Petascale Debugging

Chris January, David Lecomber, Mark O'Connor
Allinea Software
{cjanuary,david,mark}@allinea.com

May 11, 2010

Abstract

The need for tools to debug at scale is painfully known to HPC developers - yet as machine sizes have raced ahead, debuggers have not kept pace. This has resulted in a large gap between the needs of users and the range at which full debugging capabilities can work for many years. This paper outlines major developments to Allinea's DDT debugging tool to introduce production-grade petascale debugging on the Oak Ridge Jaguar Cray XT5 system. The result has raised usability and performance of debugging by multiple orders of magnitude - and has already achieved record 220,000 core debugging at ORNL.

Keywords: Petascale, Debugging, Tools

1 Introduction

Debugging is an essential component in any software developer's toolbox. In parallel computing the additional complexities of interacting processes with communication, or multi-threaded processes with race conditions are frequent sources of bugs.

Parallel debuggers such as Allinea DDT offer an alternative to the laborious insertion of *print* statements into user code to identify problems. Debuggers make controlling and observing program behaviour and diagnosing a bug simple - bringing features such as stepping and stopping processes or the inspection of variables and data at runtime.

Although the number of cores in systems has rapidly increased, debuggers have lagged orders of magnitude behind in their capabilities, and also in where users actually perceive as their limit.

Concerns with the limitations of debugging at only small thousands of processes have been raised on many occasions (eg. [3, 4]) - along with the sentiment that something must be done about it. With a handful of HPC systems at over 100,000 cores, and many more at over 8,000 cores clearly there must be a solution to finding bugs at these scales.

This paper introduces changes that Allinea is making which have increased the proven level of its debugger, Allinea DDT, to over 220,000 cores. A fundamental change in its architecture has achieved performance at extreme scale surpassing previously recorded results

on only hundreds of cores.[1]

2 A new Architecture for Debugging

The traditional architecture of parallel debuggers consists of a frontend component running on a login node, communicating directly with each MPI process or each node in a job. Debuggers necessarily communicate commands to daemons running on each node, and then process returned messages. These returned messages can indicate many results - such as data values, process state or process stacks.

This architecture places linear performance penalties on every collective operation during debugging - from setting a common breakpoint, to pausing running processes. There have also been linear penalties in terms of system resource usage, such as file handles, filesystem access or memory.

We have developed a tree architecture that broadcasts commands and aggregates messages between the frontend and the compute nodes. Many messages in both directions are easily able to take advantage of the tree - with specific merge operators applied to each response type.

The Cray XT architecture allows point-to-point TCP communication between the nodes within the 3D torus of the system. In particular, the connectivity includes the compute nodes and therefore allows the debugger to use these as part of the tree.

Specific features of the XT also enable good performance during startup - where the ability to quickly launch daemons on every node was essential. Attention was paid to ensure that the shared (Lustre) filesystem only received a fixed and constant number of file accesses - independent of the number of processes.

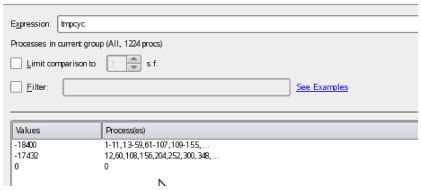

Figure 1: Examining a variable across all processes

The basic tree sees each tree node having a degree and depth that has been selected as a result of rough calculation, and actual verification at scale. It is difficult to provide a tree that will be perfect for every debugging operation as each such operation could involve a different amount of work or message traffic - for example merging of stack traces can vary according to the application, but some applications such as setting a breakpoint and returning a success or failure message always have an almost trivial merge operation.

3 Usability for Petascale Debugging

The second requirement of a petascale debugger is a usable interface through which the user will find bugs. There is a clear difference between what is sufficient and desireable for a handful of processes and the needs at a few hundred, or a few thousand processes.

DDT has been providing scalable components that reach such numbers

[1]This work is paid for as part of the Oak Ridge Leadership Computing Facility, and ORNL is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725.

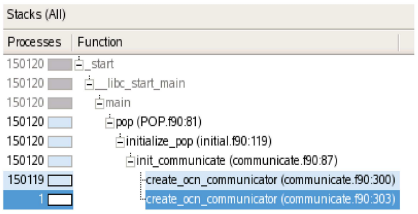

Figure 2: Examining stacks of all processes

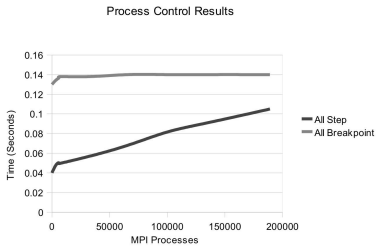

Figure 3: Time to add breakpoint or step all processes

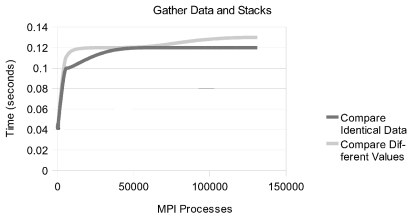

Figure 4: Time to gather data

for some time, as we note that - at scale - a debugger's job is to point out differences - something that becomes impractical to do manually when the user can no longer realistically click through each process in turn.

This difference highlighting is seen in features such as comparison of variables on multiple processes as shown in Figure 1, or in the parallel stack view of Figure 2 which groups the processes together in a tree by their current stack trace.

One innovation in this area made possible by the massive improvement in performance is that DDT will now automatically highlight when a variable changes or when its value is not the same across all processes - as the cost of doing this check is so insignificant.

4 Performance

Real HPC codes were involved in testing the performance of DDT - including S3D, GTC, MVH3/VH1 and POP - on the 12-core Jaguar XT5. Measurements were taken up to at least 131,072 cores in all cases - and also at larger sizes where machine availability permitted. Times recorded are the elapsed time until the GUI had completed updating.

Figure 3 shows both the time taken to step all processes including gathering all stacks, and the time to set a common breakpoint across all processes.

Figure 4 shows the time taken to compare data across all processes in two different scenarios - one whereby all the data is identical, representing a best case, and the second, where data is all different. In this latter case the variable evaluated was the MPI rank.

As is shown in both graphs - the approach has led to demonstrable scalability that will feel responsive to the user.

5 Other Approaches

The previous limitations led to suggestions that achieving scale could require fundamental changes in the process of debugging.

Relative debugging[5] involves user-specified verification points at which program state is compared between one instance of an application and another, running at different scales or on different architectures.

Lightweight debugging seeks to minimize the overhead of debugging but to retain some of the most important aspects of it - STAT from LLNL[2] is one of the most well known examples. This tool extracts process stacks at scale using a tree architecture and then identifies equivalence classes of processes based on these stacks. STAT can then invoke a full debugger such as Allinea DDT, on an appropriate subset of MPI processes.

Approaches such as lightweight debugging or comparative debugging, and other techniques can help the debugging process, but a strong benefit could be obtained using tools together. Current levels of interoperability are useful for examining the situation when a problem is detected - but it could be even more powerful if such tools become available as both interactive and non-integrated/off-line tools.

6 Conclusions

We have shown that full strength debugging can scale to the largest systems available today - and, given the logarithmic performance experienced, we can expect this form of debugging will reach significantly higher scales.

In each of the usual operations of debugging we have taken performance to a level that is required for using a tool to be seen as comfortable and usable.

Having broken previous debugging records, and established that petascale debugging is feasible - we are now actively considering options for increased automation and new features to add to this truly scalable debugging.

References

[1] Allinea Software, The DDT User Guide, http://www.allinea.com.

[2] Lee G., Ahn D. et al, Lessons learned at 208K: towards debugging millions of cores, in SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008.

[3] Report of the Workshop on Petascale Systems Integration for Large Scale Facilities, Lawrence Berkeley National Laboratory, Jan 2007.

[4] Report of Workshop on Software Development Tools for Petascale Computing, Aug 2007, Washington, DC

[5] Abramson D., Foster I., Michalakes J., and Sosič R., Relative debugging: a new methodology for debugging scientific applications, Communications of the ACM, volume 39, number 11, 1996.