The role of MPI in development time: a case study

Lorin Hochstein
USC Information Sciences Institute
lorin@east.isi.edu

Forrest Shull
Fraunhofer Center Maryland
fshull@fc-md.umd.edu

Lynn B. Reid
University of Chicago
lynnreid@flash.uchicago.edu

***Abstract*— There is widespread belief in the computer science community that MPI is a difficult and time-intensive approach to developing parallel software. Nevertheless, MPI remains the dominant programming model for HPC systems, and many projects have made effective use of it. It remains unknown how much impact the use of MPI truly has on the productivity of computational scientists.**

In this paper, we examine a mature, ongoing HPC project, the Flash Center at the University of Chicago, to understand how MPI is used and to estimate the time that programmers spend on MPI-related issues during development. Our analysis is based on an examination of the source code, version control history, and regression testing history of the software. Based on our study, we estimate that about 20% of the development effort is related to MPI. This implies a maximum productivity improvement of 25% for switching to an alternate parallel programming model.

Keywords: MPI, debugging, effort, productivity, case study

I. INTRODUCTION

Developing software to run on high-performance computing (HPC) systems is widely regarded as a difficult problem [3, 14, 16]. MPI [24], which is the dominant parallel programming model for writing software for HPC systems, has been identified as a major factor in the difficulty of developing HPC codes. Many parallel programming models (e.g., UPC [8], StarP [6], Chapel [5], X10 [7]) and frameworks (e.g., POOMA [30], Cactus [15], Sierra [11], Charm++ [21]) have been developed with the goal of improving the productivity of computational scientists by simplifying the task of developing parallel code that runs efficiently on large-scale HPC systems.

Previously, we have examined the impact of using MPI versus other parallel programming models through controlled experiments using graduate students as subjects solving toy problems [18, 19, 20]. In this paper, we present a study that examines the effects of MPI on programmer effort in the context of a real, large-scale HPC software project.

In particular, we are interested in understanding how MPI is used in the context of a larger project with a team of developers, and estimating its impact on a mature, ongoing software development environment. We want to understand how much of the actual software development of a larger project genuinely involves MPI once the project is at a stage where it is being used to do real science.

To address these questions, we have performed a case study [32] of the Flash Center at the University of Chicago. Such studies are increasingly being used to study HPC development time issues [4]. By focusing on an in-depth analysis of a single project, we hope to gain insights into the software development process of computational scientists and provide initial estimates on the impact of MPI that will serve as a starting-off point for future studies.

A. Background on the Flash Center

The goal of the Flash Center is to study thermonuclear flashes, events of rapid or explosive thermonuclear burning that occur on the surfaces and in the interiors of compact stars. Hosted at the University of Chicago, the Flash Center is one of the five original ASC-Alliance centers [17], started around 1997 and funded by the National Nuclear Security Administration. These centers are provided access to unclassified HPC resources that are owned by the Department of Energy.

To conduct this research, the Flash Center has developed FLASH, a modular and parallelized simulation code [13]. Of the code bases at the five original ASC-Alliance centers, the FLASH code is the largest and has the most number of external users. FLASH is an MPI-based, coupled, multi-physics application, written in FORTRAN and C. At the time this study was conducted, the Flash Center was in the process of developing a new version of FLASH (3.0) as well as maintaining a stable version for production work (2.5).

B. Prior beliefs

While this case study is largely exploratory, we did have several prior beliefs about MPI usage going into this study. MPI incurs large up-front costs, but after the initial development, we expected the maintenance costs due to MPI to be much less, because of the use of abstraction in the code to isolate the low-level communication details. Our previous studies have shown that larger HPC projects build a framework layer on top of MPI so that they do not have to modify as much MPI code over time [17]. We expected MPI usage to be compartmentalized in terms of code (i.e., MPI calls are not evenly distributed across the code base but are organized together) and in terms of developers (some developers may work on MPI-related code, but others may not touch MPI code at all). We also expected that only a small fraction of the MPI library would be used in practice.

This research was performed under the DARPA HPCS project under Air Force grant FA8750-05-1-0100 to the University of Maryland. The FLASH code is supported by the U.S. Department of Energy under Grant No. B523820 to the Center for Astrophysical Thermonuclear Flashes at the University of Chicago. The PARAMESH software described in this work was developed at the NASA Goddard Space Flight Center and Drexel University under NASA's HPCC and ESTO/CT projects and under grant NNG04GP79G from the NASA/AISR project.

SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

C. How analysis was done

We have applied an *archeological* approach [27], analyzing data that is generated naturally by the scientists as they develop the software. The three primary sources of data we used were:

- Source code
- Version control repository
- Regression testing results

The Flash Center project uses Subversion[1] [28] for doing version control. This repository maintains a history of all changes to the code base. In addition, the Flash Center has developed a regression testing system called FlashTest. Each night, FlashTest executes numerous tests of FLASH-related code on multiple architectures and multiple compilers. FlashTest maintains a history of which tests passed and failed, accessible over a web interface.

The study described in this report analyzed repository data from 02/22/2005 – 01/27/2008 and regression testing data from 11/08/2006 – 01/27/2008. To analyze the source code, we used SCLC-UH[2], which we modified slightly to deal with FLASH-specific files.

D. Scope of FLASH

Table I shows the size of the FLASH code base in terms of *source lines of code* (SLOC), which counts all lines in a file except for blanks and comments. The FLASH code is roughly 430 KSLOC, and is a combination of FORTRAN 77/90 (87%) and C (7%). The rest consists primarily of setup scripts, including FLASH-specific configuration files, FLASH-specific parameter files, makefiles, Python scripts, Perl scripts, and shell scripts. This count only includes the code in the "source" directory. It excludes various other software tools that are used by the Flash Center but are not part of the main simulation, such as the FlashView software used to visualize the output of the FLASH code, and the FlashTest automated regression test system. For the remainder of this paper, we focus entirely on the FORTRAN and C files (which we refer to as *source files*).

Note that FLASH is not a monolithic software package. Instead, it is designed as a collection of components. At compile-time, the user selects which components will be used to build an executable instance of the FLASH code. The FLASH-specific configuration files and the Python scripts assemble these components into a particular build of the system.

A significant amount of the FLASH code base consists of a reused package, PARAMESH [23, 25, 26], which handles the adaptive mesh capabilities of the code. PARAMESH is funded and developed separately; members of the Flash Center don't maintain this code directly. Since FLASH supports three versions of the package, PARAMESH code is represented three times in the code base.

TABLE I. SIZE OF CODE BASE

	With PARAMESH		Without PARAMESH	
Language	**SLOC**	**% of total**	**SLOC**	**% of total**
FORTRAN	377,149	87.2%	148,118	80.3%
C	29,058	6.7%	11,661	6.3%
Parameter (FLASH)	16,566	3.8%	16,566	9.0%
Config (FLASH)	3,841	0.9%	3,841	2.1%
Make	2,247	0.5%	1,403	0.8%
Perl	1,753	0.4%	1,753	1.0%
Python	1,576	0.4%	889	0.5%
Shell	475	0.1%	221	0.1%
Total	*432,665*	*100%*	*184,452*	*100%*

II. ESTIMATING THE IMPACT OF MPI

A. MPI by code volume

If the amount of time spent on MPI-related development issues is proportional to the time spent editing code that contains MPI, then we can estimate how much time is spent on MPI by examining the source code alone. To estimate this, we need an operational definition of "MPI-related code". Here we adopt the convention that if a file contains at least a single call to the MPI library then it is MPI-related.

The simplest way to quantify the impact of MPI is to count how many of the source files involve MPI-related code. As mentioned earlier, we consider source files to be any FORTRAN or C file. Table II shows the amount of MPI code, considering both the number of files, and the amount of code. Note the large impact of PARAMESH, which comprises a sizeable fraction of the total source code, as well as containing a great deal of the MPI code. Since PARAMESH is developed and maintained external to the Flash Center, we believe the "Without PARAMESH" entries are better indicators of the amount of effort related to MPI, showing that between **7** and **15%** of the code base relies on MPI.

In addition, the large difference in the percentages when counting using number of files versus source lines of code suggests that files that contain MPI tend to be larger. The scatter plot in Fig. 1 confirms this and suggests that there is a relationship between the size of a file and the number of MPI calls within the file. A correlation analysis yields in an R^2=0.49, which implies that 49% of the variance in the number of MPI calls in a file is explained by the size of the file.

TABLE II. AMOUNT OF MPI CODE

		# of files	SLOC
With PARAMESH	MPI	471 files	213,397 SLOC
	Total	2625 files	406,207 SLOC
	Percentage	*17.9%*	*52.5%*
Without PARAMESH	MPI	145 files	23,335 SLOC
	Total	1925 files	159,779 SLOC
	Percentage	***7.5%***	***14.6%***

[1] http://subversion.tigris.org
[2] http://code.google.com/p/sclc

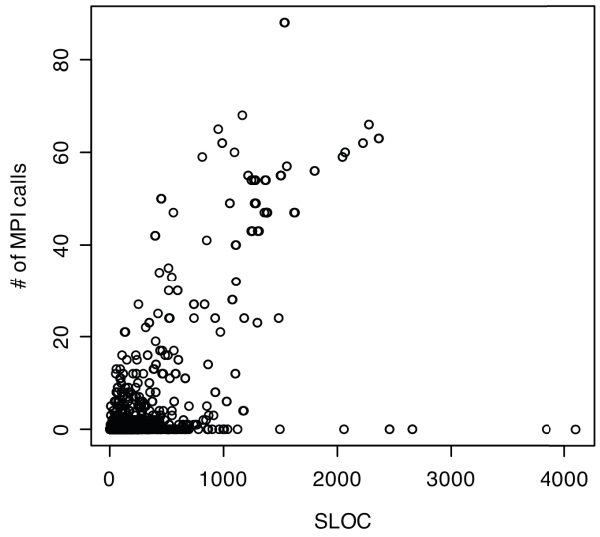

Figure 1. MPI calls vs. Source Lines of Code (SLOC)

B. *MPI by overall activity*

The data from the previous section provides a sense of how much MPI code is in the project, but it assumes that the time spent developing MPI code is directly proportional to the amount of MPI code. However, some code may be touched only once, and other code may be modified many times over the lifetime of the project. In particular, there is reason to suspect that the MPI code typically may not be touched over and over. If the developers have built infrastructure on top of MPI, then all of the MPI costs may be paid up-front, and so we might expect little MPI effort later in the lifetime of the project. This would be a powerful argument that mature projects do not suffer from the use of MPI because they have already undergone the pain of MPI development. On the other hand, it may be that this level of encapsulation is not possible to maintain over time because the addition of unforeseen features necessitates changes to the software architecture [12]. In the case of FLASH, we know that one of the major features in version 3.0 is the introduction of new algorithms related to the adaptive mesh, which we expect to require extensive MPI development.

An alternate way to measure the role of MPI is to measure the amount of time that the developers spend editing MPI code. While this is very difficult to measure directly, we can estimate this value using the version control history of the software. As previously mentioned, the Flash Center uses Subversion to manage their repository. Subversion keeps a history of all changes committed to the repository ("commits"), and assigns a numerical ID called a *revision number* to each commit. Using Subversion, it is simple to derive a *changeset* (the set of modified files associated with a commit) by comparing two successive revision numbers.

To estimate the impact of the use of MPI on development time, we calculate the percentage of MPI-related commits. We define an MPI commit as a changeset that contains at least one file with at least one MPI call. Therefore, if ten files are committed to the repository in revision 2234, and one of those files contains an MPI call, we consider that commit to be MPI-related. Even though some edits to files that contain MPI may not be MPI-related, we chose this metric to obtain a reasonable upper bound on MPI-related activity. In addition, we felt that trying to identify whether individual changes were MPI-related through a manual inspection of the code would be prohibitively time-consuming and might require that the analyst have significant understanding of the code.

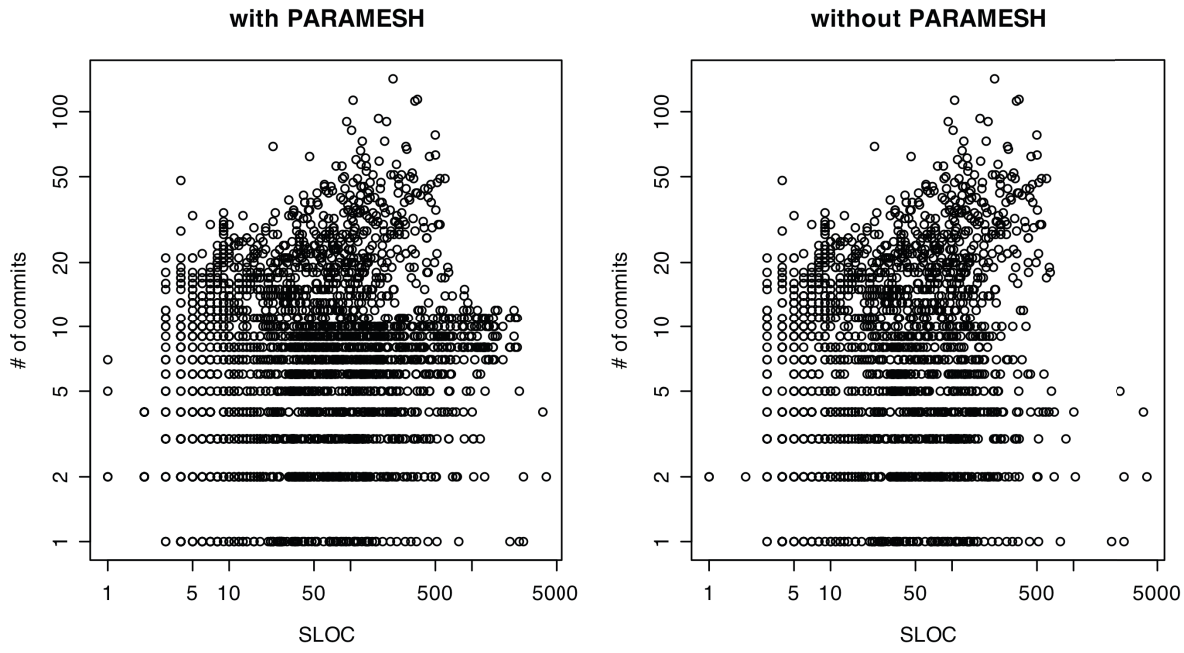

Figure 2. Number of times a file was committed versus the size of the file

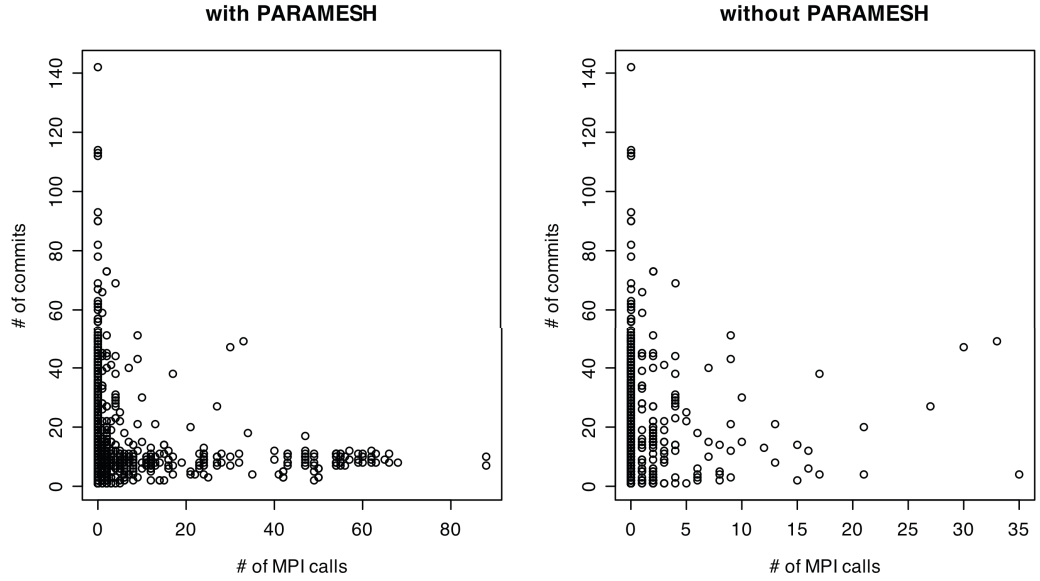

Figure 3. Number of Commits vs. MPI calls contained in the file

In the interval of data that we analyzed, there were *4110* commits that involved source files. Of these commits, there were *1220* MPI-related commits. Therefore, about **29.7%** of commits were MPI-related. While we expected there to be a relationship between the size of a file and the number of commits to a file, we saw no such correlation, as seen in Fig. 2. Similarly, we observed no relationship between the number of MPI calls and number of commits to a file, as shown in Fig. 3. While there may appear to be a negative correlation in Fig. 3 by visual inspection, a linear regression analysis in all cases yields an $R^2<.01$, which suggests that there are no meaningful linear trends in the data. Excluding the PARAMESH code does not change the results of the analysis.

C. *MPI by corrective maintenance (bugfixes)*

There are several reasons why a developer may make modification to source code. The traditional software engineering literature distinguishes between four kinds of software maintenance [29]:

- *corrective*: fixing defects (i.e., bugs)
- *adaptive:* adapting software to a new environment (e.g., porting to new machines)
- *perfective/enhancement:* adding new features
- *preventative:* reorganizing the code to simplify future changes

Using data from the FlashTest regression test database, we estimated the activity related to corrective maintenance. This database indicates which regression tests passed and failed on which days. Fig. 4 shows the number of regression tests passing (green) and failing (red) over time.

If we consider that regression test X failed on 03/26/07 and passed on 03/27/07, then we can assume that the developers modified the code to fix the test on 03/27/07. In FLASH, each regression test is associated with a set of directories that contain the source file used to build the test. We can make an educated guess about which changes on 03/27/07 were associated with fixing that test by looking at all of the commits on that day that involved files in the directories associated with regression test X.

In the data, we observed 3641 incidents where a regression test was broken on one day and fixed on the next day. Of these, we were able to associate 1850 of these with commits to source files, which is about half. In the other cases, we must assume that the regression test was fixed by some means other than modifying a source file, such as a scientist approving new results. In some cases, multiple commits were associated with a particular fix of a regression test.

TABLE III. COMMITS RELATED TO BUG-FIXING

	# of commits
Non-MPI	2366 (69.9%)
MPI	1021 (30.1%)
Total	**3387 (100%)**

Table 3 summarizes the results of this analysis. Based on these results, we estimate about **30%** of the activity related to fixing bugs involved MPI code. This is consistent with the results from Section IIB, and suggests that bug-fixing activity does not look different from the other forms of development activity with respect to how much of it involves MPI.

D. Distribution of effort across developers

Large HPC projects such as FLASH require multiple developers. We expect that the amount of MPI-related work will not be distributed evenly across the developers. At a coarse-grained level, we can see this division of labor in the Flash Center, as it is divided up into distinct groups (e.g. astrophysics, basic physics, computational physics and validation), and it is the responsibility of the Flash Code Group to maintain the software. This group is much more involved in modifying MPI code than any of the other groups. Here, we are concerned with variation within the Code Group.

As in Section IIB, we use commits as an estimate of the time spent on editing MPI-related code. Table IV shows commit activity for the ten developers who had the highest number of commits to source code. The table shows the number of commits to source code (raw, and as a percentage of the total), the number of commits that involved at least one MPI-related file (raw, and as a percentage of total), and an estimate of how much time each developer spends on MPI-related issues based on this commit data. The median time spent in MPI-related development is **17.5%**, with eight out of the ten developers spending less than 29.1% of their time on MPI-related issues. Note that developers H and I spend over half of their time on MPI-related issues.

TABLE IV. COMMITS ACROSS DEVELOPERS

Developer	# of source commits	# of MPI commits	% MPI-related development
A	666 (16.2%)	112 (12.3%)	16.8%
B	630 (15.3%)	110 (12.1%)	17.5%
C	557 (13.6%)	162 (17.9%)	29.1%
D	484 (11.8%)	81 (8.9%)	16.7%
E	358 (8.7%)	36 (4.0%)	10.1%
F	286 (7.0%)	43 (4.7%)	15.0%
G	246 (6.0%)	41 (4.5%)	16.7%
H	241 (5.9%)	122 (13.5%)	50.6%
I	115 (2.8%)	81 (8.9%)	70.4%
J	102 (2.5%)	19 (2.1%)	18.6%
Total	4110	1220	

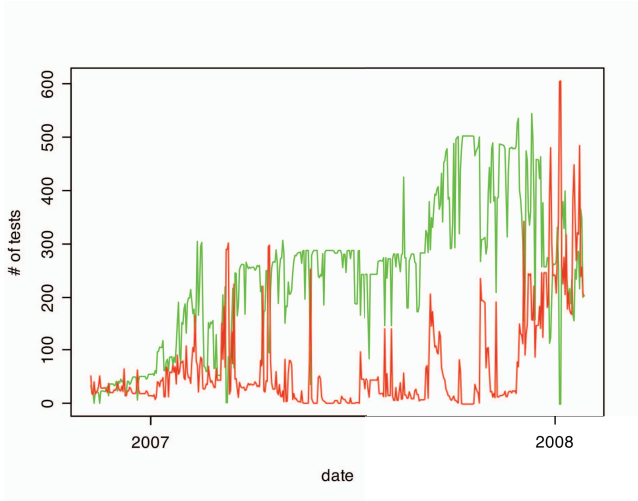

Figure 4. Regression tests passing (green) and failing (red)

III. HOW MPI IS USED IN CODE

A. Distribution of MPI code

In addition to how much time is spent on MPI-related issues, we are also interested in understanding how MPI is used throughout FLASH. Fig. 5 shows a visual representation called a treemap [31] of the source code. Each box represents a source file. The size of the box is proportional to the number of source lines of code (SLOC), and the directories are represented as enclosing boxes, with files in the same directory appearing in the same box. The color indicates the number of MPI calls: bright green indicates many calls (max=88), and darker indicates fewer calls (min=0). The color is proportional to the logarithm of the number of MPI calls to increase the visibility of files that have small but non-zero number of calls. The three versions of PARAMESH are indicated with orange rectangles. From this plot, it is evident that most of the MPI-related code is contained within PARAMESH, as suggested by Table II.

Fig. 6 shows the same data without the PARAMESH code. The color scale has changed, as the maximum number of MPI calls in a file is 35. We can see in this figure that the MPI-related files in the non-PARAMESH code are scattered about the code base.

B. Use of MPI functions

While fully functional MPI programs can be written with six basic functions (MPI_Init, MPI_Finalize, MPI_Comm_Size, MPI_Comm_Rank, MPI_Send, MPI_Recv), the MPI 1.1 standard [24] defines 215 separate functions. We wanted to understand how extensively the FLASH code used the MPI library.

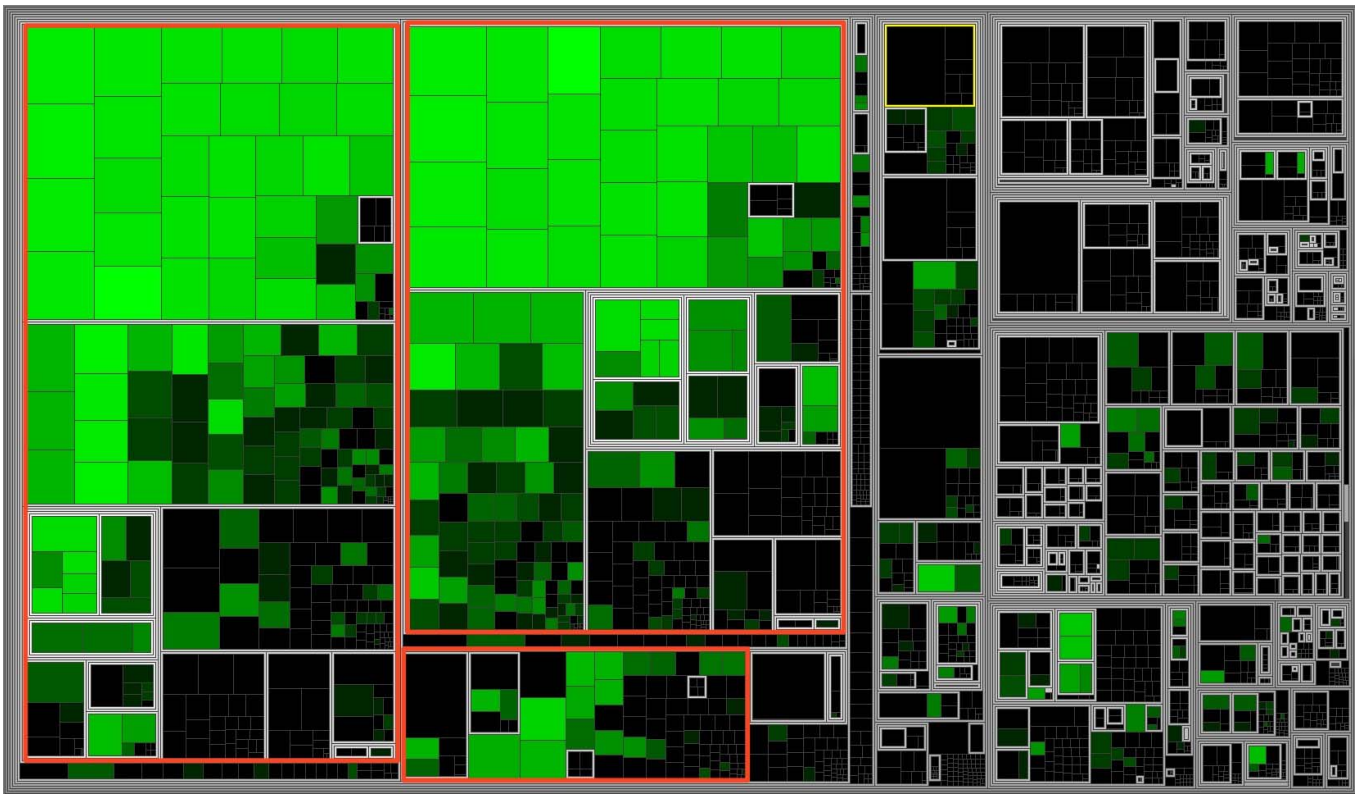

Figure 5. Treemap of source, highlighting MPI calls. PARAMESH directories outlined in orange

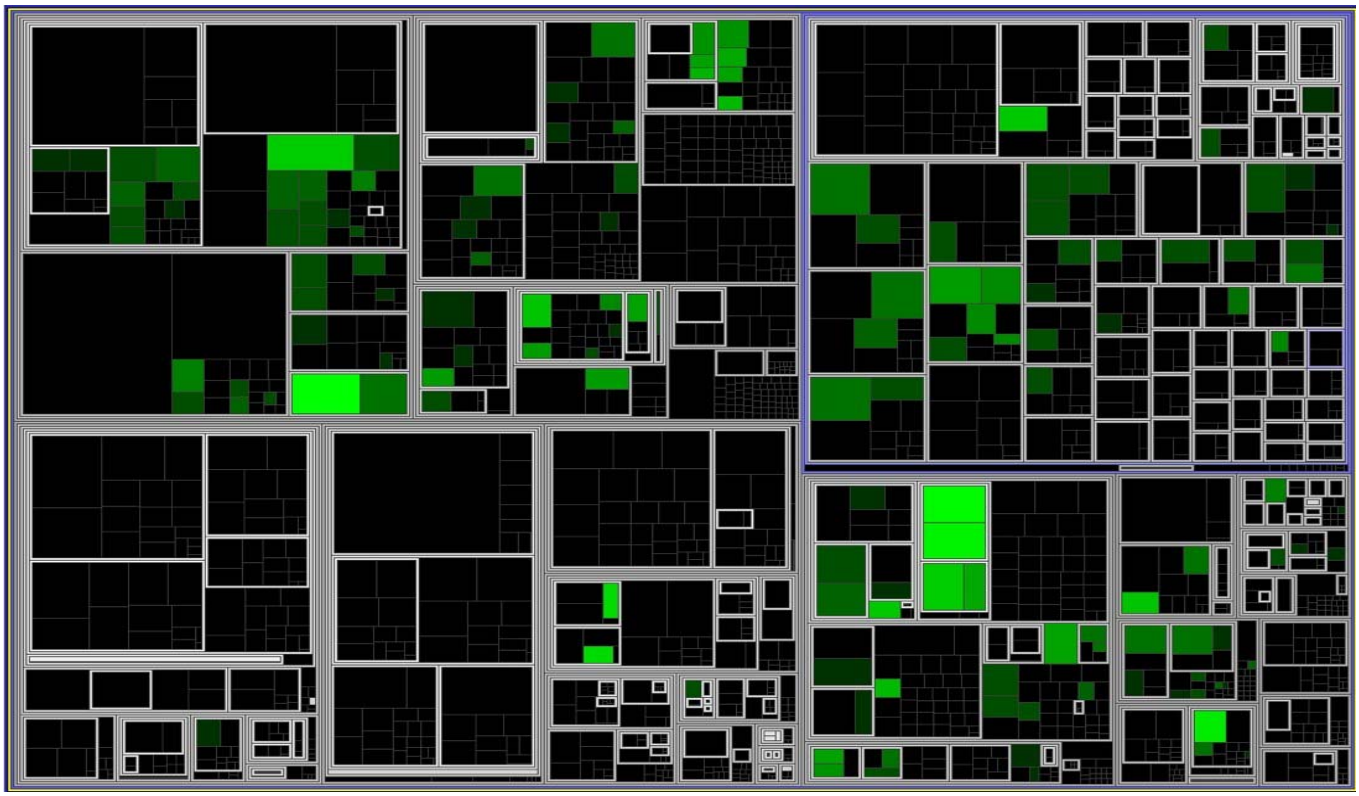

Figure 6. Treemap of source, highlighting MPI calls, without PARAMESH node

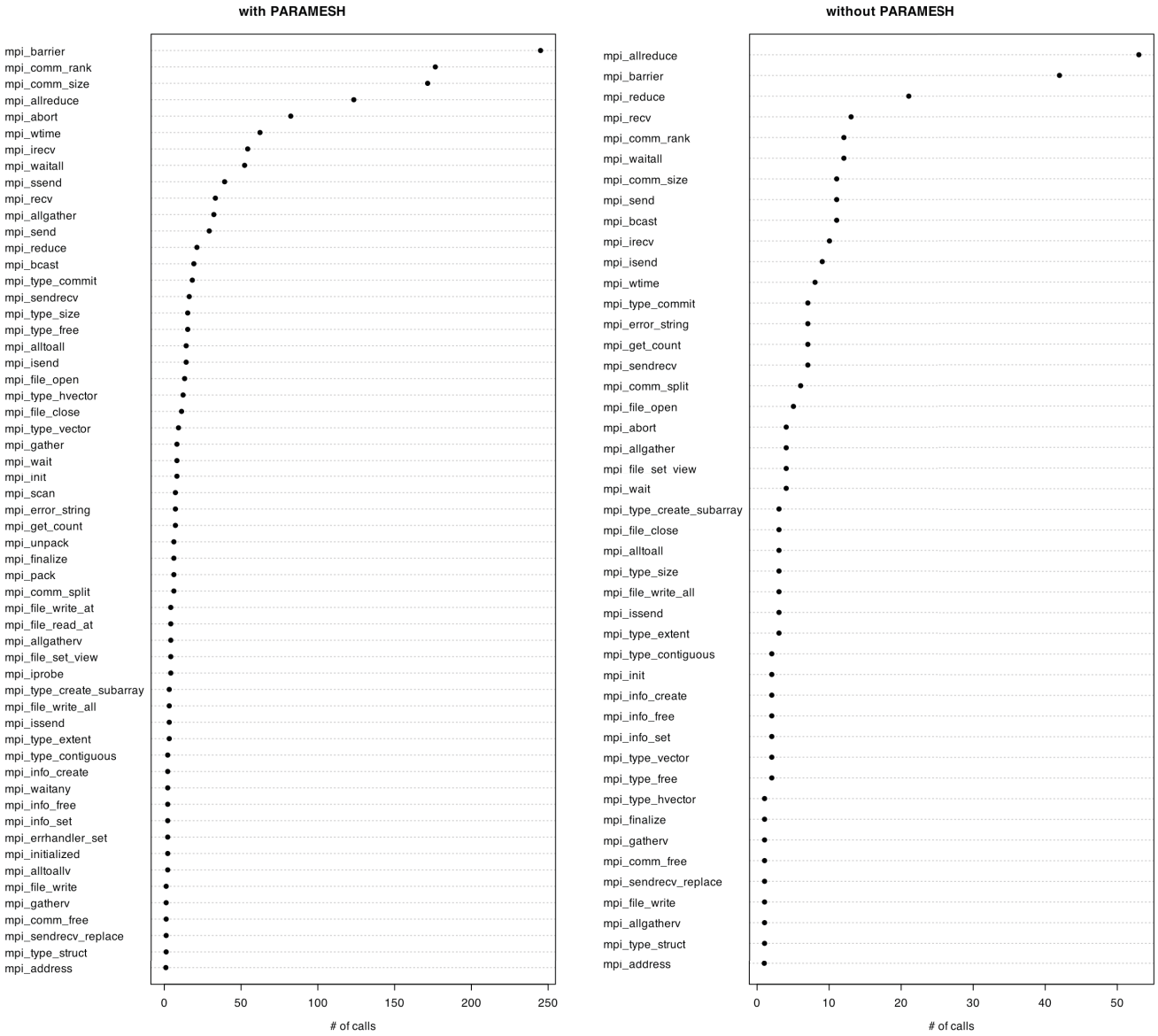

Figure 7. Frequency of MPI calls

The entire FLASH code makes use of 57 MPI functions, which represents 27% of the full range of available functions. The distribution of MPI function calls is shown as a Cleveland dot plot [9] in Fig. 7. The large numbers of barriers in the code are mostly PARAMESH-related, in either the PARAMESH files or in test files that call PARAMESH code[3]. We also note that FLASH uses more of the MPI library than we expected, although many functions are used only once. The MPI calls appear to have a Pareto distribution, as 80% of the total number of MPI function calls are due to 23% of the MPI functions in the code.

[3] In the most recent version of PARAMESH, all barriers have been removed.

If we exclude the PARAMESH code in our analysis, we see that FLASH makes use of 45 MPI functions and is dominated by barriers and allreduce. Here the distribution of MPI calls across functions is more evenly distributed, as 80% of the calls are due to 40% of the functions.

IV. DISCUSSION

Table V summarizes the estimates of MPI-related activity using the five different estimates we examined in this paper.

TABLE V. ESTIMATES OF MPI-RELATED TIME

# of files (without PARAMESH)	7.5%
SLOC (without PARAMESH)	14.6%
Activity (commits)	29.7%
Debugging activity	30.1%
Median developer activity	17.5%

This gives us an estimate of the percentage of work spent on MPI on the order of 10-30%, with a mean of about 20%. Let us assume that an alternative to MPI could completely eliminate the need for all of this development activity, and would have no impact on any other aspect of the project such as program performance. By Amdahl's Law [1], this implies a maximum programmer productivity increase of

$$\frac{1}{1-0.2} = 1.25$$

which gives an upper bound on productivity improvement of an MPI replacement at about 25%. Given the risks associated with adopting a new technology, and the startup costs associated with adopting a new technology, it is unsurprising that mature HPC projects such as the Flash Center would be reluctant to experiment with new parallel programming technologies that promise large boosts in programmer productivity [2].

The frequency and distribution of MPI function calls in FLASH suggests that the MPI code mostly uses only a small number of MPI function calls and is mostly compartmentalized to within certain subsystems, particularly those related to the mesh. Despite this, we do see many functions with a very small number of MPI function calls scattered about the code, and we see many MPI functions that are used very infrequently.

V. DISCUSSION WITH DEVELOPERS

To cross-check our results, we asked the members of the Flash Center Code Group about whether they felt the results were consistent with their experience. The group expressed that the figure of 20% of code-development effort being devoted to MPI is too high, and several people guessed that it would be closer to 10%. The developers attributed the discrepancy to four factors:

1. *Most of the MPI-related calls are found in libraries that were not developed by Flash members.* While the developers may make changes to routines that call these libraries, but they don't actually write the MPI code themselves.

2. *The effort of changes to the PARAMESH library will be over-estimated because multiple PARAMESH versions are supported.* For example, when working on a change in PARAMESH 3 calls, nearly the exact same change will have to be made for the PARAMESH 2 calls, and possibly PARAMESH 4.

3. *The study was performed during an atypical period.* During the particular time period that the study was conducted, there were some efforts to develop new parallel algorithms. However, historically, the parallel algorithms are confined to external libraries. While everybody in the group needs to be aware of MPI issues, they do not typically work with MPI very much.

4. *Counting edits to files that contained MPI calls will overestimate MPI.* Edits done to files that contain MPI may not necessarily be related to MPI-related issues. For example, a developer may edit a routine that has one MPI call within it, but the edits are completely independent of the MPI call, such as modifying `DO` loop indices above the call.

VI. THREATS TO VALIDITY

There are numerous threats to validity in this case study. The most significant threat is *construct validity*, the idea that the measures we have used do not capture the phenomenon of interest. Because we were not able to directly measure programmer time, we had to use indirect measures whose accuracies are unknown. For example, we do not know how well we can predict programmer effort from commits to the source code repository: a small change in a single commit may have come from five minutes or programming time, or may be the result of days of debugging. Further studies would be needed to evaluate how well measures such as number of commits or number of source lines of code correlate with the amount of time spent on development. Both of these measures are possibly influenced by the individual programmer's coding style. For example, the number of commits will vary depending on whether the coder wants to make sure that work is saved in small increments or wants to ensure that code is absolutely correct before entering the code base. Since the amount of MPI-related development varies significantly by developer, as discussed in Section 2.4, if the programmers who work more on MPI happen to commit more often than those who do not, it could greatly skew the results of this study.

Focusing on code that contains MPI calls may not be indicative of the total impact of MPI, as suggested by the developers in the previous section. We counted any modification to a file containing MPI as an "MPI issue," but these modifications may not have actually touched the MPI calls at all. If a change involved multiple files but only a single file was MPI-related, then the entire change was associated with MPI, which may result in overestimating the impact of MPI. As an alternative, we could have flagged only edits to MPI function calls as MPI-related, but this would lead to underestimates because it would not consider edits that ultimately affect the arguments of MPI calls. It is a difficult problem in general to identify which edits are truly MPI-related.

In addition, in this study we have assumed that all MPI-related activity is directly related to the time spent editing code. However, it may affect other activities as well, such as the time spent maintaining makefiles [22]. In addition, debugging MPI-related algorithms often requires extensive use of scarce high-performance systems. Furthermore, activities such as algorithm design cannot be captured in these types of archeological studies.

Finally, there is the challenge in interpreting how the results of a case study might apply to other projects. In particular, this study has focused on a mature application during its transition

from version 2.5 to version 3.0. We do not expect the same results to hold if we examined, for example, the development of a project working toward version 1.0.

VII. Conclusion

In this paper, we have examined the role of MPI on a large-scale HPC code development project, and characterized the degree to which coding activities deal with MPI specifically. We addressed the issues inherent in any study of software engineering issues based on archaeological data by applying a rigorous case study methodology using triangulation: combining different measures to provide insight on the same topic. The general agreement among our measures (number of files, number of SLOC, number of commits, number of debugging activities) provides confidence that the results, showing that MPI-specific issues make up a small percentage of the overall coding activities, are indicative of real phenomena.

The implications of this are important for the HPC community, as these results provide some bounds on the amount of improvement that can be expected from next-generation replacements for MPI. Although potentially significant improvements in coding effort could be achieved, at least in the ideal case, replacing MPI and keeping other factors constant should not be expected to produce increases in productivity on the order of 10X, the target goal envisioned by the DARPA HPCS program [10].

As we mentioned above, these findings are based on experiences on a single, mature system and should not be extrapolated for all potential MPI applications. Additional case studies would be useful, which could quantify the impact of MPI and its use on other real-life HPC codes. We hope that such work may also help disseminate best practices that have been found for managing the complexity and difficulty of MPI and increase team productivity.

Acknowledgment

The authors gratefully acknowledge the help of the Flash Center Code Group, without whom this work would not have been possible.

References

[1] G.M. Amdahl, “Validity of the single-process approach to achieving large scale computing capability”, AFIPS Spring Joint Computer Conference, 1967, Atlantic City, New Jersey

[2] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz, “Understanding the high performance computing community: A software engineer’s perspective”, IEEE Software, July/August 2008.

[3] M. R. Beinoff and E. D. Lazowska, "Computational science: Ensuring America's competitiveness / President's Information Technology Advisory Committee (PITAC)," National Coordination Office for Information Technology Research & Development, Arlington, VA 2005.

[4] J. C. Carver, R. P. Kendall, S. E. Squires, D. E. Post, “Software development environments for scientific and engineering software: A series of case studies”, International Conference on Software Engineering, pp.550-559, May 2007.

[5] B. L. Chamberlain, D. Callahan, H. P. Zima. “Parallel programmability and the Chapel language”. International Journal of High Performance Computing Applications, Vol. 21, No. 3, pp291-312, August 2007.

[6] R. Choy and A. Edelman, “MATLAB*P 2.0: A unified parallel MATLAB”, MIT DSpace, Computer Science collection, January 2003.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing”, 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages and Applications (OOPSLA ‘05), 2005.

[8] W.W. Carlson, D.E. Culler, K.A. Yellick, E. Brooks and K. Warren, “Introduction to UPC and language specification”, Center for Computing Sciences Technical Report, CCS-TR-99-157, May 1999.

[9] W. S. Cleveland, "Graphical methods for data presentation: Full scale breaks, dot charts, and multibased logging," The American Statistician, Vol. 38, November 1984.

[10] J. Dongarra, R. Graybill, W. Harrod, R. Lucas, E. Lusk, P. Luszczek, J. McMahon, A. Snavely, J. Vetter, K. Yelick, S. Alam, R. Campbell, L. Carrington, T.-Y. Chen, O. Khalili, J. Meredith, and M. Tikir, " DARPA's HPCS program: History, models, tools, languages," in Advances in Computers, Vol. 72, Elsevier, June 2008.

[11] H.C. Edwards and J. R. Stewart. “Sierra, a software environment for developing complex multiphysics applications”. In K. J. Bathe, editor, Computational Fluid and Solid Mechanics. Proc. First MIT Conf., pp 1147-1150, Elsevier, 2001.

[12] S. G. Eick, T. L. Graves, A. F.Karr, J. S. Marron, A. Mockus, "Does code decay? Assessing the evidence from change management data ," IEEE Transactions on Software Engineering, Vol. 27, No.1, pp.1-12, Jan 2001

[13] B. Fryxell, K. Olson, P. Ricker, F.X. Timmes, M. Zingale, D.Q. Lamb, P. MacNeice, R. Rosner, J.W. Truran, and H. Tufo, “FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes”, The Astrophysical Journal Supplement Series, Vol.131, Iss. 1, pp. 273-334, Nov 2000.

[14] S. L. Graham, M. Snir, and C. A. Patterson, "Getting up to speed: The future of supercomputing," National Academies Press, 2004.

[15] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. “The Cactus framework and toolkit: design and applications,” Vector and Parallel Processing - VECPAR 2002, 5th International Conference. Springer, 2003.

[16] Federal plan for high-end computing: Report of the high-end computing revitalization task force (HECRTF), May 10, 2004.

[17] L. Hochstein and V. R. Basili, "The ASC-Alliance projects: A case study of large-scale parallel scientific code development," IEEE Computer, Vol. 41, March 2008.

[18] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. R. Basili, J. Hollingsworth, and M. Zelkowitz, "HPC programmer productivity: A case study of novice HPC programmers," ACM/IEEE Conference on Supercomputing (SC ’05), 2005.

[19] L. Hochstein, T. Nakamura, V. R. Basili, S. Asgari, M. V. Zelkowitz, Jeffrey K. Hollingsworth, Forrest Shull, Jeffrey Carver, Martin Voelp, Nico Zazworka, and Philip Johnson, “Experiments to Understand HPC Time to Development”, CTWatch Quarterly, Vol 2, No. 4A, November 2006.

[20] L. Hochstein, V. Basili, U. Vishkin, and J. Gilbert, “A pilot study to compare programming effort for two parallel programming models”, Journal of Systems and Software, in press.

[21] L. V. Kale and S. Krishnan, “Charm++: parallel programming with message-driven objects,” G.V. Wilson, P. Lu (Eds.), Parallel Programming Using C++; MIT Press, Cambridge, MA, pp. 175–213, 1996.

[22] G. Kumfert, and T. Epperly, “Software in the DOE: The Hidden Overhead of ‘The Build’”, Lawrence Livermore National Laboratory Technical Report, UCRL-ID-147343, Feb 28, 2002.

[23] P. MacNeice, K. M. Olson, C. Mobarry, R. deFainchtein, and C. Packer, "PARAMESH : A parallel adaptive mesh refinement community toolkit.", Computer Physics Communications, Vol. 126, pp.330-354, 2000.

[24] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, Version 1.1, June 1995.

[25] K. Olson and P. MacNeice, "An Over of the PARAMESH AMR Software and Some of Its Applications",in Adaptive Mesh Refinement-Theory and Applications, Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, Series: Lecture Notes in Computational Science and Engineering, vol. 41, eds. T. Plewa, T. Linde, and G. Weirs, Springer, Berlin, 2005.

[26] K. Olson, "PARAMESH: A Parallel Adaptive Grid Tool", in Parallel Computational Fluid Dynamics 2005: Theory and Applications: Proceedings of the Parallel CFD Conference, College Park, MD, U.S.A., eds. A. Deane, A. Ecer, G. Brenner, D. Emerson, J. McDonough, J. Periaux, N. Satofuka, and D. Tromeur-Dervout, Elsevier, 2006.

[27] D. E. Perry, N. A. Staudenmayer, and L. G. Votta, "Understanding and improving time usage in software development," in Software Process. Vol. 4, A. Fuggetta and A. Wolf, Eds.: John Wiley and Sons, 1996.

[28] C. Pilato, B. Collins-Sussman, B. Fitzpatrick, Version Control with Subversion, O'Reilly Media, 2008.

[29] R.S. Pressman, Software engineering: a practitioner's approach, McGraw-Hill, 2001.

[30] J.V.W. Reynders, P.J. Hinker, J.C. Cummings, S.R. Atlas, S. Banerjee, W.F. Humphrey, S.R. Karmesin, K. Keahy, M. Srikant, and M. Tholburn, "POOMA: A framework for scientific simulation on parallel architectures", in Parallel Programming for C++, pp. 547-558, MIT Press, 1996.

[31] B. Shneiderman, "Tree visualization with tree-maps: A 2-d space-filling approach," ACM Transactions on Graphics., Vol. 11, pp. 92-99, 1992.

[32] R. K. Yin, Case study research: Design and methods, Third ed.: Sage Publications, 2002.