PGDB: A Debugger for MPI Applications

Nikoli Dryden
dryden2@illinois.edu
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign

ABSTRACT

As MPI applications scale to larger machines, errors that had been hidden from testing at smaller scales begin to manifest themselves. It is therefore necessary to extend debuggers to work at these scales, in order for efficient development of correct applications to proceed. PGDB is the Parallel GDB, an open-source debugger for MPI applications that provides such a capability. It is designed from the ground up to be a robust debugging environment at scale, while presenting an interface similar to that of the typical command-line GDB debugger. Its usage on representative debugging problems is demonstrated and its scalability on the Stampede supercomputer is evaluated.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, distributed debugging*

General Terms

Design, Performance

Keywords

Debugging, Distributed debugging, Parallel debugging, MPI, XSEDE

1. INTRODUCTION

As clusters and supercomputers continue to grow in size with the deployment of petascale machines and the increasing demands of applications, we face the challenge of ensuring that these programs continue to operate correctly and efficiently at scale.

Errors can manifest themselves exclusively at large scale, such as deadlocks and integer overflows [7], as counters overflow or rare problems only appear consistently due to the high number of processes. These situations necessitate debugging at scale, which presents its own challenges, as the debugger becomes a massively parallel application in its own right. Challenges that must be solved include managing and analyzing the large volume of data produced by debuggers and scalably presenting the results in a way that does not overwhelm the user. In order to meet these challenges, there are three main tasks: reducing tool start-up costs; utilizing hierarchical structures and reduction networks; and utilizing I/O resources efficiently [10].

PGDB [3] is the Parallel GDB, a debugger for MPI applications. It is a free and open-source debugger designed to be familiar to programmers while enabling robust large-scale debugging on multiple platforms. PGDB deploys remote tool-daemons on each node the application to be debugged (the *target application*) is running on, manages debuggers for each target application process, transmits information back to the front-end via a reduction network for efficient analysis and processing, and displays it to the user in a manner designed to emulate the environment one has when debugging a single process.

This paper discusses the architecture and deployment of PGDB and provides examples of it in use debugging representative errors. It also provides an analysis of the performance and scalability of PGDB on the Stampede supercomputer. Section 2 discusses related approaches and applications for parallel debugging. In Section 3, the architecture of PGDB is described and in Section 4 the scalability and usage of PGDB is discussed. Section 5 discusses future work to be done on PGDB and Section 6 presents some final remarks.

2. RELATED WORK

Two other major parallel debuggers with reasonable scalability are TotalView [4] and Allinea DDT [1]. However, neither are free or open-source software.

The Stack Trace Analysis Tool (STAT) [7] uses a similar architecture to PGDB, and performs a similar task, but it is far more limited in capability. Instead of providing a full, interactive debugging environment, STAT only collects stack traces from running programs. This makes for a lighter-weight tool that is sufficient for some situations, but fails to provide the necessary detail and capability in others. The Eclipse Parallel Tools Platform [6] incorporates a GDB-based parallel debugger within the Eclipse IDE. However, the debugger presently supports only a limited set of clusters, and has scalability issues that PGDB does not. Further, this requires a project to be integrated with Eclipse, whereas PGDB can work with any project.

3. ARCHITECTURE

PGDB's architecture is designed from the ground up to be scalable and draws upon current and prior tools for inspiration, including STAT [7] and the Node Prism debugger [14], as well as techniques for extending traditional debuggers for parallel debugging [8]. To this end, PGDB relies upon proven tools to form a basis for its infrastructure: GDB [2] for debugging target application processes, LaunchMON [5] for MPIR interfacing and tool daemon launching, and MRNet [13] for communication and reduction capabilities. This combination has been used in other tools (e.g., STAT above), and enables effective scalability. PGDB is written in Python, which makes it easy to extend and deploy, due to Python's ubiquity.

A core principle underlying PGDB's architecture is that most of the processes that make up the target application behave in a similar manner. There are three broad situations that apply to most applications: they are correct; the same error occurs on most or all of the MPI processes (for example, every process has a segmentation fault); or an error occurs on a small subset of the processes. In the first and second cases, almost every process in the application is in approximately the same state. In the third case, due to the collective nature of many MPI applications, after one process fails, the remainder are typically blocked in a collective operation. Hence most of the processes are again in a similar state. This enables important optimizations in communication and allows for more scalable presentation of debug information.

The basic component of PGDB is a lightweight tool daemon that is launched on each remote node the target application is running on. These deploy and manage instances of GDB, which handle the details of debugging the target application. Control of each GDB instance is done via GDB's Machine Interface, an interface designed to enable GDB to be embedded within another application. These tool daemons communicate with a front-end controller via tree-based overlay network, and take advantage of reduction filters in order to provide a more compact representation of the data for both transmission efficiency and scalable presentation.

3.1 Deployment

The deployment of PGDB consists of three steps: process acquisition, deploying tool daemons, and setting up the communication infrastructure. LaunchMON handles the bulk of this in a scalable and platform-independent manner.

Process acquisition is the process of gathering information on the target application, including the hostnames of the nodes it is running on, and the process ID corresponding to each of its ranks. This is handled by LaunchMON using the MPIR process acquisition interface [11], which is an unofficial standard supported by most MPI implementations. Using this information, LaunchMON scalably deploys a tool daemon to each remote node and provides an initial communication layer. Depending upon the environment and features supported by the MPI implementation, LaunchMON uses the MPIR tool daemon launch extension or an existing MPI launcher for the tool deployment.

Once deployed on each node, the front-end uses LaunchMON to scatter network topology information to each back-end node in order to configure the MRNet network. Only the information a particular node needs is transmitted, to avoid the excessive amount of data that would need to be broadcast at large scales. This is then used to deploy MRNet, a scalable tree-based overlay network which serves as PGDB's main communications infrastructure. MRNet's topology is configurable, but in PGDB it defaults to a tree with a constant pre-configured branching factor; the internal communication nodes can be deployed to either a dedicated allocation or co-located with other tool daemons.

3.2 Communication

PGDB has several modes of communication, all of which MRNet supports efficiently: sending messages from the front-end to a subset of the tool daemons; broadcasting messages to all the tool daemons; and sending messages from a tool daemon to the front-end. Typically, messages sent from the front-end are debug commands which are passed to GDB or control commands which manipulate the tool daemons. Because of the logarithmic height of the communication tree, each message passes through only a few intermediate nodes, and no node has to be connected to every other node; hence, the communication medium remains efficient even at large scales.

Since it is common to send messages to subsets of nodes, PGDB includes an efficient range representation that allows efficient representation of and operations on contiguous sequences of integers, in order to refer to many ranks without explicitly storing them all. This avoids memory problems that applications can face when using overly-verbose data structures [10].

As discussed above, it is typical for most of the processes in the target application to be in approximately the same state. Among processes that are in approximately the same state, information produced by the debugger typically only differs in memory addresses, data values, and the like, while the structure and other information is the same. To take advantage of this, PGDB employs a deduplication scheme based upon substitution dictionaries. For a set of output records of the same "type" (which is determined by PGDB based upon the structure of each record), the structure is extracted, and each field is filled with a substitution structure. These substitution structures store one copy of common data, as opposed to one copy of each record. As records are transmitted up MRNet, reduction filters are applied at each communication node to merge these substituted records together.

Using this approach, PGDB reduces both the amount of data and number of packets transmitted. Further, this enables easier processing and presentation of data at the front-end, as data can be displayed in aggregate.

3.3 Scalable Binary Deployment

On many clusters, the nodes PGDB is running on do not have a local disk, and so system files are loaded over a parallel filesystem. In a large-scale deployment, PGDB will have a great many processes accessing the same files simultaneously as the tool daemons start up. This can cause significant contention and overwhelm the parallel filesystem, since the operations are no longer independent at large scales. PGDB includes a scalable binary deployment (SBD) system to eliminate this problem.

The SBD system first uses an interposition layer to intercept filesystem operations. It examines the arguments given to determine whether the file is on a parallel filesystem or a local filesystem (such as `/proc`), and in the latter case, transparently passes the function calls to the originals. Otherwise, the interposition layer notifies the tool daemon running on that node, which makes a request to the front-end to load the file. The front-end accesses the file on the filesystem and broadcasts the data to all the tool daemons, on the assumption that most or all of them will be requesting the same files. The data is then cached for a period at each tool daemon, in order to avoid unnecessary re-broadcasts of the data.

When used, the SBD system significantly reduces the number of filesystem operations; as the front-end performs each operation only once, in general there are only a constant number performed during PGDB's start-up, the most intensive point in terms of filesystem operations. This in turn significantly reduces the stress on the parallel filesystem, and results in faster start-up times at large scales. At small scales, the SBD system adds a moderate amount of overhead, but does not slow down the start-up process overly much; further, if the system is unnecessary, it can be disabled.

4. USAGE AND EVALUATION

In this section, we briefly discuss some issues with the installation and deployment of PGDB to different systems, including potential security implications. We then discuss some typical usage cases of PGDB, including examples, in order to provide a sense of how it works. Lastly, we briefly discuss the scalability properties of PGDB and sources of overhead.

4.1 Installation

PGDB has only a few dependencies, all of which are widely supported, which makes for relatively easy deployment to different systems. These dependencies are: LaunchMON, MRNet, Python, GDB, and standard system tools such as a compiler. Python and GDB are typically already present on most systems; and if not are easy to install. MRNet likewise has an easy installation process. LaunchMON can be troublesome to install on some systems, but PGDB has significant documentation that provides workarounds for most such problems.

Once these dependencies are present, PGDB requires only some editing of configuration files so that it can find the LaunchMON and MRNet libraries, GDB binary, and the like. No further configuration is required, although there are many options to tune PGDB to the system. Full documentation on this is available in PGDB's manual.

PGDB has been successfully deployed and tested on a variety of different systems, including XSEDE's Stampede and Trestles systems, clusters at the University of Illinois at Urbana-Champaign, and clusters and supercomputers at Lawrence Livermore National Laboratory. A Linux virtual machine image with PGDB installed is also available for small-scale testing and development.

The most notable stumbling block to installing PGDB on systems is security settings. Debuggers can be inherent security risks, and some systems include restrictions that prevent them from operating, such as restricting or disabling the `ptrace` system call. This is a necessary operation for each GDB instance to debug the target application, and so it cannot be disabled or significantly restricted for PGDB to operate.

4.2 Usage

PGDB can be started in two different modes, depending upon how one wants to debug the target application. The first is "launch" mode, in which PGDB launches the application entirely under its control. This is done with `pgdb [-launcher` *`launcher`*`] -a` *`options`*, where *`launcher`* is an optional argument specifying the MPI launcher to use, and *`options`* specifies arguments to be passed verbatim to the launcher. The second is "attach" mode, in which PGDB attaches to a target application that is already running and launched separately. This is done with `pgdb -p` *`pid`* where *`pid`* is the process ID of the MPI launcher (such as `mpirun`) used to start the target application. These can be used to debug applications running on both interactive and batch partitions of clusters.

Once PGDB has been launched, one can begin debugging the target application. The interface, both input and output, has been designed to be as similar to GDB as possible. Because of this, anyone who is familiar with GDB should have no problems adjusting to PGDB. The primary difference in output is that instead of seeing output from only a single process, the output from every process is displayed simultaneously. In order to keep this manageable and avoid overwhelming the user, output is deduplicated and aggregated, so that data with the same structure is presented only once, instead of repeatedly. This also makes it easier to identify the processes on which something is awry. Each set of output is prefixed with the associated MPI ranks from which it came, such as `[0]` for processor 0 or `[1-5,9]` for processors 1 to 5 and processor 9.

If a user wishes to examine the output from a particular processor in detail, as opposed to aggregate, PGDB provides an `expand` command, which will display the full output for a particular processor.

By default, commands entered at the PGDB prompt are sent to every processor. Often, the user wishes to send commands to only a single processor, or a subset of them. For this, there is the `proc` command, which takes an arbitrary range of processors, followed by any PGDB command. Processor ranges are specified using MPI ranks, so a user that knows how processing is divided up in the MPI application can access particular subsets. PGDB will then execute the command on only those ranks.

As a convenience to the user, PGDB provides several ways to modify and explore the data within an application. The most basic are the `print` and `x` (for *examine*) commands. The `print` command can be used to display the value of a variable within the current frame of the application, and can evaluate complex expressions which can change the state, including function calls and assignment to variables. The `x` command reads a range of bytes starting at a given address, and is useful for examining data not easily accessible through variables in the application. The companion to `x` is `write memory bytes`, which writes arbitrary bytes to memory starting at an address.

PGDB includes a filtering system, controlled via the `filter` and `unfilter` commands, which allow data matching certain criteria, such as MPI rank or message type, to be filtered and not displayed. This filtering is done on each remote tool-daemon to avoid transmission of unnecessary data.

To make it easier to examine common STL containers, such as `std::string`, `std::vector`, and `std::map`, PGDB interfaces with and automatically loads GDB's pretty-printers. These introspect and provide human-readable representations of the contents of these containers, instead of providing opaque listings of the internal structure.

Lastly, in order to facilitate exploration of structured data such as classes, PGDB builds upon GDB's variable objects, which are exposed only through the machine interface. PGDB's interface to this is especially designed to help explore very large structures, especially in situations where, as on very large projects, one may not fully understand the structures present in the program. This is used through the `varprint` command, which takes a variable name in the program as an argument. PGDB will then do a depth-limited graph search of the structure, listing member data and the associated value recursively while indicating the class hierarchy. This interface also interacts with the pretty-printing interface above. The depth limit is in place to avoid overwhelming the user with very large data structures; it also avoids listing all elements in very large containers unless explicitly required.

PGDB also differs from vanilla GDB in the way it interacts with multi-threaded programs. PGDB runs in what is known as non-stop, asynchronous mode. This allows each thread of a program to be controlled independently of every other thread. For example, one could pause one thread while the other continue executing in order to expose deadlocks. GDB's support for debugging threads also means that PGDB has "built-in" support for debugging applications that use OpenMP, and receives support for new features when they are added to GDB.

Finally, we go through a simple example of PGDB's usage, meant to illustrate its interface and typical functioning. For this example, the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) application [9] was used as a representative sample of a moderately-sized application. LULESH is a proxy application representing a typical hydrodynamics code, and is a part of co-design efforts for exascale. The build used here had debug symbols enabled and optimizations disabled, as is typical for debugging, and had both MPI and OpenMP components enabled.

The LULESH code was modified to cause one rank of the application to have a segmentation fault during the running of the application for the purposes of this example. This error was within the `CalcKinematicsForElems` function, part of the Lagrangian code.

Since the error sometimes occurs soon after start-up, LULESH was launched fully under the control of PGDB using its launch mode:

```
pgdb -a -n 1000 -f nodelist ~/lulesh/lulesh2.0
```

This launches the `lulesh2.0` binary with 1000 processes on the nodes given in the `nodelist` file. The binary is then stopped just before entering the `main` function. There is some miscellaneous start-up information displayed, followed by `[0-1000] Done.` indicating deployment of PGDB has completed. Execution of the application is then begun, in order to reproduce the bug under control of PGDB.

```
continue
[0-1000] Thread ID 1 running.
[0-1000] Done.
[1] Received signal SIGSEGV, Segmentation fault
```

`continue` tells all processes to resume execution, and PGDB reports that on each rank, thread ID 1 is now running. It then reports that rank 1 has received SIGSEGV, indicating a segmentation fault. The other processes are still running at this point. To examine them in further detail, they need to be stopped, as GDB cannot sample a program in detail while it is running. Note that there is no need in general to stop the entire application.

```
proc 0,2-1000 interrupt
[0,2-1000] Received signal 0, Signal 0
[0,2-1000] Done.
```

The `interrupt` command is prefixed with `proc 0,2-1000` so the command is sent only to those processes. A backtrace is now issued to every process, in order to discover the overall state of the application (some output is truncated).

```
backtrace
...
[0,2-1000] #3 0x00002b38b6fb34ac in
  PMPI_Waitall at src/mpi/pt2pt/waitall.c:297
[0,2-1000] #4 0x0000000000417563 in
  CommSend at lulesh-comm.cc:843
[0,2-1000] #5 0x00000000004071ee in
  CalcQForElems at lulesh.cc:1997
[0,2-1000] #6 0x0000000000408588 in
  LagrangeElements at lulesh.cc:2446
[0,2-1000] #7 0x0000000000408a19 in
  LagrangeLeapFrog at lulesh.cc:2641
[0,2-1000] #8 0x0000000000408db1 in
  main at lulesh.cc:2762
Some results from 0,2-1000 omitted; use expand to view.
[1] #0 0x0000000000401abe in
  CollectDomainNodesToElemNodes at lulesh.cc:261
[1] #1 0x000000000040fd58 in
  CalcKinematicsForElems at lulesh.cc:1559
[1] #2 0x0000000000406d25 in
  CalcKinematicsForElems at lulesh.cc:1538
[1] #3 0x0000000000406da4 in
  CalcLagrangeElements at lulesh.cc:1612
[1] #4 0x0000000000408575 in
  LagrangeElements at lulesh.cc:2443
[1] #5 0x0000000000408a19 in
  LagrangeLeapFrog at lulesh.cc:2641
[1] #6 0x0000000000408db1 in
  main at lulesh.cc:2762
```

This easily illuminates the state of the program. Process 1 is stuck in the `CollectDomainNodesToElemNodes` function where it received the segmentation fault and every other process is waiting within `MPI_Waitall` for all requests to complete. Since process 1 has crashed, the program will deadlock in this location. We now examine the local state in this frame.

```
proc 1 info locals
[1] nd1i = 0
...
[1] elemToNode = 0x0
...
```

Examining the source code, it can be seen that the segmentation fault is caused while accessing the `elemToNode` pointer, which confirms the value seen in the code. Since this is passed to the function as an argument, we examine the other frames to identify where this goes wrong. The current frame is changed, for example, with the `proc 1 frame 1` command to change to frame #1. Alternately, every command takes an optional `--frame` argument that specifies the frame it should be run in, which can be useful when issuing commands in multiple frames. The `elemToNode` variable is not defined in frame #2, however.

```
proc 1 print elemToNode --frame 2
[1] ERROR: No symbol "elemToNode" in current context
proc 1 print elemToNode --frame 1
[1] 0x0
```

Using this information, we can localize the point at which `elemToNode` became invalid to a small range of code, which can then be examined directly. If this were insufficient, PGDB could instead be used to set watchpoints on every processor as soon as the `CalcKinematicsForElems` function was entered, in order to trace exactly where the variable was modified.

PGDB's presentation of debug output makes identifying many kinds of errors easier. Suppose instead that LULESH were experiencing a threading deadlock instead of a segmentation fault. As soon as PGDB was attached to the application and backtraces gathered, one could have a good idea where in the code the source of the error was.

4.3 Scalability

PGDB's scalability was evaluated on the Stampede supercomputer, located in the Texas Advanced Computing Center. Stampede consists of 6400 compute nodes with sixteen cores and 32GB of memory each, plus at least one Intel Xeon Phi coprocessor. The nodes use an InfiniBand interconnect in a 2-level fat-tree topology. Additionally, each node has a 250GB local disk and mounts three global Lustre filesystems. The local disk is used to store system files and provide a local scratch space. For this evaluation, the Xeon Phi coprocessors were not used.

The test application for this evaluation was the LULESH application used in the prior example, without any errors introduced, as the goal here is purely to measure PGDB's scalability. For all these tests, PGDB is launched in attach mode, to avoid the overhead of launching the target application task in the performance assessment. LULESH requires that the number of processes used be a cube of an integer; this is why the number of processes used may seem unusual.

The first factor we consider is the start-up time for PGDB. Debuggers that take too long to initialize impose a significant usability burden, and the start-up process is the most intensive portion of PGDB's operation due to the need to deploy tool daemons and load debug information about the target application. To measure this, the PGDB front-end records the time just before the LaunchMON engine is invoked to trace the MPI launcher process. It then watches the responses of the tool daemons until every one has reported itself ready to the front-end. The difference between these two times is taken as the start-up time. Tests were performed on jobs running on between 1 and 256 nodes on Stampede (between 8 and 4096 processes). To help eliminate noise due to changing machine conditions, each test was run five times and the results averaged together.

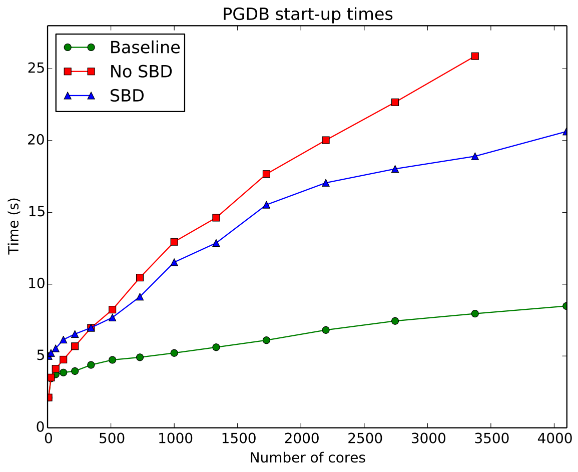

Figure 1: Start-up times for PGDB: baseline, loading files over the parallel filesystem, and loading files over the SBD system.

At each scale, three different situations were evaluated. In the first case, PGDB was launched with the SBD system disabled and all files were loaded off the local drives on each node. This illustrates the ideal baseline case for PGDB, where every file operation is truly independent. In the second and third cases, the files PGDB attempts to access were relocated to the scratch Lustre filesystem. This situation is similar to what is found on clusters where nodes lack local storage. For the second case, the SBD system was disabled; for the third case it was enabled. These data are plotted in Figure 1.

In the ideal baseline case, PGDB scales very well, taking only two seconds for start-up at the smallest scale and about five seconds at the largest scale. At small scales, loading over the parallel filesystem was also efficient, and took very little additional time. However, as the target application scales, this approach becomes less efficient. In contrast, when using the SBD system, there is initially overhead due to the additional communication this entails. Beginning at about 512 processes, the SBD system as faster than loading files over the parallel filesystem. At the very largest scale, loading files over the parallel filesystem did not complete within five minutes, and so the test runs were terminated.

The primary cause of this difference is the number of filesystem operations. In the latter two cases, the number of filesystem operations that actually reach the filesystem were also recorded with an interposition system like the one used in the SBD for up to 1000 processes. This counted the number of `open` and `fopen` function calls directed to real filesystems (as opposed to, for example, `/proc`). As every file successfully opened has some additional operations performed on it, this count provides a good proxy measure of the total number of operations performed. This data is in the table in Table 1. Without the SBD system, every filesystem operation reaches the filesystem, and so the number of operations scales linearly with the number of processes in the target application. With the SBD system, only a constant number of operations are performed, regardless of scale.

The scalability of PGDB's communication was also evalu-

Cores:	8	27	64	125	216	343	512	729	1000
No SBD:	264	891	2112	4125	7128	11319	16896	24057	33000
SBD:	33	33	33	33	33	33	33	33	33

Table 1: Number of operations performed on the filesystem by PGDB without and with the SBD system.

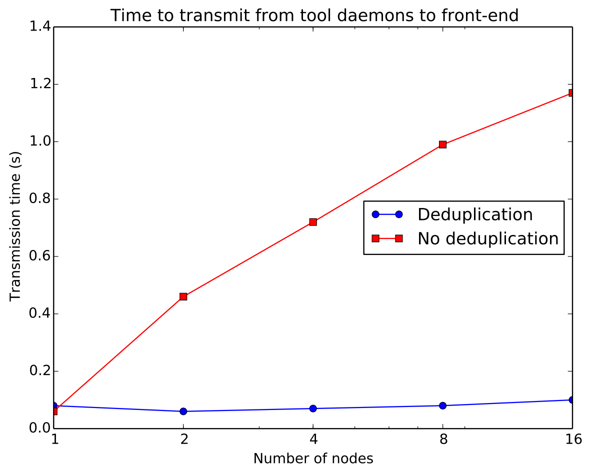

Figure 2: Time to transmit data from tool daemons to front-end with and without deduplication. (Nodes had 16 cores each.)

ated. While MRNet provides efficient communication, transmitting too much data can overwhelm the network. The deduplication system helps to avoid this case. The average time taken for a message to be transmitted from the tool-daemons to the front-end over the course of a debugging session was recorded with and without deduplication. These results are presented in Figure 2 for between 1 and 16 nodes.

As expected, without deduplication, it takes longer to transmit data at every scale but one node. The time to transmit the data scales linearly with the number of processes, due to the amount of data being produced increasing linearly. In contrast, with deduplication enabled, the transmission time grows very slowly. This is due primarily to less data being transmitted, but the fewer number of packets being sent over the network also contributes.

4.4 Overhead

We consider two aspects of overhead related to PGDB: additional memory usage and communication overhead. Note that these are a separate issue from the application being slowed by running under the control of the debugger, which is an aspect of overhead that is more the province of the underlying GDB debuggers than PGDB.

PGDB deploys one instance of the GDB debugger to each node the target application is running on, and GDB then loads the debug symbols for each process. When processes share debug symbols (for example, they have the same shared libraries or are running the same binary), only one copy of the symbols are loaded. This minimizes memory overhead as much as possible while keeping the debug symbols distributed for scalability. For some applications that have extremely large numbers of debug symbols, PGDB can be instructed to not load symbols for shared libraries unless explicitly requested, in order to further minimize overhead.

The majority of PGDB's heavy communication occurs during start-up, but this typically does not influence the application as it is paused very early in the process. When the user is not running commands and there are no events such as breakpoints being triggered, PGDB does very little communication and typically does not affect application communication patterns. When data is transferred, the deduplication support within PGDB reduces the impact, as the previous section discussed.

5. FUTURE WORK

There are three primary areas for improvement in PGDB: its scalability, its system support, and its interface. At very large scales, where the SBD system is important, loading files exclusively from the front-end node will become a bottleneck due to the height of the tree increasing the time to transfer data, especially with large files. Instead, PGDB should load files from a subset of the MRNet communication nodes, which then broadcast the files to their descendant tool daemons. This will result in smaller, independent broadcasts that are more manageable while also improving file loading times. As the number of communication nodes is quite small relative to the total number of processes in the target application, this should not overwhelm the parallel filesystem. Further work also needs to be done to investigate PGDB's performance at extremely large scales.

While PGDB has been tested and is known to work on many platforms, it has not yet been tested on Cray and IBM Blue Gene systems. Support for these should not be overly difficult, as both systems support PGDB's dependencies, but architectural and environmental differences will pose a challenge.

It is also important for PGDB to support different programming models such as Intel Xeon Phis and GPUs. Support for debugging Xeon Phis should not, in principle, be difficult to add. Intel maintains a fork of GDB as its primary debugger, and includes support for Xeon Phis in it [15], and this can likely be used without many changes to PGDB. Debugging support for GPUs is significantly harder, and GDB does not have any support for it. NVIDIA provides CUDA-GDB [12], which supports CUDA applications and may be a way forward for integrating GPU debugging support for PGDB, however the feasibility of this approach has not yet been investigated.

Lastly, PGDB attempts to emulate GDB's interface, which provides a familiar environment for some, but there are many people who are unfamiliar with GDB. A GUI interface has been proposed for PGDB, which would allow it to be more easily accessible for a wider userbase. This could also enable more intuitive presentations of the data that PGDB can collect on the target application's state.

6. CONCLUSIONS

As clusters and supercomputing platforms continue to scale, we must extend debuggers to support these scales. Unlike at small scales, in these environments debuggers become massively-parallel applications in their own right, and must be treated as such when designed and implemented.

PGDB provides a free and open-source debugging option that is capable of efficiently scaling to large problems, in terms of both its underlying architecture and its user interface. It builds upon LaunchMON and MRNet, both proven technologies for supporting tools at very large scales. Its debugging capabilities and interface leverage the familiarity and power of GDB, in order to provide a host of features. These also enable PGDB to be deployed to a wide range of system environments with relatively little difficulty.

Debuggers are an indispensable component of software development regardless of the scale one is working at. PGDB provides a quality option for this at the scales applications on clusters and supercomputers require.

7. ACKNOWLEDGMENTS

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. The author would also like to thank Aaron Black and Mike Owen of Lawrence Livermore National Laboratory, Dan LaPine and Alex Zahdeh of the National Center for Supercomputing Applications, and Marc Snir of the University of Illinois and Argonne National Laboratory for their support and advice throughout the development of PGDB.

8. REFERENCES

[1] DDT. `http://www.allinea.com/products/ddt/`. Accessed: 2014-03-10.

[2] GDB. `https://sourceware.org/gdb/`. Accessed: 2014-02-20.

[3] PGDB. `https://github.com/ndryden/PGDB`. Accessed: 2014-02-20.

[4] TotalView. `http://www.roguewave.com/products/totalview.aspx`. Accessed: 2014-03-10.

[5] D. H. Ahn, D. C. Arnold, B. Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Overcoming scalability challenges for tool daemon launching. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 578–585. IEEE, 2008.

[6] J. Alameda, W. Spear, J. L. Overbey, K. Huck, G. R. Watson, and B. Tibbitts. The eclipse parallel tools platform: toward an integrated development environment for xsede resources. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*, page 48. ACM, 2012.

[7] D. C. Arnold, D. H. Ahn, B. R. De Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[8] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden. Extending a traditional debugger to debug massively parallel applications. *Journal of Parallel and Distributed Computing*, 64(5):617–628, 2004.

[9] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[10] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. De Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons learned at 208k: towards debugging millions of cores. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–9. IEEE, 2008.

[11] MPI Forum Working Group on Tools. The MPIR process acquisition interface version 1.0.

[12] NVIDIA. CUDA-GDB. `https://developer.nvidia.com/cuda-gdb`. Accessed: 2014-05-27.

[13] P. C. Roth, D. C. Arnold, and B. P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 21. ACM, 2003.

[14] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, et al. A scalable debugger for massively parallel message-passing programs. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 2(2):50–56, 1994.

[15] Software and Services Group, Intel Corporation. Intel Xeon Phi Product Family: The GNU Project Debugger. `https://software.intel.com/sites/default/files/article/256677/intel-mic-gdb.pdf`. Accessed: 2014-05-27.