2016-01-01

An Evaluation Framework For Scientific Programming Productivity

W.K. Umayanganie Munipala Munipala
University of Texas at El Paso, umunipala@miners.utep.edu

AN EVALUATION FRAMEWORK FOR SCIENTIFIC PROGRAMMING PRODUCTIVITY

W. K. UMAYANGANIE MUNIPALA

Master's Program in Computational Science

APPROVED:

Shirley V. Moore, Ph.D., Chair

Ann Gates, Ph.D.

Raymond Rumpf, Ph.D.

Charles Ambler, Ph.D.
Dean of the Graduate School

AN EVALUATION FRAMEWORK FOR SCIENTIFIC PROGRAMMING PRODUCTIVITY

by

W.K. UMAYANGANIE MUNIPALA, BSc

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Computational Science Program

THE UNIVERSITY OF TEXAS AT EL PASO

May 2016

Acknowledgements

This work was partially funded by the Department of Energy Office of Advanced Scientific Computing Research under grant no. DE-SC0006733. We would like to acknowledge use of the Stampede supercomputer at Texas Advanced Computing Center, which is funded by the National Science Foundation. We would also like to thank the members of the Fall 2015 CPS 5401 and Spring 2016 CPS 5310 courses and the lab instructor for those courses, Dr. Natasha Sharma, for helping with the case studies.

Abstract

Substantial time is spent on building, optimizing and maintaining large-scale software that is run on supercomputers. However, little has been done to utilize overall resources efficiently when it comes to including expensive human resources. The community is beginning to acknowledge that optimizing the hardware performance such as speed and memory bottlenecks contributes less to the overall productivity than does the development lifecycle of high-performance scientific applications. Researchers are beginning to look at overall scientific workflows for high performance computing. Scientific programming productivity is measured by time and effort required to develop, configure, and maintain a simulation experiment and its constituent parts, together with the time to get to the solution when the programs are is executed. There is no systematic framework by means of which scientific programming productivity of the available tools can be evaluated. We propose an evaluation approach that compares recorded novice programming workflows to an expert workflow to identify productivity bottlenecks and suboptimal paths. Based on a set of predefined criteria we can evaluate both short-term and long-term productivity criteria. We use these results to suggest improvements to the programming environment or tools. We give preliminary results from applying this approach to two case studies involving the use of numerical libraries.

Table of Contents

List of Tables

List of Figure

s

Chapter 1: Introduction

A lot of time is spent on building, optimizing and maintaining large-scale software that is run on supercomputers. But little has been done to utilize overall resources efficiently, specifically when it comes to including expensive human resources. Also there is a less time spent on looking at overall scientific programming workflows for high performance computing. The community is acknowledging that optimizing the hardware performance such as speed and memory bottlenecks contributes less to the overall productivity than does the development lifecycle of high-performance scientific applications.

Scientific computing productivity is a quality measure of the process of achieving scientific results on high performance computing systems [1]. It combines the time and effort required to develop, configure, and maintain a scientific simulation and its constituent parts with the time to solution when the simulation is executed.

A typical scientific programming workflow comprises the following tasks:

1. Preprocessing – e.g., preparing input files, meshing, staging data
2. Developing or adapting code
3. Executing one or more simulations
4. Post-processing – e.g., analysis, visualization
5. Repeat

According to [2], scientific programming productivity encompasses:

- Software productivity: time and effort for development, maintenance, and support
- Execution-time productivity: efficiency, time, and costs to run scientific workloads
- Workflow and analysis productivity: experimental design, task coordination, results analysis, validation
- End-to-end productivity: cost in getting from scientific questions to scientific discovery

  Productivity costs are identified in [2] in terms of

  - Human resources for development and re-engineering

- Machine and energy resources
- Utility and correctness of computational results

In our work, we focus on human resources and utility and correctness objectives.

The productivity problem can be viewed as a mathematical optimization problem with constraints and objective functions. The exact constraints and objective functions depend on the relative costs of resources and on the user's goals and computing environment. Similar problems has been addressed by the software engineering community. But it is not easy to adapt these into the domain constraints in scientific computing. As scientific simulations reach larger scale, data and task decomposition became increasingly complex. The computing environment itself may be variable, for example if the user has a choice between different high performance computing systems to use.

Our approach is to clearly articulate the steps of a given workflow and the criteria by which productivity will be evaluated. We define workflow as the sequence of steps a user carries out to accomplish a programming task. We observe and measure the performance of both an expert user and the target users in carrying out the workflow. We analyze the results to determine bottlenecks and wrong paths in the user workflows, and we use the results to try to optimize the workflow.

The specific contributions of this thesis are the following:

1. We propose a generally applicable methodology for evaluating scientific programming productivity according to specified criteria.
2. We present a graphical representation of workflows that enables comparison between expert and novice traversals.
3. We show how our methodology can help identify bottlenecks and wrong paths and lead to suggestions for improving programming productivity.

Chapter 2: Related Research

2.1 Software Engineering Approach to Scientific programming Productivity

According to [3], scientific computing has a growing problem with end-to-end productivity and observed development problems are the following:

- Increasingly long and expensive development
- Higher risk of failure
- Growing maintenance costs
- Increasing difficulty of porting codes to next generation machines

The authors maintain that scientific computing productivity currently depends on multidisciplinary experts optimizing parallel code by hand. The main finding was that there exists an expertise gap between computing experts and domain scientists, as exemplified by the following aspects:

- There are vanishingly few individuals with the needed skills in scientific domain, programming languages, and hardware.
- Training takes years.
- Once skills are acquired, they are often not portable.

  According to [3], the main human resource intensive tasks are

- Porting and modifying existing parallel code
- Developing correct scientific programs
- Serial optimization and tuning
- Code parallelization and scaling

Our current research focuses on analyzing the first two tasks above with respect to productivity.

The authors maintain that the key to improving scientific programming productivity is to reduce dependence on multidisciplinary experts and increase abstraction and automation by

- Providing computational abstractions reflecting the science and math of the problem domain
- Providing hardware-independent abstractions for tuning and parallelization
- Automating mapping of abstractions to hardware

One way to achieve abstraction and automation is through use of scientific libraries. Our work focuses on evaluating the productivity of scientific computing workflows that involve the use of parallel scientific libraries.

2.2. Data Collection and Analysis Methods

Lorin et.al present the idea of combining self-reported and automated data to improve programming effort measurement in [4]. Automated data captures accurate 'typed' data processing time while observation or self-reported data captures time spent on researcher insight. Incorrect measurement can introduce unexpected bias and lead to incorrect conclusions. They have observed from a set of pilot studies that the self-reported data directly from the users were inconsistent compared to the automated data collected using tools. This has motivated them to have a passive observer reporting the programmer time in order to obtain an accurate measure. They also changed their web based interface to paper based activity log in which individual users entered start and stop times, hoping to eliminate the inconsistencies introduced, such as irritating the user when they had to enter every compilation step even when they had insignificant syntax errors. Their second study concludes that more precise logs improve accuracy. They come to the conclusion that automatically collected data can be augmented by self-reported data but the data provided by users can vary according to the user. Indirect methods that are easier to obtain, can be used if there are criteria that correlate well with effort such as lines of code number of defects.

Personal Software Process (PSP) [5] is a framework that guides software engineers for achieving better productivity for the software process, such as writing requirements, running tests, defining processes, and repairing defects. The authors provide a course and a textbook concentrating on individual user education showing them how to plan and track their work. The

PSP aims to show engineers how to manage quality from the beginning of the job, how to analyze the results of each job, and how to use the results to improve the process for the next project and track performance against these predefined goals. The framework provides a planning script that guides the user's work. They record their time and defect data which they self-summarize from the logs, measure the program size, and enter these data in the plan summary form which is delivered with the finished product. Their process goal is to improve quality of work by producing zero-defect code within a planned schedule and costs, to try to get rid of traditional test-and-fix strategy which is time-consuming and costly.

Min Zhang and Lorin Hochstein [6] introduced a method to fit a workflow model into captured data called a software engineering workflow analysis (SEWA). They analyze automatic data captured by the programming environment automatically by a tracing method and build a model that is dependent on eight factors which are coding, chunking, commenting, comparing, converting, computing, connecting and constraining. They summarize low level tasks into events under these categories. Using the above categories they implement a model for programmer activity. They have developed an open source tool to visualize time series data using a tool called ActivityGraph which is used to compute the programmer effort distributed among activities. This tool is introduced for general software development and they apply the same tool to the HPC domain using observed data from two previous case studies conducted by the University of Maryland using Hackstat and UMDInst to log the steps of the small HPC problems. They use the compile log files, editor log files and shell log files to grab the information. They categorize the data retrieved via these log files into events of the previous categories and standardized and written into spreadsheets. Using a graphical tool for visualizing programmer's changes in a chunk, they develop the heuristics based on examination of the diffs. Most of the preprocessing has been done manually, by observation. They categorize debugging and testing into one category claiming it is difficult to distinguish the different purposes of execution (similar to the problems we encountered when we drew the workflow diagrams for program execution). They

iterate through the SEWA process and refine the model with steps such as identifying and labeling chunks with new activities according to their heuristics. Finally they compare the SEWA model with the observed times to prove that their model is consistent. They state that activities such as 'thinking' can be only indicated in the observation data since there is no way to identify anything that does not involve typing on a keyboard using the too.

A pragmatic methodology for the design and evaluation of scientific workflows in research oriented web applications is presented in [7]. The authors carry out an in-depth usability study of their CoGe web application that provides a set of tools for exploring genomic datasets. Their method demonstrates how to identify bottlenecks in multi-step tasks and how to analyze bottlenecks. The visualization system introduced in this paper is used to analyze complex tasks associated with scientific workflows. This analysis leads to suggestions for improvements in the current implementation of their CoGe web application. They have carried out a follow-up study and confirmed their suggestions improved the user navigation in the web application. We have adopted a graphical representation of the workflow steps and compare novice users' paths with the expert user path. However, in contrast to [7], we assign metrics to the paths rather than only determining whether the novice user path deviates from the expert user path. Whereas this framework focuses mostly on execution performance, our evaluation framework focuses mostly on human factors and on code portability and long-term maintainability.

Studies of scientific programmer productivity are reported in [8, 9,10]. These studies all have one metric to measure programmer productivity, namely the amount of time spent carrying out tasks. This metric focuses on short term productivity. In contrast, our evaluation framework can support additional evaluation criteria, such as productivity and maintainability of the code, which affect productivity in the long term.

2.3 Lighthouse

Lighthouse is a framework for creating, maintaining, and using a taxonomy of available software that can be used to build highly-optimized linear algebra computations [11, 12]. It aims

to aid developers seeking to learn available tools for their programming tasks and to help them fit various parts together. Lighthouse assists scientists to explore the available libraries and apply the corresponding numerical software that suits their problem best. Lighthouse targets both developer and application's productivity enhancement by providing an environment that facilitates the user's selection of tools for dense and sparse linear algebra computations.

Lighthouse attempts to combine expert knowledge, machine learning-based classification of existing numerical software collections, and automated code generation and tuning to enable users to discover and apply the best available numerical software out of a several libraries covering a broad space of sequential and parallel solution methods for dense and sparse linear algebra. An organized classification of software as well as a variety of code generation and optimization capabilities and information about the code in the form of automatically extracted documentation is provided for users. Lighthouse uses taxonomy-based search for identifying solution methods with code generation and optimization capabilities to accommodate a variety of different use cases that may arise in HPC software development. They claim to provide an interface that is more accessible and user friendly than the usual HPC tools available for advanced users and to be the first framework that offers a searchable ontology of linear algebra software with code generation and tuning capabilities. They also claim that it provides functionality- and performance-based search of high performance numerical software capabilities with current support for sequential and parallel dense and sparse linear algebra computations provided by the LAPACK, PETSc, and SLEPc libraries. We use Lighthouse in our second case study to evaluate how its functionalities can help users in solving sparse linear systems using PETSc library and give suggestions for improvement.

Chapter 3: Methodology

3.1 Evaluation Framework

In order to formulate an evaluation framework for scientific programming productivity we clearly articulate the steps for the scientific programming workflow. Next we define the criteria by which productivity will be evaluated. Then we observe and measure programming behavior of both expert and novice users. We analyze the results to determine bottlenecks and detect wrong or sub-optimal paths in the novice user workflows. Finally we use the results to try to improve the programming environment.

We set up experiments that involve novice users and an expert user who is used for defining a lower bound for the measurement criteria. We observe and measure programmer behavior for each problem. The users are given the same set of problems under the same conditions. We provide the same set of steps to follow and mark the time spent on every step for each problem. Users are asked to take note of every correct and incorrect step they follow in order to get to their final solution. These notations are used to articulate the workflow diagrams for each user.

For a given case study, productivity is measured by a set of criteria related to the particular scenario. The evaluation criteria are predefined and all of a user's steps are analyzed to figure out how each step contributes to each criterion. Steps followed by a user may negatively or positively affect some of the measurement criteria. After carefully examining all the steps users have taken, a weighted or a numerical value is assigned for each criterion. We define the productivity vector by adding together all the values that come under each criterion. It is possible for one step to affect more than one criterion of the productivity vector. After completing the productivity vector, we compare the vectors for the novice users with those of the expert user to identify efficiency bottlenecks and wrong steps. Steps that diverge from the expert user's path may not necessarily be wrong steps if they do not result in a lower score for a given criterion,

rather they could be alternative valid paths to the same goal. Thus, it is important to analyze not only the paths through the workflows, but also the metric values associated with each path to detect wrong or sub-optimal paths in the user workflows.

Our productivity measurement vector is specific to each problem or case study, but the framework can be generalized by defining different criteria for the vector for other workflow evaluation problems as well.

3.2 Domain Selection

We chose the domain of numerical linear algebra to develop and test our evaluation framework for scientific programming productivity. Scientific problem solving usually requires a background in numerical analysis, high performance computing (HPC) and software engineering. It also typically involves reading documentation (when available) or researching publications outside of the developer's area of expertise. Even though there have been numerous advancements in numerical analysis and HPC libraries, selecting the correct library routines for the specific need of the user is a significantly hard task for the domain scientist. The probability of the user identifying the most portable and efficient library for a given problem is decreasing with the larger number of new versions and libraries added. Linear algebra libraries are among the most used HPC libraries by the domain scientist and these are often the most time-consuming part of scientific applications. Reducing the time for the calculations as well as using the correct version for best portability of the code is one of the most important skills of a HPC user. Since it is too complex and less efficient to use their own algorithms when solving linear algebra problems, domain scientists must know how to use the available libraries out of the vast number available that have been developed and optimized by the experts for decades. Scientists and engineers rely on linear algebra algorithms for solving problems in high-performance computing applications. Most domain scientists lack the in-depth knowledge needed for discovering and applying the most suited libraries that give the most efficient solution to a given problem. One incorrect step can have a snowballing effect on the next steps that could lead to inefficiency.

The general workflow of solving a problem using a numerical library consists of the following steps:

1. Prepare input files
2. Select appropriate method based on the problem characteristics
3. Find appropriate library routines
4. Construct the program
5. Compile, execute and debug the program
6. Validate the results

A key goal of numerical libraries is to achieve performance portability. Performance portability is defined in [13] as the amount of user code that can be compiled for diverse architectures and obtain the same, or nearly the same, performance as an architecture specialized version of that code. Performance portability is achieved by libraries through abstraction. The routines and their functionality remain the same across platforms, and the platform-specific optimizations and parallelization and hidden inside the implementation.

We defined and evaluated programming workflows for two numerical linear algebra problem types. The first case study is solution of a dense linear system using the linear algebra library, LAPACK, and working from an existing code to port it to the Stampede supercomputer. The second case study is the implementation of solving a sparse linear system from a finite element analysis using the PETSc library. We chose to carry out our experiments and evaluation using the Stampede supercomputer. Stampede is one of the most powerful supercomputers in the world. It is comprised of 6400 nodes, 102400 processor cores, 205 TB total memory, 14 PB total and 1.6 PB local storage Funded by the National Science Foundation Grant ACI-1134872 and built in partnership with Intel, Dell and Mellanox, it is located in Texas Advanced Computing Center (TACC). The cluster contains 160 racks of primary compute nodes, each with dual Xeon E5-2680 8-core processors, Xeon Phi coprocessor, and 32 GB RAM and also contains 16 nodes with 32 cores and 1 TB RAM each, 128 "standard" compute nodes with Nvidia Kepler K20 GPUs, and other nodes for I/O (to a Lustre filesystem), login, and cluster management. Currently

it us in the 8th place of the list of the world's top 500 supercomputers at http://www.top500.org/. Stampede provides a peak performance of nearly 10 petaflops (PF), or nearly 10 quadrillion math operations per second.

3.3 User Categorization

We assume that all our users have the basic understanding required for solving domain specific scientific problems. The expert HPC users understand the architecture of high performance computers and how the architecture of high performance computers affects the speed of programs run on the machine. Also how the memory access affects the speed of HPC programs, Amdahl's law for parallel and serial computing, the importance of communication overhead in high performance computing, some of the general types of parallel computers, how different types of problems are best suited for different types of parallel computers, some of the practical aspects of message passing on MIMD machines are the general facts an expert user is expected to be knowledgeable of. Other than the above stated ideas, an expert user who deals with large-scale matrix calculations on HPC machines has to have experience with compiled language programming. Usually most scientific code requires being familiar with languages like Python, FORTRAN or C in addition to a certain understanding of matrix computing and linear algebra libraries. We categorize the users who are lacking parts of the above knowledge as novice users Novice users represent future domain scientists.

3.3 Case Study 1: Porting a Dense Linear Algebra Code to Stampede

3.3.1 Legacy Code Example

We chose an existing C++ code that initializes a small linear system of three equations and three unknowns and calls two LAPACK routines to factor the matrix and then solve the system. The author of the code provided instructions on how to compile and link it using the GNU C++ compiler and the Netlib version of LAPACK. The code is written in an older style and does not user portable LAPACK data types nor does it use the recommended C++ interface. We chose this example because it is a small example of the more general problem of porting a legacy code to a new computer system. We obtained a legacy code that uses LAPACK routines to solve a dense linear system available online from the link https://dynamithead.wordpress.com/2012/06/30/introduction-to-how-to-call-lapack-from-a-cc-program-example-solving-a-system-of-linear-equations/. The author has provided custom header files for LAPACK and given hints about routines he uses for solving the dense linear equation.

The example code uses LAPACK to solve the linear system. LAPACK is one of the most widely used standard software libraries in scientific computing. Domain scientists from many disciplines rely heavily on linear algebra algorithms. The LAPACK Fortran 90 codebase provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, singular value problems and matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) [14]. LAPACK can handle simple and complex dense and banded matrices, but not general sparse matrices. LAPACK effectively make use of cache-based architectures. For C and C++ developers, a portable extended version of LAPACK called LAPACKE is provided [15]. LAPACKE uses native C data representation and allows the user to specify whether matrices are stored in row major or column major order. Row major order is usual for C/C++ codes, and column major order is used in

Fortran codes. Thus, for maximum portability and efficiency, a C/C++ developer should use the LAPACKE extension if it is available.

The LAPACK interface has become a de facto standard for numerical dense linear algebra. Documentation and a reference implementation are provided in the Netlib software repository at http://www.netlib.org/lapack/. Vendors have adopted the interface and implemented versions of the LAPACK routines tuned for their platforms and compilers. For example, on Intel platforms such as Stampede, LAPACK is part of the Intel Math Kernel Library (MKL). The Stampede User Guide has instructions for how to link Fortran and C codes with MKL and refers the user to Intel documentation for MKL. The problem for many users is that the vendor libraries include LAPACK routines, but the library is not called LAPACK, and the link line for the Netlib version of LAPACK will not work for the vendor version. To achieve the most efficient code, developers should link their code with the vendor library. If they link with the refercnce Netlib version of the library using –llapack, their code will still work but it will be inefficient, sometimes by an order of magnitude or more.

For our test case, the instructions provided by the code author are to compile and link the code using

g++ -llpack

To use the MKL library on Stampede, the user should compile and link using

icpc –mkl=sequential –I$TACC_MKL_DIR/include

where icpc is the name for the Intel C++ compiler and TACC_MKL_DIR is an environment variable that is defined by default because the Intel compiler suite and MKL library are loaded by default when the user logs into Stampede.

3.3.2 Experiment Setup

We conducted the experiment with ten users and evaluated the programming workflows against that of the expert user. Users were provided with general guidelines to follow and instructed to enter their start and end times for each step, including both correct and incorrect (or

unfruitful) steps. These records were used later to create the workflow graphs for evaluation. Later on we created an LAPACK Porting Guide to try to address some of the problems discovered with the novice workflows.

The expert user was the instructor of the CPS 5401 Introduction to Computational Science class at the University of Texas at El Paso in fall semester of 2015. The novice users were students in the class. The students had received instruction in the functionality and use of numerical libraries, including LAPACK, but had not been instructed in this particular workflow on Stampede. The students were all also co-enrolled in or had previously taken MATH 5329 Numerical Analysis in which course they had learned about numerical dense linear algebra methods. We observed the novice users while they worked on the problem. They were instructed to ask for help if they got stuck for a long time. When they asked for help, we checked that they had recorded their progress so far and gave a hint to enable them to move forward.

The ideal workflow for porting the code to Stampede consists of the following tasks:

1. Transfer the program file to Stampede
2. Lookup the correct commands in the Stampede User Guide for compiling the file
3. Insert missing include directives
4. Change the LAPACKE include directive to include mkl_lapacke.h
5. Change the data types for routine arguments to be portable LAPACK types
6. Call LAPACKE_dgetrf correctly to factor the matrix
7. Call LAPACKE_dgetrs correctly to solve the triangular system
8. Compile the program and fix any errors
9. Execute the program and validate the result

To evaluate the productivity of the workflow, we assigned metrics with respect to the following criteria:

C1. Time to complete the workflow

C2. Correctness of the program

C3. Portability of the program

C4. Maintainability of the program

After analyzing the results of the above mentioned experiment we created a porting guide to try to improve the available user documentation.

3.4 Case study 2: Solving a Sparse Linear System from a Finite Element Analysis

3.4.1 Problem Definition

Finite element analysis is a method used to solve systems of partial differential equations (PDEs) that model physical problems in application areas such as computational fluid dynamics and structural mechanics. When the PDEs are discretized, the result is a sparse linear system, often very large, that needs to be solved. Both direct and iterative methods have been developed to solve such systems. Large systems may need to be solved with iterative methods and/or in parallel because of memory or computational bottlenecks.

PETSc stands for The Portable, Extensible Toolkit for Scientific Computation and the library is developed at Argonne National Laboratory together with many collaborators. The PETSc library has been developed for the purpose of assisting computational scientists in solving PDE problems [16]. PETSc includes a collection of both direct and iterative solvers for sparse linear systems with both serial and parallel implementations. PETSc employs the Message Passing Interface (MPI) for parallel solutions. PETSc supports C, C++, FORTRAN and Python applications and contains a large number of parallel linear and nonlinear equation solvers for large-scale problems. Some of the PETSc modules deal with index sets (IS), including permutations, for indexing into vectors, renumbering, etc; vectors (Vec); matrices (Mat) (generally sparse); managing interactions between mesh data structures and vectors and matrices (DM) over fifteen Krylov subspace methods (KSP); dozens of preconditioners, including multigrid, block solvers, and sparse direct solvers (PC); nonlinear solvers (SNES); and time steppers for solving time-dependent (nonlinear) PDEs, including support for differential algebraic equations (TS).

To evaluate the sparse linear system portion of the finite element analysis workflow, we chose matrices from the DRIVCAV collection. These matrices are from modeling 2D fluid flow in a driven cavity. The physical problem represented by the driven cavity is a square in cross section, with velocity equal to zero on three walls, and equal to one at the fourth wall, in the direction parallel to the fourth wall. This results in a circulating flow, similar to that which would occur in a notch in an infinite flat plate, with the notch cut perpendicular to the free stream flow direction over the plate. To produce the matrices, the flow was modeled using the incompressible Navier Stokes equations. These were discretized using the Galerkin finite element method and linearized using Newton's method. The matrices are non-symmetric and indefinite. They are difficult to solve using iterative methods like preconditioned Krylov subspace methods, because it is difficult to find an effective preconditioner. The matrices can be successfully solved using direct methods like frontal or skyline solvers, but as the size of the matrix and the Reynolds number increases, the filling in the lower (L) and upper (U) triangular factors increases, and this ultimately limits the use of these solvers.

The specific matrices we chose were E40R000 (40 x 40 elements, Reynolds number 0, symmetric indefinite) and E40R0500 (40 x 40 elements, Reynolds number 500, real unsymmetric). Each of these matrices is 17281 by 17281, with 553956 entries. The matrices and the accompanying right hand side vectors are available on the Matrix Market website [17].

3.4.2 Experimental Setup

We evaluated the workflow using a set of eight novice users, similar to the previous experiment. These novice workflows were also evaluated against the expert user's workflow. Similar to the previous experiment timings and different paths the users took were recorded.

The workflow for solving the linear system using PETSc consists of the following tasks:

1. Download the matrix and right hand side vector from Matrix Market.

2. Convert the matrix and right hand side vector to the PETSC binary format.
3. Determine the properties of the matrix.
4. Based on the matrix properties, choose an appropriate solution method.
5. Implement the solution using the PETSc software.
6. Execute the program.
7. Evaluate the results.
8. If results are unsatisfactory, go back to step 4.

To evaluate the productivity of the workflow in two different contexts, we divided our subjects into two groups. One group used the Stampede and PETSc documentation to try to implement the workflow manually. The second group used the Lighthouse tool [12,13] to try to implement the workflow.

For each context, we evaluated productivity with respect to the following criteria:

C1. Time to develop the workflow

C2. Time to execute the program

C3. Portability of the program

C4. Maintainability of the program

C5. Accuracy of the solution

C6. Reusability of the results

Unlike in the first case study, where getting the code to run correctly with the vendor version of LAPACK would result in good performance, the performance of the PETSc workflow depends heavily on the choice of solver. With dense linear algebra methods, the number of steps is deterministic. With iterative methods for sparse systems, however, the number of steps to converge to a solution within the desired error tolerance depends on how well suited the solver method is for the problem and on the effectiveness of the pre-conditioner. Also, the program may terminate without converging to a solution if the specified maximum number of steps is exceeded, but unless the user has output whether or not convergence occurred, she may incorrectly assume that the current

approximation to the solution is correct. With PETSc, the same code can be used for different methods, with the choice of solver and pre-conditioner specified on the command line. Some level of expertise is required to know what solvers and pre-conditioners to specify and to figure out the PETSc command-line options to use to implement these choices.

Data were collected for both matrices for both groups. The subjects were instructed to solve the easier system first, followed by the harder system, and to use a relative error tolerance of 1e-10. In addition to collecting data from test subjects, we also collected data from the expert user carrying out the workflow in each of the two contexts.

Chapter 4: Results

4.1 Workflow Representation

We illustrate our observations and measurements of the user workflows using a workflow graph as in [8]. Graph nodes indicate a step. Backtracking indicates that there is no progress and the user is repeating the work at the same node. For example, because of errors or needing to try something different. Colors are used to separate main tasks from subtasks. Green indicates start and Red indicates the end. We use broken arrows to indicate a connection from the previous node to the next node in a different line. We recorded all the steps of workflows and analyzed where each novice user diverged from the expert user and also how novice users are differed from each other.

4.2 Case Study 1

4.2.1 Results

The expert workflow for case study 1 is illustrated in figure 1, and the corresponding steps with timings are given in table 1. Workflows for three of the novice users are illustrated in figures 2 through 4. The corresponding steps and user reported timings are given in tables 2 through 4 respectively. Arrows indicate progression from one activity to another. Backtrack arrows indicate that the user performed tasks that took time that did not make actual progression since they go back to do the same task again.

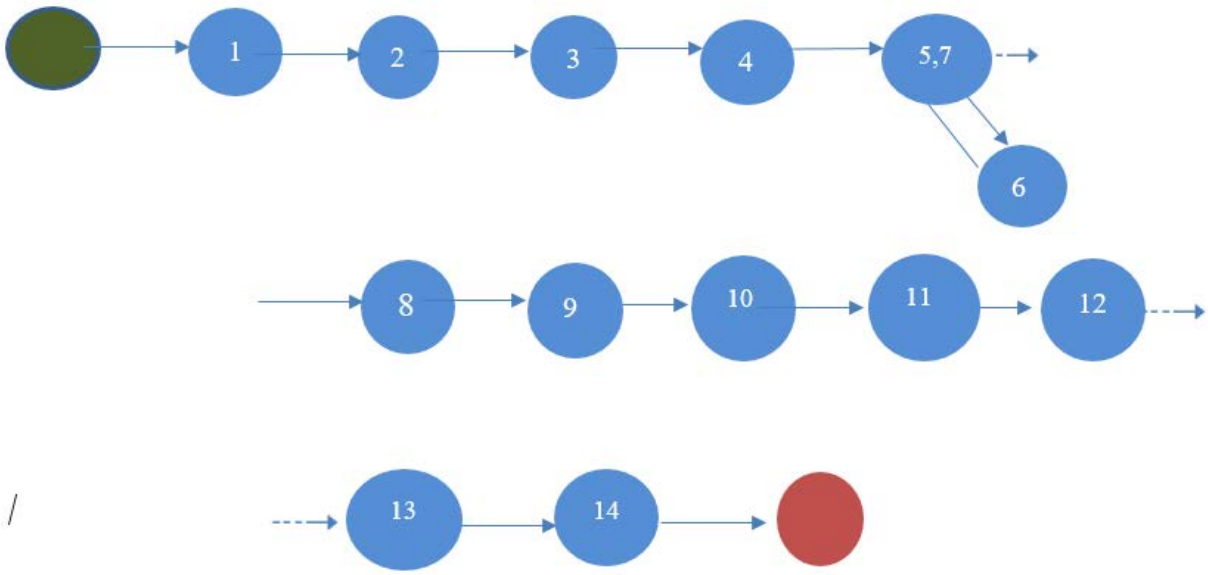

Figure 4.2.1.1 Expert Workflow for case study 1

Table 4.2.0.1.1: Workflow of the Expert User for Case study 1

Step	Task	Time (minutes: seconds)
0	Start	
1	Click on the link	1:00
2	Copy and paste on the editor	1:00
3	Save file	1:00
4	Lookup compile command in STAMPEDE User Guide	1:00
5	Compile and check for errors	0:30
6	Insert directive 'include <iostream>'	0:00
7	Recompile and look for errors	2:00
8	Replace lapacke.h with mkl_lapacke.h	2: 00
9	Lookup LAPACKE_dgetrf file reference	2: 00
10	Change data types in the call sequence	5: 00
11	Lookup LAPACKE_dgetrfs file reference	1: 00
12	change call sequences	4: 00
13	recompile	0:30
14	Run	0:30
15	End	
	Total Time	21:00

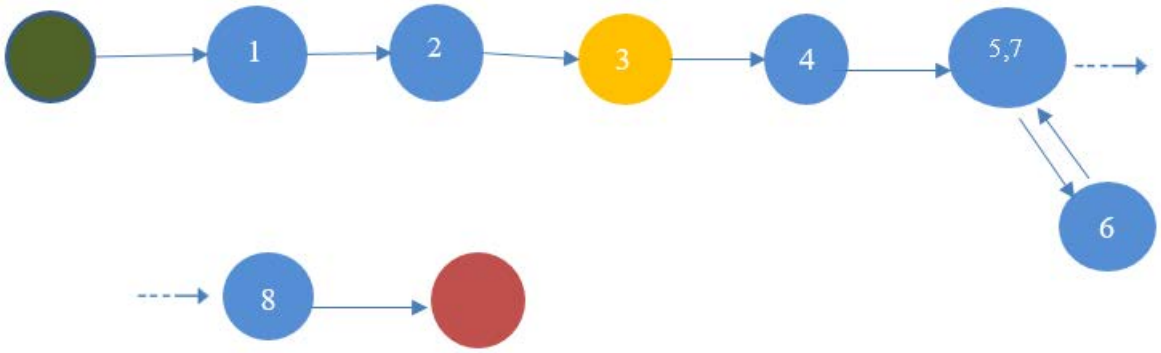

Figure 4.2.1.2. Novice user 4 Workflow for case study 1

Table 4.2.1.0.1: Workflow of the Novice User 4 for Case study 1

tep	Task	Time (minutes : seconds)
	Start	
	Click on the link	1:00
	Copy and paste on the editor	7:00
	Download code author provided header files	4:00
	Lookup compiler command in STAMPEDE User Guide	3:00
	Compile with g++ and check for errors	0:30
	Insert directive 'include <iostream>'	8:00
	Recompile and look for the error	1:00
	Run	1:00
	End	
	Total Time	25:00

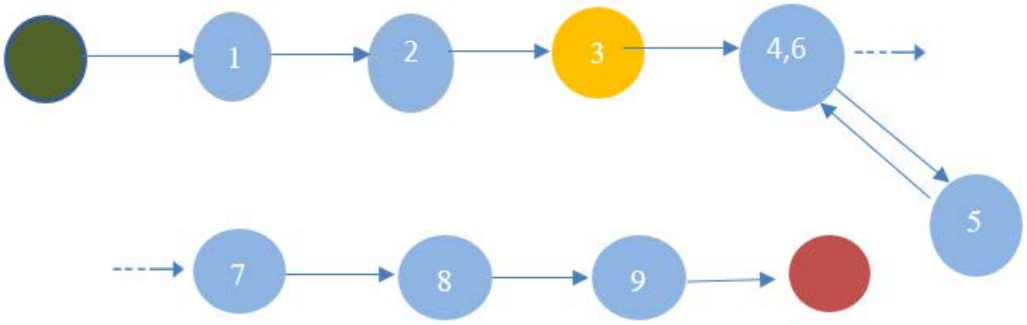

Figure 4.2.1.3:Workflow of Novice user 3 for case study 1

Table 4.2.1.0.1:Workflow of the Novice User 3 for Case study 1

tep	Task	Time (minutes : seconds)
	Start	
	Click on the link	0: 30
	Copy and paste on the editor	0: 30
	Upload author provided header files	0: 30
	compile with g++ and check for errors	1:00
	Insert directive 'include <iostream>'	0:30
	Compile and check error	1:00
	fix the error	4:00
	Recompile	0: 30
	Run	0: 30
	End	
	Total Time	9:00

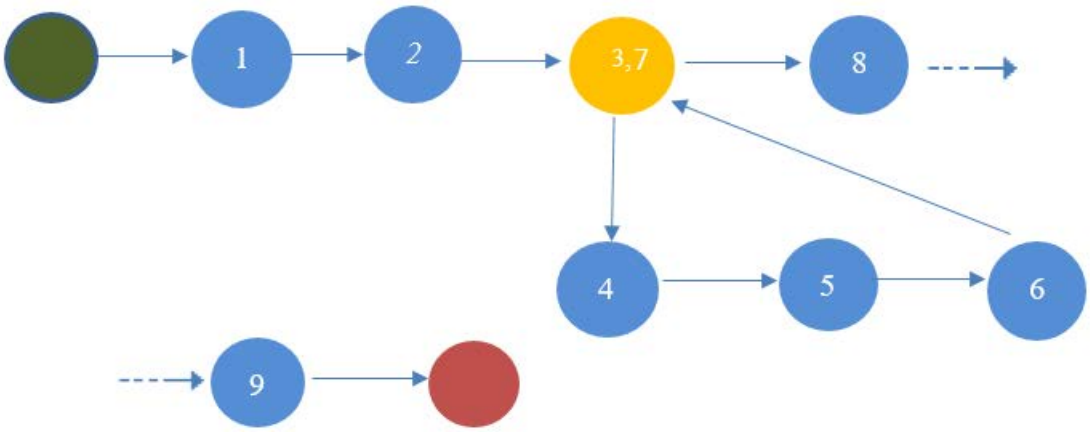

Figure 4.2.1.4 Novice user 1 Workflow for case study 1

Table 0.1.2.1.4: Workflow for Novice user 1 for case study 1

tep	Task	Time (minutes : seconds)
	Click on the link	0: 30
	Copy and paste on the editor	3: 30
	Compile with g++	0: 30
	Look for a different compiler	0: 30
	Insert directive ‘include <iostream>’	8:00
	Download the LAPACKE header files	2:00
	Recompile and look for the error	1:00
	Use a different header file	1:00
	Recompile	0:30
0	Run	0:30
	Total Time	18:00

4.2.2 Analysis

We compared the workflow diagrams of the novice users with that of the expert user and looked for the branches where the novice users deviate. These different paths were assigned evaluation metrics for each criterion in order to obtain a productivity measurement scheme.

Development Time

Development time can be negatively affected by the following deviations of the steps by spending more time than necessary. For this particular problem of porting legacy code for the use of LAPACK library on STAMPEDE machine efficiency is directly associated with time and it is negatively affected by spending more time by users spending time going on wrong paths.

Development time was affected by the following wrong steps:

1- Using g++ instead of the intel C++ compiler

2- Oblivious to the fact that LAPACK is already installed in Stampede

Correctness

If the user gets a wrong result or partially wrong result, accuracy gets a negative value with a weight. Following steps could lead users to get wrong results.

1. Positive: Correct answer
2. Negative: Being oblivious to row vs. column major ordering
3. Negative: Using Netlib reference version header files rather than MKL header files

These negative steps did not cause an incorrect answer with this particular code. Being oblivious to row vs. column major ordering did not affect the solution in this case since the matrix was symmetric, but for a nonsymmetric matrix, the result could have been incorrect. Using header and library files from different implementations of LAPACK did not affect the correctness in this case but doing this is not a best practice and could lead to incorrect results. Since the negative steps did not affect correctness for this case study, we do not include them in the metric value.

Portability

Portability can be negatively affected by using a deprecated interface and using custom header files. Usage of old interfaces could be incompatible with extensions of the library.

These are the steps that affected portability in this particular problem:

1- Negative: Not using portable LAPACK data types

For example, the type lapack_int is used for integer arguments to LAPACK routines. This portable type may be implemented differently on different platforms but still retains the same semantics.

2- Negative: Using the deprecated C interface rather than the standard LAPACKE interface

Maintainability

Long-term maintainability of the code is affected by the following steps:

1- Negative: Using the custom header files instead of MKL header files

The custom header files would need to be maintained along with the program and possibly updated.

Productivity Vector

Presented vector of measurement for evaluation takes development time of obtaining the end-to-end results, code portability, and correctness of the obtained results and maintainability of the code into consideration.

[Development-time Correctness Portability Maintainability]

1. Evaluating the workflow of User 4 against expert user

Development-time – Total time was 25 minutes against the expert user time of 20 which gives us -5 in the vector

Correctness – 0 since the result was correct

Portability – Use of custom files and not using LAPACKE data types -2Wm

Maintainability – User has used a custom header file. -1Wp

[-5 +1Wc -2Wp -1Wp]

2. Evaluating the workflow of User 3 against expert user

Development-time – Total time was 9 minutes against the expert user time of 20 minutes which gives us +11 in the vector.

Correctness – 0 since the result was correct

Portability – Use of custom files -1Wp

Maintainability – User has used a custom header file. -1Wm

[+11 +1Wc -1Wp -1Wm]

3. Evaluating the workflow of User 1 against expert user

Development-time – Total time was 18 minutes against the expert user time of 20 which gives us +2 in the vector.

Accuracy – 0 since the result was correct

Portability – Use of custom files -2Wp

Maintainability – User has used a custom header file. -1Wm

[+2 0 -2Wp -1Wm]

Depending on the weights assigned for the correctness, portability, and maintainability metrics, the most productive workflow might be that of the expert user or of the novice user who completed the task in the least amount of time. In reality, the times self-reported by the novice users are not accurate, since they did not include time asking for help or looking for answers. Novice users used a ‘quick and dirty’ approach whereas expert user took time to rewrite the code to achieve long term maintainability and portability.

4.3 Case Study 2

The expert user's workflow without using Lighthouse is illustrated in figure 5. The corresponding steps and timings are given in table 5.

4.3.1 Results

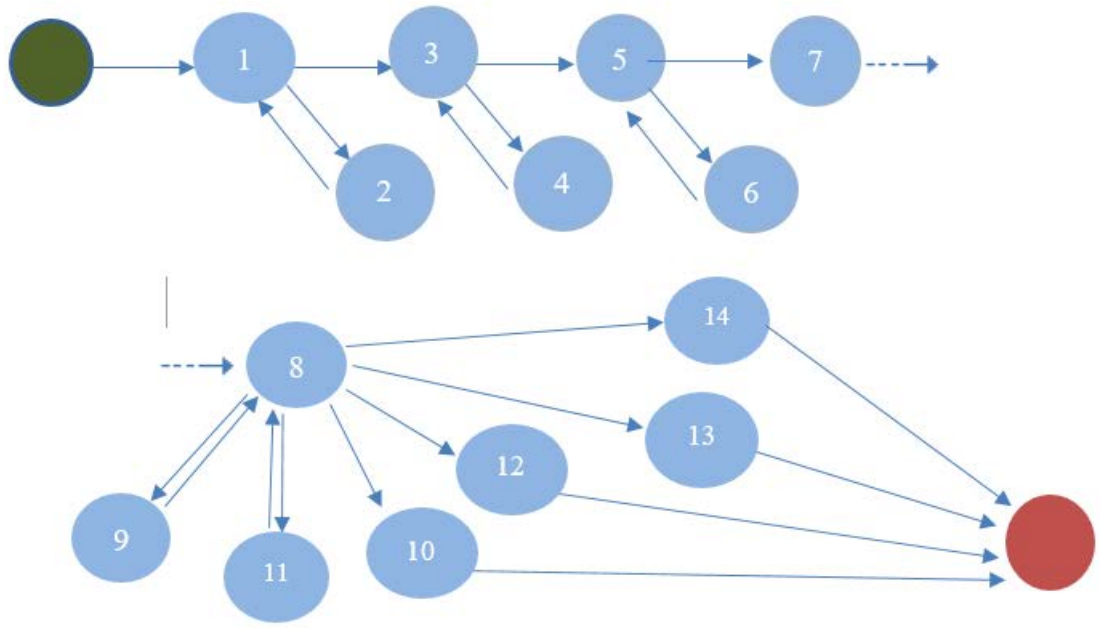

Figure 4.3.1.1: Expert User's Workflow for case study 2, without Lighthouse

Table 4.3.1.1: Workflow of the Expert User 1 for Case study 2

Step	Task	Time (minutes :seconds)
0	Start	
1	Downloaded the matrix and right hand side vector from Matrix Market	2:00
2	Looked in the PETSc examples for code that could read Matrix Market format (unsuccessfully)	2:00
3	Looked for a program to convert the matrix and right hand side vector to the PETSC binary format and found mm2petsc code	3:00
4	Compiled mm2petsc with errors	1:00
5	Switched to PETSc 3.5 from the Stampede default version of 3.6	1:00
6	Converted the matrix from Matrix Market format to PETSc format and found that the mm2petsc program doesn't work for vectors	4:00
7	Modified mm2petsc to work for vectors and converted right hand side to PETSc binary format	12:00
8	Searched for PETSc ksp examples and found existing ex18.c, modified ex18.c to read matrix and vector from separate files	8:00
9	Attempted to solve using GMRES solved without success	2:00
10	Successfully solved in 3 iterations using GMRES with fieldsplit command pre-conditioner	2:00
11	Attempted to solve using MINRES solver without success	1:00
12	Successfully solved in xxx iterations using BICG solver	2:00
13	E40r0500: Attempted to solve using GMRES without success	2:00
14	E40r0500: Successfully solved in 427 iterations using BICG	2:00
	Total Time	44:00

The expert user's workflow using Lighthouse is illustrated in figure 6 below.

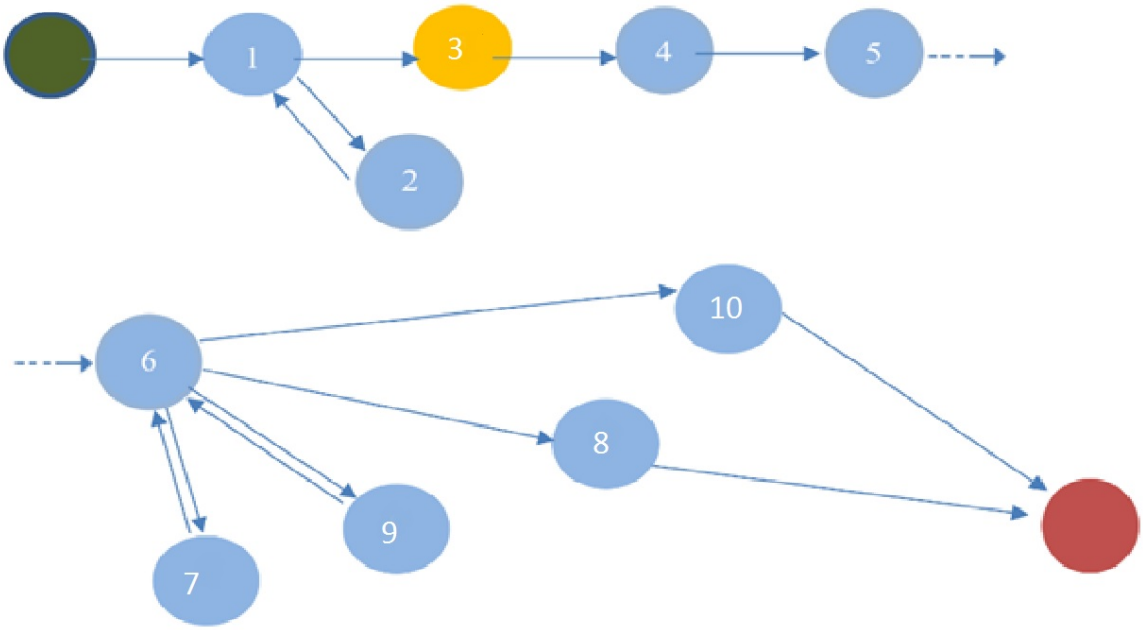

Figure 4.3.1.2: Expert User's Workflow for case study 2 using Lighthouse

Note that the export workflow using Lighthouse does not include the tasks of converting the input files from Matrix Market format to PETSc binary format nor trying different solvers unsuccessfully, since these had already been carried out in the workflow without Lighthouse which was done first. If we add in these additional 31 minutes, the total time for the expert workflow using Lighthouse comes to 54 minutes. However, the expert user spent 10 minutes generating the linear_solver.c template, most of which were spent unproductively trying to figure out why the automated matrix analysis wasn't working.

Table 4.3.1.2: Workflow of the Expert User 1 for Case study 2, with Lighthouse

tep	Task	Time (minutes : seconds)
	Start	
	Download the matrix and right hand side vector from Matrix Market.	2:00
	Lookup the PETSc examples for code that could read matrix and vector format(unsuccessfully)	2:00
	Tried Lighthouse using PETSC for sparse linear solutions by uploading the matrix E40r0000 but Lighthouse failed to analyze but it generated code template linear_solver.c	10:00
	Modified linear_solver.c to read matrix and vector format (ascii)	1:00
	Modified linear_solver.c to read separate files	4:00
	Compiled using provided makefile	5:00
	Attempted to solve using gmres solver without success	2:00
	Solved successfully using fieldsplit preconditioner in 3 iterations	2:00
	Solved successfully using bicg solver with 427 iterations	2:00
0	Solved E40r0500 using bicg solver in 546 iterations	2:00
	End	
	Total Time	32:00

We had to set up the same experiment twice because at the first attempt of the case study 2 the user group were unable to solve the problem in a realistic time frame. Note that the time indicated in the user's workflows does not indicate the time spent at the first try which was almost two hours of the whole class time, where they spent the whole time on converting input files. The time indicated in the table does not include the time they spend in the previous attempt.

The workflow for a novice user in the group that manually solved the problem is given below. A novice user's workflow without using Lighthouse is illustrated in figure 7. The corresponding steps and timings are given in table 7. The workflow diagram shows extensive

suboptimal paths indicating that the students had to get a lot of help to get to the correct path for solving the problem.

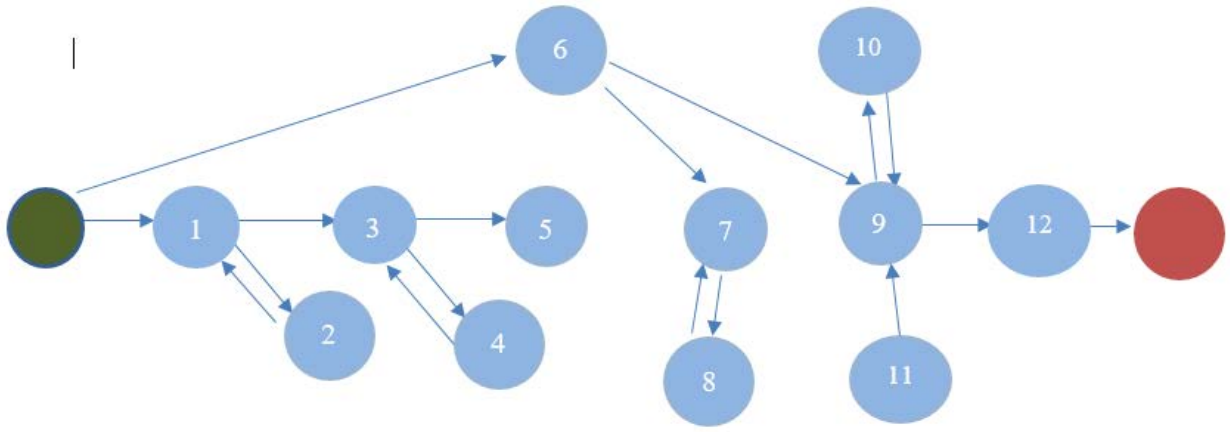

Figure 4.3.1.3: A Novice User's Workflow for case study 2 without using Lighthouse

Table 4.3.1.3: Workflow of a Novice User 1 for Case study 2, without Lighthouse

tep	Task	Time (minutes :seconds)
	Start	
	Copy and unzip file from stampede	4:00
	Looked for PETSc functions (preconditioned iterative metnods)	4:00
	Find the PETSc soclution code	3:00
	Compiled ext18.c with errors	1:00
	Search ways to run the ext18.c successfully	13:00
	Got Instructer's help and the binary file was provided.	1:00
	Copy the file linear_solver.c to Stampede	1:00
	Compiled the code without success	8:00
	Got Instructer's help and the executable file was provided.	1:00
0	For Matrix E40r0000 Attempted to solve using default command without converging	9:00
1	End	
2	Total Time	45.00

4.3.2 Analysis

The most efficient way found by the expert user to solve the symmetric indefinite system was using GMRES with the fieldsplit Shur preconditioner. The methods tried by the expert user are shown in Figure 8.

```
c557-603.stampede(9)$ ./ex18 -ksp_type gmres -ksp_rtol 1e-10 -ksp_converged_reason -f input/e40r0000.bin -rhs input/e40r0000_rhs1.bin
Linear solve did not converge due to DIVERGED_ITS iterations 10000
Number of iterations = 10000
Residual norm 0.000127714
```

```
c557-603.stampede(10)$ ./ex18 -ksp_type gmres -ksp_rtol 1e-10 -pc_type fieldsplit -pc_fieldsplit_type schur -pc_fieldsplit_detect_saddle_point -ksp_converged_reason -f input/e40r0000.bin -rhs input/e40r0000_rhs1.bin
Linear solve converged due to CONVERGED_RTOL iterations 3
Number of iterations =   3
Residual norm 8.32361e-09
```

```
c557-603.stampede(11)$ ./ex18 -ksp_type minres -ksp_rtol 1e-10 -pc_type fieldsplit -pc_fieldsplit_type schur -pc_fieldsplit_detect_saddle_point -ksp_converged_reason -f input/e40r0000.bin -rhs input/e40r0000_rhs1.bin
Linear solve did not converge due to DIVERGED_INDEFINITE_MAT iterations 2
Number of iterations =   2
Residual norm 8.88668e-05
```

```
c557-603.stampede(12)$ ./ex18 -ksp_type bicg -ksp_rtol 1e-10 -ksp_converged_reason -f input/e40r0000.bin -rhs input/e40r0000_rhs1.bin
Linear solve converged due to CONVERGED_RTOL iterations 427
Number of iterations = 427
Residual norm 3.83479e-10
```

Figure 4.3.2.1: Transcript of Expert User Session for Solution of E40R0000

Note that the GMRES method without the fieldsplit preconditioner and the MINRES method did not converge.

The most efficient method found by the expert user for solving the nonsymmetric system was the bicg method. The methods tried by the expert user are shown in Figure 9.

```
c557-603.stampede(13)$ ./ex18 -ksp_type gmres -ksp_rtol 1e-10 -ksp_converged_reason -f input/e40r0500.bin -rhs input/e40r0500_rhs1.bin
Linear solve did not converge due to DIVERGED_ITS iterations 10000
```

```
Number of iterations = 10000
Residual norm 0.00139111
```

```
c557-603.stampede(14)$ ./ex18 -ksp_type bicg -ksp_rtol 1e-10 -ksp_converged_reason -f input/e40r0500.bin -rhs input/e40r0500_rhs1.bin
Linear solve converged due to CONVERGED_RTOL iterations 546
Number of iterations = 546
Residual norm 1.22714e-10
```

Figure 4.3.2.2: Transcript of expert user session for solution of E40R0500

The expert solutions were obtained using PETSC version 3.5 and by modifying $PETSC_DIR/src/ksp/ksp/examples/tests/ex18.c to read the matrix and the right-hand side from different files. The same solutions were easily obtained by modifying the code template produced by Lighthouse to read the matrix and right-hand side from different files and to compute the residual norm. Once the basic code has been compiled and made to run, the different solvers and pre-conditioners can be explored using runtime command-line options.

The most time consuming step in the expert workflow was converting the input files to PETSc binary format. The code to do this, mm2petsc.c, was found using Google. The code had not been updated to PETSc 3.6, so the expert had to drop back to PETSc 3.5 to get it to compile. The matrix and vector manipulation examples in the PETSc 3.6 examples directory on Stampede have also not been updated to PETSc 3.6 but work with version 3.5. Lighthouse does not seem to understand input files and although it has a button to load them, does not appear to do anything with them.

None of the ten novice users were able to complete the case study 2 workflow at the first attempt. In the first attempt we divided them into two groups of five. One group was instructed to use Stampede and PETSc documentation to solve the problem. The second group was instructed to also use Lighthouse. Both groups first got stuck on the task of converting the input files from Matrix Market to PETSc binary format and spent an hour trying to figure that out. When we gave them the mm2petsc and mv2petsc codes, however, to convert the matrices and

vectors, respectively, they were able to complete this task. We spent another hour-long session observing the novice users attempting to complete the next task of finding or generating the code template and modifying it to read the matrix and right-hand side vector from separate files. The first group was unable to locate the appropriate example code in the PETSc examples on Stampede. The second group was able to generate the linear_solver.c code template using Lighthouse but was unable to modify it to read the matrix and vector from separate files.

The problem of modifying the code template to read the matrix and vector from separate files is not as trivial as it may seem. Special PETSc routines are provided to load a matrix (resp. vector) from a file and it takes several steps. The novice user would need to look up the documentation for these routines and understand it to figure out how to modify the code. The original linear_solver.c code and the modified version by the expert user are given in Appendix 2.

Although the first group of novice users did not get to it in the first attempt, we suspected that they would have been unable to determine the correct solver and preconditioner to use to solve either sparse system. In the second attempt we provided more help and guidance to the novice users and they were able to solve the problem within the allocated time. We asked half of the user group of eight to get help from the Lighthouse tool and the other half went on solving the problem without using any other tools. Since we had to give extensive help for solving the problem we evaluate productivity for the second problem more qualitatively than quantitatively.

If the code from Lighthouse is run using the run instructions provided by Lighthouse, one gets the output shown in Figure 8 and a solution vector is written to the output directory. To the novice user, it may appear that the method converged and a valid solution was generated. The expert user knew to add the option -ksp_converged_reason so that PETSc would report whether the method converged or not. The expert user also had previous knowledge of sparse iterative methods that enabled her to know which solves to try. The expert knowledge about the different solvers is summarized in Table 8.

Choosing an effective preconditioner is also an expert skill. A preconditioner transforms the sparse matrix so that it is better conditioned which will hopefully help the iterative method to converge faster or to converge at all. The expert user had previous knowledge that the fieldsplit preconditioner is often effective for symmetric indefinite systems resulting from incompressible flow CFD problems. She used Google to find the correct options to use for this preconditioner.

Table 4.3.2: Expert Knowledge about different Nonstationary Solvers

Conjugate Gradient (CG)	CG is an extremely effective method when the coefficient matrix is symmetric positive definite, since storage for only a limited number of vectors is required.
Minimum Residual (MINRES) and Symmetric LQ (SYMMLQ)	These methods are computational alternatives for CG for coefficient matrices that are symmetric but possibly indefinite. SYMMLQ will generate the same solution iterates as CG if the coefficient matrix is symmetric positive definite.
Conjugate Gradient on the Normal Equations : CGNE and CGNR	When the coefficient matrix is nonsymmetric and nonsingular, the normal equations matrices will be symmetric and positive definite, and hence CG can be applied. The convergence may be slow, since the spectrum of the normal equations matrices will be less favorable.
Generalized Minimal Residual (GMRES)	Unlike MINRES (and CG) it requires storing the whole sequence, so that a large amount of storage is needed. For this reason, restarted versions of this method are used.. This method is useful for general nonsymmetric matrices.
BiConjugate Gradient (BiCG)	It is useful when the matrix is nonsymmetric and nonsingular; however, convergence may be irregular, and there is a possibility that the method will break down. BiCG requires a multiplication with the coefficient matrix and with its transpose at each iteration.
Quasi-Minimal Residual (QMR)	The Quasi-Minimal Residual method applies a least-squares solve and update to the BiCG residuals, thereby smoothing out the irregular convergence behavior of BiCG, which may lead to more reliable approximations. In full glory, it has a look ahead strategy built in that avoids the BiCG breakdown. Even without look ahead, QMR largely avoids the breakdown that can occur in BiCG. On the other hand, it does not effect a true minimization of either the error or the residual, and while it converges smoothly, it often does not improve on the BiCG in terms of the number of iteration steps.
Conjugate Gradient Squared (CGS)	The Conjugate Gradient Squared method is a variant of BiCG that applies the updating operations for the A-sequence and the A^T-sequences both to the same vectors. Ideally, this would double the convergence rate, but in practice convergence may be much more irregular than for BiCG, which may sometimes lead to unreliable results. A practical advantage is that the method does not need the multiplications with the transpose of the coefficient matrix.
Biconjugate Gradient Stabilized (Bi-CGSTAB)	The Biconjugate Gradient Stabilized method is a variant of BiCG, like CGS, but using different updates for the -sequence in order to obtain smoother convergence than CGS.

Chapter 5: Conclusions and Future Work

Our evaluation framework presents a way to compare novice workflows against that of an expert user and at the same time gives a measure for productivity using workflow-specific criteria. We pla

n on using this schema to evaluate larger scale workflows for finite element modeling as well as other applications. Our evaluation criteria can be modified for different objectives and constraints and the productivity vector can be used with few variations. For our particular case studies we selected efficiency, accuracy, portability and maintainability as the criteria of evaluation. These criteria have to be selected related to the case study we intend to evaluate. By changing the criteria of the productivity vector, the methodology can be adapted for a different problem domain.

Our work focuses on the end-to-end solution of the problem, which is more relevant when evaluating the productivity. A process is productive when it meets the requirements of the user accurately and completely and efficiently in terms of time and resources. It is beneficial to specify requirements according to the specific problem. The productivity vector should evaluate the end-to-end productivity related to overall time and resources rather than technical attributes needed to achieve the immediate execution performance.

Different groups of users have different needs. Our approach is to compare novice user productivity to that of an expert user, using the expert workflow as a baseline. Productivity depends on the user perception of the problem and on knowledge of the methods of solution. Novice users generally prefer graphical user interfaces (GUIs) and find them easier to use. However, the GUI must support the most difficult and time-consuming steps of the workflow, as well as meet the other evaluation criteria. In our second case study using the Lighthouse tool, we found that Lighthouse does not support the most time-consuming task of pre-processing the input files. Since the Lighthouse developers state that they have tested their tool with matrices from Matrix Market, it should be easy for them to extend their tool to support conversion from the

Matrix Market (.mtx) and Harwell-Boeing formats used by Matrix Market to the PETSc binary format. The runtime efficiency and accuracy criteria are also not well-supported by the Lighthouse tool, since it gives the novice user no help in choosing an appropriate solver and preconditioner for the problem at hand, or even in determining whether or not convergence to a solution occurred. The Lighthouse tool is very helpful to novice users in generating working code that makes proper use of the PETSc routines. What is lacking is help with options for running the code that select an appropriate solver and preconditioner. The developers of Lighthouse state that they have tested their tool with large numbers of sparse matrices from Matrix Market and used the Lighthouse tool to successfully solve them. However, the developers of Lighthouse are also expert users of PETSc and have the expert knowledge to specify the appropriate options.

For future work, we will use the results of our evaluation to make suggestions to the Lighthouse developers about how they can improve their tool to increase user productivity. We will also provide recommendations to Texas Advanced Computing Center (TACC) on additions to the Stampede user guide to help users be more productive in using the MKL version of LAPACK. We plan to extend our workflow evaluation to more complex workflows, including those that use parallel processing. We plan to develop more accurate ways of collecting data for workflows. As pointed out in [4], measuring effort accurately and consistently across subjects is difficult. Self-reports can be unreliable, but not all activities can be captured automatically.

References

1. Thomas L. Sterling and Chirag Dekate. Productivity in High Performance Computing. *Advances in Computers* 72: 101-134 (2008)

2. Allen D. Malony. Monitoring Analytics for In Situ Workflows at the Exascale. *Petascale Tools Workshop,* Lake Tahoe, CA, August 2015.

3. Stuart R. Faulk, Eugene Loh, Michael L. Van De Vanter, Susan Squires, and Lawrence G. Votta: Scientific Computing's Productivity Gridlock: How Software Engineering Can Help. *Computing in Science and Engineering* 11/2009; 11(6):30-39. DOI:10.1109/MCSE.2009.205

4. Lorin Hochstein, Victor R. Basili, Marvin Zelkowitz, Jeff Hollingsworth, Jeff Carver. Combining self-reported and automatic data to improve effort measurement. *Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005*). September 2005.

5. Watts S. HumphreyTechnical Report: The Personal Software Process (PSP), Software Engineering Institute, November 2000 CMU/SEI Report Number: CMU/SEI-2000-TR-022

6. Min Zhang, Lorin Hochstein, "Fitting a workflow model to captured development data", ESEM *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement 2013*.

7. Ronak Etemadpourm, Paul Murray, Matthew Bomhoff, Eric Lyons, and Angus Graeme Forbes. Designing and Evaluating Scientific Workflows for Big Data Interactions. *IEEE International Symposium on Big Data Visual Analytics*, Septermber 2015, Hobart, Tasmania, Australia.

8. Robert Numrich, Lorin Hochstein, Vic Basili. A Metric Space for Productivity Measurement in Software Development. *Workshop on Software Engineering for High Performance Computing Applications (SE-HPCS), ICSE,* St. Louis, MO. May 2005.

9. Min Zhang, Lorin Hochstein. Fitting a Workflow Model to Captured Development Data. *Third International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*, October 2009.

10. Lorin Hochstein, Victor R. Basili, Uzi Vishkin, John Gilbert. A Pilot Study to Compare Programming Effort for Two Parallel Programming Models, *Journal of Systems and Software*, 81, 2008 (doi:10.1016/j.jss.2007.12.798).

11. Boyana Norris, Sa-Lin Bernstein, Ramya Nair, and Elizabeth Jessup. Lighthouse: A User-Centered Web Service for Linear Algebra Software. August 2014. http://arxiv.org/abs/1408.1363

12. Pate Motter, Kanika Sood, Elizabeth Jessup, and Boyana Norris. 2015. Lighthouse: an automated solver selection tool. In Proceedings of the *3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE'15)*.ACM,NewYork,NY,USA,16-24. DOI=http://dx.doi.org/10.1145/2830168.2830169

13. H. Carter Edwards and Christian R. Tron. Kokkos: Enabling performance portability across manycore architectures. *Extreme Scaling Workshop*, Boulder, Colorado, August 2013.

14. LAPACK User Guide. August 1999. http://www.netlib.org/lapack/lug/

15. The LAPACKE User Guide. November 2013. http://www.netlib.org/lapack/lapacke.html

16. PETSc User Manual, Revision 3.6, Argonne National Laboratory, June 2015, http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf .

17. http://math.nist.gov/MatrixMarket/

Appendix 1

LAPACK Porting Guide for STAMPEDE

LAPACK Porting Guide for C/C++ Programmers

These guidelines are intended to help C and C++ application developers use LAPACK routines in a way that results in efficient, portable, and maintainable code for high performance computing systems.

1. Efficiency: In order for your code to be as efficient as possible, you should use the version of LAPACK that is tuned for your computer system. You should install and use the Netlib reference version only as a last resort if no tuned version is available. Look in your system documentation to determine what library to use. The library may not be called LAPACK even though it includes LAPACK. For example, on non-Cray Intel systems, the Intel Math Kernel Library (MKL) is usually installed and includes LAPACK. Likewise, on non-Cray AMD systems, the library is ACML. On IBM systems, the library is ESSL. On Cray systems, it is Cray libsci. Although the routine names and prototypes are a de facto standard and are the same across implementations, the commands for compiling and linking are different. Please refer to your system library documentation to determine the correct compile and link commands.

2. Portability:

2.1 For maximum efficiency and portability for a C or C++ code that uses LAPACK, you should use the extended LAPACK (lapacke) interface if it is available. The LAPACKE routine names start with LAPACKE, followed by the usual LAPACK routine name, for example LAPACKE_dgesv, for the LAPACK routine DGESV that solves a linear system (SV) for a general matrix (GE) in double precision (D). Go to www.netlib.org/lapack/explore-html/files.html and expand the LAPACKE section for documentation. In the src directory, you will find the prototypes for each routine. In the example directory, you will find examples of how to use an LAPACKE routine in a C/C++ program.

2.2 For maximum portability, you should use portable LAPACK data types for arguments to LAPACKE routines – for example, lapack_int instead of int. See the LAPACKE example programs for examples of how to do this.

2.3 Some of the vendors name their header files differently from the Netlib LAPACK reference version. For example, the MKL LAPACKE main header file is named mkl_lapacke.h instead of lapacke.h, so you will need to include mkl_lapacke.h instead of lapacke.h if you are using MKL. This will make your program slightly less portable, so you should document this vendor-specific change.

3. Correctness:

3.1 For correctness, you should use the LAPACKE header files that come with the LAPACK implementation you are using, rather than downloading the Netlib reference version header files. Use the appropriate –I flag if necessary so that the compiler can find the header files for the LAPACK version you are using.

3.2 When you call an LAPACKE routine, make sure that you specify correctly whether your matrix is stored in row-major or column-major order. For C programs, the natural ordering is row-major order. See the LAPACKE example programs for how to do this.

Appendix 2

Sparse Linear Solver Code Generated by Lighthouse

```
     / * Program usage:  mpiexec ex1 [-help] [all PETSc options] */

     static char help[] = "Solves a linear system with KSP.\n\n";

     /*T
        Main operation: Solve a linear system
        Input file format: PETSc binary format (matrix and rhs in the
same file)
        Processor: 1 (sequential)
        Output format: PETSc binary format
     T*/

     #include <petscksp.h>

     #undef __FUNCT__
     #define __FUNCT__ "main"
     int main(int argc,char **args)
     {
       Vec               x, b;        /* approx solution, RHS, exact
solution */
       Mat            A;        /* linear system matrix */
       KSP            ksp;          /* linear solver context */
       PetscViewer    fd,viewer;
       PetscErrorCode ierr;
       PetscInt       its;
       PetscMPIInt    size;
       char            file[2][PETSC_MAX_PATH_LEN];      /* input file
name */
       PetscBool      flg;

       PetscInitialize(&argc,&args,(char *)0,help);
       ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
       if   (size   !=   1)   SETERRQ(PETSC_COMM_WORLD,1,"This   is   a
uniprocessor example only!");

       /*
          Determine files from which we read the linear system (matrix
and right-hand-side vector).
       */
       ierr            =            PetscOptionsGetString(PETSC_NULL,"-
f",file[0],PETSC_MAX_PATH_LEN,&flg);CHKERRQ(ierr);
       if (!flg) {
         SETERRQ(PETSC_COMM_WORLD,1,"Must  indicate  binary  file  with
the -f option");
```

```
        }

        /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - -
              Compute the matrix and right-hand-side vector that
define
              the linear system, Ax = b.
           - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - */

        /*
             Open binary file.  Note that we use FILE_MODE_READ to
indicate
             reading from this file.
        */
        ierr                                                          =
PetscViewerBinaryOpen(PETSC_COMM_WORLD,file[0],FILE_MODE_READ,&fd);CHK
ERRQ(ierr);

        /*
           Load the matrix and vector; then destroy the viewer.
        */
        ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
        ierr = MatSetFromOptions(A);CHKERRQ(ierr);
        ierr = MatLoad(A,fd);CHKERRQ(ierr);

        ierr = VecCreate(PETSC_COMM_WORLD,&b);CHKERRQ(ierr);
        ierr = VecSetFromOptions(b);CHKERRQ(ierr);
        ierr = VecLoad(b,fd);CHKERRQ(ierr);

        ierr = VecDuplicate(b,&x);CHKERRQ(ierr);

        ierr = PetscViewerDestroy(&fd);CHKERRQ(ierr);

        /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - -
                     Create the linear solver and set various options
           - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - */
        /*
           Create linear solver context
        */
        ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);

        /*
           Set operators. Here the matrix that defines the linear
system
           also serves as the preconditioning matrix.
        */
        ierr                                                          =
KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr);
```

```
       /*
         Set runtime options, e.g.,
             -ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol
<rtol>
         These options will override those specified above as long as
         KSPSetFromOptions() is called _after_ any other customization
         routines.
       */
       ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

       /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - -
                           Solve the linear system
          - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - */
       /*
          Solve linear system
       */
       ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);

       /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - -
                           Check solution and clean up
          - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - */
       /*
          Check the error
       */

       ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
       ierr         =         PetscPrintf(PETSC_COMM_WORLD,"Iterations
%D\n",its);CHKERRQ(ierr);
       //VecView(x,PETSC_VIEWER_STDOUT_WORLD);

       ierr                                                          =
PetscViewerBinaryOpen(PETSC_COMM_WORLD,"solution.petsc",FILE_MODE_WRIT
E,&viewer);CHKERRQ(ierr);
       ierr = VecView(x, viewer);CHKERRQ(ierr);
       ierr = PetscViewerDestroy(&viewer);CHKERRQ(ierr);

       /*
          Free work space.  All PETSc objects should be destroyed when
they
          are no longer needed.
       */
       ierr = VecDestroy(&x);CHKERRQ(ierr);
       ierr = VecDestroy(&b);CHKERRQ(ierr);
       ierr = MatDestroy(&A);CHKERRQ(ierr);
       ierr = KSPDestroy(&ksp);CHKERRQ(ierr);

       /*
```

```
          Always call PetscFinalize() before exiting a program.  This
routine
            - finalizes the PETSc libraries as well as MPI
            - provides summary and diagnostic information if certain
runtime
              options are chosen (e.g., -log_summary).
       */

       ierr = PetscFinalize();
       return 0;
     }
```

Modified Linear Solver

```
       */
       ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

       /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - -
                           Solve the linear system
          - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - */
       /*
          Solve linear system
       */
       ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);

       /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - -
                           Check solution and clean up
          - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - */
       /*
          Check the error
       */

       ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
       ierr         =         PetscPrintf(PETSC_COMM_WORLD,"Iterations
%D\n",its);CHKERRQ(ierr);
       //VecView(x,PETSC_VIEWER_STDOUT_WORLD);

       ierr                                                          =
PetscViewerASCIIOpen(PETSC_COMM_WORLD,"output/solution.petsc",&viewer)
;CHKERRQ(ierr);
       ierr = VecView(x, viewer);CHKERRQ(ierr);
       ierr = PetscViewerDestroy(&viewer);CHKERRQ(ierr);

       /*
```

```
           Free work space.  All PETSc objects should be destroyed when
they
           are no longer needed.
         */
         ierr = VecDestroy(&x);CHKERRQ(ierr);
         ierr = VecDestroy(&b);CHKERRQ(ierr);
         ierr = MatDestroy(&A);CHKERRQ(ierr);
         ierr = KSPDestroy(&ksp);CHKERRQ(ierr);

         /*
            Always call PetscFinalize() before exiting a program.  This
routine
              - finalizes the PETSc libraries as well as MPI
              - provides summary and diagnostic information if certain
runtime
                options are chosen (e.g., -log_summary).
         */

         ierr = PetscFinalize();
         return 0;
      }
```

Vita

W.K. Umayanganie Munipala is a Ph.D. student in the Computational Science Program at the University of Texas at El Paso. She has more than three years of research experience in the area of performance and productivity analysis in high performance computing. She holds a BSc focused on computer science, math, physics and a higher diploma in Information Technology from the University of Colombo, Sri Lanka. Her contact email address is umunipala@miners.utep.edu.