Effective Offline Parallel Debugging: A View from the Trenches*

Carlos Rosales
carlos@tacc.utexas.edu

Doug James
djames@tacc.utexas.edu

Robert McLay
mclay@tacc.utexas.edu

Bill Barth
bbarth@tacc.utexas.edu

Dan Stanzione
dan@tacc.utexas.edu

Texas Advanced Computing Center
Research Office Complex 1.101
J.J. Pickle Research Campus, Building 196
10100 Burnet Road (R8700)
Austin, TX 78758-4497

August 23, 2012

Abstract

We report on our experiences with GDBase, an innovative offline parallel debugging tool developed in 2008 as a proof-of-concept demonstration. We describe work using GDBase to debug a large-scale experiment involving a mature, stable, production-quality scientific application. As a result of this experience, we offer thoughts that may be of value to users and designers of parallel debugging technology.

1 Introduction

In 2008 Anthony Digirilamo and Karl Lindekugel, under the direction of the last author [5] [9], produced and released to the research community a product they called GDBase [14], an innovative proof-of-concept demonstration of an offline parallel debugging tool. Digirilamo and Lindekugel were at the front end of a trend that is starting to go mainstream: both Allinea's DDT [1] and Rogue Wave's TotalView [11], for example, now include integrated offline parallel debugging modes.

These products may be among the first production-quality tools supporting offline debugging, but they won't be the last. Certainly there will always be a mission critical role for high end, traditional interactive debuggers. Such tools are starting to scale well; the best of them provide the functionality and visualization tools needed to support interactive debugging sessions involving very large numbers of cores. The challenges of large-scale debugging, however, go beyond the functionality of the tools themselves [7]. Leadership class systems typically operate in a batch mode intended to maximize utilization and throughput. It is generally unrealistic to expect to schedule a large block of time to operate interactively across a substantial fraction of such a system. Even when large scale interactive sessions are possible, they can be inconvenient and expensive, and can impact system access for others. Thus non-traditional approaches to debugging, including offline modes, are likely to become more important as our systems continue to grow.

Debugging will always be difficult; debugging parallel programs at large scale is particularly so. Given the emergence of offline modes and other non-traditional approaches to large-scale debugging, there is value research and discussion regarding the design of non-traditional tools and effective use of non-traditional techniques. In that spirit, we report here our experiences employing GDBase in an effort to track down a serious bug in a large-scale experiment involving a mature and stable production-quality scientific application. Our hope to offer something of value to both designers and users of debugging technology.

Sections 2 and 3 establish the context with a brief overview of offline debugging, the GDBase project, and the defect we set out to correct. Section 4 is a chronology of the debugging effort, including commentary on our experience with the offline tools and techniques. In Section 5 offer thoughts that are largely independent of specific vendors and products: lessons learned about the process of offline debugging; thoughts on the functionality and design of tools supporting offline debugging; and open questions worthy of debate, discussion, and research. We close with concluding remarks in Section 6.

2 Offline Parallel Debugging: Concept and Implementation

In our model the term *offline debugging* describes the run-time behavior of the debugger: during execution the user-analyst does not interact with the target

application and low level debugging tools. Instead, the offline debugger replaces the human user-analyst and manages a scripted debugging session that runs in batch mode. The debugger captures data and stores the results in a database that is available to the analyst after the batch job is complete. The analyst can then query the database to filter, synthesize, and analyze the results. As is typical with any debugging task, the analyst uses the results to define subsequent debugging exercises, iterating as required. The design of GDBase reflects this model. Lindkugel engineered the initial release [9]; Digirolamo [5] later added additional functionality.

The GDBase software system has four major sub-systems:

- *Event Configuration.* These are the tools the analyst uses to define debugging activities. For straightforward events such as breakpoints and watches, the user edits a simple configuration file that GDBase reads at runtime. For more complicated events the user can write a custom script (*collection agent*) to control the session; the system provides an Application Programming Interface (API) that makes writing such scripts a straightforward task. Lindkugel and Digirolamo authored and exercised several sample demonstration scripts; see the *Event Analysis* bullet below.

- *Event Collection.* This is the software that replaces the human user and controls the debugging session during batch execution, interacting with the low level debugging engine as specified by the analyst before the run begins. GDBase writes a separate results file for each monitored process.

- *Event Management.* The GDBase infrastructure includes a tool the analyst uses to merge the individual process database files into a single aggregate database.

- *Event Analysis.* Before or after database aggregration, the analyst can use standard SQL tools to query the database as desired. In addition, the analyst can write scripts (*analysis agents*) to automate analysis. Lindkugel and Digirolamo include in the GDBase package several such sample scripts, including scripts that report the location of segmentation faults; locate deadlocked processes; and identify the sources of errors in calls to the Message Passing Interface (MPI) Allgatherv collective [9].

Certainly the sole focus of GDBase is offline debugging. In that sense, it stands in contrast to traditional, interactive debuggers, and by design invites stakeholders to compare and contrast two kinds of products and processes. In general, however, we do not see 'offline debuggers' as a separate category of tools. Instead we assume that support for offline operation is an integrated capability – a mode, if you will – that will likely soon be a part of any serious debugging tool. This is precisely what we are seeing in emerging production products [1], [11].

3 The Target Application: MGF

Lindekugel and Digirolamo [5] first used GDBase as a proof-of-concept demonstration, exercising the tool against a suite of test problems. Running a variety of applications on 1024 cores, they validated the system by locating the source of three types of segmentation faults, and mined a 100G database (two billion messages from 12,000 processes) to identify the cause of an aborted call to MPI_AllGatherv. They also demonstrated successfully a novel deadlock agent that correlates sent and received messages and then reports the probable location of deadlocked processes.

The current authors attempted the logical next step: using GDBase to help debug a major production-quality scientific code and large-scale experiment under conditions in which traditional techniques had come up short. As a warm-up problem they used GDBase to help identify a problem that prevented the cosmology code GADGET [12] from running on the TACC Lonestar cluster. They then turned their attention to the target problem that is the foundation of this paper.

The target application was a large finite element fluids code known as MGF ("Micro-Gravity Flow"), originally developed by Barth, McLay and Carey [2], [3], [4]. MGF is a mature, stable, flexible product, with capabilities that are broader than its name suggests. Current MGF research, focused on preconditioning techniques, provides opportunities to exercise the code vigorously at small to medium scale.

At large scale, however, MGF typically hung, with indications that suggested classic deadlock. While large scale operation is not yet central to the MGF team's current activities and short term research goals, they were concerned enough to contact the MVAPICH[1] [10] developers to ask for their advice and assistance. The MVAPICH group advised the MGF team by email to modify their code so it did not call the MPI_Irsend function [13], [8]. The email exchange led the MGF developers to conclude that bugs in the MPI stack were the likely source of the problem they were observing. In any case, the issue remained unresolved as MGF work focused on other priorities.

4 Chronology

The MGF issue described in Section 3 seemed the ideal case for exercising GDBase in a realistic, real world setting. The issue affected a mature, stable, non-trivial scientific application. It eluded debugging and analysis efforts by both the MGF and MVAPICH teams (and in fact there was some evidence the bug might be in the MPI stack rather than MGF itself). Moreover the problem occurred only at fairly large scale.

One additional factor made this an especially valuable exercise: the composition of the five members of the project team (the authors of this paper).

[1] MVAPICH provides the primary MPI library used to build and run MGF on TACC-based clusters.

Two participants (Barth and McLay) represented the MGF research team; they focused on matters related to the target application. The other three (Rosales, James, and Stanzione) represented the GDBase research group; their primary interest was studying the offline debugging tool and process. The collaboration proved to be a rich and rewarding one: as is often the case, the dynamic among colleagues with varying backgrounds and interests led to insights, progress, and results that would not have been possible otherwise.

Phase 0 – Preparation

The experiment of interest was an MGF test problem requiring 3072 cores that appeared to deadlock during initialization.[2] The MGF developers made a copy of this experiment available to the larger GDBase team. We were unable to reproduce the deadlock on a small test problem. We did, however, exercise several smaller test problems for this study. These smaller problems, which terminated normally and produced correct results, were scaled versions of the original, larger deadlock problem.

We began cautiously. Before beginning any serious debugging we ran a variety of MGF experiments with and without GDBase. We used the small test problems to learn how to run GDBase and MGF together, and worked out a number of kinks along the way. We also learned a great deal about GDBase and especially GDB/MI [6], the API that allows GDBase (or any other program) to call functions in the underlying debugging engine. These lessons would prove useful later.

During this familiarization phase we also spent time trying to estimate and control the size of the files that GDBase would generate during serious debugging runs against the 3072 core target problem. In the slides accompanying his thesis defense, Lindekugel [9] described a debugging effort on a 1024 core test problem that generated 100GB of GDBase output. Simple extrapolation raised some concerns for us: we feared that 3072 separate GDBase processes writing 300G of output could strain the file system and impact other users. So we scaled slowly to full size, watching carefully the sizes of the files we generated. We also considered ways to modify GDBase to reduce the output it generated, but ultimately decided that effort would take us too far afield. In retrospect we were overly cautious: our debug runs never generated more than 100MB of data (one thousand times less than Lindekugel reported), and our runs did not strain the file system.

While we were were not yet looking for bugs at this early stage, we did in fact detect and correct a problem unrelated to the deadlock. We found this particular bug more or less by accident, and felt a bit lucky when we encountered it. But this was only the first instance of an important idea that became a mantra

[2] The MGF team first encountered the deadlock while running on TACC's Lonestar cluster. We therefore decided to use Lonestar exclusively during this debugging project. In retrospect this was unncessary: we would have observed the problem on any cluster. It was probably also unwise: in general it is undoubtedly helpful to know early whether a bug's symptoms depend on the hardware and software environment.

for us: shining a light on your source code often leads to good things. We reminded ourselves that debugging tools don't fix code or even detect defects. Instead, human programmer-analysts use debugging tools the way other repair professionals use flashlights: we are all trying to shine lights in places and in ways that make it possible to find and fix problems.

Phase 1 – Locating the Defect

We set as our first debugging goal the task of determining where in the code the processes were deadlocking. MGF itself includes an extensive suite of user-configurable debug options, so even without GDBase we were able to narrow the search to a small region of a specific subroutine. We know that all processes hung in this region, but that some processes proceeded further than others.

Given the symptoms of the problem (classical deadlock), our knowledge of the code (asynchronous communications), and the history of the analysis efforts to date (in particular the MVAPICH developers' recommendation to stay away from calls to MPI_Irsend [8] and [13]), we focused on calls MPI_Irsend, MPI_Isend, MPI_Receive. Using as templates the collection and analysis agents that LindeKugel and Digirilamo included with the early releases of GDBase [5], we wrote customized collection and analysis agents (our first attempt at writing scripts for GDBase) designed to probe the behavior of MGF on calls to these functions. Largely as an exercise we initially designed these agents to execute a break at all such calls. At each such break GDBase would write a stack trace as well as collect the values of local variables and arguments to the associated MPI function.

The results confused us until we realized that we were inadvertently trapping calls to MPI functions that the MPI stack was itself making internally. The stack traces certainly told us this, but this was easy to miss when inspecting output. It seemed worthwhile to avoid collecting such internal MPI calls, but we felt there was no need to write code to parse the stack traces. Instead we decided to write our own calling functions that acted as wrappers around MGF's key MPI calls. Our thinking was that we could then write GDBase scripts that executed breaks on the names of the wrappers rather than the names of the MPI functions.

We should have known better; this proved to be a very bad idea. Writing wrappers proved more difficult than we'd anticipated (we spent several days fighting scoping and visibility issues) and involved much more code modification than any reasonable debugging effort should require. It was also entirely unnecessary. We eventually realized what should have been obvious from the beginning: the simplest approach involved breaking on line numbers rather than function names. Once again, however, luck and accident played a played a role in our detective work, and it was precisely our ill-advised attempt to put wrappers around MPI calls that ultimately exposed the bug.

The MGF build system puts a very high priority on flexibility and repeatability: virtually anything the user might want to control is available as an option at build time. One can choose to build, for example, a serial or parallel version of

Figure 1: Asynchronous Messages – Intended Behavior (send rank = 23)

MGF (the serial version makes no MPI calls), or a version that uses MPI_Isend instead of MPI_Irsend. To achieve this, the MGF source code depends heavily on macros. To write our MPI wrappers we needed to spend quite a bit of time understanding the macros, and as we worked our way through layers of macros we eventually encountered a macro-defined constant with the value 2048. When we saw that number the root cause of the deadlock jumped off the page. We sensed almost instantly that the constant 2048 represented an upper limit on the number of MPI processes and that something in the code was breaking down when there were 2048 or more tasks.[3] It didn't take long to find the place in the code that depended on the constant.

During a key portion of MGF's mesh initialization, each task sends two distinct asynchronous messages to each of its neighbors: one message that plays a role in synchronizing communication (a 'thin', or zero-byte synchronization message) and another that contains data required for the calculation itself (a 'fat' message). The program uses the MPI tag field to communicate both the

[3]In fairness to the developers, we note that the portion of the code containing this macro dates back to 1997, when a machine with 2048 processors was simply unthinkable.

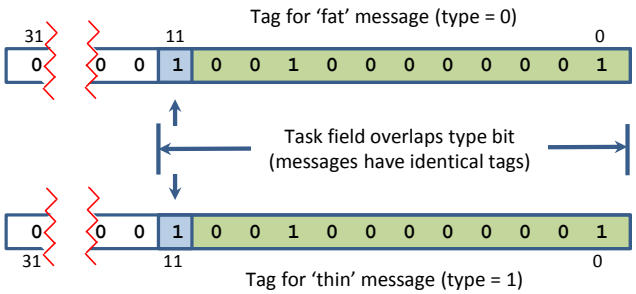

Figure 2: Asynchronous Messages – Actual Behavior (send rank = 2625)

sender and the message type. The value in the 11th bit position (equivalent to the decimal integer 2048) distinguishes between message types: a 'thin' message has the 11th bit set to one, while a fat message has the 11th bit set to zero (Figure 1). When the sender's rank is 2048 or higher, however, it takes at least 11 bits to store the rank, so the bit field used to store the rank overlaps the bit position used for message type. The result is (at least the potential for) a large number of messages with identical tags (Figure 2). More precisely, every process with rank at or above 2048 would (at least in theory) send to each of its neighbors two distinct messages with identical tags. A receiving task would be unable to distinguish between those messages, wreaking havoc on the program's execution.

At this point we were not yet interested in sorting out precisely how the program was breaking down. We first wanted to determine whether or not we'd found the root cause of the problem. That was easy to do: we simply changed the value of the constant to a higher value and ran the experiment to see if the problem went away. It did; MGF ran to completion, terminated normally, and produced correct results. But we finished this phase of the project mildly disappointed. We had found the bug and learned a great deal about GDBase, but we also knew that the tool had not yet played a major role in our success. There was more work to do, however, and we were about to be surprised.

Phase 2 – Understanding the Deadlock

We now knew the root cause of the deadlock, and we could certainly implement a plausible fix. But the MGF team wanted a more complete picture of the deadlock mechanism to help them address broader questions about their asynchronous communication strategies. Here's where things started getting interesting.

Our working assumption was straightforward: we expected that every process with a rank of 2048 or greater was sending to each of its neighbors two messages with identical tags. So we wrote agents (scripts) for GDBase that collected all relevant messages at their source, looking for 'message twins'. Writing the agents was a valuable exercise, but the results startled us. We were wrong: the agents found no message twins.

The GDBase results spoke to us sternly and unambiguously: we knew we had to revisit the deadlock and do a more careful analysis. We went back to the precise sequence of events associated with this particular sequence of messages[4] (Figure 3), and reminded ourselves that each task receives and processes all expected 'thin' messages before it sends its own 'fat' messages. This led us to a new, more nuanced theory. We conjectured that tasks were misinterpreting received messages: when they received a 'thin' synchronization message from a high ranked neighboring task they were interpreting it as a 'fat' message. Re-

[4]The MGF developers used this approach to optimize performance on a small 1997-vintage cluster. The zero-byte synchronization guaranteed that the receiver posted its receive buffer before the sender transmitted its 'fat' message. While the zero-byte synchronization is no longer necessary on modern clusters, its effect on performance is negligible.

```
// Prepare to receive 'fat' msgs; report that I've posted buffer...
for each neighbor
  MPI_Irecv( receiveBuffer for this neighbor's fat message )
  MPI_Isend( synchMessage for this neighbor )
end for

// Receive zero-byte synch msgs from neighbors...
for each neighbor MPI_Recv( synchMessage from this neighbor )

// Send 'fat' messages to neighbors...
for each neighbor MPI_Isend( fatMessage for this neighbor )

// Wait for neighbors to receive my fat messages...
for each neighbor MPI_Wait( fatMessage for this neighbor)

// Wait for neighbors to receive my zero-byte synch msgs...
for each neighbor MPI_Wait( synchMessage for this neighbor)

// Wait to receive 'fat' messages from my neighbors...
for each neighbor MPI_WAIT( fatMessage from this neighbor )
```

Figure 3: MGF Initialization Messages (Pseudo-Code)

ceivers never correctly recognized the expected 'thin' synchronization messages, so they were hanging in a wait state and never transmitting their own 'fat' messages. Eventually the deadlock would propagate even to low ranked tasks, but (we conjectured) low ranked tasks would proceed further than high ranked tasks.

Under this theory there would be no message twins, and validation would be a little trickier. Instead of looking for duplicates on the sender side, we would have to focus on the receiver side and detect discrepancies between the actual and expected nature of received messages. The tag would not help here; instead we'd have to focus on the messages' contents.

We produced and debugged collection and analysis agents to test this new conjecture, then turned them loose on MGF. This time our conjecture proved correct[5] (Figure 4). Other evidence also supported the conclusions suggested by the GDBase output. Low ranked tasks, for example, did in fact deadlock later than high ranked tasks.

We were now confident that we understood the precise mechanism behind the deadlock, and GDBase had been mission critical in helping us achieve that understanding. The tool had refuted our initial conjecture and confirmed our subsequent analysis. GDBase had proven especially valuable during the difficult phases of the debugging effort. We finished our work convinced that this was no coincidence.

[5]Frankly we were a bit surprised that GDBase saw the contents of received messages at the breakpoints to the MPI_Irecv call. We thought the message contents might be unavailable to the receiver until the receiver reached the associated MPI_Wait.

```
*** Mismatched arguments found in IRECV / ISEND completion.

   Instance 1:
       Receiving rank was 2047 and sending rank was 2048
       Expected size was 7682 but received size was 0

Receiving rank stacktrace:
Stack: Level    Function                File    : Line
       5       main    in      mgflo.C:556
       4       CmdFactory::execute     in      factory.C:42
       3       CmdFactoryImp<MeshCmd>::my_execute      in      CmdFactory.h:42
       2       MeshCmd::execute        in      mesh.C:118
       1       InternalMesh::create    in      mesh.C:210
       0       NodalSendLists::NodalSendLists  in      sendlists.C:112

Sending rank stacktrace:
Stack: Level    Function                File    : Line
       5       main    in      mgflo.C:556
       4       CmdFactory::execute     in      factory.C:42
       3       CmdFactoryImp<MeshCmd>::my_execute      in      CmdFactory.h:42
       2       MeshCmd::execute        in      mesh.C:118
       1       InternalMesh::create    in      mesh.C:210
       0       NodalSendLists::NodalSendLists  in      sendlists.C:117
```

Figure 4: GDBase Output Reporting the Source of the MGF Deadlock (Excerpt)

5 Lessons Learned and Recommendations

Our focus throughout this project has been the process of offline parallel debugging. Our primary goal has been to learn something of value to the larger community, independent of any particular choice of tools or processes. Reasonable people can differ in the conclusions they draw from experiences like the case study we've described; there are differing perspectives even among the papers' authors. Here we offer key thoughts that reflect the consensus experience of the authors, knowing that each of these observations represents a point of view rather than provable objective fact.

Workflow and Process

Regarding workflow and the debugging process, four observations stand out for us:

- *Interactivity.* When an interactive session is a practical option, such a session is most likely a more productive option than offline debugging. When interactivity is not practical, however, offline debugging can be a powerful, effective alternative, and it's important to remember that interactive sessions are often impractical for reasons unrelated to the the functionality of the debugger. Even though many interactive debuggers now scale

reasonably well to large numbers of processes, other issues remain serious constraints. These include the difficulty of scheduling large interactive sessions; the lengthy run times associated with large complex jobs; and the challenges of information overload associated with a large-scale interactive session.

- *Scripting.* We found that GDBase's most valuable feature is its support for scripting. We wrote customized collection agents (scripts that control the runtime behavior of GDBase) to probe for answers to specific questions and to limit collection activity to specific locations and types of data. We also wrote analysis agents (post-processing scripts) to look for subtle relationships and report appropriate findings. Thanks to the power of scripting, we found that GDBase became more valuable to us as our needs, goals, and questions became more sophisticated. There is a compelling lesson here: debugging at scale is most effective when the tools provide the capability to synthesize and analyze in ways that go beyond simply listing, comparing, and visualizing. Any debugging technology, whether interactive or offline, is more valuable when it supports this level of analysis. To put this more directly: our view is that robust support for scripting is absolutely essential in a tool intended for debugging at scale.

- *Filtering.* Much of the labor in this debugging effort involved writing agents to accomplish two types of filtering. We filtered up front, in advance of the offline batch job, by specifying exactly what information GDBase would collect and store. We then filtered after the fact, during post-processing, using SQL and other tools to extract relevant information from the collected data. Our debugging runs were most effective when we performed both kinds of filtering well. But we also found ourselves wanting a new type of agent that could perform a third type of filtering: a runtime or 'smart' agent that would analyze the data during execution and make real time decisions. As a minimum, such an agent would filter data on the fly by recognizing high value information. But a smart agent worthy of the name would also analyze as it collects, reporting conclusions in addition to (or instead of) raw data. We didn't write any such agents for this project, but one could certainly do so for any tool that supports runtime scripting. We believe runtime agents will begin to become important as offline debugging technology matures. Moreover, we are convinced that serious debugging at scale will prove most effective when user-analysts (and designers) think in terms of three rather than two levels of filtering: pre-process, runtime, and post-process.

- *Desktop Analysis.* Throughout this project we ran post-process analysis agents on our desktops rather than on the Lonestar cluster. At first we felt a little guilty doing this, thinking that we were not fully exercising GDBase unless we were running it end-to-end on a large cluster. Eventually we realized we were missing the point: a key (and somewhat unexpected) advantage of offline debugging is the ability to separate execution from

analysis. Using the desktop for post-process analysis (and for pre-process configuration tasks) isn't 'cheating' – it's smart!

Functionality

Regarding the functionality of offline debugging tools, our experience suggests the following:

- *Scalable Output.* In our view one important limitation of GDBase is the fact that each process writes its own database file.[6] While this approach is eminently reasonable for a prototype system, it does limit scalability: a very large offline debugging run, collecting massive amounts of data at frequent intervals, would result in many thousands of cores all writing to disk simultaneously. This would likely cripple a cluster's file system. In our view a truly robust approach to offline parallel debugging requires a output mechanism more sophisticated than this: fewer writers, fewer files, and at least some level of consolidation during program execution.

- *Parallel Support.* The GDBase prototype system supports debugging of MPI-based parallel applications. A production-quality offline debugger must include support for a wide range of parallel technologies, including threading models; accelerators and co-processors; and hybrid applications employing combinations of these languages and technologies.

- *Verbose vs. Terse Modes.* A high percentage of items in the GDBase database are routine messages that are important only in the early stages of debugging. These include messages for each process regarding initialization, availability of shared libraries, and whether the debugger succeeded in setting the requested breakpoints and watchpoints. Given our interest in controlling the size of the database, we would have found it helpful to have some degree of control over how much of this information the tool stores during a particular run. It seems reasonable to assume that any offline debugger would record some variation of this kind of information; we recommend implementations that allow the user-analyst to tune this kind of data.

Open Questions

Several open questions seem to us worthy of discussion and further research:

- *Off-the-Shelf vs. Customized Agents.* The GDBase developers included with the product's first release a number of high quality collection and analysis agents (scripts). In a perfect world perhaps we would have used these agents as-is to debug MGF. We were not able to do so, however. In most cases the provided agents were too generic and cast too wide a net; we

[6] During post-processing the analyst uses a GDBase tool to merge the database files into a single database.

needed agents that probed application-specific relationships among MPI messages from four precisely-defined locations in the code. Fortunately the generic scripts did serve the cause as effective demonstrations or templates: we had little difficulty customizing generic agents to produce the application-specific versions we needed. This raises a number of questions about collection and analysis scripts. To what extent is it possible to provide generic scripts that will prove valuable to user-analysts? What tasks would those scripts perform? How could designers produce scripts that are easy to customize (or configure) for application-specific work? Is there value in maintaining a growing archive (or multiple domain-specific archives) of customized and customizable scripts visible to the larger community? If so, what mechanisms would be most effective to manage and maintain such archives?

- *Wide Net vs Narrow Focus.* We found that our debugging runs were most effective when we scripted them so they had a narrow focus, collecting only a small, well-defined subset of information from specific points in the code. This was the opposite of our expectation going into the project: we thought we would use GDBase in a 'wide net' mode, asking it to collect large quantities of data (e.g. the values of all arguments from all MPI calls) then mining that data after the fact. We admit that focused sessions were practical for our MGF work (we knew enough about the issue to form reasonable conjectures early in the process regarding the probable cause and location of the bug). That won't always be the case, so 'wide net' sessions will sometimes be necessary. But we cannot shake the feeling that the jury is still out on what an effective 'wide net' debugging session might look like. We cannot expect to collect 'everything' when debugging complex applications at scale. What kinds of 'wide net' collections are likely to prove useful? How can we control the data collection to avoid stressing the cluster's file system? What data mining techniques will prove most useful? What practices will lead to effective workflows?

- *Collaborative Mode.* Perhaps in a perfect world all of our debugging would be interactive. In the real world some sessions may need to be offline. Is there room for a mode somewhere between the two: a session that proceeds autonomously unless and until interrupted and redirected by a human? If so, what would such a session look like? In the (distant) future it might resemble a conversation between two or more collaborators, one of whom is a highly intelligent software system capable of making decisions and recommendations regarding promising paths of inquiry. In the short term it doesn't seem unreasonable to imagine an autonomous session monitored from afar by a human analyst who wants to make some ad hoc tweaks from time to time based on intermediate results.

6 Conclusions

We are sold: this project leaves us convinced that large scale offline parallel debugging is achievable, effective, and ready for prime time. It seems clear that this technology will soon be mainstream. There is no magic bullet, however. Offline technologies can supplement but will not replace the best traditional interactive debuggers. In fact we see little need for dedicated tools that perform offline debugging. Instead, users will come to expect offline modes in high end debugging systems.

While we have learned many lessons exercising GDBase, one somewhat surprising conclusion rises above the rest for us: as problem size and complexity continue to increase, the key to effective debugging is robust support for scripting. Whether or not users are working interactively, much depends on the degree to which they can probe, discover, and analyze subtle relationships in ways that go beyond simply listing, comparing, and visualizing.

7 Acknowledgments

The authors gratefully acknowledge the National Science Foundation's support under grant 1019055.

References

[1] Allinea Software Ltd. Allinea Home Page, 2012. `www.allinea.com`.

[2] W. L. Barth. *Simulation of Non-Newtonian Fluids on Workstation Clusters*. PhD thesis, The University of Texas at Austin, May 2004.

[3] W. L. Barth and G. F. Carey. On a natural convection benchmark problem in non-Newtonian fluids. *Numerical Heat Transfer, Part B*, 50:193–216, 2006.

[4] G. F. Carey, R. McLay, G. Bicken, and W. Barth. Parallel finite element solution of 3d Rayleigh-Bénard-Marangon flows. *International Journal for Numerical Methods in Fluids*, 31:37–52, 1999.

[5] A. Digirilamo, K. Lindekugel, and D. Stanzione. A scalable framework for offline parallel debugging. In *Proceedings of the 9th LCI Conference on High Performance Clustered Computing*, April 2008.

[6] GNU Project. GDB: The GNU Project Debugger, 2012. `www.gnu.org/software/gdb/`.

[7] D. James, C. Rosales, and D. Stanzione. Offline parallel debugging: A case study report. In *Proceedings of XSEDE12: annual conference of Extreme Science and Engineering Discover Environment (XSEDE)*, July 2012.

[8] K. Kandalla. "re: [Hercura] Bio Code". Private communication with Robert McLay, Mar 2011.

[9] K. Lindekugel IV. Architecture for an Offline Parallel Debugger. Master's thesis, Arizona State University, 2008.

[10] MVAPICH Network-Based Computing Laboratory. MVAPICH Home Page, 2012. `mvapich.cse.ohio-state.edu`.

[11] Rogue Wave Software, Inc. Rogue Wave Home Page, 2012. `www.roguewave.com`.

[12] V. Springel. GADGET Home Page, 2012. `www.mpa-garching.mpg.de/gadget`.

[13] S. Sur. "re: A case we could use some help with". Private communication with Robert McLay, Feb 2011.

[14] Texas Advanced Computing Center (TACC). GDBase: An Offline Parallel Debugger, 2012. `https://github.com/TACC/GDBase.git`.