TOWARDS COMMENT GENERATION FOR MPI PROGRAMS

by

Suparna Manjunath

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences

Fall 2011

© 2011 Suparna Manjunath
All Rights Reserved

TOWARDS COMMENT GENERATION FOR MPI PROGRAMS

by

Suparna Manjunath

Approved: ____
Lori Pollock, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ____
Stephen Siegel, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ____
James Clause, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ____
Errol Lloyd, Ph.D.
Chairperson, Computer & Info Sciences

Approved: ____
Charles G. Riordan, Ph.D.
Vice Provost for Graduate and Professional Education

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

LIST OF TABLES . . . . **vii**
ABSTRACT . . . . **viii**

Chapter

1 INTRODUCTION **1**

1.1 Contributions 3
1.2 Outline 4

2 BACKGROUND **5**

2.1 Overview of MPI 5

2.1.1 Salient Features of MPI 5

2.1.1.1 SPMD Parallel Program Pattern 5
2.1.1.2 Process Groups and Communication Contexts 6
2.1.1.3 Point-to-Point Message Passing Communication 6
2.1.1.4 Collective Communication 7

2.1.2 Example MPI Program 7

2.2 MPI for Java 8
2.3 State of the Art 9

2.3.1 Tools for Understanding Parallel Programs 10
2.3.2 Comment Generation for Sequential Programs 10

3 A SUMMARY GENERATOR FOR SEQUENTIAL JAVA . . . 11

3.1 Content Selection . . . 12

3.1.1 Ending s_units . . . 12
3.1.2 Void-Return s_units . . . 12
3.1.3 Same-Action s_units . . . 13
3.1.4 Data-Facilitating s_units . . . 13
3.1.5 Controlling s_units . . . 13

3.2 Text Generation . . . 13

4 FEASIBILITY STUDY OF SUMMARY COMMENT GENERATION FOR MPI PROGRAMS . . . 15

4.1 Towards Summary Generation for MPI . . . 15

4.1.1 MPI Parallel Programming Paradigm . . . 16
4.1.2 Predominant Application Domain . . . 17
4.1.3 Base Sequential Language . . . 18

4.2 Study Design . . . 19

4.2.1 Subject MPI Functions . . . 19
4.2.2 Procedure . . . 21
4.2.3 Threats to Validity . . . 23

4.3 Results and Discussion . . . 23

4.3.1 Content Selection . . . 23
4.3.2 Text Generation . . . 26

4.4 Implications for MPI Comment Generation . . . 27

5 GENERATING COMMENTS FOR PARALLEL MPI JAVA PROGRAMS 28

5.1 Summary Generator for MPI Java 28

5.1.1 Content Selection 28

5.1.1.1 Transferable 28
5.1.1.2 Modifications and Extensions 29

5.1.2 Text Generation 31

5.1.2.1 Transferable 31
5.1.2.2 Modifications and Extensions 31

5.2 Initial Evaluation of *genSummMPI* 33

5.2.1 Gathering and Selecting Benchmarks for Study 33
5.2.2 Challenges in Evaluation 35
5.2.3 Analysis and Discussion of Generated Comments 36

5.2.3.1 *genSummMPI* Successes 36
5.2.3.2 Towards Improving the Precision of *genSummMPI* . . . 38

5.2.4 Future Enhancements 41

6 CONCLUSIONS AND FUTURE WORK 42

Appendix

A TEXT GENERATION TEMPLATES FOR MPI LIBRARY CALLS 44
B TEXT PHRASES FOR MPI CONSTANTS 54

BIBLIOGRAPHY 56

LIST OF TABLES

4.1 Characteristics of Programs Containing Subject Functions 20

4.2 Results from Simulating Automatic Content Selection
(249 manually identified s_units, 296 automatically selected s_sunits) . . 25

5.1 Characteristics of Java MPI Benchmarks 35

ABSTRACT

As parallel architectures are becoming more commonplace, the development of parallel software that exploits the potential parallelism is increasingly important. Like sequential programs, these programs need to be maintained, which requires considerable time understanding the maintenance tasks and related software or documentation. Studies have shown that good comments can help programmers quickly understand a program's functions. Unfortunately, few software projects adequately comment the code.

This thesis work investigates the problem of automatically generating descriptive summary comments for functions in MPI parallel programs and presents an approach to generate such descriptive summary comments for parallel MPI Java methods. Given the signature and body of a method, the automatic comment generator identifies content for the summary and generates natural language text that summarizes the method's overall actions. A base comment generator for sequential Java is modified for a first prototype for parallel MPI programs.

Chapter 1

INTRODUCTION

Widespread deployment of parallel architectures and networked PC clusters has triggered an increase in parallel programming demand. A significant number of large parallel applications are written in MPI [14], and it has been shown that parallel programs written in MPI can be both efficient and portable to various parallel environments. However, it is often described as being analogous to assembly language programming because the programmer handles the low level primitives for interprocess communication and data distribution [25]. With concurrency, nondeterministic execution, data distribution, and communication, these programs are particularly challenging to maintain for debugging, adding new features, and improving performance [16].

An important prerequisite to maintaining a program is finding and understanding the code relevant to the maintenance task, which can take more time than making the actual modifications [22]. Surveys (e.g., [34, 35, 37]) have demonstrated that comments help in understanding code and are considered useful during software maintenance tasks. With the added complexities of parallel program understanding, documentation is even more important. Unfortunately, few software projects adequately document the code to reduce future maintenance efforts [13, 21]. Our investigation of 6142 functions over 32 MPI open-source, parallel programs revealed that only 45% of the functions overall had any kind of leading comment, and the number of leading comments decreased to 32% (less than a third) of those functions actually involving calls to the MPI library.[1]

[1] To compare, large open-source Java programs were examined; 52% of the methods had any kind of leading comment.

An alternative to developer-written comments is to automatically generate comments directly from the source code. Recently, Sridhara et al. [32] developed a novel technique to automatically generate *descriptive summary comments* for sequential Java methods. A *descriptive summary comment* called a *summary*, summarizes the major algorithmic actions of a method, similar to how an abstract provides a summary for a natural language document [24]. Not only can such descriptive summary comments help programmers quickly understand what a method does, leading summary comments occurring before a method can also be useful for skimming a set of methods to decide which methods need to be examined in more detail. Evaluation of generated comments by human judges indicated that such generated summaries are accurate, do not miss important content, and are reasonably concise.

We performed a preliminary study to investigate the reusability of Sridhara et al.'s [32] automatic comment generation technique designed for sequential Java programs to solve the problem of automatically generating descriptive summary comments for functions in parallel MPI programs. As a part of this study, Sridhara et al.'s [32] automatic summary comment generator for sequential Java methods was applied on 22 functions from 13 open source MPI C programs. The set of statements selected as content for the summary and the generated text phrases were analyzed using human-written summary comments as a reference. Next, the content adequacy and conciseness of both the content selected and text generated was evaluated. Observations of missing or irrelevant content were categorized according to the perceived factors broadly organized as the *parallel programming paradigm,* the *scientific application domain* that dominates current MPI parallel programs, and the *base language differences*.

Results from the preliminary study suggested that the technique for content selection from the sequential Java comment generator could be reused with minor modifications, while text generation relies too heavily on language and library dependent information to be directly reused. As most of the necessary modifications for generating

summaries for MPI C programs are based on changes needed in text generation and the underlying base language change from Java to C, the remainder of this thesis work eliminates the base language factor and concentrates on comment generation for MPI Java programs to focus on the challenges of comment generation for parallel codes.

1.1 Contributions

a. A preliminary study to investigate the problem of automatic generation of comments for parallel MPI C programs with the following main contributions:

- Quantitative results from simulating automatic summary comment generation designed for sequential Java methods to generate summary comments for MPI parallel program functions, indicating that the technique for selecting content for the function summary could be reused with minor modifications, while text generation relies too heavily on language and library dependent information to be directly reused, and
- Qualitative analysis results that suggest that most of the necessary modifications for generating summaries for MPI C programs are based on changes needed in text generation and the underlying software word usage model going from Java to C, with straightforward modifications to handle MPI library calls accurately

b. A prototype comment generator for parallel MPI Java programs with the following main contributions:

- A technique for automatically identifying code statements appropriate as content for a given MPI Java method's summary comment,
- A strategy for generating natural language phrases for parallel constructs and the SPMD execution paradigm followed in a given MPI Java method, and
- Evaluation by manual analysis of the generated summary comments

1.2 Outline

The rest of the report is organized as follows: Chapter 2 gives background information on the overview of MPI and MPI specifically using Java. It also presents the state of the art of the techniques for documenting sequential programs and the tools for understanding parallel programs. Chapter 3 explains an existing summary generator for sequential Java in detail, the starting point for the approach. Chapter 4 describes our feasibility study of generating comments for MPI C programs reusing the techniques of the summary generator for sequential Java explained in Chapter 3. Chapter 5 details our approach to generate comments for MPI Java parallel programs, and initial evaluation observations of the developed prototype. Finally, Chapter 6 contains the conclusions of the work and future research directions.

Chapter 2

BACKGROUND

2.1 Overview of MPI

In the 1990s, the parallel programming community converged on two primary parallel programming standards — OpenMP for shared memory and MPI for message passing. The Message Passing Interface (MPI) is a portable message-passing standard that facilitates the development of parallel applications and libraries [14]. The standard defines the syntax and semantics of a core of library routines that enable developers to write a parallel program in a standard sequential language, such as C, C++, or Fortran, augmented with calls to the MPI library for process management, message passing, and other collective operations. Since the MPI programming model assumes distributed memory, MPI is typically used for PC clusters and distributed memory machines. MPI can also be used in conjunction with OpenMP for hybrid architectures.

2.1.1 Salient Features of MPI

To better illustrate the challenges in summary comment generation for MPI programs, this section highlights the salient features most frequently associated with MPI parallel programming.

2.1.1.1 SPMD Parallel Program Pattern

Most MPI programs follow the Single Process, Multiple Data (SPMD) pattern of parallel programming. As program execution begins, each process involved in the parallel execution receives an image of the whole MPI program and executes the same program at

their own speed. This is reflected in MPI programs by a call to MPI_Init near the start of the program to set up the MPI internal bookkeeping data structures for a common context between processes. Similarly, the program closes with a call to MPI_Finalize to clean up the shared context and shut down parallel computation.

A call to MPI_Comm_rank usually shortly follows the MPI_Init call so each process retrieves its unique id (often called the rank) from the internal MPI structures. If there are p processes executing an MPI program, they will have process ids 0, 1, ... , p-1. The process id is also used by the programmer to specify the source and destination of messages. This unique process id allows each process to take different paths through the single program in order to cause different behavior on different processes. The most common way to specify different paths of execution for individual processes is branching statements with conditional expressions using the process id, (e.g., `if (my_rank == 0)`). These are called *special conditionals* in this thesis to differentiate them from regular branching statements since they cause different behavior on different processes. Another common way to individualize process behavior is to use the process id in expressions to indicate parts of a data structure locally manipulated.

2.1.1.2 Process Groups and Communication Contexts

Almost every MPI library function includes a reference to a communicator, an MPI object that consists of a process group and communication context. Communicators enable programmers to divide processes into subgroups and control the ways the groups are allowed to communicate. Unless the programmer explicitly creates a new communicator, it is MPI_COMM_WORLD, the predefined communicator which includes all processes running the MPI program.

2.1.1.3 Point-to-Point Message Passing Communication

The most common form of communication in MPI programs is point-to-point, where a single process sends a message to another single process, which performs a

receive call to explicitly receive the message, in isolation of other processes. MPI offers the basic MPI_Send(data, count, MPI_type, source, dest, tag, communicator) and MPI_Receive(data, count, MPI_type, source, tag, communicator, status) operations with parameters to adequately match the desired paired process and send/receive the data. There are many variations of the send and receive operations that optimize the data transfer and how communication and computation overlap.

2.1.1.4 Collective Communication

MPI provides useful library functions to communicate data between many processes. All processes in the communicator specified as a parameter participate in the communication. These operations are very important in MPI. The most common collective operations are MPI_Barrier, MPI_Bcast, MPI_Reduce, MPI_Scatter and MPI_Gather.

2.1.2 Example MPI Program

Consider the MPI program in Listing 2.1 [25]. The call to MPI_Init at line 7 initializes the MPI environment. Each process retrieves its process id by calling MPI_Comm_rank, at line 8. To determine the number of processes involved in the communication, MPI_Comm_size is called at line 9.

The process with id 0 is acting as a manager process waiting to receive a message from each of the other p-1 processes and printing each message after it is received. It expects to receive the messages in the process id order. All other processes, often called worker processes, store a greetings message with their id into a message buffer and then send that message buffer to the manager process 0.

```
/* All processes except manager create and send
   greetings to the manager. The manager process
   receives message from all other processes and
   prints their message */

1  main(int argc, char* argv[]) {
2   int         my_rank;
3   int         p, source, dest;
4   int         tag = 0;
5   char     message[100];
6   MPI_Status  status;

7   MPI_Init(&argc, &argv);
8   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9   MPI_Comm_size(MPI_COMM_WORLD, &p);

10    if (my_rank != 0) {
11  sprintf(message, "Greetings from proc \%d!",
    my_rank);
12    dest = 0;
13  MPI_Send(message, strlen(message)+1, MPI_CHAR,
    dest, tag, MPI_COMM_WORLD);
14   } else {
15   for (source = 1; source < p; source++) {
16   MPI_Recv(message, 100, MPI_CHAR, source, tag,
     MPI_COMM_WORLD, &status);
17     printf("%s\n", message);
18    }
19   }
20   MPI_Finalize();
21  }
```

Listing 2.1: Example MPI Program

2.2 MPI for Java

As Java emerged as an important language in the mid '90s, there was an immediate interest in its possible uses for parallel computing, and a number of groups worked

towards MPI for Java. Bryan Carpenter's mpiJava [10] was one of the first attempts to use Java with MPI. Initially, the mpiJava project consisted of a set of JNI wrappers to a local C MPI library with the limitations of restricted portability. It had to be compiled against the specific MPI library being used. The later mpiJava projects adopted object-oriented concepts and provided an interface to the standard Message Passing Interface. Other efforts toward using Java with MPI include JavaMPI [1], automatic generation of wrappers to legacy MPI libraries; MPIJ [3], pure Java implementation of MPI closely based on the C++ binding; JMPI [2], implements Message Passing with Java's Remote Method Invocation (RMI) and Object Serialization; and MPJ Express [5], which is the most usable Java message passing library that can be executed in cluster and multicore configurations.

Though Java is a highly portable language and adheres to the philosophy of ”write once, run anywhere”, it does not have well-established scientific library bindings such as C/Fortran. Also, Java executional speeds are not suitable for HPC unlike C/Fortran. Some of the most challenging parts of Java/MPI arise from Java characteristics such as the lack of explicit pointers and the linear memory address space for its objects, which make transferring multidimensional arrays and complex objects inefficient. Hence Java is still not the most preferred language for MPI parallel programming.

2.3 State of the Art

The goal of this thesis is to develop an approach to automatically generate summary comments for functions in MPI parallel programs. This section describes the state of the art tools used for understanding parallel programs and the efforts toward comment generation for sequential programs. Beyond the summary generator for sequential Java methods [32] used as a basis in this thesis work, other efforts toward documenting sequential programs either target specific language features [8, 23], are semi-automated [15, 28, 29], or focus on studying the potential content for summaries [12, 18].

2.3.1 Tools for Understanding Parallel Programs

The demand for parallel programming coupled with the difficulties in reading and writing parallel programs prompted the development of tools for visualizing, debugging, teaching, and understanding parallel programs and their executions.

Kraemer et al. [38] made several contributions towards visualization of concurrent programs for understanding and debugging. Other visualization tools enable a programmer to visualize the execution trace of the parallel program [30, 31, 33]. Park et al. [26] developed a tool for visualizing race conditions in message passing programs. Carr et al. [11] developed the ThreadMentor tool as an aid to teaching multithreaded programming. Although useful for predicting application's behavior, none of the tools can be used to identify an overall summary of a given MPI function. Rilling et al. [27] present a program slicing algorithm for the comprehension of message passing programs. Program slicing could be useful in improving comment summaries.

2.3.2 Comment Generation for Sequential Programs

Buse and Weimer [9] automatically generate comments about exceptions thrown by a Java method. Long et al. [23] generate API function cross references. Neither are intended for generating descriptive summary comments. Haiduc et al. [17]'s case study of humans judging summaries generated by text retrieval techniques for Java methods and classes suggested that a combination of the techniques along with position information captures the content better than any individual text retrieval technique. Semi-automated approaches either automatically determine uncommented code segments and prompt developers to enter comments [15, 28], or automatically generate comments from high level abstractions which the programmer provides during development [29]. Although useful for newly created systems, none of the semi-automatic techniques apply to existing legacy systems.

Chapter 3

A SUMMARY GENERATOR FOR SEQUENTIAL JAVA

This thesis starts by exploring whether the existing sequential Java summary generator [32] is reusable to develop a summary generator for MPI parallel functions. Therefore, to make this report self-contained, this chapter briefly describes the base summary comment generator developed by Sridhara et al. [32] for Java, and illustrates the overall process in Figure 3.1. Given the signature and body of a method, the automatic comment generator (1) identifies the content, or statements, called *s_units*, to be included in the method summary, (2) lexicalizes and generates the natural language text to express the content, and (3) combines and smooths the generated text. Before any names can be analyzed for text generation, identifiers are split into individual words, abbreviations are expanded, and structural and linguistic program representations for a method are constructed. The comment generator uses information from the control flow graph, data and control dependences, and textual clues from the software word usage model (SWUM) [19] of the Java method. The summary comment generator for sequential Java methods, *genSumm*, and SWUM prototype were both implemented as Eclipse plug-ins.

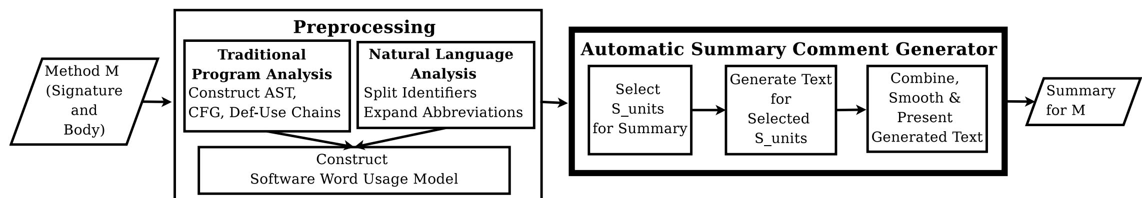

Figure 3.1: Overall Summary Comment Generation Process

The Software Word Usage Model (SWUM) provides the necessary linguistic information to both identify and express high level actions. SWUM not only captures the occurrences of words in code, but also their linguistic and structural relationships. SWUM is used to obtain the *action, theme*, and *optional secondary arguments* of a statement grouping to generate succinct and smooth descriptions.

Consider the example method signature `list.add(Item i)`, which can be captured by the phrase, "add item to list." The action is "add", the theme is "item" and the secondary argument is "(to) list". In this example, the location of the theme is the given parameter while the location of the secondary argument is the receiver object. In addition to these locations, a theme or secondary argument can be in the method name itself (e.g., `buildMenu()`). SWUM is built using naming conventions and linguistic knowledge gained from observations of thousands of Java programs.

3.1 Content Selection

The goal in content selection is to choose the important lines of code to be included in the method summary. An s_unit is the same as a Java statement, except when the statement is a control flow statement; then, the s_unit is the control flow expression with one of the `if`, `while`, `for` or `switch` keywords. Sridhara et al. [32] identified several characteristics that indicate an s_unit is a good candidate for a method's summary:

3.1.1 Ending s_units

- An ending s_unit lies at the control exit of a method. When a method returns a value, the ending s_units are the return statements; in a void return method, the ending s_unit is the last statement in the method.

3.1.2 Void-Return s_units

- An s_unit which has a method call that does not return a value or whose return value is not assigned to a variable is a void-return s_unit.

3.1.3 Same-Action s_units

- In a method M, if an s_unit has a method call c such that M and c have the same action, then the s_unit is called a same-action s_unit.

3.1.4 Data-Facilitating s_units

- Data-facilitating s_units assign data to variables used in s_units identified by the previous three heuristics.

3.1.5 Controlling s_units

- A controlling s_unit controls the execution of an s_unit previously selected by one of the earlier heuristics.

Some operations are less specific to a particular method's computational intent than to the overall program behavior (e.g., exception handling or logging when the sole intent of the method is different). Such s_units would add unnecessary information to a summary comment, and thus are identified and omitted from the summary.

A three-phase process selects the summary s_units for a method M. In the first phase, the same-action, ending, and void-return s_units are identified and added to the summary set. For each s_unit in the summary set, its data-facilitating s_units are added. Finally, the summary set is augmented with any controlling s_units. In each phase, the filter is applied to exclude ubiquitous operations along with s_unit identification.

3.2 Text Generation

The next phase is to convert each s_unit into a natural language phrase that can be understood independent of the s_unit's context within the method body. The basic s_unit is one with a single method call. The text generation strategy for a single method call is used in nested method calls, composed method calls, assignments, returns, conditional and loop expressions. The text generator first constructs subphrases for the arguments in an s_unit and then concatenates subphrases for the entire s_unit. In generating phrases for the

arguments (e.g., component, of view, to pane, of frame in the example here), the system strives to produce more descriptive phrases than contained within individual s_units. In the example, the system generates "drawing view" rather than "view". This addition of the modifier "drawing" to produce the more specific "drawing view" is accomplished via *variable lexicalization*.

```
Given: f.getContentPane().add(view.getComponent(), CENTER)
genSumm automatically generates:
/* Add component of drawing view to content pane of frame*/
```

The system did not include parameter `CENTER` in the generated text; it does not correspond to the *theme* or *secondary argument* of `add`. The text generator combines phrases when possible and removes words not needed for the summary (e.g., it dropped the 'get' in the generated text for `getContentPane()` and `getComponent()`).

One challenge in text generation is to construct noun phrases that represent variables. In English noun phrases, more specific noun modifiers appear on the left. For example, consider the variable `Document current`, which would be lexicalized as "current document". Sridhara et al. [32] developed a strategy for variable lexicalization, in which descriptive noun phrases describing variables are generated with modifiers extracted from type information.

Chapter 4

FEASIBILITY STUDY OF SUMMARY COMMENT GENERATION FOR MPI PROGRAMS

This chapter presents a preliminary study that we performed to investigate the reusability of Sridhara et al.'s [32] automatic comment generation technique designed for sequential Java programs to the problem of automatically generating descriptive summary comments for functions in parallel MPI programs.

4.1 Towards Summary Generation for MPI

The goal of this thesis in automatic summary comment generation for MPI parallel programs is similar to that of *genSumm* — to generate comments which are accurate and contain adequate content. That is, a summary comment should include all the important information required for program understanding, should be specific enough to be practically useful to the developer, and not overly general. A concise comment contains little to no redundancy and has no extra information, thereby not wasting the developers' valuable reading time.

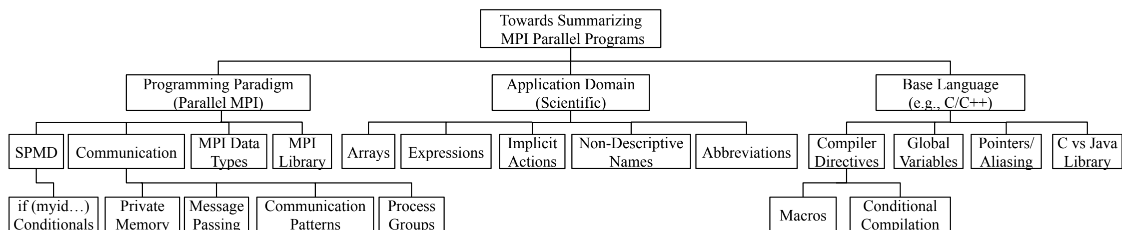

Figure 4.1: Classifying Factors Affecting MPI Comment Generation

The approach taken by this thesis to obtaining these goals was to explore how much of the existing strategies for content selection and text generation as well as the SWUM representation itself could be leveraged from *genSumm* to develop a summary generator for MPI parallel functions. The first step towards this approach was to delineate the major factors that could potentially impact the applicability of the existing techniques in the MPI parallel programming environment. These factors are used in the study presented here to explain the observations made and guide in proposing the modifications to the sequential Java approach.

Figure 4.1 shows the classification of the major factors potentially affecting the transfer of technology. The factors can be broadly categorized under (1) the MPI parallel programming paradigm, (2) the predominant scientific domain of applications written in MPI, and (3) the base sequential language of the MPI program. Each of the factors are discussed briefly here.

4.1.1 MPI Parallel Programming Paradigm

In addition to thinking about approaching a problem from a perspective of parallel execution of processes, creating a parallel program involves several steps that are not required for sequential programming, namely, task decomposition (how is the task going to be partitioned for multiple processes), mapping (which process is going to do what task), data distribution (which data does each process work on and how does it get that data), communication, and coordination between processes working on their tasks. Part of understanding an MPI parallel program is identifying how the developer has chosen to accomplish each of these functions of the parallel program.

An automatic comment generator needs to be able to capture an overall summary of what tasks are being performed, which will include data distribution, parallel tasks, communication, and coordination. A key indicator of which processes are performing which tasks is the use of the process id especially in conditionals. The parameters in the

MPI library calls determine which processes are involved in communications. The comment generator can take advantage of the known semantics of the MPI library routines.

On the collected set of 32 MPI parallel programs, it was noted that 28% of the 6143 functions contain at least one MPI library call. Thus, at least 2/3 of the functions should be able to reuse comment generation techniques for sequential programs without concern for the parallel programming paradigm factors.

4.1.2 Predominant Application Domain

Science and engineering is becoming increasingly focused on a "simulate and analyze" approach, which has been aided by computing hardware and software systems that enable large-scale computations and numerical experiments with more realistic modeling and practical performance. For cost effective networked clusters and other distributed memory architectures, scientific computing experts have been using MPI due to its portability, scalability, and efficiency. Thus, a large proportion of existing MPI parallel programs are scientific-oriented.

Scientific codes have some common characteristics. They typically involve modeling and simulation of some phenomena. This often involves computing mathematical equations, time-dependent steps, and manipulation of large amounts of data.

In the collected set of 32 open-source MPI parallel programs, simple camel-case splitting was applied to the 627,597 identifiers, which resulted in 861,144 "word occurrences", which was then compared against an online dictionary [20]. 484,294 (56%) of the word occurrences were determined to be non-dictionary words. To compare with Java, a study of 7 open-source programs showed only 23% of word instances were non-dictionary words. This can be attributed to the scientific domain where single-letter variables and abbreviations are more common. For program understanding and automatic summary generation, this implies a need to extract actions and themes at a higher level of abstraction than individual equations and in the presence of more abbreviations and less intuitive names than other domains.

4.1.3 Base Sequential Language

Most MPI parallel programs in existence today are written primarily in Fortran and C. In this study, the focus was on a single base language. C was chosen because it is found to be very prevalent in the available open source MPI programs. The features of the base language C that are believed to differ the most from Java, from the perspective of a newcomer trying to understand a program and the task of an automatic comment generator are highlighted.

C programs often take advantage of a C preprocessor, which enables the programmer to include macro definitions (`#define`) and conditional compilation (`#if`) among other directives. With macros, the source code of the function contains the macro name at each use, while the preprocessor will expand that name to its intended sequence of tokens. The programmer may have both the macro name and its expansion at hand when reading the code. It is reasonable that the comment generator may use either or both the macro name and its replacement, when analyzing a statement using a macro. Sometimes, the macro name may be more revealing of the intended action of the statement than the details of the replacement code, which may muddle the overall high level action. Conditional compilation implies that some of the source code in a function may never appear in the executable code. Both programmers and an automatic comment generator need to address the issue of how to handle source code that is not always included in the executable code.

Most programmers would agree that using global variables makes software harder to read and understand. Since any code anywhere in the program can change the value of the variable at any time, understanding the use of the variable may entail understanding a large portion of the program. An automatic comment generator also has to address the use of global variables in a function being summarized.

Similarly, while pointers are a powerful feature of C, they make it difficult to determine which locations are actually being defined or used at a given program point

because a given object may be referenced by more than one name (i.e., aliased). The summary generator needs to be able to determine the theme of an action in the presence of C pointers and aliasing. Lastly, a minor challenge in applying a Java comment generator for C is the different libraries that are used.

4.2 Study Design

The study was designed to explore whether the Sridhara et al.'s [32] automatic summary generator for sequential Java programs, *genSumm*, could be applied for generating summary comments for functions in MPI parallel programs. Specifically, the designed empirical study seeks to answer the following questions:

Content Selection: *How well does the genSumm s_unit selection work for selecting content for functions (containing MPI library calls) in an MPI parallel program? In particular, how well does it identify adequate content for a function summary, and how well does it avoid selecting content irrelevant to a summary?*

Text Generation: *How well does the genSumm text generation technique and underlying SWUM work for generating concise phrases that summarize functions in an MPI parallel program? Does the text generation result in the loss of important content inherent in the original s_unit?*

Perceived Causes: *When genSumm fails for MPI parallel programs, is it due to switching from the sequential to the parallel programming paradigm, the predominantly scientific application domain of MPI programs, differences in the language (C versus Java), or some other potential factor?*

4.2.1 Subject MPI Functions

The goal of the study was to gain insights into summary generation for MPI functions. To reach this goal, we wanted to learn from functions that use many different

common MPI library calls and parallel programming patterns without demanding a lot of human time performing manual simulation. 32 open source MPI parallel programs were collected from open source repositories (e.g., Netlib, SourceForge), high performance computing application sites (e.g., DEISA, HPCC) and individual scientists. The programs come from a variety of scientific domains and range from 400 LOC to 200,000 LOC.

Applications	**Description**	**NLOC**	**# Funcs With MPI Calls**
AMROC	Fluid solver simulator	10078	32
BlobFlow v2.02	Viscous flow simulator	3855	14
BlobFlow v3.01	Viscous flow simulator	6306	13
mdtest	Metadata evaluator	739	8
mpiBLAST	Parallel implementation of NCBI BLAST	14026	79
MPIPython	MPI integrated Python interpreter	3720	98
104.milc	Quantum Chromodynamics simulator	11496	39
BMMC	C library for Bit Matrix Multiply	1336	14
GADGET-2	Cosmological simulator	12509	41
MPI Grid Ping	Cluster Network Utility	811	17
MPI Ruby	Ruby binding of MPI	3378	141
mpiGL	openGL API	952	6
Samudra-Manthan	Information Retrieval project	1510	13
STAR-MPI	MPI collective communication routines	13830	194

Table 4.1: Characteristics of Programs Containing Subject Functions

From this program suite, the subset of functions that contain at least one MPI communication call and between 30-150 LOC was determined, to ensure that medium size functions that are reasonable for humans were analyzed to manually simulate the

comment generator. Then 22 functions were randomly chosen such that the commonly used MPI library routines were covered and selected from different programs. To provide some context for the selected functions, Table 4.1 shows the description, number of non-commented LOC and number of functions with MPI calls in each of the 13 MPI parallel programs containing the 22 functions under study.

4.2.2 Procedure

To answer the research questions, a three-step procedure was followed:

Step 1: Manually wrote expected summary comments. Two computer science graduate students independently read each of the 22 functions (including any internal comments) to understand their main intent and then wrote a summary comment for each function. They discussed their summaries and came to agreement on final summaries, which we call the *expected summary comment* for each function. The humans did not have knowledge of the algorithm for summary comment generation in Java.

Step 2: Simulated *genSumm* automatic summary comment generation. Because SWUM and *genSumm* focus on Java, three humans each individually simulated the execution of the s_unit selection phase without modification on the 22 functions from the MPI programs. They then discussed their results and agreed on the final simulated s_unit selection output.

To separate the effects of s_unit selection and text generation, focus was on the phrases generated for individual selected s_units, not the whole natural language summary. In fact, it was found that a whole summary in natural language can not be generated by using the Java summary generator without significant modification for MPI and C, because the templates used for text generation are dependent on the programming language. The text generation templates for different kinds of statements (expressions, assignment, conditional, looping,...) are not directly transferable to the C language. Thus, the text generation study was focused on how well the Java summary generator's text generation and use of SWUM for method calls could be applied to generating text for C function calls

appearing in selected s_units. If that is not reusable, then the text generation approach needs to be completely replaced.

The basic s_unit in Java methods is one with a single method call. The method call and that method's header are input to SWUM, which provides the *action, theme*, and *secondary arguments* of the method call. For each selected s_unit, any function calls were translated into Java for the purpose of simulation since SWUM is focused on Java. SWUM was then used to obtain the action, theme, and secondary arguments of the C function calls. For MPI library calls appearing in selected s_units, mpiJava [10], which is a Java interface to the standard MPI was used. Given the action, theme, and argument information for a function F, the text generation was then manually simulated by using the text generation template for method calls defined for Java:

action theme secondary-args
and get return-type [if F returns a value]

Step 3: Analyzed automatically generated comments using expected summary comments. The results produced by the simulated *genSumm* were analyzed separately for content selection and text generation, in comparison with the expected summary comments. The number of s_units that were correctly identified by *genSumm* were counted, and also the number of s_units identified that are deemed irrelevant for a summary, based on comparison with the manually written summaries. Also the number of s_units that were missing from the summary generator's output, but included in the manually written summaries were counted. At the function level, the functions for which the summary generator gave no irrelevant s_units were counted, and also the number of functions where it did not miss any relevant s_units. Each irrelevant and missing s_units were categorized by the perceived cause, namely going from sequential to the parallel programming paradigm, inherent characteristics observed in the scientific application domain, or differences between Java and the base language C.

For text generation output for C function calls in selected s_units, all the phrases were analyzed to identify any relevant content for the summary that was in the original function call statement, but lost only during phrase generation. The observations of the perceived causes when content was missing during phrase generation were categorized.

4.2.3 Threats to Validity

Since there were only a few developer written comments with which to compare the automatically generated comments, two humans, reading the code wrote the summary comments to serve as expected comment summaries. However, they wrote the comments without knowledge of how the comment generator works and came to agreement on the function's main intent and final summary comment. Since the automatic comment generator is developed for Java, humans simulated its execution on MPI parallel C programs; however, three humans agreed on the simulated output. The results obtained may not generalize to other MPI programs, base computation languages or parallel programming paradigms. To mitigate this threat, open-source programs were chosen across different problem domains representative of MPI programming.

4.3 Results and Discussion

4.3.1 Content Selection

In general, the results showed that the s_unit selection strategy of *genSumm* serves as a promising strategy for selecting content for function summaries for MPI parallel programs written in C. Table 4.2 presents the results of the analysis of the missing and irrelevant s_units selected by *genSumm* when applied to the 22 C functions containing MPI library calls. Of the 249 s_units manually selected by humans, simulated *genSumm* missed only a total of 21 s_units (8%). Furthermore, simulated *genSumm* did not miss any s_units in 15 of the 22 functions (68%).

Analysis of the 21 missed s_units indicates that almost half of the missed s_units (9 of 21) either were data facilitators for or contained MPI library calls that returned an

error code that was then assigned to a variable such that the statement did not get chosen as any of the kinds of s_units (specifically not void_return s_units). Data facilitators were not chosen because the statement they were facilitating was not selected. Currently, special conditionals e.g., (`if my_rank == 0`) are being selected only when they are controlling s_units for other already selected s_units. All of the 8 missed s_units categorized as due to Java versus C involved #ifdef conditional compilation statements, because *genSumm* did not know how to handle this construct. For example, in the following code snippet, the #ifdef acts like a controlling statement for the MPI library call, which was selected as an s_unit, thus the #ifdef should also be selected to present the main intent in the summary:

```
 #ifdef topo_aware  /*s_unit missed*/
MPI_Comm_Compare (comm,MPI_comm_world,&i); /* void return s_unit selected */
```

The last 4 missed s_units were assignment statements involving a mathematical computation with arrays and no function calls, characteristic of scientific applications. *genSumm* has no rules for selecting such statements as s_units for the summary as it is focused on method calls, which are very common in Java.

The results for generating irrelevant s_units were slightly worse; 68 of the 296 automatically generated s_units (23%) were deemed irrelevant for the function summary. For 10 of the 22 functions (45%), the summary generator selected no irrelevant s_units. A large percentage,(80%), of the 68 irrelevant s_units was due to Java versus C differences.

54 of the 68 selected s_units deemed to be irrelevant for a summary were categorized as caused by the Java versus C base language. The majority of those were selected as void-return s_units because they contained a C function call with no value returned; however, they would be considered ubiquitous operations, not specific to the function's computational intent, (e.g., `printf, scanf, malloc, free, strcpy, fclose`). For example, in the code snippet below, the `qsort` function call is correctly identified as

Content Adequacy			**Conciseness**		
	#funcs w/ at least 1 s_unit missed	**#s_units**		**#funcs w/ at least 1 irrelevant s_unit**	**#s_units**
Due to: Language	3	8	**Due to:** Language	10	54
Paradigm	3	9	Paradigm	4	14
Domain	1	4	Domain	0	0
Total Missed	**7**	**21/249**	**Total Irrelevant**	**14**	**68/296**
0 s_units Missing	**#funcs** 15	-	**0 s_units Irrelevant**	**#funcs** 10	-

Table 4.2: Results from Simulating Automatic Content Selection
(249 manually identified s_units, 296 automatically selected s_sunits)

an s_unit, and `toplist` is the theme for the function `qsort`, for which the data facilitating s_unit is selected. However, the data facilitating s_unit defines the variable using the C function `malloc`, which does not need to be explicitly mentioned in a succinct summary.

```
toplist = malloc (ntop * sizeof (struct topnode\_exchange));
qsort (toplist, ntop, sizeof(struct topnode\_exchange),domain\_compare\_toplist);
```

Similarly, the macro DEBUG was chosen but is essentially logging and not important in the summary. Simple initialization statements, especially involving array elements, were unnecessarily selected due to serving as data facilitators to s_units.

It was noted that all 14 cases where the irrelevant s_unit involved an MPI construct are ubiquitous operations and should not appear in the summary. For example, MPI_Init and MPI_Finalize occur in every MPI program, while others (e.g., MPI_Pack_Size, MPI_Success and MPI_Type_Size) are required for MPI data types, preparation for communication, or error processing, but not important in a summary. Overall, the characteristics of the scientific application domain had little impact on missing or irrelevant s_units.

4.3.2 Text Generation

Of the 296 s_units selected over the 22 functions in the MPI parallel programs used for this study, 153 s_units (52%) contained at least one function call (some contained more than one call). The text generation study was focused only on these s_units because *genSumm* does not have templates for statements with only infix expressions. Of the 185 function calls that appear in selected s_units, 117 are calls to C functions while 68 are calls to the MPI library.

Text generation for function calls involves obtaining the action, theme, and secondary arguments from SWUM, and then using the phrase generation template. SWUM's heuristic chooses the first word in a function name as the action based on the object-oriented programming language paradigm where classes contain methods to perform actions on the objects, and method names by convention commonly start with a verb indicating the method's action. Consider the Java call mylist.add("element1"). This adheres to convention. SWUM identifies the action as add, theme as element1, and secondary argument as mylist, with generated phrase "add element1 to mylist." SWUM also usually checks the part of speech of words in the method call, and upon finding a verb, uses that as the action. Due to issues with applying part-of-speech taggers to source code, SWUM uses "potential part-of-speech" rather than an actual part-of-speech tagger.

Consider a comparable C function call — list_add(mylist, "element1"). List has a verb sense. Due to names like list_add in C as opposed to list.add in Java, action identification becomes difficult. Once action identification is difficult, determining the theme and secondary arguments correctly also becomes tougher. Finally, in object-oriented classes, the class (received object) usually serves as the secondary argument, but in procedural programs, it is not so straightforward to determine the secondary arguments.

To generate text for MPI library calls in selected s_units, firstly the MPI prefix was removed so SWUM does not choose MPI as the action. Most of the actions and themes are then identified correctly, because the MPI library functions mostly start with a

verb (e.g., Send, Recv, Bcast, Scatter, Gather). The secondary arguments are incorrectly identified, due to the lack of a class structure and the difficulty in determining which parameter would serve as a secondary argument. This also causes the information about sending and receiving processes in communication calls to be lost. In addition, many of the communication calls include single letter abbreviations such as S, B, and I, which make it difficult to discern meaning.

4.4 Implications for MPI Comment Generation

The low percentages of missed and unnecessary s_units suggest that a comment generator for MPI can reuse *genSumm*'s content selection with minor modifications and obtain high accuracy. In particular, the missed s_units can be significantly reduced by treating #ifdefs as regular C conditionals and including them as s_units when they control other s_units, and also developing content selection rules for selecting expressions that involve array operations in computations with no function calls. Unnecessary s_units can be avoided by simply identifying the set of ubiquitous MPI library calls, C library calls, and other C-related operations that should be filtered from the selected s_units for MPI parallel programs. The data facilitator and void-return s_unit definitions also can be refined for MPI C functions.

Unlike content selection, the *genSumm*'s text generation requires more significant analysis and modifications to generate accurate phrases for the selected s_units in MPI C functions. MPI function calls could be handled easily by predefining templates based on the known semantics of the predefined library and information known about its parameters. These templates could include information about communicators and sending and receiving process identification. Text generation templates for infix expressions need to be developed for C. To handle C function calls, SWUM needs to be customized to the C language to obtain action, theme, and secondary arguments for imperative programs without classes, thus improving analysis of parameters and their roles.

Chapter 5

GENERATING COMMENTS FOR PARALLEL MPI JAVA PROGRAMS

The feasibility study presented in the previous chapter showed that the base language, especially C, can hamper efforts in comment generation more than the parallel nature of the code and MPI library calls. Though Java is not the most frequent choice as the base language for MPI codes, as a first prototype, in this work, the base language factor was eliminated, and we concentrated on comment generation for Java MPI programs which helped to focus on the challenges of comment generation for parallel codes.

5.1 Summary Generator for MPI Java

This section presents *genSummMPI*, a prototype of summary generator to generate summary comments for MPI Java parallel programs. *genSummMPI* is built as an extension to *genSumm* and is implemented as an eclipse plug-in. The remainder of this section describes the parts transferable from *genSumm*, and the modifications and extensions to *genSumm* to build the summary generator for MPI Java, based on what was learned from the previous study.

5.1.1 Content Selection

5.1.1.1 Transferable

The basic content selection algorithm which identifies the ending, void-return, same-action, data-facilitating, and controlling s_units is reused to select the important non-communication statements executed by individual processes.

5.1.1.2 Modifications and Extensions

Selecting MPI-specific Content. New heuristics were developed to ensure that all MPI library calls are selected as s_units, since they indicate communication and interaction between processes.

Consider the MPI library calls:

```
call1: MPI.COMM\_WORLD.Recv(message, 0, 100, MPI.CHAR, source, tag)
call2: S = MPI.COMM\_WORLD.Recv(message, 0, 100, MPI.CHAR, source, tag)
```

genSumm selects `call1` as a void return s_unit, whereas `call2` is not selected as it does not satisfy any of the rules mentioned for the s_unit selection by *genSumm*. *genSummMPI* ensures that assignment MPI calls such as `call2` are also selected for the summary.

When an s_unit within a branch of a conditional is selected, *genSumm* automatically selects the controlling "if" expression. In a code snippet such as,

```
if(my_rank == 0)
{
 System.out.println("Group Range TEST COMPLETE\n");
}
```

genSumm filters out the print statement as it considers it to be a ubiquitous statement. However, the "if" is a special conditional in MPI. *genSummMPI* ensures that all process-controlling "if" statements indicating process execution paths (e.g., if myrank == 0) are always selected.

The listings, 5.1 and 5.2, respectively, show the list of s_units selected by the content selection algorithm of *genSumm* and *genSummMPI* for an example MPI method. The listings demonstrate how the meaning of the summary could be altered if the corresponding "else" is not selected along with the conditional "if" statement.

```
16 if (rank != 0) {
19 MPI.COMM_WORLD.Send(message,rank,4,MPI.INT,0,99);
23 for (source=1; source < size; source++) {
25 MPI.COMM_WORLD.Recv(message,0,10,MPI.INT,source,99);
```

```
Summary Generated: each worker, send message to the manager. from all workers, receive
    from source and store in message.
```

Listing 5.1: Without "else" Being Selected - by *genSumm*

```
16 if (rank != 0) {
19 MPI.COMM_WORLD.Send(message,rank,4,MPI.INT,0,99);
22 else
23 for (source=1; source < size; source++) {
25 MPI.COMM_WORLD.Recv(message,0,10,MPI.INT,source,99);

Summary Generated: each worker, send message to the manager. manager, from all workers,
    receive and store in message.
```

Listing 5.2: With "else" Being Selected - by *genSummMPI*

genSummMPI ensures that along with all of the process-controlling "if" statements, their corresponding "else" branches are also identified and selected. This requires recognizing the process id variable returned by the MPI.Comm_Rank call and tracing it to conditional expressions using def-use chains [6]. Listing 5.3 show examples of special conditionals automatically detected and selected by *genSummMPI*.

```
rank/ myself/ me/ myrank/ my_pe = MPI.COMM_WORLD.Rank();
.
.
if(rank/ myself/ me/ myrank/ my_pe == 0)
.
.
else
```

Listing 5.3: Examples of Special Conditionals Selected by *genSummMPI*

Filtering out Ubiquitous MPI Calls. MPI statements that are less specific to a method's computational intent than to the overall program behavior should be omitted from the summary. The ubiquitous s_unit filter list in *genSumm* is extended to include such MPI calls. The MPI calls Init, Finalize, Rank, Size, Wtime, and Barrier are included on the ubiquitous s_unit filter list.

In summary, changes made to *genSumm*ś content selection algorithm include both adding more content specific to MPI, and filtering ubiquitous MPI related statements.

5.1.2 Text Generation

5.1.2.1 Transferable

Based on the results of the study, the *genSumm* text generator is reused for all selected s_units not involving an MPI library call or process-controlling "if" conditional.

5.1.2.2 Modifications and Extensions

Custom Text Generation Templates for each MPI Library Routine. Text generation templates were manually created for each MPI library routine (49 in total, in our prototype, as listed in Appendix A) based on their defined semantics. The goal was to automate the template creation from documentation available at MPI specification web sites, but we found that the available documentation was not in a form conducive to automatically creating templates useful for program understanding. The important parameters were identified in every MPI routine and used as placeholders in the template. During text generation, the appropriate template is applied, actual arguments are identified in the MPI call and mapped to the template to replace the appropriate place holders as shown in Listing 5.4.

```
Method signature: Rsend (java.lang.Object buf, int offset, int count, Datatype type, int
     dest, int tag)
Template: Send <argument-0> to process <argument-4> in ready mode
Example call: MPI.COMM_WORLD.Rsend(buf,0,10,MPI.CHAR,1,1)
Text generated: Send buf to process 1 in ready mode
```

Listing 5.4: Example Template, Method Call and Corresponding Generated Text for Rsend

Custom Templates for MPI Constants. A Constant-Phrase table was developed to map MPI constants to meaningful phrases to be generated. For example, MPI.TAG_UB maps

to *largest tag value*, MPI.INT maps to *integer*, MPI.LOR maps to *logical or*. Listing 5.5 shows an example of the MPI constants in a generated phrase. Appendix B contains the complete list of the templates for constants in our prototype. Similarly, the numeric constant 0 in the library call parameter positions that is used to identify a process i.e., the 'root' process, maps to generated text *the manager* as demonstrated in Listing 5.6. This differentiates the manager process from the rest of the processes and makes these phrases more accurate in MPI context.

```
Example call: MPI.COMM_WORLD.Attr_get(MPI.TAG_UB)
Template: get attribute value of <argument-0>
Text generated without constant phrase: get attribute value of MPI.TAG_UB
Text generated using constant phrase: get attribute value of largest tag value
```

Listing 5.5: Example use of Constant Phrase

```
Example call: MPI.COMM_WORLD.Send(comm,0,4,MPI.INT,0,99);
Template: send <argument-2> <argument-3> starting at <argument-0> to process <argument
    -4>
Text generated without constant phrase: send 4 integers starting at comm to process 0
Text generated using constant phrase: send 4 integers starting at comm to the manager
```

Listing 5.6: Numeric Constant 0 Identified and Generated as a Special Case

Custom Text Generation for Process-Control Expressions. In process-controlling "if" and "else" statements, the conditional expressions are analyzed and text is generated in terms of "manager" or "worker" when analysis can determine these uses. For the "if" statement `if(rank==0)`, where rank is a unique id assigned to every process in a group and identified by the Comm_Rank call, the generated phrase is *the manager:* and for its corresponding `else`, the generated phrase is *all others*. For the condition `if(rank!=0)`, the generated phrase is *each worker*, and for its corresponding `else`, the generated phrase is *manager*.

genSumm can identify the "for" statements as controlling s_units, but does not generate any text phrase for the summary. The "for" statements that contain MPI library calls

or are otherwise executed differently depending on the process id are detected by *genSummMPI* as they add important content to the summary. For the example "for" statement in Listing 5.7, if text is not generated for the `for` statement, then it would appear as if that the manager process receives a message from a single process. So, the common loop patterns used in communication were mined and we developed techniques to automatically identify these patterns. Text generation templates for generating informative phrases such as *for all processes* and *from each worker* were developed. [Appendix reference] lists the loop patterns that were recognized and the generated phrases.

```
for (int j=1; j < nprocs; ++j) {
s=MPI.COMM_WORLD.Recv(comm,0,4,MPI.INT,j,99);
```

Listing 5.7: Example "for" Loop

In summary, the modifications made to *genSumm*ś text generation techniques to handle MPI Java programs include creating text generation templates for each MPI library routine, creating a constant-phrase table to map MPI constants to meaningful phrases to be generated, and recognition of process control expressions and corresponding custom text generation for these common patterns occuring in "if", "else", and loop condition expressions.

5.2 Initial Evaluation of *genSummMPI*

To evaluate, a suite of open-source Java MPI programs was collected and the developed prototype summary comment generator, *genSummMPI*, was applied to methods containing MPI communication calls. The generated summaries were manually analyzed for accuracy in portraying the main intent of the method with its parallel characteristics, and smoothness of phrases.

5.2.1 Gathering and Selecting Benchmarks for Study

The MPI Java programs used for the analysis were gathered from the following sources:

a. IBM MPI test suite - These are the benchmarks, mostly Unit tests created by IBM (by translating the IBM MPI test suite to suite Java) to evaluate the performance of mpi-Java [10]. The test suite has three categorizations: cost of thread safety test, concurrent progress test, and computation/ communications test. There are 16 test programs, of which only 10 programs could be compiled and used as subject programs for evaluation. These programs range between 68 and 2177 NLOC and are listed in Table 5.1(labeled by 'a' in source column) where, '# Methods' gives the count of the total number of methods present in the program, '# MPI methods' gives the count of the methods containing at least one MPI call, and '# MPI calls' gives the count of the total number of MPI calls present in the program.

b. MPJ Benchmarks - These are the benchmarks designed by the Java Grande Forum to provide performance comparisons between Java execution environments, specifically for parallel execution on distributed memory multiprocessors [4]. The benchmark suite is divided into three sections: low level operations - measuring the performance of low level operations such as ping-pong, barriers and collective communications, kernels - short codes which carry out specific operations frequently used in Grande applications, and large scale applications - larger codes, representing complete Grande applications. The benchmarks range between 1130 and 2714 NLOC and are listed in Table 5.1(labeled by 'b' in source column).

c. The last set includes a few examples from the course webpage of the School of Computer Science, Cardiff University, United Kingdom [36]. There are 4 example programs between 26 to 688 NLOC. The programs are listed in Table 5.1(labeled by 'c' in source column).

[7] presents the Java version of Gadget-2, which is a massively parallel structure formation code for cosmological simulations. The aim of this project has been to illustrate that Java is able to support high performance computing. The Java version of Gadget-2 is the first real scientific application in MPI using Java and the authors are still working on

it to improve its performance and thus they have not released their source code.

Benchmarks	Source	NLOC	# Methods	# MPI Methods	# MPI Calls
Nozzle	a	2177	56	23	24
PingPong	a	113	4	1	14
potts	a	1192	137	9	11
simple	a	177	3	3	15
ccl	a	700	15	15	105
comm	a	190	6	6	30
env	a	68	4	4	21
group	a	176	2	2	12
signals	a	355	28	11	10
topo	a	350	5	5	29
MPJ section1	b	1130	64	44	99
MPJ section2	b	2063	115	58	108
MPJ section3	b	2714	219	33	47
Integrate	c	51	1	1	6
LaplaceEquation	c	688	32	1	11
VibratingString	c	611	32	1	11
Greetings	c	26	1	1	5

Table 5.1: Characteristics of Java MPI Benchmarks

5.2.2 Challenges in Evaluation

Most of the gathered benchmarks are simple programs demonstrating communication between processes using MPI. These benchmarks lack computational statements of scientific codes unlike real MPI programs in C or Fortran. Most of the methods in these benchmarks are too big for evaluation. Also, most of the statements in these methods are print statements that are filtered as ubiquitous statements by *genSumm*. The gathered MPI programs also contain some assignment/computational statements which require better handling of both content selection and text generation by *genSummMPI*.

5.2.3 Analysis and Discussion of Generated Comments

Although we lacked an adequate number of real applications to conduct a thorough study, we performed an observation study of the generated comments for those cases that lent themselves to analysis. We selected examples based on examining the generated comments and determining whether they provided evidence for success or room for improvement.

5.2.3.1 *genSummMPI* Successes

Listing 5.8 shows the generated summary comment along with the chosen s_units (statements marked 'S'), for a method in the Greetings benchmark containing MPI calls, process-controlling conditional and a loop used to command the manager process to receive a message from each of the other processes in order. Listed below are the features of *genSummMPI* that work well:

a. Selects all the MPI library calls(line 16 and 21) that add important content to the summary and filters out those MPI calls(line 8,9, and 24) that are ubiquitous.

b. Ensures that the process-controlling "if" conditional on line 11 is selected along with its corresponding "else" on line 18.

c. Text is generated for all the selected MPI calls by using an appropriate text generation template.

d. Smoothing of the text for the MPI calls using the Constant-Phrase table is evident in line 21, where MPI.CHAR is translated to *characters*, plural of the datatype used, since the count is greater than 1.

e. Process-controlling "if" and "else" conditionals are identified and text is generated in terms of "manager/ worker" as *each worker* and *manager*, respectively.

f. Unlike *genSumm*, text is generated for the "for" statement in line 19. The "for" loop is recognized to contain MPI library calls as the condition of the loop is checked against the

variable returned by MPI.COMM_WORLD.Size() and a more informative phrase such as *from all workers* is generated by analyzing the loop pattern.

```
From benchmark: Greetings
Automatically Generated Summary:
each worker, message=str.toCharArray();  [default], send message to process dest.
manager, from all workers, receive 100 characters and store in message.

1 public static void main(String[] args) throws MPIException {
2   char message[] = new char[100];
3   int source, dest;
4   int size, rank;
5   int tag = 0;
6
7   MPI.Init(args);
8   rank = MPI.COMM_WORLD.Rank();
9   size = MPI.COMM_WORLD.Size();
10
11 S  if(rank != 0)
12   {
13    String str = "Greetings from process " + rank;
14 S   message = str.toCharArray();
15             dest = 0;
16 S   MPI.COMM_WORLD.Send(message, 0, message.length + 1, MPI.CHAR, dest, tag);
17   }
18 S  else{
19 S   for(source = 1; source < size; source++)
20    {
21 S    MPI.COMM_WORLD.Recv(message, 0, 100, MPI.CHAR, source, tag);
22      System.out.println("Message: " + message.toString());
23   } }
24  MPI.Finalize();
25   }
```

Listing 5.8: MPI Java Method with Automatically Generated Summary

5.2.3.2 Towards Improving the Precision of *genSummMPI*

Handling of Computational Statements. Because *genSumm* focuses on object-oriented programming and method calls, it has no defined rules to select the computational statements which mostly are assignment statements involving arrays and mathematical calculations as shown in Listing 5.9, where S denotes the selected s_units.

```
From benchmark: Integrate
24      double psum = 0.0;
25      for(int i=nbeg;i<=nend;i++){
26         psum += (Math.sin((i-0.5)*delta))*delta;
27      }
28      double [] dval = new double[2];
29      dval[0] = psum;
30  S    MPI.COMM_WORLD.Reduce(dval,0,dval,1,1,MPI.DOUBLE,MPI.SUM,0);
```

Listing 5.9: Code Snippet Showing Computational Statements not Selected by *genSumm*

genSummMPI needs to be able to identify important computational statements with respect to method's main intent, especially prominent in scientific codes of MPI along with generating a higher level abstraction for such computational statements.

Also, *genSummMPI* needs to be able to identify and generate phrases for assignment statements. For example, for the assignment statement at line 14 of Listing 5.8, *genSumm* generates "message=str.toCharArray(); [default]". A better phrase would be "create message".

Loops Including MPI Calls. All patterns of "for" loops containing MPI calls need to have smooth generating text phrases. *genSummMPI* handles only a few cases and generates a default phrase for the rest of the patterns. An example of the default phrase generated for a complicated pattern of "for" loop is shown in Listing 5.10.

```
From benchmark: MPJ section1
For statement: for(l=0;l<MLOOPSIZE;l++)
Phrase generated: forall l from 0 to MLOOPSIZE
A better phrase to generate: loop over number of different message sizes
```

Listing 5.10: "for" Loop - Default Phrase

Handling of loops is too low level especially when nested. Higher level recognition of high level actions is required. An example of nested loops is shown in Listing 5.11.

```
From benchmark: Nozzle
169     /* calculate RMS Pressure Error. */
170     error = 0.0;
171     for (i = 1; i < imax; ++i)
172       for (j = 1; j < jmax; ++j) {
173  scrap = pg[i][j] - opg[i][j];
174  error += scrap*scrap;
175       }
```

Listing 5.11: Code Snippet Demonstrating Nested "for" Loops

Leveraging Data Flow Information. Text generated for some MPI calls requires data flow information to improve the quality of phrase.

(i) Instead of the currently generated text phrase *send arr_obj.length integers to the leader*, it would be better to generate a phrase like *send 40 integers to the leader* where, length of arr_obj is 40.

(ii) As mentioned earlier, the Constant-Phrase table maps predefined operation constants such as MPI.SUM, MPI.PROD, etc. to meaningful phrases such as sum and product, respectively. For a MPI call having Op as a field as shown in Listing 5.12, better text generation is required when the operation performed is not a predefined constant.

```
Method signature: Reduce (java.lang.Object sendbuf, int sendoffset, java.lang.Object
    recvbuf, int recvoffset, int count, Datatype datatype, Op op, int root)
Template: Combine elements in <arg-0> of each process using the <arg-6> operation, and
    store result in <arg-2> of the process <arg-7>
Example call: MPI.COMM_WORLD.Reduce(tmp_checksum,0,tmp_checksum,0,1,MPI.DOUBLE,op,0)
Text generated: Combine elements in tmp_checksum of each process using the op operation,
     and store result in tmp_checksum of the leader
```

Listing 5.12: Example Template for Reduce

Both situations require analyzing previous statements using backward data flow analysis to learn more about arr_obj.length and Op, respectively to generate more precise text.

(iii) Data facilitators are not always chosen for the variables representing theme or secondary arguments in the MPI library calls, resulting in missing content in the summary. For example, in the code snippet 5.13, 'sndbuf' is the theme variable of the MPI call 'Send'; this goes undetected losing the information on what is being sent by the 'Send' call. (The s_units chosen for this code snippet are marked "S".)

```
S  else {
                for(int j=1;j<=nlocaly;j++){
                for(int i=1;i<=nlocalx;i++){
                    sndbuf[(j-1)*nlocalx+i-1] = Math.pow(phi[j][i],0.33);  }  }
S           MPI.COMM_WORLD.Send(sndbuf,0,nlocalx*nlocaly,MPI.DOUBLE,0,99);
    }
```

Listing 5.13: Code Snippet Demonstrating the Importance of Data Facilitators

Refining s_unit Detection Inside Special Conditionals. Since we ensure that all the process-controlling "if" conditionals are chosen, a heuristic is required to find the most important s_units within that conditional. S_units chosen for the code snippet in Listing 5.14 demonstrate that an incorrect message could be conveyed if the statements within the process-controlling "if" conditionals are not chosen.

```
From benchmark: Integrate
31 S    if (myrank==0){
32          System.out.println("The integral = " + dval[1]);
33       }
34       MPI.Finalize();
35 S    r.sum();
```

Listing 5.14: Code snippet for process-controlling "if" conditional

5.2.4 Future Enhancements

a. We could use the comments present in the program to generate more precise summaries. For example, integer length; /* inch */ .

b. Template creation for MPI library calls could be automated with the availability of better documentation for MPI.

c. Communication patterns/topologies implemented by the programmer using MPI point-to-point calls need to be detected and handled carefully. But, detecting such complicated patterns is still an open issue in the literature.

Chapter 6

CONCLUSIONS AND FUTURE WORK

This thesis has investigated the problem of generating automatic comments for MPI parallel programs. It first presented an investigation of the reusability of an existing automatic comment generation technique designed for sequential Java programs to generate summary comments for parallel MPI programs. The second, and most important contribution of the thesis, is the implementation of a first prototype summary comment generator for parallel MPI Java programs. Finally, this prototype comment generator has been evaluated by manual analysis of the generated summary comments for the features that work well and the ones that need to be improved.

Much of the content selection phase of the Sridhara et al. [32] summary generator for sequential Java methods was reused, with modifications to include MPI library calls and some analysis to identify when processes are executing different code segments to include the appropriate control statements. To generate smooth natural language phrases, text generation templates were created for MPI calls, and patterns of typical process control and communication was identified so phrases that indicate high level views of the process's interactions can be generated (e.g., in terms of manager and workers). The preliminary evaluation results indicate that the overall process control and interactions are well reflected in the summaries, along with important actions of the method that appear as method calls. However, the underlying comment generator's handling of computational statements needs to be improved to gain more complete, clear summaries for these scientific codes.

As a part of future work, the future enhancements mentioned in section 5.2.4 on page 41 could be investigated along with the improvements suggested in section 5.2.3.2 on page 38. Most importantly, since there are not too many codes in Java, the base language issue has to be handled at some point to broaden the applicability of the comment generator for parallel MPI codes, which requires customizing SWUM for C and MPI as the first step.

Appendix A

TEXT GENERATION TEMPLATES FOR MPI LIBRARY CALLS

Method name: Abort

Method signature: Abort (int errorcode)

Template: abort MPI.

Method name: Allgather

Method signature: Allgather (java.lang.Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, java.lang.Object recvbuf, int recvoffset, int recvcount, Datatype recvtype)

Template: every process gathers equal amounts of data in arg-0 from each of the other processes and stores collected data in arg-4.

Method name: Allgatherv

Method signature: Allgatherv (java.lang.Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, java.lang.Object recvbuf, int recvoffset, int[] recvcount, int[] displs, Datatype recvtype)

Template: every process gathers varying amounts of data in arg-0 from each of the other processes and stores collected data in arg-4.

Method name: Allreduce

Method signature: Allreduce (java.lang.Object sendbuf, int sendoffset, java.lang.Object

recvbuf, int recvoffset, int count, Datatype datatype, Op op)

Template: combine elements in arg-0 from each process using the arg-6 operation, and store the combined value in arg-2 of every process.

Method name: Alltoall

Method signature: Alltoall (java.lang.Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, java.lang.Object recvbuf, int recvoffset, int recvcount, Datatype recvtype)

Template: each process exchanges (equal amounts of) data with all other processes, sending data in arg-0 and receiving arg-6 amounts of arg-7 data in arg-4.

Method name: Alltoallv

Method signature: Alltoallv (java.lang.Object sendbuf, int sendoffset, int[] sendcount, int[] sdispls, Datatype sendtype, java.lang.Object recvbuf, int recvoffset, int[] recvcount, int[] rdispls, Datatype recvtype)

Template: each process exchanges (varying amounts of) data with all other processes, sending data in arg-0 and receiving arg-7 amounts of arg-9 data in arg-5.

Method name: Attr_get

Method signature: Attr_get (int keyval)

Template: get attribute value of arg-0.

Method name: Barrier

Method signature: Barrier ()

Template: wait for all processes in the group.

Method name: Bcast

Method signature: Bcast (java.lang.Object buf, int offset, int count, Datatype type, int root)

Template: process arg-4 broadcasts arg-0 to all processes.

Method name: Bsend_init

Method signature: Bsend_init (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: create a persistent communication request to send arg-0 to process arg-4 in buffered mode.

Method name: Bsend

Method signature: Bsend (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: send arg-0 to process arg-4 in buffered mode.

Method name: Buffer_attach

Method signature: Buffer_attach(byte [] buffer)

Template: provide a agr-0 to MPI in user's memory to be used for buffering outgoing messages.

Method name: Buffer_detach

Method signature: Buffer_detach()

Template: detach the buffer currently associated with MPI.

Method name: clone

Method signature: clone ()

Template: duplicate and return a copy of this communicator.

Method name: Creat

Method signature: Creat(Group group)

Template: create a new communicator group arg-0.

Method name: Create_cart

Method signature: Create_cart (int[] dims, boolean[] periods, boolean reorder)

Template: create a Cartesian topology communicator whose group is a subset of the group of this communicator.

Method name: Create_graph

Method signature: Create_graph (int[] index, int[] edges, boolean reorder)

Template: create a graph topology communicator whose group is a subset of the group of this communicator.

Method name: Create_intercomm

Method signature: Create_intercomm (Comm local_comm, int local_leader, int remote_leader, int tag)

Template: create an inter-communicator between two intra-communicators with leaders arg-1 and arg-2.

Method name: Finalize

Method signature: Finalize ()

Template: finalize MPI.

Method name: Gather

Method signature: Gather (java.lang.Object sendbuf, int sendoffset, int sendcount, Datatype

sendtype, java.lang.Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)

Template: process arg-8 gathers equal amounts of data in arg-0 from each process and stores in arg-4.

Method name: Gatherv

Method signature: Gatherv (java.lang.Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, java.lang.Object recvbuf, int recvoffset, int[] recvcount, int[] displs, Datatype recvtype, int root)

Template: process arg-9 gathers varying amounts of data in arg-0 from each process and stores collected data in arg-4.

Method name: Get_processor_name

Method signature: Get_processor_name ()

Template: get the local hostname.

Method name: Group

Method signature: Group ()

Template: return group associated with the communicator.

Method name: Ibsend

Method signature: Ibsend (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: initiate a buffered mode, nonblocking send to process arg-4.

Method name: Init

Method signature: Init ()

Template: initialize MPI.

Method name: Initialized

Method signature: Initialized ()

Template: test if MPI has been initialized.

Method name: Irecv

Method signature: Irecv (java.lang.Object buf, int offset, int count, Datatype type, int source, int tag)

Template: initiate a nonblocking receive from process arg-4.

Method name: Irsend

Method signature: Irsend (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: initiate a ready mode, nonblocking send to process arg-4.

Method name: Isend

Method signature: Isend (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: initiate a standard mode, nonblocking send to process arg-4.

Method name: Issend

Method signature: Issend (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: initiate a synchronous mode, nonblocking send to process arg-4.

Method name: Pack

Method signature: Pack (java.lang.Object inbuf, int offset, int incount, Datatype datatype, byte[] outbuf, int position)

Template: pack contents of the send buffer, arg-0 into arg-4.

Method name: Rank

Method signature: Rank ()

Template: each process get its rank.

Method name: Recv_init

Method signature: Recv_init(java.lang.Object buf, int offset, int count, Datatype type, int source, int tag)

Template: create a persistent communication request to receive arg-0 from process arg-4 in standard mode.

Method name: Recv

Method signature: Recv(java.lang.Object buf, int offset, int count, Datatype type, int source, int tag)

Template: receive from process arg-4 and store in arg-0.

Method name: Reduce_scatter

Method signature: Reduce_scatter (java.lang.Object sendbuf, int sendoffset, java.lang.Object recvbuf, int recvoffset, int[] recvcounts, Datatype datatype, Op op)

Template: combine elements in arg-0 from each process using the arg-6 operation, and scatter the result (equal amounts) to each process in the group.

Method name: Reduce

Method signature: Reduce (java.lang.Object sendbuf, int sendoffset, java.lang.Object

recvbuf, int recvoffset, int count, Datatype datatype, Op op, int root)

Template: combine elements in arg-0 from each process using the arg-6 operation, and store result in arg-2 of the process arg-7.

Method name: Rsend

Method signature: Rsend (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: send arg-0 to process arg-4 in ready mode.

Method name: Scan

Method signature: Scan (java.lang.Object sendbuf, int sendoffset, java.lang.Object recvbuf, int recvoffset, int count, Datatype datatype, Op op)

Template: perform a prefix reduction on data distributed across the group using the arg-6 operation.

Method name: Scatter

Method signature: Scatter (java.lang.Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, java.lang.Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)

Template: process arg-8 scatters equal amounts of data in arg-0 to each process in the group.

Method name: Scatterv

Method signature: Scatterv (java.lang.Object sendbuf, int sendoffset, int[] sendcount, int[] displs, Datatype sendtype, java.lang.Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)

Template: process arg-9 scatters varying amounts of data in arg-0 to every process.

Method name: Send_init

Method signature: Send_init (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: create a persistent communication request to send arg-0 to process arg-4 in standard mode.

Method name: Send

Method signature: Send (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: send arg-0 to process arg-4.

Method name: Sendrecv_replace

Method signature: Sendrecv_replace (java.lang.Object buf, int offset, int count, Datatype type, int dest, int sendtag, int source, int recvtag)

Template: send arg-0 to process arg-4 and receive into arg-0 from process arg-6.

Method name: Sendrecv

Method signature: Sendrecv (java.lang.Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, int dest, int sendtag, java.lang.Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int source, int recvtag)

Template: send arg-0 to process arg-4 and receive in arg-6 from process arg-10.

Method name: Size

Method signature: Size ()

Template: get the count of the processes in the group.

Method name: Split

Method signature: Split (int colour, int key)

Template: partition the group associated with this communicator and create a new communicator for each subgroup.

Method name: Ssend_init

Method signature: Ssend_init (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: create a persistent communication request to send arg-0 to process arg-4 in synchronous mode.

Method name: Ssend

Method signature: Ssend (java.lang.Object buf, int offset, int count, Datatype type, int dest, int tag)

Template: send arg-0 to process arg-4 in synchronous mode.

Method name: Unpack

Method signature: Unpack (byte[] inbuf, int position, java.lang.Object outbuf, int offset, int outcount, Datatype datatype)

Template: unpack contents of the receive buffer, arg-2 into space specified in arg-0.

Appendix B

TEXT PHRASES FOR MPI CONSTANTS

MPI Constant : **Phrase**

MPI.BYTE : byte

MPI.CHAR : character

MPI.SHORT : short integer

MPI.BOOLEAN : boolean value

MPI.INT : integer

MPI.LONG : long integer

MPI.FLOAT : floating point value

MPI.DOUBLE : double precision floating point value

MPI.OBJECT : object

MPI.TAG_UB : largest tag value

MPI.HOST : rank of process that is host

MPI.IO : rank of process that can do i/o

MPI.WTIME_IS_GLOBAL : wtime is global

MPI.GRAPH : general graph

MPI.CART : cartesian grid

MPI.IDENT : identical

MPI.CONGRUENT : groups are identical

MPI.SIMILAR : same members, but in a different order

MPI.UNEQUAL : different

MPI.COMM_SELF : only the calling process

MPI.GROUP_EMPTY : group empty

MPI.SHORT2 : pair of short integers

MPI.INT2 : pair of integers

MPI.LONG2 : pair of long integers

MPI.FLOAT2 : pair of floating point values

MPI.DOUBLE2 : pair of double precision floating point values

MPI.PACKED : packed

MPI.LB : lower bound

MPI.UB : upper bound

MPI.ANY_SOURCE : any process

MPI.ANY_TAG : any tag value

MPI.MAX : maximum

MPI.MIN : minimum

MPI.SUM : sum

MPI.PROD : product

MPI.LAND : logical and

MPI.BAND : bit-wise and

MPI.LOR : logical or

MPI.BOR : bit-wise or

MPI.LXOR : logical xor

MPI.BXOR : bit-wise xor

MPI.MINLOC : min value and location

MPI.MAXLOC : max value and location

BIBLIOGRAPHY

[1] JavaMPI–automatic generation of MPI wrappers. http://www.hpjava.org/papers/MPJ-CPE/cpempi/node4.html.

[2] JMPI: Implementing the Message Passing Standard in Java. http://euler.ecs.umass.edu/jmpi/.

[3] MPIJ–MPI-like implementation in pure Java. http://www.hpjava.org/papers/MPJ-CPE/cpempi/node5.html.

[4] MPJ Benchmarks. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/mpj.html.

[5] MPJ Express. http://mpj-express.org.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[7] M. Baker, B. Carpenter, and A. Shaft. Mpj Express: Towards Thread Safe Java HPC. *IEEE International Conference on Cluster Computing*, 0:1–10, 2006.

[8] Raymond P.L. Buse and Westley R. Weimer. Automatic Documentation Inference for Exceptions. In *Intl Symp on Software Testing and Analysis (ISSTA), 2008*.

[9] Raymond P.L. Buse and Westley R. Weimer. Automatic Documentation Inference for Exceptions. In *ISSTA: Proceedings of International Symposium on Software Testing and Analysis*, 2008.

[10] Bryan Carpenter. The HP Java Project. http://www.hpjava.org/mpiJava.html, 2009. [Online; accessed 26-Oct-2009].

[11] Steve Carr, Jean Mayo, and Ching-Kuang Shene. ThreadMentor: A Pedagogical Tool for Multithreaded Programming. *J. Educ. Resour. Comput.*, 3(1):1, 2003.

[12] Pohua P. Chang, Scott A. Mahlke, and W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.

[13] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A Study of the Documentation Essential to Software Maintenance. In *Intl Conf on Design of Commun (SIGDOC)*, 2005.

[14] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. An Introduction to the MPI Standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1995.

[15] Timothy E. Erickson. An Automated FORTRAN Documenter. In *1st Annual Intl Conf on Systems Documentation*, 1982.

[16] Scott D. Fleming, Eileen Kraemer, R.E.K Stirewalt, Laura K. Dillon, and Shaohua Xie. Refining Existing Theories of Program Comprehension During Maintenance for Concurrent Software. In *ICPC: Proceedings of the16th IEEE International Conference on Program Comprehension*, 2008.

[17] S Haiduc, J Aponte, L Moreno, and A Marcus. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *Working Conference on Reverse Engineering (WCRE)*, 2010.

[18] M. Harman, N. Gold, R. Hierons, and D. Binkley. Code Extraction Algorithms which Unify Slicing and Concept Assignment. In *Working Conf on Reverse Engineering*, 2002.

[19] Emily Hill. *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration*. Ph.D. dissertation, University of Delaware, 2010.

[20] Emily Gibson Hill. AMAP: Automatically Mining Abbreviation Expansions in Programs - Experiment Resources. `http://www.cis.udel.edu/~gibson/amap/`, 2008.

[21] Mira Kajko-Mattsson. A Survey of Documentation Practice within Corrective Maintenance. *Empirical Softw. Engg.*, 10(1):31–55, 2005.

[22] Mik Kersten and Gail C. Murphy. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–11, 2006.

[23] Fan Long, Xi Wang, and Yang Cai. API Hyperlinking via Structural Overlap. In *ACM SIGSOFT Symp on The Foundations of Software Engineering*, 2009.

[24] Inderjeet Mani. *Automatic Summarization*. John Benjamins, 2001.

[25] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[26] Mi-Young Park, Nguyen Cao Truong Hai, Yong-Kee Jun, and Hyuk-Ro Park. Visualization of Message Races in MPI Parallel Programs. In *CIT : Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, 2007.

[27] Juergen Rilling, Hon F. Li, and Dhrubajyoti Goswami. Predicate-Based Dynamic Slicing of Message Passing Programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, 2002.

[28] David Roach, Hal Berghel, and John R. Talburt. An Interactive Source Commenter for Prolog Programs. *SIGDOC Asterisk J. Comput. Doc.*, 14(4):141–145, 1990.

[29] Pierre N. Robillard. Schematic Pseudocode for Program Constructs and its Computer Automation by SCHEMACODE. *Commun. ACM*, 29(11), 1986.

[30] R. Schiefer and P. van der Stok. VIPER: A Tool for the Visualisation of Parallel Programs. In *PDP: Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing*, 1995.

[31] Lucas Mello Schnorr, Phileppe Olivier Alexandre Navaux, and Benhur de Oliveira Stein. DIMVisual: Data Integration Model for Visualization of Parallel Program Behavior. In *CCGRID: Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*, 2006.

[32] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards Automatically Generating Summary Comments for Java Methods. In *ASE: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010.

[33] JT Stasko, V Graphics, and U Center. The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report. *Journal of Parallel and Distributed Computing*, 1995.

[34] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation. *Journal of Program Languages*, 4(3), 1996.

[35] T. Tenny. Program Readability: Procedures Versus Comments. *IEEE Trans. Softw. Eng.*, 14(9), 1988.

[36] David Walker. mpiJava Example Programmes. `http://users.cs.cf.ac.uk/David.W.Walker/CM0323/code.html`.

[37] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The Effect of Modularization and Comments on Program Comprehension. In *Intl Conf on Software Engineering (ICSE)*, 1981.

[38] Shaohua Xie, Eileen Kraemer, R. E. K. Stirewalt, Laura K. Dillon, and Scott D. Fleming. Design and Evaluation of Extensions to UML Sequence Diagrams for Modeling Multithreaded Interactions. *Information Visualization*, 8(2):120–136, 2009.