Understanding HPC Development through Automated Process and Product Measurement with Hackystat

Philip M. Johnson
Michael G. Paulding
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822
johnson@hawaii.edu
mpauldin@hawaii.edu

Abstract

The high performance computing (HPC) community is increasingly aware that traditional low-level, execution-time measures for assessing high-end computers, such as flops/second, are not adequate for understanding the actual productivity of such systems. In response, researchers and practitioners are exploring new measures and assessment procedures that take a more wholistic approach to high performance productivity. In this paper, we present an approach to understanding and assessing development-time aspects of HPC productivity. It involves the use of Hackystat for automatic, non-intrusive collection and analysis of six measures: Active Time, Most Active File, Command Line Invocations, Parallel and Serial Lines of Code, Milestone Test Success, and Performance. We illustrate the use and interpretation of these measures through a case study of small-scale HPC software development. Our results show that these measures provide useful insight into development-time productivity issues, and suggest promising additions to and enhancements of the existing measures.

1. Introduction

High performance computing systems are becoming mainstream due to decreasing costs and increasing numbers of application areas with computation and/or data intensive processing. With this interest, however, comes new challenges. For example, recent initiatives in the HPC community [8, 1] have concluded that low-level HPC benchmarks of processor speed and memory access times no longer necessarily translate into high-level increases in actual development productivity. Put another way, the bottleneck in high performance computing systems is increasingly due to software engineering, not hardware engineering.

To make matters even more interesting, high performance computing application development often differs in significant ways from the systems and development processes traditionally addressed by the software engineering community:

- The requirements often include conformance to sophisticated mathematical models. Indeed, requirements may often take the form of an executable model in a system such as Mathematica, and the implementation involves porting to the HPC system.
- The software development process, or "workflow" for HPC application development may differ profoundly from traditional software engineering processes. For example, one scientific computing workflow, dubbed the "lone researcher", involves a single scientist developing a system to test a hypothesis. Once the system runs correctly once and returns its results, the scientist has no further need of the system. This contrasts with standard software engineering lifecycle models, in which the useful life of the software is expected to begin, not end, after the first correct execution.
- "Usability" in the context of HPC application development may revolve around optimization to the machine architecture so that computations complete in a reasonable amount of time. The effort and resources involved in such optimization may exceed initial development of the algorithm.

Fortunately, there is an emerging interdisciplinary community involving both HPC and software engineering researchers and practitioners who are attempting to define new ways of measuring high performance computing systems, ways which take into account not only the low-level hardware components, but also the higher-level productivity costs associated with producing usable HPC applications.

This paper presents an approach to investigating the software engineering problems associated with high performance computing system application development. It involves the introduction of technology into the HPC development environment which unobtrusively gathers process and product data. This process and product data can be used for two purposes. First, it can be used to provide a more wholistic perspective on productivity, one that includes measures of performance, functionality, and development. Second, it can be used to provide new insight into the process of high performance system application development, which can be used to identify bottlenecks in the development process and assess the consequences of process or product changes on these bottlenecks. We have been applying this approach to an ongoing case study of high performance computing system application development in our laboratory, and this paper reports on our initial results.

The remainder of the paper is organized as follows. Section 2 introduces the technology we have developed, called Hackystat, which supports unobtrusive collection and analysis of product and process measures. Section 3 introduces "Software Project Telemetry", which is the principal approach to measurement collection and interpretation we have adopted for this research. Section 4 introduces a case study adapted from the Truss Purpose-based Benchmark (PBB) [3], which uses the problem specification but collects and analyzes an alternative set of metrics. Section 5 presents our initial conclusions from the use of these metrics and our future directions.

2. Automated process and product measurement with Hackystat

An important characteristic of our approach to understanding HPC software development and productivity is that measures of product and process must be automatically collected. This requirement limits the kinds of data we can collect, but dramatically lowers the cost of collecting these measures and provides a level of scalability for measurement not possible with expensive, manual data collection.

For the past several years, we have been developing a framework for automated software development process and product metric collection and analysis called Hackystat. This framework differs from other approaches to automated support for product and process measurement in one or more of the following ways:

- Hackystat uses sensors to unobtrusively collect data from development environment tools; there is no chronic overhead on developers to collect product and process data.
- Hackystat is tool, environment, process, and application agnostic. The architecture does not suppose a specific operating system platform, a specific integrated development environment, a specific software process, or specific application area. A Hackystat system is configured from a set of modules that determine what tools are supported, what data is collected, and what analyses are run on this data.
- Hackystat is intended to provide in-process project management support. Many traditional software metrics approaches are based upon the "project repository" method, in which data from prior completed projects are used to make predictions about or support control of a current project. In contrast, Hackystat is designed to collect data from a current, ongoing project, and use that data as feedback into the current project.
- Hackystat provides infrastructure for empirical experimentation. For those wishing to compare alternative approaches to development, or for those wishing to do longitudinal studies over time, Hackystat can provide a low-cost approach to gathering certain forms of project data.
- Hackystat is open source and is available to the academic and commercial software development community for no charge.

The design of Hackystat [6] has resulted from of prior research in our lab on software measurement, beginning with research into data quality problems with the PSP [5] and which continued with the LEAP system for lightweight, empirical, anti-measurement dysfunction, and portable software measurement [7].

To use Hackystat, the project development environment is instrumented by installing Hackystat sensors, which developers attach to the various tools such as their editor, build system, configuration management system, and so forth. Once installed, the Hackystat sensors unobtrusively monitor development activities and send process and product data to a centralized web service. If a user is working offline, sensor data is written to a local log file to be sent when connectivity can be established with the centralized web service. Project members can then log in to the web server to see the collected raw data and run analyses that integrate and abstract the raw sensor data streams into telemetry. Hackystat also allows project members to configure "alerts" that watch for specific conditions in the sensor data stream and send email when these conditions occur.

Hackystat is an open source project with sources, binaries, and documentation available at http://www.hackystat.org. There is also a public server available at http://hackystat.ics.hawaii.edu. Hackystat has been under development for approximately three years, and currently consists of around 900 classes and 60,000 lines of code. Sensors are available for a variety of tools including Eclipse, Emacs, JBuilder, Jupiter, Jira, Visual Studio, Ant, JUnit, JBlanket, CCCC, DependencyFinder, Harvest, LOCC, Office, and CVS.

3. Software Project Telemetry

A major application of Hackystat has been the development of a new approach to software measurement analysis called “Software Project Telemetry“. We define Software Project Telemetry as a style of software engineering process and product collection and analysis which satisfies the following properties:

Software project telemetry data is collected automatically by tools that unobtrusively monitor some form of state in the project development environment. In other words, the software developers are working in a “remote or inaccessable location“ from the perspective of metrics collection activities. This contrasts with software metrics data that requires human intervention or developer effort to collect, such as PSP/TSP metrics [4].

Software project telemetry data consists of a stream of time-stamped events, where the time-stamp is significant for analysis. Software project telemetry data is thus focused on evolutionary processes in development. This contrasts, for example, with Cocomo [2], where the time at which the calibration data was collected about the project is not significant.

Software project telemetry data is continuously and immediately available to both developers and managers. Telemetry data is not hidden away in some obscure database guarded by the software quality improvement group. It is easily visible to all members of the project for interpretation.

Software project telemetry exhibits graceful degradation. While complete telemetry data provides the best support for project management, the analyses should not be brittle: they should still provide value even if sensor data occasionally “drops out“ during the project. Telemetry collection and analysis should provide decision-making value even if these activities start midway through a project.

Software project telemetry is used for in-process monitoring, control, and short-term prediction. Telemetry analyses provide representations of current project state and how it is changing at the time scales of days, weeks, or months. The simultaneous display of multiple project state values and how they change over the same time periods allow opportunistic analyses—the emergent knowledge that one state variable appears to co-vary with another in the context of the current project.

Software Project Telemetry enables a more incremental, distributed, visible, and experiential approach to project decision-making. It also creates perspectives on system development that can provide new insight into HPC development processes, as we illustrate in the case study below.

4. Process and Product measures for HPC utilizing Hackystat

The development of an HPC system from a software engineering perspective raises many interesting questions. How long does such a system take to develop? Do some components take longer to develop than others? How much of the system is devoted to the sequential code, and how much is devoted to the parallelization of this code? How did the developer allocate their time during development to these activities? Do different choices of HPC tools and technologies lead to different answers to these questions? Would a different application area lead to similar or different results?

We believe that automated infrastructure for the collection and analysis of product and process data is an important first step toward enabling the HPC community to generate answers to these questions, and then use these answers to improve the tools and techniques for HPC development. The question is, what process and product measures can be both automatically collected and used to provide interesting insight into the questions raise above? This case study investigates the use of the following measures: Active Time, Most Active File, Command Line Invocations, Parallel and Serial Lines of Code, Milestone Test Success, and Performance.

The following sections describe each of these measures and illustrate them with sample data from a one week “snapshot” of development of the Optimal Truss Design problem in our case study.

4.1. The Optimal Truss Design problem

Our case study focuses on the development of a system for optimal truss design. Specifically, the system finds a pin-connected steel truss structure that uses as little mass as possible to support a load connected from three attachment points on a wall to the load-bearing point away from the wall. This problem was originally developed for use in research on Purpose-Based Benchmarks (PBBs) [3]. PBBs gather a different and complementary set of metrics in order to assess productivity in terms of acceptability to the customer.

The system is being implemented by one of the authors, Michael Paulding, and thus generally conforms to the "lone researcher" workflow for HPC development. The system development process and associated case study started in the Spring of 2004 and is still ongoing. To date, the implementation of the Optimal Truss Design problem consists of approximately 1,200 source lines of code.

The solution to the Optimal Truss Design problem developed in this case study involves several components. The first component is termed the "sequential workhorse", which includes the task of solving a truss once all of its elements are defined. Solving a truss includes the calculation of its mass, which is determined by summing the mass of each of its components (e.g. all steel joints and members). In addition to mass calculation, solving a truss also includes verifying equilibrium and deformational constraints. Equilibrium constraints require that all forces and moments within a truss net zero magnitude, thus ensuring that the truss is not accelerating. Deformational constraints require that the length of members (strut or cable) used in the truss do not exceed construction safety limits. These limits are defined and known prior to runtime.

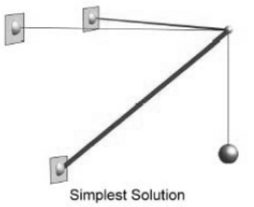


Figure 1. An unoptimized solution to the Truss problem

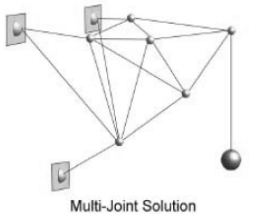


Figure 2. An optimized solution to the Truss problem

The second component of the Optimal Truss system generates the permutation of all possible truss topologies within the domain space, ensuring that the configuration with minimal mass is a global minimum. The domain space for the initial implementation is a 2-dimensional mesh of points, defining the rectangle formed between the attachment points and the load bearing point. Exploring all possible configurations results in a combinatorial explosion as the mesh size is increased and this served as the first point of parallelism in the implementation. The task of parallelizing topology generation can be equally divided among the available nodes. This can be accomplished in an "embarrassingly parallel" manner, where each row of the mesh is assigned to a different processor to permute.

The third component of the system performs geometry assignment for all trusses. After generation, a topology defines the path of each truss from the attachment points to the load bearing point, but it does not specify what type of member connects each joint. In this stage, either a strut or a cable is substituted for each member, flushing out all permutations. Once the geometry has been assigned, it can be given to a processor to compute the mass of the truss and to determine whether the topology is valid under the equilibrium and deformational constraints.

Now that the description of the Optimal Truss Design problem, used in our case study, has been explained, it is prudent to illustrate and investigate the measures applied to the problem.

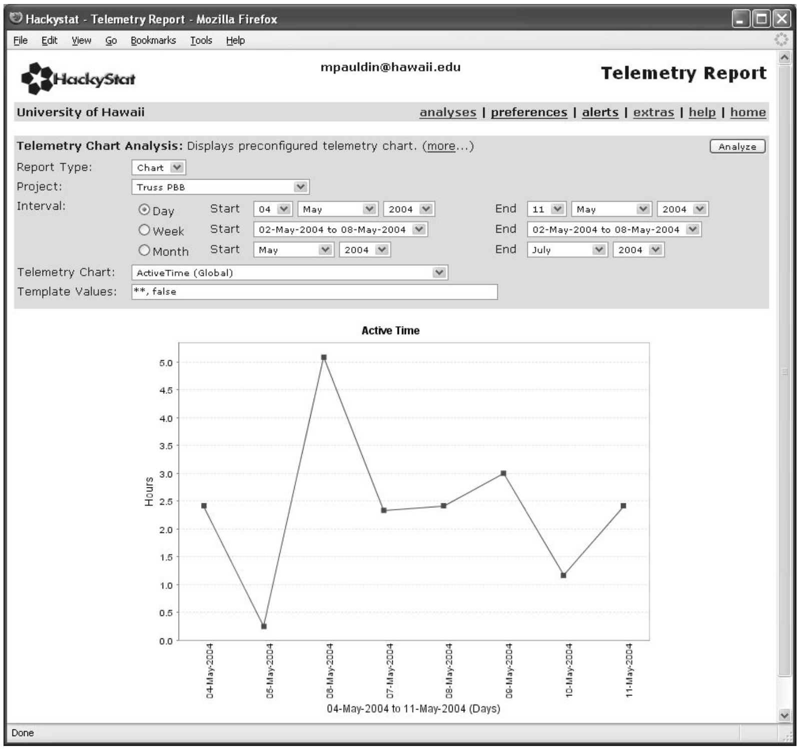
4.2. Active Time

Active Time is a measure of the time spent by developers editing source code (or other files) related to the system. Active Time can be collected automatically through the use of sensors attached to the editor used by developers. The sensors collect active time via a timer-based process inside the editor that wakes up every 30 seconds and checks to see if the active buffer has changed in identity or size since the last 30 seconds. If so, a timestamped "statechange" event is sent to the Hackystat server. Active Time does not reflect effort spent by developers on the project that does not involve editing files, including time spent viewing a file without performing editing actions. Support for non-editing activities such as "reading" is a subject of future research, but even the restricted definition of Active Time appears useful in the HPC context as a proxy for overall effort. For example, it helps a development team answer questions such as: *"How much of the overall development effort was spent editing files?"* or *"Did all team members devote equal time to writing code?"* or *"When was team effort focussed on code development during the project?"*

Figure 3 shows the Active Time associated with development of the Optimal Truss Design application for a sample period in May, 2004.

4.3. Most Active File

A measure related to Active Time is the "Most Active File". One way to abstract the raw event stream sent from an



Hackystat - Telemetry Report - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
analyses | preferences | alerts | extras | help | home
Analyze
Done

mpauldin@hawaii.edu

Telemetry Report

University of Hawaii

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

Project: Truss PBB

Interval:
● Day Start 04 May 2004 End 11 May 2004
○ Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004
○ Month Start May 2004 End July 2004

Telemetry Chart: ActiveTime (Global)

Template Values: **, false

Figure 3. Active time

editor-based Hackystat server begins by representing each day as a sequence of 288 five minute intervals. If a developer actively edits one or more files within a five minute period, then determine which file was edited most during that five minutes, and assign the "credit" for that five minute interval to that file and that file alone, which we call the "Most Active File". (We performed a calibration study which found this to be a reasonable abstraction.) The Most Active File abstraction may be useful in the HPC context as a way of determining what specific files were the focus of developer attention, and how that focus of attention changed over the course of development.

For example, Figure 4 shows the Most Active Files associated with Optimal Truss Design during the first few days of this time interval.

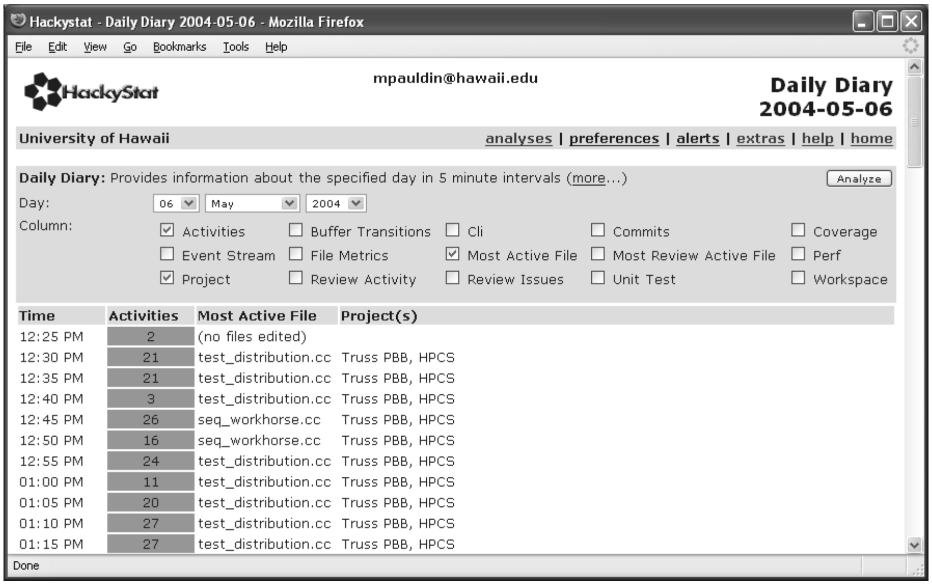
4.4. Command Line Invocations

In addition to time spent editing files in an editor, HPC development frequently involves extensive use of shell processes to invoke programs such as make, gcc, etc. We have implemented a sensor for the Unix command shell (based upon the 'history' shell mechanism) to record these command line invocations. Command Line Invocation data can be useful in the HPC context as a way of providing further insight into the types of activities performed by developers during the development of the HPC code. For example, if the HPC developer spends significant time working at the command line without concurrent editing of code, then it might be useful to develop an enhanced representation of Active Time that accounts for this type of effort as well. While the current sensor only captures command invocations and not their results, it might be useful to extend the sensor to capture the results of command line invocations in certain circumstances. For example, recording whether or not a compilation succeeded or failed as well as what types of run-time errors occur could help identify potential development bottlenecks.

Figure 5 illustrates Command Line Invocation data for a portion of one day during the development of the Optimal Truss Design system.

4.5. Parallel and Serial Size

To understand HPC software development, it helps to be able to represent both "serial" and "parallel" code. We have enhanced our size measurement tool, LOCC, with a token-based counter for C++ that allows us to count non-comment source lines of code, and determine for each line of code whether or not an MPI directive occurs on it. Thus, for HPC programs built using C++ and MPI, we can determine (a) the total number of files in the system, (b) the total non-


Hackystat - Daily Diary 2004-05-06 - Mozilla Firefox
File
Edit
View
Go
Bookmarks
Tools
Help
HackyStat
analyses | preferences | alerts | extras | help | home
Analyze
Done

mpauldin@hawaii.edu

Daily Diary 2004-05-06

University of Hawaii

Daily Diary: Provides information about the specified day in 5 minute intervals (more...)

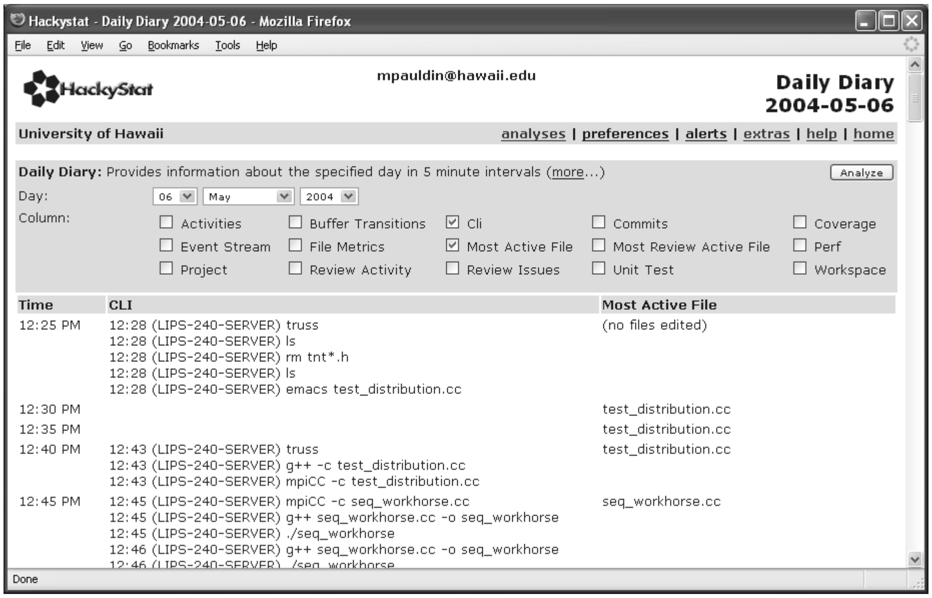
Day: 06 May 2004

Column:

☑ Activities ☐ Buffer Transitions ☐ Cli ☐ Commits ☐ Coverage
☐ Event Stream ☐ File Metrics ☑ Most Active File ☐ Most Review Active File ☐ Perf
☑ Project ☐ Review Activity ☐ Review Issues ☐ Unit Test ☐ Workspace

Time	Activities	Most Active File	Project(s)
12:25 PM	2	(no files edited)	
12:30 PM	21	test_distribution.cc	Truss PBB, HPCS
12:35 PM	21	test_distribution.cc	Truss PBB, HPCS
12:40 PM	3	test_distribution.cc	Truss PBB, HPCS
12:45 PM	26	seq_workhorse.cc	Truss PBB, HPCS
12:50 PM	16	seq_workhorse.cc	Truss PBB, HPCS
12:55 PM	24	test_distribution.cc	Truss PBB, HPCS
01:00 PM	11	test_distribution.cc	Truss PBB, HPCS
01:05 PM	20	test_distribution.cc	Truss PBB, HPCS
01:10 PM	27	test_distribution.cc	Truss PBB, HPCS
01:15 PM	27	test_distribution.cc	Truss PBB, HPCS

Figure 4. Most Active File


Hackystat - Daily Diary 2004-05-06 - Mozilla Firefox
File
Edit
View
Go
Bookmarks
Tools
Help
HackyStat
analyses | preferences | alerts | extras | help | home
Analyze
Done

mpauldin@hawaii.edu

Daily Diary 2004-05-06

University of Hawaii

Daily Diary: Provides information about the specified day in 5 minute intervals (more...)

Day: 06 May 2004

Column:

☐ Activities ☐ Buffer Transitions ☑ Cli ☐ Commits ☐ Coverage
☐ Event Stream ☐ File Metrics ☑ Most Active File ☐ Most Review Active File ☐ Perf
☐ Project ☐ Review Activity ☐ Review Issues ☐ Unit Test ☐ Workspace

Time	CLI	Most Active File
12:25 PM	12:28 (LIPS-240-SERVER) truss 12:28 (LIPS-240-SERVER) ls 12:28 (LIPS-240-SERVER) rm tnt*.h 12:28 (LIPS-240-SERVER) ls 12:28 (LIPS-240-SERVER) emacs test_distribution.cc	(no files edited)
12:30 PM		test_distribution.cc
12:35 PM		test_distribution.cc
12:40 PM	12:43 (LIPS-240-SERVER) truss 12:43 (LIPS-240-SERVER) g++ -c test_distribution.cc 12:43 (LIPS-240-SERVER) mpiCC -c test_distribution.cc	test_distribution.cc
12:45 PM	12:45 (LIPS-240-SERVER) mpiCC -c seq_workhorse.cc 12:45 (LIPS-240-SERVER) g++ seq_workhorse.cc -o seq_workhorse 12:45 (LIPS-240-SERVER) ./seq_workhorse 12:46 (LIPS-240-SERVER) g++ seq_workhorse.cc -o seq_workhorse 12:46 (LIPS-240-SERVER) ./seq_workhorse	seq_workhorse.cc

Figure 5. Command Line Invocations

commented size of each file in the system; (c) whether or not a file consists purely of serial (non-MPI) code or not; (d) for files containing MPI directives, the frequency of occurrence of each MPI directive; and (e) for files containing MPI code, what percentage of the non-comment source lines of code contained an MPI directive.

Figures 6, 7, and 8 provide perspectives on size data for the Optimal Truss Design system.

4.6. Functionality

We have defined a process for measuring the functionality of an HPC application and tracking its development progress through the use of unit testing. We have termed this approach "Progress Assessment through Milestone Tests" (PAMT).

Essentially, PAMT is a process in which HPC application designers draft the specification for the system as a set of unit tests prior to development. Each unit test is defined such that it represents a milestone, or significant aspect of application functionality. Quantitative interpretation of "significant" is determined by the application designer or program manager and is expected to vary between HPC projects. Defining the set of milestone tests prior to development provides a specification for the system and also serves as a mechanism to promote test driven design.

Once the milestone tests have been defined, the development team has a concrete set of tests to implement that, together, represent the functionality of the entire system. The development team can then implement the milestone tests in any order and their progress through the application can be monitored. System progress and functionality is measured by investigating the number of milestone tests passing in ratio to the total number of milestone tests representing the system. In most cases, a development team will begin implementation with zero milestone tests passing and finish development when all milestone tests pass.

For the Optimal Truss Design problem, a set of 10 milestone tests were defined prior to implementation. Individually, each test represents a significant functionality of the application and together they provide a specification for the entire system. For the Optimal Truss problem, the milestone tests were written in CppUnit, a unit test framework for the C++ programming language. Below is an example of a single milestone test for the Optimal Truss problem.

> **Milestone Test 4:** This test verifies that the application is capable of representing a 2-dimensional topology. In the Optimal Truss specification, a topology is defined as a set of 2 trusses that individually connect the 2 attachment points to the load bearing point. Interconnections (members) between the trusses are allowed. Therefore, for this milestone test, given 2 attachment points, a load bearing point and the number of joints, the application must be able to query:
>
> 1. Each of the trusses connecting the 2 attachment points to the load bearing point
> 2. The set of members composing one of the trusses in the topology
> 3. Given a truss, whether it is part of the topology

From this chart is is evident that during the development period from 04-May-2004 through 11-May-2004 that the Optimal Truss application progressed from 1 milestone test passing at the beginning of the interval to 5 milestone tests passing at the end. It is important to note that this interval represents a sample of the development period and does not capture start to finish. In addition, this trend indicates a consistent increase in passing milestone tests. However, it is quite possible for development to lower the number of successful milestone tests, indicated by a negative slope in the trend.

4.7. Performance

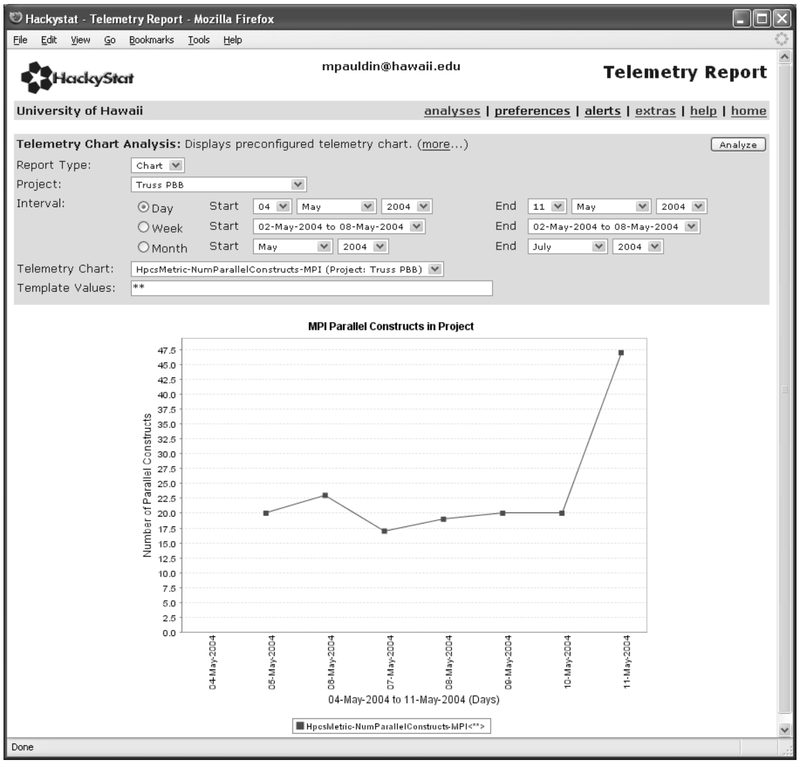
The high performance computing community has developed a broad range of standard measures to characterize parallel performance, including degree of parallelism, average parallelism, speedup, redundancy, and utilization. In this research, we are not attempting to specify the "right" performance measure for any particular application area. Instead, we advocate that performance be measured regularly throughout development using as many metrics as necessary to best characterize the application.

Performance measures are not generally interesting as absolute numbers, since the absolute values are obviously dependent upon current hardware and other physical resources. Performance measures are interesting as relative numbers, in the sense that the way they change over time tells us whether or not and to what extent developers could tune an initial implementation to improve its performance, how the code evolved to obtain this performance increase, and whether or not functionality was sacrificed in order to do so.

Figure 10 shows the execution (wall time) performance of the Optimal Truss Design system developed in the case study for a sample time interval.

5. Conclusions

After accumulating the data trends provided by the Hackystat system, we are able to gain insights that assist in understanding the development of HPC applications. From


Hackystat - Telemetry Report - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
analyses | preferences | alerts | extras | help | home
Analyze
Done

mpauldin@hawaii.edu

Telemetry Report

University of Hawaii

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

Project: Truss PBB

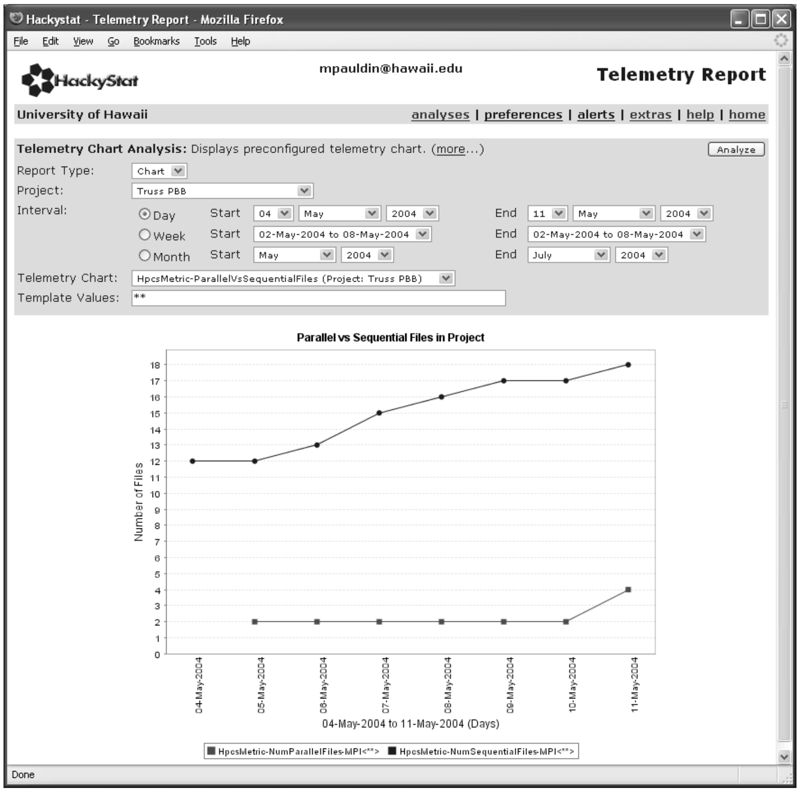
Interval:

- ● Day Start 04 May 2004 End 11 May 2004
- ○ Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004
- ○ Month Start May 2004 End July 2004

Telemetry Chart: HpcsMetric-NumParallelConstructs-MPI (Project: Truss PBB)

Template Values: **

Figure 6. Parallel vs. sequential constructs


Hackystat - Telemetry Report - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
analyses | preferences | alerts | extras | help | home
Analyze
Done

mpauldin@hawaii.edu

Telemetry Report

University of Hawaii

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

Project: Truss PBB

Interval:

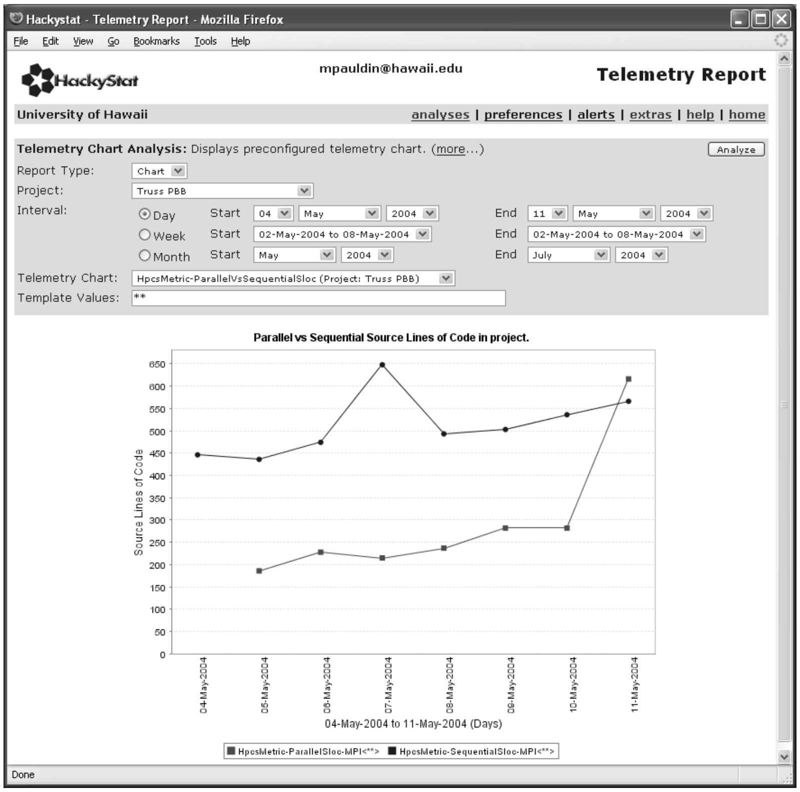
- ● Day Start 04 May 2004 End 11 May 2004
- ○ Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004
- ○ Month Start May 2004 End July 2004

Telemetry Chart: HpcsMetric-ParallelVsSequentialFiles (Project: Truss PBB)

Template Values: **

Parallel vs Sequential Files in Project
Number of Files
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
04-May-2004
05-May-2004
06-May-2004
07-May-2004
08-May-2004
09-May-2004
10-May-2004
11-May-2004
04-May-2004 to 11-May-2004 (Days)
■ HpcsMetric-NumParallelFiles-MPI<**>
■ HpcsMetric-NumSequentialFiles-MPI<**>

Figure 7. Parallel vs. Sequential Files


Hackystat - Telemetry Report - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
analyses | preferences | alerts | extras | help | home
Analyze
Parallel vs Sequential Source Lines of Code in project.
Source Lines of Code
650
600
550
500
450
400
350
300
250
200
150
100
50
0
04-May-2004
05-May-2004
06-May-2004
07-May-2004
08-May-2004
09-May-2004
10-May-2004
11-May-2004
04-May-2004 to 11-May-2004 (Days)
■ HpcsMetric-ParallelSloc-MPI<**> ■ HpcsMetric-SequentialSloc-MPI<**>
Done

mpauldin@hawaii.edu

Telemetry Report

University of Hawaii

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

Project: Truss PBB

Interval: ● Day Start 04 May 2004 End 11 May 2004

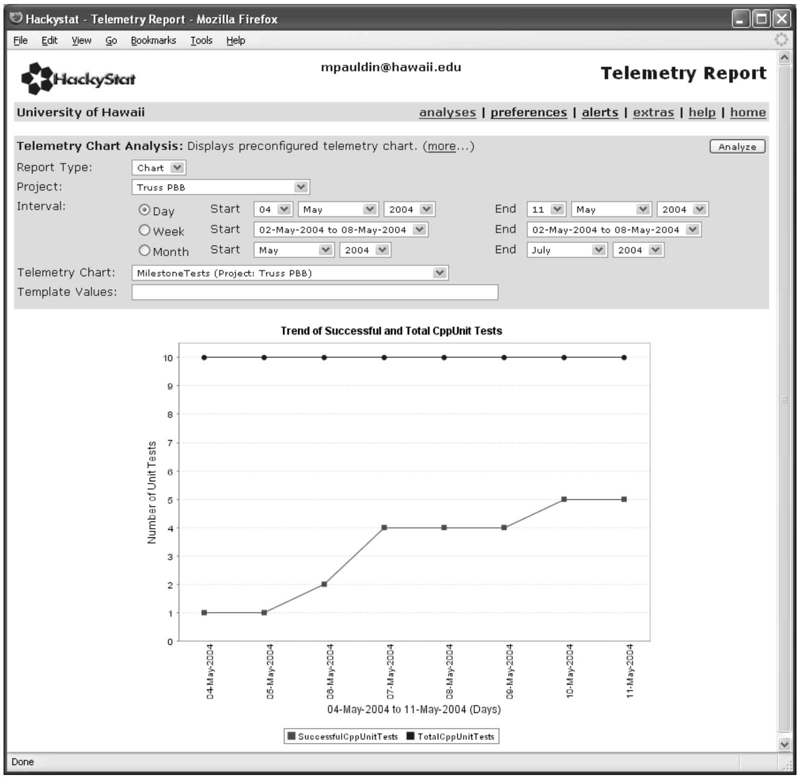
○ Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004

○ Month Start May 2004 End July 2004

Telemetry Chart: HpcsMetric-ParallelVsSequentialSloc (Project: Truss PBB)

Template Values: **

Figure 8. Parallel vs. Sequential SLOC


Hackystat - Telemetry Report - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
analyses | preferences | alerts | extras | help | home
Analyze
Trend of Successful and Total CppUnit Tests
Number of Unit Tests
10
9
8
7
6
5
4
3
2
1
0
04-May-2004
05-May-2004
06-May-2004
07-May-2004
08-May-2004
09-May-2004
10-May-2004
11-May-2004
04-May-2004 to 11-May-2004 (Days)
■ SuccessfulCppUnitTests ■ TotalCppUnitTests
Done

mpauldin@hawaii.edu

Telemetry Report

University of Hawaii

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

Project: Truss PBB

Interval: ● Day Start 04 May 2004 End 11 May 2004

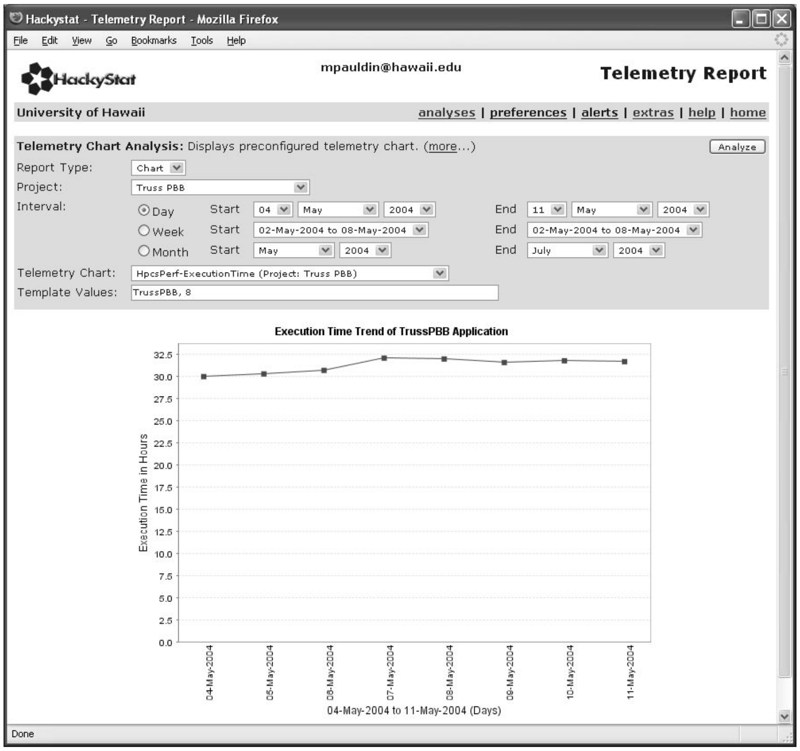
○ Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004

○ Month Start May 2004 End July 2004

Telemetry Chart: MilestoneTests (Project: Truss PBB)

Template Values:

Figure 9. Progress Assessment through Milestone Tests



Hackystat - Telemetry Report - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
HackyStat
analyses | preferences | alerts | extras | help | home
Analyze
Execution Time Trend of TrussPBB Application
32.5
30.0
27.5
25.0
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Execution Time in Hours
04-May-2004
05-May-2004
06-May-2004
07-May-2004
08-May-2004
09-May-2004
10-May-2004
11-May-2004
04-May-2004 to 11-May-2004 (Days)
Done

mpauldin@hawaii.edu

Telemetry Report

University of Hawaii

Telemetry Chart Analysis: Displays preconfigured telemetry chart. (more...)

Report Type: Chart

Project: Truss PBB

Interval:
● Day Start 04 May 2004 End 11 May 2004
○ Week Start 02-May-2004 to 08-May-2004 End 02-May-2004 to 08-May-2004
○ Month Start May 2004 End July 2004

Telemetry Chart: HpcsPerf-ExecutionTime (Project: Truss PBB)

Template Values: TrussPBB, 8

Figure 10. Truss Execution Time Performance

the graphs presented earlier, we are able to make interpretations about development activities, development progress and application performance and functionality tradeoffs.

5.1. Development Activities

From the data presented in Figure 5 we are able to interpret the developer activities of that particular day. The daily dairy for 06-May-2004 lists all the commands issued to the console on that day. Figure 5 is a sample of all the command line invocations issued, but it provides insight to the developer activities on that day.

For example, from the command line invocations and most active file data for 06-May-2004, we can observe that the developer devoted the most time on the *test_distribution.cc* file. It so happens that this file implements the distribution of 2-dimensional mesh from which the truss topologies are constructed. Furthermore, the distribution of topologies is a significant function of the Optimal Truss problem and has been designated as a milestone test of the system. Therefore, from this data, an observer can conclude that the developer was investing his efforts on implementing functionality on this day, rather than on increasing performance by optimizing code.

In addition, an observer, such as a project manager or the developer himself, can observe the active time trend in Figure 3 to understand the time invested to implement a particular milestone. For example, in Figure 3 on 06-May-2004, it is evident that the developer spent over 5 hours editing code to implement the topology distribution milestone.

5.2. Development Progress

The data presented in Progress Assessment through Milestone Tests (PAMT) chart, as illustrated in Figure 9, provides a clear illustration of the real-time progress being made on the Optimal Truss problem.

There are two trends presented in this figure, one representing the total number of milestone tests defined for the Optimal Truss problem and the other representing the number of milestone tests passing at the conclusion of each day.

In the Optimal Truss problem, the total milestone tests are represented by the horizontal line fixed at 10 unit tests. This indicates that there are 10 milestone tests encompassing the Optimal Truss problem and that the project manager has not added or removed any of these milestones during this time interval. It is quite possible that a project manager may have to alter milestones in order to meet deadlines and this analysis provides a trend for this purpose. For example, if the total milestone tests is altered, the total tests trend on the chart will move up or down accordingly.

The PAMT chart also allows an observer to track the progress made through the application. For example, in this figure, the lower trend represents the number of milestone tests passing on each day. Every time a new milestone test passes, it indicates that another unit of functionality has been added to the system provided that all previously passing tests still pass after the change.

Coupling the PAMT data with the active time data in Figure 3, an observer is also able to interpret a quantitative measurement of how much development time was devoted to a particular unit of functionality. For example, on 06-May-2004, approximately 5 hours of editing were invested to add one unit of functionality to the application. This is indicated by the number of passing milestone tests increasing from one to two on this day. In addition, an observer can quickly understand the percentage complete of the system. On 06-May-2004, the Optimal Truss problem has 2 out of 10 milestone tests passing and is therefore 20% complete.

5.3. Application Performance and Functionality Tradeoffs

When one combines the data presented in the Performance chart in Figure 10 with the PAMT Functionality chart in Figure 9, it reveals an example of the interactions between performance and functionality in HPC development.

One of the primary objectives of HPC development is to obtain the fastest possible execution time of the system. This goal influences developers to frequently (if not constantly) think about or perform optimization on their code.

However, as functionality is added to the application, it is common for the performance of the system to decrease, indicated by an increase in execution time.

The data presented in figures 10 and 9 reveal this performance and functionality tradeoff. For example, execution time between 04-May-2004 and 07-May-2004 increaseses roughly from 30.0 hours to 32.5 hours. On the other hand, 3 additional milestone tests, representing system functionality, are implemented successfully during this interval. This indicates that three units of functionality have been added at the cost of an addition of approximately 10% in execution time.

Data presented in these figures allow project managers to understand how functionality increases affect system performance. It also gives them a starting point to determine which functionality should be optimized in the case where the performance degradation is not acceptable. Trends such as these enable project managers and developers to understand the development process and make in-process decisions to affect the development outcome.

In conclusion, we have found that Active Time, Most Active File, Command Line Invocations, Parallel and Serial Lines of Code, Milestone Test Success, and Performance constitute an interesting set of process and product measures that can be automatically collected during HPC development. As our case study continues, we will look for other opportunities to use this measures to gain insight into opportunities to improve high performance computing.

References

[1] The DARPA high productivity computing systems program. http://www.highproductivity.org/.

[2] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[3] J. Gustafson. Purpose Based Benchmarks. *International Journal of High Performance Computing Applications*, 12(1):14, 2004.

[4] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, New York, 1995.

[5] P. M. Johnson and A. M. Disney. The personal software process: A cautionary case study. *IEEE Software*, 15(6), November 1998.

[6] P. M. Johnson, H. Kou, J. M. Agustin, C. Chan, C. A. Moore, J. Miglani, S. Zhen, and W. E. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon, May 2003.

[7] P. M. Johnson, C. A. Moore, J. A. Dane, and R. S. Brewer. Empirically guided software effort guesstimation. *IEEE Software*, 17(6), December 2000.

[8] D. A. Reed, editor. *The Roadmap for the Revitalization of High-End Computing*. Computing Research Association, 2003.