ABSTRACT

Title of Document: INVESTIGATING THE EFFECTS OF HPC NOVICE PROGRAMMER VARIATIONS ON CODE PERFORMANCE.

Rola Alameh
Master of Science, 2007

Directed By: Professor Victor R. Basili
Department of Computer Science

In this thesis, we quantitatively study the effect of High Performance Computing (HPC) novice programmer variations in effort on the performance of the code produced. We look at effort variations from three different perspectives: total effort spent, daily distribution of effort, and the distribution of effort over coding and debugging activities. The relationships are studied in the context of classroom studies. A qualitative study of both effort and performance of students was necessary in order to distinguish regular patterns and define metrics suitable for the student environment and goals. Our results suggest that total effort does not correlate with performance, and that effort spent coding does not count more than effort spent debugging towards performance. In addition, we were successful in identifying a daily distribution pattern of effort which correlates with performance, suggesting that subjects who distribute their workload uniformly across days, pace themselves, and minimize interruptions achieve better performance.

INVESTIGATING THE EFFECTS OF NOVICE HPC PROGRAMMER VARIATIONS ON CODE PERFORMANCE.

By

Rola Alameh.

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2007

Advisory Committee:
Professor Victor R. Basili, Chair
Professor Jeffrey K. Hollingsworth
Professor Marvin Zelkowitz

Dedication

I dedicate this thesis to the strongest women I know, my mother and my sister.

To mom: although you were a single mother with three kids, you never gave up. You sacrificed so much, and worked so hard to provide for us. You instilled in me my self-confidence and a sense of responsibility, taught me the value of education, and encouraged me to follow my beliefs and values, no matter what people think. I hope some day I can repay you for all you have done for me and for the entire family.

To my sister: I can't help but smile when I think about you. You are my best friend and I have so much fun hanging out with you and talking to you. I've always looked up to you: if you hadn't gotten your bachelor's degree from AUB, I would've never thought it was possible for me to get mine there as well (and in the same major!). If you hadn't gotten your Ph.D. from MIT, I would've never have thought about getting a master's degree, nor about continuing my graduate studies outside of Lebanon. Even my English wouldn't be this good if it weren't for you. I arrived to the University of Maryland almost a week after the semester had started and I didn't even know who my academic advisor was, what courses were offered… Nevertheless, it seemed to everyone that I hit the ground running. I couldn't have done it, if you hadn't been there, running by my side.

Mom, Nadine, you are truly my inspiration and I thank you both for helping me become the person that I am.

Acknowledgements

First I would like to thank the professors who allowed us to run studies in their classes, especially John Gilbert and Alan Edelman. I would also like to thank all the students who participated in the studies. This work wouldn't have been possible without the participation of so many people.

I'm most grateful to my advisor, Vic Basili, for his support and guidance throughout this study and my stay at UMD. He has been a great inspiration and I have gained a lot from his expertise and enthusiasm. I also thank the other members of my thesis committee: Jeff Hollingsworth for closely following my work and giving me comments and suggestions, and Marv Zelkowitz for his valuable feedback.

Thanks go to the students from the University of Mannheim who worked with me: Steffen Olbrich and Nabi Zamani. My work would have been much harder if it hadn't been for the work they did and the tools they developed for the HPCS team.

I would like to thank current and former members of the HPCS team, especially Lorin Hochstein, Taiga Nakamura and Nico Zazworka, who had joined the team as students before me and who shared their knowledge with me and helped me get up to speed on the team's activities. Special thanks also go to Ananta Tiwari from the University of Maryland who gave much of his time to get me started with parallel programming and help me out when I faced problems.

My fiancé, Waleed, might be happier than I am that I finished. We both have been waiting for this so we can finally be together. I can never thank him enough for

his love, patience and understanding. I'm very lucky to have someone willing to wait for me for two years, a period during which he accommodated our long distance relationship by staying up late every night waiting for me to finish my classes and study work, only to talk to me and see me through Skype.

Finally, I thank my family for everything they have done for me: my brother-in-law, Hisham, for encouraging me to get a master's degree and for rushing the International Education Services to send me the necessary paperwork to get my visa; my brother, Rani, who inspires me to go on and never give up and who nourishes my mind every week with political debates, graphic design basics and lessons in literature, arts and sociology; my sister, Nadine (aka Nano), for being my best friend, role model and my personal orientation counselor, as she helped me prepare for the TOEFL, SAT and GRE (that was fun ☺), filled out college applications with me and edited my resume; and last, but certainly not least, my mother, Nada, for all the sacrifices she made, her unconditional love and support and the delicious food she made during my last semester, which spared me the agony of choosing what and where to eat every day

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Overview

Many fields of scientific research which aim at studying physical phenomena through computer simulation, such as computational science and engineering, have entered the mainstream of High Performance Computing (HPC). Single processor computers are not capable of efficiently running interesting problems in these fields because they involve processing large amounts of data, and performing complex calculations, requiring an impractical amount of time to complete. Therefore, scientists must write software to run on the more powerful, parallel machines.

However, while computer performance can be dramatically improved, programming these machines remains difficult. Expert programmers in the HPC field are rare because HPC code development requires knowledge in both the HPC domain and the scientific domain. So scientists find it increasingly costly and time consuming to write, port, execute and evaluate software in this domain. Indeed, the consensus in the community is that today's technologies are inadequate to HPC users. For this reason, productivity has become as important as performance in the HPC domain.

For example, the DARPA High Productivity Computing Systems (HPCS) project has been concerned with the question: How do HPC hardware, software and human factors affect the development of an HPC program? It has focused on the issues of low efficiency, scalability, and software tools and environments, in order to

provide a petaflop-class computer that is substantially easier to program and use. Its view of productivity can be summarized by the following equation:

Time to solution = development time + execution time

In this thesis, we are trying to understand the effect of novice HPC programmer variations, such as programmer effort, on performance. Effort and performance can obviously be mapped to development time and execution time respectively. Therefore, if the hypothesized relationship exists and is strong enough, minimizing the time to solution will depend on a balance - probably a delicate balance - between development time and execution time. This balance would therefore be a potential indicator of success in HPC and a key factor in improving productivity.

1.2 Objectives

As mentioned earlier, many HPC developers are scientists who are experts in their application domain, but have limited knowledge in the HPC domain. For this reason, and given the available data we have from participating universities, we focus our study's context on senior and graduate students taking a graduate class on HPC, as an approximation to the average HPC developer.

Our objective is to characterize, in the above mentioned context, the relationship between the performance of the written HPC code and the following variables:

- total effort spent by the programmer working on the problem,

- distribution of the programmer's effort over time,
- distribution of the programmer's effort over debugging and coding activities,
- and the size of the written code.

The results of the study are of interest to many stakeholders and can be useful in many perspectives:

- Vendor perspective: the process of finding and fixing defects ("debugging") in the code is costly and time-consuming. We, therefore, expect that time spent debugging doesn't contribute work towards performance as effectively as time spent coding new functionality. Our results can then be used to give advice to vendors on the tools needed to improve programmer's productivity.
- Programmer perspective: our results concerning the effects of different patterns of effort distribution over time can help the programmer decide how and where to spend their time. We compare two main effort distribution patterns, which we found through our qualitative study of the subjects as will be discussed in section 3.2:
    - The slow and steady approach: the programmer works regularly on the problem, spending almost equal periods of time every day
    - The fast and furious approach: the programmer works intensely on the problem for one or more days, then abandons it for some time, and goes back to it later.

- Management perspective: one of the differences between the two patterns above is the significant break in the middle of development. This can happen for many reasons, one of which is the obligation to work concurrently on a different problem. Our study can help the task assigner or manager distribute the load of different projects on the programmers, with the knowledge that giving them many projects to work on might result in the interleaved pattern of work, and the effect that has on their productivity.

- Educational perspective: the managerial perspective can also be applied on a smaller scale in classrooms, where professors or instructors of HPC related courses can direct students to the most efficient work strategy for their assignments and projects.

1.3 Organization of thesis

The thesis is organized as follows: Chapter 2 describes related work and provides information on tools previously developed for the HPCS studies and that were used in this work. Chapter 3 provides the research methodology and hypothesis discovery process. Chapter 4 lays out the data results. Chapter 5 provides the interpretation and observations drawn from the results. Chapter 6 concludes the thesis with a summary of conclusions and future work.

Chapter 2: Related work

2.1 Related literature

Over the years, many programming models and languages have been developed for writing HPC applications. Dongarra et. al. [1] presents some guidelines for choosing particular programming models to use. Hochstein [2] also summarizes programming models and languages that are available for writing HPC applications today.

As computer architectures and programming models advance and increase in complexity, software development is getting harder. However, empirically understanding how practitioners carry out the software process is challenging due to lack of an existing body of domain-specific knowledge and difficulty in getting subjects and projects to participate in such studies. Hochstein [2] provides an empirical approach for iteratively and incrementally collecting useful knowledge about a particular software domain into an experience base. He applies his methodology to the HPC domain, and uncovers the following facts:

- “Students are able to get reasonable performance, despite essentially no performance tuning.”

- "Problem size is not necessarily related to the difficulty in achieving speedup" since students got better performance in their class projects than in their assignments.

- "Experts achieve better performance than novices, but not necessarily with less effort"

- There exists a "measurable reduction in effort with experience with having solved that problem before in a different model".

- The difference between MPI and OpenMP defects isn't significant, contradicting the hypothesis that "shared memory programs are harder to debug".

In a more extensive study of HPC defects, Nakamura [3] presents another empirical approach for building, storing and evolving domain-specific knowledge, applied in turn to HPC defects. In the process, he identified defect patterns from empirical data, developed a corresponding classification scheme and constructed HPCBugBase[1], which enables these patterns to be stored and shared.

Of the few studies into programmer productivity in HPC is the survey work of Pancake [4], which discusses the conceptual gap between tool builders and tool users in HPC and indicates where the developers should focus their efforts in order to attract the parallel users. Pancake also answers questions concerning the allocation of

[1] http://www.hpcbugbase.org

effort in developing parallel applications and preferences in tool origin based on the programmers' different backgrounds (Computer Science, Engineering and Science).

The relationships between some HPC domain parameters and variables have been studied in specific situations. There have been empirical studies to quantify the effect of parallel programming technologies on effort. Szafron & Schaeffer [5] compared the Enterprise Parallel Programming System (PPS) to a message-passing library through a controlled experiment. They based their comparison on usability, which for PPS was determined by features such as the learning curve, programming errors, performance, compatibility with existing software and integration with other tools. According to their definition of usability, they found that Enterprise has greater usability features but NMP produces better performance. Enterprise required considerably fewer lines of code than NMP and saved programmers time during coding, so they could focus on the design of the parallel solution without unnecessarily distracting details. Hochstein [6] also compared two programming models, OpenMP and MPI, with respect to their impact on development effort. He concluded that novice programmers spend significantly less effort with OpenMP compared to MPI.

The effect of the programming models and languages on the performance has also been studied, as in the work of Berlin et. al. [7], which evaluated the impact of a number of parallel programming languages (MPI, UPC, OpenMP, Java, C/Pthreads) and their features for their performance and ease of use. Another example is the experimental comparison of OpenMP, HPF and MPI conducted by Berthou et. al. [8] on various platforms according to many criteria including efficiency, scalability, and

portability. As a conclusion, the authors recommended different parallelization strategies for different codes.

The tradeoffs between performance gains and effort were described by Morton [9] in the context of porting Fortran/PVM code to the Cray T3D. One of the lessons reported in this work is that programmers must spend extra work to be able to take advantage of the low-level facilities offered by the Cray T3D for attaining substantial performance gains. He also notes that encapsulating optimization details and incorporating them in applications through high-level routines can reduce coding effort and allow researchers to focus on their science objectives instead of spending more time on the details of low-level programming.

Rodman and Brorsson [10] investigated the programming effort involved in integrating message passing into a shared memory program in order to preserve the easy implementation of the shared memory approach and take advantage of the benefits of message-passing for performance critical tasks.

2.2 Related tools

2.2.1 UMDInst

The data used in this study was collected as part of the HPCS project from students in different universities in the United States. The details of the data used will be discussed in the methodology section. In this section, we will give an overview of the tools used to collect this data.

It is important that the tools used provide an objective way of collecting data and conserving its accuracy. We therefore used automatic data collection from the programmer's environment, which in addition reduces the overhead on subject side and the researcher side. This is supported by our instrumentation package, UMDInst which automatically collects software process data in a Unix-based, command-line development environment, which is commonly used in HPC.

UMDInst [11] is a collection of scripts that wrap programs and tools involved in the software development process (compiler, editor…). When one of these programs is invoked, the wrapper logs the useful information (such as timestamp, username, command used …) before passing execution to the intended program. The logged information differs depending on the program that was invoked. For example, a compiler invocation captures the following data:

- a timestamp
- contents of the source file that were compiled
- the command used to invoke the compiler
- the return code of the compiler
- the time to compile

The UMDInst package includes Hackystat sensors [12] to instrument supported editors such Emacs and vi, and to capture shell commands and timestamps. Whenever subjects are editing, compiling or running their codes, the captured

snapshots of their programs and commands are written to a local log file. At the end of the study this log file contains all intermediate versions of the whole development process of HPC programs. It is then possible to use the collected data to estimate total effort as well as to infer development activities (debugging, parallelizing, tuning), as will be described in the methodology section.

2.2.2 CodeVizard

As will be described in subsection 3.1.2, for the purpose of distinction between work and rework phases, analysis of a single version of code is not sufficient. We need to analyze the progress of versions over time, which would be extremely challenging to do manually. We therefore need a tool that supports this kind of evolutionary investigation of a software process. Our task is further complicated by our need for implicit data, data that is not directly accessible but implicitly contained in the code. As will be discussed later, the number of lines of code added, deleted, modified and their locations are essential in our approach to determine the phase in question.

For these purposes, we used CodeVizard [13], which is a visual diff tool designed specifically for the needs of HPCS researchers to visualize software evolutionary data. CodeVizard displays all versions of the source code lined up with transitions describing the amount and location of added, deleted and modified lines of code. The strength of CodeVizard is that it allows the user to control the level of abstraction and the level of detail at which to show the data. The abstract view allows the analyst to quickly capture interesting patterns. However, this high level view is

not always sufficient to draw conclusions. Therefore, the interactive features of CodeVizard, which allow the analyst to view the code at a lower level of detail, are very useful. Its visual and semantic zooming capabilities let the user decide on and gradually modify the level of detail needed whenever needed. Figure 1, Figure 2 and Figure 3 present three different views that can be obtained of CodeVizard. Green, red and blue lines represent added, deleted and modified lines respectively. One of the most important features of CodeVizard is that the user is in total control of the zooming-level, and that zooming can be increased and decreased in small continuous increments.

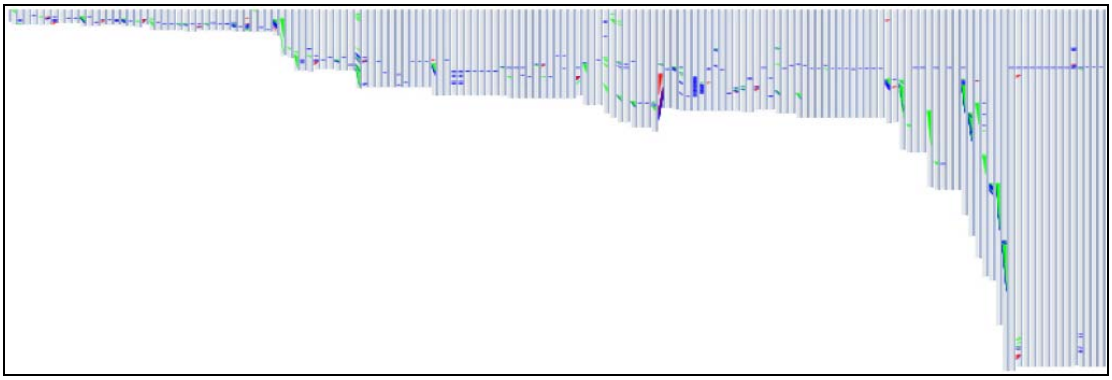

Figure 1: High-level view of a subject's code throughout development [13]

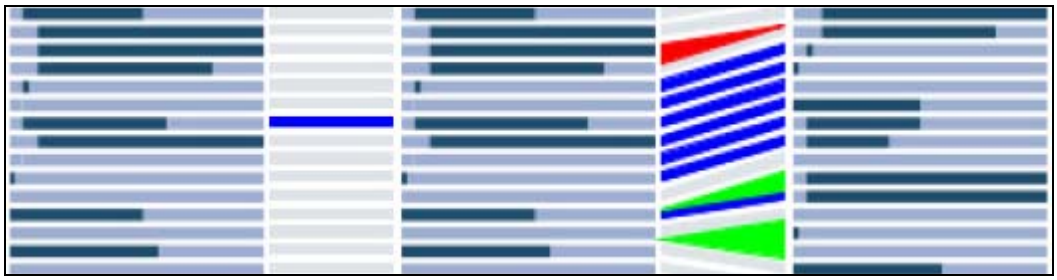

Figure 2: Medium-level zoom view of a series of versions [13]

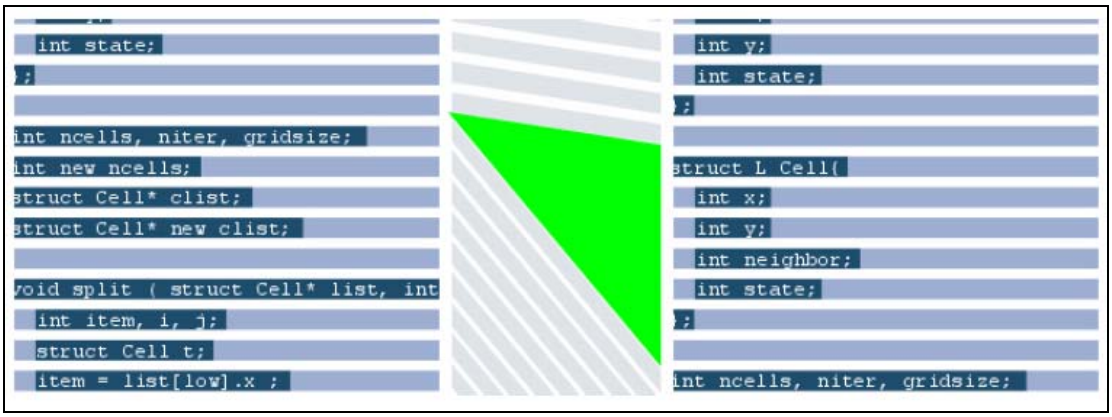

Figure 3: Low-level zoom view of a series of versions. The code is actually readable at this level. [13]

Chapter 3: Methodology

As mentioned in the related work subsection, there have been many studies characterizing the relationship between the programming model, the architecture, the effort spent by the programmer and the resulting performance. However, these studies were mainly qualitative and based on case studies involving a specific architecture and usually one programmer or one programming group working on a problem. Our goal is to quantitatively study the relationship between effort and performance in a more general context involving many programmers and allowing us to look for distinguishing characteristics that have a significant influence on the performance of the program.

Therefore, our methodology for tackling this new territory is iterative. Figure 4 is a graphical representation of our methodology. The flowchart shows the phases that the data goes through from the moment it is collected to the point where it is used to draw results. The left-hand side illustrates the data processing phases. We have already discussed the automatic data collection tool in subsection 2.2.1. We have also presented the features of the CodeVizard tool in subsection 2.2.2. In the following subsections, we will discuss an additional tool, APMS which was specifically designed and implemented for this study. We will also present the method used to distinguish coding and debugging phases using CodeVizard. Finally, we present a brief overview of the iterative process in order to illustrate how the hypotheses and metrics were generated and used.

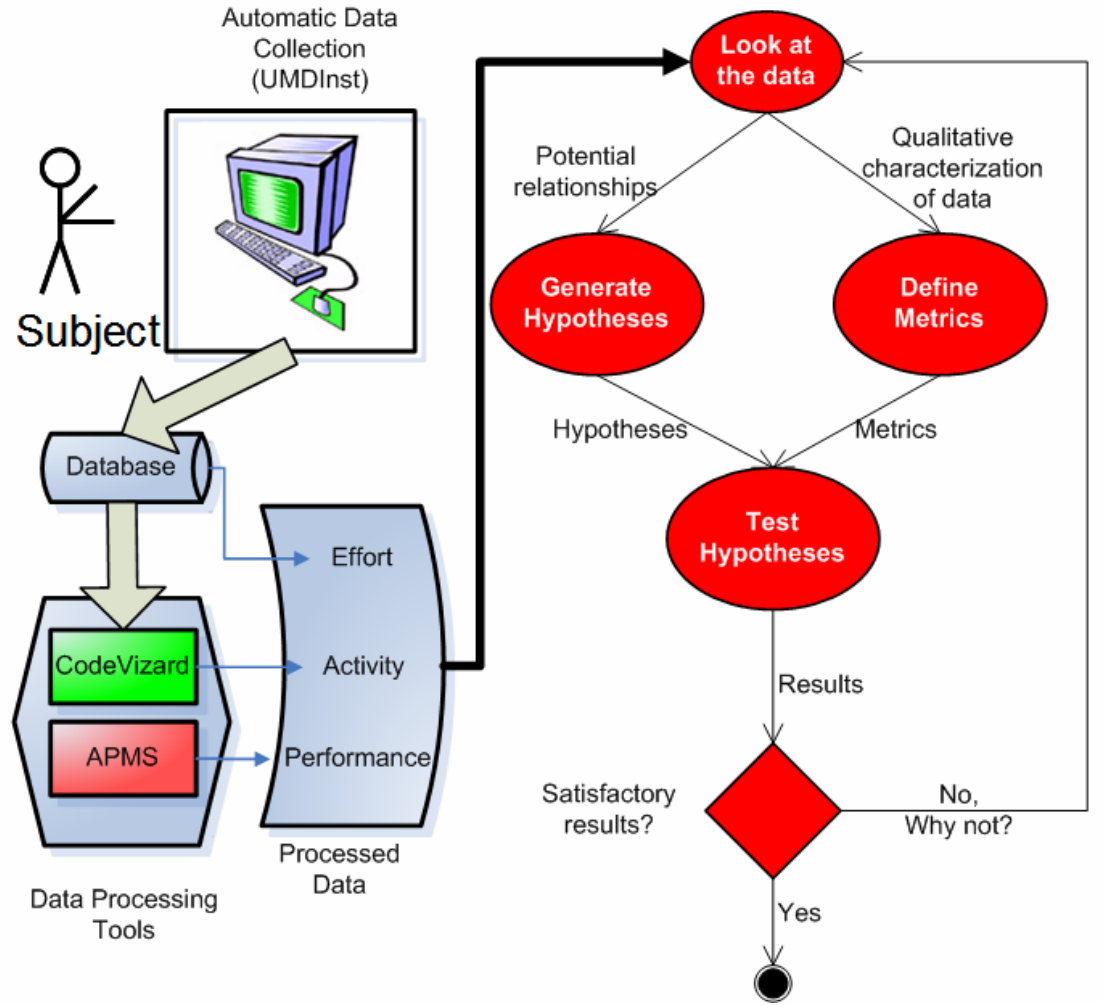

Figure 4: Flowchart representation of the methodology

We first state the steps of our iterative methodology:

1. Look at the data: In this step, we look at the data from class 1 in order to identify potential relationships and qualitatively characterize the data. In the first iteration of this process, we looked at the raw data in order to build a basis for the following iterations. After this step, we have some description of the data, which allows us to formulate informed hypotheses, test them, define them more clearly and generate new hypotheses in later steps and iterations. Therefore, in later iterations, we first process the data in a manner that suits our purposes and the

perspective that we are planning to look at it. This step consists of a brainstorming phase in which we try to identify some interesting patterns. These are then informally verified against data from class 1 to become potential relationships (for example, does it look like most of subjects in class 1 follow the pattern? Is there a notable exception to the pattern? …). Hence qualitative characterization of the data is used to determine whether the proposed relationships are promising.

2. Generate hypotheses and metrics: in this step, hypotheses are generated based on the initial data observations done and potential relationships specified in the previous step. In addition, the qualitative data characterization is useful in this step because it sheds light on the properties of the subjects and their distinguishing features, which serves as the basis for defining the necessary metrics needed to verify the hypotheses. Again in this step, the iterative methodology allows us to incrementally evolve our hypotheses from a qualitative description in the first iteration to a more quantitative model in later iterations.

3. Test the hypotheses: the hypotheses generated in the previous step are tested. This is the last step in each iteration, after which we either exit the process or go back to the first step. The decision to reiterate or terminate depends on the quality of support we get for the hypotheses. If the tests are unsatisfactory, we go back to looking at the data, now from a different perspective, trying specifically to identify the outliers in the data, and find out what aspect distinguishes them from other data points. This will help us modify the existing hypotheses to take into consideration the newly identified patterns, or generate new hypotheses.

We will now present the infrastructure needed, hypotheses generated, and metrics used in the application of the described methodology.

3.1 Infrastructure

3.1.1 Automated Performance Measurement System (APMS)

Capturing the characteristics of a program's performance requires running it on multiple sets of inputs. Typically in HPC these inputs include: the number of processors the program should be executed on, and problem-specific parameters such as grid sizes, number of iterations, or convergence thresholds. Furthermore even running one set of inputs multiple times can be useful since run time fluctuates between runs due to external factors of the runtime environment. Therefore, measuring performance of codes by hand, on every combination of inputs, can be tedious, time consuming and error prone.

To complicate matters, we want to perform a longitudinal study across similar assignments from different semesters and different universities. Unfortunately, while the assignments from different classes are similar, details such as command line arguments, and input file formats often change from assignment to assignment. We needed a tool that could automate both running the programs and eliding these minor differences. For this reason, we developed the Automated Performance Measurement System (APMS) which is a web-based tool that automates the process of running a large number of student codes. We will now present the important features of APMS. For more information about the system's design and functionality, consult the appendix and/or [14]

APMS is designed to make the job of measuring faster and easier and to provide all information the user would get if the program was executed manually. This includes process exit status, error and warning messages generated, and output produced. The system is accessed through a web browser. The user can select source files from the database, define input parameters and a set of possible values for each of them, then start the compilation, execution and performance measurement on a target cluster, and finally observe the results.

APMS measures performance metrics for a set of parallel programs on a set of input parameters. The system reads source code from and writes results to the HPCS database, to enable the user later to aggregate performance data with other data that was captured during the classroom studies (e.g. workflow, effort and defect data). The system is fully automatic: once the criteria for programs to run are defined, the user doesn't need to interact with the system while the programs are executing. This feature is useful since although individual runs of programs are typically only a few seconds, a full performance study could require hundreds or thousands of individual program executions.

Alongside the performance measurement, APMS provides the user with the capability of deciding whether program outputs are correct. Even if performance is the only variable being studied, correctness must also be verified since performance measurement results are only valid if the program produces the correct result. Judging the program correctness is not done automatically; instead the tool aims to visually display program outputs along side a known good output to facilitate the comparison

of their contents, and provides an interface for the users to record their correctness decision.

To collect performance measurement for MPI + C programs, we are using the PAPI library [15], which provides a consistent interface to the performance counter hardware found on most parallel machines. The tool currently supports collecting the real time, CPU time (time spent in user mode), number of floating point instructions (FLOPS), and FLOPS per second. Additional PAPI events could easily be added, provided they are supported by the PAPI library and by the cluster that the code is run on.

Performance metrics vary depending on the purpose of measurement. If the goal is to measure overall speedup, the user would need to measure the total execution time. If the user needs more detailed information (like measures related to the machine architecture), lower-level metrics are needed, such as floating point instructions per second, memory and cache accesses. For this reason, the tool was designed to give the user freedom in choosing the performance metrics that should be measured.

One tricky part of developing such a system is how to specify the adaptation layer to manage mapping program parameters and input file formats between the formats used in different classes. In fact, it turned out that frequently the assignment for a class was under specified, and even within one class, students did not use the same pattern to pass input parameters to their programs. For example, some classes

expected input options from a specific file name, while, others read them from command line arguments and some had a mix of both techniques.

To handle these combinations we came up with a simple language that allows specifying parameters, their corresponding values, and the format for passing them to the program. Our approach is based on the separation between the parameter values and their order and format when passing them to the program. For a user, this means following these steps:

1. Specifying the value for each parameter

2. Specifying for each parameter whether it should be passed on the command-line or through a file.

3. Specifying the order for the command line parameters and input file parameters

These steps are discussed in further detail and with examples in the appendix.

3.1.2 Work and rework phase distinction using CodeVizard

One of our objectives as part of the HPCS project at the University of Maryland was the identification of development phases in the recorded student programs. Our approach divides the development process into phases of work and rework: work being the implementation of new features and rework structural or functional modifications to existing features.

Usually rework is the result of the current program not producing the expected output, or running too slowly, which is why it's closely associated with debugging.

For this reason, we base our methods on Nakamura's reading-based code analysis method [3]. Nakamura developed this method in order to identify what defects exist in the code, when they were inserted and when they are fixed. Our goals are more general, since we're not interested in the type of defect, or how it was fixed. On the other hand, information about the insertion time of a bug and its fix time can be very useful in determining a major subset of the rework periods. We found Nakamura's approach very suitable for our goals, with a few modifications. Below is a summary of some heuristics adapted from Nakamura for our purposes:

- "Familiarize yourself with the code": we look at the last version of the source code to understand the code structure, the algorithm used to solve the problem, communication pattern, and language features used.

- "Look at big changes to determine their intention: a typical pattern of code history consists of big changes interspersed between a series of small changes." CodeVizard allows us to see the size of the change by visually graphing the lines of code added, deleted and modified. We modified Nakamura's approach in that we don't look at any kind of big change, but specifically big additions. Common activities indicated by big additions are: additions of new functionality, addition of comments, debugging (addition of many interspersed debug statements). Therefore, inspecting these changes at a lower level was necessary to determine their intentions.

- "Look at small changes before big changes to locate fixes": since we are not interested in locating the fixes, but simply determining when the rework starts

and ends, we modified this step to looking at interesting changes between big additions changes, again to determine their intention. A common pattern is that developers make a big change after they finish debugging. Therefore, the changes before the big additions often represent rework activities, such as deletion of all debugging code, commenting/uncommenting blocks of code, inserting a comment block.

- Skip rework versions: The number of versions for each file can be up to several hundreds. A pattern we noticed is that work usually occurs in only a small number of versions, whereas rework occupies significantly more versions. Hence, a good strategy to save effort and time was skipping some of the rework versions, i.e. not to inspect them at low levels of detail. After studying a few subjects, we followed the following guidelines for skipping versions:

    - skip one line change versions: usually this turns out to be commenting or uncommenting a line of code, inserting or deleting a “printf” statement for debugging.

    - skip consecutive versions where changes are collocated: when a series of changes occurs in the same block of code is a good indicator of rework.

Figure 5 shows how the high-level view can quickly help us determine what versions to skip and what versions to inspect more closely.

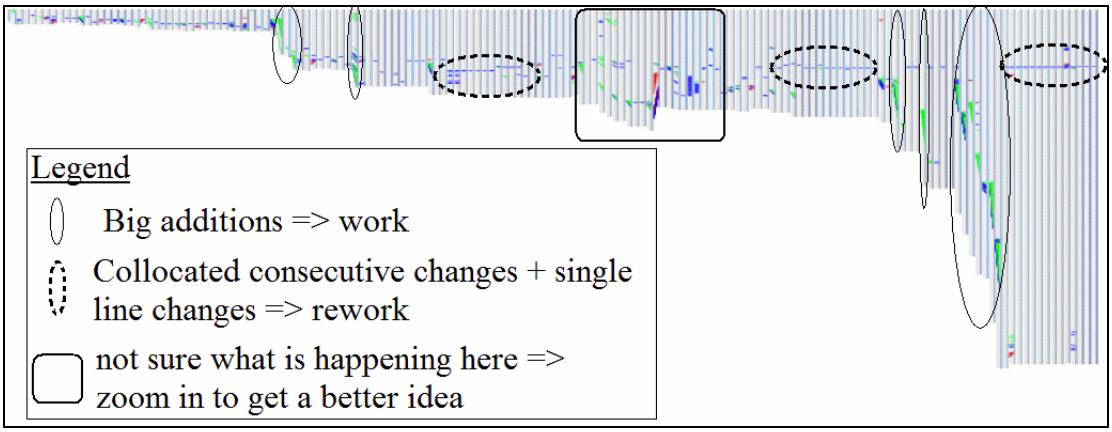

Figure 5: The use of CodeVizard in distinguishing work and rework phases

3.1.3 Data used

Our approach is opportunistic. We draw our data from a series of studies performed by the University of Maryland [6] . The data available was collected from over 20 HPC classes at several universities in the United States. Some classes focus on the HPC systems, while others emphasize parallel algorithms, but all courses require a number of programming assignments using several programming models or languages.

A common criticism of classroom based studies is that the problems are not representative of the types of problems that HPC practitioners actually solved. Choosing more relevant problems is difficult for two reasons: first the students don't have much time to solve their assignments (a couple of weeks) and second the classes don't require numerical computing knowledge. This limits the complexity of the assignment problems and differentiates them from actual HPC problems. In order to minimize this difference, we chose the assignment that was the most complex to solve using the MPI programming model: the conjugate gradient (CG) problem. The CG

method is an iterative algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive definite.

We chose to study MPI because on one hand it is the most dominant parallel programming technology in terms of numbers of users. However, MPI is difficult to use because it requires the programmer to deal with communication details at a very low level of abstraction. Therefore, MPI is a good representative for the programming environments used by scientists, making performance gains attainable, but at a cost.

Fortunately, MPI assignments compose a significant part of the collected data that we have. We found in our database three assignments in which students solve the CG problem using MPI with C as base programming language. Class assignments 1, 2 and 3 emphasize in their statements getting good performance. The class 1 assignment states that "there will be credit for performance" in the grading policy, class 2's specifies that performance forms 50% or the grade, and class 3's requires the students to report the performance gains they achieved and to comment on them.

Class	Number of students with MPI versions	Number of students with correct programs
1	12	7
2	8	4
3	5	3

Table 1: Number of subjects in each class

Using APMS, we ran the last version of each student's work and inspected the outputs to judge correctness. Table 1 shows the number of students who solved the problem in MPI and the number of students with correct versions found in every

class. Those last subjects form our sample data. Due to the small number of subjects in classes 2 and 3, we decided to aggregate the data. This was possible because both assignments required the students to implement only the conjugate gradient routine and because the students had the same amount of time (two weeks) to solve the problem. The assignment of class 1 on the other hand was different from the others in that it also required the students to implement a parallel input generation function in addition to the conjugate gradient function. This prevented us from aggregating all classes together. The aggregated data from classes 2 and 3 will henceforth be referenced as belonging to class 2+3.

The students differ in backgrounds (Computer Science, different areas of engineering, sciences) and hence in experience with generic programming and knowledge of related scientific fields. All the students, however, are new to parallel programming, and they have to learn the new models in a short period of time in order to solve the assignments.

The combination of elements of assignment, programming model, HPC expertise and performance goal makes the subjects our closest available to novice programmers in real HPC projects. Of course, studying experts is also interesting. However, it has proved hard to get them to participate in such observational studies. Experts working in the industry are discouraged from participating to protect the work's confidentiality and possibly the competition from other companies in the industry. University professors teaching the HPC classes don't view themselves as experts since they're not regularly engaged in HPC programming. In addition, experts whether working in academic or industry institutions, might also be apprehensive to

participating in a study and worried about how much of their time it would take up, even though the effort overhead involved in our studies are minimal.

Fortunately, the CG problem has a well-known benchmark, the NAS CG benchmark[2]. The CG benchmark is designed to test irregular long distance communication using unstructured matrix vector multiplication. It is part of a set of programs developed by the Numerical Aerodynamic Simulation (NAS) program with the objective of evaluating parallel supercomputer performance. The NAS Parallel Benchmarks (NPB) are specified as "pencil and paper" benchmarks: their details are specified only algorithmically, in order to keep the benchmarks generic. For example, the CG benchmark gives the developer the freedom to choose the suitable data structures for the particular architecture to be used. However, the nature of the computations and the expected results are specified in great detail in order to allow verification of result correctness and performance [16].

Understanding the difficulty and effort involved in implementing the benchmarks from scratch, as well as the possible ambiguities in the technical specification, the benchmark developers also provided sample code. In addition, benchmark implementations are available from vendors. These implementations were verified by the NAS division. We use the NPB 3.2-MPI version for our purposes. For the CG benchmark, it is required that the input matrix A be used explicitly and generated using the provided subroutine called makea. This subroutine may not be changed [16]. Although using this benchmark won't give us an idea of the effort of

[2] http://www.nas.nasa.gov/Resources/Software/npb.html

the programmers, it can be used as a good example of the performance that an expert can achieve.

We used the NAS CG benchmark for another purpose: when comparing performance across multiple classes and multiple students, we must make sure that the inputs we are using are consistent in all of our tests. Therefore, we use the input generating functions of the CG benchmark and plug them into the student code. This was possible for classes 2 and 3, because the instructor gave the students a very strict harness to work according to and they had also provided the input generating functions. In addition, students weren't allowed to modify the main function of the harness. Hence it was easy to simply replace the call to the given input generation function with the CG function call in the main code. Class 1's harness on the other hand was looser and the students were required to write their own input generation functions which made it impossible to use the benchmark input generation functions without significantly modifying the student's code. Luckily the data generators that the students were supposed to write were similar to input set S of the benchmark code, i.e. they used the same size matrix and the same number of non-zeroes per row. For this reason, assignment 1 code was only tested on set S of the benchmark while B and C were tested on sets S, W, A, B and C. Table 2 shows the defining parameters for the benchmark classes. As for the number of processors the programs were tested on, we used 1, 2, 4, 8 and 16 processors because that is the number of processors students were required to report on.

Benchmark code input set	Matrix size	Number of non-zeroes per row	Number of iterations
S	1400	7	15
W	7000	8	15
A	14000	11	15
B	75000	13	75
C	150000	15	75

Table 2: NAS benchmark code set parameters

3.2 Hypothesis generation process

Our hypothesis generation process was based on our largest data sample, class 1. We use this class to generate hypotheses which we test, at a later stage, against all classes. At the end of this subsection, all our generated hypotheses will be listed, and their numerical verifications will be presented in the data and results sections. In this subsection we only present our initial look at the data and how we used it to find clues to possible hypotheses.

We start with the hypothesis that was reported in all studies of effort and performance in HPC: achieving performance requires significant effort. To get a sense of whether or not this hypothesis is true, we look at raw effort and performance data from class 1. Looking at Figure 6, we would expect students 66, 68, 70 and possibly 78 to have done better than 69, 71 and 81 with regards to performance.

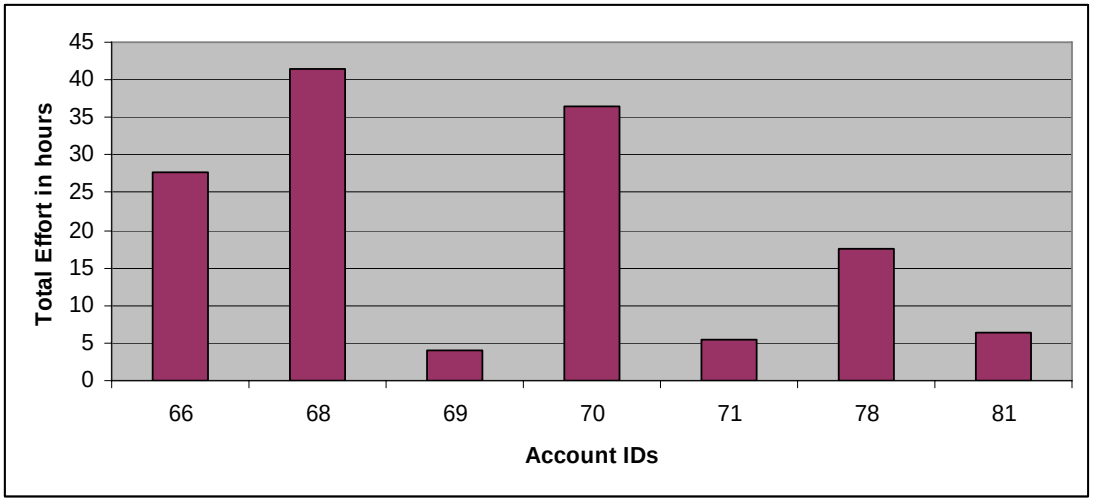

Figure 6: Total effort in hours for subjects in class 1

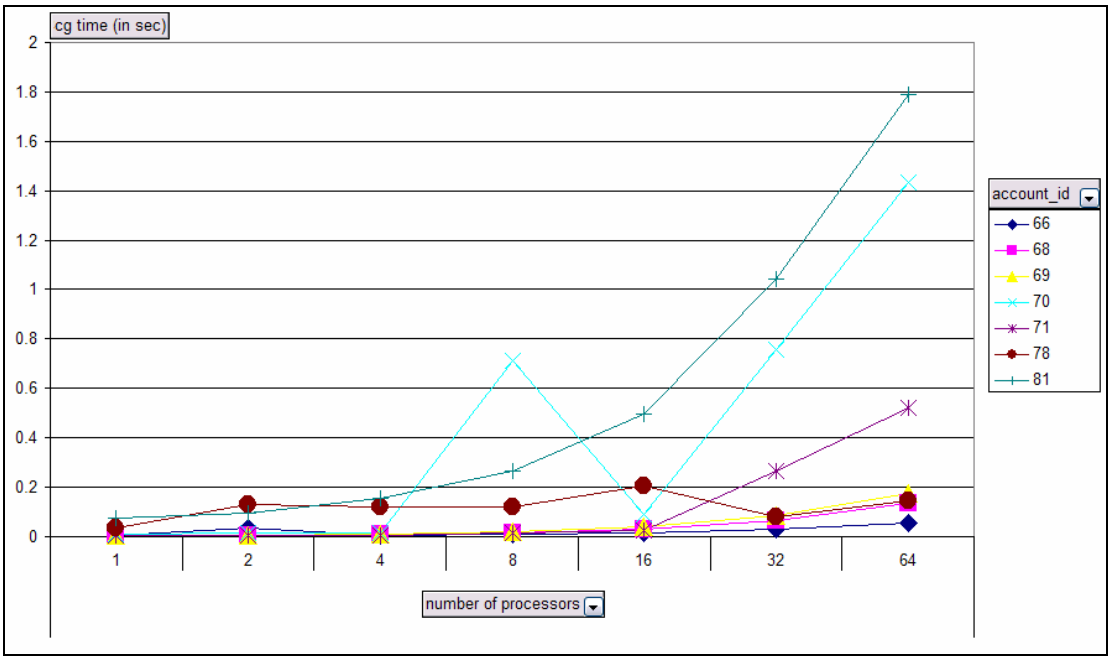

Figure 7: Time spent in CG routines for subjects of class 1 on 1, 2, 4, 8 and 16 processors

Comparing our expectations with the performance graphs for these subjects (Figure 7), we find that 66 and 68 do in fact have the best times, 81 has the worst times, and 78 is somewhere in the middle of the spectrum. On the other hand, 70's performance is not so good, as it is not consistent and takes much more time on 16

processors. Subject 69 doesn't meet our expectations either: its performance is comparable to 66 and 68's, although the effort spent by that subject is much smaller.

Although most of the data points seem to adhere to the hypothesis, we found it interesting how someone can spend so little effort and get such good performance, and someone else can spend so much effort and get such poor performance. Perhaps, there is another factor that could explain this, and make our hypothesis even stronger. The total effort contribution of the subject wasn't enough to explain the observed differences. Our attention therefore turned to not only the end result (the total effort), but how it was reached. In their paper, Numrich et. al. [16] defined a metric space for measuring individual team member contributions to a software development project. They note that "two students can contribute equally but follow very different paths". We therefore go back to our first step in iterations, looking at the data, this time looking at effort from a different perspective: how it is distributed daily.

For class 1, this yields the graph shown in Figure 8. The order of subjects has been changed to improve the visibility of those with low effort. The day numbers in this figure (and in all remaining figures involving the day variable) correspond to the number of the day with respect to a year. For example, "133" represents the 133rd day of a year.

We notice two patterns in this graph:

1. There are 3 kinds of working days: high effort days, medium effort days and low effort days.

2. Certain subjects, such as 81, 66 and 70 take a break at some point in the development process.

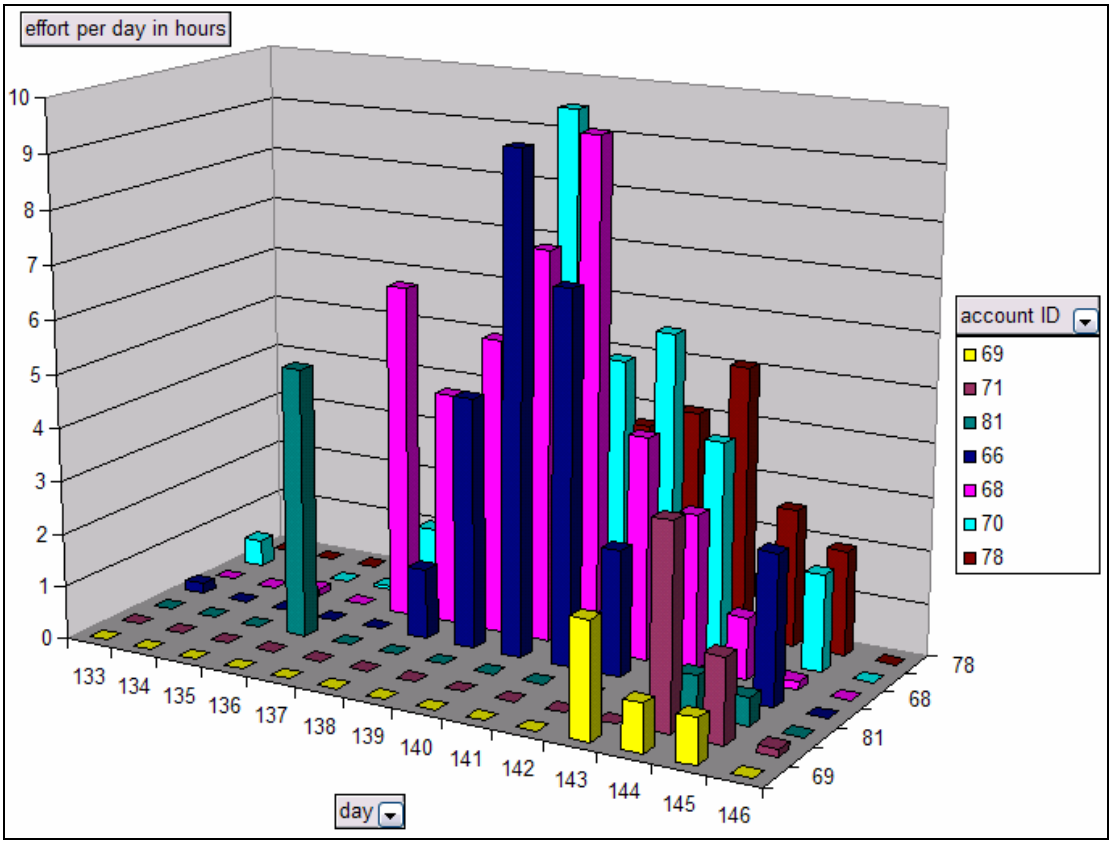

Figure 8: Effort per day in hours for each subject in class 1

Using these patterns we distinguish subjects into two categories:

1. Slow and steady: these are the subjects who work on a regular daily basis, most of their days are medium working days and they take at most one break day.

2. Fast and furious: these are the subjects who are not consistent in distributing their effort across days. We see for these students, a significant number of breaks and low effort days, and some number of high effort days.

Our second hypothesis states that slow and steady wins the race. The closer a subject's effort pattern is to the slow and steady extreme of the spectrum, the better their performance. Investigation of this hypothesis is left to later sections.

A third perspective for looking at the effort data is to analyze the distribution of effort over activities, where "each kind of activity corresponds to some work that advances the student toward the solution of a problem" [16]. The extent of advancement of each activity can be viewed as a power rating. As a first attempt at this problem, we chose to simply divide the student activities into coding and debugging. Our coding activity includes writing serial code, parallelizing it and adding functionality. The debugging activity is any activity in which the subject is trying to locate and/or fix a compile-time error, a run-time error, or a functionality error (the code is not behaving as expected).

We would expect that the coding activity has a higher power rating than debugging, i.e. effort spent coding counts more towards performance than effort spent debugging. Distinguishing between these two activities is done using CodeVizard, as described in subsection 2.2.2.

Our last hypothesis relates the performance of novices to that of experts. We hypothesize that some novices do achieve the performance that experts achieve.

We conclude this section by a list of all generated hypotheses:

- For novices, total effort and performance are correlated.

- Novices who spend more than 15 hours working on a problem achieve better performance.

- Slow and steady wins the race: novices who spend their time more regularly and pace themselves achieve better performance than others.

- Debugging counts less towards performance than coding.

- Some of the best novices perform as well as experts.

3.3 Metrics used

3.3.1 Effort Scoring

As mentioned in subsection 2.2.1, our instrumentation package UMDInst collects timestamps of various computer events. Therefore, the best method for estimating effort is the interval-based method, where effort is estimated by adding up time intervals between events. We must however be careful not to count "time intervals that represent non-working gaps between work sessions" [18]. The question is: what time interval is long enough to be considered a non-working gap? A series of studies by Hochstein et. al. [18] revealed that in the presence of sufficient automatically collected data (compile, edit, shell events), the most reasonable threshold between work and non-work intervals is 45 minutes. Hence, total effort can be expressed through the following equations:

$E = \sum_i f(t_i)$ where t_i represents an interval between two captured consecutive events

$$f(t)=\begin{cases} t & t \le 45\ minutes \\ 0 & otherwise \end{cases} \qquad equation\ \ 1$$

As for the daily distribution of effort, we need to define a measure that captures the steadiness of the student work and its speed. Therefore, we define the following quantities:

- Effective work day: a day in which more than half an hour is spent.
- Medium work day: a day in which effort between half an hour and 5 hours is spent.
- Break day: a day in which less than half an hour is spent
- $Effective\ \ work\ \ percentage = 100 \times \frac{number\ of\ \ effective\ work\ days}{total\ number\ of\ \ effective\ days}$
- $Medium\ work\ \ percentage = 100 \times \frac{number\ of\ \ medium\ work\ days}{total\ number\ of\ \ effective\ days}$
- Break length: number of consecutive break days.

The medium work ratio and effective work ratio are used to assess the steadiness of the subject's work. The closer they are to 100%, the steadier the work is. The reason we are defining two such ratios is that we would like to check whether working for too long in a single day is in favor of producing better performance. Our hypothesis is that it is not, because at some point, we expect productivity to start decreasing.

Break length is easy to measure since we observed that all students have at most one block of consecutive break days, usually slightly after the mid-development point in time. An important distinctive feature between the slow and steady approach and the fast and furious approach is the break length: on one hand, slow and steady subjects don't usually take breaks; fast and furious subjects, on the other hand, take long breaks, especially between two extensive work days. Therefore, we incorporate a penalty for breaks in our metric. The break length is used to determine the break penalty as follows: 10 points for a break between 1 and 3 days, 20 for break duration of 3 and 6 days … The reason we are using a break penalty is our assumptions that after a break a certain amount of time is needed to get up to speed to the last status of the program, and the longer the break, the more time is needed to catch up. This "catching up" time is similar to the debugging activity time in that it doesn't contribute as much as other activities to the performance of the problem.

Our scoring function for daily effort distribution is then expressed as:

Effective/medium work percentage - break penalty

Our final effort measure depends on the distribution of effort over the coding and debugging activities. Let α_1 and α_2 be the power rating of the coding and debugging activities respectively. Then, following the same derivation process described in [16] and as shown in Figure 9, time is divided into intervals T_i such that only one activity is performed in the interval. In general, more than one event occurs within these activity intervals, so each interval is also divided into subintervals t_{ij}.

The effort spent within each interval is estimated according to equation 1 (i.e. subintervals with length more than 45 minutes are assumed non-active intervals).

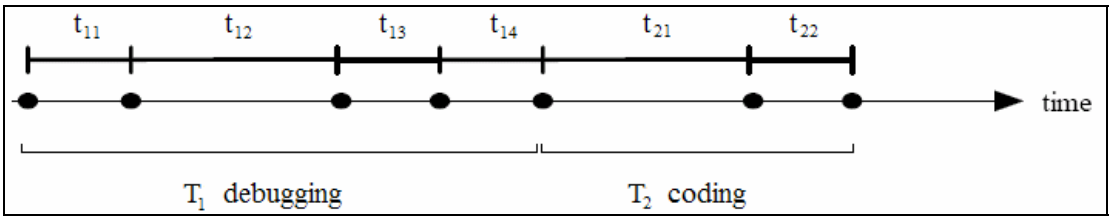

Figure 9: Interval division in activity-weighted effort estimation

Total work is then calculated as a weighted sum of the estimated effort in each interval:

$$W = \sum_i \alpha_i \times w(T_i)$$

where T_i represents an interval, α_i is either α_1 or α_2, depending on the interval activity (coding or debugging), and $w(T_i) = \sum_j f(t_{ij})$

where t_{ij} is a subinterval contained within T_i and $f(t_{ij})$ is as defined in equation 1.

3.3.2 Performance Scoring

Performance measurement is less straightforward than effort measurement because the performance of a particular program depends on many factors, such as inputs, machine architectures, compiler and library implementations. For this study, as previously mentioned we unified the input to use the benchmark input generation code and we have also unified details related to machine architecture, compiler and library implementations through the use of APMS, which allowed us to easily run all the programs on the same cluster.

Another complicating factor is what measure to use. For example, using speedup is a measure of scalability, rather than strictly a measure of performance. It is possible that, given two programs (A,B), program A exhibits better speedup on N processors than program B, but program B actually runs faster than program A [19]. In classroom assignments, both scalability and performance are important. To keep our analysis simple, we are looking to use one measure that combines them both.

Our study of the performance graphs of the subjects over 1, 2, 4, 8 and 16 processors revealed to us that performance on one processor generally determines performance on multiple processors, meaning that in general if subject a did better on one processor than subject b, the same can be said about more processors in general. In some cases, execution time on a certain number of processors increases dramatically and then goes back to the regular trend of the particular subject, as is the case for subject 70 in figure 2. As we will see in data from the two remaining classes, a subject could have very good performance, but suddenly at some point execution time increases, and then later goes back to normal. This kind of behavior is not completely bad since it only misbehaves in one case. In addition, if this misbehavior occurs on 4 processors, we consider it better than if it occurs on 8 processors, since the goal in HPC is to run efficiently on the greatest number of processors possible.

We are then looking for a metric which rewards good performance from the start, increases the reward as we increase the number of processors if the performance remains consistent and incurs penalties if performance deteriorates. Rewards and penalties should both increase as the number of processors increases. For this purpose, we use the following scoring function for performance:

$$PF = T_1 + 2T_2 + 4T_4 + 8T_8 + 16T_{16}$$

where T_i represents the execution time on i processors, for $i \in \{1,2,4,8,16\}$.

PF is a weighted sum of execution times and therefore the smaller this performance score, the better the performance. The significance of this function can be seen graphically. Figure 10 shows the execution times of three hypothetical subjects: ideal, actual 1 and actual 2. In the ideal case, we observe linear speedup, where the execution time on two processors is exactly half the execution time on one processor and so on. In the actual cases, the subjects don't achieve linear speedup, and as we generally observed from the student data, actual 1 starts with better performance than actual 2 and conserves it.

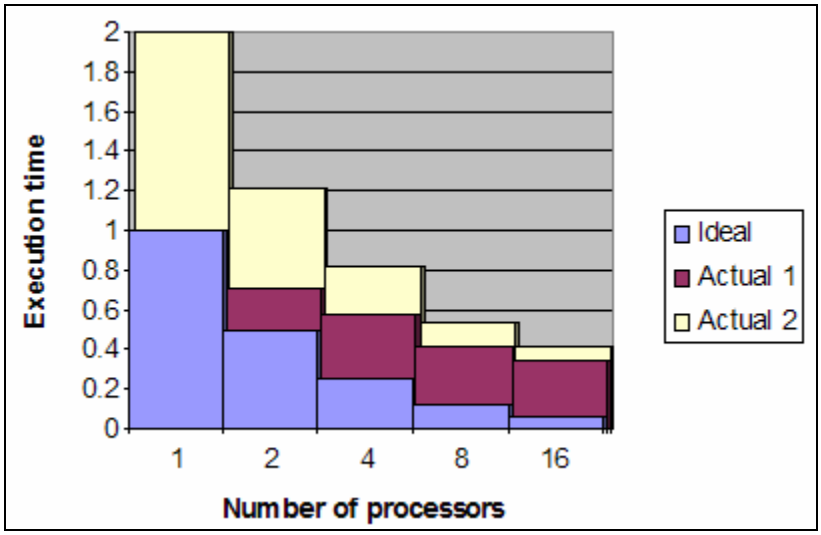

Figure 10: Hypothetical execution times for 3 subjects: ideal, actual 1 and actual 2

Figure 11 represents the performance score for the same hypothetical subjects. Each bar represents the product of the execution time by the corresponding number of

processors. So the performance score can be viewed as the total area contained within the bars.

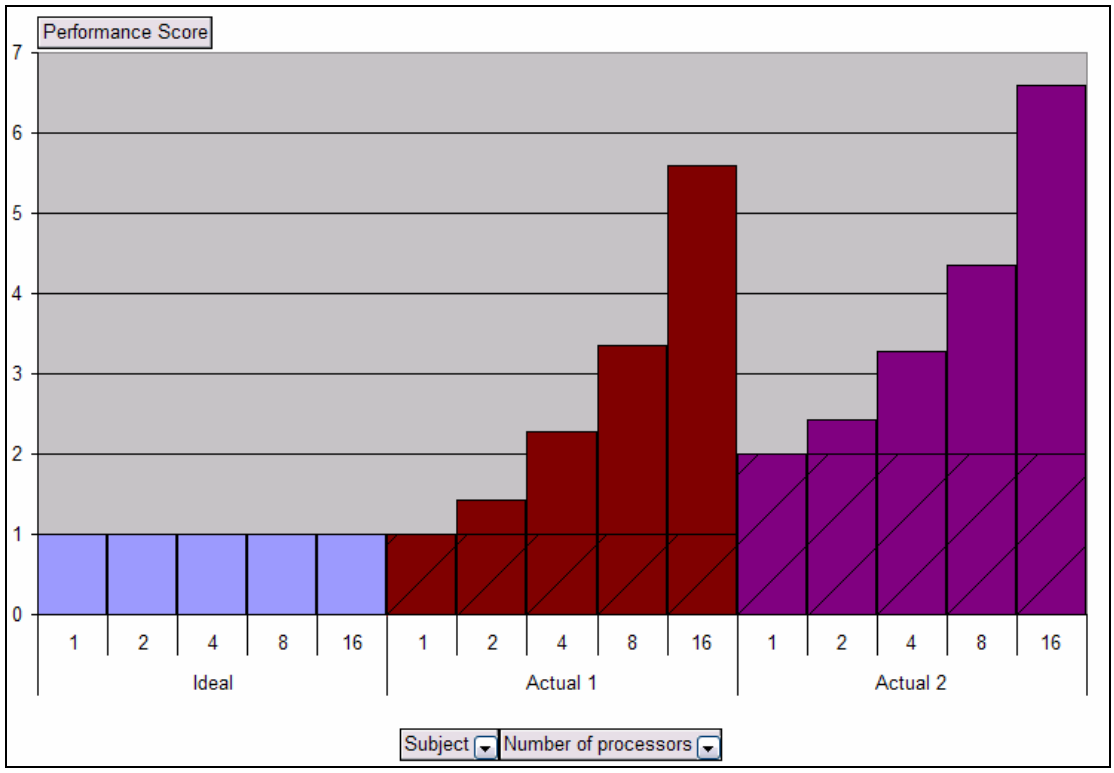

Figure 11: Graphical representation of the performance score

For the ideal case, since $T_1 = \frac{T_2}{2} = \frac{T_4}{4} = \frac{T_8}{8} = \frac{T_{16}}{16}$, the area is a rectangle with height T_1. The performance score of each actual subject can be viewed as the sum of two areas: the hashed area and the plain area. The hashed area is rectangular, similar to the ideal case performance score graph. In fact, the hashed areas represent the performance score of a subject's program which starts with the same time as the actual subject on one processor, but scales ideally (time on 2x processors is half the time on x processors). The difference between the two areas is governed by the time on one processor. Therefore, the subject with a smaller hashed area has better execution times than the subject with a larger hashed area, especially that we

observed in our samples better time on one processor implies generally better time on all other numbers of processors. The non-hashed areas on the other hand are the difference between the actual performance and the corresponding ideal scaling performance. Hence it's a measure of the scaling property of the code: the subject with a smaller non-hashed area wrote a program that scales better. Indeed, from Figure 11, the non-hashed areas indicate that the ideal subject scales better than actual 1 and that actual subject 2 scales better than actual subject 1 (which can be easily verified from Figure 10 which shows that actual 2 starts at a much higher execution time than actual 1, but ends at a slightly higher execution time).Thus, our performance score is a sum of a performance measure and a scalability measure.

Also note that since time on higher numbers of processors has a higher weight (equal to the number of processors), inconsistencies on higher numbers of processors will count more than inconsistencies on lower numbers of processors. The same applies for good performance rewards. Another observation to keep in mind when reading the study results is that a high performance score is indicative of bad performance and/or bad scalability features.

As mentioned in the data used subsection, we tested programs from classes 2 and 3 on five different benchmark input sets. The performance score presented above applies for each one of these sets. Moreover, the same reasoning applies with respect to the increase in data size. Once again, we want to give more weight to bigger problem sizes because they are closer to real life situations and real life goals. Therefore, we use a weighted average of the performance scores on all input sets to reach the final performance score. The weight for each input set was determined in

order to comply with the matrix size ratios of the classes. Table 3 shows the weights for each input set.

Benchmark code set	S	W	A	B	C
Weight	1	5	10	50	100

Table 3: Weights given to various input sizes for NAS benchmark in performance scoring function

Chapter 4: Data and results

In this chapter we will present data and results related to each hypothesis in turn, in the same order as they were discussed in the hypothesis generation process subsection.

4.1 Hypothesis 1: Total effort and performance are correlated

Table 4 shows the execution times in seconds for each student in class 1 on 1, 2, 4, 8 and 16 processors, as well as the calculated performance score and the total effort in hours.

Subject / Number Of processors	66	68	69	70	71	78	81
1	0.005	0.003	0.003	0.012	0.003	0.037	0.075
2	0.073	0.011	0.012	0.027	0.009	0.259	0.191
4	0.021	0.037	0.046	0.060	0.028	0.477	0.615
8	0.094	0.133	0.166	5.691	0.105	0.981	2.106
16	0.255	0.511	0.615	1.453	0.427	3.316	7.932
Performance score	0.448	0.695	0.843	7.243	0.572	5.069	10.919
Total effort in hours	27.63	41.35	3.92	36.43	5.49	17.45	6.30

Table 4: Execution time in seconds, final performance score and total effort in hours for subjects in class 1

Figure 12 shows the plot of the total effort for subjects in class 1 versus their performance. It also includes the linear trendline for this data as well as the correlation coefficient squared. The correlation coefficient R is equal to -0.118, indicating a weak negative correlation.

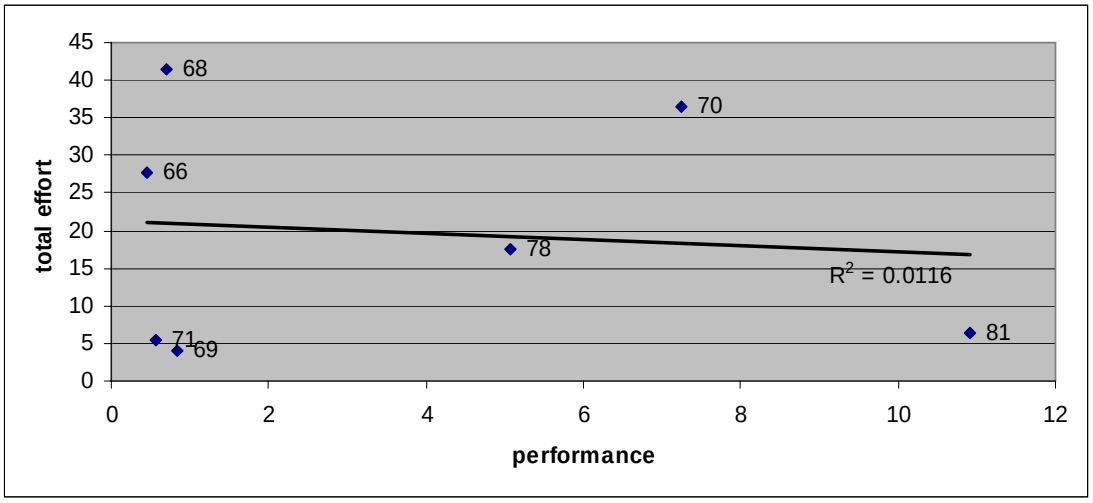

Figure 12: Scatter plot of total effort versus performance score of subjects in class 1

We now present similar data for class 2+3. Because for these classes, performance was tested on 5 input sets of the benchmark, we show in Table 5 the performance score of each subject on each one of these input sets, and then the final performance score. Finally we show the total effort in hours for the subjects. Figure 13 is a scatter plot of the total effort of subjects versus their final performance score. The correlation coefficient for this set of data is 0.484. The relationship in this set is stronger than in the first set. However it is in a direction opposite than hypothesized, i.e. as the total effort increases, the performance score increases, meaning that performance actually deteriorates. Therefore, we have found no evidence supporting our hypothesis.

Subject / Benchmark input sets	301	302	304	308	219	221	223
S	0.03	0.06	0.24	0.80	0.11	0.07	0.06
W	0.09	0.78	0.12	25.36	0.19	0.33	0.32
A	0.22	4.95	0.30	184.78	0.37	1.11	1.11
B	1.50	2327.98	4.27	3590.74	5.30	8.58	8.57
C	15.27	2630.77	15.53	6036.93	19.26	25.99	25.98
Final performance score	9.66	2286.32	10.66	4730.15	13.23	18.31	18.30
Total effort in hours	10.91	21.82	16.23	16.67	10.01	4.32	17.24

Table 5: Execution time in seconds, final performance score and total effort in hours for subjects in class 2+3

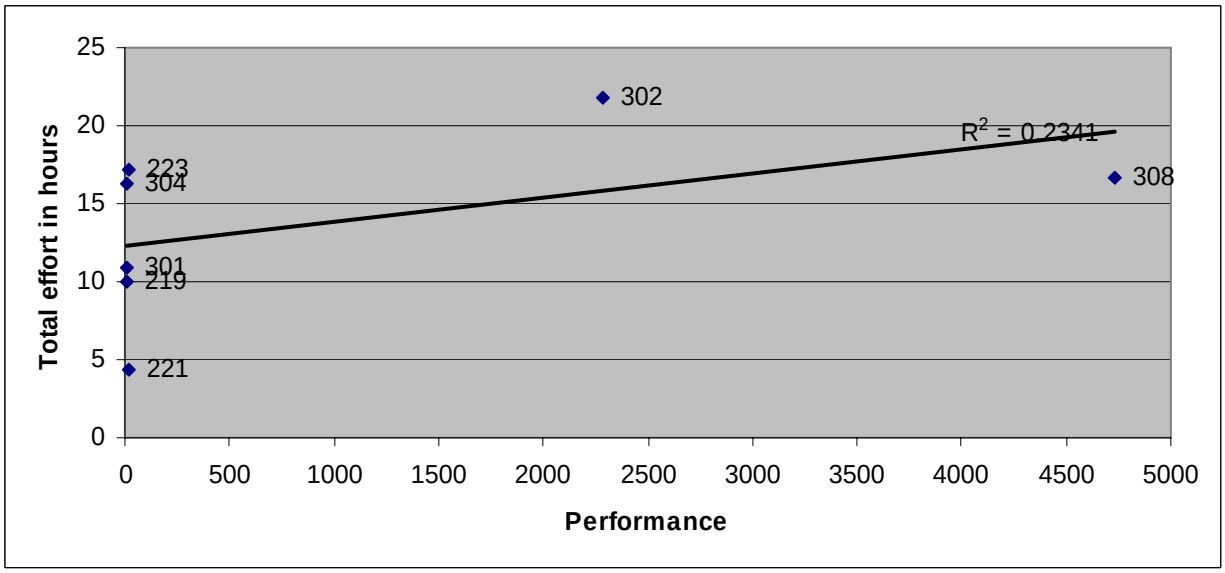

Figure 13: Scatter plot of total effort versus performance score of subjects in class 2+3

4.2 Hypothesis 2: Novices who spend more than 15 hours working on a problem achieve better performance.

Our goal is to make inferences about the novice subjects divided into two populations, one that spends more than 15 hours working on the problem (population 1) and one that spends less (population 2). We denote by μ_1 and μ_2 the respective means of the two populations and define $\delta = \mu_1 - \mu_2$. The difference between the means of our samples provides a point estimate of δ. In our situation, the populations variances are not known, but we can assume that they are equal, and hence can be estimated by s_p^2: $s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 2)s_2^2}{(n_1 - 1) + (n_2 - 1)}$

where n_1 and n_2 denote the size of samples 1 and 2, and s_1^2 and s_2^2 their respective variances [20].

We would like to test whether population 1 does on average better than population 2 in terms of performance, i.e. whether its mean performance score is less. As shown in Table 6 and Table 7, we divide the subjects in each of class 1 and class 2+3 into two categories, depending on whether they have spent more than 15 hours working on the assignments.

	Total effort more than 15 hours		Total effort less than 15 hours	
	Subject	Performance	Subject	Performance
	66	0.45	69	0.84
	68	0.69	71	0.57
	70	7.24	81	10.92
	78	5.07	-	-
Average	3.36		4.11	
Variance	11.20		34.77	
s_p^2	20.63			

Table 6: Division of subjects in class 1 depending on the total effort spent

	Total effort more than 15 hours		Total effort less than 15 hours	
	Subject	Performance	Subject	Performance
	302	2286.32	301	9.66
	304	10.66	219	13.23
	308	4730.15	221	18.31
	223	18.3	-	-
Average	1761.36		13.73	
Variance	5064169.74		18.88	
s_p^2	3038509.40			

Table 7: Division of subjects in class 2+3 depending on the total effort spent

We formulate the following hypothesis test:

H_0: $\delta = 0$

H1: $\delta < 0$

and use the pooled t statistic [20] with a one-tailed distribution to calculate its p-value. The result is 42% for class 1 and 84% for class 2+3. Because the values are large, the evidence suggests that we cannot accept our hypothesis.

4.3 Hypothesis 3: Slow and steady wins the race

Our slow and steady hypothesis states that novices who spend their time more regularly and pace themselves achieve better performance than others. For this hypothesis, we used two different scoring functions, one depending on the medium work ratio and the other on the effective work ratio as described in section 3.3.1. Our goal is to assess whether working for too long (more than 5 hours) actually deteriorates the subject's productivity.

The results shown in Table 8 and Table 9 show support for our hypothesis, since the correlations with effort score (a) found are large and negative, i.e. the closer the effort distribution is to the slow and steady pattern the better the performance. The difference however between the correlation of performance and effort scores a and b is not consistent: in class 1, the two correlations are almost identical unlike in class 2+3. Therefore, we have evidence indicating that out hypothesis is true when considering medium work while we don't have any evidence to conclude either way, whether working too hard is detrimental.

Subject	66	68	69	70	71	78	81
Total number of work days	13	11	3	13	3	5	10
Number of medium work days	4	4	3	5	2	4	2
Number of effective work days	6	8	3	8	2	5	3
Break length	2	0	0	1	0	0	7
Medium work ratio (in %)	30.77	36.36	100.00	38.46	66.67	80.00	20.00
Effective work ratio (in %)	46.15	72.73	100.00	61.54	66.67	100.00	30.00
Effort score a (in %)	20.77	36.36	100.00	28.46	66.67	80.00	-10.00
Effort score b (in %)	36.15	72.73	100.00	51.54	66.67	100.00	0.00
Performance score	0.448	0.695	0.843	7.243	0.572	5.069	10.919
Correlation of performance with effort score a	-0.58						
Correlation of performance with effort score b	-0.59						

Table 8: Effort score and correlation with performance for subjects in class 1

Subject	301	302	304	308	219	221	223
Total number of work days	10	9	7	12	3	2	6
Number of medium work days	5	5	5	5	2	1	4
Number of effective work days	6	7	5	6	3	1	5
Break length	0	1	1	0	0	0	0
Medium work ratio (in %)	50.00	55.56	71.43	41.67	66.67	50.00	66.67
Effective work ratio (in %)	60.00	77.78	71.43	50.00	100.00	50.00	83.33
Effort score a (in %)	50.00	45.56	61.43	41.67	66.67	50.00	66.67
Effort score b (in %)	60.00	67.78	61.43	50.00	100.00	50.00	83.33
Performance score	9.66	2286.32	10.66	4730.15	13.23	18.31	18.30
Correlation of performance with effort score a	-0.72						
Correlation of performance with effort score b	-0.40						

Table 9: Effort score and correlation with performance for subjects in class 2+3

We also look at the data from a view similar to the one described in section 4.2: it's interesting to see whether there is a certain threshold for the effort score, such that subjects with higher scores perform better on average than subjects with lower scores.

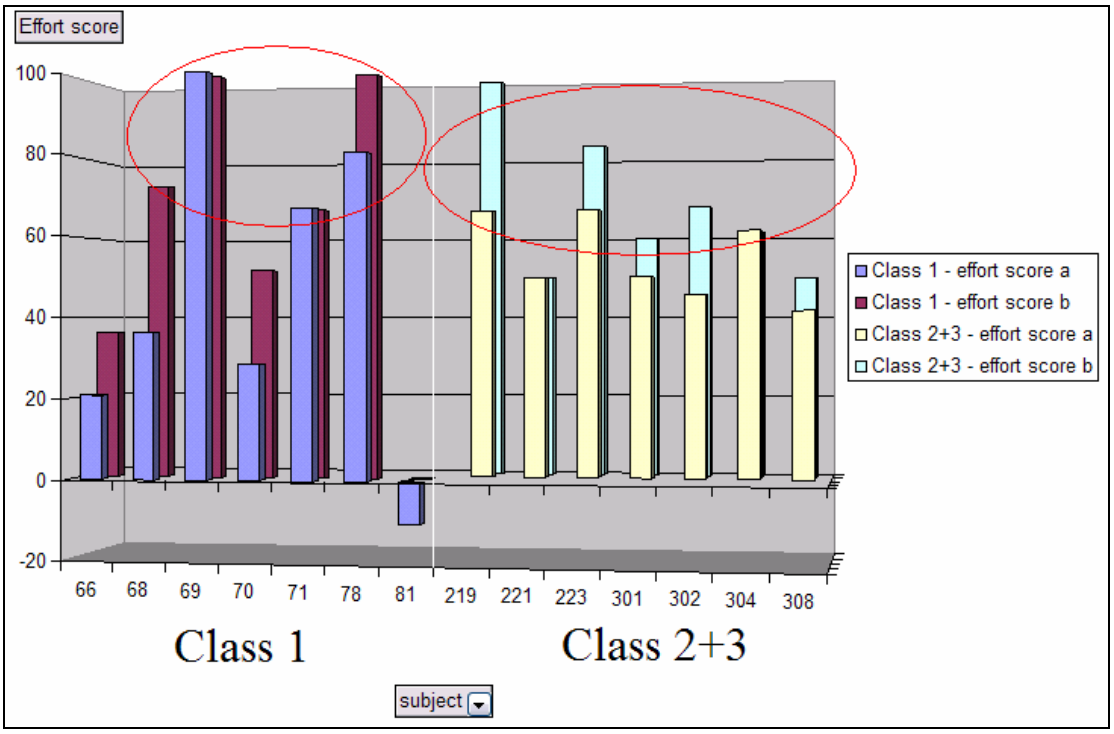

Figure 14: Effort scores a and b for classes 1 and 2+3

Looking at Figure 14, we can visually group subjects into those with high effort scores (indicated by red circles in the figure) and those with low effort scores. From this graph, it seems that 60% is a reasonable number for use as cutoff between "high" effort and "low" effort. In addition, since the effort score represents the effective fraction of days in which the subjects worked for significant amounts of time, it is reasonable to expect that a novice who spends 60% or more of his days working significantly on the problem produces an efficient program. For this reason, we look at the data in a slightly different manner, with the objective of investigating whether there is significance difference in performance between subjects who score more and less than 60%. We follow the same approach described in section 4.2 by dividing our subjects into two categories accordingly. Effort scores a and b for class 1 are shown in Table 10 and Table 11 and for class 2+3 in Table 12 and Table 13.

	Effort score (a) more than 60%		Effort score (a) less than 60%	
	Subject	Performance	Subject	Performance
	69	0.84	66	0.45
	71	0.57	68	0.69
	78	5.07	70	7.24
	-	-	81	10.92
Average	2.16		4.83	
Variance	6.36		26.40	
s_p^2	18.38			

Table 10: Division of subjects in class 1 depending on effort score (a)

	Effort score (b) more than 60%		Effort (b) score less than 60%	
	Subject	Performance	Subject	Performance
	69	0.84	66	0.45
	71	0.57	70	7.24
	78	5.07	81	10.92
	68	0.69	-	-
Average	1.79		6.20	
Variance	4.78		28.22	
s_p^2	14.16			

Table 11: Division of subjects in class 1 depending on effort score (b)

	Effort score (a) more than 60%		Effort score (a) less than 60%	
	Subject	Performance	Subject	Performance
	304	10.66	302	2286.32
	219	13.23	308	4730.15
	223	18.3	221	18.31
	-	-	301	9.66
Average	14.06		1761.11	
Variance	15.11		5065323.26	
s_p^2	3039200.00			

Table 12: Division of subjects in class 2+3 depending on effort score (a)

	Effort score (b) more than 60%		Effort (b) score less than 60%	
	Subject	Performance	Subject	Performance
	304	10.66	308	4730.15
	219	13.23	221	18.31
	223	18.3	301	9.66
	302	2286.32	-	-
Average	582.13		1586.04	
Variance	1290797.29		7414088.62	
s_p^2	3740113.82			

Table 13: Division of subjects in class 2+3 depending on effort score (b)

Finally, Table 14 shows the p-values calculated for each class and each effort score classification.

Effort score / Class	a	b
1	23%	9%
2+3	16%	28%

Table 14: P-values calculated with pooled t statistic for classes 1 and 2+3 using effort scores (a) and (b)

The numbers presented in Table 14 show some evidence that subjects who score above 60% with respect to effort score (a) have on average better performance than those who score less. The results using effort score (b) on the other hand are inconsistent. They are better for class 1 but worse for class 2+3 in comparison to effort score (a).

4.4 Hypothesis 4: debugging counts less towards performance than coding

The activity-weighted effort metric discussed in subsection 3.3.1 depends on the weights assigned to each of the coding and debugging activities. Since this is the first attempt in trying to quantify the effect of these activities on productivity, we don't have any reason to pick these weights. For this reason, we tried a range of values for the weights in which debugging has a lower effect than coding. In Table 15 and Table 16, the different activity-weighted effort calculations for class 1 are shown in columns beyond column 2. Each column is titled with the corresponding a/b ratio of the activity weights that were used to calculate its values, where a is the coding activity weight and b is the debugging activity weight.

Subject	Performance score	0.75/ 0.25	0.9/ 0.1	0.05/ 0.95	1/ 0	1.05/- 0.05	1.1/- 0.1	1.25/- 0.25	1.5/- 0.5
66	0.448	6.71	5.84	5.10	4.08	3.22	0.70	0.70	-3.50
68	0.695	7.29	5.57	4.20	2.18	0.66	-4.16	-4.16	-12.31
69	0.843	1.07	0.96	0.86	0.74	0.63	0.30	0.30	-0.25
70	7.243	12.17	11.45	11.19	9.66	9.14	7.25	7.25	3.97
71	0.572	3.36	3.44	3.53	3.61	3.69	3.93	3.93	4.34
78	5.069	7.15	6.96	7.09	6.59	6.46	6.00	6.00	5.28
81	10.919	1.79	1.62	1.47	1.29	1.12	0.62	0.62	-0.20
Correlation		-0.02	0.09	0.15	0.20	0.25	0.30	0.37	0.36

Table 15: Activity weighted effort and correlation with performance for class 1

Subject	Performance score	0.75/ 0.25	0.9/ 0.1	0.05/ 0.95	1/ 0	1.05/- 0.05	1.1/- 0.1	1.25/- 0.25	1.5/- 0.5
301	9.66	4.45	3.68	3.41	3.20	2.88	2.63	1.86	0.58
302	2286.32	2.02	0.77	0.35	0.1	-0.35	-0.76	-2.01	-4.10
304	10.66	6.24	4.89	4.44	4.13	3.44	3.04	1.73	-0.46
308	4730.15	3.61	2.83	2.57	2.35	2.06	1.81	1.05	-0.23
219	13.23	6.35	4.81	4.29	3.92	3.26	2.78	1.28	-1.16
221	18.31	5.25	4.03	3.63	3.27	2.82	2.43	1.23	-0.77
223	18.30	5.14	3.64	3.14	2.73	2.15	1.67	0.21	-2.22
Correlation		0.33	0.22	0.18	0.14	0.09	0.04	-0.10	-0.28

Table 16: Activity weighted effort and correlation with performance for class 2+3

As you can see, we experimented with negative weights for debugging, basically meaning that time spent debugging doesn't only contribute less than time spent coding, but also decreases the effectiveness of time spent coding. For our hypothesis to be verified, the correlation must be negative and large, indicating that counting debugging as less powerful than coding accompanies improved performance. However, correlation variation in class 1 is going in the opposite direction of class 2's, and neither supports our hypothesis: while in class 1, the correlation starts negative and small and ends positive and medium, in class 2 it starts positive and medium and ends negative and small.

4.5 Hypothesis 5: Some of the best novices perform as well as experts

We ran the CG-NAS benchmark with exactly the same configurations as used with the subject programs. Its performance score for input set S was 0.21 (needed for comparison against Class 1 subjects) and its final performance score (on all benchmark was 90.24. We found two subjects from class 2 who did better than the benchmark (301 and 304) and all subjects from class 3 actually did better than the benchmark.

Chapter 5: Analysis

5.1 Interpretation of results

Our hypothesis generation process was based on refining initial hypotheses, therefore it is not a surprise that hypothesis 1 tested out negatively: the total amount of effort doesn't correlate with performance. Looking at Figure 12 from class 1, we can see the reason for that: subjects 69 and 71 spent the least amount of effort but their program performance was among the best. Subject 70 on the other hand spent a significant amount effort but suffered with performance. Because we only have a small set of data available, we can't be sure why this is so. One possibility is that subjects 69 and 71 are exceptionally bright and so they didn't have trouble, and subject 70 was exceptionally weak. Another possibility is that subjects 69 and 71 have more experience than their classmates, and subject 70 has no experience with programming for example: unfortunately we can't verify that because subjects didn't submit their experience and background questionnaires for this class. Yet another possibility is that subjects 69 and 70 did in fact spend a significant amount of time, but this time was not captured because they did some of their development on a non-instrumented machine. A fifth possibility is that our hypothesis is not quite right, that more effort doesn't correspond to better performance, but rather that effort above a certain threshold corresponds to performance above a certain threshold. This reasoning led to hypothesis 2.

Before discussing the results from the second hypothesis, there are a couple of remarks to make about class 2+3 with respect to hypothesis 1. From Figure 13, we can immediately see two outliers in terms of performance, subjects 302 and 308. These are another example of subjects who spend a significant amount of time and still suffer with performance. Figure 15 shows the remaining five subjects alone, in order to clearly see how the points are dispersed.

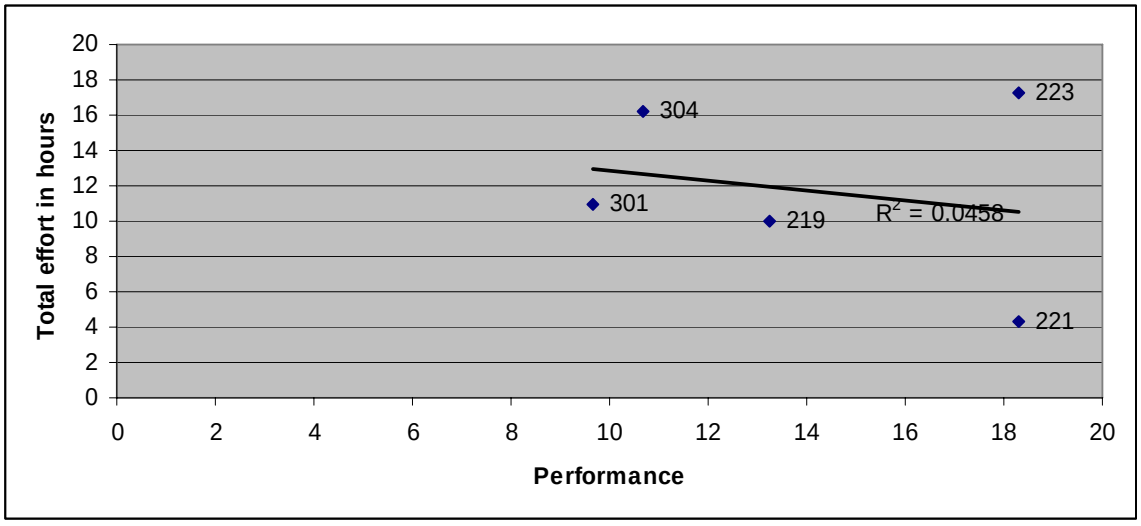

Figure 15: Scatter plot and trendline of 5 subjects in class 2+3

The correlation for these 5 points alone is -0.21, still a weak correlation with the outliers eliminated. The reason is similar to the possible reasons presented for class 1. Take subjects 221 and 223, they both have same performance results, even though subject 223 spent more than four times the amount of effort that subject 221 spent.

Results from the second hypothesis suggest there is no significant difference on average between novices who spend more than 15 hours working on a problem and those who spend less. From tables 6 and 7 we see that the outliers from the first hypothesis are still outlier for this one. Therefore, we turn to the daily distribution of effort; perhaps these novices who are spending a lot of time and still suffering with performance are not spending their time "well".

Results from the third hypothesis test are promising: we have a strong negative correlation between the effort scores and the performance scores, indicating that a high level of effort correlates with good performance. The correlation using effort score a is stronger, especially in class 2+3 than the correlation with effort score b, which gives us an indication that working too hard (we've seen subjects who worked more than 12 hours a day!) doesn't improve the productivity: productivity either remains the same or decreases. Taking a look at our outliers in this view, we see that subjects 69 and 71, who had the least amount of total effort, have high effort scores (Table 8). So although they spent a small amount of time in total, they divided their time wisely and worked regularly without taking long breaks. The opposite can be seen for subject 70, who had spent a significant amount of effort but without performance results. The effort score for this subject is among the lowest scores, meaning he or she didn't work regularly and/or took long breaks... The same goes for subjects 302 and 308 from class 2+3 (Table 9).

Our last attempt at explaining the differences was the investigation of the effect of the amount of coding and debugging performed on the performance. The results from the two classes were contradictory to each other, and neither one strongly

supported our hypothesis. Studying the numbers in table 15 and 16 doesn't indicate that the result is skewed by one or two outliers. Therefore, we have no evidence supporting hypothesis 4. One possible reason for this negative test is that we did not determine the objective of the debugging phase: was the programmer trying to fix a functional problem, a compiling error, or a performance defect [3]?

Our last hypothesis aimed at characterizing the novice performance patterns in order to compare them with expert performance patterns. We found some subjects who did better than the benchmark. We noticed however that those subjects are confined to class 2+3. This can be justified by Figure 7: none of the student code from class 1 actually speeds up over more processors. The main reason for this is that we could only run the student code for this class on the smallest size provided by the benchmark; therefore the communication overhead is too large given the small amount of computation. The benchmark code, on the other hand, does achieve speedup. Therefore, none of the subjects from class 1 scored better than the benchmark. We can see however that certain students (66, 68, and 69) did do better than the benchmark in terms of time to run: even though the execution times kept increasing from 1 to 16 processors, the time on 16 processors was slightly less than the benchmark time on 16 processors. However, when we continued testing to up to 64 processors as shown in Figure 16, the performance for these students becomes inferior to the benchmark even in terms of pure execution time.

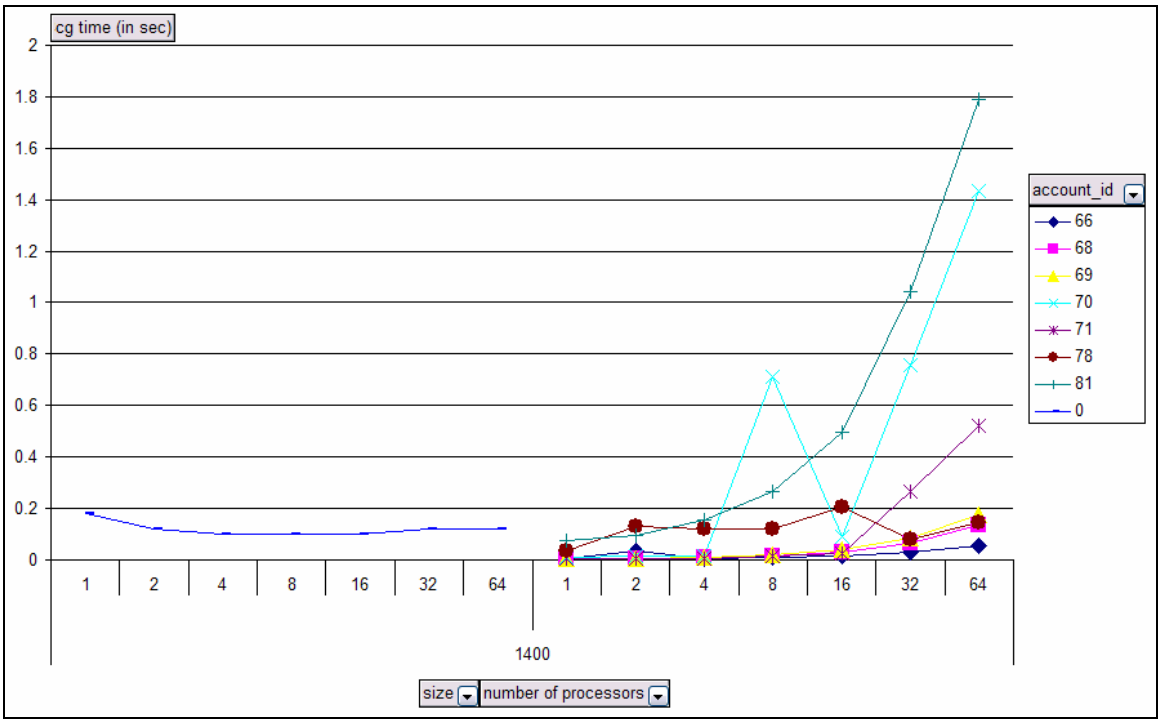

Figure 16: Execution times of subjects in class 1 and the benchmark on input set S. Subject 0 indicates the benchmark

5.2 Threats to validity

The fact that our studies were run across classroom environments poses threats to validity that must be considered when interpreting the results. We eliminated as many differences between the classroom as possible, by choosing the same assignments and running the student programs on the same machine. Some factors however remained outside of our control. For example, there is a slight difference between the requirements of the class 1 assignment and class 2+3's assignment: class 1 assignment requires the implementation of two data generators, in addition to the CG routine implementation. Apart from the extra effort that this must have required from subjects in class 1, this was one of the main reasons why we were limited to only testing class 1 programs on one benchmark input set.

In terms of internal validity, we faced two main threats to validity: instrumentation and selection. As discussed in the methodology section, our effort data is collected through automated instrumentation. This instrumentation was installed on the parallel machine the students were using. Since it's highly unlikely that students gained access to a machine other than the one provided for the class, we can be sure that we have captured all activities related to parallel programming. However, it is possible that subjects tackled the problem by first writing a serial solution, and then parallelizing it. It is then possible that we have underestimated the serial effort, since they could have done the serial programming on any non-instrumented computer. Another problem with the automated instrumentation is that it doesn't capture the time that was spent thinking about the problem. This could have been accounted for by asking the subjects to log their activities, including thinking about the problem, and submit their effort logs. Although this has been done in certain classroom studies, this wasn't part of the experimental setup in the classes that we are studying.

Concerning selection, the subjects come from different backgrounds and different experience levels. At the beginning of every classroom study, a background questionnaire is distributed to the subjects, containing questions about their majors, whether they are graduate students, what experience they have in software development in general, and in parallel software development in particular… Unfortunately, not all subjects submit these forms, and some don't respond to all the questions… therefore, we have neither background nor experience data from class 1, but we have some incomplete data from class 2+3, as shown in Table 17.

Subject	Graduate student	Current major	Years of software development experience
301	Yes	Computer science	4
302	Yes	Computer science	0
304	Yes	Mechanical engineering	0
308	Yes	Chemical engineering	3
219	Yes	Electrical engineering and computer science	NA
221	No (senior).	Electrical engineering and computer science	NA
223	Yes	Biology	NA

Table 17: Available background and experience data from class 2+3 subjects

What we are certain of is that all subjects are new to parallel programming. All subjects are graduate students, except one (221) who is a senior. Therefore, we don't see much difference between the subjects from this perspective. Their backgrounds vary between computer science, science and engineering. They also vary in software development experience. However, due to the incompleteness of the data, we couldn't measure the effect of experience or background within our subject pool. For example, we already noted that subjects 302 and 308 were outliers with high amounts of effort but poor performance. This could be due to the fact that subject 302 has no experience in programming, or that subject 308 doesn't have a computer science background. We have also noted that subjects 221 and 223 have similar performance scores although subject 221 spends significantly less time. This difference might be due to the fact that subject 221 is a computer science major and hence more used to programming concepts and methods.

The main external validity threat we face is the difference between the goals of novices and those of experts and practitioners. While students work on a particular problem in order to run it as fast as possible and make it run even faster on increasing numbers of processors, practitioners typically want to complete a run on the available resources. Students have more flexibility in the queuing systems that they use, especially if they're dedicated to students. Therefore their queue wait time is short and their run times are also expected to be short due to the relative simplicity of their problems compared to the problems that practitioners try to solve. For practitioners, they are given a set of resources and some allocation time, which they have to work according to. So, they are less concerned with speedup than getting the job done with the given constraints. Therefore, to extend our study and our results to the domain of experts and practitioners, a different definition for the performance metric is needed, one that suits their incentive and goal.

Chapter 6: Conclusion

6.1 Summary of results

In this thesis, we study the effect of human programmer variations on the performance of the code they produce. Although it's widely accepted that available programming models and languages make it difficult for scientists to write solutions for their problems and that improving the performance of a certain parallel program requires a significant amount of effort, few studies have attempted to quantify the relationship between the effort spent and performance produced. We have qualitatively characterized effort spent by novices and their code performance and quantitatively studied the relationship between the two.

6.1.1 Qualitative characterization of effort for novices

We have identified two dominant patterns in the daily distribution of effort among novices: the slow and steady subject spends similar amounts of effort daily and rarely takes a break; the fast and furious subject spends great amounts of effort some days and minuscule amounts of effort on others, with long breaks.

6.1.2 Qualitative characterization of performance for novices

The performance of code written by novices varies on two dimensions: execution time and scalability. For a significant data size, subjects achieve speedup, i.e. execution time is reduced as the number of processors increases. When the data size is too small, however, speedup is harder to obtain to the point that many of the

codes lose their scalability properties. Occasionally, execution time suddenly increases on a certain number of processors, but returns to normal afterwards. In general, we noticed that subjects with better execution time on one processor have better execution times on multiple processors. This allowed us to simplify our performance scoring function. We also found a few subjects who perform better than the benchmark code.

6.1.3 Relationship between effort and performance for novices

We investigated the relationship between effort and performance from various perspectives:

- Total effort did not correlate with performance
- Effort scores, indicating the proximity of subjects to the slow and steady approach, correlated with performance
- Effort spent debugging doesn't count less than effort spent coding towards performance

6.2 Future work

In order to confirm our results, the study executed needs to be duplicated. Duplication will also allow us to expand our sample size, provided certain conditions are preserved, like the problem assigned, the assignment requirement, the time the subjects have to solve the problem, the programming model used… This will allow researchers to aggregate data easily and minimize threats to validity. As we previously discussed, in our small samples, a couple of outliers were able to

negatively skew the results. Increasing the sample size will make it clear if we got a false negative due to the small sample size, or if the hypothesis is not applicable.

A second avenue of research is the study of the effect of other programmer variations, such as programmer background and serial programming experience, on performance. The interaction of the different factors is also an interesting area of study. For example, does a computer science student spend less time to attain a certain performance level than a science or engineering student? Or does a computer science student spend less time debugging the code? From the activity perspective, as more is uncovered in the workflow of developers in the HPC domain, the effect of different activities on the performance can be studied.

One of the most important extensions to this research is studying experts in the HPC domain. As we already noted, the work environments and the goals of experts are different than those of novices. Therefore metrics and hypotheses need to be adjusted to their situations. In addition, experts solve more complex problems than novices. Studying experts will therefore surely present more questions to the research community to answer.

Appendix A

In this appendix we will elaborate on two important aspects of the APMS tool: the instrumentation module and the adaptation layer language.

A.1 Instrumentation module

The instrumentation module in APMS handles the necessary code changes needed in order to collect the performance measurements. The PAPI library allows the user to specify a list of events to monitor, such as level 1 data cache accesses and misses, total cycles, floating point operations and its interface provides a list of high level and low level function calls, which are used to start, stop and read the counters for the specified list of events. The high level and low level functions differ in the degree of flexibility they provide and overhead they require. In general, the low level functions are meant for experienced developers wanting fine-grained measurement and control of the library. The low level interface can be used in conjunction with the high level interface[15]. The instrumentation module essentially inserts the function calls in their appropriate positions in the code.

In order to allow users of APMS to customize instrumentation to their own needs, we have encapsulated the PAPI library calls into three functions, shown below:

```
1  int SIRONIerror(int num)
2  {       if ( SIRONIretval < PAPI_OK)
3    {         printf("error: %d %d\n", num, SIRONIretval);
4           exit(1);  }
5         return 0; }
```

Function 1: SIRONIerror()

```
1   int SIRONIinit()
2   {
3     SIRONIretval = PAPI_library_init(PAPI_VER_CURRENT);
4   {
5     SIRONIretval = PAPI_library_init(PAPI_VER_CURRENT);
6     if (SIRONIretval != PAPI_VER_CURRENT && SIRONIretval > 0) {
7       fprintf(stdout,"PAPI library version mismatch\n");
8       exit(1); }
9     if (SIRONIretval < 0)
10    {
11            printf ("1\n");
12            printf("%s\n",PAPI_strerror(SIRONIretval));
13            perror(" ");
14            exit(1);
15    }
16    SIRONIretval = PAPI_is_initialized();
17    if (SIRONIretval == PAPI_NOT_INITED)
18    {
19      printf ("papi  not initialized!!!! \n");
20          exit(1);
21    }
      SIRONIretval = PAPI_flips(&SIRONIrtime1, &SIRONIptime1, &SIRONIflpins1,
22  &SIRONImflips1);
24    SIRONIerror(4);
27    return 0;
28  }
```

Function 2: SIRONIinit()

```
1   int SIRONIfin()
2   {
      SIRONIretval = PAPI_flips(&SIRONIrtime1, &SIRONIptime1, &SIRONIflpins1,
3   &SIRONImflips1);
4
5     SIRONIerror(6);
6     int SIRONIrank;
7   int SIRONIsize;
8     MPI_Comm_rank(MPI_COMM_WORLD, &SIRONIrank);
9
10    char SIRONIfile [1024];
11          sprintf (SIRONIfile, "%dMeasurementResults.txt", SIRONIrank);
12          FILE * SIRONImetrics = fopen (SIRONIfile, "a");
13          fprintf (SIRONImetrics,"%f %f %f %lld \n", SIRONIrtime1, SIRONIptime\
14  1, SIRONImflips1, SIRONIflpins1);
15                MPI_Barrier(MPI_COMM_WORLD);
16                close(SIRONImetrics);
17  }
```

Function 3: SIRONIfin()

These functions are defined in a header file “SIRONIheaderf2.h”. The SIRONIerror() function is an error handling function which is not generally called directly. Instead, it is used in the other two functions. The SIRONIinit() function call is inserted in the user code following the call to MPI_init(). Therefore, this function should contain all PAPI library initialization code, event set definitions, and any PAPI library calls that start the hardware counters. On the other hand, the SIRONIfin() function call is inserted directly before the MPI_finalize() call. Therefore, this function should contain all PAPI library calls that stop the hardware counters and reads their values. In our case, the SIRONIinit() function initializes the PAPI library, and calls the high level function PAPI_flips(), which starts counters for the wall time, process time, and number of instructions, and the SIRONIfin() function stops these counters, reads their values and writes them to a file.

Users of APMS can freely modify the code in the header file to suit their needs, as long as they retain the prototypes of the 3 function described above and they follow the coding constraints of the PAPI library.

A.2 Adaptation layer language

We now discuss in detail the use of our adaptation layer language to solve the problem of non-uniform input formats across class assignments. As mentioned in subsection 3.1.1, the user executes the following three steps:

1. Specifying the value for each parameter: When users choose the codes they want to run, the parameters for the corresponding problems are shown. For instance, if the user chooses to run programs that implement the game of life problem, the

grid size, number of iterations and initial configuration will appear. For each parameter, the user can enter multiple values separated by commas. This instructs APMS to generate multiple input sets, using all possible combinations of the specified parameters and to run the selected programs using each one of these sets. Figure 17 shows an example where the game of life programs are selected and Figure 18 shows the corresponding input parameters shown. In this example we see that the programs are to be run on 2, 4, and 8 processors. We also see that the number of iterations is specified as a comma-separated list (100, 200). In this case, APMS will run two cases: one case with the specified input file, grid size and 100 iterations and the other with the same specified input file, and same grid size but 200 iterations. Each one of these cases will be run on 2, 4, and 8 processors respectively.

2. Specifying for each parameter whether it should be passed on the command-line or through a file. The “grid_size” and “iterations” parameters are specified as a command line parameter, while the “start_grid file” configuration is passed as an input file.

3. Specifying the order for the command line parameters and input file parameters: space is provided for each category of the parameters (command line and input file). The order is specified using the parameter names as shown in the parameter list. As shown in Figure 19, if the required order for the game of life is grid size, followed by number of iterations, and these parameters are to be passed through the command line, the order will be specified as: “#grid_size# #iterations#”. The initial configuration is usually passed through an input file. It is then specified as

"#startgrid_file#". We also noticed that many programs expect the first line of the file to contain the number of lines in the file; therefore we implement a simple function which counts the lines that exist in the file and inserts it. To use this function in this manner, the syntax is:

"#startgrid_file.CountLines#

#startgrid_file#"

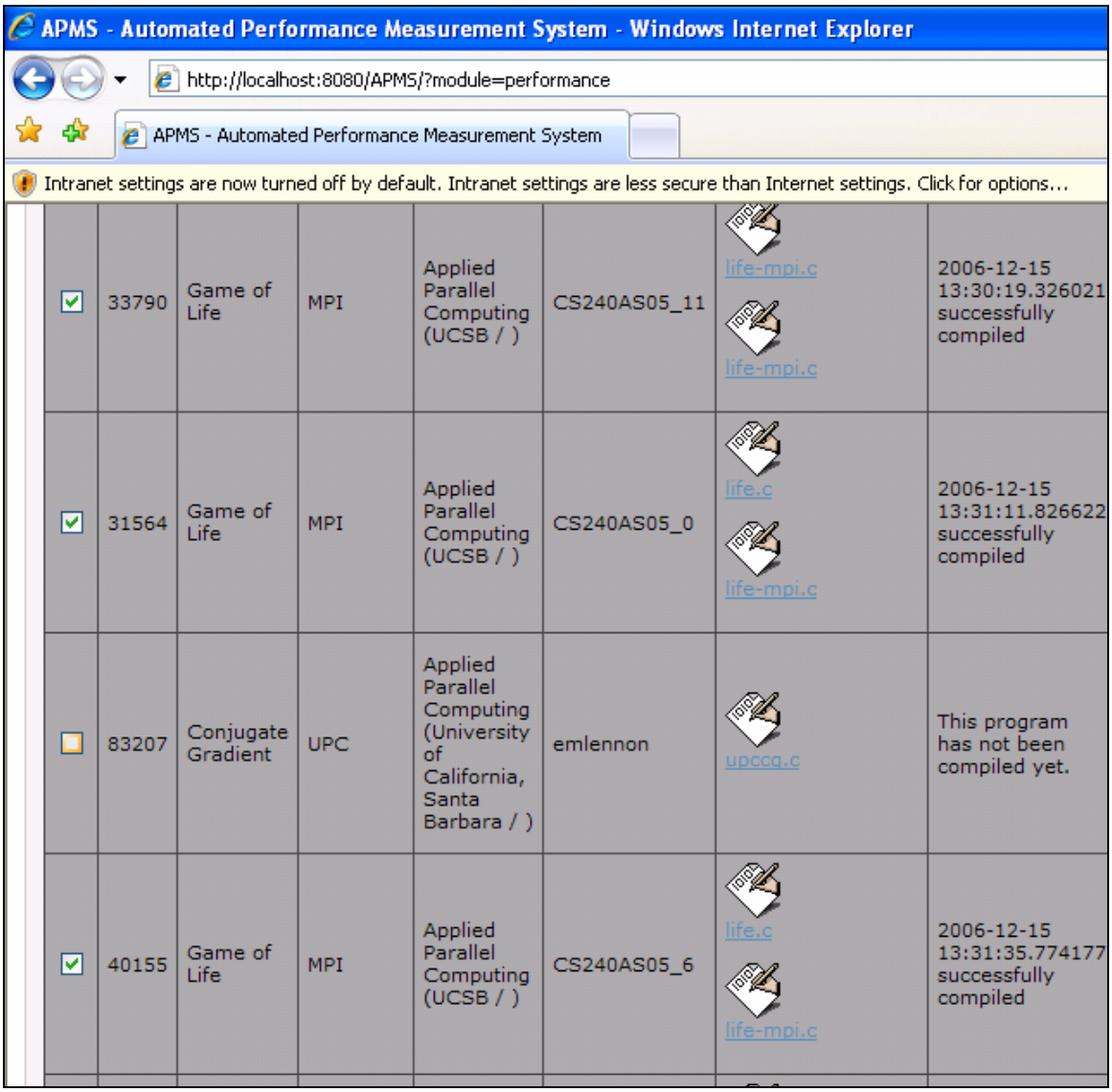

Figure 17: Source code selection page

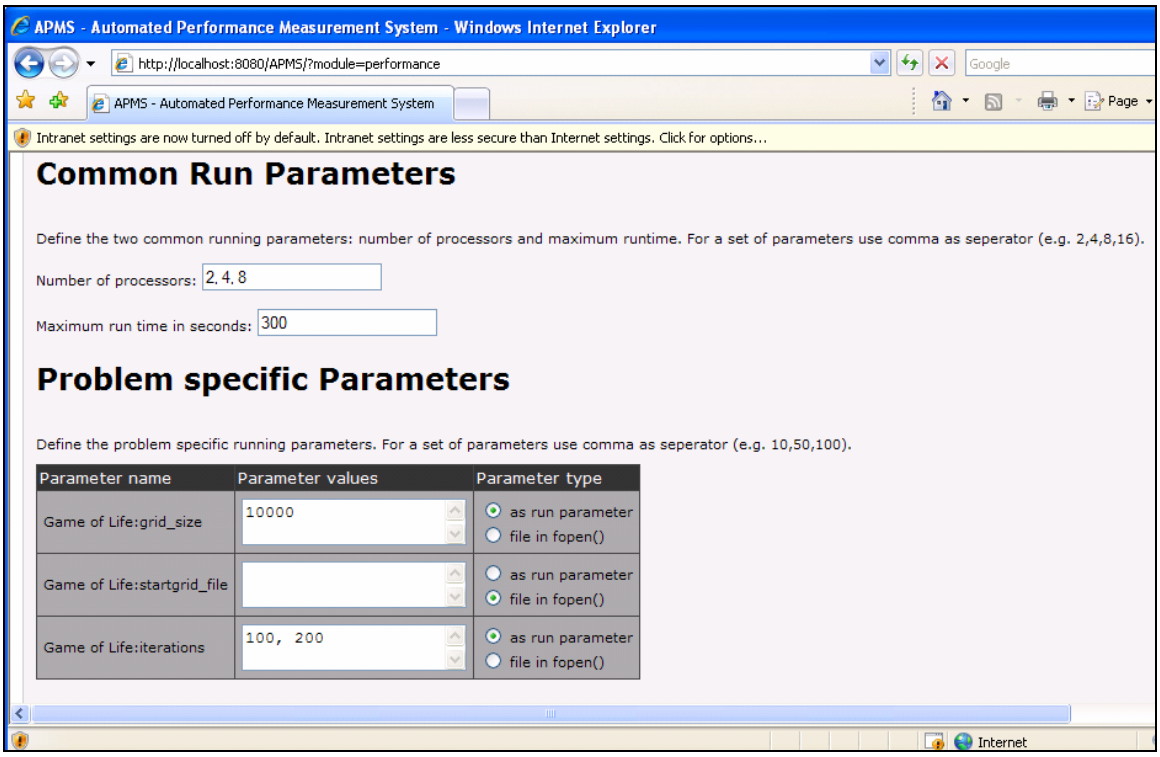

Figure 18: Parameter value specification

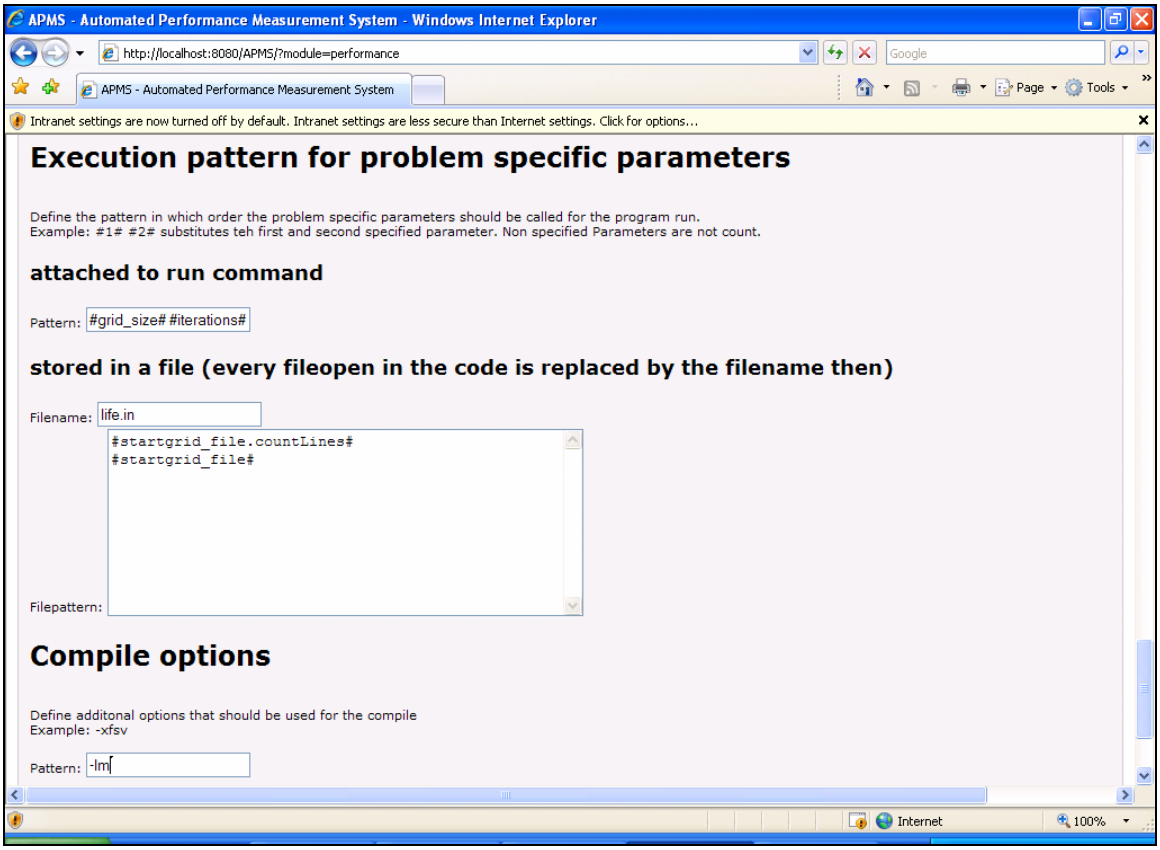

Figure 19: Parameter order specification

Bibliography

[1] Jack Dongarra, Ian Foster, Geoffrey C. Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *The Sourcebook of Parallel Computing*. Morgan Kaufmann, 2002.

[2] Lorin Hochstein. *Development of an Empirical Approach to Building Domain-specific Knowledge Applied to High-end Computing*. PhD thesis, University of Maryland, College Park, 2006.

[3] Taiga Nakamura. *Recurring Software Defects in High End Computing*. PhD thesis, University of Maryland, College Park, 2007.

[4] Cherri M. Pancake and Curtis Cook. What Users Need in Parallel Tool Support: Survey Results and Analysis. *In SHPCC '94: Proceedings of the Scalable High-Performance Computing Conference*, pages 40-47, Knoxville, TN, US, 1994.

[5] Duane Szafron and Jonathan Schaeffer. An Experiment to Measure the Usability of Parallel Programming Systems. *Concurrency: Practice and Experience* 8(2), pages 147-166, 1996.

[6] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. *In SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Konstantin Berlin, Jun Huan, Mary Jacob, Garima Kochhar, Jan Prins, Bill Pugh, P. Sadayappan, Jaime Spacco, and Chau-Wen Tseng. Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures. *In 16th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, October 2003.

[8] Jean-Yves Berthou and Eric Fayolle. Comparing OpenMP, HPF, and MPI Programming: A Study Case. *International Journal of High Performance Computing Applications*, 15(3): pages 297–309, 2001.

[9] Don Morton, K.Wang, and D. O. Ogbe. Lessons Learned in Porting Fortran/PVM Code to the CrayT3D. *IEEE Parallel & Distributed Technology: Systems & Technology*, 3(1): pages 4–11, 1995.

[10] Andreas Rodman and Mats Brorsson. Programming Effort vs. Performance with a Hybrid Programming Model for Distributed Memory Parallel Architectures. *In Euro-Par '99 parallel processing: 5th International Euro-Par Conference*, pages 888-898, 1999

[11] Lorin Hochstein, Taiga Nakamura, Forrest Shull, Nico Zazworka,, Victor R. Basili, and Marvin V. Zelkowitz. An Environment for Conducting Families of Software Engineering Experiments. *To appear in: Advances in Computers,* Elsevier, Boston MA vol.74, 2008

[12] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Qin Zhang, Aaron Kagawa and Takuya Yamashita. Practical Automated Process and Product Metric Collection and Analysis in a Classroom Setting: Lessons Learned from

Hackystat-UH. *In International Symposium on Empirical Software Engineering*, Los Angeles, California, August, 2004

[13] Christopher Ackermann and Nico Zazworka. CodeVizard: Combining Abstraction and Detail for Reasoning about Software Evolution.

[14] Rola Alameh, Nico Zazworka and Jeffrey K. Hollingswoth. Performance Measurement of Novice HPC Programmers' Code. *In ICSE '07: Proceedings of Third International Workshop on Software Engineering for High Performance Computing Applications*, Minneapolis, MN, May 2007.

[15] Shirley Browne, Jack Dongarra, N. Garner, G. Ho, and Philip Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High-Performance Computing Applications* 14(3): pages 189-204, fall 2000

[16] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *International Journal of Supercomputer Applications* 5(3), pages 66-73, 1991.

[17] Robert W. Numrich, Lorin Hochstein, and Victor R. Basili. A Metric Space for Productivity Measurement in Software Development. *In ICSE '05: Proceeding of Second International Workshop on Software Engineering for High Performance Computing System Applications*, pages 13-16, St. Louis, MO, May 2005.

[18] Lorin Hochstein, Victor R. Basili, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, and Jeffrey Carver. Combining Self-Reported and Automatic Data to Improve Programming Effort Measurement. *In FSE '05, Foundations of Software Engineering,* Lisbon, Portugal, September 2005.

[19] Sartaj Sahni and Venkat Thanvantri. Performance Metrics: Keeping the Focus on Runtime. *IEEE Parallel and Distributed Technology*, 4(1): pages 43-56, Spring 1996.

[20] Rudolph J. Freund and William J. Wilson. *Statistical Methods*. Academic Press, Revised Edition, 1997.