MPIGDB: A Flexible Debugging Infrastructure for MPI Programs

Robert Underwood, Bogdan Nicolae

MPIGDB: A Flexible Debugging Infrastructure for MPI Programs

Robert Underwood
runderwood@anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Bogdan Nicolae
bnicolae@anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

ABSTRACT

The advent of flexible science workflows that span traditional HPC, the cloud, and beyond requires more than ever efficient, scalable, portable, featureful debugging tools. This work presents the design and implementation of MPIGDB a flexible debugging infrastructure for MPI programs. MPIGDB is designed to be highly capable at scale, available across platforms, easily interactive, and an extensible way to debug distributed MPI programs. This work demonstrates the scalability of this our approach to 128 processes on debugging memory access violations in a heat diffusion code as an example use case while providing features competitive with proprietary debugging tools such as GPU kernel debugging, mixed language support, and scripting abilities. Together these tools allow users to debug quickly challenges wherever their science is run.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Massively parallel and high-performance simulations.*

KEYWORDS

Debugging, MPI

ACM Reference Format:
Robert Underwood and Bogdan Nicolae. 2023. MPIGDB: A Flexible Debugging Infrastructure for MPI Programs. In *Proceedings of the 13th Workshop on AI and Scientific Computing at Scale using Flexible Computing Infrastructures (FlexScience '23), June 20, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3589013.3596675

1 INTRODUCTION

Scientific computing workflows are beginning to utilize increasingly diverse computing environments. Flexible computing infrastructures leverage increasingly heterogeneous system architectures with diverse hardware both within (e.g. CPU+GPU) and between nodes (e.g. different types of CPUs between nodes), heterogeneous software stacks (e.g. high-level languages such as Python or Julia with low-level languages C++ or Cuda), running across distributed computing environments, on a mixture of transitional HPC, cloud, and edge devices. All of this heterogeneity of system architectures presents the possibility for complex software bugs to be surfaced through the interactions of these components that require flexible visibility into the entire stack across distributed nodes to isolate and debug these bugs quickly.

Scientists need a *capable*, *available*, *extensible*, and *interactive* tool that can be used to debug *distributed* workflows on heterogeneous HPC systems. A key requirement is a method to interrogate their systems in order to track down complex software bugs across multiple layers of the stack and multiple nodes. To this end, they need access to read and potentially modify the distributed state of their workflow at once. State-of-art tools either do not provide sufficient introspection ability across the layers of the software stack and the heterogeneous hardware, or are not available on all platforms due to licensing or software limitations. Furtheremore, they are non-extensible (making it difficult to quickly automate new tasks as users encounter them), lack sufficient interactivity (making it more time consuming to explore complex system interactions), or lack the ability to correlate complex issues across distributed nodes. Thus, designing a debugging environment for flexible computing is challenging.

Without a view into the state on distributed compute nodes, users will fail to quickly determine the root cause of a problem influenced by the behavior of a process on another node. For the purpose of this work, we focus on easy-to-quantify bugs to enable direct comparisons with state-of-art approaches. However, it is important to note that our proposal can be used to identify several complex and pathological bugs in distributed systems quickly and efficiently. For example, we have used this tool to: (1) detect a missed MPI collective call caused by a thrown C++ exception caught before termination which causes the application to hang; (2) fix a hang triggered by a process waiting to be notified to flush data; (3) fix a use after free error in mixed python and C++ code; (4) identify the root cause of a hang in a distributed system as an improperly configured network connection in a server-client program. All these behaviors are difficult to debug without the ability to inspect a distributed state, and our tool was successful in facilitating this capability.

This work introduces `MPIGDB`, an open source[1], *capable*, *available*, *interactive*, and *extensible* tool for debugging *distributed MPI applications* across this diverse computing ecosystem at scale to address these challenges posed by flexible science applications. Our tool exposes powerful *capabilities* in GDB to tackle diverse hardware by providing a consistent method to print variables throughout the stack and get backtraces on both serial and multi-threaded CPU codes as well as GPU kernels on diverse systems. It leverages capabilities to see stack traces of high-level languages like Julia and Python together with low-level library context from C, C++, or Cuda codes to trace issues across multiple levels of the stack. It is easily *extensible* with both scripts and command extensions to productively and automatically address common tasks and to more

[1] https://github.com/robertu94/mpigdb

FlexScience '23, June 20, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0166-5/23/06…$15.00
https://doi.org/10.1145/3589013.3596675

easily span the high-level to low-level language divide. It is *available* without the high cost or relying on nonstandard functionality. It also provides key *interactive* capabilities allowing the scientists to explore different aspects of their software dynamically as they root cause bugs in their codes. Lastly, it tackles the key challenge of debugging *distributed* MPI programs by introducing a portable startup mechanism combined with extensions to more easily debug MPI programs through a single interface with up to 128 processes.

Our key contributions are the implementation of `MPIGDB`, bringing highly capable, available, interactive, and extensible debugging to distributed MPI programs in a way that scales to 128 processes – existing tools have many of these attributes, but `MPIGDB` uniquely offers them all for distributed programs. We accomplish this by providing a novel portable mechanism to attach a debugger instance to a collection of MPI ranks and a series of debugger extensions to make facilitate work with large collections of processes and a command line utility that ties these seamlessly together. Lastly, we demonstrate the scalability and interactivity of our approach with a case study on a memory access violation in a MPI program.

2 BACKGROUND

In this section, we describe commonly used and state-of-the-art tools for debugging MPI programs and we define the criteria of capability, availability, extensibility, and interactivity of distributed processes and evaluate the various approaches by them. After that, we highlight the capabilities included in the GDB that allow it to be converted into a distributed debugger with the power to span heterogeneous hardware and software stacks and finally describe the current state of the art for attaching debuggers to MPI processes.

2.1 Debugging for MPI Programs

We highlight the most common approaches to debugging MPI programs at scale and consider how they address issues of capability, availability, extensibility, and interactivity when debugging MPI distributed programs. Users take a variety of approaches to debugging MPI programs including serial debuggers, parallel debuggers, specialized memory access tools, and tracing tools.

We rank and summarize the various approaches based on capability, availability, extensibility, and interactivity for distributed programs in Table 1 as none, low, **mid**, or high.

Tools that have high **capability** allow detailed and arbitrary inspection and possibly modification of system state across CPUs and GPUs and across high and low-level language boundaries. Tools with **mid** capability have some of these abilities, and low capability tools cannot access the arbitrary state of programs on their own. Tools that have high availability are easy and free to install as an unprivileged user on a system whereas low availability tools lack one of these. Tools are **extensible** if the tool is open source and users can provide capabilities to automate tasks; tools are **mid** extensible if it lacks one of these or if the user interface degrades regularly when this functionality is used; and low if it is technically possible to extend but impractical requiring extensive programming in a low-level language like C. Tools are high **interactive** if they allow the user to refocus their debugging efforts (e.g. change their queries and modifications made to running processes) mid-execution, **mid** if there are some to refocus most but not all efforts, low if the debugging efforts largely cannot be refocused without high overhead, and none if it is not possible until the program terminates. Lastly, tools are ranked high for **distributed** programming if they are specially designed to work in distributed use cases, low if they work in distributed use cases but lack the ability to correlate arbitrary states between processes without another tool, and none if they cannot be used to debug a distributed program. We rank coredumps as "?" because the ability to use them in distributed contexts depends on the tool used to analyze them.

Serial Debugging Tools There are a few approaches that attempt to use classical single processes debuggers like GDB to debug MPI programs. The simplest approach is only a single MPI rank with *GDB*. This is a major drawback as many issues often occur at scale. There are a few prior approaches used widely in the community to attempt to extend this to distributed programs. One can use programs like *screen*, *tmux*, or *xterm* to spawn multiple consoles each with its own GDB instance. But this can be unwieldy needing to switch between dozens of windows to debug at scale and control each of them individually, and does not provide an easy mechanism to debug interactions between processes.

Related to serial debuggers are *core-dumps* [1] which store the entire process state into a file to be later inspected by debuggers this limits the user to post-mortem debugging. However, coredumps may be disabled by the system administrator using the `ulimit` facility on Linux systems making it unavailable to users. Some debuggers allow attaching to multiple core files.

Parallel Debuggers There are a handful of tools that are specially designed to debug MPI programs. *TotalView* [7] and *DDT* [6] are proprietary debugging tools designed specifically for MPI. These tools are highly capable of allowing debugging mixed CPU and GPU codes, correlated stack traces between Python and C/C++, and memory access violation detection, and provide highly interactive graphical interfaces, but have limited availability due to high cost and are not easily extensible because they are closed source. A free alternative is *The Eclipse Parallel Tools Project (EPTP)* [32] (last released in 2018) which is now unmaintained and does not work with modern MPI installations that have removed MPIR support [4]. Another academic effort was *PGDB* [15] last released 2014 builds a new interface on top of GDB and introduced I/O optimizations to improve the scalability of the debugger by changing how shared libraries were loaded by the debugger, but today will not build on modern platforms, and as reported by the author to have "several known major bugs and [be] somewhat out of date" [15].

Specialized Memory Access Violation Detectors While not specifically designed for distributed programs, a common approach is to use a memory access violation on each rank of an MPI program. These will not identify the breadth of issues that a serial or parallel debugger will surface, but automate the detection of a common class of errors. However, because they lack full access to the distributed state of the program, they cannot find errors caused by complex interactions between nodes. Some tools in this category include *Valgrind-Memcheck* [25] and related tools like *Dr. Memory*[12] which provides memory access debugging and/or race detection just in time compilation to a virtual machine which can identify invalid access patterns at a substantial slow down. Valgrind-Memcheck offers some capabilities to distributed programs by validating correct usage of MPI routines, and interactive capabilities

by allowing the virtual machine to be inspected as a GDB remote at a substantial performance penalty. A related set of tools include compiler-based *Sanitizer*[28, 29] which provides memory access debugging and/or race detection supported by compiler-assisted tracking included in compilers like GCC and Clang. These tools have substantially less overhead than virtual machine-based tools but require recompilation.

Tracing and Profiling Tools Beyond tools that allow interrogating and modifying program state, there are tools that allow recording a detailed record of program execution. These tools often have low or extremely low overhead but provide limited ability to analyze system state. Examples include *printf debugging* which relies on non-standard [8], but widely implemented and occasionally unreliable I/O forwarding of `stdout` in the MPI implementation and requires recompilation and execution to study different parts of the system. Additionally, *Kernel Tracing Tools* have been considered for debugging MPI programs using kernel-level tracing facilities such as `ftrace`[24], `ebpf/bpftrace`[3, 9], or `dtrace`[20]. These facilities require administrative privileges limiting their availability where scientists lack administrative access, and typically operate at a lower in the stack allowing the debugging of for example recording both kernel and user stack traces when particular large queuing times occur with low overhead but need to be merged across nodes to provide a cohesive picture of system performance.

The *STAT* [11] is one such tool that aggregates traces across multiple nodes to provide merged stack traces from core-dumps or running processes which has limited capability to explore system state, especially on heterogeneous devices such as GPUs or display more complex system state. There are also profiling tools that are aware of distributed programs such as vampir [27] and tau [30] which can automatically gather performance timing information and aggregate them across nodes, but these tools lack the capability to interactively interrogate and modify system state.

2.2 Features in GDB for Parallel Debugging

While many are familiar with GDB's features as a debugger for serial programs, recent versions of GDB introduce a number of features that allow it to be converted and extended into a capable parallel debugger for diverse hardware and software systems. First, some key terminology. GDB refers to program execution spaces [2] being debugged as `inferiors`. Each `inferior` has a `target` connection. target connections can be `native` (direct children of the `gdb` process or a process that `gdb` attaches to if gdb has has appropriate permissions on the target process), a `corefile`, or `remote` (a child process of or attached to by a `gdbserver` process possibly on a remote node). Remote processes are interacted with by GDB over a `tcp` or an IP, Unix, or Serial socket connection.

While debugging multiple operating system threads within a process simultaneously has been supported since GDB 4, GDB 10.1 introduced the ability to debug inferior processes simultaneously – allowing distributed processes and those with different binaries. This ability is most useful with `non-stop` mode which allows GDB interacting with one operating system thread while allowing other threads to continue running as normal. Non-stop mode allows the user to debug only specified aspects of their system at a time and allows the rest of the program to continue as normal producing much less overhead.

[2]most commonly an operating system thread, but has other meanings in embedded contexts

method	capability	availability	extensible	interactive	distributed
screen/tmux/xterm	low	high	low	low	low
serial GDB	high	high	high	high	none
CoreDumps	high	low	low	none	?
TotalView/DDT	high	low	**mid**	high	high
EPTP	**mid**	low	**mid**	high	high
PGDB	**mid**	low	**mid**	high	high
Valgrind	**mid**	high	low	low	low
AddressSanitzer	**mid**	high	low	none	low
Printf	**mid**	high	high	none	low
Stat	low	high	low	none	high
Kernel Tracing	high	low	high	none	low
Vampir/Tau	low	high	low	none	high
MPIGDB	high	high	high	high	high

Table 1: Summary of Comparison of available methods for debugging MPI processes. A rating of none indicates no capability, low indicates some capability but it may be hard to use or otherwise limited, mid indicates additional capabilities but behind the state of the art, and high indicates state-of-the-art capabilities

Additionally, GDB can be extended with additional commands using Python. These commands can execute any of the existing features of GDB to manipulate, and extract information from to provide higher-level commands or perform common tasks. For example, the Python developers provide a set of facilities to get the current Python backtrace, or to print Python variables [2]. Similar functionality was developed by Google for v8 which powers NodeJS [5] and for Julia.

Lastly, GPU runtime vendors such as Nvidia, Intel, or AMD do not build debuggers from scratch – they build their own debugging tools as extensions using the Python extension capabilities on top of a largely unmodified GDB and release their versions as open source allowing them to be incorporated into 3rd party tools because of the GDB's GPL license. This means that we can incorporate these features into MPIGDB to provide these capabilities to debug GPU programs. We use these features as a foundation upon which to build MPIGDB as a capable debugger for distributed programs.

2.3 MPI Debugging Process Attachment

Debuggers like *TotalView*, *DDT*, and `MPIGDB` need a mechanism to find and attach to processes of MPI programs. This is enabled by two efforts within the MPI community: `MPIR` [22] and `PMIx` [13]. The older standard MPIR provides a consistent function which can be a consistent break-point that debuggers can use to wait for program startup after process location information has been exchanged and a mechanism to enumerate the processes and their locations from a global variable. This mechanism while not part of the formal

MPI standard, until recently was implemented in major MPI implementations [22]. As of version 5.0, OpenMPI removed support for MPIR in favor of the new standard PMIx[13] PMIx provides a more comprehensive and scalable approach to process startup and enumeration. However, not all sites have MPI implementations that are enabled or are new enough to support PMIx requiring users to either use different approaches at different sites or forgo PMIx.

In addition to MPIR and PMIx, the 4th edition of the MPI standard includes section 11.5 "Portable Process Startup" which requires implementations to provide an executable called `mpiexec` which handles at least a certain number of well-known arguments. A lesser known aspect of this standard is what the standard calls "Form A" which allows starting a collection of potentially different executables each with different arguments using the syntax like `mpiexec -n 3 ProcA : -n 2 ProcB` which starts 3 copies of process ProcA and 2 of ProcB. This syntax corresponds to the `MPI_COMM_SPAWN_MULITPLE` call introduced with MPI2. While calling `MPI_COMM_SPAWN_MULTIPLE` routine at runtime for dynamic process management is administratively disabled in running jobs on many clusters[16], MPICH and OpenMPI-based implementations (including Cray's) bless its usage in `mpiexec` and is widely implemented from our testing providing an alternative to interfacing with `MPIR` or `PMIx` directly to start the debugger processes. We use the portable mechanism of "Form A" to start up processes, we re-write the users' command to `mpiexec` to insert `MPIGDB_HELPER` which launches the MPI process under the GDB server.

3 DESIGN AND IMPLEMENTATION OF MPIGDB

`MPIGDB` is designed to be a *highly capable at scale*, *widely available*, *easily extensible*, *easily interactive* debugger for *distributed MPI programs*. In this section, we highlight how these key design principles affect the design of MPIGDB, and then provide implementation details of key aspects of our work in the following subsections.

Highly Capable at Scale `MPIGDB` is designed to enable all of the capabilities (e.g. GPU kernel debugging, cross-language stack traces) offered by the GDB debugger and enable them to be used easily at scales up to 128 processes in a distributed context. We accomplish this by building on the solid foundation of the recent features added to GDB described in Section 2.2, and allowing the user to supply a vendor-provided GDB with the ability to debug GPU kernels. We also use only the standard GDB user interface unlike [6, 7, 15] so users can adopt new features added to GDB as they are introduced to GDB or vendors extensions without waiting for them to be added to `MPIGDB`. We evaluate scalability in Section 5.

Widely Available `MPIGDB` is open source and designed to be easy to install and use across the scientific computing continuum. Because MPIGDB is implemented as two statically linked self-contained applications written in Rust, installation is as simple as copying these two programs to each node. Implementing `MPIGDB` in Rust made it easy to create a self-contained executable and provided simpler modern interfaces to process management. 92% of supercomputers on the November 2022 top500 list run architectures for which Rust is a tier 1 architecture, the remaining run a tier 2 architecture[3] suggesting it will work on most systems. Beyond HPC systems, `MPIGDB` runs well on laptops, cloud systems, and virtual machines (including the Windows Subsystem for Linux) running Linux, and can connect to embedded devices using the GDB serial port remote capability supported by various micro-controllers to support uses cases where these devices stream this information to HPC clusters. Additionally supporting Windows and MacOS with a similar architecture should be possible[4], but was out of scope for this effort as it would require writing extensions commands for `LLDB` and `windbg.exe`, but requires additional expertise that the authors do not have. Additionally, the tool only relies on capabilities in widely available versions of GDB and functions of MPI implementations specified in the standard and implemented in all common implementations of MPI. `MPIGDB` will use the underlying GDB and MPI implementations available on the system. If the system lacks a sufficiently new installation, installing the required dependencies as an unprivileged user with Spack [17] is simple and straightforward. We have successfully installed and used MPIGDB on leadership class machines at ALCF and OLCF, university systems, clouds, and laptops demonstrating its availability.

Easily Extensible `MPIGDB` is designed to be easy to extend with new commands and automate with scripts. We leverage the underlying extension architecture in GDB to provide extensions and scripting capabilities. We also introduce a series of specialized commands to make it easier to write GDB scripts that interact with distributed programs. This functionality is especially important for debugging larger groups of processes where interacting with individual processes would be both tedious and time-consuming to have a human in the loop at all times. Instead, scientists need to be able to quickly automate large aspects of their debugging process and drop to interactive debugging to explore where it provides the most value to understand complex distributed states. Therefore, we provide mechanisms to apply conditional breakpoints (apply an action on a particular line of code is reached), catch-points (apply an action when things like signals, system calls, or C++ exceptions occur), trace-points (lower overhead versions of breakpoints but can only collect more limited pre-determined information) and watch-points (apply actions when a particular region of memory or variable is modified) to subsets of processes to give scientists and attach scripts to these breakpoints to provide fine-grained control of which aspects a scientist will debug interactively and which they will automate. We describe these extension commands in Section 3.2 and evaluate their use in Section 5.

Easily Interactive the user is designed to be able to control and inspect the collection of distributed processes from a single interface. We leverage recent features in GDB to interactively provide aliases for (e.g. clients=1-10, servers=11-16) and apply commands frequently used GDB commands to these subsets of processes with convenient shortcuts. We also provide mechanisms to easily move between ranks of a program and inspect state interactively from groups of processes when a break-point, catch-point, or watch-point is reached, and then quickly continue until the next point is reached. We discuss this syntax further in Section 3.2. Interactivity

[3] indicating less compiler testing, but known to compile

[4] to support LLDB, we need non-stop mode to be implemented. At the time of publication, this feature has been worked on, but is not yet available https://www.moritz.systems/blog/implementing-non-stop-protocol-compatibility-in-lldb/

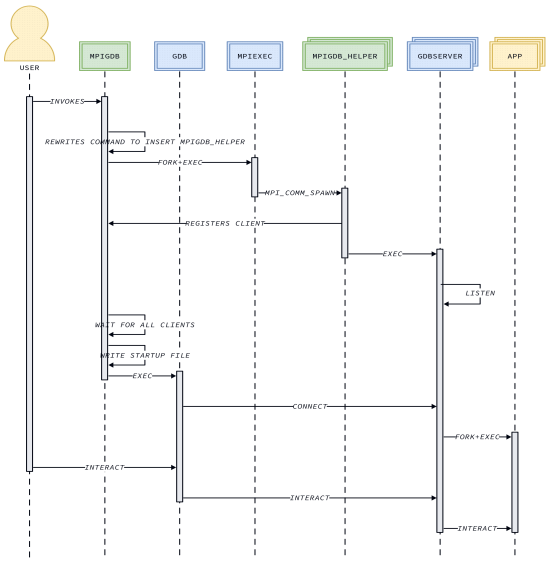

Figure 1: Startup Sequence Diagram. MPIGDB interacts with MPIEXEC to start multiple instances of MPIGDB_HELPER which in turn start gdbserver and eventually app instances. MPIGDB then spawns GDB which then connects to these gdbservers

is difficult to demonstrate on paper, but a video demo can be found with the software[5] which showcases the interactivity.

Distributed MPI programs A defining aspect of our tool is that it is designed to be used with distributed MPI programs. We make it easy to launch the debugger over a collection of processes, specify topology and other launcher-specific arguments to the underlying `mpiexec` command, and provide commands to examine and control large groups of processes at once. We describe more about the distributed process startup and debugger attachment in Section 3.1.

3.1 Software Architecture, Process Startup, and Attachment

`MPIGDB` consists of two programs: `MPIGDB` – the front-end that handles configuring the debugger, and `MPIGDB_HELPER` a shim that runs on the worker nodes to handle process registration and launch the debugger instances. In this section, we describe how these components work together with `GDB`, `mpiexec`, `gdbserver` instances, and the user's `app` instances to debug distributed MPI programs.

We present an overview of the design and key interactions in Figure 1. First, the user invokes `MPIGDB` with a combination of the MPI flags that they want to pass on to `MPIEXEC`, flags to GDB itself such as additional startup scripts, and flags for the application(s) that they want to start. This can be as simple as `MPIGDB -np 8 - ./ProcA` to launch 8 instances of `ProcA`. Additionally, with `mpiexec`, the user can use a "Form A" like syntax to spawn different program instances on different ranks. For example, the user could specify `MPIGDB -np 8 - ./ProcA : -np 4 python ProcB.py` to launch 8 instances of `ProcA` and 4 instances of the python program `ProcB.py` using `mpi4py` in the same `MPI_COMM_WORLD`.

`MPIGDB` then evaluates these parameters and re-writes them in a way to insert the invocation of `MPIGDB_HELPER`. It performs this re-writing by converting the invocation to a long "Form A" expression with different arguments for each rank invoking `MPIGDB_HELPER`. While there is a limit to the maximum length of a command line, we did not hit it even when running with 128 processes. An alternative in this case is to use so-called startup files which instead store these arguments in a file, but this would require programming `MPIGDB` to emit the distinct syntax used by each MPI implementation.

These arguments describe what TCP port the gdbserver instance should listen on and other key metadata required to launch the program and register the process with the GDB instance. The `MPIGDB_HELPER` is responsible for starting the gdbserver processes which then, in turn, starts each rank of the user's application, and informs the main MPIGDB process of how to connect to the `gdbserver` instance `MPIGDB_HELPER` spawns using a TCP socket listening in the `MPIGDB` executable.

After each `MPIGDB_HELPER` has reported how to connect to its gdbserver instance, the main `MPIGDB` process writes a startup file containing a generated GDB script that defines as extension commands and the instructions to connect to each of the gdbserver instances, and commands to configure GDB to be more suitable for MPI debugging by enabling features like non-stop mode and disabling pagination. These commands are implemented using GDB's Python API for Python command extensions. The user can add additional startup scripts to be run by GDB using the command `–MPIGDB_dbg_arg -x –MPIGDB_dbg_arg /path/to/script.gdb`.

For GPU debugging, we allow changing the `gdbserver` and `gdb` commands started by `MPIGDB` and `MPIGDB_HELPER` out for their CUDA, ROCm, or OneAPI equivalents, and this works as expected. We can also override `MPIGDB_HELPER` to support running for example with `valrind`'s `vgdb` mode which requires a different syntax for specifying which port to listen on, and this too works as expected.

3.2 Commands for Parallel Debugging

We provide a set of specialized commands for debugging a set of processes above the standard gdb commands using GDB Python command extensions. `mpic` is equivalent to GDB's `continue` command, but for all processes in the background. This command is foundational and often included in every `MPIGDB` script to start executing all processes after startup has completed and break-points are set. `mpict` is `continue` this thread, and switch to the next one that is stopped if there is one. This command allows the user to quickly cycle through processes as they each reach a breakpoint. `mpip` print on all or a subset of threads using `-t $tid`. This provides a basic mechanism to see values on a group of processes as formatted by `GDB`'s pretty printers. More advanced printers that look for summarized changes from groups of processes are easy to write using Python commands using `gdb.value()` function. `mpib` break on all or a subset of threads using `-t $tid`. This command too is fundamental in allowing the user to focus on key parts of their program for interactive debugging by stopping at key interesting points. `mpiex` execute a command on on all or a subset of threads

[5] http://github.com/robertu94/mpigdb

using `-t $tid`. This command provides a connivance method to run commands on a set of processes. It improves over `thread apply all` in that it allows targeting subsets of processes efficiently with ranges. and `mpiw` perform a command when all processes have exited. This command becomes the equivalent of the `quit` command in a serial GDB script. Using quit does not work because GDB does not provide any mechanism to wait for a background command to run to completion. Additionally, GDB does not update the exit status of inferiors while GDB or python scripts are running meaning that GDB cannot busy loop to wait for events to occur. Instead, GDB needs to register a callback with GDB's event system with a callback script in Python – `mpiw` provides a shortcut for this.

For commands that take a thread id indicated by `$tid` above, they accept specifications like `1-4` to address threads 1 to 4, `1,4` to address threads 1 and 4, or `1-4,7` to address threads 1 through 4 and 7. Additionally, connivance variables can be used to define aliases for groups of processes to provide an easy mechanism to repeat access groups of processes. This kind of filtering makes it easy to focus for example on only client processes, only server processes, or those addressing an important portion of a larger array.

4 COMPARED APPROACHES

The scope of capabilities of native debugging tools is too vast to do an exhaustive comparison in a single paper. As such, this work focuses its evaluation on debugging an out of bounds memory accesses – a common source of bugs [19, 21, 31] in applications which can yield unexpected behavior and crashes that are often not seen until at scale and can be difficult to root-cause without access to the full distributed state from the application. In this section, we focus on a few key alternatives for debugging these kinds of errors:

native release is the application when compiled with `CFLAGS="-O2 -g"`. This is a common configuration that while it provides little benefit for debugging it provides a good intuition about the native execution speed and behavior of the application forming a baseline for comparison. It is also comparable to the performance impact of tracing-like approaches like printf debugging.

valgrind-memcheck and similar tools operate by just-in-time (JIT) compiling native instructions to a specialized virtual machine which is equipped with specialized tools to detect out-of-bounds memory access patterns [25]. The virtual machine tracks the entire address space and marks regions as invalid, undefined, or initialized in its shadow map as the program executes to identify out-of-bounds or invalid accesses. In our experiments, we will run Valgrind on our baseline. Valgrind-memcheck while nearly 100× slower than native serial release execution is accurate at identifying bugs. Later valgrind was modified to have limited MPI awareness to detect misusages of the MPI API [23]. `valgrind-memcheck` also provides a mode `vgdb` that allows `gdb` to connect to the virtual machine to inspect program state over a `UNIX` or `TCP` socket. More recently `Dr. Memory` provides a faster alternative [12], but it lacks some of the more advanced features such as the `vgdb` capabilities, and would not build on our system. However, even MPI-aware valgrind has only limited capabilities and will not allow the robust kinds of interactions allowed by GDB without vgdb mode.

Address Santizer [28] – part of Google's compiler sanitizer project incorporated into LLVM and GCC compilers [10] – takes the approach of using compiler instrumentation to insert calls to update its shadow map. This approach requires substantially less overhead than Valgrind – about 2× slower than native serial release execution. When the program encounters a memory error, the program prints a concise diagnostic and terminates creating a corefile if configured [1]. Recently, Address Sanitizer has been accelerated by specialized hardware for ARM processors [10] allowing a more memory-efficient and higher-speed execution. While the vastly improved performance makes Address Sanitizer preferable over Valgrind-Memcheck, it can not catch all of the kinds of bugs that Valgrind can and requires recompilation which is not possible in all cases. However, like valgrind-memcheck, Address sanitizer doesn't allow the user to examine the global state of MPI applications.

MPIGDB-asan We also include a configuration that combines the non-hardware assisted Address Sanitizer with our tool[6]. Combining our tool with `MPIGDB` provides the global awareness of `gdb` with the efficient memory error detection capabilities of Address Sanitizer. This represents a reasonable configuration to use to debug memory access violations in production code. We also consider a baseline where **MPIGDB** is attached to this executable to measure the overhead of running under our tool. This allows us to understand the overhead of running `MPIGDB` without any additional impacts from Address Sanitizer.

We were not able to obtain access to or build another parallel debugger that would work on our platform stressing the issues of availability for these tools. We also do not consider serial debugging techniques described above as using them at scales of 8 or more processes without even considering even larger scales would be greatly impractical. The one more scale-able serial approach core dumps approach is disabled by the system administrators on our platform, and if it were not would take nearly 3 GB for a simple 21MB process over 128 processes. Programs using more realistic memory amounts

5 EVALUATION

In this section, we conduct our evaluation and compare the proposed solutions at various scales. Our evaluation is deliberately simple to facilitate comparisons, we have successfully used this tool to debug 4 more complex issues mentioned in the introduction.

We evaluate our proposed solution with a simple heat diffusion program over a square domain using a 5-point stencil written in C++ and representative of large scientific simulations such as in fluid dynamics, mechanical stress, or thermodynamics. The program uses MPI's C interface to distribute the stencil calculations across the various ranks and exchanges information between the ranks at each time step. The number of time steps is scaled to achieve a runtime of a few seconds for the native application, and the size of the domain is scaled to occupy about 1% of system memory in total per node. While this program is far shorter and uses far less memory than realistic codes to enable quick experimentation, real-world programs that use more memory and take longer will even further stress the performance of these debugging tools whose runtime scales on the amount of memory per process. For example,16GB per process would take nearly 2TB of persistent storage just to store the core files for 128 processes.

[6] the hardware-assisted version was not available on our platform

We introduce a memory access bug that reads data from out of bounds on the last time-step that occurs on all ranks caused by off-by one error in an array indexing operation. Introducing the bug at the end of execution requires the code to nearly execute to completion stressing the ability of the debugging tool not to effect the runtime of the code when bugs are not identified. While this particular bug is artificial, these kinds of errors are pervasive in distributed MPI programs [14, 18, 23].

We compare the approaches based on the wall clock time from when `mpiexec` is called to the time it takes for the system to identify the memory access violation, report it, and terminate the program. Because the bug happens at the end of the program execution it represents a particularly frustrating class of bugs that take a substantial amount of time to reproduce and may only occur at a scale where reproduction is costly in terms of core hours of an allocation or financially prohibitive on public clouds, since the cost is directly proportional to the runtime and the amount of resources.

Without MPIGDB, users would likely need to run the code once with a specialized memory access violation tool to get a line number where the invalid access occurred, and if there were multiple accesses on that line the user would then need to run the program again after re-compiling to add additional `printf` or other debugging statements to track down first which index triggered this invalid access and which rank performed that access. After that, the user would need to potentially run the program several more times to determine the cause for the particular invalid access each with additional accumulated cost for each subsequent run. With MPIGDB, users need only run their code once with both `Address Sanitizer` and `MPIGDB`, and place a break-point at the Address Sanitizer runtime function `__asan::ReportGenericError` documented in the Address Sanitizer documentation[10] to inspect and determine which index that was out of bounds and on which ranks, and then could quickly refocus their debugging efforts to what variables affect the values of those indexes and which of their mental models were violated in a single interactive execution.

We run the experiments on the Polaris platform at Argonne's Leadership Computing Facility (ALCF). Polaris nodes have AMD Zen 3 Milan CPUs with 32 cores, 512 GB of DDR4 RAM, 4 Nvidia A100 GPUs, and are currently connected with a Slingshot 10 interconnect. Polaris does not provide access to a proprietary parallel debugger like TotalView or DDT. Polaris represents a complex HPC system with scale, heterogeneous hardware with CPUs and GPUs, often runs mixed Python codes and C++ codes when running AI applications based on TensorFlow or PyTorch, and is about to adopt a novel interconnect as it migrates from Slingshot 10 to Slingshot 11 – contexts that without advanced tools scientists would have challenges debugging their codes, and would be underserved by current tools.

We run each of the 5 configurations described in the compared approaches using 32, 64, and 128 processes. Experiments use 4 nodes, with up to 32 processes per node. These scales represent several processes that are infeasible to debug using separate individual GDB instances running in separate Xterm windows while running on sufficient nodes to require the debug scaling rather than the debug queue on Polaris. Processes are distributed to nodes in a round-robin fashion to have an even number of processes per node.

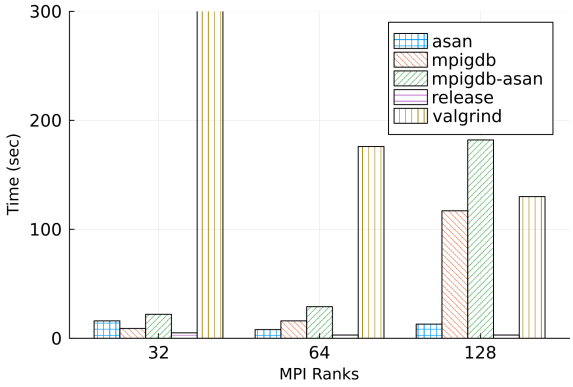

Figure 2: MPIGDB Runtime Overhead. MPIGDB has a low overhead at 32 and 64 processes. It has marginally larger overhead than valgrid-memcheck at 128 processes due to communication overheads, but provides more capabilities to inspect a distributed state.

Although `MPIGDB` allows for interactive debugging, for a fair comparison with the non-interactive tools and consistency in timing, We use a gdb script to provide the commands to start the program and exit when the processes are complete. This script contains just two commands `mpic` and `mpiw quit` – both of which are extensions to GDB added by `MPIGDB`, which instruct the program to start and then terminate when all processes exit.

Each of the approaches with memory access violation detection capabilities `valgrind`, `asan`, and `MPIGDB-asan` identify this bug and produce a backtrace to its location. `MPIGDB-asan` goes further to allow us to deeply inspect the state of the program across all ranks of the MPI program, and potentially modify this state. With this we can quickly identify the index that is out of bounds and that it was caused by the off-by-one error by inspecting the state interactively. The other approaches required re-running the program again with additional printf commands to isolate the off-by-one error.

Figure 2 shows the results of this experiment. Address Sanitizer performs well and around 2× the consistently around the release performance on 32 processes. However, unlike release, Address Sanitizer doesn't scale as well as the number of nodes increases. Profiling using Linux `perf` reveals this caused by contention on the collection and printing by mpiexec of verbose messages to `stdout` for each worker process which gets more verbose as the number of nodes increases as each rank encounters the memory bug.

Valgrind shows a scaling behavior not observed by the other approaches becoming more efficient as the scale increases. We attribute this to the fact that this problem exhibits strong scaling – that is each node solves less and less of the problem as the number of ranks increases, and thus each rank has fewer memory accesses, and thus less overhead from Valgrind per process. We would not expect this behavior from problems that exhibit weak scaling.

`MPIGDB` with and without address sanitizer performs very well up to 64 processes with performance comparable to native and the address sanitizer configuration. With up to 64 processes, is about 2-4 times slower than native – sufficiently low overhead to allow it to be useful wherever the runtime of Address Sanitizer is acceptable. `MPIGDB` begins to have substantially more overhead at close to 128

processes but is still comparable to Valgrind on 64 threads indicating it may still be useful at this scale. Profiling the MPIGDB with Linux `perf` at 128 processes indicates that the orchestration between the various `gdbserver` threads and gdb is the bottleneck. Even if MPIGDB isn't doing anything, simply monitoring the scheduling of processes by the operating system is sufficiently verbose to overwhelm the main process of processing these messages. We discuss possible fixes as future work in the conclusions to reduce the communication bottleneck.

6 CONCLUSIONS

`MPIGDB` provides a *capable*, *available*, *interactive*, and *extensible* approach for debugging *distributed MPI applications* with up to 128 processes. It provides similar debugging *capabilities at scale* as expensive proprietary tools such as debugging GPU kernels with CPU code, mixed language debugging, and more. It is designed to be highly *available* without high costs or relying on non-standard functionality in MPI or GDB. It is designed to be highly *extensible* to enable additional commands and scripts to be added for productive debugging of distributed applications. It is lastly designed to be *easily interactive* allowing users to quickly zone in on bugs in their codes that are distributed across multiple nodes using MPI.

Without tools like `MPIGDB` users need to run programs multiple times to track down complex issues because they lack visibility into the distributed state of applications running with MPI. Our tool provides a vast improvement in the ability of scientists to debug their applications in flexible science environments.

For future work, we see there is great promise in combining check-pointing libraries with parallel debugging capabilities like those in `MPIGDB`. A check-pointing library such as VeloC [26] could be combined with `MPIGDB` to more efficiently dump the state of large applications and then restore and debug only a relative portion rather than simply running the program until a crash occurs while using less storage than a simple core-dump based approach giving scientists an even more flexible approach to debugging applications by restarting from a checkpoint closer to a late occurring bug in their application.

Additionally, more work is needed at large process scales or to implement dynamic debugger attachments to already running MPI applications. There are several possible paths forward to achieve scalable performance at higher process counts: (1) consolidate the MPI ranks to talk to a smaller number of `gdbserver` instances. Doing this would likely require integrating with PMIx into MPIGDB to enumerate their locations and coordinate process startup. (2) utilize a more scalable socket polling mechanism in GDB such as `epoll` or `io-uring` for the event loop implementation instead of the current UNIX `poll` based implementation. (3) utilize collectives and broadcast techniques between the gdbservers to reduce the number of messages that gdb needs to process. (4) disable more verbose aspects of the gdb remote protocol that are less relevant in the HPC context where decisions can be made in the edge at the GDB server instead of by the `gdb` instance.

ACKNOWLEDGEMENTS

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration

REFERENCES

[1] core(5) - Linux manual page.
[2] DebuggingWithGdb - Python Wiki.
[3] eBPF - Introduction, Tutorials & Community Resources.
[4] FAQ: Debugging applications in parallel.
[5] GDB JIT Compilation Interface integration · V8.
[6] Linaro Forge.
[7] TotalView.
[8] MPI: A Message-Passing Interface Standard Version 3.1, June 2015.
[9] bpftrace, Apr. 2023. original-date: 2018-08-31T04:34:44Z.
[10] sanitizers, Apr. 2023. original-date: 2014-09-03T23:49:51Z.
[11] Arnold, D. C., Ahn, D. H., de Supinski, B. R., Lee, G. L., Miller, B. P., and Schulz, M. Stack Trace Analysis for Large Scale Debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium* (Mar. 2007), pp. 1–10. ISSN: 1530-2075.
[12] Bruening, D., and Zhao, Q. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization (CGO 2011)* (Apr. 2011), pp. 213–223.
[13] Castain, R. H., Solt, D., Hursey, J., and Bouteiller, A. PMIx: process management for exascale environments. In *Proceedings of the 24th European MPI Users' Group Meeting* (Chicago Illinois, Sept. 2017), ACM, pp. 1–10.
[14] Cotroneo, D., Pietrantuono, R., Russo, S., and Trivedi, K. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. *Journal of Systems and Software 113* (Mar. 2016), 27–43.
[15] Dryden, N. PGDB: A Debugger for MPI Applications. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment* (Atlanta GA USA, July 2014), ACM, pp. 1–7.
[16] Enterprise, H. P. XC Series Programming Environment User Guide 1705 S-2529.
[17] Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., and Futral, S. The Spack package manager: bringing order to HPC software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin Texas, Nov. 2015), ACM, pp. 1–12.
[18] Gaynor, A. Introduction to Memory Unsafety for VPs of Engineering, Aug. 2019.
[19] Gopalakrishnan, G., Hovland, P. D., Iancu, C., Krishnamoorthy, S., Laguna, I., Lethin, R. A., Sen, K., Siegel, S. F., and Solar-Lezama, A. Report of the HPC Correctness Summit, Jan 25–26, 2017, Washington, DC, May 2017. arXiv:1705.07478 [cs].
[20] Gregg, B., and Mauro, J. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, Mar. 2011. Google-Books-ID: jseJ56fUjJgC.
[21] Gunawi, H. S., Do, T., Sono, A. L. A., Hao, M., Leesatapornwongsa, T., Lukman, J. F., and Suminto, R. O. A Study of Issues in Scalable Distributed Systems.
[22] John DelSignore. The MPIR Process Acquisition Interface. Tech. rep., Mar. 2018.
[23] Keller, R., Fan, S., and Resch, M. Memory Debugging of MPI-Parallel Applications in Open MPI.
[24] Molnar, I. ftrace: function tracer, Feb. 2008.
[25] Nethercote, N., and Seward, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices 42*, 6 (June 2007).
[26] Nicolae, B., Moody, A., Gonsiorowski, E., Mohror, K., and Cappello, F. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2019), pp. 911–920. ISSN: 1530-2075.
[27] Resch, M., Keller, R., Himmler, V., Krammer, B., and Schulz, A., Eds. *Tools for High Performance Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
[28] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. AddressSanitizer: A Fast Address Sanity Checker. ACM.
[29] Serebryany, K., and Iskhodzhanov, T. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York New York USA, Dec. 2009), ACM, pp. 62–71.
[30] Shende, S. S., and Malony, A. D. The Tau Parallel Performance System. *The International Journal of High Performance Computing Applications 20*, 2 (May 2006), 287–311.
[31] The Chromium Projects. Memory safety, 2019.
[32] tibbitts@us.ibm.com, B. T. Eclipse Parallel Tools Platform (PTP) | The Eclipse Foundation.