

A Scalable Framework for Offline Parallel Debugging

Karl Lindekugel, Anthony DiGirolamo, and Dan Stanzione Ph. D.
{klindeku, adigiro, dstanzi}@asu.edu

Fulton High Performance Computing, Arizona State University

Abstract. Detection and analysis of faults in parallel applications is a difficult and tedious process. Existing tools attempt to solve this problem by extending traditional debuggers to inspect parallel applications. This technique is limited since it must connect to each computing processes and will not scale to next generation systems running on hundreds of thousands of processors. Additionally, existing techniques for monitoring programs and collecting runtime information can scale but are unable to provide enough interaction to find complex software faults. This paper describes a novel parallel application debugger that combines parallel application debugging and a programmable interface with runtime event gathering and automated offline analysis. This debugger is shown to diagnose several common parallel application faults through offline event analysis.

1 Introduction

High Performance Computing systems continue to grow in size and complexity. Applications for these systems are growing in complexity to match. The recent advent of multi- and many- core chips has only accelerated this trend, with large scale applications now requiring tens to hundreds of thousands of threads to achieve maximum performance. Debugging technology has remained fairly constant in recent years, with most effort still focused on interactive debugging schemes. While commercial debuggers now exist that will execute at reasonably large scales, it is not clear that large scale interactive debugging is compatible with the way large systems are operated, or that the information presented in this way is useful to the developer at very large scales. This paper describes a novel approach to parallel application debugging that moves the debugging process from interactive to offline. A framework has been constructed that allows debugging activities to be automated and information about program state to be collected in a relational database for later analysis.

The scale and scope of future high performance computing systems present several challenges to the development of monitoring and debugging tools. First, the sheer volume of data generated by debugging and monitoring tools requires an efficient infrastructure for collecting and organizing this data. This infrastructure must support the collection and storage of a wide variety of system and application data and make this data available to users and tools in a standard manner. Second, this flood of information must be presented to the user in a readable and useful way. Visual representation of data is untenable for applications and systems executing at very large scales. Future debugging and performance tools must be able to automate the discovery of key information to enable researchers and developers to act on the collected metrics in a timely manner.

An additional challenge is integration with the software stack on production systems. Executing applications at a very large scale presents unique challenges, and many applications need to be debugged at scale on the target systems. Modern leadership computer systems may cost millions of dollars per month to operate, and many sites are

loathe to make a substantial fraction of the system available for debugging purposes in an interactive way. Most monitoring systems, especially most current debugging systems, require an interactive session between the debugging user and the application being analyzed.

Traditional application debuggers execute alongside the program with user interaction in real time. This provides a simple framework for the developer to explore the execution of the application, but it also requires presenting execution information to the developer that can be understood in a meaningful manner. Conversely, the monitoring of application performance is traditionally done by collecting information during the program execution for analysis afterwards. This information is then processed by performance tools to be presented to the user in an appropriate manner. Scaling interactive debuggers to large numbers of execution threads presents problems not only in managing thousands of concurrent threads but also in the presentation of debugging data in a timely and understandable manner for user interaction. Altering the model of parallel debugging to one similar to performance monitoring presents a solution to the challenges of processing information from hundreds of thousands of threads. Collecting debugging information during execution and developing tools to perform analysis of this data after execution will allow the developer to effectively work with debugging information at large scales.

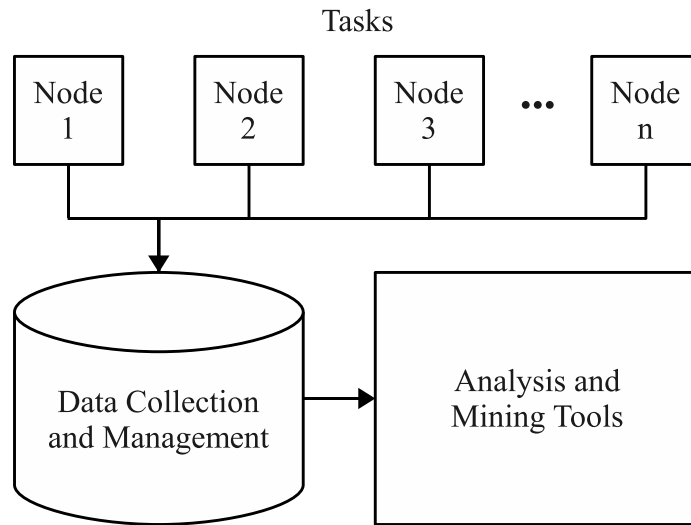


Fig. 1. The three main components of the GDBase system are shown above. Event gathering instances run alongside MPI tasks on each node. During execution, each instance collects information which is stored locally. Each instance then reports its information to a central database. This stored information can be queried by the user through analysis tools. See figure 2 for more information on the individual tasks. Figure 4 shows additional information the data collection and management component. Figure 5 shows the analysis side of GDBase.

In response to these challenges a framework for analysis of large scale computing systems has been developed. The architecture shown in Figure 1 collects information from running tasks on an individual basis and then aggregates the information from each run in a manner that allows developers to analyze and data mine the collected information for relevant metrics. Such a framework would allow debugging parallel

applications running in modern high performance computer sites using threads that range in the hundreds of thousands. This paper examines the development of the GDB Database (GDBase) parallel debugger built with this framework.

2 Related Work

2.1 Traditional Debuggers

Serial applications are typically debugged using an interactive debugger designed for fault detection such as the GNU Debugger[1]. These debuggers allow developers to investigate the state of a running application, modify this state, or monitor the application for changes in state such as execution of functions or modification of variables. This functionality allows developers to track down faults as well as investigate the state of a program during a crash. Fault detection can also be accomplished using event trace analysis or using time travel debuggers such as BDB[2] which allow the execution to be rolled back to previous points of execution and continued with a modified state. Time travel style debuggers are limited by the expense of storing execution state regularly and difficulties in freezing or rolling back communication state.

The development of efficient debuggers built upon trace analysis also permits users to move forward and backwards in viewing and analyzing application state. The Scalable Omniscient Debugger[3] uses a parallel event collection database to quickly store and query application event data collected from a running application. This data can be queried and visualized using the debugger front end to find faults and examine application behavior. This framework also allows the parallel search of collected data.

Applications can also be examined using a debugger to check for correctness. These techniques, such as assertions[4], model verification[5], trace analysis[6], or statistical sampling[7] evaluate the behavior or state of the program against a set of expected behaviors or values. Assertions are typically compiled in to an application during debugging runs and check program state against a set of rules, reporting an error if a rule is violated. Model verification and trace analysis compare the executing behavior and results of an application against a second, ideal model of behavior. Statistical sampling builds upon the ideal model idea but only compares state and behavior intermittently using a random sampling technique. This reduces the overhead required for data collection with out a significant change in efficiency.

2.2 Parallel Debuggers

Parallel application debugging requires combining performance analysis, fault detection, and correctness checking in a distributed, asynchronous, and concurrent environment that presents new challenges[8]. Parallel applications present new debugging challenges in addition to those of serial applications such as deadlocks, race conditions, and precedence errors[9]. Deadlocks occur when distributed applications have some dependence of behavior that can be placed into a state where the application cannot continue. Race conditions occur when the proper behavior of an application is affected when one task performs an action either faster or slower than expected. Precedence errors occur when the order of operations performed impacts the expected behavior of an application. These parallel errors can either cause incorrect behavior or the total failure of the application if the unexpected behavior is not handled by the program. The challenges of parallel application debugging are made more difficult by the distributed and usually uncoordinated environments in which they must run.

Parallel correctness checking has been developed by extending the techniques used on serial applications to parallel domains. Model verification can still be used against parallel applications but the size of parallel application problems could make this method prohibitively expensive. Research has found a system to demonstrate correctness and identify the source of some faults[10] through a distributed event analysis mechanism. Other work using message logging is able to identify inefficient messaging patterns in distributed applications[11].

Several popular debugging packages have extended standalone debuggers to provide interactive parallel application debugging across multiple machines. This technique is used by several parallel debuggers including TotalView[12], Distributed Debugger (DDB)[13], and PDT[14]. The application to be debugged is loaded by a central debugger on to the parallel machine and controlled by the user. The user can then issue debugging commands to individual processes or groups of processes and see the results using a tree of aggregates and broadcasters to reach each node[15][16]. A few systems aim to find application faults using only limited aspects of execution, such as message passing[17]. Research has also explored the parallel examination of performance event data collected from parallel programs[18].

Debugging systems designed to analyze programs for correctness are limited in their ability to detect faults. Since the events are collected without user interaction there is no straightforward method to investigate improper behavior or faults. Interactive debuggers are limited to detecting faults but require network connections to each node in a running application and must be able to control these processes without introducing any improper behavior themselves. The requirement to hold network connections to each node and provide an interface for control limits the scalability of these interactive debuggers. Interactive debuggers also make it difficult to debug applications on clusters where access to nodes is granted through a batch queuing system where a job could execute at any time.

3 Design

3.1 Overview

GDBase provides an architecture capable of scaling to very large numbers of processors, provides the functionality of GDB, and works in cooperation with batch queuing systems. GDBase gathers runtime information from a programmable GDB instance, collects this information to a distributed event database, and provides a mechanism for analysis of this data. This is accomplished through an architecture that combines an execution engine capable of processing debugging commands (Figure 2), event logging (Figure 4), and an analysis system to process the collected events (Figure 5). Each of these components represents a separate phase of debugging. During program execution debugging commands are processed and logged to a local event collector. This process is shown in Figure 2. After execution is complete events from debugging tasks are collected into a central location as shown in Figure 4. Once all events from a particular debugging task are collected analysis agents may operate on them, see Figure 5, to determine the cause of a crash, trace variables, and more.

3.2 Controlling The GNU Debugger

An application to be debugged is loaded via the GNU Debugger[1], a freely available and proven tool for applications on a wide variety of platforms. This feature rich debugger is controlled via its *machine interface* allowing for straightforward processing

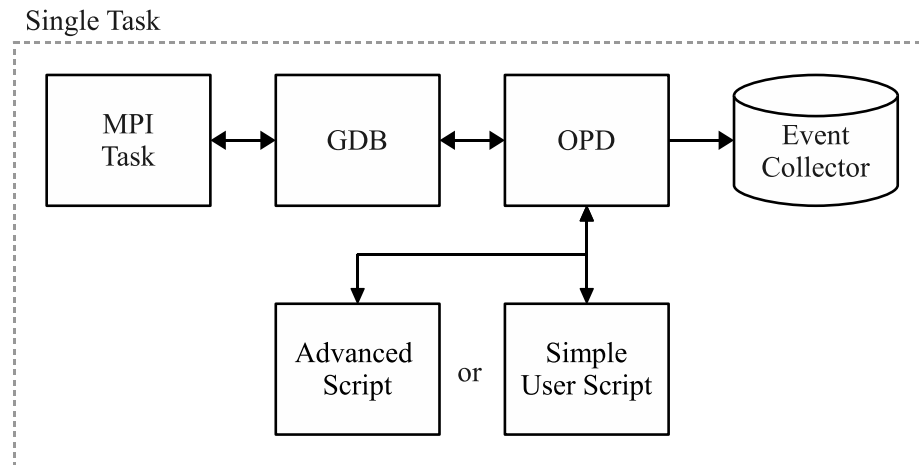


Fig. 2. A diagram of a single GDBase task running on a node. GDBase launches the MPI application under GDB control and logs events to local disk. GDBase’s behavior may be controlled via the simple debugging specification. Optionally, additional control may be defined using an advanced script interface.

```

@bp myfunction
    a
    b
    x
@bp myfile.c:11
    myarray[45]
    myarray[11] % 32
@watch myfunction q

```

Fig. 3. An example debugging specification file for controlling GDB during application execution is shown above. The script is interpreted by GDBase’s programmable interface. Comments begin with the # symbol. The first command @bp sets a breakpoint upon entering the function myfunction. When that breakpoint is reached it logs the stack trace along with the value of a, b, and x. The second @bp command sets a breakpoint on line 11 of the myfile.c file. When reached, the element at the 45th index of myarray and the value of the expression myarray[11] % 32 is recorded. The last command @watch sets a watchpoint on the variable q when the program enters the myfunction method. The new value of q is then logged whenever a write to that variable occurs.

and parsing of commands and results. This interface allows the application developer to use the full power of the GNU Debugger across an entire parallel application at once. The GNU Debugger can be controlled via the programmable interface to generate application events, investigate the program state, or even modify the state of the executing program.

The programmable interface presents an easy to use scripting language to debug the application without the need for an interactive session or user interface. The programmable interface is loaded by each node at the beginning of the debugging job and responds to events generated by the application and the debugger. Once activated by the debugger this interface can be used to explore the state of the application, modify variables or memory of the application, or generate custom events for later analysis. If

a debugging script is not available the default behavior of the application is to monitor for memory faults and program crashes and create event outputs. This can then be analyzed to find the faults in the application. An example of a debugging script is shown in Figure 11.

An alternative to developing a script to debug the application is to write a debugging specification file for the application. This specification format is loaded by the parallel debugger and allows the user to specify variables to watch or breakpoints to set. At each breakpoint a number of expressions can be evaluated and the results, along with a stack trace, are logged. This is the format shown in Figure 3. This allows users to collect common debugging information without learning to write a debugging script.

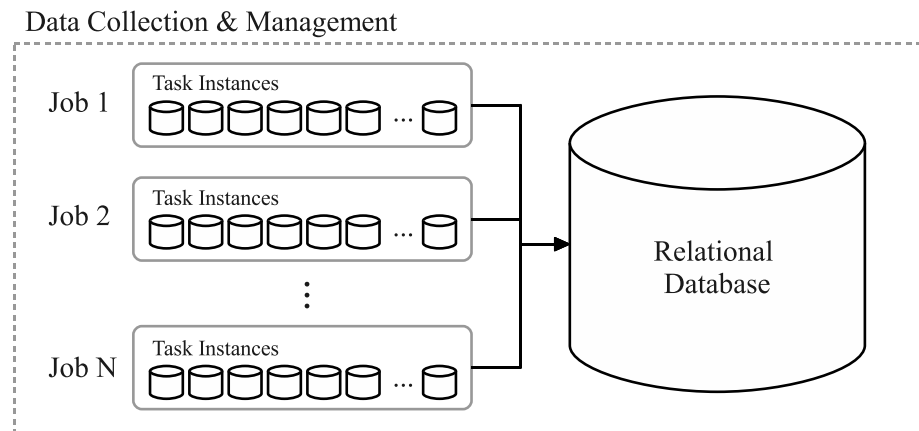


Fig. 4. Shown above are multiple jobs gathering events from each of their tasks. This process begins after an application has finished executing, the events are collected and stored in a master database. Since multiple jobs can be stored, analysis tools can compare data between runs.

3.3 Event Collection

One challenge of debugging large systems is managing the large volume of event information generated by each instance. Events generated by the application being debugged by GDBase are collected in a relational database specific to the individual process. This database can either be stored on local storage, local memory, or on parallel scratch space. Using a relational database allows the analysis system to easily query large amounts of collected data as well as provide a standard interface for additional tools to interact with the data. Using a separate database for each MPI task, either on shared storage or on local storage, minimizes the amount of contention between tasks in storing event data. The individual databases are combined at the end of the program run for later analysis.

3.4 Offline Analysis

Another challenge of large scale debugging or performance monitoring is how to analyze the resulting information or present it for the user in a manageable way. GDBase uses offline analysis tools or agents to reduce the large number of generated events from

Analysis & Mining Tools

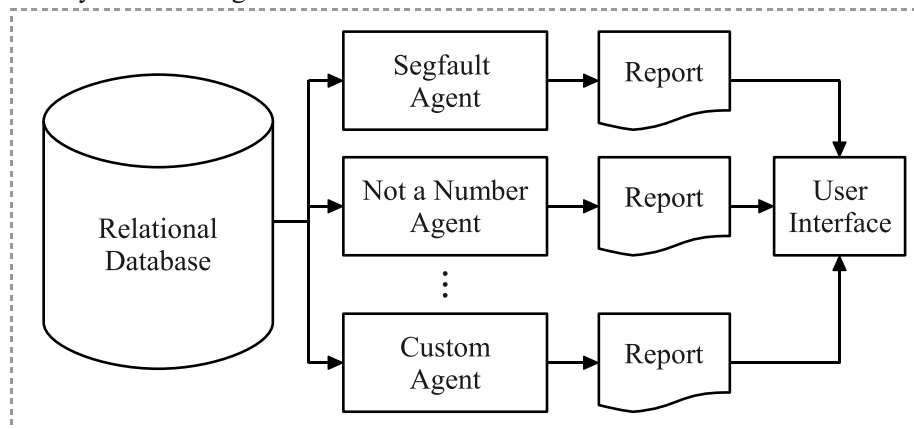


Fig. 5. Events from debugging runs can be analyzed after a run is complete and the events stored in the master database. Each analysis tool, or agent, is modularly designed to search for a specific type of error and generate a report. Reports can then be gathered into a user interface. Such an interface has yet to be developed.

each processes in to a relevant few reports. Once the debugging session for the application is completed the resulting event logs are collected to a central database for analysis and evaluation. Each analysis agent evaluates the events for specific error types and notifies the user if one has occurred. Currently two analysis agents have been developed based on user feedback. The first checks the run for a segmentation fault and reports back the location in code and state of the application at the crash. The second checks watched variables for a transition to an invalid numeric value, Not a Number. Not a Number errors, or NaN errors, can occur as the result of mathematics operations that cause numbers to become too large, small, or return an imaginary value such as $\sqrt{-1}$. The agent then reports the location in code of this change, the rank of the processes on which it occurred and several of the previous variable values prior to the change. The development of additional agents is made easier since the event data can be accessed using the common *SQL* interface.

4 Implementation

GDBase uses GDB's machine interface to maintain control over the target application. Currently, it supports applications running in MPI environment or in serial, GDBase itself does not use MPI. GDBase's only requirement is that applications the user wishes to debug be compiled with `-g` to enable debugging. Naturally, running applications under GDB control can incur a noticeable performance hit. Each MPI task has a GDB instance running along side it as seen in Figure 2. In the current version, *all* debugging messages are stored in a relational database located on a parallel scratch space as the application runs. Although storing the database on parallel scratch creates network overhead this is mitigated by having a separate database for each task. This prevents the database and file system from having to coordinate with other tasks reading and writing the associated files. This architecture allows the system to quickly gather large numbers of events in a scalable manner.

The user has two options in gathering data from a running application. Debugging information such as breakpoints and watched variables can be logged using the specification interface shown in Figure 3 or debugging can be controlled directly through the TCL scripting language (see Figure 11). This allows expert developers and debuggers to control and inspect the state of the application with a great deal of freedom and flexibility. These developers would then need to expand the analysis system to examine and report using the data they have collected. The average developer can inspect the state of the application and use existing analysis tools by using the simple interface.

A framework for the analysis of event logs has been developed using the Python scripting language to address the challenges of understanding the large volume of debugging events that can be generated from a single run. The ease of text manipulation and database interaction allows such analysis agents to be developed quickly using Python. Two analysis modules have been developed at the request of our users, one for segmentation fault detection and one for variable watches. A segmentation fault in a parallel application on our system produces a simple error message and kills the job, providing little information for the user to determine the location of the failure. Using the default GDBase reporting the segfault analysis tool will report back the node number of any segfault, the location in the source code that the fault occurred in, and the current execution stack. A second tool examines watched variable information and reports back the code location where a floating point variable transitions to NaN.

Additionally, GDBase has been engineered to work properly in a batch queuing environment allowing use of the debugger from within non-interactive job scripts. Specifically, the current implementation works properly with the Torque batch system and with both OpenMPI and MPICH message passing implementations as well as the MPIEXEC launcher. Launching GDBase is done through a wrapper that sets up GDBase, executes the debugger in parallel using MPIEXEC, cleans up after execution, and produces an end of run report.

5 Results

5.1 Overhead

Debugging using GDBase can be significantly easier at larger scales than comparable debugging manually. To compare the relative effort of using GDBase to debug a simple debugging session was done using both GDB for each process and GDBase. A GDB session for each instance was chosen since it is an available option on many clusters as part of the MPICH execution environment. Each debugging session attempted to inspect the state of a calculation at ten iterations by using a break, check, continue pattern of debugging. Results, shown in Figure 6, demonstrate the high personal cost of this form of interactive debugging.

<i>Execution Type</i>	<i>Time (S)</i>	<i>Percent Extra</i>
No GDBase	39.00s	-
GDBase, No script	40.04	2.67%
GDBase, Break on Iteration	40.65	4.23%
GDBase, Break on MPI Send/Recv	45.50	16.66%

Table 1. Impact of GDBase on Application Run Time

The overhead of executing an application using GDBase is shown in Table 1. Shown is the effect of a few debugging scenarios against a locally developed Finite Difference Time Domain(FDTD) code across several different numbers of tasks. FDTD is a popular algorithm for electrodynamics simulation. This implementation is parallelized through three dimensional domain decomposition with ghost planes to reduce communication. The impact on execution time from the GDBase system is heavily influenced by the number of breakpoints and cost of breakpoints through the GDB interface.

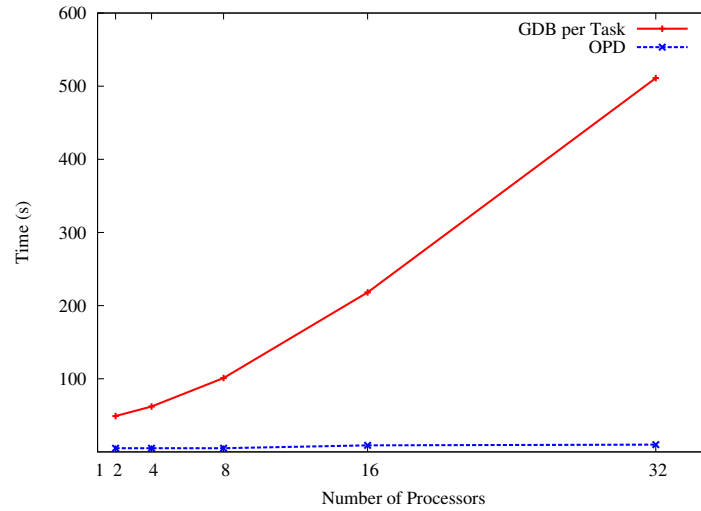


Fig. 6. Comparison of time to debug between per-task GDB Debugging and GDBase

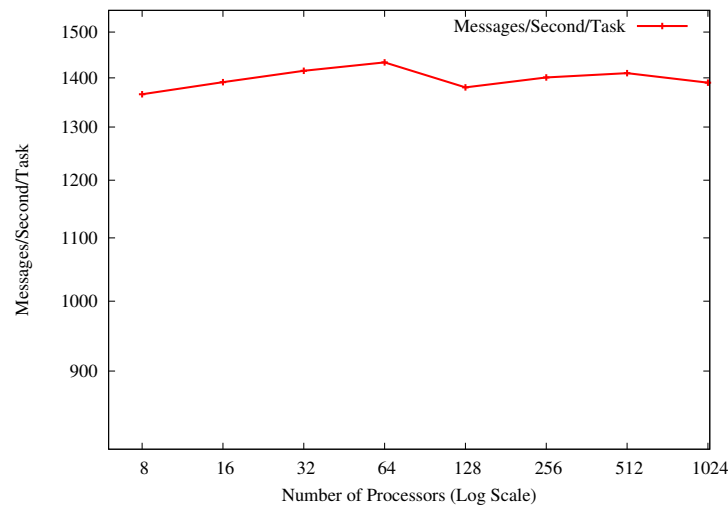


Fig. 7. Message Storage Speed Per Task (Local Disk)

Figure 7 illustrates the scalability of the message storage system on both local storage and a shared high speed scratch space. Since each debugging task interacts solely with its own database there is no impact from locking or synchronization between tasks in storing events. A scalable system for storing events is integral to the overall scalability of GDBase.

5.2 Usage

One example of a traditional software fault, a memory segmentation error, can be debugged with ease on a serial application. This common software failure can be difficult for users to discover on a parallel system since some MPI implementations will return communication errors instead of notice of an application segfault when a task crashes. Finding the location of a crash is the simplest form of fault detection and is the default behavior of the Offline Parallel Debugger. The GDBase debugger will monitor the application for a memory segmentation fault, log the location and stack trace, then produce this data afterwards using an analysis agent.

<pre># mpiexec ftdt mpiexec: Warning: task 37 died with signal 11 (Segmentation fault). mpiexec: Warning: tasks 1-63 died with signal 15 (Terminated).</pre>	<pre>GDBase End of execution report OPD End of execution report PBS JOBID: 147738.moab.local DatabaseID: 110 Statistics: start: 2008-03-13 21:36:33 end: 2008-03-13 21:58:53 elapsed: 00:22:20 ncpus: 64 Messages: 1346 Detector Results: SEGFAULT Job crashed on rank: 37 At: main in ftdt/ftdt.c:366 With stack: 0 main in ftdt/ftdt.c:366 With locals: MPI_Comm comm = 139 int * p = (int *) 0x4 int i = 132 int t = 4245505 int n = 4 int sg_interval = 1 FILE * fp = (FILE *) 0x0 size_t start1 = 2640000 size_t endl = 0 float simtime = 0</pre>
--	--

Fig. 8. A side by side example of an MPI application running with and without GDBase under MVAPICH. The left hand side shows the error output provided by MVAPICH and the right hand side shows GDBase's output which includes the location of the crash

As an example we will compare an MPI application crashing with and without GDBase. First, a crash is observed in the parallel application during normal execution

(without using GDBase). See the left hand side of Figure 8 for error output. Next, we launch the application under OPD control by invoking `opdexec mpi_target`. This requires that the application be compiled with debugging flags. Once the run is completed, data from each task is collected and analyzed to find the location of the crash. The right hand side of Figure 8 shows example output of task zero experiencing a segmentation fault.

Another common error is when an application, during some iterative calculation, produces a NaN value. Many computations are not perturbed by this value and will simply carry it through to the end of the execution. Finding where this value was initially set, and the values taken by the variable in previous iterations, can help the developer find the calculation error leading to the erroneous results. This analysis was initially requested by a user when a change in compilation lead to NaNs in the output. By running the application several times and tracing the affected data points we can point to the line of code causing the error. An example of this usage follows in Figure 9.

```
NaN found on rank 0 point fdttd.c:240 expression pEx[4][4][4]
File : Line          New          -          Old
fdttd.c : 373        0.00000131   -        0.00000262
fdttd.c : 371        0.0000006   -        0.00000131
fdttd.c : 373        0.00000024   -        0.0000006
fdttd.c : 431        0.00000012   -        0.00000024
fdttd.c : 432        -0.00000012  -        0.00000012
fdttd.c : 433        -nan(0x400000) -        -0.00000012
```

Fig. 9. An example GDBase Detection of Watched Variable Becoming Not A Number (NaN)

Finally, the flexibility provided by GDBase through its programmable TCL interface allows expert developers to inspect or interact with their application with the full power of GDB on any task. For example, a debugging script (Figure 11) could be used to check an intermediate array of data for plausible values during execution without modification of the original program. The arbitrary events generated by custom developed debugging scripts can be analyzed and extracted from the event database using *SQL*. An example is shown in Figure 10 of just such a script with a simple *Python* based tool for reporting.

```
res = db.query("SELECT * from messages where job_id=%d
              and key LIKE '%user.overflow%' order by id asc"
              % (jobid) ).dictresult()
print "Rank Location"
for r in res:
    print "%s:\t%s" % (r['rank'], r['value'])
```

Fig. 10. Python Script to Report Overflow Values collected by the user TCL script

First a TCL script is written to break during the beginning of each iteration of a computational code and inspect the values in an array. TCL loops are used to inspect

each value, generating an event if the value is outside of the specified range. See Figure 11.

```

proc user_setup {} {
  gdb_setBreakpoint "fddd.c:366" "user_segflt"
  set output [gdb_lastOutput]
  db_logMessage "user.break" $output
}

proc user_segflt {} {
  set ilim [ gdb_evalExpr {nperp2+2} ]
  set jlim [ gdb_evalExpr {nperp+2} ]
  set klim [ gdb_evalExpr {nperp3+2} ]
  for { set i 0 } { $i < $ilim } { incr i } {
    for {set j 0} { $j < $jlim } {incr j} {
      for {set k 0} { $k < $klim } {incr k} {
        set output [gdb_evalExpr "pEx\[$j\]\[$i\]\[$k\]" ]
        if { $output > 1.0 } {
          db_logMessage "user.overflow" "$j, $i, $k: $output"
        }
      }
    }
  }
  gdb_continue
}

```

Fig. 11. TCL Script to examine all elements of a 3 dimensional array for incorrect values

Now, the job is executed using *opdexec* and data is collected. GDBase automatically detects the user TCL script which is shown in the post execution report (Figure 12).

```

GDBase End of execution report
PBS JOBID: 146409.moab.local
DatabaseID: 82
Statistics:
  start:  2008-03-08 23:07:24
  end:    2008-03-08 23:23:01
  elapsed:      00:15:37
  ncpus:      1000
  Messages:    31101
  Executed:    fddd.par.tcl
Detector Results:
Rank  Location
0      3, 7, 4: 1.234455
256    9, 32, 16: 9.999999999E499

```

Fig. 12. Report from end of execution with TCL debugging script and cluster user report generated

Finally, the events generated by the execution can be examined to see if an event was raised by the user level code. This snippet (Figure 10) of Python code reports if any such events had been raised. Using the SQL interface to do the comparison utilizes the database's indexing and searching for quick result turn around. The resulting report is shown in Figure 12.

6 Conclusions and Future Work

The development of larger and larger parallel machines will require new developments in debugging systems. GDBase demonstrates a novel approach to tackle the challenges of extracting debugging information at large scales and presenting that information to the user in a manageable way. The debugging system is demonstrated to collect debugging information via GDB and scale to over a thousand processors. The preliminary analysis agents demonstrate that even large volumes of information can be quickly queried and relevant events automatically extracted without the user exploring the data manually.

Continued development of the GDBase will focus on enhancing the debugging facilities provided through the programmable interface and the development of complex analytics for the collected data. Such analytics will explore the automatic detection of communication deadlocks and race conditions.

A few issues which will be explored in the development of GDBase exist. The design of GDBase permits enormous amounts of debugging information to be collected. There is however a trade off in the amount of data for debugging information collected and the impact on overall execution time by the monitoring system. There must be balance between the amount of debugging information collected versus the amount of detail needed when searching for program errors. The type of messages collected and the amount of aggregation must be finely tuned to address all levels of program faults.

In addition to the debugging analysis tools GDBase's could be extended to support other tools besides GDB in order to log different types of events. For instance, using a profiling tool to log performance related events such as floating point utilization, memory accesses, or network usage. Such additional tools could supplement the debugging effort or be used separately with the logging and analysis infrastructure to examine other aspects of program behavior on large systems.

The collected event data presents a number of opportunities for analysis and exploration. Additional tools to examine this data could detect common communication mistakes or find bottlenecks in execution. Additionally, the collection of data from several executions could allow analysis tools to compare behavior between runs. Application developers could analyze this data to find regressions between changes or find edge cases in end user use. Site operators could compare performance and debugging information from large numbers of jobs to create detailed usage reports or detect the impact of system changes on application behavior.

References

- [1] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA, 2006.
- [2] B. Boothe, "Efficient algorithms for bidirectional debugging," in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, (New York, NY, USA), pp. 299–310, ACM, 2000.

- [3] B. Lewis and M. Ducasse, "Using events to debug java programs backwards in time," in *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (New York, NY, USA), pp. 96–97, ACM, 2003.
- [4] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski, "Algorithmic debugging with assertions," in *Workshop on Meta-Programming in Logic*, pp. 501–521, 1988.
- [5] G. Jost and R. Hood, "Relative debugging of automatically parallelized programs," *Automated Software Engg.*, vol. 10, no. 1, pp. 75–101, 2003.
- [6] A. Quick, "Integrating functional modeling and trace-driven parallel debugging," in *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, (San Diego, California), pp. 223–225, 1993.
- [7] L. Jiang and Z. Su, "Context-aware statistical debugging: from bug predictors to faulty control flow paths," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, (New York, NY, USA), pp. 184–193, ACM, 2007.
- [8] J. M. Francioni and C. M. Pancake, "A debugging standard for high-performance computing," *Sci. Program.*, vol. 8, no. 2, pp. 95–108, 2000.
- [9] J. Huselius, "Debugging Parallel Systems: A State of the Art Report," Tech. Rep. 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.
- [10] D. Kranzlmüller, *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, GUP Linz, 2000.
- [11] F. Wolf, B. Mohr, J. Dongarra, and S. Moore, "Automatic search for patterns of inefficient behavior in parallel applications," *Concurrency Practice and Experience*, 2005 accepted.
- [12] TotalView Technologies, *TotalView Debugger Users Guide: version 8.3.0*, 2007.
- [13] T. Sienkiewicz, J.; Radhakrishnan, "Ddb: a distributed debugger based on replay," *Algorithms and Architectures for Parallel Processing, 1996. ICAPP '96. 1996 IEEE Second International Conference on*, pp. 487–494, 11-13 Jun 1996.
- [14] C. Clmenon, J. Fritscher, and R. Rhl, "Visualization, execution control and replay of massively parallel programs within annai's debugging tool."
- [15] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden, "Extending a traditional debugger to debug massively parallel applications," *J. Parallel Distrib. Comput.*, vol. 64, no. 5, pp. 617–628, 2004.
- [16] C.-P. Chen, "The parallel debugging architecture in the intel debugger," in *PaCT* (V. E. Malyskin, ed.), vol. 2763 of *Lecture Notes in Computer Science*, pp. 444–451, Springer, 2003.
- [17] B. Krammer, M. S. Miller, and M. M. Resch, "Mpi application development using the analysis tool marmot," 2003.
- [18] M. Geimer, F. Wolf, B. Wylie, and B. Mohr, "Scalable parallel trace-based performance analysis," in *Proceedings of the 13th European Parallel Virtual Machine and Message Passing Interface Conference (EuroPVM/MPI, Bonn, Germany)*, pp. 303–312, Springer, 2006.