ABSTRACT

Title of dissertation: RECURRING SOFTWARE DEFECTS IN HIGH END COMPUTING

Taiga Nakamura
Doctor of Philosophy, 2007

Dissertation directed by: Professor Victor R. Basili
Department of Computer Science

This dissertation presents an empirical approach for building, storing, and evolving knowledge about domain-specific software defects. It is based on an iterative methodology, where the patterns of defects are derived from the combination of pattern identification heuristics and the validation and evolution of the patterns by domain experts. The approach consists of three main activities: (1) Pattern development through reading-based defect analysis of source code versions, (2) Reactive pattern refinement through a variation of structured interviews, and (3) Packaging available knowledge about the defects, such as symptoms and advice about prevention, into derivative artifacts, such as lectures and tools.

This approach has been applied to the domain of high performance computing (HPC), to build defect patterns that consist of a classification scheme for defect types and subtypes, and specific defect examples representing that type. For each defect sub-type there is provided a description, a set of symptoms that can help identify if the defect is present, a set of potential causes of the defect, and suggestions about cures and preventions.

I verified the feasibility of the methodology within the constraints of available research opportunities in the HPC domain. I conducted several empirical studies, evaluating the reliability of the heuristics, the generality of the defect pattern data though expert opinion, and the usefulness of the patterns through classroom studies and expert opinion.

The main outputs produced are an experience base which stores and shares these patterns, so that HPC practitioners can access them at various levels of abstraction and submit feedback and an evolved and evaluated set of educational materials that can be used to help minimize the defects made by novice programmers.

RECURRING SOFTWARE DEFECTS IN HIGH END COMPUTING

By

Taiga Nakamura

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Acknowledgements

First, I would like to thank all of the participants of empirical studies. This work would not have been possible without so many people who were willing to participate in this work.

I am most grateful to my advisor Vic Basili for guiding my dissertation research. During my time as his student, he always gave me his full support and continued to express his faith in me and in my ability to complete this dissertation. It has been a great honor to work with him.

I thank the current and former members of the Empirical Software Engineering Group at the University of Maryland: Vic Basili, Marv Zelkowitz, Forrest Shull, Jeff Carver, Lorin Hochstein, Sima Asgari, Daniela Cruzes, Nico Zazworka, Rola Alameh, Alessandro Sarcia, and Patricia Costa. I also thank the members of the HPCS productivity team, especially Jeff Hollingsworth, Philip Johnson, Nicole Wolter, and Michael McCracken, for their comments and feedback.

I would also like to thank the professors and experts who were willing to let us collect data through interviews and classroom studies: Henri Casanova, Jacqueline Chamé, Bill Dorland, Anshu Dubey, Alan Edelman, John Gilbert, Mary Hall, Jeff Hollingsworth, John Lewis, Glenn Luecke, Ed Luke, Kristi Maschhoff, David Mizell, Aiichiro Nakano, Rodric Rabbah, Allan Snavely, Alan Sussman, Uzi Vishkin, and Klaus Weide.

I thank the members of my dissertation committee: Vic Basili, Marv Zelkowitz, Jeff Hollingsworth, Alan Sussman and Bill Dorland. Their valuable feedback has improved the quality of this work.

Thanks go to the students from the University of Mannheim who worked with me: Nico Zazworka, Martin Veolp, Andreas Anstock, Steffen Olbrich, and Nabi Zamani.

I would like to thank Lorin Hochstein who, I am sure, gave me more time than he wanted and also provided extensive feedback during the early and later stage of this work. The discussions with him were always stimulating and interesting.

Finally, I thank my wife, Noriko, for her support, love, and patience. She helped me find the strength to continue when I needed it the most.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Among the many goals of software engineering, preventing and detecting software defects (bugs) has always been an important yet challenging one; developers find debugging takes a considerable portion of development time; defects found at a later development phase tend to increase cost. Even worse, defects often remain uncaught in the production code, and many failures have resulted in major incidents.

Ten years ago, Lieberman [55] described debugging as “the dirty little secret of computer science”. He wrote that “*despite the progress we have made in the past 30 years ... we still face some embarrassing facts about software development ... Debugging is still, as it was 30 years ago, largely a matter of trial and error.*”

Unfortunately, after all the efforts of the software engineering community and evolution of theory and technologies in software engineering, paradigms such as object-oriented and component-based approach, programming languages and tools, debugging has hardly been made easier. On the contrary, the situation only seems to worsen. The increase in size and complexity of software systems, and more diverse hardware/network environments contribute to this tendency. These factors imply a different approach is necessary to tackle this problem.

In this dissertation, we hypothesize that having “knowledge” about recurring defects plays a vital role when a developer is writing and debugging the code. Consider the following observations.

1. Debugging is a human problem. Today's software development is still dependent on the ability of individual developers to construct the software. This will not change until software development is completely automated, no longer involving human ingenuity. Until that happens, it is totally up to the developers how many defects are created, found and fixed.

2. Although everyone makes defects, debugging ability varies among developers. For example, experienced developers tend to know "how to debug" better than novice developers. Such knowledge can be about the characteristics of frequently occurring defects in their project, techniques to prevent and/or diagnose problems, or how to use tools. Obviously, everyone is a novice at first. Therefore, it is vital to improve a novice's ability to debug by providing such knowledge. And everyone is a novice when learning a new language, or working on a new software domain.

3. An ideal for developers might be to use technology (tool and/or language) that can automatically detect defects even if they don't have prior knowledge about these defects. The availability of good tools can indeed affect the efficiency of debugging. However, no tool can be made without first recognizing which types of defects occurs frequently enough that it is worth developing a tool to detect them. In this sense, knowledge of recurring defects is a valuable input to researchers and vendors who need to understand the cost-benefit tradeoffs

in developing tools.

Therefore, we further hypothesize that if developers can somehow acquire knowledge about recurring defects, they can either avoid them or find/fix them more easily. Unfortunately, such knowledge is mostly implicit; it exists within the brain of individual developers, and is rarely shared. Moreover, developers may not always be aware of what they really know themselves. As a result, everyone has to spend time going through a painful learning process to obtain the necessary knowledge by themselves. To change the situation, we need a means of extracting, accumulating and sharing such knowledge.

In this dissertation, we develop a framework for building “knowledge” about recurring defects. There are many obstacles that have to be overcome to realize it. The goal is to provide solutions to these obstacles and show the feasibility of this approach.

1.2 Problem domain: high performance computing

Before describing the problem in more detail, we define the domain of the software we focus on.

One important technology trend in recent years is a shift to parallel architecture. As the performance of single processors is reaching a limit, a wide range of computer platforms, from high-end supercomputers to desktop PCs, are moving towards multi-processor architectures. Programming for parallel architecture involves defects that did not exist in serial programs, and debugging is even more difficult

[19, 24].

Throughout this dissertation, we address the problem of building knowledge of defects specifically in the domain of high-performance computing (HPC). In this domain, we expect that the types of defects that recur in source code are different from defects in other areas of software development because of the following factors that make HPC unique:

- **Platform**: Powerful computation power required of today's HPC systems is achieved by massively parallel systems. Software that is designed to run on such systems is prone to certain kinds of defects that simply do not occur on conventional machines, e.g., concurrency defects and defects related to multiple levels of memory hierarchy. In addition, other defects may be much more difficult to isolate.

- **Performance**: Since emphasis is put on both correctness and performance, an HPC program can contain performance defects even if it would produce correct output. Although few applications are fully optimized to squeeze out every last bit of speed, it is important that the execution completes within the time constraints for their user. Achieving good performance on multiple processors is often difficult.

- **Language**: To leverage the parallel system resources, developers usually use special HPC languages and libraries such as MPI [33], OpenMP [27], UPC [18], Co-Array Fortran [64] and Titanium [75], each with their own ways of handling issues such as communication and synchronization. New models continue to be developed which claim to simplify these tasks and achieve good performance, but they may also bring a new kind of difficulty.

- **Developers**: HPC systems are often developed by scientists and graduate students who have not had formal training in software engineering. They tend to prefer known, stable technologies to minimize learning costs and maximize portability across platforms.

- **(Lack of) tools**: Tool use (integrated development environments (IDEs), graphical debuggers, defect detection tools, profiling tools, etc.) is lower than in other software development domains [20].

- **Portability**: Portability is very important for HPC applications since they must be run on various platforms depending on the computational resources available. New computers are continually developed to replace older machines.

- **Validation**: Given the nature of HPC applications, it is not unusual that both the implementation and the underlying scientific theories are constantly changing during development. In this environment, the correct outputs are not always known, so debugging is particularly challenging and costly.

Many researchers focus on the problem of debugging parallel programs by developing new theories and tools to prevent or detect defects. Again, knowledge of recurring defects is a foundation for such efforts. However, few studies have been conducted to understand what kinds of defects are really problematic in this domain.

1.3 The problem

The problem addressed by this dissertation is building and evolving patterns of defects in HPC based on empirical data, and developing an experience base using the accumulated knowledge to demonstrate its usefulness. See Chapter 3 for more details.

1.4 Proposed solution

This dissertation presents a methodology for iteratively and incrementally building the knowledge of domain-specific defect patterns through an empirical, human-based approach. It uses a reading-based approach to identify recurring defects in the change history data, and the identified defect patterns are stored in the experience base as initial content. The content plays a vital role for extracting further knowledge from domain experts through semi-structured interviews. The accumulated knowledge can be packaged into derivative artifacts. We use the data collected through our involvement in the DARPA High Productivity Computing Systems (HPCS) program. [49]

1.5 Organization of the dissertation

The dissertation is organized as follows. Chapter 2 describes related work. Chapter 3 provides the problem statements. Chapter 4 provides the methodology and problem-solving approach. Chapter 5 describes the approaches for data collection and presents identified patterns and a defect classification. Chapter 6 describes the design and implementation of the experience base, and describes how the content of the experience base can be used to extract the tacit knowledge of experts. Chapter 7 discusses the development of derivative artifacts such as educational materials and recommendations for technology providers. Chapter 8 concludes the dissertation with a summary of contributions, lessons learned, and future avenues of research.

Chapter 2

State of the Art in the Field

This chapter reviews the state of art research on the topics related to this dissertation.

2.1 High performance computing (HPC)

2.1.1 Parallel programming models and languages

Deborah L. Wince Smith, Council of Competitiveness President, recently stated that "HPC has been and will continue to be a key ingredient in America's innovation capacity."[1] For any branch of science or engineering, computational power is really a "third leg" of the stool, along with theory and experimentation. There is always a demand for greater computer power to solve problems in less time with higher accuracy, and vast investments are being made to realize the desired performance.

The meaning of "high performance" has dramatically changed over the years, and so has the technologies supporting it. A recent trend is the massive use of parallelism. All high-end super computers today are leveraging parallel processing capability. (For example, the world's fastest supercomputer today consists of more

[1]http://www.compete.org/hpc/grand_challenge.asp

than a hundred thousand processors. [2])

As systems become more and more complex, software development is getting harder. Unfortunately, since today's compilers are not sophisticated enough to generate a program that runs efficiently on a parallel platform, an HPC application needs to be written explicitly to support parallelism.

Over the years, many programming models and languages have been developed for writing HPC applications. They were created to provide good performance under massively parallel environments.

Developers of HPC applications need to use an appropriate programming model and language, such as the following:

- Message passing: Each process executes within a separate address space and processes communicate by exchanging messages. This model provides explicit control of interactions between processes. MPI [33] is the most popular technology. PVM [71] is another example.

- Shared memory: All processes are executed in a shared address space, so they can communicate through memory access. OpenMP [27] is an example of the technologies supporting shared memory model.

- Global arrays: Each process resides in a separate address space, but a process can also access a restricted set of memory elements in the address space of other processes. Examples of technologies supporting global arrays are UPC [18], Co-array Fortran [64] and Titanium [75].

[2]http://www.top500.org/

- Declared distribution: The programmer provides hints as to how the data should be distributed, and the parallelism is extracted by the compiler or run-time system. High Performance Fortran [14] and MATLAB*P [25] are examples of technologies supporting this model.

Many HPC languages are defined on top of "base" languages such as Fortran, C, and C++. For example, The MPI specification defines a set of data types and library functions for C and Fortran. OpenMP statements are written directives in the base languages. This makes learning these HPC languages easier for those who are already familiar with the base language, and it can also make porting easier.

There are also new HPC languages such as X10 [21] and Chapel [17] currently in active development. While they are designed and implemented in the hope of increasing productivity, generally it takes a long time for a new HPC language to become stable, available on many platforms and get adopted by a mass development community. Typical HPC program may last 20-30 years and thus developers are reluctant to try new languages which may not be actively supported for the lifetime of the application.

There are also hybrid approaches, in which multiple programming models are combined to achieve better performance. For example, the combination of MPI and OpenMP is used to exploit the characteristics of the architecture consisting of distributed clusters, each of which has shared-memory access.

Dongarra et. al. [29] provides the guidelines for choosing whether or not to use particular programming models. Hochstein [38] also provides a good summary of programming models and languages that are commonly used to write HPC applications today.

2.1.2 Software engineering in HPC

Since not all software engineering theories and practices are applicable to HPC, it is important to understand what practices are currently used in real HPC projects, and what practices should be chosen in specific contexts.

Current practices

Hochstein [39] describes the state of software engineering practices and use of technologies observed in five Advanced Simulation and Computing (ASC)-Alliance projects, all of which are currently in active use for scientific research and active development. Some key observations related to this dissertation are summarized as follows:

- **Base language and parallel programming model**: Most of the codes are written as a mixture of C/C++ and Fortran. All codes make use of the MPI library to achieve parallelism. In addition, each code make use of external libraries for features such as I/O, mesh operations, geometry, linear algebra, differential equations, etc. in parallel environment.

- **Code architecture**: While libraries wrap underlying operations for low-level communication from developers, encapsulation is not perfect. The developers are still required to write raw MPI code to achieve the desired functionality.

Some of the codes use a layered approach which hides the details of message-passing, so that a programmer can add additional functionality without writing MPI code. However, these abstraction layers had to be written from scratch.

- **Version control**: The projects use version control systems such as CVS and Subversion to coordinate changes to the code. They are integrated into their development process, and developers are automatically notified by email whenever code is checked into the repository.

- **Testing**: Almost all projects use some form of regression testing to catch defects introduced by modifying the code. Some projects have an automated system for running regression tests, and others run the regression tests manually.

- **(Lack of) formal review**: We have not seen any projects that have adopted a formal process for approving code before it is checked in to the repository. Individual developers are responsible for performing testing before commits are made.

- **Algorithmic defects**: Finding and fixing algorithmic defects is much more challenging than finding and fixing coding defects.

Related empirical studies

The effect of software engineering practices on HPC has been studied. For example, Morton [61] has reported lessons learned in porting Fortran/PVM code to the CrayT3D and described tradeoffs between performance gains and effort for porting. In their experience, coding effort can be reduced by “encapsulating the details of optimization and incorporating them in applications through high-level subroutines.” Berthou et. al. [11] compared OpenMP, HPF, and MPI to evaluate the impact of programming models and language features on the performance of applications and presented the recommended parallelization strategy.

Berlin et. al. [10] conducted evaluated impact of programming language features on the performance of parallel applications on cluster architectures. Hochstein [41] has reported the impact of programming models on development effort. They reported that novice programmers spend significantly less effort with OpenMP compared to MPI.

2.2 Defects

2.2.1 Role of defects in software development

Debugging is one of the most time-consuming activities in software development, taking from 30 to 70 percents of the total development time [69, 48]. Although it varies significantly depending on context and how “debugging” is defined, defects are major bottlenecks to development productivity. Many empirical studies have reported anecdotes on debugging and defects [30, 50, 55, 66, 72]. They emphasize the importance and difficulty of understanding defects.

In earlier research, people have sought generic strategies for avoiding, preventing and resolving defects. For example, Boehm et. al. suggested “10 techniques that can help reduce the flaws in your code" based on empirical research on defect reduction [12]. Researchers have continually put effort to understand, evaluate and predict the characteristics and behaviors of defects. One key observation is that contexts play an important role as most projects and organizations differ [69], so defects must be understood along with various context variables they are associated with.

2.2.2 Defect detection

There are countless techniques and technologies proposed and applied for finding defects. We describe three major approaches.

- Static approach: Static analysis techniques, including dataflow analysis, type checking, model checking and other formal methods, are powerful tools for detecting software defects. The theory behind these techniques is closely related to programming language and compiler semantics.

- Dynamic approach: Defects are found by executing the software. Testing is the most common practice for finding defects in the software.

- Human approach: Reading-based methods [51] are known to be an effective inspection technique. This type of approach depends on the insights of human developers to understand the problem and the implementation.

In practice, debugging is performed by a combination of these approaches.

2.2.3 Defect classifications and measures

Measuring defects has been used as a means of evaluating software development, and predicting success/failure of the project and the product. Defining defect classifications is a key step to conduct data analysis.

Chillarege [23][22] proposed Orthogonal Defect Classification (ODC). It defined 8 defect type attributes (function, interface, checking, assignment, timing/ serialization, build/package/merge, documentation and algorithm) and 6 defect trigger attributes (bug-fix, DB-recovery, exception handling, timing, workload, user code and unknown). Other researchers have defined other classification schemes. Ko [52] summarizes classification schemes defined in various languages, expertise and programming contexts.

These classification schemes have been found useful. For example, they can be used to differentiate the occurrences of defects of different types depending on development phases. One key observation is that the classification schemes are really context-dependent.

2.2.4 Defect patterns

In recent years, practitioners have been aware of the usefulness of establishing ”patterns” for software defects. Defect patterns are different from design patterns [35] and anti-patterns [16][15], which have been widely recognized as a useful form of knowledge about good and bad software designs in software engineering. While a design pattern is intended to describe a solution to frequently occurring problems independent of a particular application domain or a program language, the description of a defect pattern (or a bug pattern, in a more common terminology) is made at a much lower level, as defects are closely coupled with the details of the implementations. For this reason, the applicability of defect patterns tend to be limited to a specific programming language, a property of software, or an application domain. Some patterns only describe the characteristics of the defects, while others go deeper and provide advice for avoiding and/or detecting them.

One of the programming languages for which defect patterns are widely studied is Java. Allen [2] is frequently referred to as one of the earliest works on Java bug patterns. PMD [26] and FindBugs [44] are examples of analysis tools which implement detectors for various Java defect patterns to find them automatically. Rutar [67] provides a comparison of several bug finding tools for Java.

PHP is another language for which defect patterns are actively studied. Houston [43] describes "five beginner mistakes to avoid" in PHP programming. PHP-Sat [13] is a static analysis tool for detecting defects in PHP applications.

Howard [45] presents defect patterns related to security. This is an example of the patterns focused on a particular aspect of software. Kumar [53] presents patterns of computer intrusions.

Michail [60] describes defect patterns in GUI applications and a method to help users avoid them. Sullivan [70] is a comparison of software defects in database management systems and operating systems.

Again, it should be emphasized that the patterns described in each of the works above are very different from each other. This means that defect patterns need to be identified for each context of interest.

Another point is that while these works explain the defect patterns themselves, there is little explanation on how these patterns were identified. In most cases, the patterns seem to derive from the personal experience of the researchers involved in documenting them. A methodology for developing defect patterns in a new domain is missing, as well as a methodology to validate patterns. (One possibility for validation is that after the patterns are implemented as a tool, patterns are evaluated against real code by applying them. Williams [73] used code mining techniques to identify defect patterns.) Therefore, when we build patterns for HPC, the methodology for doing so becomes important. The emphasis is put on making sure the patterns are supported by empirical evidence.

Defect patterns in parallel computing and HPC

Defect patterns in generic parallel computing have been reported [19, 32, 5, 4]. They almost exclusively discuss synchronization issues such as deadlock, race conditions, and performance issues, with a few exceptions of sequential defects such as segmentation fault, bus error, operand range error, and floating point exception [3]. One key observation, however, that we present in the later chapters, is that the defects that are “known” to be common in parallel computing are not necessarily dominant in HPC applications. Therefore, defect patterns developed for generic parallel computing do not necessarily cover the defects that are important in HPC applications.

There are only a few research results on defect patterns specific to HPC. NERSC [63] provides a tutorial for debugging HPC applications, which includes a few HPC-specific defects in addition to the generic defects in numerical computations. The research closest to this dissertation is conducted by Suess [62]. They reported common mistakes found in students' code written in OpenMP and discussed how to avoid them.

2.3 Knowledge building

2.3.1 Experience bases

Knowledge bases (KB) in general are defined in various ways. The Cyc Knowledge Base [54], the world's largest general knowledge base, defines itself as *a formalized representation of a vast quantity of fundamental human knowledge: facts, rules of thumb, and heuristics for reasoning about the objects and events of everyday life.* It is designed to provide a machine-readable form of knowledge so that it can, for example, allow AI researchers to implement machine learning, natural language processing, semantic web, and other fields of study which require a means to perform automatic reasoning. The other type of knowledge base is intended to store a human-readable form of knowledge about a particular topic of interest such as an organization, software support, etc. The typical content includes articles, FAQs (frequently asked questions), hints and tips, and troubleshooting guides.

An experience base is a knowledge base which stores experience packages. An experience base in the domain of software engineering was first introduced with the purpose of organizational learning. Basili [7] discusses its importance in the context of a quality improvement paradigm (QIP). In QIP, while individual projects pursue their goal to develop software, the experience factory collects and maintains knowledge across projects so that they can understand where they can reuse architectures, designs, what functionality each product has, and how to estimate the cost of adding new features or changing existing ones. An experience base functions as a core of the experience factory, which provides an infrastructure to support projects, analyze project data and package experience.

In CeBASE [6], the experience base was organized *to accumulate empirical models in order to provide validated guidelines for selecting techniques and models, recommend areas for research, and support software engineering education.* As is represented by the "No Silver Bullet" principle [34], we cannot expect technology or practice to be effective in all situations. CeBASE is aimed at accumulating empirical knowledge and experience to support decision-making in specific contexts.

While CeBASE has achieved a certain degree of success, a great diversity of generic software makes its goal very challenging, since there are often too many context variables affecting "what works and what doesn't." In this dissertation, we attempt to build an experience base for a specific aspect of software development (defects) in a particular domain (high performance computing). We believe this approach will lead to a more useful base, as the target is more focused yet still large enough to be interesting.

2.3.2 Educational psychology

Educational psychology is defined by the American Psychological Association [59] as "the branch of psychology concerned with studying how people learn from instruction, and with developing educational materials, programs, and techniques that enhance learning. Educational psychologists conduct scientific research both to advance theory–such as explaining how people learn, teach, and differ from one another and to advance practice–such as determining how to improve learning. Although perhaps best known for studying children in school settings, educational psychologists also are concerned with learning and teaching people from infancy through old age, in school and outside of school."

We review several key concepts in educational psychology which are related to this dissertation.

Development of reasoning

While there are many things taught in education, what is most closely related this dissertation is the growth of reasoning: the handling of logical relations. Gordon [36] characterized the process of developing an ability of reasoning with "the capacity to use symbols, concepts and abstractions (and) identify the classification and definition of experience." The process is explained using the following key notions.

- Conception and abstraction: the use of symbols and concepts
- Controlled association: detection of hidden similarities
- Classification and definition: introduce order whether among ideas or among

forms of conduct

Learning problem solving

Henderson [37] has conducted a series of interviews to understand the beliefs of instructors of physics on the three types of learning activities:

- Using feedback while/after working on problems
- Working on problems (practicing)
- Looking/listening to example problem solutions or lectures

The key observation is that instructors believe that while learning using feedback while/after working on problems is widely believed to be effective, they have mixed opinions on the effect of learning by looking/listening to example problem solutions or lectures. “One instructor did not believe that students can learn how to solve problems without actually working on problems. Another instructor thought that students might be able to learn something without working on problems, but that actually working on problems would be more effective. All five instructors who believed that learning can take place by looking/listening described the general student action of looking/listening to example problem solutions.”

Constructivism

Constructivism is a theory which encourages learners to build the structure of knowledge by themselves, or teach new knowledge by making use of the concept that learners already possess. Zan [76] states that "student's active construction should be facilitated and promoted" in the constructivistic approach. Note that constructivism is not referring to pedagogy, which defines concrete methods and strategies of teaching, but how learning should occur.

In this approach, the learning process is explained by two complementary processes of accommodation and assimilation [65]. Accommodation is a process to "fit theory to practice" by fitting the internal representation of some ideas to fit the realities, while assimilation is a process to "fit practice to theory", by interpreting the success and failure of the actions and aligning the internal knowledge representation. Educational materials should be designed to promote the process. Since each learner has different background, the actual construction process is unique.

2.4 Research methodologies

In empirical research, the quality of the data and validity of the research results depend on the experimental methodologies selected for data collection, data analysis and validation. The contexts in which the research is conducted vary in each study, and so does which methodologies are applicable and appropriate.

2.4.1 Quantitative methods

Quantitative methods are most suitable when the object of the research can be modeled with numerical metrics, and a sufficient amount of data can be obtained. For example, experiments in natural science almost exclusively use a quantitative approach. On the other hand, in research areas involving human behavioral aspects, the phenomenon to be measured tends to be harder to reproduce, as not all parameters can be controlled. Campbell [28] describes possible experimental designs and the associated threats to validity.

The quantitative data is usually analyzed using a statistical method. It is widely accepted as a convenient means to make objective judgment when interpreting research outputs. However, caution must be taken since statistical methods are too often misused and misunderstood [46].

An example of a quantitative approach applied to software engineering can be found in the study of defect data for evaluating development processes or products. In a traditional development process, in the testing phase the testers fill out defect reports (known as change requests) when the test cases fail. If the development process is under control, the number of defects or defect rate diminishes as the testing phase progresses. Project managers can then decide when to stop testing, or they can estimate how much more has to be spent for further debugging and testing. If the number of defects does not follow this diminishing pattern, it indicates something is going wrong with the software process. To rephrase, this type of research uses the occurrences of defects as a metric for studying the software process. While simple approaches just counts the number of defects, a more sophisticated type of study counts the defects by type.

In software engineering, many technical aspects of software development can be modeled with numeric measures and analyzed with a quantitative approach.

2.4.2 Qualitative methods

While quantitative methods based on statistical significance have become the standard way of processing research results in natural science, limitations exist when they are applied in other research areas such as in handling the complexity of issues involving human behavior.

On one hand, getting data suitable for quantitative analysis is hard when human behavior aspects are of interest. On the other hand, there are often cases where researchers are interested in more than just statistical significance. Therefore, qualitative research methods are used to compensate for the limitations of the quantitative approach.

In a qualitative approach, the focus is put on obtaining a deeper understanding of the topic. Seaman [68] stated that "the principal advantage of using qualitative methods is that they force the researcher to delve into the complexity of the problem rather than abstract it away. The results are richer and more informative." On the other hand, "qualitative analysis is generally more labor-intensive and exhausting than quantitative analysis. Qualitative results often are considered 'softer' or 'fuzzier' than quantitative results." To make qualitative analysis feasible and as efficient as possible, various methods for data collection, analysis and validation have been developed. We summarize those related to this dissertation.

Structured interviews

Structured interviewing is a method which collects information using a list of pre-written questions. In its simplest form, all interviewees are asked the same questions in a standardized order. Litkowski [56] describes that "when essentially the same information must be obtained from numerous people for a multiple case-study evaluation or a single case-study evaluation, it may be beneficial to use structured interviews."

The advantages of structured interviews are an ability to replicate interviews easily. The method is usually quite reliable. It should be noted, however, that the quality and usefulness of the information heavily depends on the questions and how they are asked. A substantial amount of pre-planning is required to make the interview effective [57]. The questions and other materials presented to the interviewees need to be carefully composed, as they determine the framework of the information being collected in the interview. A question format can also affect the characteristics of the information that can be obtained.

Coding

Coding is a method of qualitative data analysis for extracting quantitative variables from qualitative data [31]. It is often used to transform the primary data represented as words or pictures to numerical or categorized data. The cost and reliability of the transformation depends on the characteristics of the original qualitative data.

Maxwell [58] states that “the goal of coding is not to count things, but to ‘fracture’ the data and rearrange them into categories that facilitate comparison between things in the same category and that aid in the development of theoretical concepts. Another form of categorizing analysis involves organizing the data into broader themes and issues.”

Expert validation

Expert validation is a method of validating the results of qualitative analysis by showing there is an agreement between the researcher’s analysis and the description given by an expert. This method assumes that that the analysis results are comprehensible to the expert. When conducting expert validation, presenting the results in an organized way is important. Expecting experts to read through the unstructured analysis results is asking too much, and may affect their judgment.

Chapter 3

Problem Statement

This chapter defines the problem we will address in this dissertation using the Goal, Question and Metrics (GQM) template [8], and presents a set of research questions associated with the goals.

3.1 Problem Statement

The high-level goal of this dissertation is "to construct a methodology and technology to collect, improve and apply knowledge about domain-specific defect patterns using an empirical approach." Our research is specifically targeted on software development for the high performance computing domain. Through a series of empirical studies, we will identify patterns of recurring defects in HPC applications, develop classification schemes that cover them, and continually refine the patterns. The major challenge is to choose an appropriate approach, conduct the study with available testbeds, and formalize results so that we can iteratively evolve knowledge.

The actual process to execute this research involves the construction of an experience base. Not only does the experience base system store all the results, it should also provide a means for stakeholders in the community to access the content and provide feedback.

3.1.1 GQM goal

Analyze various empirical data on application development in order to characterize it with respect to patterns of recurring defects from the point of view of researchers in the context of HPC.

3.2 Hypotheses and research questions

Pursuing the above goal leads to an attempt to answer the following research questions.

3.2.1 Hypothesis about the existence of defect patterns

H1: There are recognizable classes of defects that frequently appear in the applications for high performance computing.

The first set of questions addresses whether there exist distinctive defect patterns in the HPC domain. These are the most fundamental questions for this dissertation.

- *RQ1-1: What are domain specific defects in HPC?*
- *RQ1-2: Can we identify defect patterns (causes, symptoms, potential cures and preventions, and examples)?*
- *RQ1-3: Can we define a set of templates to describe information about defect types at different levels of abstraction?*

3.2.2 Hypothesis about the collection and description of knowledge

H2: Common defects can be identified by reading-based source code analysis

The next set of questions is related to the feasibility of data collection and accumulation, including identification of defect patterns and development of an experience base.

- *RQ2-1: Can we build heuristics to detect defects in the code that can be used in a reliable way by others?*
- *RQ2-2: Can we classify defects in a way that is clear to experts and allows them to add information?*
- *RQ2-3: Is it possible to automate the detection of some classes of HPC defects?*

3.2.3 Hypothesis about the refinement of knowledge

H3: We can build and evolve the patterns based upon expert knowledge

The following set of questions is related to the refinement of knowledge.

- *RQ3-1: Can we extract new knowledge by presenting existing knowledge in a structured way?*
- *RQ3-2: Can we build and evolve a defect experience base with usable information?*

3.2.4 Hypothesis about the application of knowledge

H4: Knowledge accumulated in the experience base can be packaged into a useful form

- *RQ4-1: Can teaching novices about defect patterns reduce the number of defects made?*
- *RQ4-2: Can we provide recommendations to researchers of future defect detection tools? Can we develop our own data analysis tool based on the recommendations?*

3.3 Constraints

3.3.1 Difficulties in finding defects for developers

A precondition for obtaining defect reports from developers is that developers recognize defects themselves, since they can never report defects of which they are unaware. There are several factors that make defects harder to identify from developers' point of view.

- Since HPC code is often required to work in various hardware and software environments, portability is important. If a defect does not cause a failure on the specific environment the developer is working, the existence of the problem may not be detected at all by testing.
- If a failure is detected under specific runtime conditions such as input data, parameters and the number of processes/threads, extensive testing is necessary

to detect the existence of a defect. A similar situation is that the behavior of the program is non-deterministic due to concurrency, in which case finding a defect may need many executions.

- Testing as a means to validate the code is a means of knowing whether something is wrong. In a simple situation, correctness of the program is often validated by checking the output against known correct values. When the correct answer is not known, validating correctness becomes harder. In some application areas such as numerical analysis and random simulation, the output may contain numerical errors that are hard to verify as reasonable.

- As stated in Chapter 1, poor performance of HPC code is considered a defect even if the output is correct. In practice, few programs are optimized to "squeeze out the last drop of performance" from a particular architecture. It does not matter as long as the code runs fast enough to provide useful output within time and resource constraints. Even if the program is unacceptably slow, some problems are just inherently hard to parallelize. Therefore, it is a difficult judgment whether the code has a performance problem which can be fixed with a known reasonable method.

3.3.2 Difficulties in recording defects for developers

Even if developers did recognize defects, recording them can involve additional difficulties.

- If they can make a record of defects while they are debugging, they have an opportunity to create an accurate log. However, as debugging is a human-intensive task, developers need to focus on the main job of investigating what is wrong with the code. Doing a “side job” may interfere with the debugging activities.

- If recording of defects are retrospective, developers may not remember all defects they encountered. Whether a defect is recorded depends on how strongly developers remember it. This is not necessarily bad, since the defects which required the most effort to track down, are the ones in which we are the most interested in, and are more likely to remain in the memory of the developers. Nevertheless, this factor can damage the completeness of defect reports.

- Developers are not necessarily willing to record all defects. For example, they may not want to report defects if they seem so simple that they feel embarrassed to let others know they made such an error. This tendency largely varies depending on the mind set of individual developers. External factors can also affect this. In an extreme example, if developers are evaluated based on the number of defects they made, it would be natural for them to not report all of them.

3.3.3 Difficulties in collecting data on defects for researchers

In traditional software engineering research, the primary data source for software defects is change requests, which are maintained following the development practice each project follows. Unfortunately, this kind of data is difficult to obtain in many software engineering environments, and this approach is not applicable. In most scientific computing projects, no formal change requests are created and maintained.

Figure 3.1 summarizes potential data sources of defects organized into several categories.

- Raw data: Source code contains the “raw” defect data. All other data sources are indirect and abstractions.[1]

- Unstructured data: Many data sources, such as mailing list archives, contain information on defects. It is embedded in other unrelated information, and it doesn’t have a common format.

- Structured data: In a well-organized project, defects are recorded in a defect tracking system. Information has a predefined structure.

- Defect patterns: If some defect patterns are already known in the domain of interest, they can be provide a useful clue for determining more patterns. At this level, the data contains abstracted information on the defects, such as symptoms and possible techniques for resolving or preventing them.

[1]Raw defect data can also exist in a design document. It is out of scope of this dissertation.

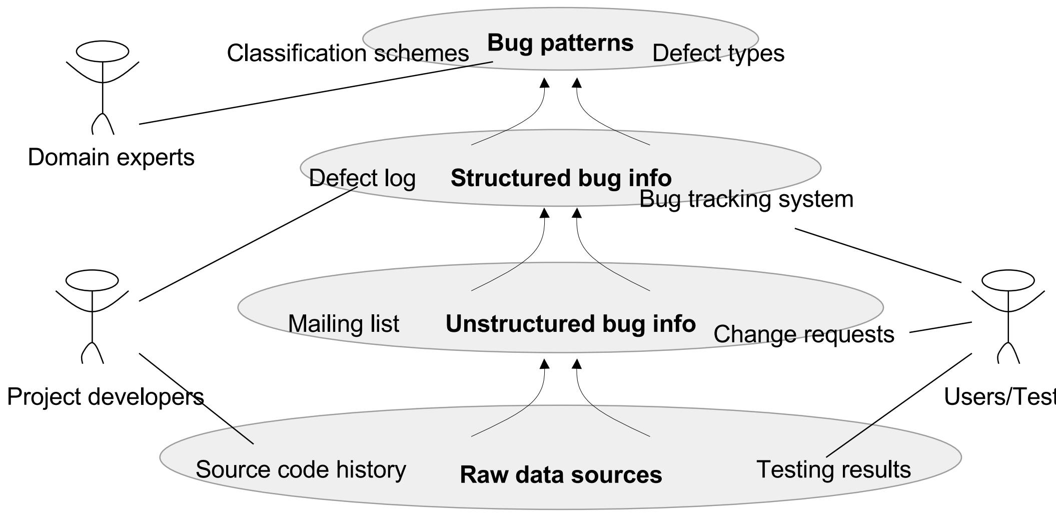


Figure 3.1: Possible data sources for analysts

In general, the more abstract and structured the available data sources are, the easier the data collection tends to be. Unfortunately, many of these data sources are not readily available in the HPC domain. For example, few HPC projects actively use a defect reporting and tracking system. Defect tracking is mainly accomplished through informal communiation among project members [39], thus the consistent information cannot be obtained. The mailing list archives are often available, but since emails contain a lot of information irrelevant to the defects, extracting defect data requires significant effort. In this dissertation, we assume the following data is available.

- Source code with change history. Many HPC projects adopt a code management system. Some of them are publicly accessible, while others are for internal use which require permissions to access. In this dissertation, we attempt to capture the change history data at a finer granularity.

- Defect patterns. This type of data source exists as knowledge hidden in the brain of experts. In this dissertation, we attempt to extract such knowledge.

3.3.4 Difficulty in finding defects for researchers

The judgment of whether or not a certain behavior is a defect requires the understanding of what the code is supposed to do. The goal of a scientific project is often as simple as "perform computation X". However, X can involve an advanced scientific theory as background. Furthermore, X keeps changing as the project progresses. Therefore, researchers need to overcome the learning cost for each project they analyze.

Chapter 4

Methodology and Problem Solving Process

While many practitioners recognize the usefulness of defect patterns, and there are patterns proposed for various languages, properties of software, and application domains, there seems to be no established methodology for developing defect patterns in new contexts. A methodology for identifying and evolving defect patterns is important because defect patterns tend to be context-dependent and, therefore, need to be built for every context of interest. In this chapter, we propose a general methodology for building, refining and applying defect patterns for a new domain from empirical data, and demonstrate its realization in the high performance computing domain. The core of our methodology is based on the construction of an experience base. Knowledge of recurring defects is incrementally and iteratively accumulated through three separate, but inter-related activities. We describe the methodologies for each part of the problem-solving process with the underlying rationale based on the HPC constraints.

4.1 Problem solving approach

Our solution is to build an experience base which can store and share patterns of recurring defects, so that HPC practitioners can access them at various levels of abstraction and submit feedback. To realize this under the various constraints described in the previous chapter, we take an approach that consists of three different, yet closely coupled sub-processes which are illustrated in Figure 4.1.

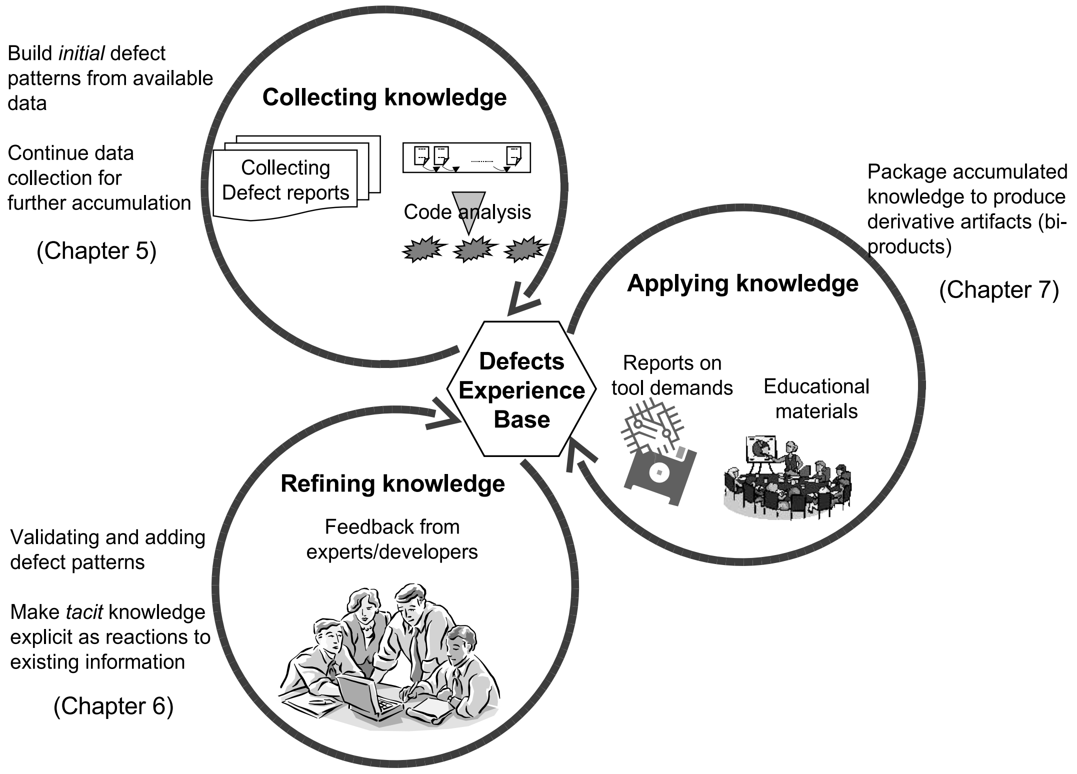

Build *initial* defect patterns from available data
Continue data collection for further accumulation
(Chapter 5)
Collecting knowledge
Collecting Defect reports
Code analysis
Package accumulated knowledge to produce derivative artifacts (bi-products)
(Chapter 7)
Applying knowledge
Defects Experience Base
Reports on tool demands
Educational materials
Refining knowledge
Feedback from experts/developers
Validating and adding defect patterns
Make *tacit* knowledge explicit as reactions to existing information
(Chapter 6)

Figure 4.1: Problem solving approach

- Knowledge collection: pattern identification for bootstrapping and continual data accumulation
- Knowledge refinement: reactive pattern refinement based on expert feedback
- Knowledge packaging: development of derivative artifacts and validation of usefulness

By conducting them iteratively and in parallel, the content of the experience base is gradually evolved. In the rest of this chapter, we describe each sub-process in more detail.

4.2 Opportunistic and iterative research

Since resources to run software engineering studies are generally scarce, maximum effort should be made to extract as much information as possible from the available opportunities.In this research, we collect data through a series of classroom studies as well as interactions with HPC domain experts. We need to actively motivate potential study participants to provide us with data by explaining the background of the research project and emphasizing the importance of their help. At the same time, research methods should be designed to minimize the overhead of participating in the study. Each study opportunity may become available only at specific time, so we should be always prepared to work with new participants while conducting the study and analyzing the results from existing data sources.

An iterative approach implies the data collection and analysis are conducted in parallel and each phase is repeated over time. One important observation is that the analysis can become increasingly efficient as more knowledge is accumulated. For example, it is possible that a defect that was missed in previous analysis is detected later using the knowledge established from other data. Suppose three data sets A, B and C were analyzed in this order, and a new defect X was identified when analyzing the data set C. If the defect X was also in the data set A or B but missed in the first iteration, it can be detected in the second iteration since the analysis was performed more carefully to identify the defect X.

Therefore, the process of knowledge building cannot be driven by a single activity. Instead, it forms a feedback loop in which knowledge is incrementally built and evolved as both new and existing data are repeatedly visited. The purpose of the analysis can also change over iterations. Early iterations are generative and are focused on identifying individual defects to create initial pattern definitions, while later iterations are confirmatory and focused on assessing whether previously identified defect patterns are observed in other data too.

4.3 Process 1: collecting knowledge

4.3.1 Purpose

Since the experience base is merely an empty container in the beginning, we need to prepare an initial content that is interesting enough to convince others to access the experience base. Therefore, the first sub-process is to capture data and build knowledge of recurring defects. As it forms a basis for the entire body of knowledge to be accumulated, the quality of the analysis in the first sub-process determines whether the experience base can be widely accepted. For this reason, it is worthwhile to perform as much analysis as possible.

Furthermore, since we have control over the data we create, it provides an opportunity to obtain more detailed, consistent and interpretable results than the data coming from others. Therefore, this sub-process is also important for continual data accumulation and improvement even after the initial content has been successfully built. As more data comes in, the knowledge and the analysis method are gradually evolved.

It should be emphasized, however, that the goal of this sub-process is not necessarily to gain "perfect" knowledge. It should rather be considered as a driving force to boost the second sub-process, which provides an opportunity to expand the body of knowledge to broader contexts than are possible in process one.

4.3.2 Research methods and rationale

Reading-based code analysis

The primary method of the data analysis is code inspection. We inspect the change history data to identify defects that exist in each version of the source code. To mitigate the labor-intensiveness of this approach, we develop a set of heuristics to help locate defects.

A rationale is that the source code history is the most commonly available data type. Other data types (e.g. defect tracking entries) are harder to obtain in the HPC domain, and tend to be inconsistent. Therefore, methods that require other data sources are generally not applicable.

Qualitative data collection

As described in Chapter 3, the primary data available in the HPC domain is raw data (i.e., source code versions). Since the initial analysis begins with the state in which the characteristics of recurring defect types are unknown, it needs to be exploratory and focused on "discovering" a set of defects that appear to occur frequently. What is most important at this phase is to record as much context information as possible. Therefore, the primary analysis results are recorded as text.

Inductive pattern definition

We take an inductive, bottom-up approach to define "defect types". A defect type represents a set of similar defects grouped together. Each defect type is derived by grouping similar defect examples observed in the data analyzed. The benefit of this approach is that the defect types are automatically given support from concrete defect examples, so the knowledge associated with them has rich context information.

To define defect types, a **coding technique** is used to identify common characteristics of defects and group similar defects together. The coding is done using a set of heuristics. We also introduce a set of templates to describe the information so that it can be recorded in a consistent manner.

4.3.3 Characteristics

Input: The primary data available come from a series of empirical studies conducted at various universities and academic institutions across the United States. As described in the previous chapter, the types of available data sources in the HPC domain are generally limited. The data we use will be described in more details in Chapter 5.

Output: Delivered out of this sub-process is the information on specific HPC defects identified in the data analyzed as well as the description of defect types. These outputs of data analysis are expanded to the description of individual defects that were reported by subjects or found in code analysis, identified patterns, and classification schemes to cover them. The results are a classification scheme, a set of defect patterns, and an experience base.

Actors: To realize this sub-process, there should be two kinds of actors involved. The first type of actor is the HPC developers (study subjects) who grant access to their raw data (source code history) or provide a record of defects. They can have varied levels of experience. Data coming from various kinds of developers provides a variety of contexts in which defects are made. Therefore, as long as they are willing to provide data, they can play this role whether they are novice programmers or they are experienced professionals. The quality of data often depends on how motivated they are to cooperate, especially if providing the data in a useful form requires extra effort from the subject. The second type of actor is researchers (analysts) who collects and analyzes the data based upon a set of procedures or heuristics. In addition to coordinating the data collection process with developers, the researchers need to have the skill to process the data appropriately. For example, they need an ability to analyze the code if the data source is source files and use a coding technique to identify common characteristics. To check that the heuristics embedded in the coding technique are transferable and provide consistent results, we need to perform a reliability study. In this dissertation, the author and other members of the HPCS project play the role of the researcher.

4.4 Process 2: validating and refining knowledge

4.4.1 Purpose

The purpose of the second sub-process is to evaluate and validate the patterns developed in the first process, and add more knowledge to refine the patterns. This sub-process is crucial, because:

- The analysis is reading-based, which does not guarantee correctness by itself.
- The results from the analysis does not cover the entire domain space of interest as it is linked to available sources. To obtain knowledge about defects beyond the available data scope, we need to refine what exists with the help of experts representing other experiences, types of expertise, etc..

4.4.2 Research methods and rationale

Expert validation

A precondition for this sub-process is that the HPC community already possesses a great deal of implicit knowledge about recurring defects at the level of individual programmers. Part of the ability of experienced developers is related to their understanding of the problems that occur during development, as well as of the strengths and the weaknesses of the languages, what techniques and precautions can help them prevent or resolve a certain type of defects effectively, etc. We make use of their knowledge to validate the analysis results obtained in the previous process. Getting feedback from multiple experts allows us to confirm whether they make similar comments to the existing content and view the defect information as relevant to them.

Reactive pattern refinement

Unfortunately, however, the knowledge of experts usually stays within the minds of individuals and is rarely made explicit. It is not unusual that even the developers are not fully aware of what they know [9]. Therefore, without any clues it is hard for them to provide such knowledge even if they are willing to do so. We addresses this difficulty by making use of the knowledge developed in the first subprocess. In particular, we use a variation of **semi-structured interviews**, in which we obtain a reaction to the existing content in the experience base. The content to be presented is organized carefully beforehand, so that it can "contextualize" the questions being asked. By stimulating them with a carefully prepared summary of the existing content, we attempt to obtain feedback in the form of support or refutation of the existing content (whether or not they agree that the current patterns are valid), a possible addition or modification to the content (what information is missing), and suggestions on new defects types that are not covered by the existing patterns. The obtained feedback is reflected in the pattern definitions.

Note that the knowledge obtained from experts contains the information that cannot be obtained with the initial bottom-up approach, such as advice on how to detect, resolve or prevent each defect type. This kind of knowledge is critically important to make the defect patterns useful.

4.4.3 Characteristics

Input: The input to this sub-process is the content of the experience base. In our approach, the content is reorganized specifically to ask for feedback, and used in the form of semi-structured interviews.

Output: Output is additions, deletions or modifications to the current content in the forms of comments and suggestions that are either supportive or critical, or concrete examples of additional defects as code fragments (snippets). The obtained feedback is reflected in the content of the experience base.

Actors: To realize this sub-process, there should be experts who are willing to review the content and provide feedback. They provide the **expert evaluation** of the results to date. There should also be an interviewer who finds appropriate experts, prepares the necessary review artifacts, conducts interviews, and reviews the results. In this dissertation, we play a role of the interviewer.

4.5 Process 3: packaging knowledge

4.5.1 Purpose

Knowledge is only useful when it's actually used. Once sufficient amount of knowledge has been accumulated, the third sub-process is to package it to produce useful derivative artifacts. We can consider these artifacts as bi-products of the knowledge-building process. The specific artifacts derived from the experience base in the HPC domain are:

- Educational material that can be used for teaching common defect patterns to novice HPC developers.
- A recommendation document which describes potential technological needs for detecting and preventing defects.

The purpose of packaging knowledge into these artifacts is to extend the use of the experience base by making access to the knowledge easier. We also evaluate the potential usefulness of the accumulated knowledge by testing whether the derivative artifacts can actually contribute to decreasing defects.

4.5.2 Research methods and rationale

Controlled experiment

To evaluate whether the knowledge obtained through this research is useful, we conduct a semi-**controlled experiment**. We use the educational material in actual graduate-level HPC courses and test whether the students who were taught defect patterns make fewer defects in a subsequent programming assignment. As there is a limit on what we can control in a classroom environment, ideal experiment designs are difficult to implement. In particular, since we are not allowed to treat students in the same class unfairly, randomly splitting the students into two groups and give the lecture to only one of them is not possible. We make use of the data from multiple HPC courses using an identical programming assignment so that the evaluation is still possible while minimizing threats to validity within the given constraints.

4.5.3 Characteristics

Input: Input to this activity is the knowledge accumulated in the experience base.

Output: The deliverables are various kinds of secondary products packaged into a more usable form for specific stakeholders. Specific products to be built include a report on the requirements of future tool development for HPC defects, and educational materials to teach novice HPC developers about defect patterns.

Actors: While anyone who accesses the experience base to make use of the content can be a potential actor of this sub-process, few of them make concrete products. In this dissertation, we assume we play the role of the creator of these bi-products. We also conduct an experiment to test them to evaluate the usefulness of the accumulated knowledge.

Chapter 5

Collecting Defect Patterns

The first activity of the problem solving approach is to collect data for identifying patterns of defects as a means for “bootstrapping” the experience base, since the base would never be used without some initial data. As we argued in Chapter 3, while software defects exist everywhere, collecting them in a usable data format requires us to overcome several challenges. We used two forms of data collection strategies to identify defects, which were doable within the data collection opportunities we had. The first approach is based on code reading, which has been found to be effective for identifying defects from source files written by students. Moreover, since we have control over what we collect, continual data collection provides stable support for the experience base even after the initial content has been built. The second approach is based on self reporting from developers. Our study indicates that collecting defect information from developers’ self reports is also useful, although the recorded defects are less complete.

The task of identifying initial defect patterns requires exploratory analysis, which manual analysis is best-suited for. Other less labor-intensive methods are difficult to implement because of the uncertainty of the characteristics of the defects to look for, the characteristics of the data available and general lack of applicable technologies.

We present a set of heuristics for efficiently reviewing a history of code changes, and demonstrate the feasibility of the reading-based approach by presenting the results of the analysis conducted by the author using the data from a series of classroom studies. We also present the results of the reliability study to show the heuristics of the method can be taught to other analysts and the analysis based on code reading is repeatable with a proper tool support.

5.1 Data Sources

The primary data we use comes from a series of software engineering studies conducted in universities and academic institutions across the United States. These studies were conducted by the Experiment Software Engineering Group at University of Maryland, as a leader of the Development Time Working Group in the DARPA High Productivity Computing Systems (HPCS) project. To explore various issues related to HPC productivity, the group has collected various kinds of quantitative and qualitative data. Note that the data collection activities started in 2003, and the author has only contributed to the efforts made since fall 2005. Therefore, old raw data was provided by other member of the project. However, all the analysis related to defects was conducted by the author.

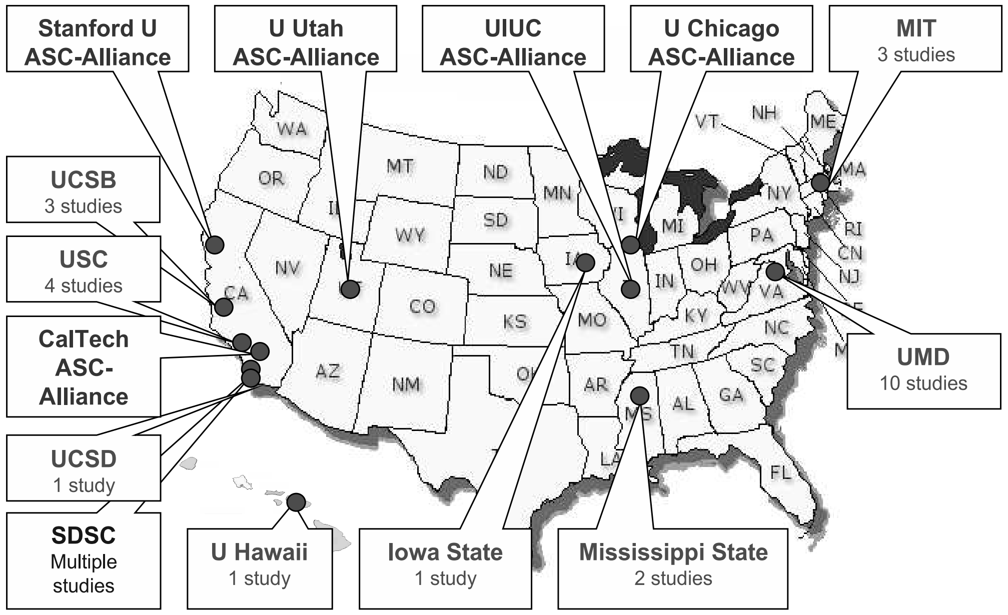


Figure 5.1: Locations in which the study was conducted

Figure 5.1 shows the locations in which we did the study. The characteristics of the data are described below.

5.1.1 Subjects in classroom studies

The subjects of these studies are graduate students taking a course on the topics closely related to high performance computing. Some focus on the HPC systems, while others put an emphasis on parallel algorithms, but all courses teach several programming models/languages for HPC and provide programming assignments which use them. The students have varied backgrounds with experience in generic programming and knowledge of related scientific fields, etc. What is common among the students is that they are new to these programming models (this is the reason they took these courses). During the course, they are required to learn a new programming model over a short period of time, and apply it to solve the assignments. To receive a good grade, they are usually motivated to make effort to produce a good solution.

In real HPC projects, developers are often scientists and students whose primary interest and expertise is not programming. When they first start working as developers in an HPC project, or they adopt a programming model with which they don't have prior experience, they need to learn and use it in their project in short time. In this sense, while the level of expertise in the scientific area is very different, their situation is expected to be similar to the students participating in the classroom studies. Therefore, in the context of defect research, studying the behavior of students is considered as a relevant approximation of many real developers.

5.1.2 Problems used in classroom studies

Table 5.1 summarizes the problems used as assignments. The number in each cell in the table represents how many courses were used the corresponding problem and the programming model as an assignment. Appendix A provides the detailed description for these problems. The size of the problems is smaller than any real HPC projects, and the typical duration of an assignment is one to three weeks. They are intended to help students learn and understand specific aspects of HPC programming. They are often simplified versions of the larger problem which appears in real projects. Therefore, defects that appear in the code solving the problems are expected to be a subset of defects in real HPC projects.

Table 5.1: Summary of the problems and programming models used as assignments in classroom study

	MPI	OpenMP	UPC/CAF	Matlab*P	XMT-C
Embarrassingly parallel					
Buffon-Laplace needle problem	2	2		2	
Dense matrix-vector multiply	1	1			
Nearest neighbor					
Game of life	3	1	1	1	
Sharks and fishes	2	2	1		
Grid of resistors	1	1		1	
Laplace's equation	1			1	
Quantum dynamics	1	1		1	
All-to-all					
Sparse matrix-vector multiply	1				1
Sparse conjugate gradient	2	2	1	1	
Matrix power via prefix	1	1			
Other					
Sorting	2	1			
(Shared memory)					
LU decomposition		1			
Shallow water model		1			
Randomized selection					2
Breadth-first search					1

In each assignment, the students are given a written description of the problem, and asked to implement a parallel solution to that problem. The detailed settings vary across assignments. The students are either asked to write the entire code from scratch, write a parallel version based on the sequential code which was provided by the professor, or write a particular function which can be linked with the given “skeleton” code. For some problems, professors give additional hints by explaining a parallel algorithm to be used. The criteria for grading also vary.

In each study, we collect data from 5 to 20 students who have registered for that course and agreed to participate in the study. By collecting the data from students solving the same assignment, we can obtain a variety of solutions to each problem. This is useful for identifying defect patterns, as we can directly compare implementations to figure out commonality and variability in the defects they make.

We have repeated the study with multiple professors. In each year, the professor teaching the course uses the same or similar problem sets, with potential updates on the teaching methods.

Finally, some problem sets were used as a programming assignment by different professors. There are differences in the details of the problem definitions, so it provides further variations in the data.

5.1.3 Data collected in classroom studies

We have collected the following data.

- **Source code snapshots**: While writing a solution to an assignment, students are expected to make, find and fix defects in their code. Analyzing a series of "intermediate" code versions allows us to identify defects which do not show up in the final submission. Our group developed a small instrumentation tool to capture code snapshots every time the code is compiled. The tool works as a wrapper to a real compiler and saves all source files specified as command arguments. This process is performed in background, so once the tool is set up the data collection is completely transparent from the students' point of view. We installed the tool on the HPC machines which students were given access to, and asked them to perform all the development tasks there so that we can obtain the data for all compiles. Of course, we cannot force them to do so, and we do not get data if the code is developed in other places. For example, it is possible that a student writes a sequential version on their own PC first. Fortunately, however, in order to finish an assignment most students have to compile and run the parallel code on an instrumented machine anyway, because most of them do not have access to other HPC environment, nor are they willing to set up their own parallel execution environment.

- **Other quantitative data**: Using the same instrumentation tool, we collect other data on development activities such as shell command history and interactions with a job queue. We also use Hackystat [47] to record activities with text editors such as vi and emacs. These data are useful for studying other aspects of HPC development. We only use the timestamp values from these data to estimate the effort spent between each compile. Again, once the tool is set up these data can be obtained automatically and transparently.

- **Self-reported data (qualitative data)**: There are other kinds of data that cannot be automatically collected with the above approach. Such data include the experience of students, or what activities the students performed outside of the machine. To collect such data, our group developed a web-based system to allow students to enter information by themselves. For example, we ask students to fill in background questionnaire at the beginning of the semester, and post-study questionnaire at the end. We also ask them to keep logs of development activities. For this research we use self-reported manual defect logs.

All collected data is post-processed, cleaned up and put into a database. Keeping the data in a database is not only convenient for data maintenance but also useful for conducting analysis and storing analysis results. For example, with a set of SQL queries we can easily retrieve all source code snapshots that belong to a particular student for a particular assignment. Each version is assigned a unique ID, which can be used to describe which version contains a particular defect identified.

5.2 Method/technique for reading-based code analysis

The primary data collection approach we employ is to analyze the source files. Since source code is where defects actually exist, code analysis can be considered as the most direct way to collect defects. We take a series of versions of the code, inspect them, and record identified defects for each of them. Information on when each defect was inserted to/removed from the source code indicates how long the defect stayed in the code, which implies the difficulty of debugging it.

While there are other known approaches for defect analysis, they are harder to apply when we try to identify initial defect patterns. Below we compare a reading-based approach with other approaches.

- **Reading-based analysis (inspection)**: The approach which we take is software inspection, or more specifically, code review. The method involves manual code reading by a human analyst. Unlike an ordinary code review, which is conducted in the review phase of the development process, we mainly conduct the analysis offline after all the development is complete. More importantly, we are not only interested defects in the latest version but also those eliminated during development. An implication of the reading-based human analysis is that it is labor intensive. For this approach to be feasible, scalability is an important factor. Both the size/complexity of the code and the analyst's understanding of the code can affect the efficiency of the analysis.

- **Tool-based analysis**: Code analysis can be faster and easier with the use of appropriate tools. As we describe later, we developed a tool for assisting a reading-based method. However, it would be even more desirable if we can just apply some existing defect detection tools which directly detect defects in the given source code without human intervention. Unfortunately, we do not necessarily know in advance what kinds of defects to look for in the code when starting analysis to build an initial set of defect patterns. Since we need to explore unknown classes of defects, we cannot expect that just applying existing tools can uncover all defects of interest (remember there are huge differences in the types of defects in different contexts.) Tools for automatic defect detection can be used effectively once the patterns of defects begin to be formed. If there is an existing tool which can detect a certain defect type, we can just apply it to the data we have. If there is no such a tool, as we describe in Chapter 7, we can provide other technology providers with the information on potential tool demands for these defects.

- **Testing**: Testing is a common practice to find defects in software. To test each version of the source files, the code needs to be compiled, executed with prepared test cases, and verified if the output matches the expected result. Since this process involves program executions, testing is classified as a "dynamic analysis" approach. An inherent limitation of this approach is that it can only capture failures that surfaced during the conducted test runs. Since many "difficult" HPC defects surface under specific conditions, they are often difficult to find with a small number of test cases. Furthermore, testing becomes even more difficult when investigating intermediate versions. Some versions may not even be compiled. Even if they do compile, it is often the case that they only have partial functionality implemented. For example, there may be a version which only implements the initialization process, runs for only one iteration step, and/or computes something completely different to produce debugging output. What is a "correct" behavior is different from version to version, and it is not easy to automatically figure out the behavior to be expected for each version, let alone preparing test cases to verify the output. In our approach, we do not use testing as a primary method of defect detection. We only compile and run the code to confirm that some code fragment contains a defect.

5.2.1 Methodology for analyzing source code

Now we describe a methodology for analyzing source code data to identify defects. The purpose is to allow an analyst to start from the state where the knowledge of defects is not available for the domain of interest, and identify defects to build patterns iteratively and incrementally. Iteration occurs at two levels. At an individual level, each analyst gradually increases the understanding of defects and use it to re-inspect the data to identify defects that were missed in previous trials. At a higher level, he/she also uses feedback from the knowledge refinement process, which is described in the next Chapter, to re-inspect the data. In an early iteration, it is natural that an analyst misses some defects. As the patterns become mature and new defect types are recognized, an analyst can recognize more defects. Note that the goal of the activity described in this Chapter alone is not to achieve "perfect" defect detection. It is usually not possible to check whether all defects in the given source code have been identified. The patterns are therefore evolutionary.

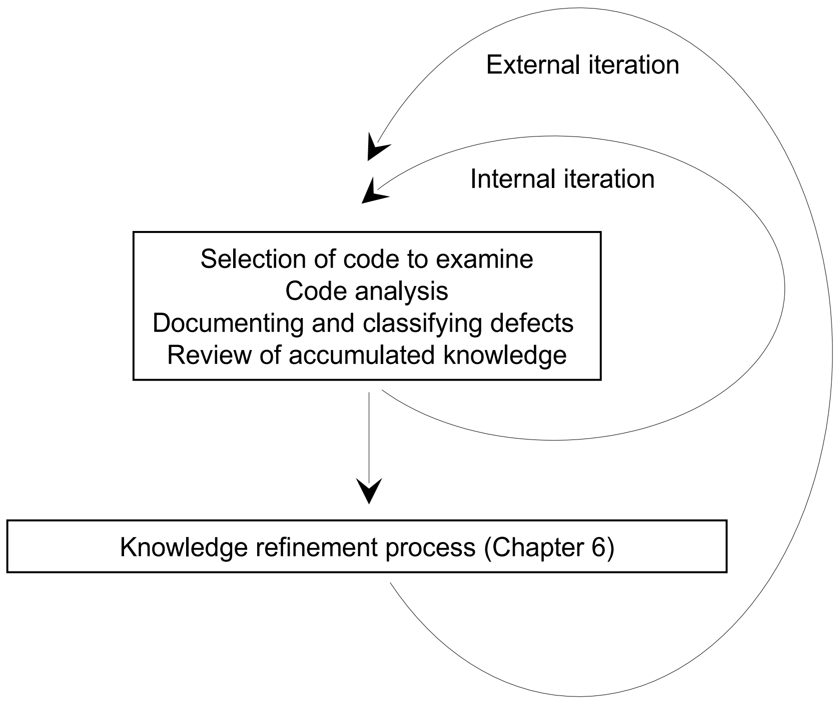

External iteration
Internal iteration
Selection of code to examine
Code analysis
Documenting and classifying defects
Review of accumulated knowledge
Knowledge refinement process (Chapter 6)

Figure 5.2: Methodology for reading-based code analysis

The iteration consists of several key steps.

- Selecting the code to examine
- Code analysis
- Documenting and classifying defects
- Reviewing the accumulated knowledge

Selecting the code to examine

- There is some flexibility with which code to examine first. The efficiency depends on the order.

- A general approach to be recommended is to begin with code for the problem an analyst is already familiar with. Examining many solutions to the same problem provides an analyst with an insight on a typical algorithm and program structure used for that problem.

- If there is no such prior knowledge, it is recommended to begin with code that is easy to understand. The code for smaller problems tends to be easier to understand, although there are always exceptions.

- If the chosen code turns out to be difficult, it should be marked as delayed and other code should be tried. The code analysis is iterative, so it is possible that by examining the same code multiple times, more defects are identified.

Code analysis

While the actual code analysis largely depends on an analyst's ability to read code, we need to provide systematic support to make the analysis efficient and less painful. There are some heuristics we propose.

- **Familiarize yourself with the code**: Look at a particular version of the source code to understand the code structure, the algorithm used to solve the problem, communication pattern, language features used, naming conventions for variables/functions, coding styles (e.g., many small functions vs. a few large functions, OO-like vs. procedural). **It is often useful to read the latest (final) version**, as it is the end result of the development efforts. (All intermediate versions converge toward this version.) **It is also useful to**

read the early versions too, as the code tends to be small and simple, and thus easier to understand attributes such as programming style and initial structure.

  - Caveat: some people start from a code which is unrelated to the problem, such as a "hello world" example, or the code they have previously written for a different problem. Do not spend too much time in trying to understand the initial version in these cases.

- **Look at changes to examine intermediate versions**: **Examine a "diff" between particular versions** and examine what has changed to simulate what a developer did to produce a solution to the problem.

- **Look at big changes to determine their intention**: Remember that we captured the source code snapshots every time the compiler was invoked, so the granularity of code history is not uniform. If the data is fine-grained enough, a typical pattern of code history consists of "big" changes interspersed between a series of "small" changes. Suppose we measure the size of a change with the number of lines added and deleted. Common activities indicated by big changes are:

  - Additions of new functionality
  - Code refactoring (e.g., reordering functions in a file, variable renaming, etc.)
  - Addition of comments

  - Attempt for debugging (e.g. commenting/uncommenting a big code block, addition of print statements, etc.)
  - Completion of debugging (e.g. deletion of print statements)

  Therefore, we recommend looking at the big changes to determine their intentions to obtain the general understanding of how the code has been developed. Note, however, it is necessary to look at the unchanged parts of the code as well, because (1) the defects may exist in the parts that have not been changed at all, as some defects may not have been noticed by the subject, and (2) the defects may exist in the global logic, instead of being localized in the region of the fix, and (3) it may be easier to understand the whole code if the change is big.

- **Look at small changes before big changes to locate fixes**: We recommend examining the small changes before big changes, especially if that big change represents an addition of new functionality, because a common pattern is developers make a big change after they finish debugging. Therefore, the change before the big change that often represents a "fix" of the defect.

- **Take notes on the meaning of changes**: During the analysis, it is not unusual that some changes just do not seem to make sense. They could be random tries and errors to explore a possible cause of the problem, an attempt to fix a defect under a wrong assumption, or an introduction of a new defect. When it is difficult to judge whether it is a defect to be recorded, it is recommended to take notes on the current interpretation of such a change, so that it can be reexamined later after investigating other versions.

- **Skip versions**: The number of versions for each file can be up to several hundreds. Skipping "insignificant" versions is a good strategy to accelerate the analysis. However, it is important not to lose track of the high-level flow of the code change. Therefore, it is recommended to first go through all versions (except those which are identical to the previous version) quickly, and mark which versions should be examined more carefully.

Recording defects

We need to record defects. This is what is written down during the analysis. Reading the source code and identifying defects is an intensive task, so the procedure for immediate recording should be simple.

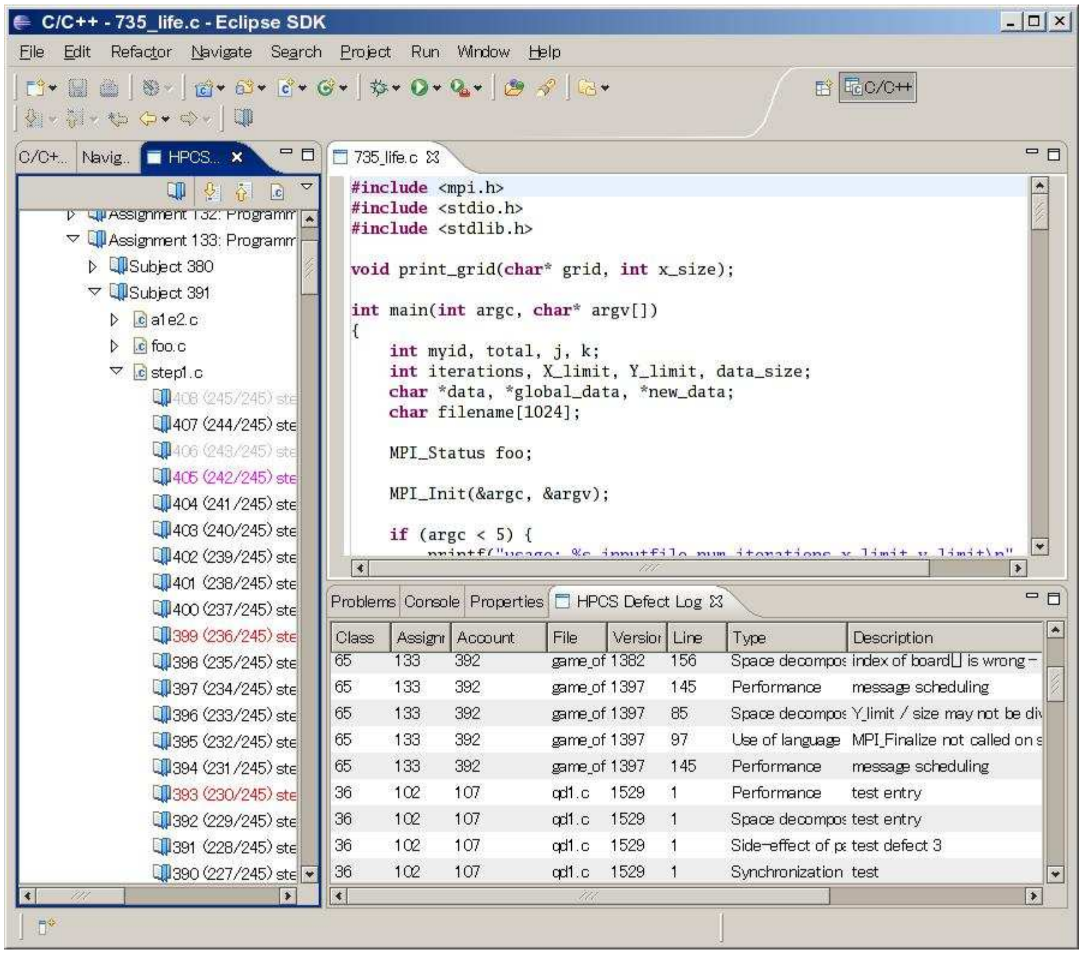
- **Location of the defect**: We record the source file version (ID) and the line number where the defect exists. For the purpose of data processing we assume it is possible to link each defect to a particular line.

- **Defect type**: Classify the defect into a particular defect type. This information didn't exist in initial defect recording but it was added after a classification scheme is developed. We will discuss defect types and classification schemes later in this chapter as well as in the next chapter.

- **Description**: Description of the defect.

Tool support

We developed a tool which provides a user interface for defect analysis. It is written as an Eclipse[1] plugin. Figure 5.3 shows a screenshot of the tool.


C/C++ - 735_life.c - Eclipse SDK
File Edit Refactor Navigate Search Project Run Window Help
C/C++
C/C+.. Navig.. HPCS...
735_life.c
Problems Console Properties HPCS Defect Log

Assignment 132: Programm
Assignment 133: Programm
Subject 380
Subject 391
a1e2.c
foo.c
step1.c
408 (245/245) ste
407 (244/245) ste
406 (243/245) ste
405 (242/245) ste
404 (241/245) ste
403 (240/245) ste
402 (239/245) ste
401 (238/245) ste
400 (237/245) ste
399 (236/245) ste
398 (235/245) ste
397 (234/245) ste
396 (233/245) ste
395 (232/245) ste
394 (231/245) ste
393 (230/245) ste
392 (229/245) ste
391 (228/245) ste
390 (227/245) ste

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

void print_grid(char* grid, int x_size);

int main(int argc, char* argv[])
{
    int myid, total, j, k;
    int iterations, X_limit, Y_limit, data_size;
    char *data, *global_data, *new_data;
    char filename[1024];

    MPI_Status foo;

    MPI_Init(&argc, &argv);

    if (argc < 5) {
        printf("usage: %s inputfile num_iterations x_limit y_limit\n"
```

Class	Assignr	Account	File	Versior	Line	Type	Description
65	133	392	game_of	1382	156	Space decompos	index of board[] is wrong –
65	133	392	game_of	1397	145	Performance	message scheduling
65	133	392	game_of	1397	85	Space decompos	Y_limit / size may not be div
65	133	392	game_of	1397	97	Use of language	MPI_Finalize not called on s
65	133	392	game_of	1397	145	Performance	message scheduling
36	102	107	qd1.c	1529	1	Performance	test entry
36	102	107	qd1.c	1529	1	Space decompos	test entry
36	102	107	qd1.c	1529	1	Side-effect of pa	test defect 3
36	102	107	qd1.c	1529	1	Synchronization	test

Figure 5.3: Analysis tool

The functionality of the tool:

- The tree view in the left pane can display all source code data in the database sorted by class, assignment and subject. The names of the files are listed under

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

the subject who wrote them, and the code versions are listed under each file. The code versions are color-coded based on the number of lines added, deleted or changed from the previous version: in the default configuration, gray means no change, black means less than 10 lines of change, pink means 10-50 lines of change, and red means 50 or more.

- The top right view is a source file view. The source files are displayed using the C/C++ and Fortran development environments for Eclipse, and all the standard functionality such as highlighting keywords, searching and navigation is available. By selecting two items in the source tree view, it is also possible to display a side-by-side diff.

- By focusing on a particular part of the source file view and pressing a defect button, a dialog window for entering defect information pops up. The file ID and the line number are automatically recorded. All information is stored into the database.

- The table view in the bottom right can display the list of defects identified and recorded. If a particular defect entry is clicked, the corresponding source file is opened and the defect location is focused. By doubleclicking the entry it is possible to modify an existing entry.

Reviewing the accumulated knowledge

As described above, in the first iteration the knowledge available is minimal. In case of the author, only basic things such as that concurrency is a common source of issues in parallel computing in general are known initially. After some data is analyzed, it is useful to review the defects identified so far and reflect the knowledge obtained to the subsequent analysis, because the analysis can be faster and more effective if similar defects exist in other code and the analyst knows what defects to look for.

5.2.2 Analysis results

Qualitative results

The primary result that directly come out of the analysis is the list of defects identified in the source code. Table 5.2 lists the defects that were identified two or more times. While these defects are not directly verified to be valid in this phase, it suggests that it is possible to identify some distinctive defects by analyzing problems at this level of complexity. Therefore, the basic feasibility of the code analysis method was confirmed. This was not obvious, because one anticipated criticism of this approach is the reading-based method is too labor-intensive to get any meaningful results.

Table 5.2: List of defects

Name	Short description
Bottleneck in message scheduling	Inappropriate message scheduling causing performance bottleneck
Bottleneck with file I/O	Performance problem due to multiple processes accessing the file or filesystem at the same time
Corrupted file output	File corruption because the data is written to the same file by multiple processes/threads at once
Dependency on the number of processes	An implementation that runs correctly only with specific number of processes
Excessive use of collective communication	Scalability problem due to an excessive use of collective communications
Fragmented Messages	Messages are sent in too small chunks
Hidden Serialization in Library Functions	Library functions containing internal serialization
Inadequate Communication Pattern	Inadequate communication pattern leading to a performance problem

Table 5.3: List of defects (continued)

Message Type Mismatch	Data type mismatch between message sender and receiver
Missing Barrier	Missing barrier
Missing MPI_Finalize	Missing finalization function
Missing wait	Missing wait function
Overlapped memory areas	Overlapping memory buffers for sending and receiving
Passing NULL to MPI_Init	Invalid parameter to the initialization function
Potential deadlock	Deadlock that can occur under specific conditions
Using the same randomization seed in all processes	Loss of the degree of randomness due to improper initialization
All processes hold the entire memory space	Inefficient memory allocation

Table 5.4: List of defects (continued)

Calling omp_get_num_threads in a serial section	The method works properly only in parallel section
Calling upc_free from multiple threads	Only one thread may call upc_free for each allocation
Missing upc_barrier before exit	upc_barrier should be called before exit to avoid an issue with some threads exiting before others finish using the data.
upc_memget or upc_memput from/to multiple threads	The function can only be used with a shared object with affinity to any single thread

Note that since our interests are in identifying HPC defects, we excluded some type of defects from the analysis.

- Simple typo that can be caught as a compile error, which is usually found and fixed immediately. Some typo that are simple but not caught with a compiler (e.g., = vs. ==) causes a problem, and we record such defects.

- Missing error checks. Almost all code we inspected has some statements missing error checks. We accept it unless the error checks are related to the characteristics of parallel execution.

- Missing memory de-allocation. Many codes do not free memory at the end of the program. While this is considered a memory leak in generic programming, this is usually acceptable as long as the program is completed and the memory is returned to the operating system. However, if the memory leak occurs inside the main computation loop and the memory consumption keeps increasing, we record it as a defect since this can cause real issues.

- Simple defects in the base language that has nothing to do with parallelism nor a parallel language feature.

There are some additional observations.

- As many students solving the same problem produce similar implementation, the analysis can be done efficiently by just looking for "anomalies" from a standard solution.
- The defects that do not always produce a failure seem to be quite common. These defects are expected to be hard to debug.

Classification

In our approach, the defect classification scheme is defined and refined through the iterative process. An initial classification scheme is shown in Table 5.5, which was built from the defects in Table 5.2 by coding defect descriptions and grouping similar defects together. The scheme is then refined based on the feedback from experts. We will discuss the refinement process in the next chapter. The defect types added in the refinement process are not included except for the memory management type, which is needed to present the quantitative results.

Table 5.5: Initial classification scheme for HPC defects

Type	Sub-type	Brief Definition
Use of language features	—	Erroneous use of parallel language features
Space decomposition	—	Incorrect mapping between the problem space and the problem memory space
Side-effect of parallelization	I/O hotspots	Serial constructs causing correctness and performance defects in parallel contexts
	Hidden serialization in library functions	
Synchronization	Deadlock	Incorrect/unnecessary synchronization
	Race	
Performance	Load Balancing	Performance defects in parallel contexts
	Message scheduling	
Memory management	—	Inadequate memory management
Algorithm	—	Program logic not matching the intended purpose of the code

Quantitative analysis

The quantitative analysis is conducted in the following way. Each raw defect record consists of the information on the location of the defect (source file version and line number) and the defect type. Two quantitative measures, the defect occurrence and duration, are computed from the raw data.

The defect occurrence, i.e., how frequently the defect occurs, is the most direct measure to assess the importance of defect patterns. It can be computed by simply counting the number of defect instances identified in the data analyzed.

The defect duration, i.e., how long the defect stayed in the source code, is also important because it represents relative difficulty of fixing that defect. We use the following algorithm to calculate the defect duration.

1. Suppose a defect is identified in a particular line of the version X. Compare it the previous version $(X-1)$ of this file, and check if the content of the line also exists in $(X-1)$.

2. If the corresponding line does not exist in $(X-1)$, that means the defect was first inserted in X. If it does exist, check previous versions $(X-2)$, $(X-3)$, etc. until the version the defect was inserted is found. If the corresponding line exists in the first version of the source file too, the defect was inserted in the very beginning.

3. Similarly, compare version X with the subsequent versions $(X+1)$, $(X+2)$, etc. until the version no longer contains the line containing the defect. If the corresponding line exists in the final version, it means the defect was never

fixed.

The version comparison can be implemented using a text diff algorithm which matches the content of two files and determines which lines have changed. There are a number of different diff algorithms, but they can usually detect changes in blocks so the content can be matched to some extent even if they are not in the same line number. We implemented a tool to calculate the defect duration using the Python difflib library [2].

Note that this is an approximation of the actual defect duration, since if the line containing a defect is modified the defect is not fixed, the duration may be calculated shorter. To mitigate this problem, we allow multiple defect entries to be grouped as a set pointing to the same defect instance. The duration of this defect is then the range that covers the duration of all entries in the group. For example, if the duration of the entry A is (X, Y) and the duration of the entry B is (Z, W), and A and B point to the same defect instance, the duration is $(\min(X, Z), \max(Y, W))$.

Once the duration (X, Y) is determined, an effort metric is further computed using an empirical algorithm for estimating effort presented in [40]. The algorithm is to use the duration of the development time for which the defect stayed in the code as a measure of effort. If the subject never takes a break, the effort is simply calculated by (the timestamp for Y) - (the timestamp for X). Since development is interrupted for various reasons, we need to exclude break time to obtain a meaningful estimation. The algorithm is to assume that a time period with no development activities longer than a threshold t is a break and assign a compensation value t_c as the effort spent

[2]http://docs.python.org/lib/module-difflib.html

during this period. Based on the observational study conducted by Hochstein [40], we set t to 45 minutes and t_c to 5 minutes.

Figure 5.4 and 5.5 show the boxplots of the distributions of defect occurrences per subject. Each figure presents the distributions of the number of defects identified from the solutions of the Game of Life problem in MPI and C written by 21 students. There were 264 defect instances identified in 1,397 source file versions. The distributions are shown by defect type: (1) Language usage, (2) Side effect, (3) Space decomposition, (4) Synchronization, (5) Performance, (6) Algorithm, (7) Memory management, (8) Other, and (9) Total. Figure 5.4 presents the distributions for defects that were resolved during development, while Figure 5.5 shows the defect distributions for those that remained unfixed in the final version.

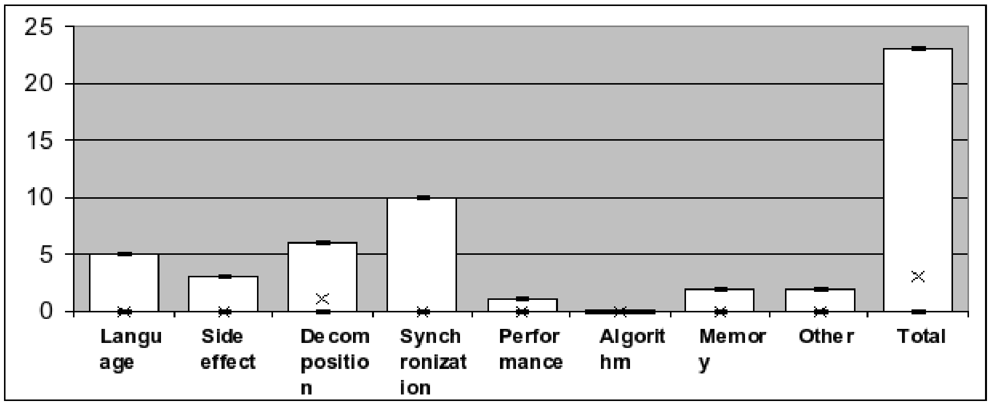


Figure 5.4: Distribution of the number of defects per subject (defects that were fixed during development

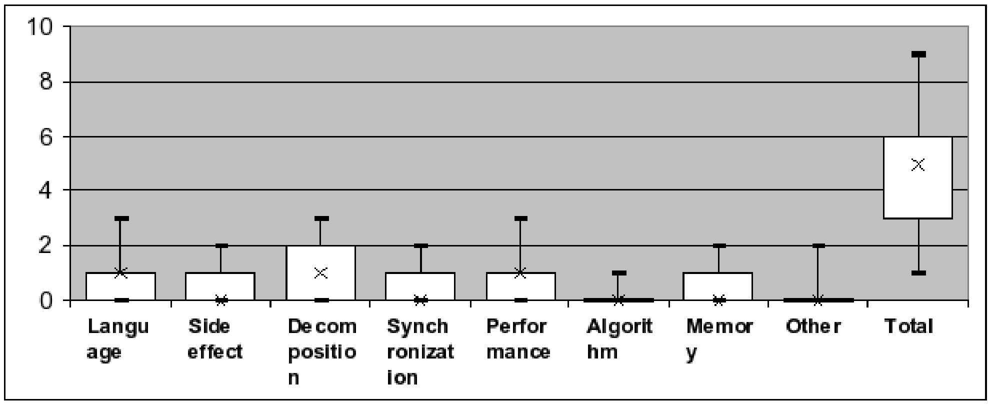


Figure 5.5: Distribution of the number of defects per subject (defects that were never fixed)

The analysis results show that each student made 10.2 defects (5.4 resolved, 4.8 not resolved) on average during development. One observation is there are a considerable percentage of defects that were never fixed. The defects left in the final version do not necessarily lead to a failure (i.e., a run-time error which the students can observe), as some HPC defects surface only under specific conditions and input parameters, or validating whether the output is correct is difficult. Unless students test the code extensively, these defects stay unnoticed.

Figure 5.6 shows the distribution of estimated time spent to fix defects for each defect type. This indicates how difficult the defects were to find and fix.

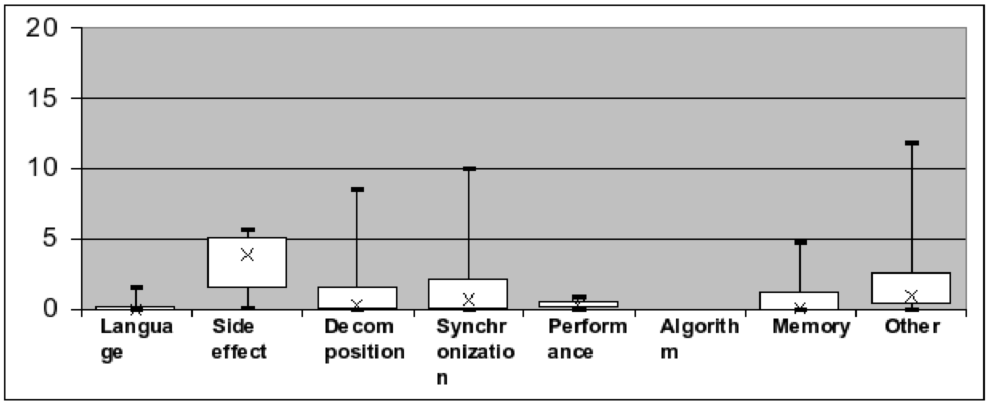


Figure 5.6: Distribution of the estimated time (in hours) spent to fix defects by type

Figure 5.7 shows the same results for the defects that were never fixed, assuming the duration of the defect ends at the final version. This indicates how early they were inserted to the code.

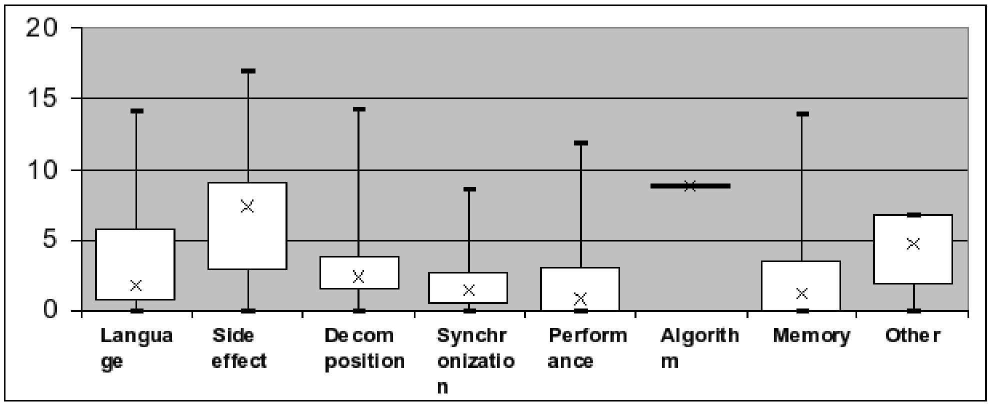


Figure 5.7: Distribution of the estimated defect duration (in hours) for those never fixed

The results indicate there are differences in average time to fix between defect types. For example, the side effect of parallelization type (e.g. I/O hotspots) is inserted early in the code, and it takes time to fix.

While the dataset is a representative of novice HPC developers solving a small problem, it is not clear how the quantitative results can be generalized to broader contexts. For example, few algorithmic defects have been found in this dataset, even though many HPC practitioners suggest algorithmic defects are most common yet hard to debug.

5.2.3 Reliability study

Since the above results were obtained solely based on the analysis by the author, the following questions come up.

- Are the analysis results valid? As we previously argued, the primary purpose of the analysis is to develop initial defect patterns which is verified and refined in a separate sub-process, so the validity of the qualitative results (defect lists) are evaluated there. On the other hand, the reliability of the quantitative results (occurrences and time to fix) needs a different kind of evaluation.

- Can other analysts repeat the reading-based method (including the heuristics) to identify defects? This is important when other researchers repeat the study in the HPC domain as well as other software domains.

A general difficulty in studying these questions is the analysis requires enough understanding of the problem and the solution. We conducted a reliability study with a graduate student who has previously taken an HPC course in which we conducted a classroom study, and thus has basic knowledge and experience with

HPC development at the time of the reliability study. We chose the Game of Life problem in MPI and C, since the subject of the reliability study has solved the same problem before, thus understood the problem and the basic approach for a parallel solution well.

For the study, we used the dataset from one particular student, which consists of 192 unique source file versions (245 overall versions including 53 unchanged ones). The study was conducted in two separate sessions. In the first session, we provided the description of the problem and the heuristics, and told the subjects to go through investigate intermediate versions. In this session, the purpose was to verify whether they can apply the heuristics to identify defects that were found and fixed during development. In the second session, we presented the classification scheme and told them to identify defects in the final version. The purpose was to verify whether the defects can be detected with the help of already established knowledge about the defect patterns.

The results are summarized as follow.

- The subject reported 14 defects, while the analysis by the author has identified 16 defects. 9 out of these defects were common. This supports these are indeed defects, and the reading-based method can reliably detect them.

- All of 5 defects only reported by the subject of the reliability study are confirmed to be valid. They have been excluded because they are simple enough to be detected by a compiler. The version of the heuristics presented to the subject didn't explicitly say what kinds of defects should be excluded. That information has been added.

- The analysis of 7 defects which were missed by the subject are summarized as follows.
  - One defect missed in the final version is that the program fails to run correctly when it is run with one process.
  - Four defects missed in the intermediate versions are of the space decomposition type: incorrect conditions for determining the boundary of the local problem space and incorrect indexing related to parallelism. Note that these defects were identified in the intermediate versions which the heuristics suggest to skip. They were caught by the author since this data was analyzed more thoroughly than the heuristics suggest. Considering the amount of time the subject spent analyzing the code, missing these defects is acceptable. However, additional training or heuristics may be needed to detect decomposition defects effectively.
  - Two defects missed in the intermediate versions are defects that do not have to do with parallelism: one is a pure sequential defect related to the use of the fscanf() function in C which a compiler doesn't detect, and the other one is a pure error which is to assign assign 0 instead of 1 when the cell is alive. Again, it is understandable that these were missed, since the heuristics given to the subject did not cover them. We can argue these are not as important as others for the purpose of building patterns, although they have stayed in the code for long time.

5.3 Data collection from self-reported defect logs

One major criticism to the reading-based approach is that it is too labor-intensive. If there is an easier, less costly means to collect defect data which can be used to identify patterns, it is natural to argue that it should be tried before taking the reading-based approach.

An example of such an alternative approach is to ask developers to report the errors they made. Project developers are usually the best people to be asked about the defects that occurred in the project code, as they are the ones who actually insert, find and fix defects. Similarly, students participating in a classroom study should be aware of the issues they ran into and how they managed to solve them. Therefore, if we can make them keep a record of the defects, we can then use it to identify patterns. A self-reported defect report is more or less "subjective" as developers can report defects only if they recognize and remember them. It is expected that the more impressive defects are, the more likely they will be reported. This is desirable because developers are expected to better remember the defects they had difficulty in finding and fixing, and our goal is to reduce debugging costs by identify such defects.

In the rest of this section, we describe experience from two studies in which we attempted to obtain self-reported defect data from novice (student) developers.

5.3.1 Modes of data collection from self-reported defect log

Defect recording can be concurrent (i.e., developers fill out defect forms during development, as soon as they find and fix defects), or retrospective (i.e. they remember the characteristics of defects and fill out the forms later). Concurrent recording can minimize the risk of losing information because of forgetting, but since debugging is a labor-intensive task, recording defects while working can interfere with their thinking process. Retrospective recording can be implemented easily, but it has a risk of losing information about defects that were made but forgotten. In the studies we conducted, we took a concurrent approach.

We developed and used several versions of defect forms. The form can be a paper or a computer-base logging tool. Appendix B presents the actual forms actually used.

5.3.2 Results

Iteration 1: pilot classroom study

In the first study, we asked students to fill in the defect form 1, which is shown in Appendix B. Out of 11 students who participated in the study, 8 students filled in at least one defect entry. The total number of defect entries collected was 29. However, 12 of these defects reported referred to a compile-time error. The remaining 17 defect reports were examined and classified using the classification scheme presented in Table 5.5. The results are summarized as follows.

- Sequential error leading to incorrect outputs, e.g., typing '-' instead of '+' in one of the conditions, typo in variable names, failure to initialize a variable, underflow due to integer division, wrong copy/paste. (6)

- Defects related to side-effects or parallelization, e.g., failure to understand which part of the program is executed by all processes and which part is run by one process (5)
- Defects related to synchronization, e.g., missing blocking, I/O race condition. (3)
- Defects related to parallel language features, e.g., failure to call an initialization function (2)
- Defects related to space decomposition, e.g., array out of bounds (1)

Since we could not capture source code data in this course, we were not able to directly compare the results with those from the code analysis. One observation, however, is that 6 of the 17 defect entries (including the compile-time errors) describe issues which can also occur in sequential programming using a base language, thus only 11 would be classified as HPC defects, which means each student who was willing to to keep a defect log recorded only 1.4 defects on average, as opposed to 5.4 resolved defects per subject observed in the code analysis. Therefore, a hypothesis is that self-reported defect logs do not provide the data as complete as what the code analysis. Nevertheless, since the goal of this process is not to achieve perfect detection, the data obtained can still be useful as the source of base patterns. Moreover, the fact that the reported defects can be classified into the defect types suggests that these types are reasonably defined.

Iteration 2: revised classroom study

In the next study, we revised the defect form to explicitly ask whether the students think each defect is related to parallelism. The defect form 2 is also shown in Appendix B. We made this version of the defect form available online on the web system, so that students can directly fill-in defect entries on the computer.

We were not able to collect much data from this study either. Out of 10 students who participated in the study and provided us with other data such as source code snapshots, only 5 students filled in at least 1 entry, and the total number of entries recorded is 13 (2.6 defects per student). Below is a summary of these defects.

- None of the entries were simple compile-time errors, and only one entry was related to a pure sequential defect, which is mistyping '==' as '=' in the *if* statement.
- Five entries were related to space decomposition.
- Two entries were related to synchronization, e.g., too many barriers causing a performance problem, and incorrect output order due to improper synchronization.
- Two entries were related to the use of parallel language features.

- One entry was reported to be related to insufficient memory allocation, although the student didn't report how he verified it.
- Two entries were related to side-effects parallelization.

Again, the results suggest that the self-reported logs can provide data on HPC defects to a limited degree, and that they support that the initial classification is reasonably defined. If the self-reported log is not an effective means of primary data collection, it is possible to use a mixed method. For example, if the subjects are presented with the results of code analysis right after they finished the assignment, they may be able to recall the problems they had and verify whether or not they actually recognized the same defects during development. To make this possible, the code analysis needs to be performed near real time.

One caveat, however, is that we received complaints from the study subjects that keeping logs is a labor-intensive task for them. While we want to collect as much data as possible, we need to keep the data collection simple from their perspective. The students are usually busy with working on an assignment itself, and if the data collection requires too much overhead, they ignore the procedure or enter incorrect information. As a result, the obtained data can be inaccurate and inconsistent. Moreover, collecting defect data can involve additional difficulties since some people seem to feel uncomfortable about admitting they have made some defects. Extra care must be taken to ensure that the data is kept anonymous, and the collected information is never used against the subjects. For example, it would be difficult to collect defect logs in an environment where the defect rate is used to evaluate the performance of individual programmers. In the next chapter, we attempt to obtain peoples' knowledge using a very different approach, so that we can obtain information avoiding these issues.

Chapter 6

Developing Experience Base and Extracting Expert Feedback

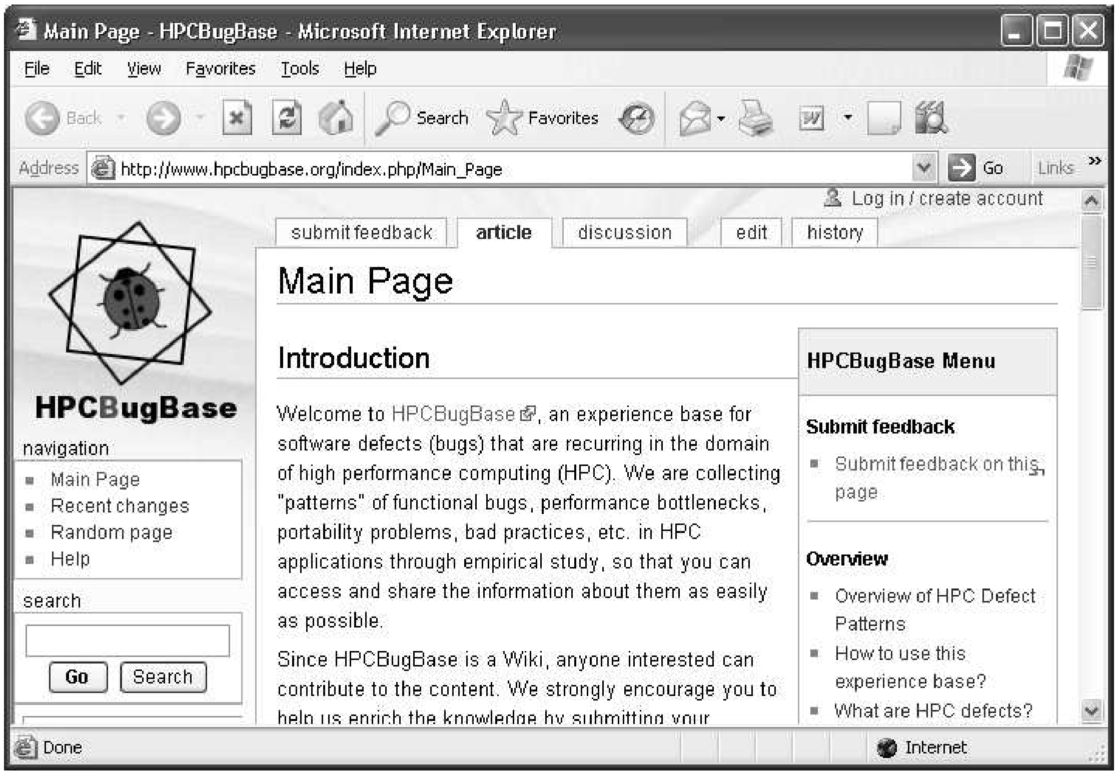
Having identified a set of recurring defects in the previous chapter, we now discuss the second activity of the problem solving approach: knowledge refinement. There are two main goals for this activity. The first goal is to organize the analysis results into the structures which can represent the knowledge of "defect patterns" at various levels of abstraction. We built an experience base system named HPCBugBase[1] and put the structured information as its initial content. Users of the experience base system are expected to have different demands; some users may want to directly look at raw defect examples as they appear in the source files, while others may be more interested in the description of higher-level defect types. The experience base system needs to provide a good interface to allow various users to access the information as they wish. At the same time, it is important that users can keep track of the relationships between the information at different levels: how high-level patterns are supported by low-level empirical evidence. Furthermore, the system should also have a mechanism to allow users to provide feedback. We discuss how we structure the patterns, and describe how we design and implement the experience base.

The second goal is to evolve the patterns. Since the initial defect patterns in the experience base were derived from the data having limited contexts, they

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

need be refined in order to cover a broader range of defects that appear in real HPC applications. As we discussed earlier, the HPC community already possesses a great deal of implicit knowledge about recurring defects at the level of individual practitioners, but such knowledge is mostly hidden within the minds of individuals. We demonstrate that our approach can help the process of making tacit knowledge explicit by conducting a series of interviews with different types of HPC stakeholders. By presenting them with the initial content in the experience base, all interviewees are able to provide feedback, which results in the enhancement of defect patterns and the experience base system. The interview results also show how the views of various users on defects are very different. This diversity supports the need of the experience base system.

6.1 Organizing knowledge of HPC defects



Main Page

Introduction

Welcome to HPCBugBase, an experience base for software defects (bugs) that are recurring in the domain of high performance computing (HPC). We are collecting "patterns" of functional bugs, performance bottlenecks, portability problems, bad practices, etc. in HPC applications through empirical study, so that you can access and share the information about them as easily as possible.

Since HPCBugBase is a Wiki, anyone interested can contribute to the content. We strongly encourage you to help us enrich the knowledge by submitting your

Figure 6.1: HPCBugBase: defect experience base

Based on the low-level raw data obtained from the code analysis, we prepared the initial content for the defect experience base. Figure 6.1 is a screenshot of the front page of the experience base. Figure 6.2 describes the content structure we introduced to organize the content. There are five categories defined.

- Classification scheme
- Defect type
- (Description of) specific defects

- Individual defect instance (code sample)
- Article

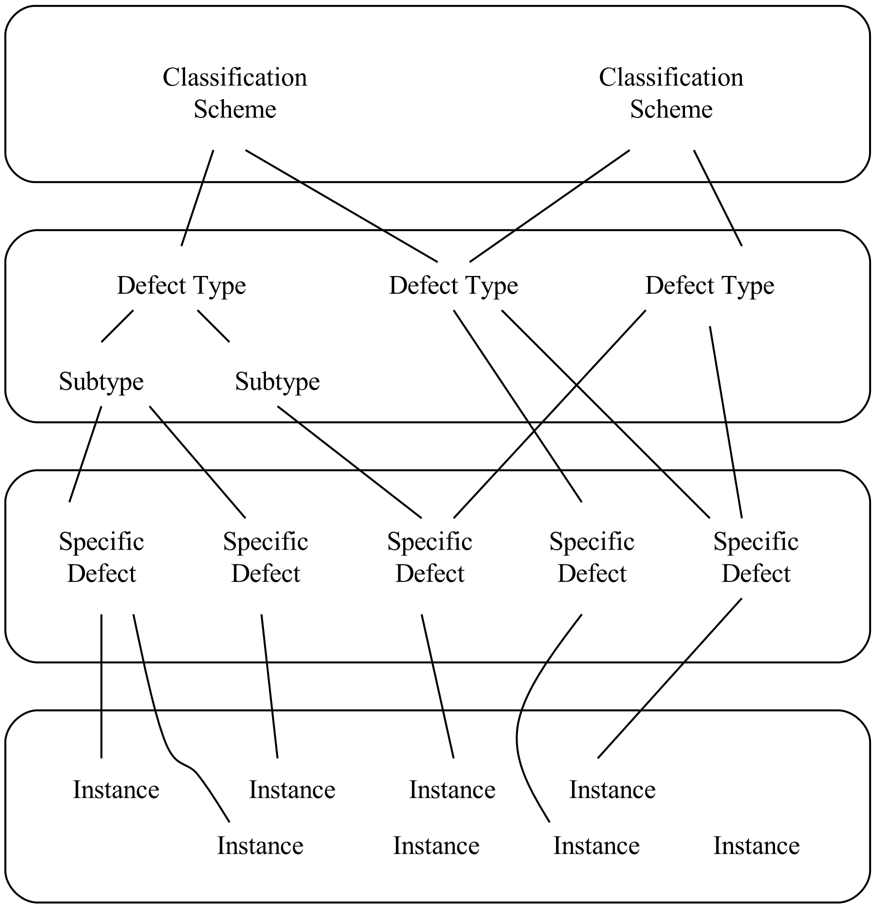


Figure 6.2: Structure of pages

Each content category is described below. Note that the structure is not permanent, so it can be changed if the feedback from users suggests a better page structure.

6.1.1 Defect classification scheme

A defect classification scheme is a collection of defect types, which we describe next. It is intended to cover all defects recurring in the HPC code. As we keep evolving the scheme, we obtain multiple versions of the scheme. There are also schemes developed by other researchers.

As shown in the Table 5.5 of the previous chapter, the initial classification scheme consists of top-level defect types: use of language features, side-effect of parallelization, space decomposition, synchronization, performance, and algorithm. In addition, the catch-all type called "other defect types" has been prepared to allow users to suggest a new defect type.

6.1.2 Defect type

A defect type is an abstract representation of a set of defects. It is defined by comparing and grouping similar defects together.

When discussing "defect patterns" in this dissertation, we mainly refer to the knowledge described at this level. In addition to the characteristics of defects that belong to the defect type, an important content of each defect type is the knowledge on how to detect defects and how to avoid defects.

It is possible to define defect types with different abstraction levels, which form a type hierarchy. The defect types used as top-level types of a classification scheme have a highest level of abstraction, as they are meant to cover a broad range of defects independent of language and platform. The "sub-types" with lower level of abstraction are context specific, and they can include more detailed advice on defect detection and prevention.

For example, we define “performance defect” as one of the top-level defect types. While it covers a quite wide range of defects, since there are so many defects identified that damage execution speed, giving reasonable advice for how to detect and/or avoid them is difficult. Therefore, we define “message scheduling problem” as a sub-type of the performance defect to describe more specific information.

Template

The template for defect types includes the following entries.

- Name of the defect type (**name**)
- List of sub-types and specific defects that belong to the defect type (**entries**)
- Advice for detection, how does someone know there is a defect? (**symptoms**)
- Advice for solving and avoiding a defect (**cures and preventions**)

6.1.3 (Description of) specific defect

The description of each individual defect identified constitutes the information at the second lowest level. Typically each defect description corresponds to multiple instances of the same or very similar defects. At this level, we focus on describing a defect itself: why it is a defect. If necessary, sample code or pseudo-code is used.

Template

Template

- Name of the defect (**name**)
- What was wrong in the code? (**fault description**)
- List of defect instances (**examples**)
- Other findings and contexts (**other findings**)

6.1.4 Individual defect instance (code sample)

We call the actual samples of HPC applications containing defects the "defect instances". The information stored in the entries of this can be the complete or partial source files with the information on the location of defects, or the links (URL or reference information) pinpointing the location of some source code containing defects. They are the lowest-level data directly connected to individual occurrence of defects in HPC applications.

While it is not always possible to provide a copy of the code samples in the experience base due to copyright issues, concrete examples are important to present how each defect appears in actual implementations. They can also be used to evaluate a defect detection tool by testing whether it can detect the defect instances.

Template

- Link to a defect description (**defect**)

- Source code containing the defect (**code**)
- Where the defect was found: file, version and line number, or a link to the source code containing the defect (**location**)
- When the defect was inserted into the code and when it was fixed (**time to find and fix**)
- Other findings and contexts (**other findings**)

6.1.5 Article

Finally, there are other kinds of information that are useful but do not fit into the structure above. Currently we do not use a template for these articles.

- Anecdotal information about defects
- Good programming practices
- Information on potential data sources
- Thoughts on available technologies
- Links to external references

6.2 Design and implementation of HPCBugBase

There are various types of potential users who could make use of empirical knowledge about defects. The experience base must be flexible enough to meet the needs of each of these users. At the same time, the base must support incremental evolution of the content. In this section, we describe the requirements of the EB, and a design that meets these requirements.

6.2.1 Potential users

The experience base system should interact with the users in the "give-and-take" manner. On one hand, users act as "consumers" of knowledge who access the content to obtain the information they need. On the other hand, they are also expected to act as "producers" of new knowledge as feedback to the existing content. Of course, not all users have to act as both, but the key is to encourage as many users as possible to participate in the knowledge refinement process.

- **HPC developers**: As consumers, their motivation is to identify defect samples similar to theirs to help debug their code. When they are designing a new project, they might also look for hints to prevent defects. As producers, they can submit their defect samples and/or suggestions to improve the system.
- **Technology providers**: As consumers, their motivation is to use the information in the system to verify whether their technology such as a programming language or a defect detection tool can solve the problems that occur in HPC systems. Their interest is in the empirical support of their technology by confirming it can be used to avoid and/or detect the defects recurring in real HPC projects. If they are designing a new technology, their motivation is to identify defect-prone language features and tool demands. As producers, they can propose new language/ language feature to prevent particular types of defects.

- **Professors**: Professors who are teaching an HPC-related course need and information on the defects made by novices so that they can provide more effective lectures and assignments. As consumers, their motivation is to plan a course to teach how to avoid defects. As producers, they can submit their experience in teaching in HPC.

- **Students**: Students learning an unknown HPC language in an HPC course are usually novice HPC programmers. Their experience can vary from none to some basic level. As consumers, their motivation is to learn common pitfalls in HPC programming. They want to know when they should be careful about making defects. As producers, they can commit their defect samples.

- **Software engineering researchers**: As consumers, Software engineering researchers who seek good software engineering practices for HPC need to understand problems in HPC development from empirical evidence. As producers, they can propose practices suitable for HPC. They can also submit the results of empirical study to evaluate claims of tools/languages.

Figure 6.3 illustrates the characterization of these users with regard to the relative experience and expertise in software development using an HPC language.

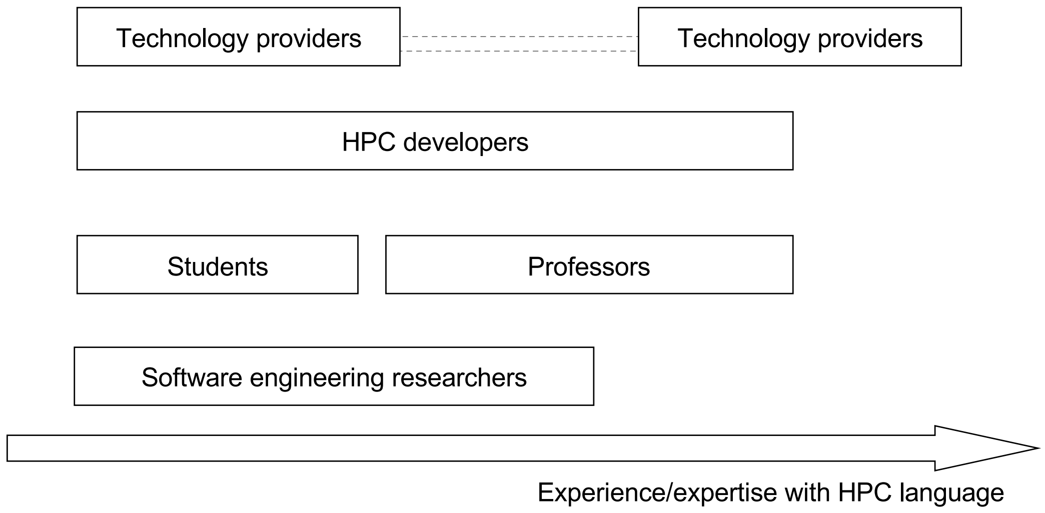


Figure 6.3: Characterization of the users of HPCBugBase

6.2.2 Requirements

In order to fulfill the requirements of the users above, the experience base system must provide the following.

- A means to access the data. The content of a base Wiki system consists of a set of pages containing texts and images, with the links to other pages. Unless pages and links are carefully structured, it can end up with many unstructured pages and users can easily get lost as they move between pages by clicking the links. The other way to access Wiki content is by search. Its effectiveness depends on whether appropriate keywords are embedded in the content.
- A means to input the data. Entering new data should be easy and efficient enough. Authoring individual Wiki pages manually can be time-consuming.
- A means to accept feedback from users. While a base Wiki system provides

an interface to allow users to directly edit the Wiki text, they need to learn a small number of rules and grammars, which can be an adoption barrier. Since we do not want to lose the opportunity to capture the response of users just because they do not feel comfortable with Wiki editing, we should provide a simpler, easier-to-use interface.

- A means to assess the quality of the content. As more users put information into the system, it becomes an important issue how to manage the information on the reliability of each entry. A mechanism for quality assurance is needed.
- A means to present credits: As the input from users is vital, there should be a mechanism to remark who has contributed to the content. This can be a good motivation for some users to contribute to the content.
- A means to ensure anonymity: Complementary to the previous requirement, there are also cases in which we should not reveal which applications the defect samples came from, and/or which developers made them.

As we implement the experience base system using a Wiki, some of the requirements above are automatically fulfilled. When the base Wiki system lacks the necessary functionality, however, we need to extend the system to add necessary features.

- A form-based interface to accept user feedback
- An interface to allow users to submit defect examples
- An interface to allow users to add keywords for each entry

Finally, there are several non-functional requirements to keep the system usable.

- Response time: should be short enough under normal conditions.
- Availability

6.2.3 Implementation

The front end of the Wiki interface is a set of "pages", each of which can store arbitrary texts as ordinary web documents. While being able to describe detailed information in free texts can make the system flexible, it is desirable from the efficiency of data access that these pages are organized to follow a certain structure.

The second activity is to refine the identified patterns through the reviews by domain experts. This activity is closely coupled with the development and improvement of an experience base (EB) for HPC defects. As described in Chapter 4, the EB is the core of the entire problem solving approach in this dissertation. It should (1) store the defect patterns identified by data collection activities, (2) encourage stakeholders in the HPC community to access existing patterns to gain knowledge and submit feedback to refine the patterns, and (3) serve as a repository of empirical evidence which forms the basis for derivative products developed by packaging accumulated knowledge. While all of these functions above are important, the requirements on the user interface for the first and third functions are less severe as these are mainly accessed by software engineering researchers who are usually highly motivated to use the system from the beginning. On the other hand, the second function, which we will discuss in this chapter, needs to be accessed by broader users, thus requires a good use interface. As described in Chapter 1, many developers of an HPC project are scientists and students who are not necessarily willing to use a complicated technology if they have to make a lot of extra efforts. Therefore, it is crucial that the experience base system is designed and implemented to be useful and easy enough for them, because otherwise the knowledge stored in the system would never be accessed. In the first part of this chapter, we describe the design and implementation of the web-based experience base system named HPCBugBase.

Front page user interface

The front page is the entry point of the experience base website, unless a user directly visits other pages from external search engines or other links.

- Brief introduction
- The list of the current defect types (classification scheme)
- Shortcut links to the important features of the system

Backend database

The base Wiki system stores all the content in MySQL database. The source files and defect samples are stored in a separate database.

Content structure

The system directly stores the content structure described in Section 6.1. Each "Wiki page" contains the text describing an entry describing one of the five categories. We use the functionality of the base Wiki system to tag the page with the category it belongs to.

The following data entry templates have been defined. We do not define a template for the "Article" category because the content in this category is free-format text.

When recording the various kinds of defect data described above, one approach is to use data entry forms with pre-defined formats. While this approach makes the data uniform and the data analysis easier, there is a risk of making data input difficult and losing information. In this experience base, we provide several templates for data input. However, the use of the templates is voluntary. When the data does not fit the existing templates, to the user may create a new template or enter the data entry in a free-form manner. Even if the template is used, it should be permitted to have some items left blank when they are not available.

Initial content

We have entered the initial content generated from the data obtained in Chapter 5. At this phase, all content was entered manually.

This scheme has been written in a separated page using the template for defect classification schemes. In addition, the scheme has been put on the front page so that it is visible to the first-time users.

Feedback form

As a Wiki system, the content can be directly edited by anyone if they are familiar with a Wiki grammar. There are problems. It is tedious to directly modify the wiki content Even if they know the grammar, modifying the content consistently with existing content may not be easy Changes are visible to other users.

We have modified the base Wiki system to allow users to submit their feedback by just filling out the forms. On the top of each page, there is a link to the feedback form.

6.3 Evaluating and refining defect patterns

In this section, we describe a series of interviews we conducted to validate the usefulness of the knowledge stored in the experience base system. Through the interviews, we attempt to obtain insights on several things. First, we would like to evaluate the validity for the initial defect patterns from several aspects. The following questions should be answered.

- Are the initial defect patterns derived from the data analysis "valid", i.e., are they indeed the type of defects observed in HPC applications?
- If it is a valid pattern, whether it is unique to novice programmers, or it is also frequently occurring in the code written by an experienced developer

Second, we would like to extract new knowledge about HPC defects from the interviewees. Since we attempt to let them provide it as feedback to the existing patterns, it leads to the validation of the hypothesis that presenting the initial patterns helps extract tacit knowledge. The knowledge obtained is incorporated into the experience base system, so this is really the realization of knowledge refinement.

- Whether they can suggest a new defect example for the existing defect types
- Whether they can suggest a new defect type
- Whether they can suggest advice for defect detection and/or prevention

Finally, we would like to characterize the interviewees with regard to the perception of defect patterns.

- Which defects they think are important
- How they would use the experience base

6.3.1 Interviewees

The interviewees are selected from the experts who agreed to help us in order to cover a variety of stakeholders in the HPC community. Due to the above purposes, we did not interview complete novices. Figure 6.4 illustrates the type of the interviewees.

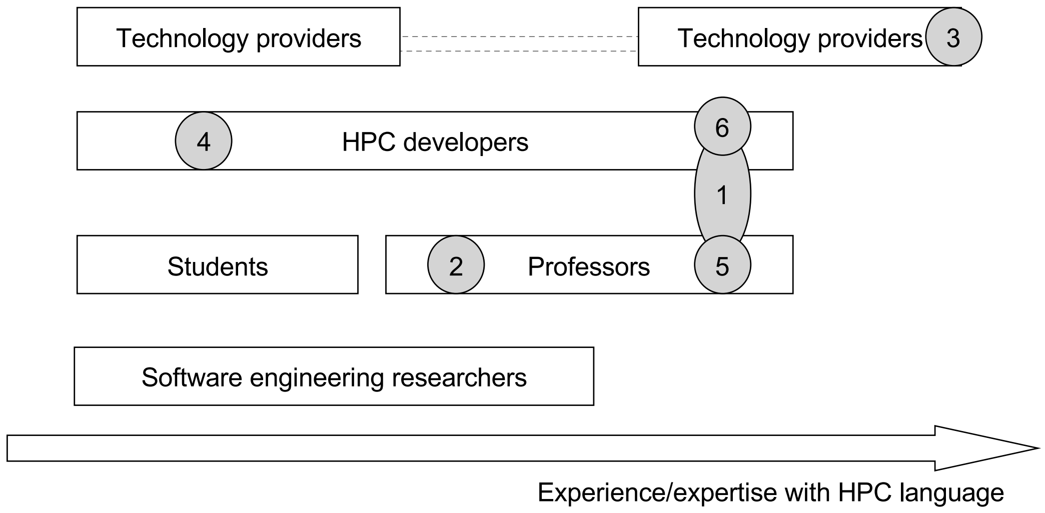


Figure 6.4: Characterization of the interviewees

- Interviewee 1, 5 and 6 are considered to have similar expertise in HPC languages. Interviewee 1 is both an HPC developer and a professor. Interviewee 5 is a professor teaching courses for HPC programming. Interviewee 6 is an experienced HPC developer who is actively working with raw MPI programming.

- Interviewee 4 is also an experienced programmer, and he is one of the core developers in an HPC project. However, he never directly modifies the code that would affect the low-level processing for parallelism. He does not have prior experience with a specific HPC language.

- Interviewee 2 is a professor with expertise in programming language and compilers. His interests and technical expertise are close to technology providers. He has deep insights on generic parallel programming and emerging languages, although he does not have as much experience with HPC languages as interviewee 1, 5, or 6.

- Interviewees 3 (interviewed as a group of three people) work for a technology vendor. They have high-end expertise in parallel scientific computing, and develop applications, libraries and benchmarks for advanced HPC technologies.

6.3.2 Interview procedure

The first interview was conducted in person. All the subsequent interviews were done by phone. In an in-person interview, we presented the interview material as well as the content of the experience base using a projector. In the phone interviews, we sent the URL of the experience base system beforehand so that the interviewees they can look at the content in front of their PC. During the interview, we presented the interview material that was prepared as the "overview page" on the experience base. The overview page was carefully organized beforehand, so that it can help the interviewees efficiently walk through the top-level defect types and understand the context of the study. The idea is to demonstrate that tacit knowledge about defect patterns which usually exists in the brain of experts can be made explicit more easily if they are presented with initial knowledge they can respond to. The interviews were conducted in the following order.

1. Introduction of the interview. Our motivation for building an experience base, differentiating factors for HPC which can affect the characteristics of recurring defects and the information we want to get from the interview are explained. A question to be asked is as follows:

- Are there anything unclear?

2. Explanation of the sample problem used to explain defect types and sequential implementation. Questions to be asked are as follows:

    - Is the sample problem clear?

3. Explanation of each defect type and examples. For each type, the following questions are asked.

    - Is this defect type clearly described?
    - Is the defect type named appropriately?
    - Is this defect type frequently observed in HPC code?
    - Is this defect also made by experienced HPC developers, not only novices?
    - Have you seen a bug of this type in your project?
    - Can you think of other examples for this defect type?
    - Other comments on this defect type?

4. Other defects

    - New defect type?
    - Restructuring of the defect classification scheme?

5. Explanation of the experience base system, especially about the mechanism to accept feedback

    - What else would you like to see in the experience base system?

In each section, we ask these questions to get response in a structured way. At the same time, we encourage the interviewees to speak freely as they think of something to say. When necessary we ask follow-up questions to clarify their comments and get additional thoughts about particular topics that come up.

6.3.3 Evaluation criteria

We processed the interview results qualitatively and quantitatively. The qualitative measures are defined from the answers to the evaluation form we ask the interviewees to fill out after the interview. The questions include:

- Whether they think the examples are good representatives for the defect type described.
- Whether they think the content is clearly written.
- Whether the interviewees would recommend using the EB system to each user type defined in Section 6.2.1. The answers imply whether they think the content is valuable to these users. An expected reaction is they would recommend the EB to at least one user type. For other types of users, we hope to get insights on why they wouldn't recommend the current system.

The success criteria is that the interviewees return a positive response to these questions. As we interview people with different backgrounds, experiences and interests, we further demonstrate the diversity of people's perception on defects by comparing the reactions of interviewees.

The quantitative measures are defined in terms of the amount of increased knowledge, including:

- The number of newly identified defect types
- The number of newly identified defect examples for existing defect types
- The number of newly identified techniques for defect preventions

The success criteria is that each interviewee can provide at least one contribution for any of these. In the subsequent sections, we summarize the results of the interviews.

6.3.4 Interview 1

The first interview was conducted in person. In a two-hour interview, one new defect type, five new defect examples, and three techniques for defect prevention have been suggested.

Added defect type

- Memory management, with three sub-types

Added defect examples

- For space decomposition: inconsistency in the assumption on the origin of a Fortran array.
- For side-effects of parallelization: I/O error due to the global limit on the number of files that can be opened at the same time

- For synchronization: example of all-to-all communication causing race conditions
- For synchronization: change in the output value due to the different order of summation in an integral operation- hard to distinguish actual defects from acceptable side-effects of parallelism
- For algorithm: problem with the use of the PETSc library

Newly added techniques for defect prevention

- For misuse of language features: overload the operator in an appropriate way. The interviewee recommended the use of Fortran 90's interfaces and presented concrete examples using the code from his project.
- For space decomposition: prepare a function that manipulates a global and local index, and test that part carefully
- For performance defects: when you need to organize some complicated communication patterns, it should be centralized so that it can be optimized independent of other modules

These techniques can be interpreted as concrete instances of modular programming or encapsulation in the term of software engineering. The interviewee also suggested the encapsulation at the level of developers, i.e., by restricting who can modify the particular parts of the code.

Qualitative evaluation and lessons learned

This was the first expert validation conducted, and we were encouraged by the positive reaction to the interview material as well as the concept of a defect experience base. The interviewee quickly responded to the examples presented and provided us with related examples as well as additional insights on the defect types. What we found especially valuable was advice on defect prevention techniques, which is hard to derive in the data collection process described in the previous chapter.

6.3.5 Interview 2

The second interview was conducted by phone. The interviewee was a professors who has taught a programming course using an emerging platform and programming model.

Added defect type

- Subtype of performance defect: ratio of computation and communication

Added defect examples

- For synchronization defect: Loss of messages when multiple processes sending to the same "mailbox" in the parallel programming language for Cell processor

6.3.6 Interview 3

The third interview was also conducted by phone. In this interview, three people were interviewed together. They work for a technology vendor.

Added defect examples

- For Erroneous use of language features: small corruption of the data caused by incompatible data type and data size

Newly added techniques for defect prevention

- For space decomposition defect: a standard approach to mitigate this issue is to explicitly define parameters related to the interactions (e.g., the rank of left-hand and right-hand process.) A technique which was useful in sequential programming, such as computing adjacent processes with a modulo operation can be rather harmful in parallel computing since it complicates boundary conditions. If they were writing a sequential program knowing that program was going to be parallelized, they would have written the sequential code differently in the first place.

Qualitative evaluation and lessons learned

The interviewees generally commented that while defect types look reasonable, the examples presented are far simpler than what they deal with in real projects. They often run into more fundamental issues such as hardware failures or compiler bugs.

The interviewees have decades of experience in this area, and during the interview, they were able to point out potential defects that can occur by just looking at a sequential example code. This suggests that they actually have a lot more knowledge than what we could extract. In this sense, the interview material we prepared did not "stimulate" them well enough.

Another observation is that interviewing multiple people at once is not a good idea, because individual views on defect patterns are so different that asking the questions to a group seems to inhibit free responses rather than validating and reinforcing each other.

6.3.7 Interview 4

The fourth interview was also conducted by phone. The interviewee was an experienced application developr in an HPC project, but as his job responsibility was not directly related to low-level parallelism, he did not have prior experience with an HPC language.

Added defect type

- Failure to understand other people's code (library code) due to lack of documentation

Added defect examples

- I/O bottleneck: the temporary code inserted for debugging caused a problem
- Memory management: the library code caused an out-of-memory problem. They had to change the OS settings to fix the issue until the library was rewritten.

6.3.8 Interview 5

The fifth interview was also conducted by phone.

Added defect type

- Subtype of performance defect: Amdahl's Law defect (failure to scale up to a large number of processors)
- Subtype of performance defect: failure to exploit locality
- Subtype of performance defect: too much synchronization

Newly added techniques for defect prevention

- For space decomposition: The parallel codes should be written correctly from the start to run correctly on one processor
- For Erroneous use of language features: Understand semantic subtleties and variations. Understand the subtleties of variations of the language features

Qualitative evaluation and lessons learned

- The interviewee managed to provide extensive feedback on the classification scheme taxonomy: renaming of defect types, restructuring of defect types and specific defect examples. On the other hand, he did not provide specific defect examples. It is understandable, since he is a professor rather than a project developer

- His criticism was the description of some defect types was too abstract from his point of view. He argued that if students are presented a pattern description that is too abstract, it becomes difficult for them to determine whether it is related to the problem in their code.

- He stressed the importance of performance defects, based on the argument that the whole point of HPC is to gain good performance.

- He strongly supported the idea of educating students with defect patterns. He used the content in his course.

6.3.9 Interview 6

The sixth interview was also conducted by phone. The interviewee was an experienced developer in an HPC project, but unlike the interviewee 4, the interviewee 6 was actively using MPI in a daily job.

Added defect examples

- Language usage: forgetting to add an extra argument (IERR). Compiler won't catch this
- Space decomposition: failure to distinguish operator functions which request an index with guard cells or without them
- Synchronization: excessive use of non-blocking communication degrading performance
- Performance: use unnecessary broadcast to distribute info to processors
- Memory management: out-of-memory error because in an adaptive mesh the particles were congregated in a fairly small area of space
- Algorithm: redundant guard cells when calling the Paramesh library
- Algorithm: problems with interpolation to find the values of guard cells
- Algorithm: initialization of the domain in parallel

Newly added techniques for defect prevention

- Language usage: refer to the good documentation in the reference manual, frequently asked questions (FAQs), and support pages.

Qualitative evaluation and lessons learned

- The interviewee has managed to provide many new defect examples for given defect types. She recognized and remembered defects very well on instance basis, but she did not suggest changes to defect types. This is direct opposite to the interview 5, where the interviewee suggested a lot of restructuring but no concrete examples. Different type of experts can contribute to different aspects of knowledge

- The interviewee was able to clearly tell which examples we presented were made by non-novices. She pointed out she has seen one particular defect (potential deadlock) was made by a fairly experienced peer developer. She was also able to point out she hasn't seen some defects in the real code for long time.

6.3.10 Discussions

Validity of the initial content

- The interviewees confirmed that the experience base contained real defects made by various types of HPC developers. Note that some "errors" in the description were found and corrected during the interviews, i.e., some code examples had a typo or other defect which was not related to the defect they were meant to illustrate. While this was an intended effect, it suggests that the defect types ad examples have been examined by the interviewees very carefully.

- The evaluation of the relevance of the defect types and examples varied across interviewees. As described below, their comments led to the evolving the existing content and obtaining the insights on the characteristics of interviewees.

Knowledge refinement

No defect types and examples we originally recorded were deleted, and there were several useful inputs. Table 6.1 below shows the number of "contributions" from each interviewee (or interviewees for the third interview). Note that the column I1, I2, etc. in the table represent the interviewee 1, 2, etc., respectively. The results show that all interviewees managed to add new items. Note that the content on the experience base was updated after the third interview, so the last three interviewees were able to look at the contributions made by the first three interviews.

Table 6.1: The number of contributions from interviewees

Interviewee	I1	I2	I3	I4	I5	I6
# of added new defect types	1	1		1	3	
# of added new examples	5	2	1	2		8
# of added new prevention techniques	3		1		2	1

Table 6.2: Updated classification scheme

Top-level defect type	Sub-type	Brief definition
Use of parallel language features	–	Erroneous use of parallel language features
Space decomposition	–	Incorrect mapping between the problem space and the problem memory space
Side-effect of parallelization (hidden serialization)	I/O hotspots	Serial constructs causing correctness and performance defects when accessed in parallel contexts
	Hidden serialization in library functions	
Synchronization	Deadlock	Incorrect/unnecessary synchronization
	Race	

Table 6.3: Updated classification scheme (continued)

Performance	Message scheduling	Performance defects in parallel contexts
	Load balancing	
	Failure to exploit locality	
	Excessive synchronization	
	Communication vs. computation ratio	
	Scalability problems	
	Memory hierarchy	
Memory management	**Memory allocation**	Inadequate memory management
	Memory cleanup	
Algorithm	–	Program logic not matching the intended purpose of the code
(Environment)	External libraries	Failure to understand/use code written by other people
	Compiler	

The updated classification scheme after these interviews is shown in Table 6.2 and 6.3. The newly added types are shown in bold fonts. There are two new top-level defect types that did not exist in the initial classification scheme: Memory Management and Environment. Both defect types are not unique to HPC applications, but the interviewees pointed out these tend to be more frequent and serious due to the differentiating factors of HPC. Memory management often becomes a problem because an HPC application often needs to be executed near the limit of the resources available, so a small unexpected behavior easily leads to a defect. The memory management type defects had been classified "Other", as they did not seem prevalent in the student data. The Environment defect type was not represented in the student data. For example, defects related to the misuse of external libraries especially is often difficult to avoid because they are so many serial and parallel libraries of which the quality, execution performance and the applicable semantics are not clearly documented. As in other software domains, there is general lack of documentation.

There are several sub-types of performance defects that have been added through the expert validations. Note that these new types are found to be similar to the taxonomy of performance bottlenecks developed by Hollingsworth [42], which include "ExcessiveSyncWaitingTime", "SyncRegionTooSmall", "ExcessiveIOBlockingTime", "TooManySmallIOOps", "CPUbound", "tooMuchPauseTime" and "WAY-tooMuchPauseTime".

Finally, Interviewing experts turned out to be an effective way for obtaining advice to prevent defects. This kind of knowledge was hard to derive from code analysis.

The above results confirm the basic feasibility and validity of the knowledge extraction process using the existing content on the experience base through a series of interviews.

Characterization of interviewees

While the number of the interviewees is too small to derive statistically significant results, the following observations can be made about the interviewees.

- HPC developers and professors who have advanced experience with the current HPC languages (interviewee 1, 5, and 6) were able to provide more additions (as shown in the Table above) than other interviewees. They recognized some defect types as recurring even in the code written by experienced developers. They were also most enthusiastic about the concept of collecting HPC defect patterns into the experience base.

- There were differences among these interviewees on what defects they think is important. Each user we talked to had very different views. For example, interviewee 5 put more emphasis on performance defects than other defect types. Interviewee 1 said "Side-effects of Parallelization" (in particular the examples with file I/O) is challenging, and he suggested a new defect type "Memory Management". On the other hand, interviewee 6 named "Space Decomposition", "Synchronization" and "Memory Management" as particularly challenging defect types, but "Side-effects of Parallelization" is not a frequently occurring defect type in their projects since they can mostly avoid it by using appropriate libraries. This means the experience base should cover various defect types which have different occurrences depending on the contexts.

- Interviewees 3 expressed they do not make the defect examples in the experience base, probably because they are "too experienced" to consider these defects as issues. The problems they often run into while debugging are related to the issues such as hardware failures or compiler bugs. While hardware are currently out of our scope, in the future it may be worth considering whether the experience base can also cover the type of defects they make.

- Interviewee 4 does not have prior experience with HPC languages, and naturally he has never seen defects that belong to the defect types such as "Erroneous Use of Language Features". He pointed out that the significance of defect examples may not be easy to understand until he actually deals with them. In the next Chapter, we will discuss the development of the education material for novice HPC developers and students.

Other observations

- In general, interviewees had many things to say about defects. Our approach for capturing their knowledge using the initial defect patterns was successful and the interviewees actively commented on the patterns we presented. However, it seems important to convince interviewees that our purpose is to obtain knowledge of defects rather than to evaluate the ability of an interviewee. As suggested from the results in Chapter 5, once they feel as if their ability is evaluated they tend to become more cautious and reluctant to admit they made any. Therefore, a question such as “What defects did you make?” does not work very effectively. In the interviews, we focused on the questions about defects themselves, and avoided directly asking who made them. When the interviewees quoted specific defects they have seen, they always talked about other people’s examples.

- As described above, interviewee’s perception of defects is quite diverse. Therefore, to capture as much information as possible, we should interview them individually rather than as a group. This may depend on the culture of the group. If they need to convince peer developers that they are a good developer, and if admitting some defects are recurring in their code is considered harmful, they inevitably inhibit each other.

- The interviewees 3 can be considered as both very experienced HPC developers and technology providers. Therefore, it may be interpreted as suggesting a gap between what technology providers think is important and the problems novice and experienced developers run into in real projects. This leads to the development of the recommendations for technology providers in the next Chapter.

6.3.11 Threats to validity

- Selection of the interviewees: We could not choose interviewees from random population. Selection was opportunistic but chosen from a mix of stakeholder types.

Chapter 7

Applying and Packaging Knowledge of Defect Patterns

The third activity of the problem solving approach is to apply the knowledge obtained to develop derivative artifacts. As described in the previous chapter, the knowledge stored in the experience base is structured with several levels of abstraction, so that users can use it and return feedback in various ways. In this chapter, we attempt to further package the knowledge to make it useful for specific purposes. The benefit of offering such artifacts is an easier access to the knowledge by those who are not necessarily motivated to use the experience base directly.

In particular, we develop two kinds of artifacts.

- **Educational material**: The first artifact is educational material that can be used for training novice HPC developers. A typical place where the material can be used is an HPC course offered at universities. It can be also used by a lead scientist in an HPC project who needs to train a new project member in short time. The base artifact is a set of presentation slides. The full slides take about an hour to go though. They can be used as a whole class lecture or they can be tailored (e.g., by selecting the slides for a particular pattern or replacing some examples) depending on the needs of the professor and the class goals. In addition, we have prepared a supplemental homework assignment to help in the understanding of the material. The assignment is to identify defects from the given source code containing seeded defects.

- **Technology recommendations**: The second artifact is a description of potential technological needs in high performance computing community. The purpose is to provide developers of tools and languages with useful information about what capabilities would be useful in preventing and detecting defects based on the empirical data we have collected. The main artifact is a recommendation document. If some of the recommendations are picked up and a solution is provided by a technology developer, there would certainly be benefits the HPC community. For a technology developer, the artifact can become a source of research ideas.

7.1 Educational material

A professor teaching a graduate/undergraduate course usually gives students one or more programming assignments to make sure they understood of the topics being taught, such as HPC architectures, parallel languages, parallel algorithms, and HPC programming itself. To produce a solution for the given assignment problem, students need to develop and debug the code, making their own tries and errors. While this is a necessary and important experience to understand difficulties in writing an HPC application and to learn how to deal with it, it is often time-consuming and inefficient. If a professor explicitly teaches recurring defect types before the assignments, students can anticipate the type of defects they may have to deal with, and use the knowledge to make debugging more efficiently and improve the quality of the final solution.

In a real HPC project, scientists cannot spend much time learning programming, since their main purpose is to accomplish some scientific task rather than becoming a good programmer. Learning basic usage and syntaxes of an HPC language is usually not very difficult. What is difficult is learning practical issues: how to avoid, detect or fix defects to create a program that works. Having educational material on defect patterns can reduce such learning efforts.

We have developed educational materials with the goal to fulfill these needs. Since we created the first version of the artifacts, it has been used in six graduate-level courses (including the two courses used for evaluation, which we describe later). The content has been incrementally improved according to the feedback from the professors. We describe the content of the material, and demonstrate its usefulness.

7.1.1 Content of the artifacts

The base artifact is a presentation package consisting of a set of slides. The content is organized as follows. The actual slides are presented in Appendix C.

- Introduction
- Problem and sequential solution
- Description of defect type 1 (Erroneous use of parallel language features)
- Examples of defects that belong to defect type 1
- Description of defect type 2 (Space Decomposition)

- Examples of defects that belong to defect type 2
- ...
- Description of defect type 5 (Performance defect)
- Examples of defects that belong to defect type 5
- Closing

Introduction

The introductory part consists of two slides which explain the background of the material: the difficulty of debugging and testing parallel code, the concept of developing an experience base to collect and share common defect types, and the differentiating factors of high performance computing that can affect the characteristics of defects. This part is intended to motivate students.

Problem and sequential solution

In early versions of the slides, defect types were explained using the real examples taken from various assignment problems. This was inconvenient because the problems being solved must be explained before explaining the defect examples.

Therefore, we defined a simple problem which can be explained easily. Figure 7.1 illustrates the problem.

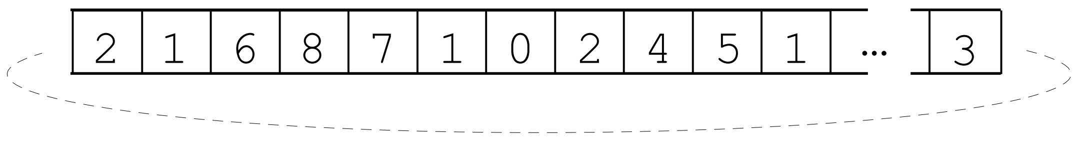


Figure 7.1: The problem used in the educational material

The problem is a discrete time step simulation of N cells, each of which holds an integer from 0 to 9. In Figure 7.1, the initial cell values are cell[0]=2, cell[1]=1, ..., and cell[N-1]=3. In each step, cells are updated using the values of neighboring cells:

$$\text{cellnext}[x] = (\text{cell}[x-1] + \text{cell}[x+1]) \mod 10 \tag{7.1}$$

For example, the cell values from the initial state are cellnext[0]=(3+1), cellnext[1]=(2+6), etc. The last cell is assumed to be adjacent to the first cell. This step is repeated for the number of steps specified by an input parameter, and the final cell values are returned as output.

Following the problem description, a straightforward sequential implementation in C is presented. This is used as the basis for explaining the defect examples. All the examples are modified to fit to a scenario for developing a parallel solution of this problem using MPI. This way, only one problem needs to be explained at the beginning.

Description of defect type

Each "description of defect type" is directly derived from a top-level defect type in the experience base. To make the description concise, each defect type is summarized in one-page slide.

The description has been improved based on the refinement process described in the previous chapter.

Examples

Examples are chosen from the defect instances in the experience base, and modified to fit the process of writing a parallel code of the sample problem. Each example corresponds to a particular phase of implementation:

- Adding “housekeeping” MPI constructs (header statements, initialization and finalization functions.)
- Decomposing the problem space into processes and allocating an appropriate memory space
- Parallelizing data initialization
- Implementing communications between processes

Closing

In the summary slides the content refers to the experience base website so that students who need more detailed information on each defect type and examples can visit the experience base system.

7.1.2 Homework

We also provide material that can be used as a homework assignment. A typical scenario would be to give it out to students right after the lecture on the defect types is presented. In this homework, the students are asked to inspect the given source code and identify defects. The source code is a (faulty) solution to the approximation of the number pi by a Monte Carlo simulation, and several defects are embedded by the author. The students are asked to describe why each defect they identified is considered a defect, which defect type it belongs to, and how it can be fixed. The actual content of the homework is shown in Appendix C.

The purpose of the homework is to assist in understanding the lecture on the defect types. To answer the homework, the students are naturally made to review the presentation material. The problem itself is kept simple enough to be solvable in short time.

7.1.3 Evaluating usefulness of the educational material

We have evaluated the usefulness of the educational material in two ways. The first evaluation was conducted by analyzing the source code written after the students listened to the lecture and compared the results with the base data we developed in Chapter 5. The second evaluation was conducted by interviewing students who actually the course and sat through the lecture.

Code analysis

The quantitative evaluation of the usefulness was performed by testing the material in the actual class and measuring whether there was any effect on the performance of students in the subsequent programming assignments. We conducted a study in two graduate courses. The data collection in these courses is the same as other "regular" studies, but the students were given a lecture and a homework about the defect patterns. To measure the effect, the timing of the lecture and the homework was set after the lecture on basic MPI programming and before the assignment to be measured.

The GQM goal is described as follows.

- Analyze the source code data to evaluate it with respect to defect occurrences and time to fix compared to the base data in the context of classroom studies using the defect lecture material.

An ideal experiment would be to randomly split the students into two groups and give the lecture to only one of them. However, since we conduct an evaluation in real classes, there is a limit on what we can control to make the mode of evaluation close to ideal. In particular, we are not allowed to treat students in the same class unfairly. Therefore, we used the results from other courses taking the same assignment problem as reference data. These courses used exactly the same problem description and sample input data, so the context factors are as homogeneous as possible. Figure 7.2 shows the design of the experiment. Note that we do not use the term "control group", as we use data from multiple courses without an explicit control over the configuration of the lecture and the selection of students.

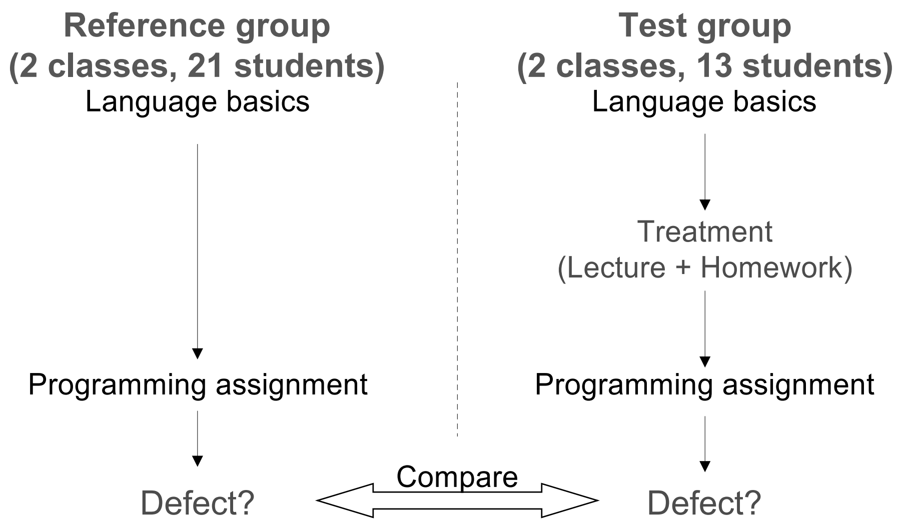


Figure 7.2: Design of the evaluation of the educational materials

In particular, we chose the Game of Life problem. Before the evaluation, we had the data for 21 students from two “ordinary” courses which used exactly the same problem description. These data are used as reference data. In the evaluation, we asked two more courses to use the same problem description, but after the lecture on the defect patterns and the homework is provided.

The source code snapshots were analyzed to identify defects as we did in Chapter 5. To perform quantitative comparison, all versions are inspected rather than skipping the versions using the heuristics.

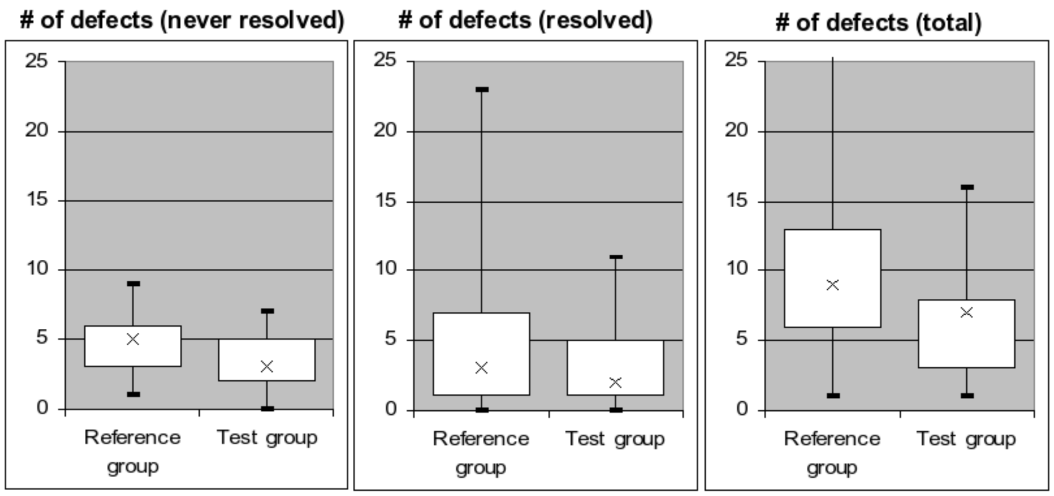


Figure 7.3: Comparison of the number of defects identified for each student

Figure 7.3 presents the boxplots for the number of defects identified for each student. The statistical analysis indicates that the test group made fewer total defects at a statistically significant level (P = 0.048). It is a 34% improvement on average. Furthermore, the test group left fewer defects in the code at a statistically significant level (P = 0.029). It is a 31% improvement on average, and it suggests that students with the knowledge of defect patterns can create a higher-quality product. The test group found 42.3 % of defects on average while the reference group found 39.8 % of the defects on average.

To further interpret the results, we investigated whether particular defects were made by each student. The defects chosen are the ones most commonly observed in this problem, and each of them represents the top-level defect type to which it belongs. These tend to remain unfixed in the final version. Furthermore, except for the last one ("all processes holding the entire memory space"), the same defects were explicitly covered by the lecture.

- Missing MPI_Finalize on some execution path (Use of parallel language feature)
- Problem space may not be equally divisible (Space decomposition)
- File access by multiple processes (Side-effect of parallelization)
- Potential deadlock (Synchronization)
- Message scheduling bottleneck (Performance)
- All processes holding the entire memory space (Memory management)

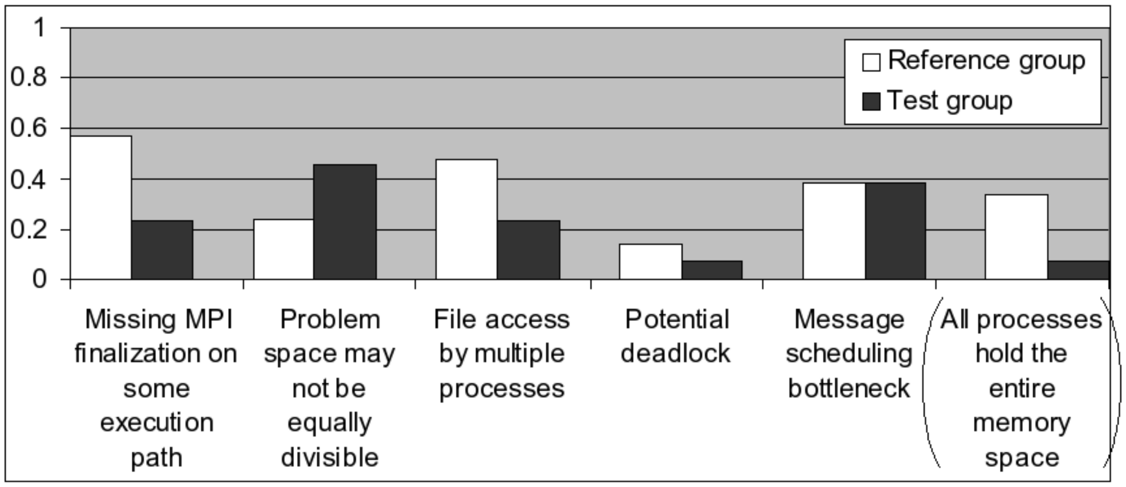


Figure 7.4: Comparison of the percentage of students who made specific defects

The result is shown in Figure 7.4. The defect rates for the test group seem to have improved except for the "Problem space may not be equally divisible" and "Message scheduling bottleneck". Out of these six specific defects, 4 have decreased by half. On the other hand, the defect type ("message scheduling bottleneck") had about the same occurrence, and the defect type ("problem space may not be equally visible") has nearly doubled. One explanation is this defect is harder to avoid even if they know its existence. Another explanation is that the current lecture material does not explain space decomposition defects and performance defects. Further studies are required to investigate if either explanation is plausible, or whether there are other reasons.

Finally, Figure 7.5 shows the comparison of the effort measure (estimated time to fix defects) for both groups. The test group spent 0.66 hours (median) while reference group spent 2.93 hours (median) to fix defects.

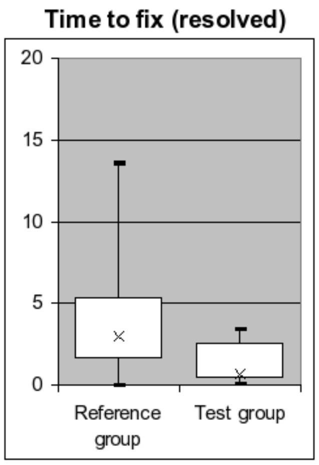


Figure 7.5: Comparison of average time to fix

There are several threats to validity of this study.

- Since the reference and test data were obtained from different classes, it is possible that the effect was caused by some factors other than the defect lecture. It is desirable that the study be repeated with more subjects. (internal validity)

- It is possible that just the extra time spent on an additional lecture was responsible for the improvement. (internal validity)
- It is possible that fewer defects were identified from the test group simply because data analysis is not as complete as that of the reference group. Validation of the analysis results by other analyst is desirable. Note that the reliability study in Chapter 5 was conducted using the data from one student in the test group and no defects identified by the analyst were missed by our analysis. (internal validity)
- The evaluation was performed with a particular assignment problem (Game of Life). The result may not be extended to other problem sets. (external validity)

Interview

We interviewed as a group three students who attended the lecture. We asked whether the lecture material on the defect patterns was clear, whether the material was useful, and whether there was any information that was not provided but would have been useful.

- The interviewees all agreed the lecture material was clear enough to understand the defect types and the examples. One interviewee commented the lecture was also useful as an introduction of MPI programming, because other lectures of the course was more focused on the programming model rather than programming itself.

- One interviewee commented that examples were easier to understand than the description of defect types. Another interviewee agreed she had to go through the examples to understand the definition of the defect types. The third interviewee said he was able to understand the definition from the start, but the examples were certainly useful.

- One interviewee said she was able to be careful not to make the same defects described in the slides. The lecture helped the interviewee avoid a lot of mistakes that she could have done without the lecture. For example, when she ran into a synchronization problem, she was able to quickly capture the defect because she had had an idea of what could go wrong from the lecture and identify the cause of the problem: the program had a deadlock, and the cause was the order of send/recv was not in the correct order. It was not exactly the same defect example, but the lecture material helped her think about what to look for and what are possible problems. Other interviewees agreed the lecture helped them write a program more carefully and avoid some defects because they knew beforehand what some defects look like, so the lecture was useful for both programming and debugging.

- One interviewee commented she wanted to learn more about MPI programming before the defect lecture. The other interviewees commented they had an adequate preparation to understand the defect lecture material.

- They actually encountered the defects explained the lecture, and the material helped them avoid these defects.

- The interviewees took about two hours to finish the homework. They commented that they looked through the lecture material while working on the homework, so it gave them a chance to review the material carefully. One interviewee commented that some defects were very easy to find since they were exactly the same defects that were explained in the lecture, and other defects were harder to find. They said that having the answers to the homework returned was also useful.

- They commented that they think the classification scheme was reasonably defined. One particular defect made by one interviewee was how to use scatter/gather with a 2-dimensional array. It was not in the examples given in the lecture, but it can be classified as the "Erroneous use of parallel language features" type. Richer set of examples for each of the defect type might have been even helpful.

- One interviewee had a defect which didn't fit into the classification scheme- it was a sequential defect- a typo between "==" and "=" which led the program to deadlock, and the interviewee looked for synchronization defects.

In summary, there appears that the lecture had a positive influence on the thinking and the activities of the students who attended the defects lecture and did the assignment.

7.2 Recommendations for technology developers

Good technology can greatly reduce debugging effort. It would be beneficial if we can provide potential developers of tools and languages with some qualitative information on demands of HPC developers, based upon our observations, interviews and analysis results. A technologist is expected to use the information to gain insights on technology demands from empirical point of view.

The survey indicates there are actually various technologies available for HPC development [**?**, 1]. However, people often complain about the general lack of tools they can use. This seems to imply there are discrepancies between the functionalities provided by existing tools and what developers really want. One clue from the results of the previous chapter is the technology providers are too experienced and consider many defects as "too simple" to deal with. They try to provide a solution to much more advanced issues, but they are not necessarily what developers consider as dominating problems. Therefore, providing insights on technology demands from empirical studies can be beneficial for both HPC developers and technology developers who are looking for research ideas.

The content of the recommendations in this section is preliminary. While it is based on the insights from the data we collected, and the comments gathered from the interviews, it has not been verified by domain experts nor has it been presented to technology developers to obtain feedback. Since the focus is on describing demands rather than assessing the feasibility of individual recommendations, not all items may have an immediate solution. The validity of the recommendations should be further assessed as the content is updated.

7.2.1 Type of tools

One difficulty in debugging an HPC application is that getting machine resources for "debugging runs" is not easy. To debug a parallel HPC program, developers often need to run the code on a large number of processors. In many HPC machines, a job with many processes has to wait time for long time in a job queue. Having to waiting for several hours just to run a debugging run is not a very productive environment for debugging.

If defects can be identified without actually executing the code, that is a particularly attractive approach for HPC development. There are not many static analysis tools applicable to HPC applications. "Traditional" languages such as FORTRAN 77, Fortran 90, or C, with MPI are not a popular target for researchers programming language and static analysis.

7.2.2 Defect types which are desirable to be detectable

Erroneous use of parallel language features

Is it possible to check common language misuse?

- Illegal parameters (compiler passes but illegal)
- Function misuse

Space decomposition

Space decomposition defects are common in MPI programming. While much effort has been spent in the area of parallel compilers to derive appropriate decomposition [74], a general solution to the detection of decomposition defects seems to be difficult to implement. Hence, it is possible that some specific type of decomposition defects might still be detectable under certain conditions. For example, when a sequential code is available, it would be interesting to explore whether it is possible to check if the parallel code covers the entire problem space, by examining:

- Is it possible to check if the memory buffer is supposed to contain guard cells?
- Is it possible to check whether the code refers to memory using a local index or a global index?
- Is it possible to check if the array origin declaration is consistent with use?
- Is it possible to check whether indexing in the serial code would be difficult to parallelize?

Performance

Performance defects are often checked with debuggers and profiling tools, which keep track of how the code is executed at runtime. An interesting challenge would be to check performance bottlenecks statically. Since there are so many factors that can contribute to performance bottlenecks, it would be difficult to estimate how fast the code runs in a general case. Again, detecting some specific type of performance defects would be beneficial. Such examples include:

- Is it possible to detect load balancing defects using a static approach?
- Is it possible to detect communication bottlenecks using a static approach?

7.2.3 Some observations

- **Demands**: Development tools and languages used in HPC development are different from those used in other software domains. Although HPC applications are developed, debugged and executed in the most advanced computer environments, the tools being used tend to be “behind” other domains in some sense. For example, few modern GUI-based tools for developing and debugging applications are used by HPC developers, while such tools are commonly available in other domains nowadays. Many developers rely on primitive debugging methods such as inserting print statements, and they often complain about the general lack of useful tools [39].
- **Conservativeness**: On the other hand, many HPC developers also say they are not going to and/or are not able to change the current development practice unless they are convinced the change will benefit them. While this largely depends on the discipline of individual developers, they tend to be more conservative than developers in other domains. One possible reason is they have to put emphasis on the availability of technology across platforms, and the continued availability of these tools over time. They are afraid to depend on a certain technology that may not be available in all the platforms they

need to access now and in a future. This is especially true for a programming language, to which the application is strongly bound. Debugging tools might be easier be adopted as they are easier to switch if they become unavailable. However, many developers may be still hesitant unless the tool is mature and stable enough for long term use.

- **Legacy assets**: An HPC application often needs to work with an existing code: external library, or their own old code. Therefore, the technology to be adopted is bound to these constraints. Compatibility with the existing assets is very important for adoption. Transition to a new technology, if it ever happens, is a gradual transition. So new technologies must be made to co-exist with old technologies.

Chapter 8

Conclusion

8.1 Summary of contributions

The contributions of this dissertation can be divided into two categories: those that advance knowledge of software defects in high performance computing, and those that advance research methodology in empirical software engineering.

8.1.1 Methodological contributions

Iterative method for building knowledge of defects

The main methodological contribution of the dissertation is the proposal for a methodology for building patterns of domain-specific defects from empirical data. While many practitioners recognize the usefulness of defect patterns, and there are patterns proposed for various languages, properties of software, and application domains, there seems no established methodology for developing defect patterns in new contexts. This is important because unlike design patterns, defect patterns tend to specific to the contexts of software domains.

Approach for bottom-up knowledge building using the analysis of low-level data

One contribution of this work is an approach of software engineering research when the available data is not in a format convenient for analysis and processing. In a conventional approach, researchers usually stayed with "superficial" analysis of the data in such a situation. In this dissertation, we showed that high-level insights can be obtained by manually analyzing source file data and building necessary knowledge bottom up. The analysis is deeper and more specific.

Heuristics for analyzing code history data

Another contribution of this work is a set of heuristics for analyzing code history data efficiently and reliably. They form a basis that are effective and they can be further evolved.

Infrastructure for defect data analysis

Another contribution is the tools developed for collecting, organizing and analyzing data. Again, they form a base set that can evolve.

Reactive knowledge refinement through structured interviews

Another contribution is an approach for extracting the tacit knowledge of domain experts through a variation of structured interviews. The key idea is that experts can provide their knowledge more easily when they are presented with concrete base materials they can respond to, rather than just being asked what they know about the topic. In this dissertation, we used the results of code analysis to develop initial defect patterns and packaged them into interview material so that the interviewees could understand the contexts of the problem.

The combination of bottom-up knowledge building and top-down knowledge refinement is essential to the successful construction of defect patterns. Although the initial defect patterns built from the low-level analysis alone often do not cover all contexts of interest, their existence plays a critical role in enabling validation and addition of further knowledge by domain experts.

SE education for debugging

Teaching defect patterns is not a common practice in education. A broader implication of this work is that it may be worth considering the use of this kind of educational materials for a generic programming course.

8.1.2 Domain-related contributions

Identifying patterns of software defects in HPC

One of the main contributions of this work is the construction of defect patterns in HPC software. While practitioners in the HPC community possess a great deal of knowledge on defects recurring in HPC applications, such knowledge has mostly been hidden in the brain of individual practitioners. We have identified defect patterns from empirical data, developed a classification scheme and iteratively refined the patterns by getting feedback from experts.

Constructing HPCBugBase

The main product out of this work is an experience base system that stores and shares the defect patterns constructed above. The system is available publicly on the web and the content is actively updated and maintained. The information in the experience base is described at several levels of abstraction so that users with different interests can access information as it is needed. The system also provides an interface to accept feedback from users. This contribution provides a starting point for broader acceptance of the defect patterns into the HPC community.

Development of educational material

Another product is an educational material that can be used to teach the characteristics of common defect types to novice HPC developers and students. Being able to learn concrete examples of recurring defects for a new language together with advice for detecting and/or preventing them seems beneficial.

Development of heuristics for code analysis

Another product is a set of heuristics which was developed to mitigate the labor-intensiveness of a reading-based approach for defect analysis. A preliminary reliability study suggests that the heuristics can help analysts perform the analysis consistently.

Development of base data for further research

All the analysis results are stored into the database. This data, together with other data we collected, represents the current state of practice in HPC development. Further research is possible by reusing this data.

8.2 Review of research questions

8.2.1 Questions about the existence of defect patterns

- *RQ1-1: What are domain specific defects in HPC?*: We have identified a class of distinctive defects which seem to be closely related to the characteristics of software development in HPC. One defect type is caused by the parallel features of HPC languages; One defect type is related to the decomposition of the problem space; One defect type is related to concurrency and synchronization; One defect type is related to side effects of parallelization; One defect type is related to various performance issues; One defect type is related to memory management; One defect type is related to algorithms; Finally, one defect type is related to external libraries and compilers. Although these do not necessarily appear exclusively in HPC applications, they are considered particularly important in the HPC domain in terms of frequency, impact and difficulty in debugging.
- *RQ1-2: Can we identify defect patterns (causes, symptoms, potential cures and preventions, and examples)?*: Yes, the knowledge of defect patterns has been

collected and refined through a series of studies using the iterative methodology.

- *RQ1-3: Can we define a set of templates to describe information about defects at different levels of abstraction?*: Yes, we have defined templates for defect types, specific defects and defect instances.

8.2.2 Questions about the collection and description of knowledge

- *RQ2-1: Can we build heuristics to detect defects in the code that can be used in a reliable way by others?*: We have developed a set of heuristics to help identify defects from the change history. The results of the reliability study indicates the heuristics can be used by other analyst to repeat the code analysis and the analysis results match to some extent, although there are further rooms for improvement to increase the completeness of the analysis.

- *RQ2-2: Can we classify defects in a way that is clear to experts and allows them to add information?*: Yes, the initial defect types built from the results of data analysis have been verified by experts and new information has been added.

- *RQ2-3: Is it possible to automate the detection of some classes of HPC defects?*: Not in this work. We have built a tool set for assisting code analysis but our method is still largely manual. We have created a recommendation for technology developers so that they can develop an automated defect detection tool for some defects.

8.2.3 Questions about the refinement of knowledge

- *RQ3-1: Can we extract new knowledge by presenting existing knowledge in a structured way?*: Yes, through a series of structured interviews we extracted new knowledge by presenting existing knowledge

- *RQ3-2: Can we build and evolve a defect experience base with usable information?*: Yes, we have developed a Wiki-based public defect experience base system which stores and shares defect patterns at various levels of abstraction.

8.2.4 Questions about the application of knowledge

- *RQ4-1: Can teaching novices about defect patterns reduce the number of defects made?*: Yes, the result of the experiment indicates teaching defect patterns was effective in reducing the occurrence of total defects and improving the quality of the final version.

- *RQ4-2: Can we provide recommendations to researchers of future defect detection tools? Can we develop our own data analysis tool based on the recommendations?* We have provided some recommendations. We have not developed our own tools.

8.3 Lessons learned

8.3.1 Enthusiasm

We have observed that HPC stakeholders generally agree on the importance of reducing debugging cost and the possibility of mitigating the problem through the construction of an experience base. The professors who have used the educational material said the lecture on defects was beneficial for their class. There are also other professors who volunteered to do the lecture on defects using our material in a future. They are all potential users of the defect experience base.

8.3.2 Subtleties of questions related to defects

Extracting knowledge of defects from developers involves certain subtleties that do not exist when asking for other kinds of information, because some people seem to think that admitting they have made some defects is something embarrassing. When we asked people to provide us with information on any defects, they all talked about defects in the code written by others rather than themselves. An attempt to ask developers to provide information on the defects they made themselves could face difficulty.

8.3.3 Diversity of HPC stakeholders

Through the interviews we conducted, we observed that the views on what defect types are more frequent than others and how they approach the problems seem to be diverse.

The views seem to depend on their background and the area of expertise, as well as the characteristics of the tasks they do.

8.4 Future work

8.4.1 Adoption of the defect experience base

In this dissertation, we developed an experience base system that has basic functionality. While we have obtained positive feedback from the experts we have interviewed with, the use of the experience base by general public is scarce. To make the experience base successful, we must attract many people in the HPC community to visit and use it. An ultimate goal is to build a group in the HPC community who are enthusiastic about building and maintaining community knowledge about defect patterns, and the experience base should serve as a portal for them. Toward this goal, the implementation of the experience base should be improved as needed.

There are some possible improvements to the system; A better mechanism for collecting community feedback is desired, as the input and maintenance of the content is currently heavily dependent on the author. A better mechanism for searching content from symptoms and/or error messages is also desirable since currently the queries just look for keywords in the content of each entry.

8.4.2 Expanding the research scope

There are several possibilities for further research reusing the results of defect analysis.

Cross-variable analysis

One direction of future research is to conduct analysis across other variables related to developer productivity. For example, if the execution performance is measured for all compiling versions, it is possible to correlate it with the defect data to evaluate the impact of performance defects.

This is expected to lead to deeper understanding of broader productivity issues.

Evaluating new technologies

Another direction is to repeat the empirical study for emerging programming models being developed in the HPCS project, such as X10 and Chapel. These new technologies are expected to provide better productivity compared to conventional languages such as MPI and C. Using the results of this work as a basis, it is now possible to evaluate how well these new technologies really contribute to the improvement of productivity. Occurrence and time to fix for each defect type are useful measure for comparison.

Direct analysis of large projects

Yet another direction is to broaden the scope of the reading-based data analysis to full-scale projects. Directly analyzing real HPC applications can increase the credibility of the defect patterns. Collecting the code history data specifically for the study can be challenging. There are confidentiality issues, projects are competing with each other. Data from public/internal source code repositories are easier to obtain as many HPC projects use some code management system. However, the granularity of code commit is much less frequent than per-compile snapshots used in this work. Many developers test and debug the code before committing it to the repository, so analyzing code history from the repository doesn't provide the information on debugging at the level of individual developers. Of course not all defects are fixed before the code is committed. It is possible to observe defects that are hard to find and thus appear in the repository. Another challenge is to understand the code, as it is likely to be based on much more complex mathematical background. An analyst may need help from project developers to understand the meaning of the code.

Assessment of the criticality of defects

Throughout this dissertation, the measure of importance of defects was based on frequency and duration: defects are worth describing as patterns if they appear in HPC applications repeatedly, and their impact is large if they stay in the code for long time (i.e., the defect is hard to find and fix). To further extend the measure, it is possible to introduce other measures, such as whether the defect actually causes a failure or performance degradation in specific execution environments. One approach is to actually run the code in many environments, measuring execution time and checking the correctness of output. Another approach is to examine the characteristics of the identified defects and generate test cases that would make these defects observable in test runs.

Simulation models

Another direction is to build a model of developer workflow based on the defect data. The model simulates the development activities using the parameters such as the occurrence of defects and the time to fix them and derives possible bottlenecks through simulations.

8.4.3 Improving tool support for data analysis

To improve the efficiency of data analysis, further refinement of the tools is desirable. Our group is now developing a new tool which can visually display a series of source files with the changed locations. As we develop various tools, integration becomes important for efficiency.

Chapter A

Description of Problems in Classroom Studies

In this chapter, we provide the descriptions of the problems which the HPC applications we analyzed are solving. The problems range from simple ones used in classroom studies to real academic projects. Some problems were used in multiple classroom studies with slightly different settings, such as whether a sequential implementation was provided at the beginning, what hints were given, or what was grading criteria.

A.1 Problem: Buffon-Laplace

The Buffon-Laplace problem is a kind of random simulation. Imagine that a needle of length l is dropped onto a floor with a grid of equally spaced parallel lines distances a and b apart, where l is less then a and b. The probabilty that the needle will land on at least one line is given by:

$$p = \frac{2l(a+b) - l^2}{\pi ab}$$

Thus, by running a Monte-Carlo simulation of needle drops, we can estimate π.

A.2 Problem: Dense matrix-vector multiply

The problem is to implement a parallel program which computes the the product of a given dense matrix and a dense vector.

A.3 Problem: Game of Life

The Game of Life problem is a well-known cellular automaton simulation named by John Conway. In its basic form, which was employed in all of the studies we conducted, it is a simulation on a two-dimensional grid space. Each cell in the grid has either the "alive" or the "dead" status. In every discrete time step (or "generation"), the cell status is updated based on the number of live neighboring cells, with the following rules.

- Live cells with 2 or 3 live neighbors will stay alive in the next step
- Dead cells with exactly 3 neighbors will become alive in the next step
- Cells with less than 2 live neighbors or more than 3 live neighbors will die in the next step

Intuitively, cells with too few neighbors die because of loneliness, and those with two many neighbors die because of over-population.

This problem has been found interesting because these simple rules can generate many "interesting" patterns.

A.4 Sharks and fishes

The sharks and fishes problem is similar to Game of Life, but has more complex cell states and update rules. Each grid cell has either a fish or a shark, or it is empty. A fish and a shark has an age, which is incremented at each step. The update rules are as follows:

- If the cell has a fish:
  - It tries to move a neighboring (up, down, left or right) empty cell or stays in the current cell if no neighboring cell is empty.
  - If a fish can move, and if it reaches a breeding age, it leaves behind a fish of age 0 ("breeds") and its age is reset to 0.
- If the current state is shark:
  - It "eats" a fish (kill the fish and move to that cell) if there is a fish in any of the neighboring cells. A shark dies if it reaches a starvation age.
  - If it reaches to a breeding age, it breeds according to the same rule as a fish.

A.5 Problem: Grid of resistors

The problem is to compute the voltages and the effective resistance of a $2n+1$ by $2n+2$ grid of 1 ohm resistors if a battery is connected to the two center points. This is a discrete version of finding the lines of force using iron filings for a magnet. A successive overrelaxation (SOR) method with red-black ordering should be used to solve the problem.

A.6 Problem: Laplace's equation

The problem is to implement a solution to a simple discretization of Laplace's equation:

$$\Delta T = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

on a 2-D rectangular grid.

A.7 Problem: Quantom dynamics

The problem is to parallelize a simulation of dynamics of a particle, such as an electron, which follows the law of quantum mechanics.

A.8 Problem: Sparse matrix-vector multiply

The problem is to implement a algorithm in parallel to multiply a sparse matrix with a dense vector.

A.9 Problem: Sparse conjugate Gradient

The problem is to solve a symmetric positive definite system of linear equations $A\vec{x} = \vec{b}$ by the conjugate gradient method. It is a iterative approach, and it uses a function for sparse matrix-vector multiply as a sub-routine.

Each step of the iteration refines an approximated solution $\vec{x}_k$. Set initial parameter values to $\vec{x}_0 = 0$ and $\vec{r}_0 = \vec{d}_0 = \vec{b}$. The parameters are updated with the following equations.

$$\begin{aligned}
\alpha_k &= \frac{\vec{r}_{k-1}^T \vec{r}_{k-1}}{\vec{d}_{k-1}^T A \vec{d}_{k-1}} \\
\vec{x}_k &= \vec{x}_{k-1} + \alpha_k \vec{d}_{k-1} \\
\vec{r}_k &= \vec{r}_{k-1} - \alpha_k A \vec{d}_{k-1} \\
\beta_k &= \frac{\vec{r}_k^T \vec{r}_k}{\vec{r}_{k-1}^T \vec{r}_{k-1}} \\
\vec{d}_k &= \vec{r}_k + \beta_k \vec{d}_{k-1}
\end{aligned}$$

These involve one matrix-vector multiplication and two vector dot products per iteration.

In a parallel implementation, a matrix and a vector are decomposed by rows. a matrix-vector product is computed by (1) broadcasting the vector and (2) computing partial rows of the product. A more optimized version can determine which vector elements are needed by each process and only send to them instead of broadcasting the entire vector.

A.10 Problem: Matix power via prefix

The problem is implement a parallel solution for a given $n \times n$ matrix A to compute all its powers A, A^2, ..., up to A^l.

A.11 Problem: Sorting

The problem is to sort a set of N elements.

A.12 Problem: LU decomposition

The problem is to implement a parallel solution to LU decomposition, i.e., factor a square matrix into two matrices L and U, where L is lower triangular with ones on its diagonal, and U is upper diagonal.

A.13 Problem: Shallow water model

The problem is to parallelize a simulation of the shallow water model, which is part of the SPEC benchmark suite.

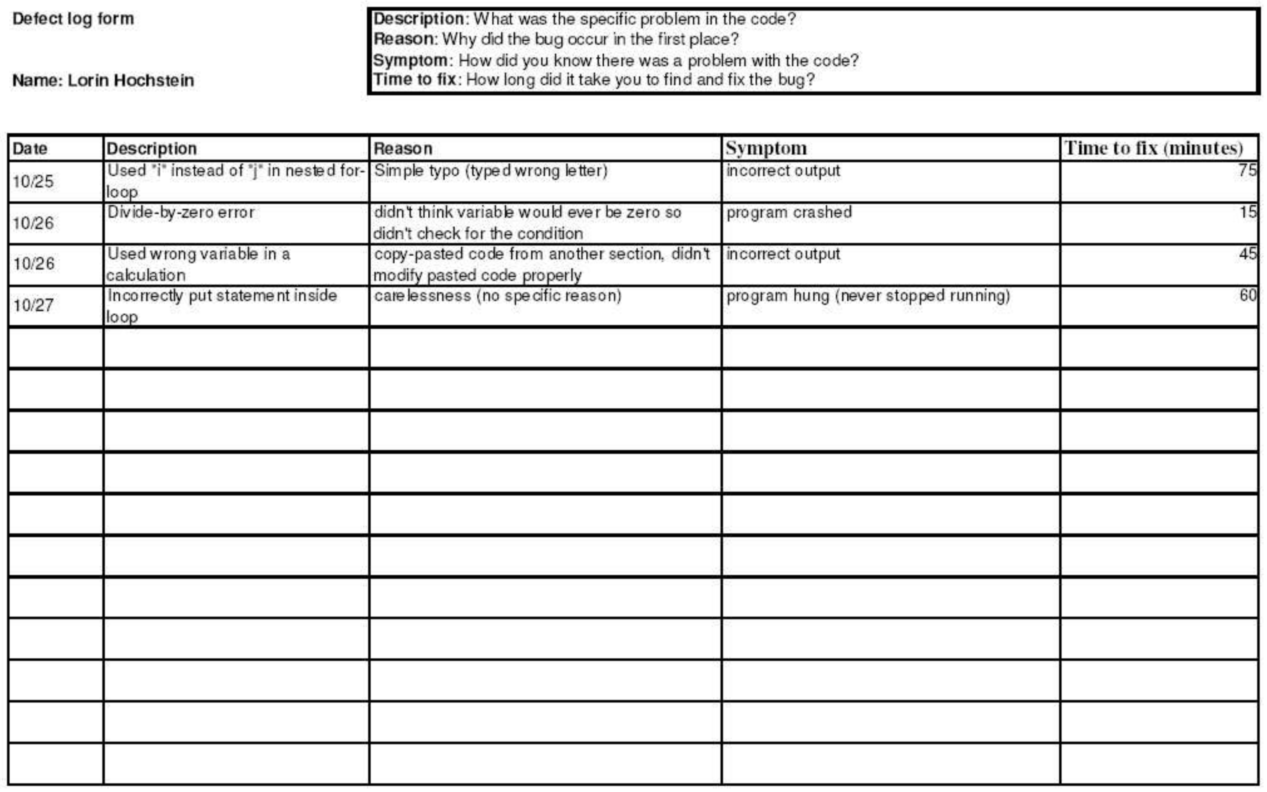
A.14 Problem: Randomized selection

The problem is for a given a set of N integer elements and an integer k ($1 \leq k \leq N$), find the k-th smallest element in the set in expected linear time.

Chapter B

Self-reported Defect Forms

B.1 Form 1



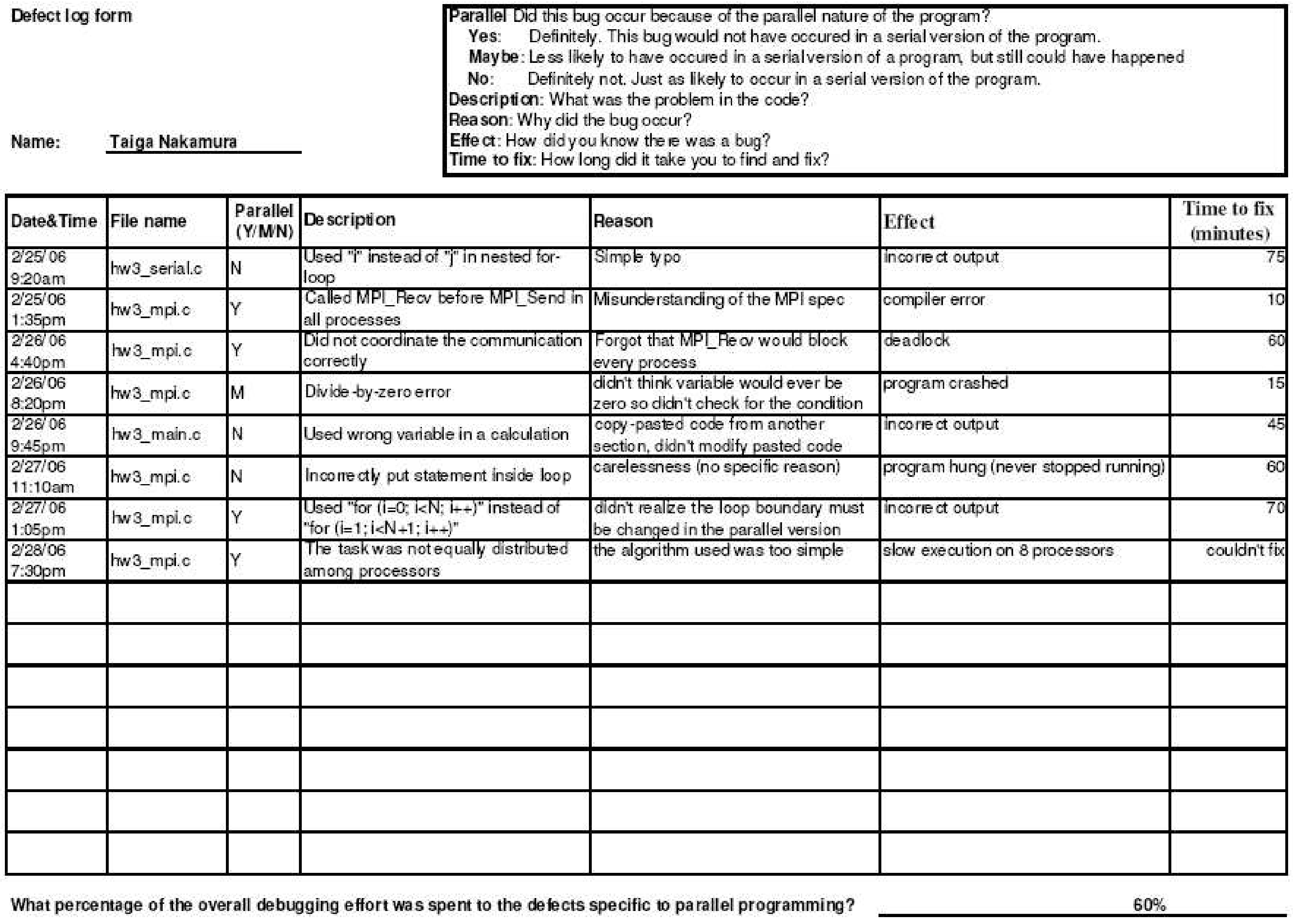
Defect log form

Name: Lorin Hochstein

Description: What was the specific problem in the code?
Reason: Why did the bug occur in the first place?
Symptom: How did you know there was a problem with the code?
Time to fix: How long did it take you to find and fix the bug?

Date	Description	Reason	Symptom	Time to fix (minutes)
10/25	Used "i" instead of "j" in nested for-loop	Simple typo (typed wrong letter)	incorrect output	75
10/26	Divide-by-zero error	didn't think variable would ever be zero so didn't check for the condition	program crashed	15
10/26	Used wrong variable in a calculation	copy-pasted code from another section, didn't modify pasted code properly	incorrect output	45
10/27	Incorrectly put statement inside loop	carelessness (no specific reason)	program hung (never stopped running)	60

B.2 Form 2



Defect log form

Name: Taiga Nakamura

Parallel Did this bug occur because of the parallel nature of the program?
Yes: Definitely. This bug would not have occured in a serial version of the program.
Maybe: Less likely to have occured in a serial version of a program, but still could have happened
No: Definitely not. Just as likely to occur in a serial version of the program.
Description: What was the problem in the code?
Reason: Why did the bug occur?
Effect: How did you know there was a bug?
Time to fix: How long did it take you to find and fix?

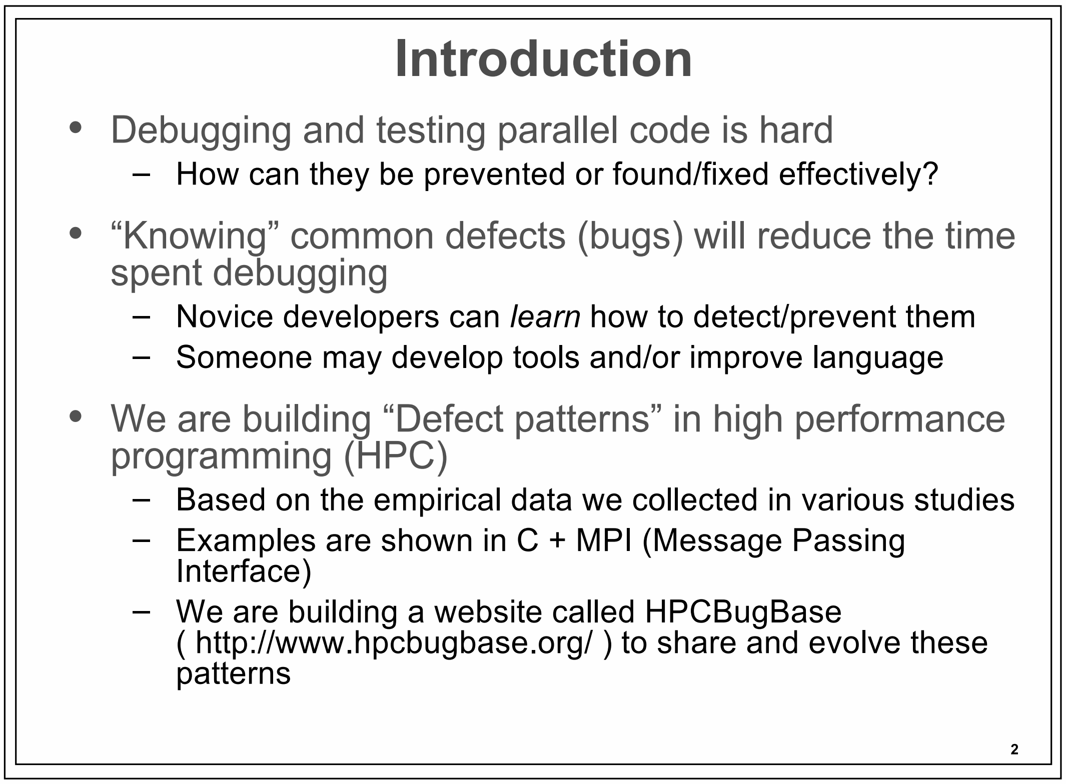
Date&Time	File name	Parallel (Y/M/N)	Description	Reason	Effect	Time to fix (minutes)
2/25/06 9:20am	hw3_serial.c	N	Used "i" instead of "j" in nested for-loop	Simple typo	incorrect output	75
2/25/06 1:35pm	hw3_mpi.c	Y	Called MPI_Recv before MPI_Send in all processes	Misunderstanding of the MPI spec	compiler error	10
2/26/06 4:40pm	hw3_mpi.c	Y	Did not coordinate the communication correctly	Forgot that MPI_Recv would block every process	deadlock	60
2/26/06 8:20pm	hw3_mpi.c	M	Divide-by-zero error	didn't think variable would ever be zero so didn't check for the condition	program crashed	15
2/26/06 9:45pm	hw3_main.c	N	Used wrong variable in a calculation	copy-pasted code from another section, didn't modify pasted code	incorrect output	45
2/27/06 11:10am	hw3_mpi.c	N	Incorrectly put statement inside loop	carelessness (no specific reason)	program hung (never stopped running)	60
2/27/06 1:05pm	hw3_mpi.c	Y	Used "for (i=0; i<N; i++)" instead of "for (i=1; i<N+1; i++)"	didn't realize the loop boundary must be changed in the parallel version	incorrect output	70
2/28/06 7:30pm	hw3_mpi.c	Y	The task was not equally distributed among processors	the algorithm used was too simple	slow execution on 8 processors	couldn't fix

What percentage of the overall debugging effort was spent to the defects specific to parallel programming? 60%

Chapter C

Educational Material

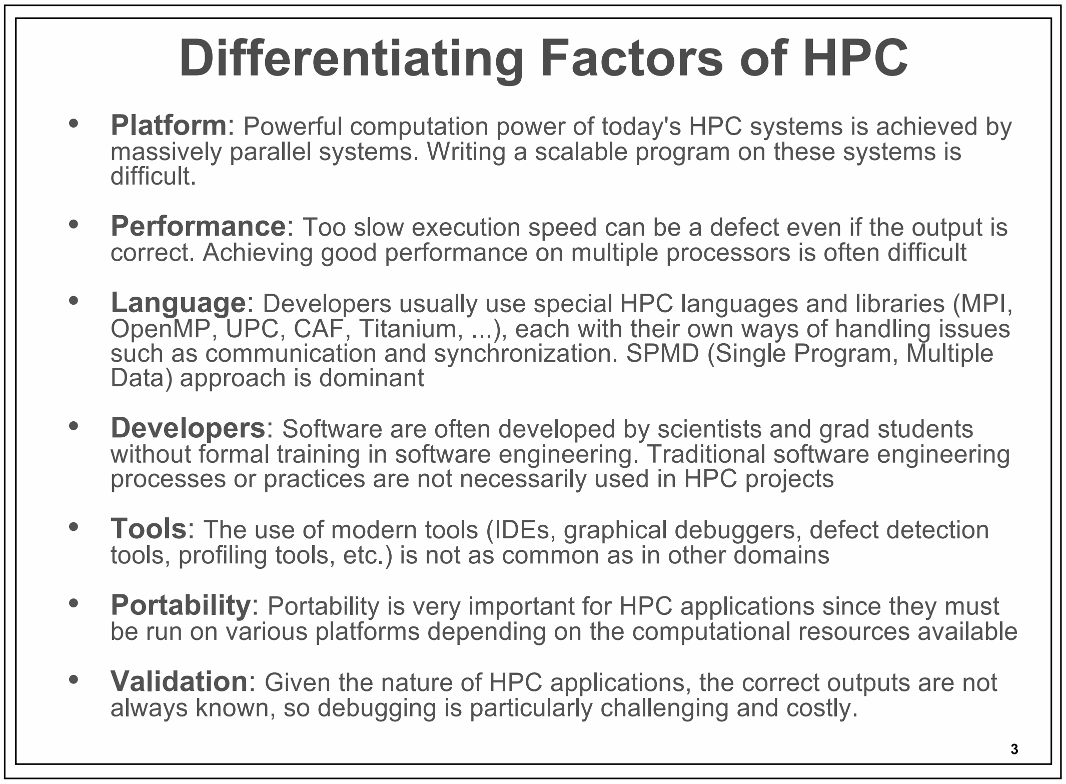
C.1 Lecture slides for defect patterns



Introduction

- Debugging and testing parallel code is hard
  - How can they be prevented or found/fixed effectively?
- "Knowing" common defects (bugs) will reduce the time spent debugging
  - Novice developers can *learn* how to detect/prevent them
  - Someone may develop tools and/or improve language
- We are building "Defect patterns" in high performance programming (HPC)
  - Based on the empirical data we collected in various studies
  - Examples are shown in C + MPI (Message Passing Interface)
  - We are building a website called HPCBugBase (http://www.hpcbugbase.org/) to share and evolve these patterns

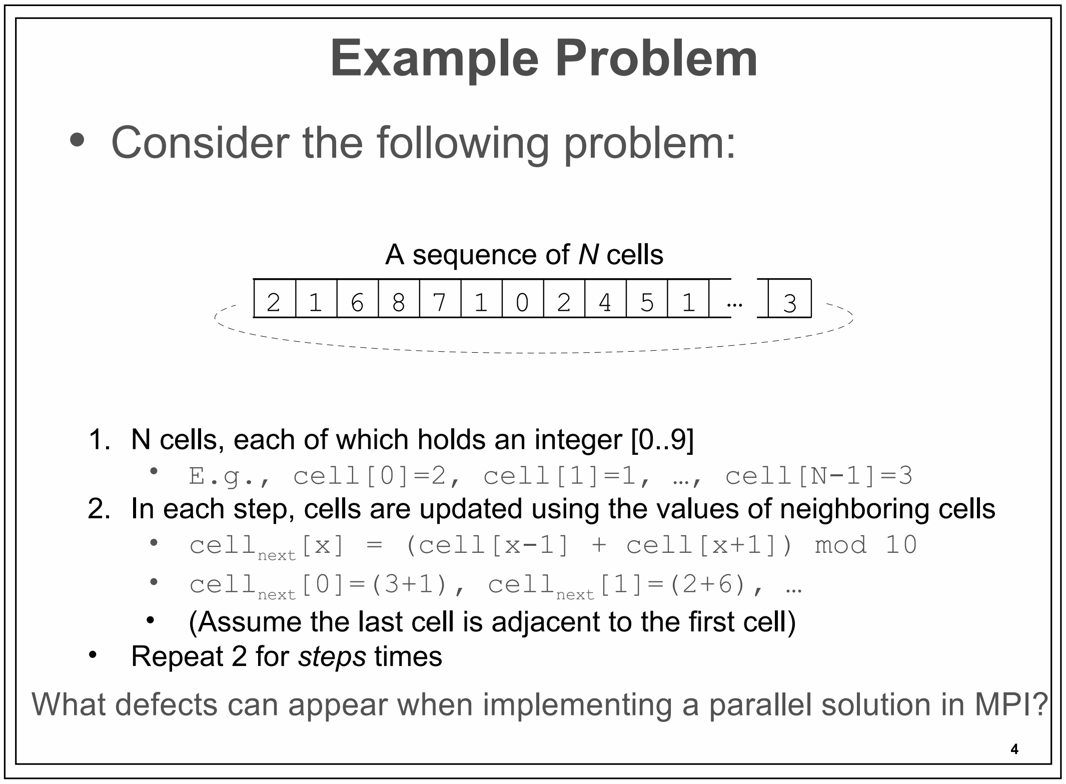
2



Differentiating Factors of HPC

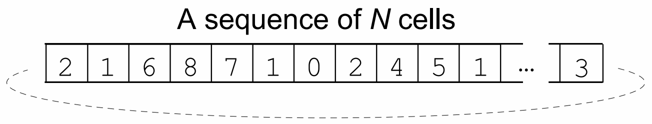
- **Platform**: Powerful computation power of today's HPC systems is achieved by massively parallel systems. Writing a scalable program on these systems is difficult.
- **Performance**: Too slow execution speed can be a defect even if the output is correct. Achieving good performance on multiple processors is often difficult
- **Language**: Developers usually use special HPC languages and libraries (MPI, OpenMP, UPC, CAF, Titanium, ...), each with their own ways of handling issues such as communication and synchronization. SPMD (Single Program, Multiple Data) approach is dominant
- **Developers**: Software are often developed by scientists and grad students without formal training in software engineering. Traditional software engineering processes or practices are not necessarily used in HPC projects
- **Tools**: The use of modern tools (IDEs, graphical debuggers, defect detection tools, profiling tools, etc.) is not as common as in other domains
- **Portability**: Portability is very important for HPC applications since they must be run on various platforms depending on the computational resources available
- **Validation**: Given the nature of HPC applications, the correct outputs are not always known, so debugging is particularly challenging and costly.

3



Example Problem

- Consider the following problem:



1. N cells, each of which holds an integer [0..9]
   - `E.g., cell[0]=2, cell[1]=1, …, cell[N-1]=3`
2. In each step, cells are updated using the values of neighboring cells
   - $\texttt{cell}_{\texttt{next}}\texttt{[x] = (cell[x-1] + cell[x+1]) mod 10}$
   - $\texttt{cell}_{\texttt{next}}\texttt{[0]=(3+1), cell}_{\texttt{next}}\texttt{[1]=(2+6), …}$
   - (Assume the last cell is adjacent to the first cell)
- Repeat 2 for *steps* times

What defects can appear when implementing a parallel solution in MPI?

4

First, Sequential Solution

- Approach to implementation
  - Use an integer array `buffer[]` to represent the cell values
  - Use a second array `nextbuffer[]` to store the values in the next step, and swap the buffers
  - Straightforward implementation!

5

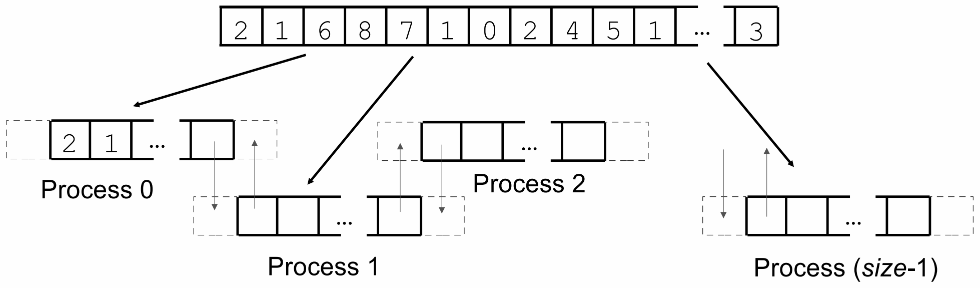
Sequential C Code

```
/* Initialize cells */
int x, n, *tmp;
int *buffer     = (int*)malloc(N * sizeof(int));
int *nextbuffer = (int*)malloc(N * sizeof(int));
FILE *fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }
for (x = 0; x < N; x++) { fscanf(fp, "%d", &buffer[x]); }
fclose(fp);

/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 0; x < N; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}

/* Final output */
...
free(nextbuffer); free(buffer);
```

6

Approach to a Parallel Version

- Each process keeps (1/size) of the cells
  - *size*:number of processes



- Each process needs to:
  - update the locally-stored cells
  - exchange boundary cell values between neighboring processes (nearest-neighbor communication)

7

Recurring HPC Defects

- Now, we will simulate the process of writing parallel code and discuss what kinds of defects can appear.
- Defect types are shown as:
  - Pattern descriptions (symptoms, causes, cures & preventions)
  - Concrete examples in MPI implementation

8

Pattern: **Erroneous use of parallel language features**

- "Simple" mistakes that are common for novices: language usage, choice of function, etc.
  - E.g., forgotten mandatory function calls for init/finalize
  - E.g., inconsistent parameter types between send and recv

Symptoms:

- Compile-type error (easy to fix)
- Some defects may surface only under specific conditions
  - (number of processors, value of input, hardware/software environment…)

Causes:

- Lack of experience with the syntax and semantics of new language features

Cures & preventions:

- Understand subtleties and variations of language features
- In a large code, confine parallel function calls to a particular part of the code to make fewer errors

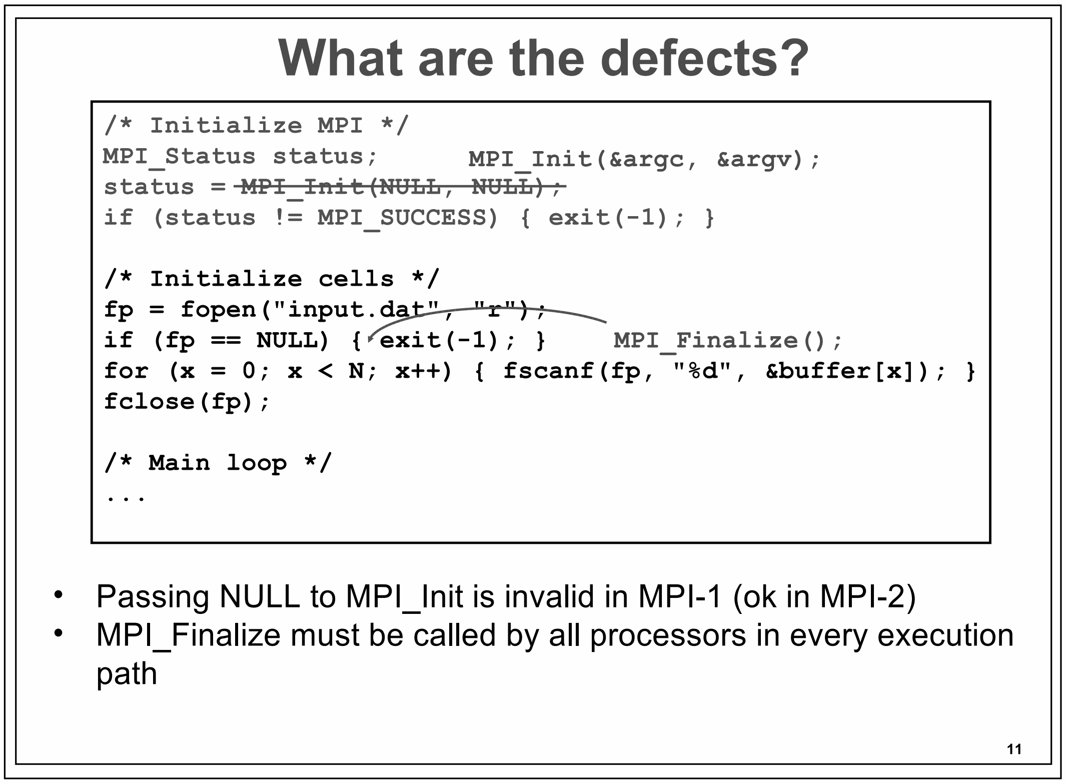
9

Adding basic MPI functions

```
/* Initialize MPI */
MPI_Status status;
status = MPI_Init(NULL, NULL);
if (status != MPI_SUCCESS) { exit(-1); }

/* Initialize cells */
fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }
for (x = 0; x < N; x++) { fscanf(fp, "%d", &buffer[x]); }
fclose(fp);

/* Main loop */
...

/* Final output */
...

/* Finalize MPI */
MPI_Finalize();
```

What are the bugs?

10

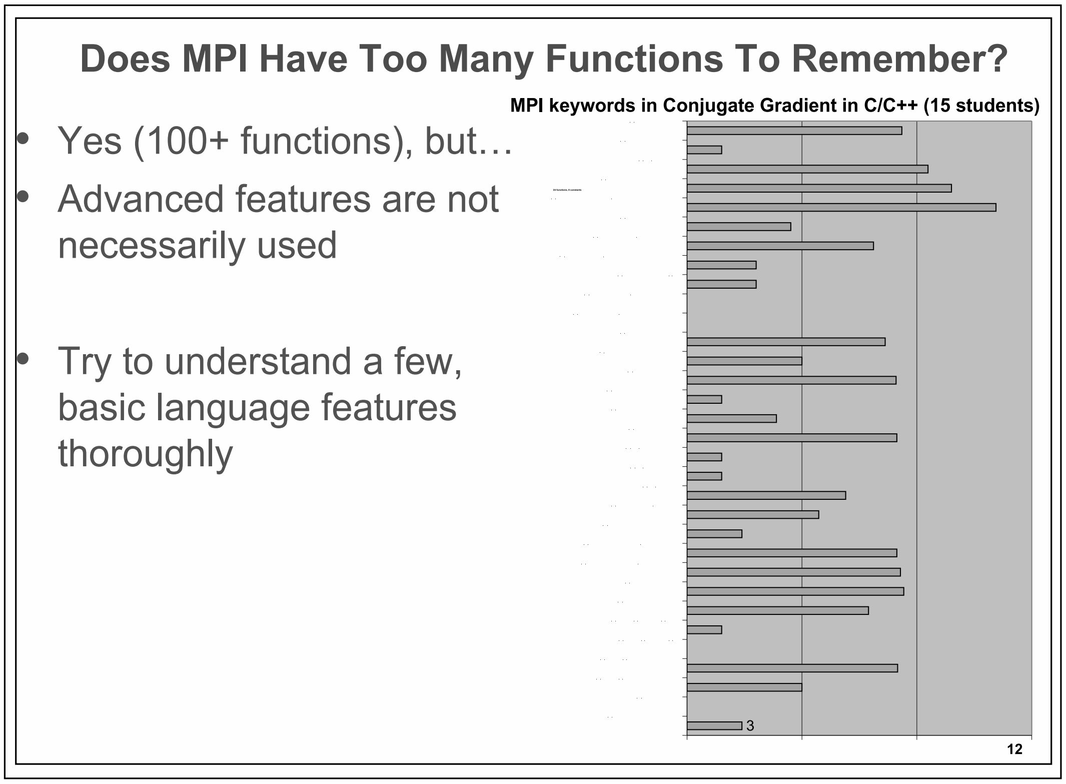


What are the defects?

```
/* Initialize MPI */
MPI_Status status;      MPI_Init(&argc, &argv);
status = ~~MPI_Init(NULL, NULL);~~
if (status != MPI_SUCCESS) { exit(-1); }

/* Initialize cells */
fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }     MPI_Finalize();
for (x = 0; x < N; x++) { fscanf(fp, "%d", &buffer[x]); }
fclose(fp);

/* Main loop */
...
```

- Passing NULL to MPI_Init is invalid in MPI-1 (ok in MPI-2)
- MPI_Finalize must be called by all processors in every execution path

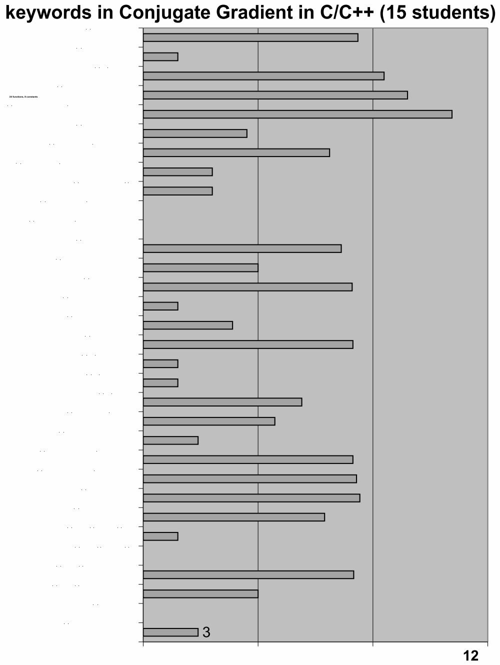
11



Does MPI Have Too Many Functions To Remember?

- Yes (100+ functions), but…
- Advanced features are not necessarily used
- Try to understand a few, basic language features thoroughly

MPI keywords in Conjugate Gradient in C/C++ (15 students)



12

Pattern: **Space Decomposition**

- Incorrect mapping between the problem space and the program memory space

Symptoms:

- Segmentation fault (if array index is out of range)
- Incorrect or slightly incorrect output

Causes:

- Mapping in parallel version can be different from that in serial version
    - E.g., Array origin is different in every processor
    - E.g., Additional memory space for communication can complicate the mapping logic

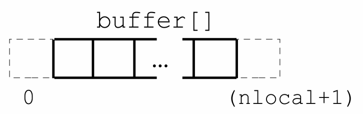
Cures & preventions:

- Validate array origin, whether buffer includes guard buffers, whether buffer refers to global space or local space, etc. - these can change while parallelizing the code
- Encapsulate the mapping logic to a dedicated function
- Consider designing the code which is easy to parallelize

13

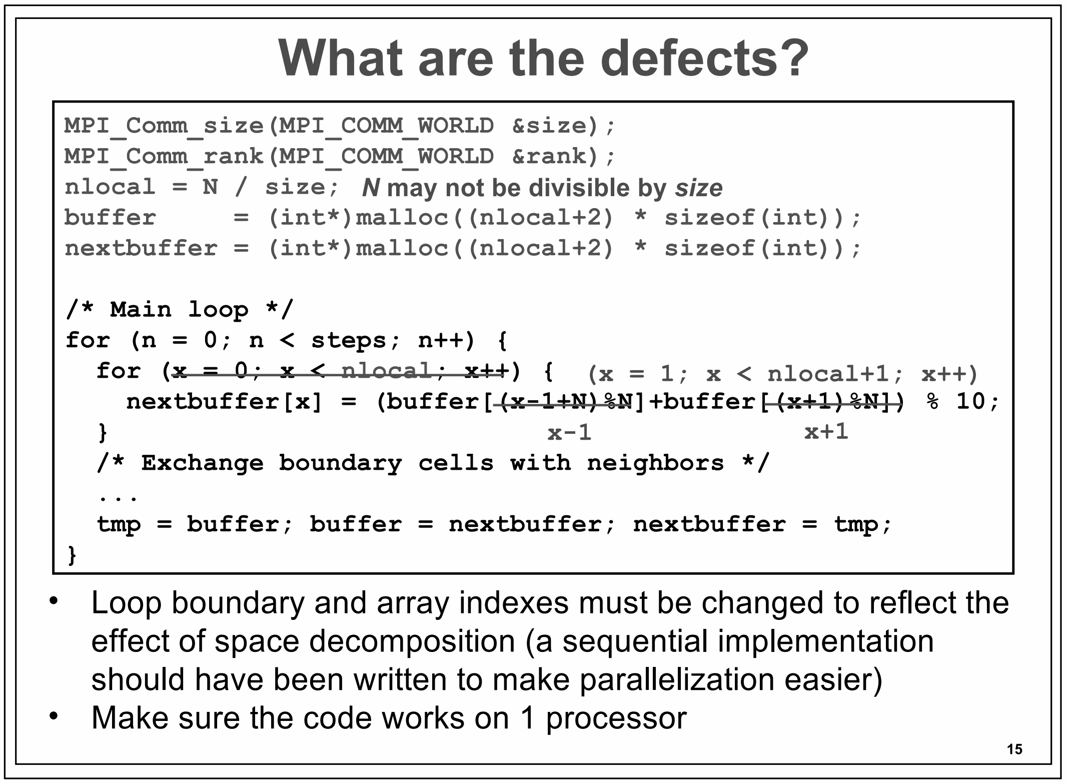
Decompose the problem space

```
MPI_Comm_size(MPI_COMM_WORLD &size);
MPI_Comm_rank(MPI_COMM_WORLD &rank);
nlocal = N / size;
buffer     = (int*)malloc((nlocal+2) * sizeof(int));
nextbuffer = (int*)malloc((nlocal+2) * sizeof(int));

/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 0; x < nlocal; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  ...
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```



What are the bugs?

14

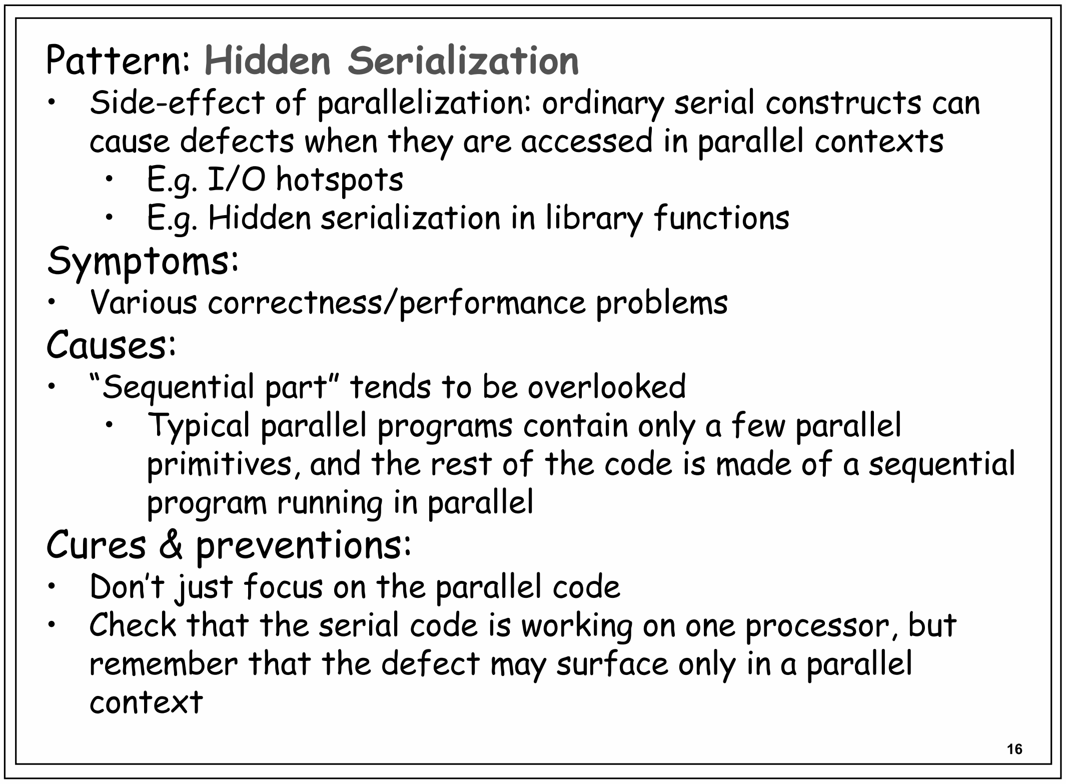


What are the defects?

```
MPI_Comm_size(MPI_COMM_WORLD &size);
MPI_Comm_rank(MPI_COMM_WORLD &rank);
nlocal = N / size;  N may not be divisible by size
buffer     = (int*)malloc((nlocal+2) * sizeof(int));
nextbuffer = (int*)malloc((nlocal+2) * sizeof(int));

/* Main loop */
for (n = 0; n < steps; n++) {
  for (~~x = 0; x < nlocal; x++~~) {  (x = 1; x < nlocal+1; x++)
    nextbuffer[x] = (buffer[~~(x-1+N)%N~~]+buffer[~~(x+1)%N]~~) % 10;
  }                               x-1              x+1
  /* Exchange boundary cells with neighbors */
  ...
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- Loop boundary and array indexes must be changed to reflect the effect of space decomposition (a sequential implementation should have been written to make parallelization easier)
- Make sure the code works on 1 processor

15



Pattern: **Hidden Serialization**

- Side-effect of parallelization: ordinary serial constructs can cause defects when they are accessed in parallel contexts
  - E.g. I/O hotspots
  - E.g. Hidden serialization in library functions

Symptoms:

- Various correctness/performance problems

Causes:

- "Sequential part" tends to be overlooked
  - Typical parallel programs contain only a few parallel primitives, and the rest of the code is made of a sequential program running in parallel

Cures & preventions:

- Don't just focus on the parallel code
- Check that the serial code is working on one processor, but remember that the defect may surface only in a parallel context

16

Data I/O

```
/* Initialize cells with input file */
fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }
nskip = ...
for (x = 0; x < nskip; x++) { fscanf(fp, "%d", &dummy);}
for (x = 0; x < nlocal; x++) { fscanf(fp, "%d", &buffer[x+1]);}
fclose(fp);

/* Main loop */
...
```

- What are the defects?

17

Data I/O

```
/* Initialize cells with input file */
if (rank == 0) {
fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }
for (x = 0; x < nlocal; x++) { fscanf(fp, "%d", &buffer[x+1]);}
for (p = 1; p < size; p++) {
  /* Read initial data for process p and send it */
}
fclose(fp);
}
else {
  /* Receive initial data*/
}
```

- Filesystem may cause performance bottleneck if all processors access the same file simultaneously
  - (Schedule I/O carefully, or let “master” processor do all I/O)

18

Generating Initial Data

```
/* What if we initialize cells with random values... */
srand(time(NULL));
for (x = 0; x < nlocal; x++) {
  buffer[x+1] = rand() % 10;
}

/* Main loop */
...
```

- What are the defects?

- (Other than the fact that rand() is not a good pseudo-random number generator in the first place…)

19

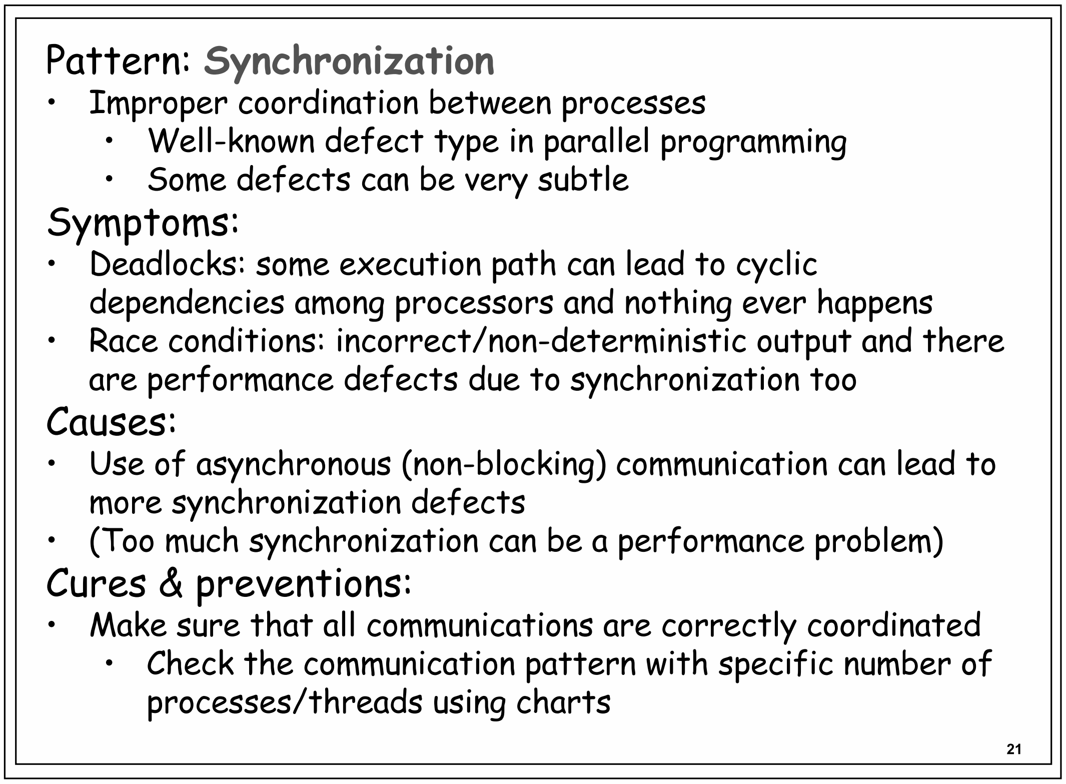
What are the Defects?

```
/* What if we initialize cells with random values... */
~~srand(time(NULL));~~    srand(time(NULL) + rank);
for (x = 0; x < nlocal; x++) {
  buffer[x+1] = rand() % 10;
}

/* Main loop */
...
```

- All procs might use the same pseudo-random sequence, spoiling independence (correctness)

- Hidden serialization in the library function rand() causes performance bottleneck

20



Pattern: **Synchronization**

- Improper coordination between processes
  - Well-known defect type in parallel programming
  - Some defects can be very subtle

Symptoms:

- Deadlocks: some execution path can lead to cyclic dependencies among processors and nothing ever happens
- Race conditions: incorrect/non-deterministic output and there are performance defects due to synchronization too

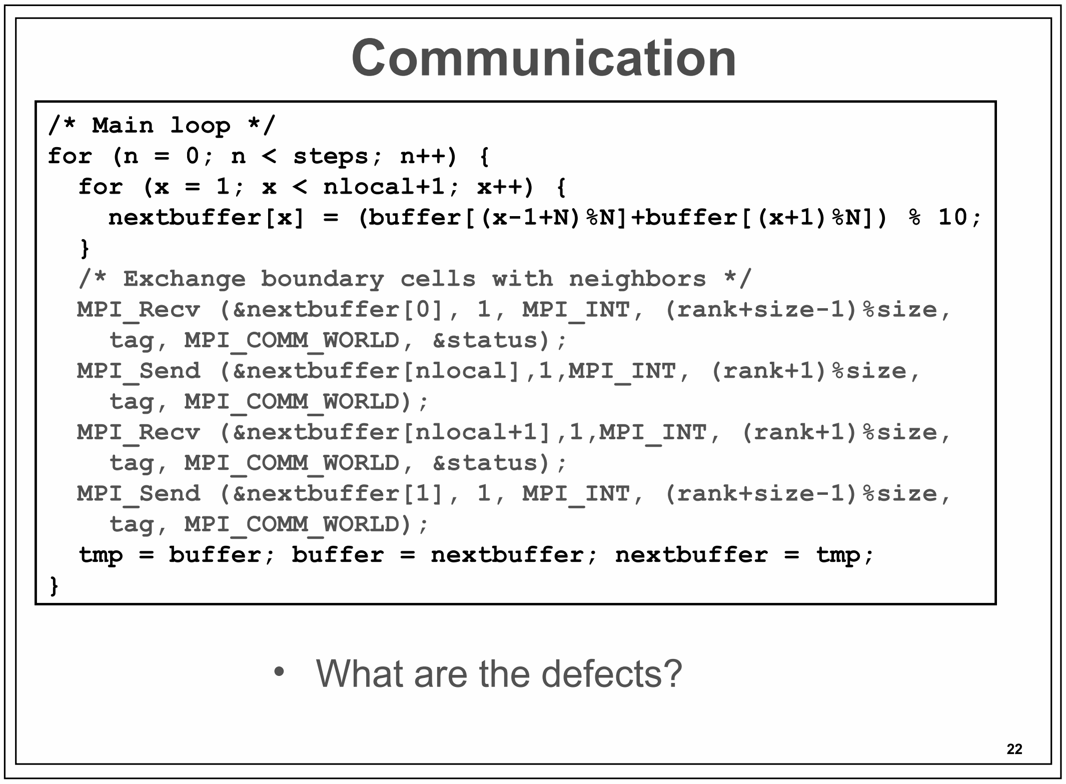
Causes:

- Use of asynchronous (non-blocking) communication can lead to more synchronization defects
- (Too much synchronization can be a performance problem)

Cures & preventions:

- Make sure that all communications are correctly coordinated
  - Check the communication pattern with specific number of processes/threads using charts

21



Communication

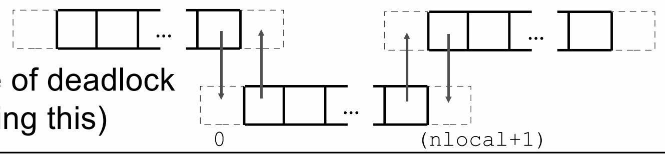
```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- What are the defects?

22

What are the Defects?

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[nlocal],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[1], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```



- Obvious example of deadlock (can't avoid noticing this)

23

Another Example

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Ssend (&nextbuffer[nlocal],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Ssend (&nextbuffer[1], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- What are the defects?

24

What are the Defects?

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Ssend (&nextbuffer[nlocal],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Ssend (&nextbuffer[1], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- This causes deadlock too
- MPI_Ssend is a *synchronous* send (see the next slides.)

Yet Another Example

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Send (&nextbuffer[nlocal],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[1], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- What are the defects?

Potential Deadlock

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Send (&nextbuffer[nlocal],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[1], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- This may work (many novice programmers write this code)
- but it can cause deadlock with some MPI implementation, runtime environment and/or execution parameters

27

Modes of MPI blocking communication

- http://www.mpi-forum.org/docs/mpi-11-html/node40.html
  - **Standard** (MPI_Send): may either return immediately when the outgoing message is buffered in the MPI buffers, or block until a matching receive has been posted.
  - **Buffered** (MPI_Bsend): a send operation is completed when the MPI buffers the outgoing message. An error is returned when there is insufficient buffer space
  - **Synchronous** (MPI_Ssend): a send operation is complete only when the matching receive operation has started to receive the message.
  - **Ready** (MPI_Rsend): a send can be started only after the matching receive has been posted.
- In our code MPI_Send won't probably be blocked in most implementations (each message's just one integer), but it should still be avoided.
- A "correct" solution for this defect could be:
  - (1) alternate the order of send and recv
  - (2) use MPI_Bsend with sufficient buffer size
  - (3) MPI_Sendrecv, or
  - (4) MPI_Isend/recv

28

An Example Fix

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  if (rank % 2 == 0) { /* even ranks send first */
    MPI_Send (..., (rank+size-1)%size, ...);
    MPI_Recv (..., (rank+1)%size, ...);
    MPI_Send (..., (rank+1)%size, ...);
    MPI_Recv (..., (rank+size-1)%size, ...);
  } else {             /* odd ranks recv first */
    MPI_Recv (..., (rank+1)%size, ...);
    MPI_Send (..., (rank+size-1)%size, ...);
    MPI_Recv (..., (rank+size-1)%size, ...);
    MPI_Send (... , (rank+1)%size, ...);
  }
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

29

Non-Blocking Communication

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Isend (&nextbuffer[nlocal],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &request1);
  MPI_Irecv (&nextbuffer[0], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &request2);
  MPI_Isend (&nextbuffer[1], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &request3);
  MPI_Irecv (&nextbuffer[nlocal+1],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &request4);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- What are the defects?

30

What are the Defects?

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Isend (&nextbuffer[nlocal],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &request1);
  MPI_Irecv (&nextbuffer[0], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &request2);
  MPI_Isend (&nextbuffer[1], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &request3);
  MPI_Irecv (&nextbuffer[nlocal+1],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &request4);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- Synchronization (e.g. MPI_Wait, MPI_Barrier) is needed at each iteration (but too many barriers can cause a performance problem)

31

Pattern: **Performance defect**
- Scalability problem because processors are not working in parallel
  - The program output itself is correct
  - Perfect parallelization is often difficult: need to evaluate if the execution speed is unacceptable

Symptoms:
- Sub-linear scalability
- Performance much less than expected (e.g, most time spent waiting),

Causes:
- Unbalanced amount of computation
- Load balancing may depend on input data

Cures & preventions:
- Make sure all processors are “working” in parallel
- Profiling tool might help

32

Scheduling communication

```
if (rank != 0) {
  MPI_Ssend (&nextbuffer[nlocal],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
}
if (rank != size-1) {
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Ssend (&nextbuffer[1], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
}
```

- Complicated communication pattern- does not cause deadlock

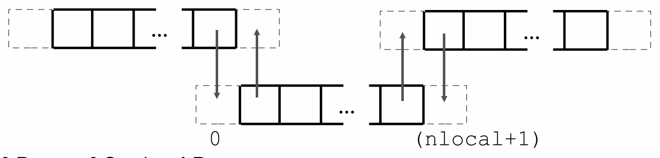
What are the defects?

33

What are the bugs?

```
if (rank != 0) {
  MPI_Ssend (&nextbuffer[nlocal],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
}
if (rank != size-1) {
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT,(rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Ssend (&nextbuffer[1], 1, MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
}
```

- Serialization in communication : requires O(size) time (a “correct” solution takes O(1))



1 Send → 0 Recv → 0 Send → **1 Recv**
2 Send → **1 Recv** → **1 Send** → 2 Recv
3 Send → 2 Recv → 2 Send → 3 Recv

34

C.2 Homework

Identify correctness/performance problems in the MPI program given at [URL], which is supposed to calculate the approximation of the number pi by a Monte Carlo simulation. For each defect found, describe briefly (1) why it is considered a defect, (2) which of the defect types presented in the lecture it belongs to (or it belongs to none), and (3) how it can be fixed.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 1000003L

int main()
{
  long n, k, i;
  int rank, size;
  double x, y;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  srand(time(NULL));
```

```
n = N;

k = 0;

for (i=0; i<n/size; i++) {

  char inside;

  x = rand()/((double)RAND_MAX);

  y = rand()/((double)RAND_MAX);

  if (x * x + y * y < 1.0) {

    inside = 1;

  }

  else {

    inside = 0;

  }

  if (rank == 0) {

    int j;

    if (inside == 1) k = k + 1;

    for (j=1; j<size; j++) {

      MPI_Status status;

      MPI_Recv(&inside, 1, MPI_CHAR, j, 0, MPI_COMM_WORLD, &status);

      if (inside == 1) k = k + 1;

    }

  }

  else {

    MPI_Send(&inside, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }

  }

  if (rank == 0) {

    printf("%f\n", 4.0 * k / ((double)n));

  }

  return 0;

}
```

Chapter D

Case Study Results

D.1 Results

Reference group

# of defects (Total)	# of defects (Never resolved)	# of defects (Resolved)
9	3	6
11	5	6
18	6	12
7	4	3
13	6	7
6	6	0
13	3	10

(Continued)

# of defects (Total)	# of defects (Never resolved)	# of defects (Resolved)
10	7	3
13	7	6
14	4	10
5	5	0
10	7	3
16	9	7
6	6	0
6	3	3
2	2	0
28	5	23
6	3	3
8	4	4
1	1	0
5	4	1

Test group

# of defects (Total)	# of defects (Never resolved)	# of defects (Resolved)
8	7	1
8	0	8
1	1	0
9	4	5
2	0	2
7	5	2
3	3	0
3	2	1
5	2	3
8	4	4
3	3	0
16	5	11
12	7	5

D.2 Statistical analysis

F-test (two-sample for variance): the null hypothesis is not rejected and the conclusion is that the two variances do not differ significantly.

	Reference group	Test group
Mean	9.857	6.538
Variance	37.03	18.60
F	1.991	
$P(F \leq f)$ one-tail	0.111	
F threshold for .05	2.544	

t-test (two-sample, assuming equal variances): the difference is significant at the .05 level for the defects that remained unfixed in the final version as well as the whole defects, while it is not significant the defects that were resolved during the development.

Test	Value
t (total)	1.714
$P(T \leq t)$ one-tail (total)	0.04815
t (never resolved)	1.966
$P(T \leq t)$ one-tail (never resolved)	0.02901
t (resolved)	1.106
$P(T \leq t)$ one-tail (resolved)	0.1385
t threshold for .05	1.694

Bibliography

[1] R. J. Allan, Y. F. Hu, and P. Lockey. Parallel application software on high performance computers. survey of parallel numerical analysis software. Technical report, CLRC Daresbury Laboratory, 1999.

[2] Eric Allen. *Bug Patterns in Java.* Apress, October 2002.

[3] M. Antonioletti and E. Breitmoser. Software engineering and code development for hpc applications. Technical report, Edinburgh Parallel Computing Centre, 2000.

[4] Cyrille Artho. Finding faults in multi-threaded programs, 2001.

[5] Scott Barber. Diagnosing symptoms and solving problems. *Software Test & Performance*, pages 14–20, July 2005.

[6] Victor Basili, Roseanne Tesoriero, Patricia Costa, Mikael Lindvall, Ioana Rus, Forrest Shull, and Marvin Zelkowitz. Building an experience base for software engineering: A report on the first CeBASE eWorkshop. *Lecture Notes in Computer Science*, 2188:110–??, 2001.

[7] Victor R. Basili and Gianluigi Caldiera. Improve software quality by reusing knowledge and experience. *Slone Management Review*, 37(1):55–64, 1995.

[8] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Using experiments to build a body of knowledge. In *Ershov Memorial Conference*, pages 265–282, 1999.

[9] Patricia Benner. *From Novice to Expert: Excellence and Power in Clinical Nursing Practice.* Prentice Hall, 1st edition edition, January 2001. ISBN: 0130325228.

[10] Konstantin Berlin, Jun Huan, Mary Jacob, Garima Kochhar, Jan Prins, Bill Pugh, P. Sadayappan, Jaime Spacco, and Chau-Wen Tseng. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In *16th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, October 2003.

[11] Jean-Yves Berthou and Eric Fayolle. Comparing openmp, hpf, and mpi programming: A study case. *International Journal of High Performance Computing Applications*, 15(3):297–309, 2001.

[12] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

[13] Eric Bouwers. Analyzing php, an introduction to php-sat. Utrecht University, Software Technology Colloquium, 2007.

[14] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, Sanjay Ranka, and Min-You Wu. Compiling fortran 90d/hpf for distributed memory mimd computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, 1994.

[15] William J. Brown, Raphael Malveau, Hays McCormick III, and Thomas Mowbray. *Anti-patterns: Refactoring software, architectures and projects in crisis.* John Wiley & Sons, 1 edition edition, February 2001. ISBN: 0471197130.

[16] William J. Brown, Hays W. McCormick, and Scott W. Thomas. *Anti-Patterns and Patterns in Software Configuration Management.* John Wiley & Sons Inc., April 1999. ISBN: 0471329290.

[17] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The cascade high productivity language. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, 2004.

[18] William W. Carlson. Introduction to upc and language specification. Technical Report CCS-TR-99-157, 1999.

[19] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course.* The MIT Press, 1990.

[20] Jeffrey Carver, Lorin Hochstein, Richard Kendall, Taiga Nakamura, Marvin Zelkowitz, Victor Basili, and Douglass Post. Observations about software development for high end computing. *CTWatch Quarterly*, 2(4A):33–38, November 2006.

[21] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[22] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992.

[23] Ram Chillarege, Wei-Lun Kao, and Richard G. Condit. Defect type and its impact on the growth curve. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 246–255, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[24] Jong-Deok Choi and Sang Lyul Min. Race frontier: reproducing data races in parallel-program debugging. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 145–154, New York, NY, USA, 1991. ACM Press.

[25] Ron Choy and Alan Edelman. Parallel matlab: Doing it right. *Proceedings of IEEE*, 93(Issue 2):331– 341, 2005.

[26] Tom Copeland. Static analysis with pmd, 2 2003.

[27] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[28] Julian C. Stanley Donald T. Campbell and. *Experimental and Quasi-Experimental Designs for Research.* Houghton Mifflin Company, July 1963.

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

[30] Marc Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, 1997.

[31] Douglas Ezzy. *Qualitative Analysis: Practice and Innovation.* Routledge, 2003.

[32] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. *ipdps*, 00:286b, 2003.

[33] Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical Report UT-CS-94-230, Message Passing Interface Forum, 1994.

[34] Jr. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[36] Kate Gordon. *Educational Psychology.* Henry Halt and Company, 1917.

[37] Charles Henderson, Kenneth Heller, Patricia Heller, Vince H. Kuo, and Edit Yerushalmi. Students learning problem solving in introductory physics - forming an initial hypothesis of instructors' beliefs. In *Proceedings of the Physics Education Research Conference*, 2002.

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

[39] Lorin Hochstein and Victor R. Basili. Hpc- the asc-alliance projects: A case study of large-scale parallel scientific code development. Technical Report CS-TR-4834 and UMIACS-TR-2006-50, University of Maryland, 2006.

[40] Lorin Hochstein, Victor R. Basili, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, and Jeff Carver. Combining self-reported and automatic data to improve programming effort measurement. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 356–365, New York, NY, USA, 2005. ACM Press.

[41] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2005. IEEE Computer Society.

[42] Jeffery K. Hollingsworth. *Finding Bottlenecks in Large Scale Parallel Programs.* PhD thesis, University of Wisconsin-Madison, 1994.

[43] Daryl L. L. Houston. Phpitfalls: Five beginner mistakes to avoid. Digital Web Magazine, 2004.

[44] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[45] Michael Howard, David LeBlanc, and John Viega. *19 Deadly Sins of Software Security.* McGraw-Hill Osborne Media, July 2005. ISBN: 0072260858.

[46] Douglas H. Johnson. The insignificance of statistical significance testing. *Journal of Wildlife Management*, 63(3):763–772, 1999.

[47] Philip M. Johnson. Project hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis. Technical Report csdl2-01-13, University of Hawaii, 2001.

[48] Capers Jones. *Tutorial programming productivity: issues for the eighties, 2nd ed.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1986.

[49] Jeremy Kepner. Hpc productivity: An overarching view. *International Journal of High Performance Computing Applications*, 18(4):393–397, 2004.

[50] Sunghun Kim, Kai Pan, and Jr. E. E. James Whitehead. Memories of bug fixes. pages 35–45, 2006.

[51] John C. Knight and E. Ann Myers. An improved inspection technique. *Commun. ACM*, 36(11):51–61, 1993.

[52] Andrew J. Ko and Brad A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41–84, 2005.

[53] Sandeep Kumar. *Classification and Detection of Computer Intrusions.* PhD thesis, Purdue University, August 1995.

[54] Douglas B. Lenat. Cyc: a large-scale investment in knowledge infrastructure. *Commun. ACM*, 38(11):33–38, 1995.

[55] Henry Lieberman. The debugging scandal and what to do about it (introduction to the special section). *Commun. ACM*, 40(4):26–29, 1997.

[56] Kenneth Litkowski. Using structured interviewing techniques. Technical Report GAO/PEMD-10.1.5, United States General Accounting Office, 1991.

[57] Chris Livesey and Tony Lawson. *AS Sociology for AQA.* Hodder Arnold, 2005.

[58] Joseph A. Maxwell. *Qualitative Research Design: An Interactive Approach.* Sage Publications, Inc., 2005.

[59] Richard E. Mayer, David Berliner, Carol Dwyer, Barbara McCombs, Sharon McNeely, James Pellegrino, Sigmund Tobias, and Anita Woolfolk. Educational psychology q and a. *Division of Educational Psychology of the American Psychological Association.*

[60] Amir Michail and Tao Xie. Helping users avoid bugs in gui applications. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 107–116, 2005.

[61] Don Morton, K. Wang, and D. O. Ogbe. Lessons learned in porting fortran/pvm code to the crayt3d. *IEEE Parallel & Distributed Technology: Systems & Technology*, 3(1):4–11, 1995.

[62] Michael Süb nad Claudia Leopold. Common mistakes in openmp and how to avoid them - a collection of best practices. In *International Workshop on OpenMP (IWOMP 2006)*, 2006.

[63] NERSC. Nersc debugging tutorial. Website, http://www.nersc.gov/nusers/help/tutorials/debug/.

[64] R. W. Numrich and J. K. Reid. Co-array fortran for parallel programming. Technical Report RAL-TR-1998-060, RAL, 1998.

[65] Jean Piaget. *The Psychology of Intelligence.* Routledge, 1950.

[66] Lutz Prechelt. Accelerating learning from experience: Avoiding defects faster. *IEEE Software*, 18(6):56–61, 2001.

[67] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[68] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[69] Forrest Shull, Vic Basili, Barry Boehm, A. Winsor Brown , Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, page 249, Washington, DC, USA, 2002. IEEE Computer Society.

[70] Mark Sullivan and Ram Chillarege. A comparison of software defects in database management systems and operating systems. In *IEEE Twenty-Second International Symposium on Fault-Tolerant Computing*, Boston, Massachusetts, July 1992. IEEE. ISBN: 0-8186-2875-8.

[71] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The pvm concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, 1994.

[72] David Ungar, Henry Lieberman, and Christopher Fry. Debugging and the experience of immediacy. *Commun. ACM*, 40(4):38–43, 1997.

[73] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005.

[74] Michael Wolfe. *High Performance Compiler for Parallel Computing.* Addison-Wesley, 1996. ISBN 0-8053-2730-4.

[75] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance java dialect. *Concurrency-Practice and Experience, Java Special Issue*, 1998.

[76] Betty Zan, Carolyn Hildebrandt, Rebecca Edmiaston, and Christina Sales. *Developing Constructivist Early Childhood Curriculum: Practical Principals and Activities.* Teachers College Press, 2001.