INTEGRATING PERFORMANCE ANALYSIS IN PARALLEL SOFTWARE ENGINEERING

by

DAVID POLIAKOFF

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2015

THESIS APPROVAL PAGE

Student: David Poliakoff

Title: Integrating Performance Analysis in Parallel Software Engineering

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Allen D. Malony	Chair
and	
Scott L. Pratt	Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2015

THESIS ABSTRACT

David Poliakoff

Master of Science

Department of Computer and Information Science

June 2015

Title: Integrating Performance Analysis in Parallel Software Engineering

Modern computational software is increasingly large in terms of lines of code, number of developers, intended longevity, and complexity of intended architectures. While tools exist to mitigate the problems this type of software causes for the development of functional software, no solutions exist to deal with the problems it causes for performance. This thesis introduces a design called the Software Development Performance Analysis System, or SDPAS. SDPAS observes the performance of software tests as software is developed, tracking builds, tests, and developers in order to provide data with which to analyze a software development process. SDPAS integrates with the CMake build and test suite to obtain data about builds and provide consistent tests, with git to obtain data about how software is changing. SDPAS also integrates with TAU to obtain performance data and store it along with the data obtained from other tools. The utility of SDPAS is observed on two pieces of production software.

CURRICULUM VITAE

NAME OF AUTHOR: David Poliakoff

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR
Millsaps College, Jackson, MS

DEGREES AWARDED:

Master of Science in Computer Science, 2015, University of Oregon
Bachelor of Science in Computer Science, 2012, Millsaps College

AREAS OF SPECIAL INTEREST:

HPC, Parallel Computing, Performance Analysis

PROFESSIONAL EXPERIENCE:

Graduate Intern, Oak Ridge National Laboratory

Graduate Intern, Sandia National Laboratory

GRANTS, AWARDS AND HONORS:

First Place Student Paper Section, 2011 LA/MS Section of the Mathematical Association of America

ACKNOWLEDGEMENTS

This work benefitted from extensive work on the parts of Professor Allen Malony, Professor Mike Heroux, Dr. Jeremy Templeton, Dr. Karla Morris, Dr Lindsay Erickson, and Sandia National Laboratory

For Hadley

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Modern computational software codebases are large. For example, Trilinos is a project ”to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems” [**?**]. It provides a broad array of functionality, from simple C++ utility classes, to MPI wrapping functionality and distributed data structures, to solver libraries directly usable by domain scientists to solve problems. It has 344 authors listed in its git repository, and over 3 million lines of code written over years of development, with a need to be compatible with Fortran and support C++11. It also must provide functionality for traditional CPUs as well as future accelerators. Trilinos leverages the expertise of authors from the domain sciences, database experts, web experts, and experts in many computational technologies. All of this creates significant software engineering complexity simply to maintain functionality, the Trilinos team has developed a system called TriBITS (Tribal Build, Integration, and Testing System) to provide a software engineering framework which manages this complexity. TriBITS does not provide an ability to track performance, it is not possible to answer questions such as ”how has the performance of Trilinos changed in the past year?”

MOAB is a piece of software out of Sandia National Laboratories built on top of Trilinos. It has been in development for a little under two years, and is intended to solve problems in fluid dynamics. It is built in C++, and has 34 thousand lines of code written by four authors. It is developed by three mechanical engineers and a computer scientist. This early in the development, changes to existing code happen rapidly, functionality goes from being prototyped to being part of the codebase in weeks. The team lacks the time to manually answer questions like ”how did my change from using one Trilinos data structure to another impact all of the tests in my test suite?”

Each of these teams lacks the ability to track how their performance is changing without significant manual effort on the part of developers. They do have well defined tests, they use the CMake system to build software, which includes the CTest system to test software. They do use git to manage versions of their software, giving information about who authored changes to the code and an author provided summary of the changes they made. We leveraged these abilities to provide a way to describe the state of the software, and developed a system to track the performance of the software as it is developed. Moab and Trilinos were chosen as examples to demonstrate the utility of this system.

CHAPTER II

MOTIVATION

This work is motivated by studies done on maintaining the functionality of software systems. One canonical result in software engineering is that the time between bugs being introduced and being discovered will drastically increase the cost of maintenance (Boehm, 1988; Schofield, 2008). That is, if a bug is introduced to software, discovered, and immediately removed, the cost is as simple as refactoring the code causing the bug. If the bug stays in the code for years, other code which relies on that bug might need to be refactored, this could be thousands of line of code which are impacted by the bug. This has led to the development of systems which monitor code bases for functionality bugs. CDash is a system which tracks bugs in codebases as they are developed, the aim being to minimize the time to discovery of buggy code.

A similar argument establishes the need to track the performance of code. If a bug is introduced which doubles the runtime of a piece of code, and that bug is immediately discovered and fixed, the cost is minimal. If the bug is introduced, sits in the code for years and is only discovered then, all of the code which relies on it might require refactoring. Worse, early code can establish the standard practices for a codebase. In one example from the Moab team, early on we used Trilinos to manage the distribution of data structures across MPI processes in our code. We unknowingly started using a code path in Trilinos which added data to these structures and immediately distributed those changes to other processors. We would make these function calls repeatedly, without needing that data to be distributed immediately. Moving to an idiom which added data to the data structure but only distributed the data when necessary was a much faster process. Because we were tracking our performance, we immediately noticed the problem, and did not use the slower code path in other parts of the code. If we had not noticed the bug, we might have made the same error in other parts of our code, drastically increasing the cost of refactoring the bug out of the code. In fact, there have been project and domain specific systems to track performance in the past (Pradel, Huggler, & Gross, 2014)

From canonical software engineering, we know that to resolve functionality bugs immediately is to resolve them cheaply. If a bug is fixed before other code relies on it, the fix is simpler. The same forces which cause these phenomena apply to performance bugs as well. Further, these performance bugs can establish idioms which are used in other places in the code, so the cost to fix scales not only with the code which directly uses the buggy code, but with any other parts of the software which use the idioms established in the buggy code. This is our motivation for introducing a system which tracks performance, it is an attempt to allow users to quickly discover their performance bugs in order to minimize the cost of bug fixing.

CHAPTER III

DESIGN

To understand the design of the system, it is beneficial to look at the kinds of questions the system is intended to help answer. A programmer might want to refactor a function and see how the performance of their tests is impacted. This requires a few capabilities, described in Figure 1.

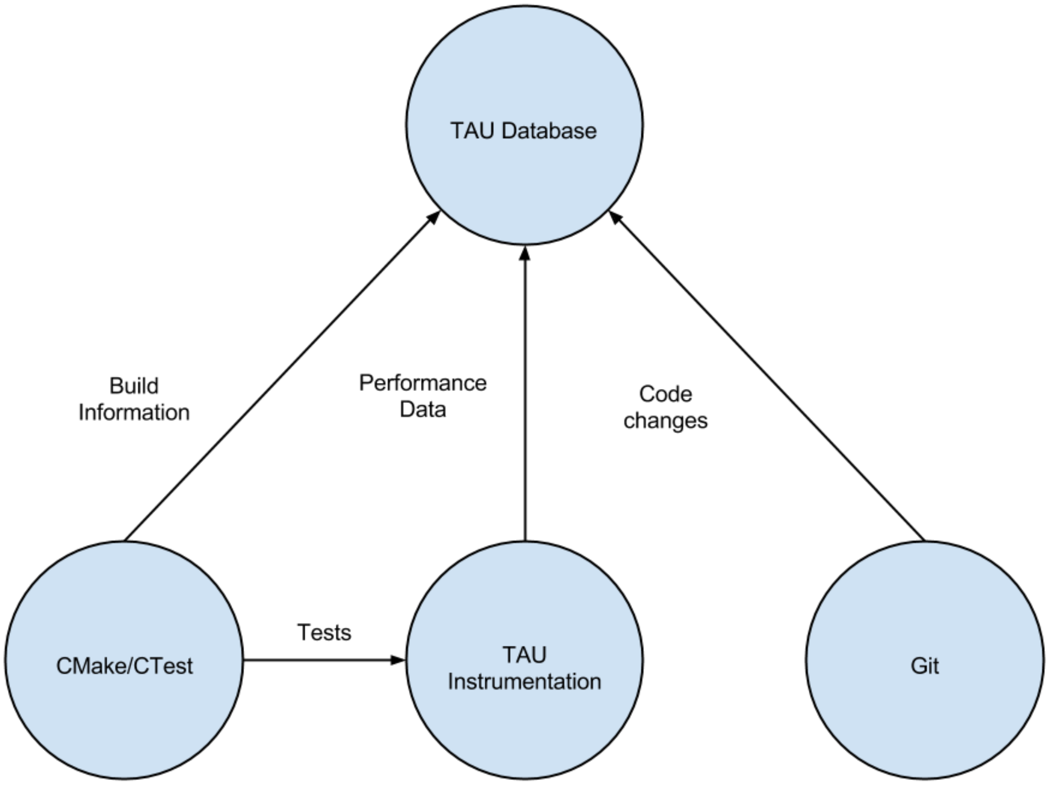


FIGURE 1. The components of the system, and the information they provide one another

1. The ability to measure the performance of a test

2. The ability to define a consistent test which can be measured

3. The ability to know what changes were made to a piece of code

4. The ability to store all of these results

5. The ability to view all of these results

The first constraint in the system is slightly complicated in that it has to be able to obtain performance data from a broad array of architectures. Modern computational systems like Trilinos are intended to work on GPU and Xeon Phi, to use MPI where appropriate, and to work with any future architectures that arise. In particular, needing data from MPI is a significant constraint, as MPI runs on multiple nodes of a supercomputer (Forum 1994). They also can use Fortran, C, or C++. This constraint of needing performance data on a broad array of architectures was satisfied by TAU (Shende 2006). Given an executable on most HPC architectures, TAU will provide performance data for a run of that executable

However, on its own TAU only knows about source code and executables. As this system is designed to analyze test which change over time, there is a need to be able to associate an executable with a test. Further, there is a need to know the characteristics of that test not present in the executable. Knowing whether a run of the test came from a debug or release version of the code provides valuable information, comparing performance data from builds which are not meant to be compared will provide faulty data. The CMake/CTest system satisfied these constraints, CMake allows you to define builds and tests (Martin & Hoffman, 2006). Further, it stores extensive data about these artifacts in a format other tools can use.

However, CMake only defines a build and test. It can not provide data about what the source code for a given test looked like eight code revisions ago. It only provides data about the status of a test at a given point in the development process.

For this data the system relies on git. Git is a utility for tracking the changes made to a project, which is a perfect fit for this constraint in the project. When changes are made to a project, git tracks the specific changes, and also requires a message to be added along with the changes to describe the revision (Chacon 2009).

With these three constraints satisfied, the types of questions we set out to answer are answerable. CMake gives us tests, TAU gives us performance data, and git gives us revision information. But all of this data needs to be extracted and stored. The extraction will be described in the implementation section. For storage and correlation of data from different sources, a database is ideal. TAU comes with its own database system for tracking performance data (Huck 2005). In this design we extend the TAU database to accept the information from these other sources and associate it with its own performance data.

Finally, the need to view data is done through a modern web interface. In these collaborative development teams information needs to be shared over distances and the web is ideal for that. Further, there may eventually need to be an ability to restrict certain data for confidentiality reasons, and this is a well explored problem in web development. The details of this design are described in their own section.

This design was arrived at through a series of compromises. The biggest sacrifices in the design were in the loss of automation and other convenience for the sake of security, and in the choice of which displays of data to develop among the many displays different teams might want to answer different questions.

One natural design would be to have a web interface which automatically speaks to a database to provide data. This is a common practice. However, in a a national lab environment, some data can not be shared, and some ports can not be accessed remotely. Here we sacrificed the convenience of following standard practices in order to maintain security, data is pulled from the database in a user-controlled way and uploaded to a web server.

The greatest design decision that had to be made in all of this is which questions to allow a user to answer. These decisions are discussed in detail in the next section, interface development.

CHAPTER IV

WEB INTERFACE DEVELOPMENT

One issue that was immediately apparent was the volume of data this system would produce, and the number of analyses that could be run on it. In one brainstorming session with the Trilinos team, the authors thought of some views we might want to support.

1. For a given test among possible thousands of tests, treat the commit history as a timeline, and display the performance of various functions as the software is updated. Include version control information for significant commits, with commit messages included to explain the significant performance changes. This was intended to provide a general view of the health of a given test across time

2. If a user clicks on a specific run anywhere in the interface, display the performance data on that trial, including build characteristics, diffs on the file, and the commit message associated with the change. This was intended to answer any questions about a seemingly anomalous data point, if a test increases in runtime by a factor of two, this should identify why.

3. For a given programmer of the software, how do their commits impact the performance of the software? Who are the programmers providing the best commits? Where is each programmer doing the best development? This is intended to help map tasks to programmers. For instance, if one programmer does great work on Zoltan2 part of the project and another does well in Stokhos, this would be valuable information for a manager to know.

4. For a given function in the program among possible thousands of functions, treat the commit history as a timeline, and show how the function has changed its performance across the various CTest tests. This was intended to be used by a developer who refactored a given function in order to figure out whether their changes had a positive or negative effect

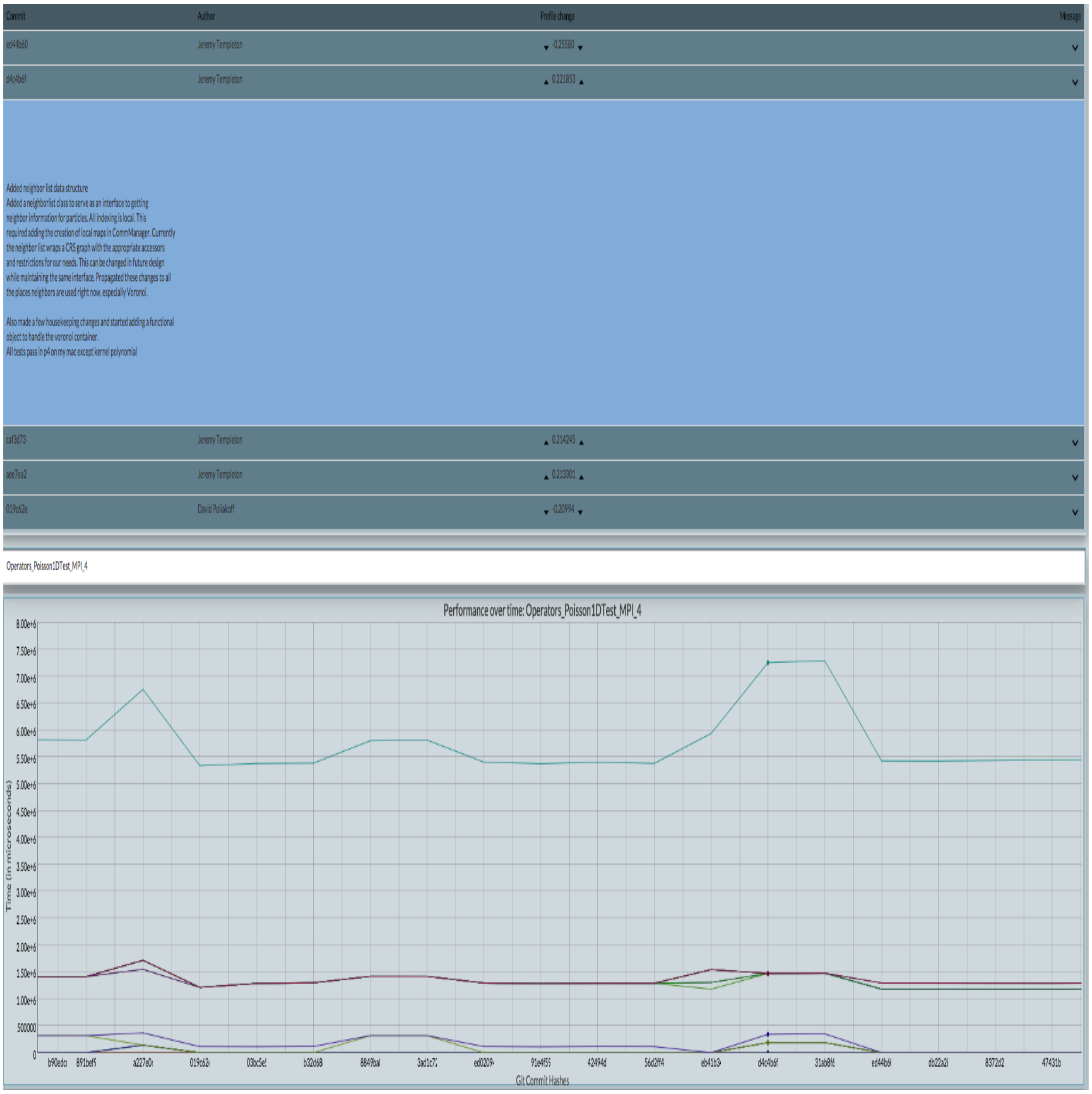
5. For a given commit, check its impact on the entire test suite. For each test, how did the commit impact performance? Were there any functions which were particularly impacted by the commit? These are the questions this view would answer

6. Whole repository view. Given some weighting of test importance to performance, provide an analysis of the health of the entire repository. This is intended for a high level manager to be able to look at one page and get a general overview of the health of the repository.

7. More tentatively, do multiple repository analysis. MOAB relies on Trilinos, is it possible to provide views of how changes in Trilinos impact MOAB performance? Is it possible to see how a change made in MOAB saw better performance of Trilinos functions and share that data with other users of Trilinos?

There is a wide variety of data there, with a wide variety of possible displays. We needed to be able to display this information to the user in an intuitive and non overwhelming way. After a consultation with groups within the Trilinos project, as well as conversations with other developers, we decided that the view which looks at a test across time was the natural starting point. In listening to developers, the questions asked were most frequently something along the lines of ”how has our ability to do a given task our software is meant to accomplish changed over time?” As tasks are often represented with tests, we settled on this view. We decided that the view would show a given test. On top would be a table containing the most significant commits that impacted the test, with their author, commit message, and change to performance. This would allow a user to quickly see the biggest changes to a test. For more in-depth information, we added a graph. This graph would plot various functions in the test in different colors, with commits going along the X axis and time spent in that function going along the Y axis. This allows you to see the performance of various functions in the test over time.

Figure 2 shows part of the interface, the legend is large and is omitted. Here we see the interface described in bullet point 1, with a timeline of commits, different functions represented by different colors, and a table containing significant commits, with one commit expanded to display the commit message. This interface is viewable online, which demonstrates that it is easily shared with collaborators (Poliakoff 2015). The interface takes advantage of a number of web technologies, with Twitter Bootstrap providing the look of the table, Polymer providing the ability to manipulate the data stored in the table and graphs, and Dygraphs providing the graph itself. In addition, standard JS/CSS/HTML/JQuery are used(Bidel 2015). This leads to an interface that can be viewed by external collaborators, in languages that are familiar to any web developers, providing extensibility and maintainability.



Commit	Author	Profile change	Message
ed44b60	Jeremy Templeton	▼ -0.25580 ▼	∨
d4c4b6f	Jeremy Templeton	▲ 0.221853 ▲	∨
Added neighbor list data structure Added a neighborlist class to serve as an interface to getting neighbor information for particles. All indexing is local. This required adding the creation of local maps in CommManager. Currently the neighbor list wraps a CRS graph with the appropriate accessors and restrictions for our needs. This can be changed in future design while maintaining the same interface. Propagated these changes to all the places neighbors are used right now, especially Voronoi. Also made a few housekeeping changes and started adding a functional object to handle the voronoi container. All tests pass in p4 on my mac except kernel polynomial			
caf3d73	Jeremy Templeton	▲ 0.214245 ▲	∨
aee7ea2	Jeremy Templeton	▲ 0.213301 ▲	∨
019c62e	David Poliakoff	▼ -0.20994 ▼	∨

Operators_Poisson1DTest_MPI_4

FIGURE 2. The web interface showing a test in which performance bugs are introduced and resolved

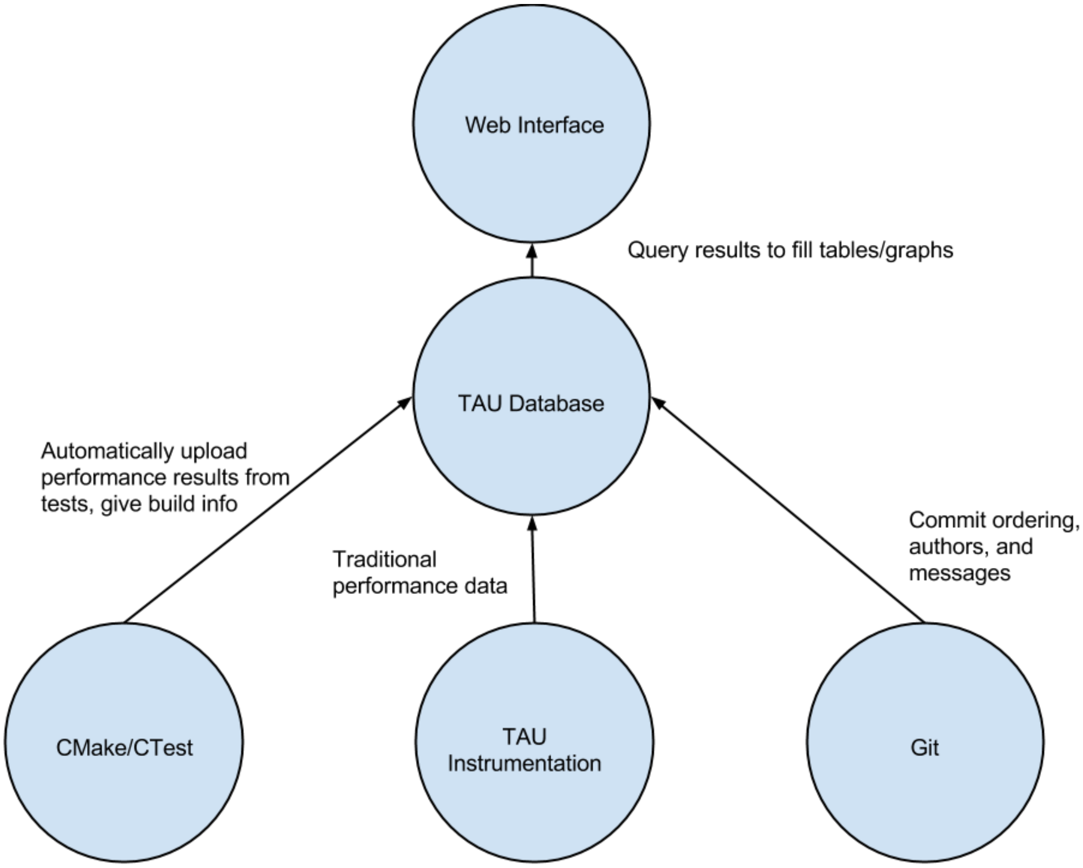

Web Interface
Query results to fill tables/graphs
TAU Database
Automatically upload performance results from tests, give build info
Traditional performance data
Commit ordering, authors, and messages
CMake/CTest
TAU Instrumentation
Git

FIGURE 3. The mechanisms by which the components of the system interact with one another

CHAPTER V

IMPLEMENTATION

In this section we discuss how the above design choices were implemented. Figure 3 shows how the implementation is tied together, it is a lower level view than the design. In short, various tools which provide data upload that data to a TAU database modified to accept the data, queries are then run on that data to provide output usable by a web interface designed to accept that data.

DATABASE MODIFICATION

The most common organization of data in a TAU database is a hierarchy, with applications containing multiple trials, and those trials being associated with the various timers, counters, and metadata which TAU collects. For this work, TAU needed some additional context, an ability to store and recognize the special data being added to it by the other tools. First, a relationship was established between these trials and version control information in a git repository, including an order in which commits were applied. This allowed for questions such as what are the ten functions behaving most differently from the last commit to this one? The database also had to be aware of the CTest notion of tests, to answer questions like how did the behavior of the test MultiplyMatrices change from the past commit to this one? In addition, the CMake build characteristics needed to be specially identified so that people could quickly ask questions like which build properties are different between trial X and trial Y to ensure that any difference in data between two tests was caused by changes to the code and not just different build flags.

Further, a set of customized queries was developed to answer some of the questions this additional information exposes relevant data for. The database was modified to allow the storage of data to answer the questions in the preceding paragraph, but the data still needed to be filtered, grouped, and displayed in a way that users would find sensible. This required a query which found the major functions of a test and got their runtime across the different commits in the code base. It also required the ability to observe a function across different tests across different commits.

CTEST MODIFCATIONS AND USAGE

At the beginning of this work, CTest lacked necessary functionality. Specifically, it was not possible to run a command to operate on the results of a test. Adding in this functionality was a simple patch. Now it is possible to tell the testing system to run the commands to upload necessary created data to a database. The more useful contribution made by the authors is the ability to respond to different test states, to do something different based on whether a test passes or fails. Once this functionality was in place, SDPAS had the functionality it needed. CTest knows about the results of CMake, so we were able to tag the profiles created by a test with metadata from the CMake build process. Further, we were able to obtain whether the test passed or failed and avoid rewarding tests that failed quickly rather than passing slowly. This part of SDPAS was valuable independently of the version control information, the ability to reproduce builds and attempt to reproduce questionable performance data is valuable for performance analysis. If a test shows an inexplicable change in runtime it is now possible to recreate some of the features of the build to see whether the result is repeatable.. Finally, the work of the TriBITS team in following version control data inside CMake was leveraged to get version control information uploaded to our own database.

Existing CTest functionality was essential to this system. CTest already provides a way for users to express as test of the software, often getting performance data was just a matter of increasing the problem size given to these tests or increasing the number of MPI processes being used to solve a problem. This ability to have comparable units to compare between commits was essential to SDPAS.

GIT ENFORCEMENT OF PERFORMANCE MEASUREMENT

Thus far, everything about the system has focused on what it can do. One issue often encountered in software engineering tools is that people will stop using them at the first sign of inconvenience. A user could very well decide they wanted to add functionality immediately and bypass the performance measurement. Some development teams want this, but some may want rigorous enforcement of all commits being analyzed. In the MOAB teams case, we wanted performance data on every commit. Git provides hooks to allow actions to happen every time a commit is pushed to a database, even to reject a pushed commit in the event the commit does not conform to some policy of the maintainer of the repository. The MOAB team used this functionality to ensure that every commit has tests run on it, and performance data uploaded to a database. The Trilinos team sees such a volume of commits that this approach was not desired, and so this functionality was not used.

CHAPTER VI

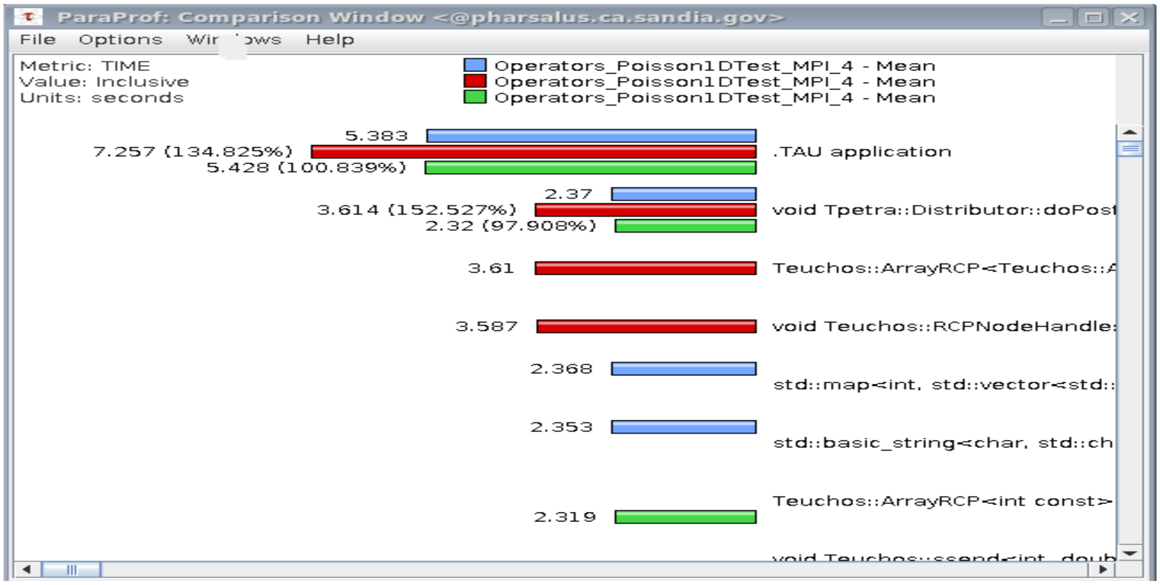
EVALUATION

Thus far, everything about the system has focused on what it can do. One issue often encountered in software engineering tools is that people will stop using them at the first sign of inconvenience. A user could very well decide they wanted to add functionality immediately and bypass the performance measurement. Some development teams want this, but some may want rigorous enforcement of all commits being analyzed. In the MOAB teams case, we wanted performance data on every commit. Git provides hooks to allow actions to happen every time a commit is pushed to a database, even to reject a pushed commit in the event the commit does not conform to some policy of the maintainer of the repository. The MOAB team used this functionality to ensure that every commit has tests run on it, and performance data uploaded to a database. The Trilinos team sees such a volume of commits that this approach was not desired, and so this functionality was not used.

MOAB

MOAB is undergoing rapid development, with new capabilities introduced every day. This means that an ability to quickly detect performance bugs is essential, any delay could lead to significant amounts of code being built on top of broken software, leading to expensive bugs to fix. Figure 4 shows the resolution of one such bug.

Figure 1 showed a test in the MOAB suite at a time that represents the utility of this kind of software. The expanded commit message is one where a team member added functionality to the test. This increased the runtime by approximately 20



Metric: TIME
Value: Inclusive
Units: seconds

ParaProf: Comparison Window <@pharsalus.ca.sandia.gov>
File
Options
Wir 'ows
Help
Operators_Poisson1DTest_MPI_4 - Mean
Operators_Poisson1DTest_MPI_4 - Mean
Operators_Poisson1DTest_MPI_4 - Mean
5.383
7.257 (134.825%)
5.428 (100.839%)
.TAU application
2.37
3.614 (152.527%)
2.32 (97.908%)
void Tpetra::Distributor::doPost
3.61
Teuchos::ArrayRCP<Teuchos::A
3.587
void Teuchos::RCPNodeHandle:
2.368
std::map<int, std::vector<std::
2.353
std::basic_string<char, std::ch
Teuchos::ArrayRCP<int const>
2.319
void Teuchos::ssend<int, doub

FIGURE 4. A TAU view of the different versions of the code

percent. Because this testing happens automatically, we saw this result the next day, with only time for one more commit to enter the repository. With the legend, we were even able to isolate which function was taking longer. we were able to call the collaborator and ask whether this increase was expected. It wasnt, we looked with the collaborator at those specific trials, and figured out which function was the problem.

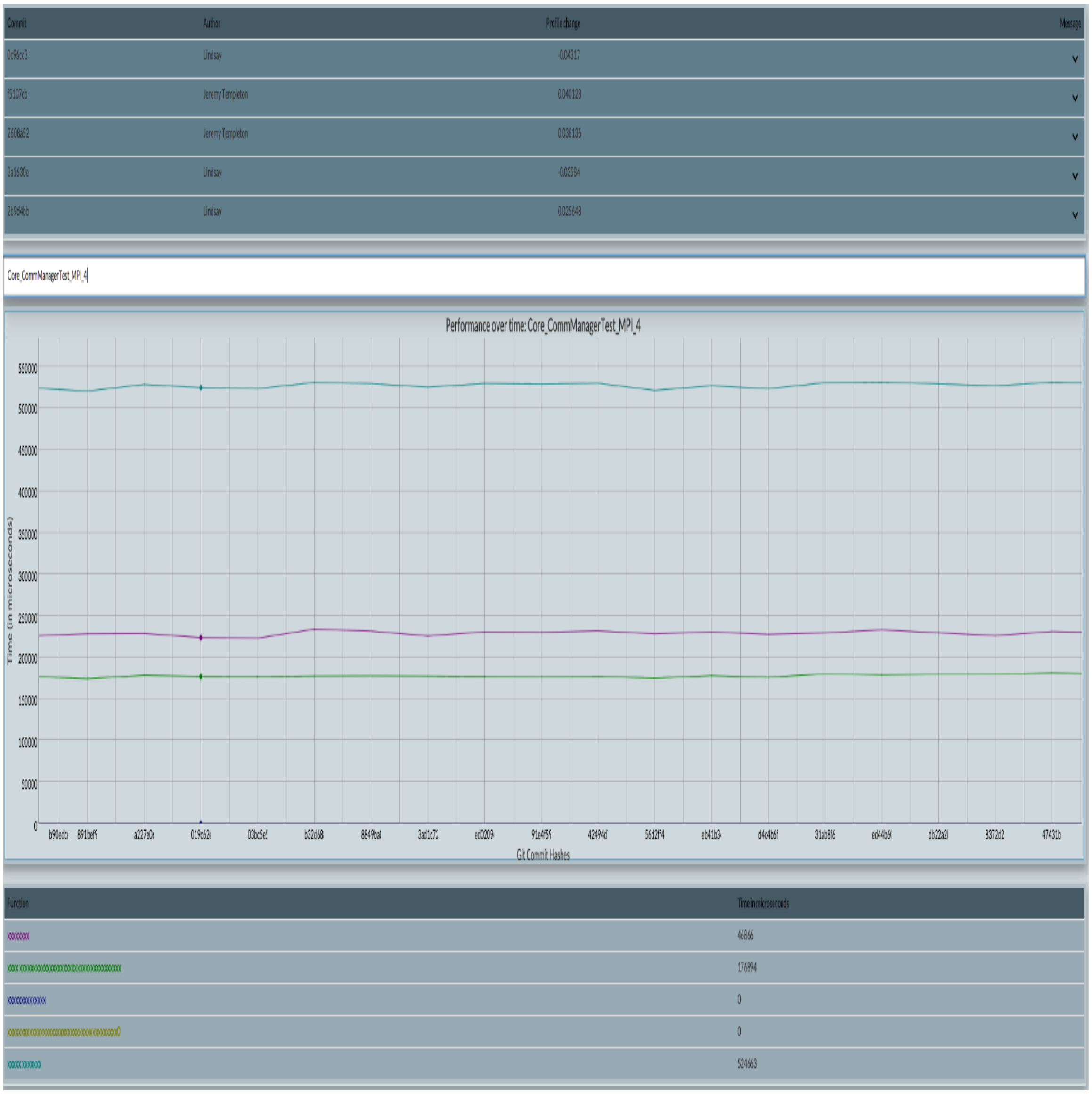
Because we had the version control information and test name, we were able to select these trials from the TAU database, and view them in TAU itself. In one case, the problematic function is one called doPostsAndWaits. This is where Trilinos distributes data in its data structures. We saw that this code was getting called more, implying that we were distributing data more often than necessary. We looked through the modified code, found where this was happening, and fixed it. Figure 4 shows this action, showing the performance of code before the introduction of a bug, while a bug is in the code, and after the bug has been resolved.

There are a number of these small hills in the graph, in which a bug is introduced and quickly fixed. Two of the surprising results of this work were how often these performance bugs get introduced and how easy they often were to fix. Often users would change code around the interface to a distributed object in Trilinos, the changed MOAB code would use some slower Trilinos code. Fixing this would be as easy as pointing it out and then moving to some quicker code in Trilinos. As mentioned in the motivation, fixing these bugs early meant not building code on top of them, making refactoring cheaper. It also meant we did not make the same mistakes in other parts of the code. In the absence of this knowledge, the shapes of these performance graphs could be a set of ever higher plateaus as buggy code is introduced, not fixed, and then replicated elsewhere.

The system was largely effective in MOAB, commits happened slowly enough and impacted a large enough portion of the code that every commit could be looked at. Still, there were difficulties. Among them was the cost of detecting an insignificant commit. Figure 5 shows a test of the very simple CommManager, a class in MOAB. This code has not changed at all, the slight change in performance is attributable to noise. Still, we had to verify it was insignificant by looking at these flat lines every few commits. Future work would include an ability to pinpoint significant changes and provide an alert to the user when they occurred. That said, the cost to the MOAB team to use the system is fairly low, all of the data collection and publication are automated, once the system is set up the only costs are in maintaining a TAU build of the software and observing the interface.

TRILINOS

The involvement with Trilinos was informative due to the wildly different needs of the two teams. In the MOAB case, there was a team responsible for all of MOAB,



Commit	Author	Profile change	Message
0c96cc3	Lindsay	-0.04317	∨
f5107cb	Jeremy Templeton	0.040128	∨
2608a52	Jeremy Templeton	0.038136	∨
3a1630e	Lindsay	-0.03584	∨
2b9d4bb	Lindsay	0.025648	∨

Core_CommManagerTest_MPI_4

Function	Time in microseconds
xxxxxxxx	46866
xxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	176894
xxxxxxxxxxxxx	0
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0	0
xxxxx xxxxxxx	524663

FIGURE 5. A stable git repository. Right now there is no automatic categorization of ”interesting” and ”uninteresting” graphs, requiring additional user work.

and that was who we would talk with. In the case of Trilinos, there are a multitude of packages developed by different teams with different goals. Further, it is a much more active repository. This leads to an incredible volume of changing code.

The Trilinos work began after the MOAB work, so many poor design decisions from the MOAB focused design were tested. Among these were a dropdown to select tests that worked for MOABs 40 tests, but was obviously unworkable on the 700 Trilinos tests. It was possible to look through 40 tests and ignore the boring ones, but manually scanning 700 tests is infeasible. The practice of testing every commit was okay with Moab, which would have at most a few commits a day, but Trilinos gets more than 10 commits a day. The MOAB team could focus on one build of the software and be happy with the resulting data. Trilinos has teams which want different build configurations, different customized instrumentations, some teams want different builds of TAU or other underlying libraries. One place SDPAS initially failed is working in this much larger environment.

One problem which remains is the tracking of different builds in the same interface. If different people want to track different builds in SDPAS, they need multiple SDPAS installs. Future work will involve finding ways to allow these multiple builds in a single SDPAS install with obvious differentiation between different builds in order to avoid invalid comparisons.

What has been solved is the problem of commit volume. For Trilinos, we have moved to a nightly testing model. Each night the performance is tested, each morning the performance is looked at. In cases where somebody finds tests of interest, they can see all the commits made that night. If they are particularly interested in what happened, they can even schedule a run of all the commits that night and the graph will be populated with this additional data. The problem of scanning

700 tests was solved through a web library. Tests in Trilinos have names of the form Package_Subpackage_Subpackage..._CapabilityArea_TestName. Autocomplete libraries exist for web interfaces, these provide the ability to specify each underscore-separated part of the name independently.

The problem of managing such large codebases is not completely solved. However, even in this case users are able to follow tests of interest, or to track how changes they made to code impacted various tests.

CHAPTER VII

FUTURE WORK

Much of the future work is mentioned as flaws in the current methodology. Work needs to be done in identifying the difference between significant and insignificant commits. This would significantly reduce the cost of observing performance. There are a few possibilities. First is to simply use a threshold, and report all performance shifts greater than that threshold. The issue with that is in large codebases, changes can be slow but consistent. That is, there may not be a commit which changes runtime by 25%, but there might be a number of commits in succession which each raise runtime performance by 1%, leading to the same result. Theres also the possibility of having tests which are meant to represent the health of the codebase as a whole, and only run the full suite if those tests show change. Ultimately, the work of filtering out insignificant commits has solutions, the main stumbling block is in picking the solutions users can agree on.

Another related flaw is in the handling of noise in performance data. Currently the way this works is that if the user sees a significant performance change, they can manually rerun the test and see whether the result was noise or significant. The system captures information about the CMake build, so it is possible to redo old builds. If the previous problem of identifying significant commits were solved, this would be a valuable bit of work.

Of great interest is the ability to provide richer information to users out of this data. The possibilities in this area are extensive.

1. Currently the work focuses exclusively on analysis at runtime and tying that back to commit messages. No work was done in tying the changes back to

actual code. That is, if the system sees a change in function MultiplyMatrices at a given commit, there is no investigation of the contents of the diff for that commit in MultiplyMatrices.

2. The move into analyzing multiple repositories that depend on one another has not started. That is, if Moab and Albany both rely on Trilinos, it would be interesting to inform members of Albany about how changes in Moab reduced their runtime in Trilinos. This sharing of expertise among users of a library would be valuable to the computational science community

3. Moving the other direction, it would be nice if Trilinos knew that Moab used it, and could have information about how changes to Trilinos impacted performance in Moab.

There is a lot of research that can be done in this area.

Additional work is in a tighter integration with git. Currently, the system looks at commit messages. Git also provides the ability to look at differences in source code between versions. So we can know that a given commit increased time in MultiplyMatrices, but do not know the source code changes to MultiplyMatrices without moving back to using git from the command line. This tighter integration, with views into the source code itself, would allow for the study of these performance artifacts in greater detail. There is a large space for automated performance analysis to be done on this, if we see that one code which moves from using fast code paths in Trilinos to slow ones suffer, we could warn other code bases when they make similar moves. Much of the future potential of this project is in deeper reasoning about the source code being developed, tied to analysis being done over multiple repositories in order to share knowledge between development teams.

Finally, one issue which did not come up in either Trilinos or Moab but which will certainly come up in the future is the definition of test used in SDPAS. Currently a test is a single run of an application. Scaling tests are common in computational science, the ability to reason about how scaling changes over time would be valuable. More generally, providing greater flexibility in how users define tests would allow for a greater variety of software to use this system.

CHAPTER VIII

CONCLUSION

Currently, high performance computing teams lack a way to analyze how changes to their software impact performance. In this thesis we described a system called SDPAS to track performance. The modifications necessary to the component systems were described. The utility of various views of changes in performance in software developed were described. We observed the use of the system on two different applications, Moab and Trilinos. While the test did show the value of observing performance in software development processes, many future areas of research were discovered.

REFERENCES CITED

Heroux, M. (2015). The Trilinos Project. Retrieved June 8, 2015, from http://trilinos.org/

Schofield, J. (2008). Security-Enhanced Quality Assurance, Testing and Project Management. 7th Annual QAI&QAAM Regional Conference.

Boehm, B., & Papaccio, P. (1988). Understanding and controlling software costs. IIEEE Trans. Software Eng. IEEE Transactions on Software Engineering, 1462-1477.

Martin, K., & Hoffman, B. (2006). Mastering CMake: Updated for CMake version 2.2. Clifton Park, New York: Kitware.

Shende, S. (2006). The Tau Parallel Performance System. International Journal of High Performance Computing Applications, 287-311.

K. A. Huck and A. D. Malony (2005). Perfexplorer: A performance data mining framework for large-scale parallel computing. 2005 ACM/IEEE conference on Supercomputing

M. P. Forum (1994) Mpi: A message-passing interface standard.

Chacon, S. (2009). ¡i¿Pro Git¡/i¿. Berkeley, CA: Apress

Poliakoff, D. (2015). Git Performance Viewer. Retrieved June 8, 2015, from http://nic.uoregon.edu/ poliadz/dist/tableImage7.html

Pradel, M., Huggler, M., & Gross, T. (2014). Performance regression testing of concurrent classes. ¡i¿Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014¡/i¿.

Bidel, E. (2015). Polymer - Welcome. Retrieved June 8, 2015, from https://www.polymer-project.org/