anything on the topic, had I been told that HPCs run nonstandard and often customized operating systems it would have made perfect sense to me and largely been what I would have expected. I did, however, find it fascinating that HPCs tend to operate using very few different scripts, making common usage patterns predictable.

Having concluded my research for this paper, I am unaware if the security technique discussed has ever been utilized. This does make me question how effective it would actually be — this method was first proposed in 2017 and was conducted in association with the NERSC, if it really was a groundbreaking innovation wouldn't it be common practice four years later? It might be, of course, but I was unable to find any HPC facility foolish enough to have made their security measures publicly available as any organization concerned with security will not freely disclose said security measures.

References

[1] Allcock, W., Felix, E., Lowe, M., Rheinheimer R., and Fullop, J. 2011. Challenges of HPC monitoring. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. Association for Computing Machinery, New York, NY, USA, Article 22, 1–6. 10.1145/2063348.2063378

[2] Austin, B. et al. 2014 NERSC Workload Analysis. 2015. US Department of Energy. Retrieved October 10, 2021 from http://portal.nersc.gov/project/mpccc/baustin/NERSC_2014_Workload_Analysis_v1.1.pdf.

[3] DeMasi, O., Samak, T., and. Bailey, D. H. 2013. Identifying HPC codes via performance logs and machine learning. In *Proceedings of the first workshop on Changing landscapes in HPC security (CLHS '13).* Association for Computing Machinery, New York, NY, USA, 23–30. 10.1145/2465808.2465812

[4] Peisert, S. 2017. Security in High-Performance Computing Environments. *Communications of the ACM 60*(9), 72-80. 10.1145/3096742

[5] Software in the Public Interest. Frequently Asked Questions. Retrieved October 8, 2021 from https://www.open-mpi.org/faq/

[6] Software in the Public Interest. Open MPI Documentation. Retrieved October 8, 2021 from https://www.open-mpi.org/doc/v4.1/.

[7] Riken Center for Computational Science. About Fugaku. Retrieved October 11, 2021 from https://www.r-ccs.riken.jp/en/fugaku/about/

[8] Malin, A., and Van Heule, G. 2013. Continuous monitoring and cyber security for high performance computing. In *Proceedings of the first workshop on Changing landscapes in HPC security (CLHS '13)*. Association for Computing Machinery, New York, NY, USA, 9–14. 10.1145/2465808.2465810

[9] MPI Forum. 2021. MPI: A Message-Passing Interface Standard Version 4.0. https://www.mpi-forum.org. (2021)

[10] Strohmaier, E., Dongarra, J., Simon, H., and Meuer, M. 2021. June 2021 TOP500. Retrieved October 19, 2021 from https://www.top500.org/lists/top500/2021/06/