A refactoring catalogue and tool for refactoring C/C++ HPC code

Sunna Berglind Sigurðardóttir

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
University of Iceland
2020

A REFACTORING CATALOGUE AND TOOL FOR REFACTORING C/C++ HPC CODE

Sunna Berglind Sigurðardóttir

60 ECTS thesis submitted in partial fulfillment of a
Magister Scientiarum degree in Software Engineering

Advisors
Helmut Neukirchen
Morris Riedel

External Examiner
Markus Götz

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, May 2020

A refactoring catalogue and tool for refactoring C/C++ HPC code
Refactoring for HPC code
60 ECTS thesis submitted in partial fulfillment of a M.Sc. degree in Software Engineering

Faculty of Industrial Engineering, Mechanical Engineering and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Hjarðarhagi 2-6
107, Reykjavik, Reykjavik
Iceland

Telephone: 525 4000

Bibliographic information:
Sunna Berglind Sigurðardóttir, 2020, A refactoring catalogue and tool for refactoring C/C++ HPC code, M.Sc. thesis, Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland.

Printing: Háskólaprent, Fálkagata 2, 107 Reykjavík
Reykjavik, Iceland, May 2020

Dedicated to my daughter, always believe in yourself.

Abstract

High-Performance Computing (HPC), or supercomputing, has become a big part of our daily life even though people might not be aware of it. The use of HPC is growing and the need for HPC is expected to keep growing. HPC code is typically written by domain experts who are not experienced in software engineering so when it is modified most often the complexity of it increases, losing readability and maintainability. One of the techniques developed to decrease this complexity is refactoring. Refactoring is a technique for improving the internal structure of software without changing its external behavior. It has been widely used for all kinds of software but for HPC code it is still evolving and different definitions of HPC refactoring have surfaced. This thesis discusses previous work on HPC refactoring and C/C++ refactoring and introduces a new definition of HPC refactoring which focuses not only on improving readability and maintainability but includes performance and portability as well. With this definition in mind a new catalogue for HPC refactoring is started along with implementing a semi-automated refactoring. The five refactorings in the catalogue are aimed at improving performance of HPC code written in C/C++ with detailed descriptions of how to apply them. The implemented refactoring demonstrates the advantages of semi-automated refactorings for their users by limiting the risks inevitably followed by code changes.

Útdráttur

Ofurtölvuvinnsla (e. HPC) er orðinn stór hluti af okkar daglega lífi þó fólk átti sig mögulega ekki á því. Notkun ofturtölvuvinnslu hefur aukist og þörfin fyrir ofurtölvuvinnslu mun væntanlega aukast áfram. Ofurtölvuvinnslukóði er yfirleitt skrifaður af sérfræðingum sem hafa ekki mikla reynslu af hugbúnaðarverkfræði þannig að þegar kóða er breytt eykst flækjustig hans almennt einnig, sem hefur neikvæð áhrif á læsileika og viðhaldsþægni. Ein af aðferðunum sem hafa verið þróaðar til að lágmarka flækjustigið er endurþáttun kóða (e. refactoring). Endurþáttun er aðferð sem snýr að því að bæta innri byggingu hugbúnaðar án þess að breyta ytri hegðun hans. Þessi aðferð hefur mikið verið notuð fyrir allskonar hugbúnað en fyrir ofurtölvuvinnslukóða er hún enn í þróun og mismunandi skilgreiningar hafa komið fram. Í þessari ritgerð er fjallað um þá vinnu sem unnin hefur verið í sambandi við endurþáttun ofurtölvuvinnslukóða og C/C++ endurþáttun og kynnir til sögunnar nýja skilgreiningu á endurþáttun ofurtölvuvinnslukóða sem leggur ekki einungis áherslu á að bæta læsileika og viðhaldsþægni heldur einnig frammistöðu og flytjanleika (e. portability). Með þessa skilgreiningu í huga er ný skrá fyrir endurþáttun ofurtölvuvinnslukóða kynnt ásamt þróun hálf-sjálfvirkrar endurþáttunar. Fimm endurþáttanir eru í skránni en þær leggja áherslu á að bæta frammistöðu ofurtölvuvinnslukóða sem er skrifaður í C/C++ ásamt því að lýsa því í smáatriðum hvernig á að beita þeim. Sú endurþáttun sem þróuð er sýnir kostina sem hálf-sjálfvirkar endurþáttanir hafa fyrir notendur en þær takmarka áhætturnar sem óumflýjanlega fylgja breytingum á kóða.

Contents

List of Figures

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CDT	C/C++ Development Tooling
CUDA	Compute Unified Device Architecture
EPL	Eclipse Public License
GPU	Graphics Processing Unit
HPC	High-Performance Computing
IDE	Integrated Development Environment
JDT	Java Development Tools
LTK	Eclipse Language Toolkit
MP	Message Passing
MPI	Message Passing Interface
PDE	Plug-in Development Environment
PTP	Parallel Tools Platform
SDK	Software Development Kit
TU	Translation Unit
UI	User Interface
UML	Unified Modeling Language

Acknowledgments

First and foremost I would like to thank my supervisor Dr. Helmut Neukirchen, Professor of Computer Science and Software Engineering at the University of Iceland, for his invaluable guidance, help, and patience throughout this project. I would also like to thank Dr. Morris Riedel and Dr. Markus Götz for their efforts as secondary supervisor and external examiner respectively. Last but not least I would like to thank my family, my parents for always believing in me, my sister for being my number one pep talker, my husband for his endless support, and my daughter for always knowing how to put a smile on my face.

1 Introduction

High-Performance Computing (HPC), or supercomputing, has become a big part of our daily life even though people might not be aware of it. Whether it is simulating effects of natural disasters, or even predicting them, or developing new treatments within the health care system based on personalised medicine, HPC makes a difference for everyone. The use of HPC is growing and the need for HPC is expected to keep growing [25].

Larger data sets require more computing power which then at times is not reached without using some kind of parallel processing. Therefore more and more parallel code is created by all kinds of scientists, not only computer scientists, software engineers or programmers. In addition to new code there is also a lot of existing code in use that needs to be maintained. Experience in creating code that is readable and maintainable can often be lacking for those who have focused on other things than coding or feel that the code itself does not matter as much as the outcome. In fact, even if the coding is flawless and follows all the strictest rules when it is designed it will eventually age [43] as time goes on and code is added or changed maybe by multiple different people. This reduces readability and maintainability, so that once beautiful code becomes a mess.

One solution to the software aging problem is code refactoring, a term first used and described by William Opdyke [41] in his Ph.D. thesis. It was again described by Martin Fowler [21] as the process of changing a software system to improve its internal structure without altering the external behavior. He presented a catalogue where each refactoring is presented in a certain format from which the catalogue in this thesis is inspired by. Automating or semi-automating the refactorings can be extremely valuable. Refactorings are often semi-automated, i.e. depend on user input, where the user might need to e.g. name a new parameter or manually select a certain expression to refactor. Automated refactorings detect the refactoring opportunity and apply the refactoring without user input, but they are rare. Automated or semi-automated refactorings remove the *human error* part of changing the code, making it easy for those who might not have as much experience in programming to maintain and refactor their code by pressing a few buttons, so to speak. With the automated or semi-automated refactorings the need for excessive testing for each small change is minimized as we assume the functionality of the refactoring is correct. This makes the refactorings much more available to users with different

backgrounds along with decreasing the effort needed to apply them.

When improving HPC code without changing the external functionality the goal might be different. In HPC the focus often is performance, although some argue that the improvement needs to be significant in order to justify the work [3]. The goal of the computational scientist is not the same as the goal of the computer scientist or the software engineer, where the computational scientist focuses on maximizing the scientific output versus the software engineer's focus on improving code performance. The "conventional" refactoring methods can already be used to improve readability and maintainability. This of course also applies to HPC code but the added focus for refactoring specific to HPC would be to improve performance and make sure the parallel resources are being used to their fullest extent along with looking at portability between systems. By making the refactorings semi-automatic it also minimizes the effort needed for improvement, hopefully bringing all kinds of scientists on board even for minor performance improvements. Automation makes refactoring more accessible for all and helps those less familiar with the practise, e.g. the HPC domain experts, adopt it.

1.1 Thesis objectives

The objective of this thesis is to introduce a catalogue for HPC refactorings, with the goal of improving performance or readability and maintainability, along with implementing a refactoring from the catalogue as a proof of concept that by making these refactorings semi-automatic we can make it easier for every user to apply them. This catalogue is just a start and has the hopes of growing bigger with time as refactorings are added and even new refactorings discovered. The developed implementation described in this thesis allows the user to semi-automatically apply the selected refactoring. The refactoring is implemented as an Eclipse plug-in [9] using the Eclipse Plug-in Development Environment (PDE) [18]. It is largely based on the "conventional" refactorings already available in Eclipse C/C++ Development Tooling (CDT) [13]. The Eclipse environment is ideal for refactoring implementation as many refactorings are already available there both through Java Development Tools (JDT) [14] and CDT. JDT and CDT are open-source projects which allows a developer to dive into the source code making it unnecessary to "reinvent the wheel" for the basic structure common to most or all of the refactorings.

1.2 Outline

The outline of this thesis is as follows: Chapter 2 discusses the foundations for the later chapters of this thesis. It covers HPC, what it is and how to use the parts which are relative to this thesis. In the same chapter the meaning and process of refactoring are covered along with the Eclipse project. Chapter 3 describes related work that has already been done on refactoring HPC code and the Eclipse CDT project which among multiple other things includes refactoring for C/C++ code. A refactoring catalogue for refactoring HPC code is introduced in Chapter 4 where each refactoring is described in a format similar to the one Fowler [21] proposes. Chapter 5 describes the implementation and testing of one of the refactorings described in the catalogue. A case study to validate the functionality of the implemented refactoring is performed in Chapter 6. This thesis is concluded with a summary and an outlook in Chapter 7.

2 Foundations

This chapter discusses the foundations on which this thesis is built. Section 2.1 describes High-Performance Computing (HPC) and the Message Passing Interface (MPI) along with the motivation behind using parallel processing. Refactoring in general, independent from HPC, is discussed in Section 2.2. It also covers the usual refactoring format and examples. Section 2.3 discusses the Eclipse Project that is used to implement automated HPC refactorings, and different parts of it.

2.1 High-Performance Computing (HPC)

Verschelde [50] introduces supercomputing and HPC as synonyms using the following definition:

> Doing supercomputing means to use a supercomputer and is also called high performance computing.

In relation to that definition Verschelde also discusses supercomputers and the definition of a supercomputer:

> A *supercomputer* is a computing system (hardware, system & application software) that provides close to the best currently available sustained performance on demanding computational problems.

Hager and Wellein [27] discuss that high-performance computing deals both with the implementation (code) and the hardware it runs on. In their book, "Introduction to high performance computing for scientists and engineers", Fortran, C and C++ are used for examples as these programming languages are to this day popular for parallel programming.

Basili et al. [3] discuss the software engineer's perspective on HPC. They mention the scientist's need for increasing performance up to a certain point. Scientists will not give up every other aspect for performance unless there is significant improvement. They also discuss the fact that scientists are not all computer scientists and therefore

look differently at performance gains.

2.1.1 Parallel Processing

Barr and Hickman [2] define parallel processing as follows:

> *Parallel processing* is the simultaneous manipulation of data by multiple computing elements working to complete a common body of work.

Parhami [42] discusses the need for parallel processing in the endless search for higher-performance digital computers. Moore's law [29], as seen in Figure 2.1, will eventually reach a limit according to physical laws [42]. Parhami describes the *Speed-of-Light Argument* which introduces that there is a limit to how fast signals can travel as they will never travel at a speed beyond the speed-of-light. When this limit has been reached the only option for better performance is using multiple processors. High-performance computers obviously obey the same physical laws which makes the hunt for even higher performance look to parallel processing as well. Parhami [42] summarizes the motivation for parallel processing:

- Higher speed, or solving problems faster.
- Higher throughput, or solving more instances of given problems.
- Higher computational power, or solving larger problems.

2.1.2 Message Passing Interface (MPI)

The MPI Forum [30] publishes the MPI standard [8]. The standard defines MPI as follows:

> MPI (Message-Passing Interface) is a *message-passing library interface specification.*

Hager and Wellein [27] discuss the creation of MPI. At first when parallel computers came to the HPC market the only way to compute in parallel was to use explicit Message Passing (MP). This parallelization method is both complicated and tedious although it is the most flexible way to solve the problem. Computer vendors saw that efficient message-passing facilities were needed but until the early 1990s they only provided non-portable libraries for efficient message-passing facilities. Finally MPI came to be as a result of a joint standardization effort. This needed to be done

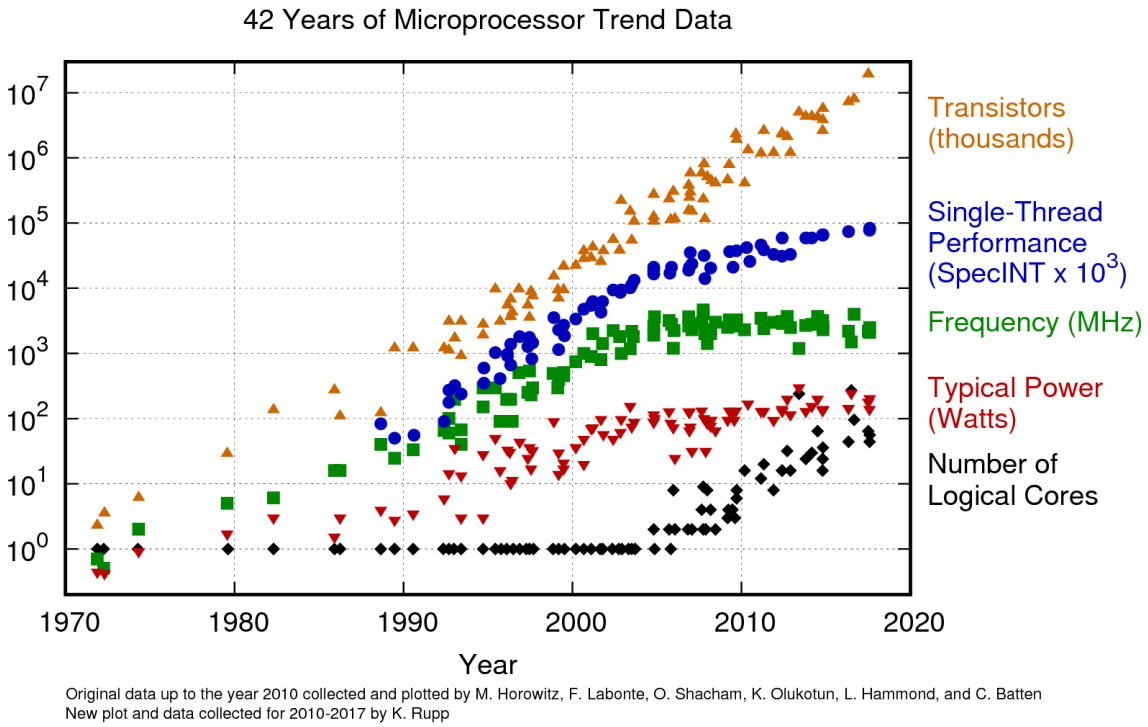

Figure 2.1: The yellow triangles show exponential growth of transistor counts in the years 1971–2017 [47].

in order to make parallel programs that were portable between platforms a reality. Today, the MPI standard contains communication routines along with facilities for efficient parallel I/O and an MPI library is a fundamental part of any HPC system installation. The focus here is on the MPI concepts that are relevant for the HPC refactorings developed in this thesis.

Hager and Wellein [27] describe communication using MPI. If a parallel computer is of the distributed-memory type, message passing is required as one processor can in no way directly access another processors' address space and therefore needs some functions for message passing which then use the communicator as an argument. Examples of MPI's point-to-point message passing functions are `MPI_Send`, `MPI_Recv`, `MPI_Isend`, `MPI_Irecv`, and examples of collective calls are `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather`, `MPI_Allgather`. Barker [1] describes them in the following way:

- `MPI_Send` is a blocking send of data in the buffer.
- `MPI_Recv` is a blocking receive of data into the buffer.
- `MPI_Isend` is a non-blocking send of data in the buffer.

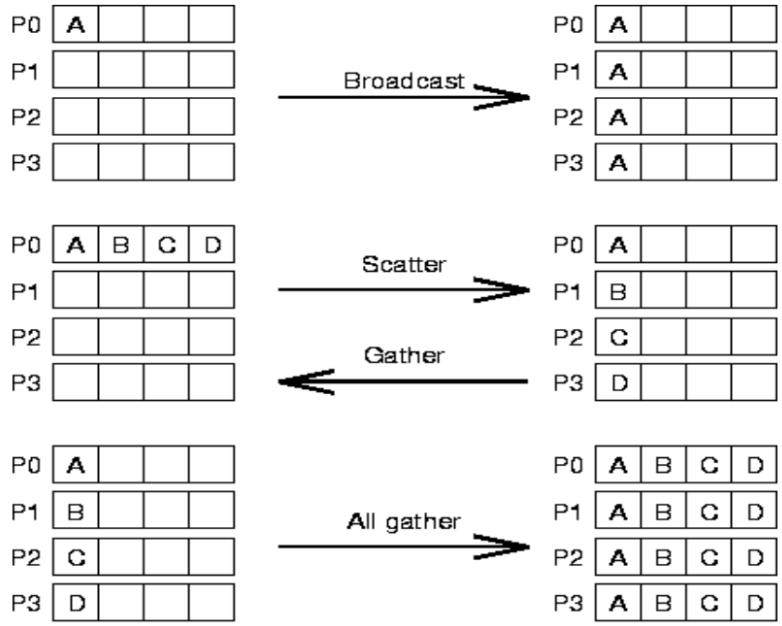

Figure 2.2: Data movement of collective communication `MPI_Bcast`*,* `MPI_Scatter`*,* `MPI_Gather`*, and* `MPI_Allgather` *[1].*

- `MPI_Irecv` is a non-blocking receive of data into the buffer.
- `MPI_Bcast` is a collective communication of type data movement, sending data from root to all processes in the communicator in a blocking way, see Figure 2.2.
- `MPI_Scatter` is a collective communication of type data movement that sends data from one processor to multiple other processors, see Figure 2.2.
- `MPI_Gather` is a collective communication of type data movement that collects data from multiple processors to one processor, see Figure 2.2.
- `MPI_Allgather` is a collective communication of type data movement that sends data from all processors to all other processors, see Figure 2.2.

Until the message buffer being sent or received is safe to use, the blocking calls suspend execution of the process. When using a non-blocking call the communication is just initiated and the programmer later has to check the success of the communication and the status of data transfer by using either `MPI_Wait` or `MPI_Test`. When using a collective call, such as `MPI_Bcast`, all processes withing the communicator are involved. Hager and Wellein [27] describe the communicator used within MPI. When a parallel environment is initialized, MPI sets up a communicator known as *world communicator*, called `MPI_COMM_WORLD`. Every process that has been started as

a part of the parallel program is described by the world communicator but if needed, subsets of it can be defined. Almost every MPI call requires a communicator as an argument.

2.2 Refactoring

This section discusses code refactoring in general. Section 2.2.1 describes software aging where the need for refactoring often rises. The following Section 2.2.2 discusses refactoring definitions and application.

2.2.1 Software aging

Parnas [43] introduced the notion of *Software Aging*:

> Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limits its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable.

Software aging is inevitable and has with time grown to be a significant challenge, with old software programs being necessary components in today's society.

Parnas [43] discusses the two types of software aging. First is failure to modify the software to meet changing needs and the second is the result of the changes that have been made. He calls this two types *Lack of movement* and *Ignorant surgery*. Lack of movement causes the users to become dissatisfied and will choose a new product over the old one when the costs of converting and retraining are outweighed by the benefits. Ignorant surgery refers to the fact that the people who make the changes are usually not those who designed the programs in the first place, therefore without full understanding of the original design concept. Changes usually degrade the program structure and invalidate the original design concept. This results in software becoming very expensive to update with time, when these modifications are made more often.

2.2.2 Refactoring

The term *refactoring* was first coined and used by Opdyke [41]. In his Ph.D. thesis he discusses that refactorings are reorganization plans that support change at an intermediate level. The work on refactorings has been continued since then and when referring to refactoring many look to Martin Fowler and his book on refactorings. Fowler [21] defines refactoring in two ways, depending on if refactoring is referred to as a noun or a verb. The two definitions are as follows:

> Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
>
> Refactoring (verb): to restructure software by applying a series of refactorings without changing its observable behavior.

Best practice when developing software is designing first before the code itself is written. Thus, in the beginning the code should be well designed and without the need for refactoring. As time goes on, a need for changes in the code arises and this design will become outdated. Quick fixes and small additions can quickly make a mess of the once well-designed code. As mentioned in Section 2.2.1 this would be described as a form of software aging, more specifically *Ignorant surgery*, when small parts of the code are changed without looking at the bigger picture or the original concept of the code. It can then be overcome with code smells, which are the parts of code that have the opportunity to be refactored. Refactoring can take this messy code and with many small changes return it to its former well-designed glory without any changes to the external behavior [21]. Fowler [21] discusses how to locate bad smells in code and how to use refactorings to fix them. He then lists a catalogue of refactorings written by himself and others. Each refactoring has the same five part format in the book:

- Name of refactoring
- Summary of the situation where the refactoring can be used along with what it does
- Motivation behind the refactoring
- A step-by-step mechanic for using the refactoring
- An example of a use of the refactoring

One of the refactorings Fowler [21] lists is the Rename Method refactoring shown in

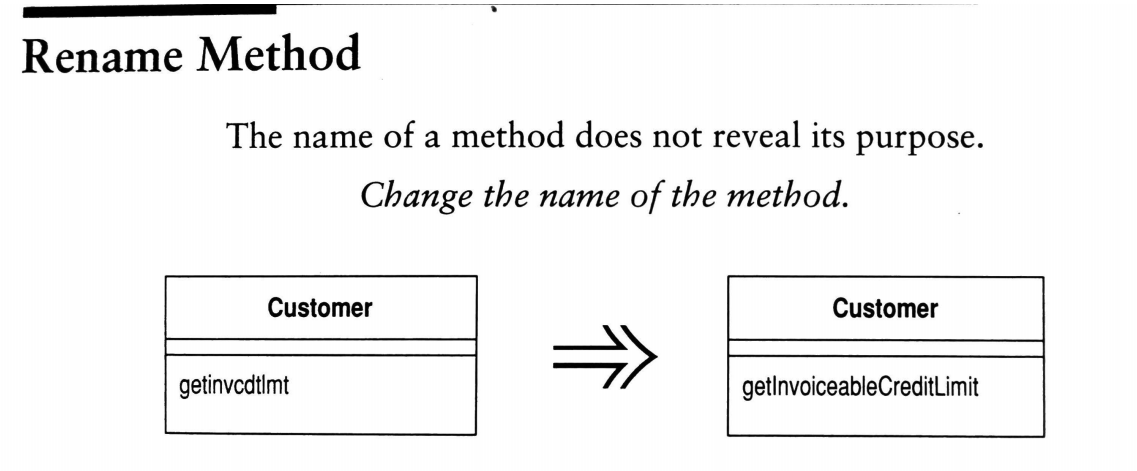

Figure 2.3: Excerpt from Fowler's [21] Rename Method refactoring

Figure 2.3. It is a relatively simple refactoring that changes the name of a method where the method name is not descriptive of the purpose of the method. The mechanics behind this refactoring are as follows [21]:

- Check to see whether the method signature is implemented by a superclass or subclass. If it is, perform these steps for each implementation.
- Declare a new method with the new name. Copy the old body of code over to the new name and make any alterations to fit.
- Compile.
- Change the body of the old method so that it calls the new one.
- Compile and test.
- Find all references to the old method name and change them to refer to the new one. Compile and test after each change.
- Remove the old method.
- Compile and test.

When looking at this mechanics list it is obvious how important testing is. Whenever there is a code change that could have impact on the functionality of the code, it is tested. This is done to minimize the possibility of changing or creating problems in the code functionality where this should only change the code design.

In his book, Fowler [21] assumes that the mechanics of each refactoring are applied manually. Each step of a refactoring that introduces changes is followed by either a *Compile* or a *Compile and test* step. When a refactoring step does not include changing the testable behaviour (e.g. adding code that is not yet used anywhere) a *Compile* should follow, but if the testable behavior is changed it is necessary that a *Compile and test* step follows. The purpose of the *Compile* step is checking for things that the programmer might have missed, i.e. checking for typos, which lead to syntax errors, or omissions, e.g. forgetting to change an identifier, causing semantic errors. Fowler describes that for manual application, the automated tests are used as safety nets to grab problems that might arise at each step of applying a refactoring, which is the purpose of the *test* part in the *Compile and test* step. Fowler [21] describes each step in the mechanics of each refactoring to be as small as possible but in practise usually larger steps are taken (backing out of a step and taking the smaller steps if a bug arises). While keeping each step as small as possible, the source code and functionality is kept in a valid state after each step (making it possible to pause a refactoring).

As Fowler and Opdyke [41] point out, a part of refactoring safely is trusting that the compiler catches errors that the programmer misses and trusting that the tests catch errors that the programmer and their compiler miss. In contrast to manual application, automated refactorings do not need to rely this heavily on the automated tests. Assuming that the automated refactoring is implemented correctly according to the mechanics there should not be a need to have the same safety net of automated tests.

Fowler recently published a new edition of his book [22], Refactoring, where among other things he changes the code examples from Java to JavaScript. The structure of the book is the same as the previous one but it has been revised and includes new refactorings along with new code examples.

2.3 Eclipse

In November 2001, IBM created The Eclipse Project [9] which is still used by millions of developers. Almost three years later, in January 2004, The Eclipse Foundation, an independent not-for-profit corporation, was created to establish an open community around Eclipse. A part of The Eclipse Foundation is Eclipse, a free and open-source Integrated Development Environment (IDE). Eclipse Software Development Kit (SDK) [10], shown in Figure 2.4, is made up of three parts, the Eclipse platform and the plug-in tools for it, Java Development Tools (JDT) and Plug-in Development Environment (PDE) which is described in Section 2.3.1.

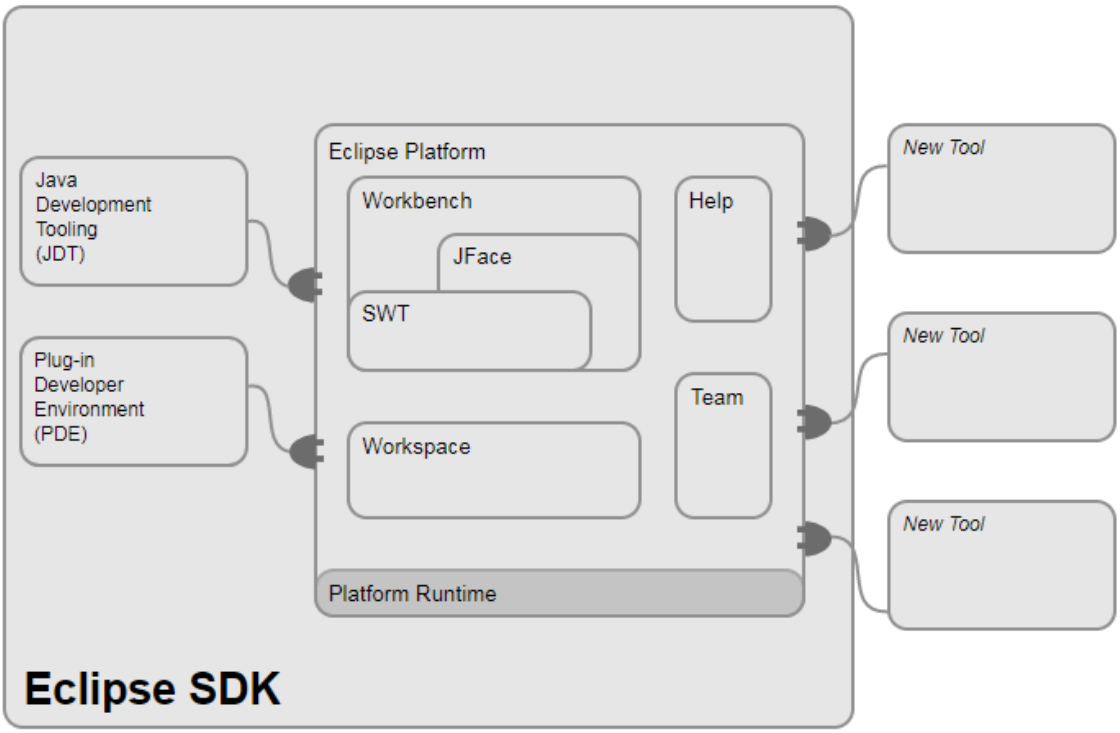

Figure 2.4: Eclipse SDK [19]

The Eclipse platform [19] is structured around the concept of plug-ins, which are structured bundles of data and/or code contributing functionality to the system. Each subsystem in the Eclipse platform is itself structured as a set of plug-ins. Each of these sets implement some key function. The Eclipse platform has a workspace [20], consisting of projects, which is the directory where a developer's work is stored.

2.3.1 Plug-in Development Environment (PDE)

One part of The Eclipse Project [10] is the PDE project. Eclipse already contains a number of plug-ins for developers and with the PDE [18] project they can themselves build plug-ins for Eclipse, defined by projects in the workspace. PDE provides tools for everything these developers need for Eclipse plug-ins, creating, developing, testing, debugging, building and deploying. PDE also simplifies building plug-ins for Eclipse by providing a number of views and editors. Within the PDE a developer can among other things:

- Create their own plug-in manifest file, the plugin.xml
- Specify their plug-in runtime
- Specify required plug-ins for their plug-in
- Define extension points

Developers can both create plug-ins for themselves to use or they can decide to share them, e.g. via Git, for other developers to use or contribute to in any way.

2.3.2 Eclipse Language Toolkit (LTK)

In 2004 at the EclipseCon conference the idea for Eclipse having a more generic language IDE infrastructure was brought up [16]. Eclipse already had a powerful functionality when working with Java and people were looking to be able to have that support using other programming languages. At that time this was being done by cloning JDT and replacing the parts specific to Java with a different language. This is obviously not efficient and hard to maintain as JDT keeps growing resulting in constantly having to catch up to these changes. The first attempt at a solution for this was introduced in Eclipse 3.0. It goes by the name Eclipse Language Toolkit (LTK) and contains two plug-in projects with infrastructure for language-independent refactorings:

- `org.eclipse.ltk.core.refactoring`
- `org.eclipse.ltk.ui.refactoring`

These two plug-in projects contain both the definition of a refactoring along with abstract classes in the core and also the UI components needed. The `Refactoring` class represents the life cycle of a refactoring while the `Change` class contains a validation of whether the refactoring is appropriate and performs the workspace modifications that need to be done. Frenzel [23] describes this life cycle as a predefined procedure that every refactoring in Eclipse follows:

1. The refactoring is started by the user.
2. An initial quick check is performed to determine whether the refactoring is applicable at all in the context desired by the user (`checkInitialConditions()`).
3. The user is asked for additional information if necessary.

4. After all necessary information is available for executing the refactoring, a careful check is triggered (`checkFinalConditions()`) and the individual changes in the source text are calculated (`createChange()`).

5. The preview dialogue displays the changes; the user confirms them and the LTK applies them to the workspace.

2.3.3 C/C++ Development Tooling (CDT)

Schaefer [48] describes the start of CDT in the summer of 2001 when the first CDT summit was held in Canada. Throughout the years the mission of the CDT project still stays the same:

> Build a community driven IDE that helps make the lives of developers who work on C/C++ applications better [48].

Based on the Eclipse platform the CDT [13] project provides a fully functional C/C++ IDE. This project includes among many other things source code refactoring and code editor with syntax highlighting. This set of Eclipse plug-in provides C/C++ extensions to the Eclipse workbench, simplifying tools that were otherwise used from the command line, such as building, compiling and debugging [11]. In the Eclipse workbench CDT is used through the C/C++ perspective which consists of an editor and a few views.

CDT is an open source project which makes it possible for other programmers to edit and/or add their own code with the possibility of sharing later on [11].

Refactoring support in CDT

Refactoring code written in C/C++ does not come without its challenges. Schaefer [48] describes the macro preprocessor as the number one enemy of refactoring. With refactoring working at the source level, the preprocessor taking the source code and converting it into something that might be very different can cause problems. Developers can take this to account when implementing their code, keeping their code clean and minimizing the use of the preprocessor and then be able to use the refactorings.

In Figure 2.5, Rüegg [46] shows the history of refactoring methods in CDT. For a few years there only was support for one refactoring in CDT, the *Rename* refactoring, released in 2004. A simple example of the *Rename* refactoring is shown in Figure 2.6.

Four years after the *Rename* refactoring was released, in 2008, more refactoring methods were released:

- Getters and Setters
- Hide Method
- Implement Method
- Extract Constant
- Extract Function

This group of refactorings includes the result of work done in Switzerland, further described in Section 3.2. In 2009 the *Extract Local Variable* method was released and in 2011 a *Toggle Function* method.

2.3.4 Abstract Syntax Tree (AST)

Schorn [49] discusses different models to programmatically introspect C/C++ code using CDT Application Programming Interfaces (APIs). One of these models is using the Abstract Syntax Tree (AST) which contains every detail about the code. It is a tree of nodes which represent the syntax.

Figure 2.7 shows a simple example of how a CDT AST is structured. In the top right

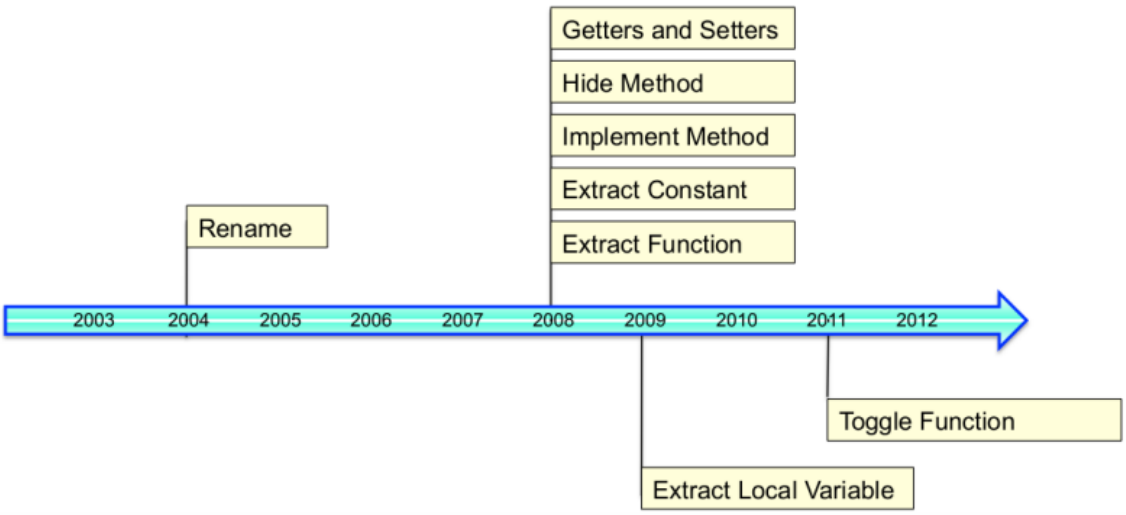

Figure 2.5: CDT Refactoring history [46]

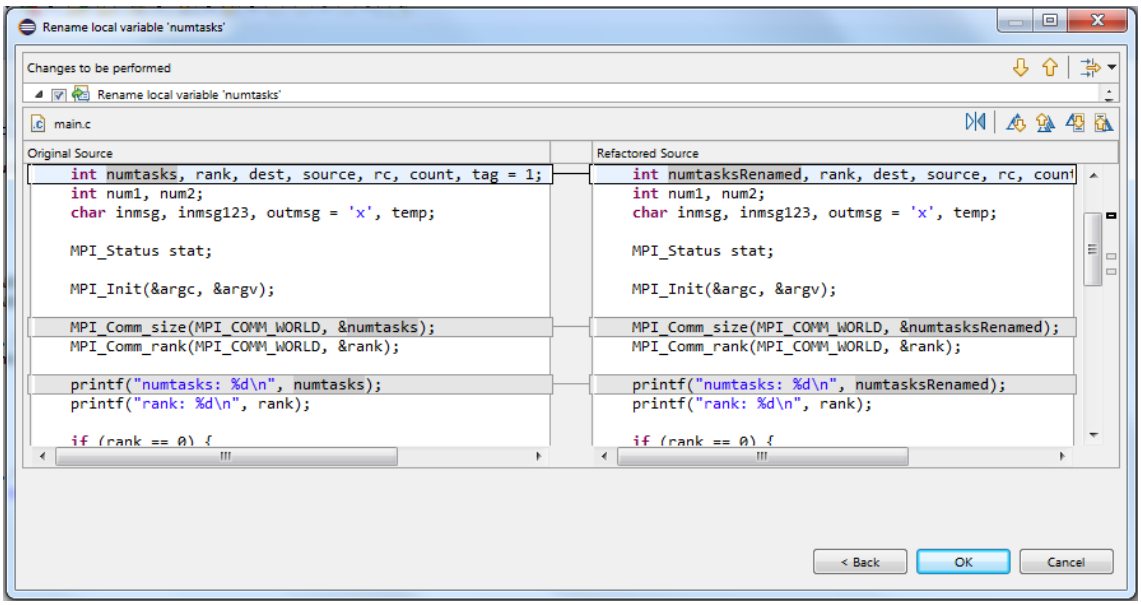

Figure 2.6: Rename refactoring example

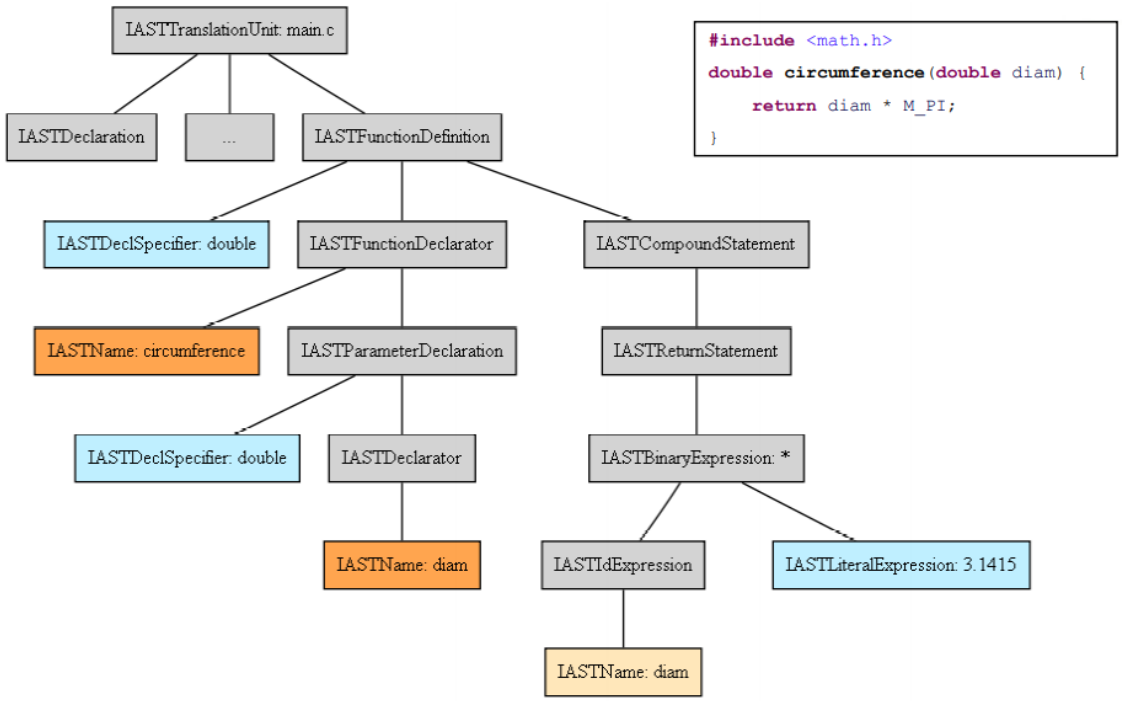

Figure 2.7: Example of a CDT Abstract Syntax Tree (AST) [49]

corner is a simple code example from which the AST is built. The top node is the `IASTTranslationUnit` itself for the file `main.c` and from there the tree shows the children for each node, such as the function definition node for the `circumference` function to the right. This goes on with each node having different types and representing different information until the leaf nodes are reached. The leaf nodes contain some identifiers that can be seen in the code (`double, circumference, diam`).

Rüegg [46] describes the usage of AST in CDT refactorings. To be able to get an AST of type `IASTTranslationUnit`[1] a Translation Unit (TU) (`ITranslationUnit`[2]), which is a source file with all included headers, is necessary. There are two ways to obtain an AST: either it can be created by calling the `getAST()` method from the `ITranslationUnit` class or by calling the `getAST(ITranslationUnit, IProgress-Monitor)` method from the `CRefactoring`[3] class. The latter method uses the AST cache of `CRefactoringContext` which inherits from LTK's `RefactoringContext`[4] class. To traverse the AST, two approaches are available:

- The methods `getParent()` and `getChildren()` from the `IASTNode`[5] class can be called.
- Using the visitor design pattern which is a subtype of `ASTVisitor`[6] and has visit methods for each node type.

[1] In package `org.eclipse.cdt.core.dom.ast.IASTTranslationUnit`
[2] In package `org.eclipse.cdt.core.mode.ITranslationUnit`
[3] In package `org.eclipse.cdt.internal.ui.refactoring.CRefactoring`
[4] In package `org.eclipse.ltk.core.refactoring.RefactoringContext`
[5] In package `org.eclipse.cdt.core.dom.ast.IASTNode`
[6] In package `org.eclipse.cdt.core.dom.ast.ASTVisitor`

3 Related work

This chapter discusses previous work related to refactoring HPC code and HPC refactoring tools (Section 3.1). It also covers existing work in relation to refactoring in Eclipse CDT in general (Section 3.2) and provides a summary (Section 3.3).

3.1 Refactoring and High-Performance Computing (HPC)

Kjolstad et al. [28] are the first to discuss in 2010 the need for refactoring tools for HPC programming and consider them to be lagging behind the tools improving the productivity of desktop programmers. While they do not provide a definition of HPC refactoring, they discuss the concerns of the HPC programmer with getting the most performance from the available hardware. They also mention other concerns they find equally, or even more, important which conflict with the performance goal, e.g. numerical stability, scalability, maintainability, and portability. These things the HPC programmer has to balance but does not have a lot of development tool support for that. They consider that refactoring tools can help HPC programmers explore performance optimizations safely since they do not change the behaviour of the code. A first HPC refactoring catalogue is provided with the intention of being built and worked on over time. The refactorings in the catalogue have a few different purposes. Some introduce parallelism, some prepare the code for parallelism, some improve the performance of parallel code and others help the HPC programmer speed up the sequential components of the parallel application. The catalogue is split up in two categories and within each category, a few refactoring suggestions are listed. The two categories and examples of the refactorings they contain are:

- Parallel MPI Refactorings
  - Make Communication Asynchronous
  - Create MPI Datatype from Struct
  - Remove Superfluous Barriers

- Sequential C Refactorings
  - Restrict Pointer
  - Split Loop
  - Organize Block as Load/Compute/Store

Each refactoring listed in the catalogue has a description of the refactoring and why it would be useful for HPC programming. Kjolstad et al. [28] also discuss frameworks for advanced refactoring and come to the conclusion that the best option is using Eclipse with CDT and Parallel Tools Platform (PTP)[1].

Basili et al. [3] discuss in 2008 the HPC community from a software engineer's perspective. They describe how the goals of computational scientists are to maximize scientific output rather than program performance. This is different from the computer scientist who focuses on improving code performance to save computing time. As many computational scientists receive their software training from other fellow scientists they generally lack formal training in software engineering. According to Basili et al. computational scientists are not driven entirely by performance and will not sacrifice significant maintainability for modest performance improvements. Their concern is rather portability for the codes to be able to run on multiple current and future HPC systems.

Egawa et al. acknowledge in 2014 [6] (and again in 2017 [7]) the difference between the goal of traditional refactoring and the refactoring needed for HPC code. While traditional refactoring can be used on HPC code to improve maintainability and readability of the code the main focus in HPC fields has always been performance and getting the most performance from the existing HPC systems. This means that many HPC applications are optimized for a specific HPC system which lessens the performance on other systems significantly. They move the focus to performance portability that can achieve high-performance in multiple systems to respond to the recent changes in the complexity and variety of the HPC supercomputers resulting in HPC programmers spending massive effort developing and migrating their codes to future systems. In 2014, Egawa et al. [6] first introduced the concept of HPC refactoring as follows:

> HPC refactoring is the technique to keep a high performance portability by identifying the system-specific parts of the code.

[1] The Eclipse PTP project [15] provides an IDE to support C/C++ and Fortran parallel application development. In addition to Fortran support, PTP mainly adds runtime features, such as a parallel debugger, and therefore it is not relevant for the static refactorings covered in this thesis.

In 2017 Egawa et al. [7] updated this definition:

> HPC refactoring is defined as the technique to keep a high-performance portability rather than readability and maintainability by identifying the system-specific parts of the code.

The group has developed a design of an HPC refactoring catalogue[2] which is a database of code optimization patterns that are system-aware. In the group's HPC refactoring catalogue there are more than 40 optimization patterns, mostly written in Fortran but a few in C. An example of the optimization patterns in the catalogue are:

- Extending loop length (7 patterns)
- Moving out statements from the loop (6 patterns)
- Improving thread level parallelism (1 pattern)
- Utilizing libraries (6 patterns)

The refactorings covered in this section transform code in a way that it stays in the same programming language. An example of refactoring that transforms code from one language to another is refactoring towards Graphics Processing Units (GPUs). Damevski and Muralimanohar [4] describe in 2011 a refactoring tool to extract GPU kernels, i.e. an Eclipse-based tool for extracting loops written in plain C to Compute Unified Device Architecture (CUDA) kernels. In 2019, a special section on Refactoring Software to Heterogeneous Parallel Platforms was published in The Journal of Supercomputing volume 75. It contains 12 articles and an editorial [24] containing research contributions towards refactoring software to heterogeneous parallel platforms, but not detailed refactorings nor tool implementations. These refactorings are out of scope for this thesis that covers refactoring staying within the same programming language, i.e. C/C++.

3.2 Refactoring in C/C++ Development Tooling (CDT)

Graf, Sommerland and Zgraggen [26] discuss their refactoring plug-ins for Eclipse CDT. They describe the current refactoring support in CDT as lacking. At the time of their work there was only one refactoring available within CDT, the *Rename*

[2]This catalogue used to be online, but is currently not accessible.

refactoring. They have implemented a plug-in which provides further refactorings for C++ developers and their work is now a part of CDT so it does not have to be downloaded separately. The plug-in includes following refactorings:

- Declare Method
- Extract Baseclass
- Extract Method
- Extract Subclass
- Hide Method
- Implement Method
- Move Field/Method
- Replace Number
- Separate Class

Their refactoring implementation makes use of the CDT AST and generates code from the AST using a specified writing visitor. The implementation described in Chapter 5 uses their implemented refactorings for CDT as a starting point.

3.3 Discussion

As shown in this chapter there is clearly a need for refactoring methods and the appropriate tools being incorporated into HPC programming. Code performance can often be a trade-off for code readability which also has negative impact on code maintainability. On the other hand code performance is also something to look at while refactoring the code so as to avoid having negative impact on performance. Scientists want significant improvement in code performance if they are to sacrifice code maintainability. Work has been done on creating a catalogue for refactoring methods specific to HPC code but few have been implemented as automatic or semi-automatic plug-ins for easy use. Even though different definitions and understandings of HPC refactoring are mentioned in Section 3.1, there is currently no generally accepted definition of HPC refactoring. In Chapter 4, a further definition is provided that tries to include all aspects of HPC refactoring.

4 Refactoring catalogue

The main contribution of this chapter is a refactoring catalogue for C/C++ HPC code. The underlying definition of HPC refactoring is provided in Section 4.1. Section 4.2 contains a list of the refactorings covered in this thesis and Section 4.3 describes the refactorings in further detail according to Fowler's format discussed in Section 2.2.2.

4.1 HPC refactoring definition

As described in Chapter 3 Kjolstad et al. [28] considers HPC refactoring tools for helping with performance optimization while Egawa et al. [7] defined HPC refactoring with respect to high-performance portability.

In this thesis, the definitions from Fowler [21], Kjolstad et al. [28], and Egawa et al. [7] are essentially combined and unified, resulting in a new definition of HPC refactoring:

> HPC refactoring is a technique that aims to make HPC code easier to read and modify or to improve performance or portability.

4.2 Refactoring catalogue overview

This is a list of the HPC refactorings which are described in this thesis. The refactorings are inspired by existing publication by Kjolstad et al. [28], lecture slides [45], and personal discussion [44]. The refactorings in this catalogue are just a start and currently focus on improving the performance and/or making the code easier to read and modify rather than the portability aspect from the definition, as portability aspects are already covered by Egawa et al. [7]. Each refactoring consists of a name along with a description, motivation, mechanics, and example(s). The refactorings described in the catalogue are as follows:

- Synchronous communication to asynchronous communication
- Multiple send to broadcast
- Multiple send to scatter
- Multiple receive to gather
- Multiple broadcast to all-gather

4.3 Refactoring descriptions

This section contains the details of the refactorings listed above. Each refactoring is described and detailed mechanics and examples show how they are applied to code.

4.3.1 Synchronous communication to asynchronous communication

Description

Replace the synchronous methods `MPI_Send` and `MPI_Recv` with the asynchronous methods `MPI_Isend` and `MPI_Irecv` along with a call to the `MPI_Wait` function.

Processor A:
```
MPI_Send(buffer)   //Send to processor B
*code*             //Executes after it is safe to re-use the send buffer
```

Processor B:
```
MPI_Recv(buffer)   //Receive from processor A
*code*             //Executes after it is safe to re-use the received buffer
```

Processor A:
```
MPI_Isend(buffer)  //Send to processor B
*code*             //Executes immediately
MPI_Wait(...)      //Wait until it is safe to re-use the send buffer
*rest of code*     //Executes after it is safe to re-use send buffer
```

Processor B:

```
MPI_Irecv(buffer)   //Receive from processor A
*code*              //Executes immediately
MPI_Wait(...)       //Wait until it is safe to use received buffer
*rest of code*      //Executes after it is safe to use received buffer
```

Motivation

The communication functions `MPI_Send` and corresponding `MPI_Recv` are blocking functions, meaning that when a processor calls either function the process is blocked until the call has returned. When working with multiple processors that means that when each processor gets to its communication call it is idle until all other processors have completed their work and gotten to the corresponding communication call. If the workload is split unevenly this means performance could be lost to wait time instead of being used for (maybe time consuming) calculations that are not dependent on the message buffer being used for communication. This refactoring aims to improve performance as the processors do not have to wait for the other processors to send or receive the message. The sending processor can execute code that is not dependent on the receive while waiting and vice versa for the receiving processors.

Mechanics

- Add new variable of type `MPI_Request`.
  - A parameter of type `MPI_Request` is required as an input parameter for `MPI_Isend` and `MPI_Irecv` but not for `MPI_Send` and `MPI_Recv` so if one does not exist a variable of that type needs to be created. Take care that it is visible within the scope of the refactored `MPI_Send`/`MPI_Recv` function calls.
  - If applicable, an already existing `MPI_Request` variable can be used instead of creating a new one.
- Compile.
- Replace `MPI_Send` with `MPI_Isend` directly followed by `MPI_Wait` or `MPI_Recv` with `MPI_Irecv` followed by `MPI_Wait` depending on which communication call should be refactored.
  - Identify the parameters of `MPI_Send` [38]/`MPI_Recv` [36] and correspond-

ing parameters for `MPI_Isend` [35]/`MPI_Irecv` [34]:

* `MPI_Isend` uses all the parameters from `MPI_Send` in the same order along with adding the *request* parameter as the last parameter. This *request* parameter is the `MPI_Request` parameter created in the first step of this refactoring.

* `MPI_Irecv` uses the first six parameters from `MPI_Recv` in the same order but the last parameter of `MPI_Recv` is not used. Instead the last parameter is the *request* parameter of type `MPI_Request`, mentioned above.

– Identify the input/output parameters needed for the `MPI_Wait` [39] call:

* First parameter of `MPI_Wait` is the *request* parameter mentioned above.

* Second parameter of `MPI_Wait` is *status*. This *status* parameter of type `MPI_Status` is a required parameter of `MPI_Recv` call, ordered in last place. Use that parameter from `MPI_Recv` as the second parameter of `MPI_Wait`.

- Compile and test.

- Move the `MPI_Wait` calls downwards in the code, as far as possible while taking data dependencies into account.

  – If this refactoring is to have any changes on performance the `MPI_Wait` calls need to be moved away from their original placement right below `MPI_Isend` or `MPI_Irecv`. `MPI_Wait` should be as far away from `MPI_Isend` or `MPI_Irecv` as possible.

  – Note that there is a difference between read or write access to the message buffer. In the case of `MPI_Isend` it is possible to still read while waiting for a write access, i.e. the `MPI_Wait` can be located after performing a read access to the send buffer but it must be located before performing a write access to the send buffer (preferably the last line before a write access occurs). For `MPI_Irecv` the `MPI_Wait` must be located before any access to the receive buffer occurs, whether it is read or write access (preferably `MPI_Wait` is located at the last line before read or write access to the receive buffer occurs).

- Compile and test.

Comments

For the program to run without any changes to the functionality after the refactoring, the code executed between the calls `MPI_Isend` or `MPI_Irecv` and the corresponding `MPI_Wait` has to take care to not use the buffer or anything depending on the `MPI_Isend` and `MPI_Irecv` calls. If this is not taken into account the program might be accessing data when it was not yet safe to do so. The `MPI_Wait` must therefore be placed before this data is accessed (according to the different read or write access restrictions described previously in the mechanics), i.e. before the message buffer is used again in the corresponding block of the program.

Examples

This section contains a few simple examples of the refactoring in action. Listing 4.1 shows the code before any alterations. Listing 4.2 displays the code after the refactoring has been applied once to replace `MPI_Send`. Listing 4.3 shows the changes that are made to the code when the refactoring is applied once to replace `MPI_Recv`. The last code example in Listing 4.4 shows the code after the refactoring has been applied twice, replacing both `MPI_Send` and `MPI_Recv`.

One processor sending a message to another processor using blocking `MPI_Send` and `MPI_Recv`

Listing 4.1 shows a simple example of a program that uses two processors. One sends a message and the other receives said message, both using blocking communication functions. The print functions following the MPI communication calls show the difference in read or write access to the message buffer.

Listing 4.1: Before refactoring – Simple code example with synchronous communication

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char** argv){
5
6    int numtasks, rank, dest, source, rc, tag=1;
7    char inmsg, outmsg;
8
9    MPI_Status Stat;
10
11   MPI_Init(&argc, &argv);
```

```
12
13    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14
15    if (rank == 0){
16      dest = 1;
17      source = 1;
18      MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
19
20      if (numtasks % 2 == 0){
21        outmsg = 'e';
22      } else {
23        outmsg = 'o';
24      }
25      rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
           MPI_COMM_WORLD);
26      printf("Send returned\n");
27      printf("Safe to read outmsg: %c\n", outmsg);
28      printf("Safe to modify outmsg: %c\n", ++outmsg);
29    }
30    else if (rank == 1){
31      dest = 0;
32      source = 0;
33      rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
           MPI_COMM_WORLD, &Stat);
34      printf("Receive returned\n");
35      printf("Safe to read inmsg: %c\n", inmsg);
36      printf("Safe to modify inmsg: %c\n", ++inmsg);
37      if (--inmsg == 'e'){
38        printf("The number of processors is even\n");
39      } else {
40        printf("The number of processors is odd\n");
41      }
42    }
43
44    MPI_Finalize();
45  }
```

Replace `MPI_Send` only

Listing 4.2 shows the code from Listing 4.1 modified so that it uses a non-blocking `MPI_Isend` followed by a `MPI_Wait` instead of the blocking `MPI_Send`. The `MPI_Wait` call has been moved to line 27 in the block containing `MPI_Isend` which means that the first processor will execute the code in line 25 since it does not access the send buffer and line 26 as well since it only performs a read access to the send buffer. Line 28 performs a write access to the send buffer so the `MPI_Wait` call is placed before that line.

Listing 4.2: After refactoring – `MPI_Send` replaced in code example

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char** argv){
5
6    MPI_Request isendRequest;
7    int numtasks, rank, dest, source, rc, tag=1;
8    char inmsg, outmsg;
9
10   MPI_Status Stat;
11
12   MPI_Init(&argc, &argv);
13
14   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16   if (rank == 0){
17     dest = 1; source = 1;
18     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
19     if (numtasks % 2 == 0){
20       outmsg = 'e';
21     } else {
22       outmsg = 'o';
23     }
24     rc = MPI_Isend(&outmsg, 1, MPI_CHAR, dest, tag,
          MPI_COMM_WORLD, &isendRequest);
25     printf("Send returned\n");
26     printf("Safe to read outmsg: %c\n", outmsg);
27     MPI_Wait(&isendRequest, &Stat);
28     printf("Safe to modify outmsg: %c\n", ++outmsg);
29   }
30   else if (rank == 1){
31     dest = 0; source = 0;
```

```
32      rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
           MPI_COMM_WORLD, &Stat);
33      printf("Receive returned\n");
34      printf("Safe to read inmsg: %c\n", inmsg);
35      printf("Safe to modify inmsg: %c\n", ++inmsg);
36      if (--inmsg == 'e'){
37        printf("The number of processors is even");
38      } else {
39        printf("The number of processors is odd");
40      }
41    }
42
43    MPI_Finalize();
44  }
```

Replace `MPI_Recv` only

Listing 4.3 shows the code from Listing 4.1 modified so that it uses a non-blocking `MPI_Irecv` followed by a `MPI_Wait` instead of the blocking `MPI_Recv`. As with Listing 4.2 the `MPI_Wait` call has been moved away from the blocking communication call so that it is located directly before the received buffer is used again, but in this case it needs to be located before any access is performed to the receive buffer, read or write. This means that the second processor will execute the code in line 32 and then wait for it to be safe to access the receive buffer.

Listing 4.3: After refactoring – `MPI_Recv` replaced in code example

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char** argv){
5
6    MPI_Request irecvRequest;
7    int numtasks, rank, dest, source, rc, tag=1;
8    char inmsg, outmsg;
9
10   MPI_Status Stat;
11
12   MPI_Init(&argc, &argv);
13
14   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
```

```
16    if (rank == 0){
17      dest = 1; source = 1;
18      MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
19      if (numtasks % 2 == 0){
20        outmsg = 'e';
21      } else {
22        outmsg = 'o';
23      }
24      rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
           MPI_COMM_WORLD);
25      printf("Send returned\n");
26      printf("Safe to read outmsg: %c\n", outmsg);
27      printf("Safe to modify outmsg: %c\n", ++outmsg);
28    }
29    else if (rank == 1){
30      dest = 0; source = 0;
31      rc = MPI_Irecv(&inmsg, 1, MPI_CHAR, source, tag,
           MPI_COMM_WORLD, &irecvRequest);
32      printf("Receive returned\n");
33      MPI_Wait(&irecvRequest, &Stat);
34      printf("Safe to read inmsg: %c\n", inmsg);
35      printf("Safe to modify inmsg: %c\n", ++inmsg);
36      if (--inmsg == 'e'){
37        printf("The number of processors is even");
38      } else {
39        printf("The number of processors is odd");
40      }
41    }
42
43    MPI_Finalize();
44  }
```

Replace both `MPI_Send` and `MPI_Recv`

Listing 4.4 shows the code from Listing 4.1 when both `MPI_Send` and `MPI_Recv` have been replaced with `MPI_Isend` and `MPI_Irecv` respectively followed with `MPI_Wait`. For the sending processor the `MPI_Wait` call has been moved downwards away from `MPI_Isend` to be directly before the send buffer is modified in line 30 but after it has been read in line 28. For the receiving processor the `MPI_Wait` call has been moved downwards away from `MPI_Irecv` until the received buffer needs to be accessed again, whether it is to read it or to modify, i.e. the `MPI_Wait` call is moved to line 37. This is essentially the result when the refactoring is applied twice.

Listing 4.4: After refactoring – Both `MPI_Send` and `MPI_Recv` replaced

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char** argv){
5
6    MPI_Request newRequest;
7    int numtasks, rank, dest, source, rc, tag=1;
8    char inmsg, outmsg;
9
10   MPI_Status Stat;
11
12   MPI_Init(&argc, &argv);
13
14   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16   if (rank == 0){
17     dest = 1;
18     source = 1;
19     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
20
21     if (numtasks % 2 == 0){
22       outmsg = 'e';
23     } else {
24       outmsg = 'o';
25     }
26     rc = MPI_Isend(&outmsg, 1, MPI_CHAR, dest, tag,
          MPI_COMM_WORLD, &newRequest);
27     printf("Send returned\n");
28     printf("Safe to read outmsg: %c\n", outmsg);
29     MPI_Wait(&newRequest, &Stat);
30     printf("Safe to modify outmsg: %c\n", ++outmsg);
31   }
32   else if (rank == 1){
33     dest = 0;
34     source = 0;
35     rc = MPI_Irecv(&inmsg, 1, MPI_CHAR, source, tag,
          MPI_COMM_WORLD, &newRequest);
36     printf("Receive returned\n");
37     MPI_Wait(&newRequest, &Stat);
38     printf("Safe to read inmsg: %c\n", inmsg);
39     printf("Safe to modify inmsg: %c\n", ++inmsg);
40     if (--inmsg == 'e'){
```

```
41          printf("The number of processors is even");
42        } else {
43          printf("The number of processors is odd");
44        }
45      }
46
47      MPI_Finalize();
48  }
```

4.3.2 Multiple send to broadcast

Description

Replace multiple `MPI_Send` and corresponding `MPI_Recv` calls with one `MPI_Bcast` call.

Processor A:
code
for every receiving processor
(e.g. in a loop):
`MPI_Send(...)` //Blocking send to other processors
code

Other processors:
code
`MPI_Recv(...)` //Receive from processor A
code

Processor A:
code
`MPI_Bcast(...)` //Send message to a group of other processors simultaneously
code

Other processors:
code
`MPI_Bcast(...)` //The group waits for a message from processor A
code

Motivation

When the same message needs to be sent from one processor to multiple other processors it may be done using a loop, e.g. a for loop. The `MPI_Send` call would then be called for each run of the loop, making the calls as many as the number of processors the message needs to be sent to. With `MPI_Send` being a blocking communication call, each call in the for loop blocks the program while waiting for the send buffer to be safe for re-use, e.g. by the respective processor sending a received signal via `MPI_Recv`. This can take a lot of time for the sending processor sending multiple unicast messages to the other processors and even the receiving processors that are late in the loop of messages being sent and thus have to wait longer than other processors to receive the message. By transforming these loops of blocking `MPI_Send` and corresponding `MPI_Recv` calls to one collective `MPI_Bcast` call that sends the same message to all other processors in the same communicator simultaneously, this refactoring aims to improve performance of the software and reduces the amount of messages sent over the communication infrastructure if it physically supports broadcast communication.

Mechanics

- Create a new communicator comprising the sending processor and all receiving processors contained in the loop.
  - If a communicator comprising all the desired processors already exists it can be used and a new one does not need to be created.
  - If the message should be broadcast to all processors the communicator `MPI_COMM_WORLD` should be used and a new communicator does not need to be created.
- Compile.
- Replace the blocking `MPI_Send` call and the corresponding blocking `MPI_Recv` calls with one `MPI_Bcast` using the communicator created in first step. For this transformation there are two possible actions:
  If there is a for loop containing only the `MPI_Send` call from the source, replace the whole for loop containing `MPI_Send` with one `MPI_Bcast` call. Replace the corresponding `MPI_Recv` with `MPI_Bcast`.
  Else, replace the `MPI_Send` call with `MPI_Bcast` and corresponding `MPI_Recv` with corresponding `MPI_Bcast`. `Move MPI_Bcast` outside of loop.

  - Identify the input/output parameters of `MPI_Send` [38], `MPI_Recv` [36], and how to map them as parameters for `MPI_Bcast` [32]:
    * First parameter of all three communication calls is the address of the buffer, i.e. the buffer used in `MPI_Send` or `MPI_Recv` (depending on which call is being switched out) is used as buffer of `MPI_Bcast`.
      · If the `MPI_Send` and `MPI_Recv` calls use different message buffers, here might be an opportunity to unify the separated send and receive buffer into one buffer to be able to merge the `MPI_Bcast` calls later.
    * Second parameter of all three communication calls is *count*, number of entries in the buffer. The *count* parameter of `MPI_Send` or `MPI_Recv` is used as the *count* parameter of `MPI_Bcast`.
    * Third parameter of all three is *datatype*, i.e. the *datatype* parameter of `MPI_Send` or `MPI_Recv` is used as the *datatype* parameter of `MPI_Bcast`.
    * Fourth parameter of `MPI_Bcast` is *root* which is the rank of the broadcast root. The rank of the sending processor is used as this parameter.
    * Fifth parameter of `MPI_Bcast` is *comm*, the communicator. This parameter is also the sixth parameter of `MPI_Send` and `MPI_Recv`, i.e. use *comm* from `MPI_Send` or `MPI_Recv` as the *comm* parameter for `MPI_Bcast`.

- Compile and test.
- If possible, merge `MPI_Bcast` calls (e.g. move them outside of if statements) so that all processors that previously called either `MPI_Send` or `MPI_Recv` call `MPI_Bcast` at the same location in the code (merged to one line instead of two).
  - For this to be possible, the two `MPI_Bcast` calls need to use the same message buffer.
- Compile and test.
- Delete empty for loops and if/else statements.
- Compile and test.

Comments

This refactoring can become complicated or even not suitable if the sending processor is doing anything other than sending the same data to the other processors (if the data is not the same, but the sender rather sends different partitions of an array to different processes, the Multiple receive to gather refactoring in Section 4.3.4 might be applicable). The two examples below show the simplest cases where this refactoring is applicable. Even a small change to these examples might make the refactoring impossible to apply, e.g. if for each `MPI_Send` in the for loop something else depending on the loop count needs to be done directly after each message is sent or the data to be sent varies from receiver to receiver.

The first step of the mechanics is creating a new communicator if an applicable one does not exist already. This can cause overhead and should be taken into account for each use when determining if this refactoring is suitable with respect to increasing performance.

Examples

This section contains two examples of the refactoring being applied to a simple code. Listing 4.5 and Listing 4.7 show two different code examples and listings 4.6 and 4.8 show the result after the changes have been made.

Send message from one processor to all other processors with no other processing (simplest case)

Listing 4.5 shows the simplest code example. In this example the if statement contains only a for loop containing the `MPI_Send` calls as can be seen in lines 16–20. In the same way the else statement in lines 21–23 contains only the `MPI_Recv` calls performed by all the receiving processors.

Listing 4.5: Before refactoring – Simple code example with multiple synchronous send

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(int argc, char** argv){
5
6   int numtasks, rank, tag=1, source=0;
7   char msg = 'x';
8
```

```
9     MPI_Status Stat;
10
11    MPI_Init(&argc, &argv);
12
13    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
14    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16    if (rank == source) {
17      for (int i = 1; i < numtasks; i++){
18          MPI_Send(&msg, 1, MPI_CHAR, i, tag,
               MPI_COMM_WORLD);
19      }
20    }
21    else {
22      MPI_Recv(&msg, 1, MPI_CHAR, source, tag,
           MPI_COMM_WORLD, &Stat);
23    }
24
25    MPI_Finalize();
26  }
```

Refactor to `MPI_Bcast`

Listing 4.6 shows the result of the refactoring being applied to the code in Listing 4.5. As shown the whole if/else block (along with the for loop and the `MPI_Send`/`MPI_Recv` calls) has been replaced by a single `MPI_Bcast` call in line 16 and therefore all steps of the refactoring have been applied. The `MPI_Bcast` call uses the `MPI_COMM_WORLD` communicator since all available processors are being used.

Listing 4.6: After refactoring – Code modified to use `MPI_Bcast`

```
1   #include <stdio.h>
2   #include <mpi.h>
3
4   int main(int argc, char** argv){
5
6     int numtasks, rank, rc, tag=1, source=0;
7     char msg = 'x';
8
9     MPI_Status Stat;
10
11    MPI_Init(&argc, &argv);
```

```
12
13    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
14    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16    MPI_Bcast(&msg, 1, MPI_CHAR, source, MPI_COMM_WORLD);
17
18    MPI_Finalize();
19  }
```

Send message from one processor to all other processors with some other calculations

Listing 4.7 shows an example of code that does a little more than just send and receive. Simple print functions have been added to both the sending and receiving processors (lines 20 and 29) and an insignificant increment to a count variable is done in every other loop (lines 21–23). Line 24 makes the software wait for one second before continuing. These extra lines do not hold any real value for the software (except informational printing) but are added for the purpose of the example.

Listing 4.7: Before refactoring – Code example with other calculations

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char** argv){
5
6    int numtasks, rank, rc, tag=1, source=0;
7    char inmsg, outmsg = 'x';
8    int count = 0;
9
10   MPI_Status Stat;
11
12   MPI_Init(&argc, &argv);
13
14   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
15   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16
17   if (rank == source){
18     for (int i = 1; i < numtasks; i++){
19       MPI_Send(&outmsg, 1, MPI_CHAR, i, tag,
           MPI_COMM_WORLD);
20       printf("Source has sent message to rank %d.\n", i);
```

```
21        if (i%2 == 0){ //For every other send, increment
             count
22            count = count + 1;
23        }
24        sleep(1);
25      }
26    }
27    else {
28      MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
           MPI_COMM_WORLD, &Stat);
29      printf("Rank %d has received message from
           source.\n", rank);
30    }
31
32    MPI_Finalize();
33  }
```

Refactor to `MPI_Bcast`

Listing 4.8 shows the code from Listing 4.7 after the refactoring has been applied. For this example the last step of the refactoring is not applicable since the `MPI_Bcast` call cannot be moved outside of the if/else statements. This is because of the additional code being executed inside the for loop and if/else statements. Both the print function in line 20 and if statement condition in line 21 depend on the for loop count so the loop cannot be eliminated as well as the print function in line 30 also depends on the current rank.

Listing 4.8: After refactoring – More complex code modified to use `MPI_Bcast`

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char** argv){
5
6    int numtasks, rank, rc, tag=1, source=0;
7    char inmsg, outmsg = 'x';
8    int count = 0;
9
10   MPI_Status Stat;
11
12   MPI_Init(&argc, &argv);
13
```

```
14    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
15    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16
17    if (rank == source){
18      MPI_Bcast(&outmsg, 1, MPI_CHAR, source,
          MPI_COMM_WORLD);
19      for (int i = 1; i < numtasks; i++){
20        printf("Source has sent message to rank %d.\n", i);
21        if (i%2 == 0){
22        count = count + 1; //For every other send,
23                           // increment count
24        }
25        sleep(1);
26      }
27    }
28    else {
29      MPI_Bcast(&inmsg, 1, MPI_CHAR, source,
          MPI_COMM_WORLD);
30      printf("Rank %d has received message from
          source.\n", rank);
31    }
32
33    MPI_Finalize();
34  }
```

4.3.3 Multiple send to scatter

Description

Replace multiple `MPI_Send` and corresponding `MPI_Recv` calls with one `MPI_Scatter` call when a different message has to be sent to each processor.

Processor A:
code
for every receiving processor:
set message to be sent
`MPI_Send(...)` //Blocking send to other processors
code

Other processors:
code
`MPI_Recv(...)` //Receive from processor A
code

Processor A:
code
collect messages to be sent
to an array
`MPI_Scatter(...)` //Send array of messages to a group of other processors simultaneously
code

Other processors:
code
`MPI_Scatter(...)` //The group waits for their message from processor A
code

Motivation

When a processor needs to send a message to multiple other processors it may be done using a loop, e.g. a for loop. As discussed in Section 4.3.2 this can take a lot of time with overhead for each operation. What is different here is that the message being sent is not the same for all receiving processors. The solution is replacing the multiple blocking `MPI_Send` and corresponding `MPI_Recv` with preferably only one collective `MPI_Scatter` call. The `MPI_Scatter` call sends from one processor different messages to all other processors in the same communicator and receives for all other processes simultaneously, thus aiming to improve performance and increase understandability of the software.

Mechanics

- Create a new communicator comprising the sending processor and all receiving processors contained in the loop.
  - If a communicator comprising all the desired processors already exists it can be used and a new one does not need to be created.
  - If the message array is to be scattered to all available processors the

communicator `MPI_COMM_WORLD` should be used and a new communicator does not need to be created.

- Compile.
- If it does not anyway exist yet: create an array to contain all the messages that will be sent.
  - The size of the array should be the same as number of processors in the communicator used multiplied with the number of elements sent in each send/receive communication.
- Compile.
- Populate the array with the messages that will be sent.
  - Take care that the elements in the array are in the same order as processor ranks, i.e. the first n elements in the array are sent to processor with rank 0, the next n elements in the array are sent to processor with rank 1, etc. where n is the number of elements sent to each processor [51].
- Compile.
- Replace the blocking `MPI_Send` call and the corresponding blocking `MPI_Recv` calls with `MPI_Scatter`.
  - Identify the input/output parameters of `MPI_Send` [38], `MPI_Recv` [36], and corresponding parameters for `MPI_Scatter` [37]:
    - First parameter of `MPI_Scatter` is the address of the send buffer. The array created and populated in the previous steps of this refactoring should be used for this parameter.
    - Second parameter of `MPI_Scatter` is the number of elements sent to each process. The corresponding parameter is the second parameter from the `MPI_Send` call which contains the number of elements in the send buffer.
    - Third parameter of `MPI_Scatter` is the data type of the send buffer elements and has a corresponding third parameter from the `MPI_Send` call that should be used.
    - Fourth parameter of `MPI_Scatter` is the address of the receive buffer, the receive buffer from the `MPI_Recv` call (the first parameter of

`MPI_Recv`) should be used.

  - Fifth parameter of `MPI_Scatter` is the number of elements in the receive buffer, the number of elements sent should be used for this parameter.

  - Sixth parameter of `MPI_Scatter` is the data type of the receive buffer elements and has a corresponding third parameter from the `MPI_Recv` call that should be used.

  - Seventh parameter of `MPI_Scatter` is the rank of the sending process which is the same rank as performs all the `MPI_Send` calls.

  - Eighth parameter of `MPI_Scatter` is the communicator. Refer to the first step of this refactoring in deciding which communicator to use.

- Compile and test.

- If possible, move `MPI_Scatter` outside of if statements so that all processors that previously called either `MPI_Send` or `MPI_Recv` call by exactly one `MPI_Scatter` call.

- Compile and test.

- Delete empty for loops and if/else statements.

- Compile and test.

Comments

The first step of the mechanics is creating a new communicator if an applicable one does not exist already. This can cause overhead and should be taken into account for each use when determining if this refactoring is suitable with respect to increasing performance.

Examples

This section contains an example of the refactoring being applied to a simple code. Listing 4.9 shows a simple code example where a different message is being sent to multiple processors and Listing 4.10 shows the result after applying the changes.

Send different messages from one processor to all other processors

Listing 4.9 shows a simple code example where the if statement in lines 26–34 contains two for loops. The message is calculated and used to populate an array in the first for loop in lines 27–30. The messages are then sent in the second for loop in lines 31–33 using `MPI_Send` for each send. Each `MPI_Send` sends one element from an array to the receiving processor. The else statement in lines 35–37 contains only the `MPI_Recv` call.

Listing 4.9: Before refactoring – Simple example sending different message to multiple processors

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include <math.h>
5  #include <time.h>
6
7  int main(int argc, char** argv){
8
9    int numtasks, rank, i, tag=1;
10   int randomInt, msgsize=1;
11   int *randomInts = NULL;
12
13   MPI_Status stat;
14
15   MPI_Init(&argc, &argv);
16
17   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
18   randomInts = malloc(sizeof(int)*numtasks*msgsize);
19   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20
21   printf("numtasks: %d\n", numtasks);
22   printf("rank: %d\n", rank);
23
24   srand( time(NULL) + rank );
25
26   if (rank == 0){
27     for (int i = 1; i < numtasks; i++){
28       randomInt = rand() % 10;
29       randomInts[i] = randomInt;
30     }
31     for (int i = 1; i < numtasks; i++){
32       MPI_Send(&randomInts[i], msgsize, MPI_INT, i, tag,
           MPI_COMM_WORLD);
```

```
33      }
34    }
35    else{
36      MPI_Recv(&randomInt, msgsize, MPI_INT, 0, tag,
          MPI_COMM_WORLD, &stat);
37    }
38    MPI_Finalize();
39  }
```

Refactor to `MPI_Scatter`

Listing 4.10 shows the result of the refactoring being applied to the code in Listing 4.9. The for loop in lines 27–30 now only populates an array that was already existing in the first example and the `MPI_Scatter` call has been moved out of the if/else statements.

Listing 4.10: After refactoring – Sending different messages to processors using `MPI_Scatter`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include <math.h>
5  #include <time.h>
6
7  int main(int argc, char** argv){
8
9    int numtasks, rank, i, tag=1;
10   int randomInt, msgsize=1;
11   int *randomInts = NULL;
12
13   MPI_Status stat;
14
15   MPI_Init(&argc, &argv);
16
17   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
18   randomInts = malloc(sizeof(int)*numtasks*msgsize);
19   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20
21   printf("numtasks: %d\n", numtasks);
22   printf("rank: %d\n", rank);
23
```

```
24    srand( time(NULL) + rank );
25
26    if (rank == 0){
27      for (int i = 1; i < numtasks; i++){
28        randomInt = rand() % 10;
29        randomInts[i] = randomInt;
30      }
31    }
32    MPI_Scatter(randomInts, 1, MPI_INT, &randomInt, 1,
         MPI_INT, 0, MPI_COMM_WORLD);
33    MPI_Finalize();
34  }
```

4.3.4 Multiple receive to gather

Description

Replace multiple `MPI_Recv` and corresponding `MPI_Send` calls with one `MPI_Gather` call when one processor is getting messages sent from multiple other processors.

Processor A:
code
for every sending processor:
`MPI_Recv(...)` //Receive messages from other processors
code

Other processors:
code
set message to be sent
`MPI_Send(...)` //Send a message to processor A
code

Processor A:
code
`MPI_Gather(...)` //Gather messages from all other processors simultaneously
process received array of messages
code

Other processors:
code
set message to be sent
`MPI_Gather(...)` //Send a message to processor A
code

Motivation

When a processor needs to collect a message from multiple other processors it may be done using a loop, e.g. a for loop. There the receiving processor uses the `MPI_Recv` call for every processor it needs to collect a message from while the sending processors use `MPI_Send`. This is essentially the opposite of the refactorings in sections 4.3.2 and 4.3.3 where the root processor is sending a message to all other processors. The same goes here for idle time while waiting for every blocking `MPI_Send`/`MPI_Recv` pair and the overhead that comes with each operation. Instead this can be done with one collective `MPI_Gather` call where one processor gathers message from all other processors in the communicator.

Mechanics

- Create a new communicator comprising the sending processor and all receiving processors contained in the loop.
  - If a communicator comprising all the desired processors already exists it can be used and a new one does not need to be created.
  - If the root processor is receiving messages from all other available processors the communicator `MPI_COMM_WORLD` should be used and a new communicator does not need to be created.
- Compile.
- If it does not anyway exist yet: create an array to contain all the messages that will be received.
  - If such an array exists already then a new one does not need to be declared.
  - The size of the array should be the same as the number of processors in the communicator used multiplied with the number of elements sent in

each send/receive communication.

- Compile.
- Replace the blocking `MPI_Send` call and the corresponding blocking `MPI_Recv` calls with a `MPI_Gather` call. If needed, process the received message array. (The former receiving parameter from `MPI_Recv` should be used as the respective part of the received array, i.e. the first n elements in the array are received from processor with rank 0, the next n elements in the array are received from processor with rank 1, etc. where n is the number of elements received from each processor [51].)
  - Identify the input/output parameters of `MPI_Send` [38], `MPI_Recv` [36], and corresponding parameters for `MPI_Gather` [33]:
    - First parameter of `MPI_Gather` is the starting address of the send buffer. This can be found as the first parameter of the `MPI_Send` call.
    - Second parameter of `MPI_Gather` is the number of elements in the send buffer which can also be found as the second parameter of the `MPI_Send` call.
    - Third parameter of `MPI_Gather` is the data type of the send buffer elements which can also be found as the third parameter of the `MPI_Send` call.
    - Fourth parameter of `MPI_Gather` is the address of the receive buffer. The array created in the previous step of this refactoring should be used for this parameter.
    - Fifth parameter of `MPI_Gather` is the number of elements for any single receive which can also be found as the second parameter of the `MPI_Recv` call.
    - Sixth parameter of `MPI_Gather` is the data type of the receive buffer elements which can also be found as the third parameter of the `MPI_Recv` call.
    - Seventh parameter of `MPI_Gather` is the rank of the receiving process which can also be found as the fourth parameter of the `MPI_Recv` call.
    - Eighth parameter of `MPI_Gather` is the communicator. Refer to the first step of this refactoring in deciding which communicator to use.

- Compile and test.
- If possible, move `MPI_Gather` outside of if/else statements so that all processors that previously called either `MPI_Send` or `MPI_Recv` call by exactly one `MPI_Gather` call.
- Compile and test.
- Delete empty for loops and if/else statements.
- Compile and test.

Comments

The first step of the mechanics is creating a new communicator if an applicable one does not exist already. This can cause overhead and should be taken into account for each use when determining if this refactoring is suitable with respect to increasing performance.

Examples

This section contains an example of the refactoring being applied to a simple code. Listing 4.11 shows a simple code example where different messages are being sent to one processor and Listing 4.12 shows the result after the changes have been made.

Multiple processors sending a message to one processor

Listing 4.11 shows a simple code example where the if statement in lines 26–29 calculates the message for each processor (except for the root) in line 27 and sends it in line 28 using `MPI_Send`. The else statement in lines 30–34 contains only one for loop where the root processor receives a message (into an array) from all other available processors using `MPI_Recv`.

Listing 4.11: Before refactoring – One processor receiving messages from all other processors

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include <math.h>
5  #include <time.h>
6
7  int main(int argc, char** argv){
8
9    int numtasks, rank, i, tag=1;
10   int randomInt, msgsize=1;
11   int *randomInts = NULL;
12
13   MPI_Status stat;
14
15   MPI_Init(&argc, &argv);
16
17   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
18   randomInts = malloc(sizeof(int)*numtasks*msgsize);
19   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20
21   printf("numtasks: %d\n", numtasks);
22   printf("rank: %d\n", rank);
23
24   srand( time(NULL) + rank );
25
26   if (rank != 0){
27     randomInt = rand() % 10;
28     MPI_Send(&randomInt, msgsize, MPI_INT, 0, tag,
          MPI_COMM_WORLD);
29   }
30   else{
31     for (int i = 1; i < numtasks; i++){
32       MPI_Recv(&randomInts[i], msgsize, MPI_INT, i, tag,
            MPI_COMM_WORLD, &stat);
33     }
34   }
35
36   MPI_Finalize();
37 }
```

Refactor to `MPI_Gather`

Listing 4.12 shows the result of the refactoring being applied to the code in Listing 4.11. The if statement in lines 26-28 now only calculates the message for each processor (except for the root) and line 29 contains an `MPI_Gather` call used by all processors. The messages were already being received into an array so a new one does not need to be created, but rather the existing one is used.

Listing 4.12: After refactoring – One processor gathering messages from multiple other processors using `MPI_Gather`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include <math.h>
5  #include <time.h>
6
7  int main(int argc, char** argv){
8
9    int numtasks, rank, i, tag=1;
10   int randomInt, msgsize=1;
11   int *randomInts = NULL;
12
13   MPI_Status stat;
14
15   MPI_Init(&argc, &argv);
16
17   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
18   randomInts = malloc(sizeof(int)*numtasks*msgsize);
19   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20
21   printf("numtasks: %d\n", numtasks);
22   printf("rank: %d\n", rank);
23
24   srand( time(NULL) + rank );
25
26   if (rank != 0){
27     randomInt = rand() % 10;
28   }
29   MPI_Gather(&randomInt, msgsize, MPI_INT, randomInts,
       msgsize, MPI_INT, 0, MPI_COMM_WORLD);
30
31   MPI_Finalize();
32 }
```

4.3.5 Multiple broadcast to all-gather

Description

Replace multiple `MPI_Bcast` calls with one `MPI_Allgather` call.

All processors:
code
for every processor in a communicator:
`MPI_Bcast(...)` //Broadcast data to all other processor in the communicator
code

All processors:
code
`MPI_Allgather(...)` //All processors send their data and receive the data from other processors in one call
code

Motivation

When data needs to by collected by all processors from all processors it may be done using a loop, e.g. a for loop where, for each run of the loop, `MPI_Bcast` is called. This makes the number of `MPI_Bcast` calls the same as the number of processors in the communicator. Each operation has an overhead so rather than calling an MPI operation, in this case `MPI_Bcast`, multiple times it is better to call once an MPI collective, here `MPI_Allgather` [45].

Mechanics

- Identify the data being broadcast from each processor.
- If it does not anyway exist yet: declare a new array to hold the data to broadcast.
- Compile.

- Identify the parameters in the `MPI_Bcast` calls [32]:
  - First parameter of `MPI_Bcast` is the buffer containing the data to broadcast.
  - Second parameter of `MPI_Bcast` is an integer *count* describing the number of entries in the buffer.
  - Third parameter of `MPI_Bcast` is a handle *datatype* describing the data type of the buffer.
  - Fourth parameter of `MPI_Bcast` is an integer representing the rank of broadcast root.
  - Fifth parameter of `MPI_Bcast` is a handle *comm* representing the communicator.
- Replace the `MPI_Bcast` with an `MPI_Allgather` call outside of the for loop. It should be located before the loop if the `MPI_Bcast` call is at the top of the loop and after the loop if `MPI_Bcast` is last in the loop. The parameters of this call are as follows [31]:
  - First parameter of `MPI_Allgather` is starting address of the send buffer. Set this parameter as `MPI_IN_PLACE`.
  - Second parameter of `MPI_Allgather` is an integer describing the number of elements in the send buffer. Set this parameter as the *count* parameter from the `MPI_Bcast` call.
  - Third parameter of `MPI_Allgather` is a handle describing the data type of the send buffer elements. Set this parameter as the *datatype* parameter from the `MPI_Bcast` call.
  - Fourth parameter of `MPI_Allgather` is the address of the receive buffer. Set this parameter as the array declared in the second step of the refactoring.
  - Fifth parameter of `MPI_Allgather` is an integer describing the number of elements received from any process. Set this parameter also as the *count* parameter from the `MPI_Bcast` call.
  - Sixth parameter of `MPI_Allgather` is a handle describing the data type of the receive buffer elements. Set this parameter also as the *datatype* parameter from the `MPI_Bcast` call.

  - Seventh parameter of `MPI_Allgather` is a handle representing the communicator. Set this parameter as the *comm* parameter from the `MPI_Bcast` call.

- Compile and test.
- If the for loop is now empty, delete it.
- Compile and test.

Comments

This refactoring is only applicable when the `MPI_Allgather` call can be removed from the for loop without affecting the functionality of the code. Therefore it will not be applicable for every situation, rather the simpler ones.

Examples

This section contains an example of the refactoring being applied to a simple code. Listing 4.13 shows a simple code example and Listing 4.14 shows the result after the changes have been made.

Each processor broadcasting a message to all other processors

Listing 4.13 shows the simplest code example. As seen in lines 29–31 the for loop contains only the `MPI_Bcast` call, sending random numbers calculated at each processors to all other processors in the communicator. The example already declares an array which will hold all the numbers after the broadcasts.

Listing 4.13: Before refactoring – Simple code example with multiple broadcasts

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 #include <math.h>
5 #include <time.h>
6
7 int main(int argc, char** argv){
8
9   int numtasks, rank, i;
```

```
10    int randomInt, msgsize=1;
11    int *randomInts = NULL;
12
13    MPI_Status stat;
14
15    MPI_Init(&argc, &argv);
16
17    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
18    randomInts = malloc(sizeof(int)*numtasks*msgsize);
19    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20
21    printf("numtasks: %d\n", numtasks);
22    printf("rank: %d\n", rank);
23
24    srand( time(NULL) + rank );
25    randomInt = rand() % 10;
26    randomInts[rank] = randomInt;
27    printf("rank %d: randomInt %d\n", rank, randomInt);
28
29    for(i = 0; i < numtasks; i++) {
30      MPI_Bcast(&randomInts[i], msgsize, MPI_INT, i,
          MPI_COMM_WORLD);
31    }
32
33    MPI_Finalize();
34  }
```

Refactor to `MPI_Allgather`

Listing 4.14 shows the result of the refactoring being applied to the code in Listing 4.13. The whole for loop that contained only the `MPI_Bcast` call has been exchanged for one `MPI_Allgather` call, sending all the random numbers from every processor to all other processors in one call. Line 29 shows how the `MPI_Allgather` call uses the array of random numbers rather than broadcasting every single random number with `MPI_Bcast` calls.

Listing 4.14: After refactoring – Simple code modified to use `MPI_Allgather`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include <math.h>
```

```
5  #include <time.h>
6
7  int main(int argc, char** argv){
8
9    int numtasks, rank, i;
10   int randomInt, msgsize=1;
11   int *randomInts = NULL;
12
13   MPI_Status stat;
14
15   MPI_Init(&argc, &argv);
16
17   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
18   randomInts = malloc(sizeof(int)*numtasks*msgsize);
19   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20
21   printf("numtasks: %d\n", numtasks);
22   printf("rank: %d\n", rank);
23
24   srand( time(NULL) + rank );
25   randomInt = rand() % 10;
26   randomInts[rank] = randomInt;
27   printf("rank %d: randomInt %d\n", rank, randomInt);
28
29   MPI_Allgather(MPI_IN_PLACE, msgsize, MPI_INT,
       randomInts, msgsize, MPI_INT, MPI_COMM_WORLD);
30
31   MPI_Finalize();
32 }
```

4.3.6 Summary

This chapter introduced a first version of a new HPC refactoring catalogue. The catalogue is not meant to be complete and more refactorings can be added. In the catalogue, the mechanics are written with manual application in mind, but they can also describe on a high level the source code transformation steps that a refactoring tool needs to apply. For a refactoring tool, the compile and test steps can be omitted assuming that the tool is implemented correctly. This thesis will continue by describing the implementation of the first refactoring (Synchronous communication to asynchronous communication) in Chapter 5.

5 Plug-in implementation

This chapter discusses the implementation of the refactoring plug-in and its unit testing. The implementation of the refactoring plug-in allows the automated application of the mechanics of a refactoring. As a proof of concept, one HPC refactoring was implemented within the plug-in, *Synchronous communication to asynchronous communication* (see Section 4.3.1), also referred to in this chapter as SyncToAsync. The implementation allows the user to semi-automatically apply the refactoring, i.e. the user needs to provide parameters in the beginning, but then the transformation is fully automated.

The license and availability for the source code are covered in Section 5.1. The refactoring chosen for implementation is Synchronous communication to asynchronous communication (see Section 4.3.1 for further description). The software dependencies of the implementation are discussed in Section 5.2. Section 5.3 covers the general approach of the implemented refactoring and the workflow of the refactoring plug-in is discussed in Section 5.4. The plug-in structure is described in Section 5.6 and Section 5.7 describes the main functionality class of the refactoring plug-in implementation. Section 5.8 discusses the unit testing implemented for the refactoring plug-in. Finally, the challenges faced in implementing the refactoring plug-in and unit testing are discussed in Section 5.9.

5.1 License and availability

The source code for the implementation described in this thesis is made available under the terms of the Eclipse Public License (EPL) v1.0 which is available at `http://www.eclipse.org/legal/epl-v10.html`. The source code for this implementation is long-term archived and can be found via `https://doi.org/10.5281/zenodo.3753576`.

5.2 Software dependencies

The plug-in developed as part of this thesis is intended to refactor code written in C/C++. Eclipse already provides some options for general refactoring of code written in C/C++ within the CDT plug-in, such as the *Rename* and *Extract constant* refactorings. With Eclipse CDT being open source (EPL) it is an obvious platform choice for implementing this refactoring plug-in with the intention of adding to the refactoring options within CDT.

The software dependencies of the implementation of this refactoring plug-in and unit testing are:

- Eclipse IDE for Eclipse Committers 2019-09 R (4.13.0)
- Java 1.8
- Eclipse LTK
- Eclipse CDT 9.9.0

This project was originally implemented using the PDE feature of Eclipse Neon. Later it was migrated to Eclipse IDE for Eclipse Committers which was released in September 2019.

5.3 General approach

Source code refactorings can either be done purely by doing modification on the source code level or by modifying the AST itself and replacing the source code with the string representation of the changed AST sub-tree. The implemented refactoring described in this chapter is based on existing CDT refactorings, which are in fact based on LTK [23] (see Section 2.3.2). Using CDT's method *getTranslationUnit*[1] an AST is retrieved. This AST is used to find needed AST nodes, e.g. finding the node that contains the full `MPI_Send` expression that is subject of refactoring, and finding references for variable names within the AST. It is also used for creating a CDT *rewriter*[2]. The *rewriter* collects descriptions of modifications to AST nodes and translates them into text edits. The *rewriter* does not actually modify the original AST but the text edits it provides can be applied, e.g. to provide a preview or to persistently apply the changes to the original source code [5].

[1] `org.eclipse.cdt.core.dom.ast.IASTNode.getTranslationUnit()`
[2] Of type `org.eclipse.cdt.core.dom.rewrite.ASTRewrite`

The transformations in the implementation are done by seeking out AST nodes and retrieving their string representations. These strings are transformed according to the refactoring (e.g. `MPI_Send` to `MPI_Isend`) and a new AST node is created. The newly created AST node is added to the *rewriter* along with information on where to place it in the AST (by specifying the AST node before which the new AST node will be inserted). The *rewriter* is also used to store information on which AST nodes to remove (e.g. removing `MPI_Send` after `MPI_Isend` is added). CDT's `CRefactoring`[3] class takes care of converting the collected modifications within the *rewriter* into the change object that the LTK refactoring framework requires [5].

5.4 Workflow

Figure 5.1 shows a Unified Modeling Language (UML) activity diagram for the plug-in workflow, starting from the user selecting the SyncToAsync refactoring to the refactoring being applied to the code. LTK and CDT dictate the high level workflow as initial conditions are checked as part of the LTK refactoring framework (see Section 2.3.2) and the changes are collected by CDT's *rewriter* (see Section 5.3) after checking the final conditions. The final conditions for this implementation consist of checking if the user input is an allowed input. The user may neither use a name that is actually a known C/C++ identifier nor use a name that is already used by a variable in the code. This is checked by the input page "on the fly" while the user is typing. If the final conditions are met, the changes are calculated and applied to the source by LTK. These changes can then be applied to a temporary AST and the resulting source code is shown to the user in a *Preview* (see Chapter 6 for a visual impression of the workflow) window without having to change the original AST. Therefore it becomes possible for the user to review the changes without having to apply them directly. If the user cancels, the changes are simply not applied to the original AST.

When the user selects the SyncToAsyncRefactoring refactoring the plug-in starts by checking the initial conditions (according to LTK refactoring framework) and the rest of the refactoring process continues from there. As the diagram in Figure 5.1 shows the user can cancel the refactoring at any point after the initial conditions are checked. If the user cancels no changes are made to the code and the refactoring process is cancelled.

Figure 5.2 shows the logic behind the *Calculate changes* (and *Calculate changes and show preview*) action in Figure 5.1 in more detail. The rewriter for collecting the modifications is initialized in the first step. The first decision checks if the name that the user provided for `MPI_Request` already exists – if not, a new AST node needs to

[3] In package `org.eclipse.cdt.internal.ui.refactoring.CRefactoring`

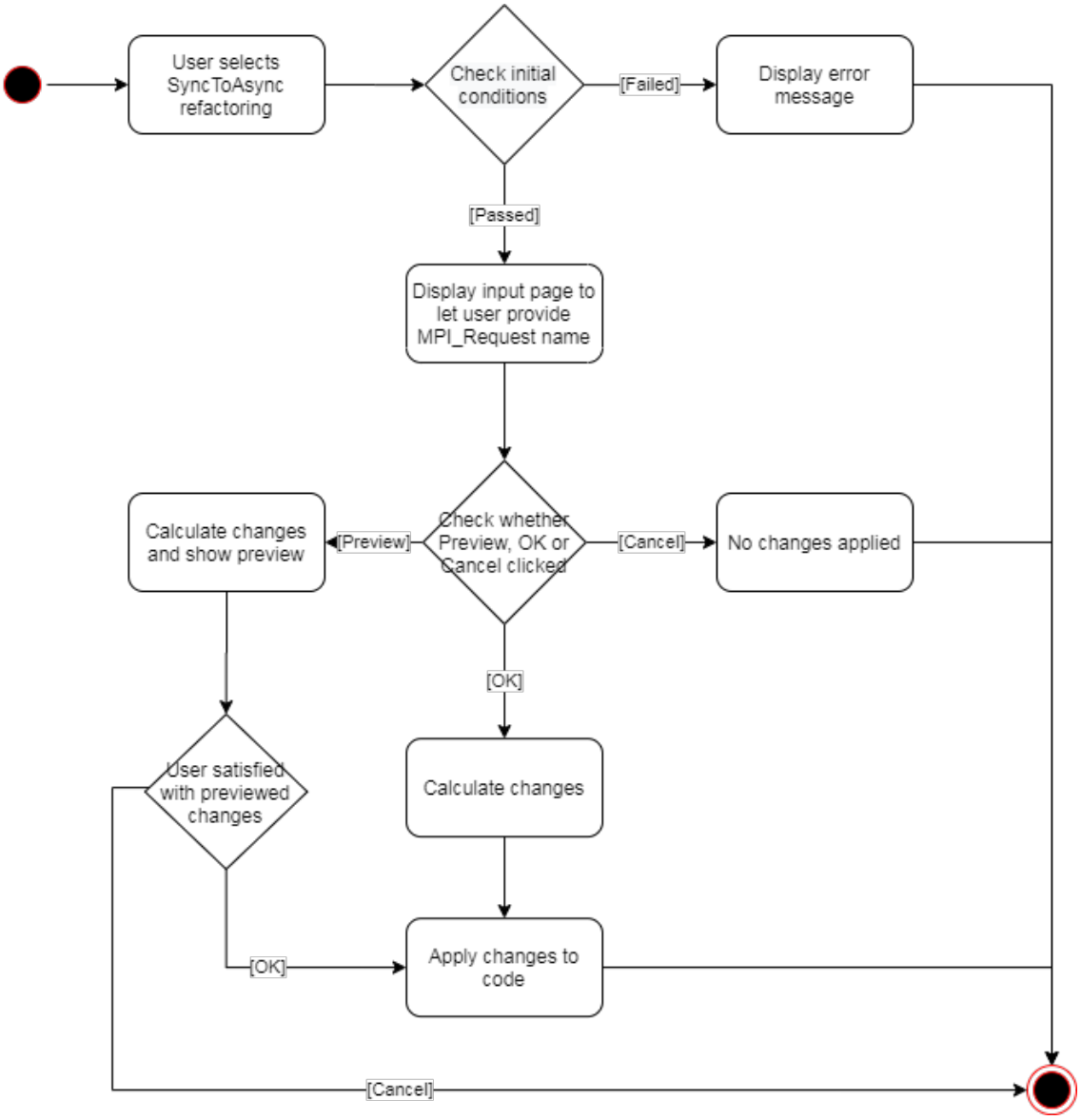

Figure 5.1: SyncToAsync refactoring UML activity diagram

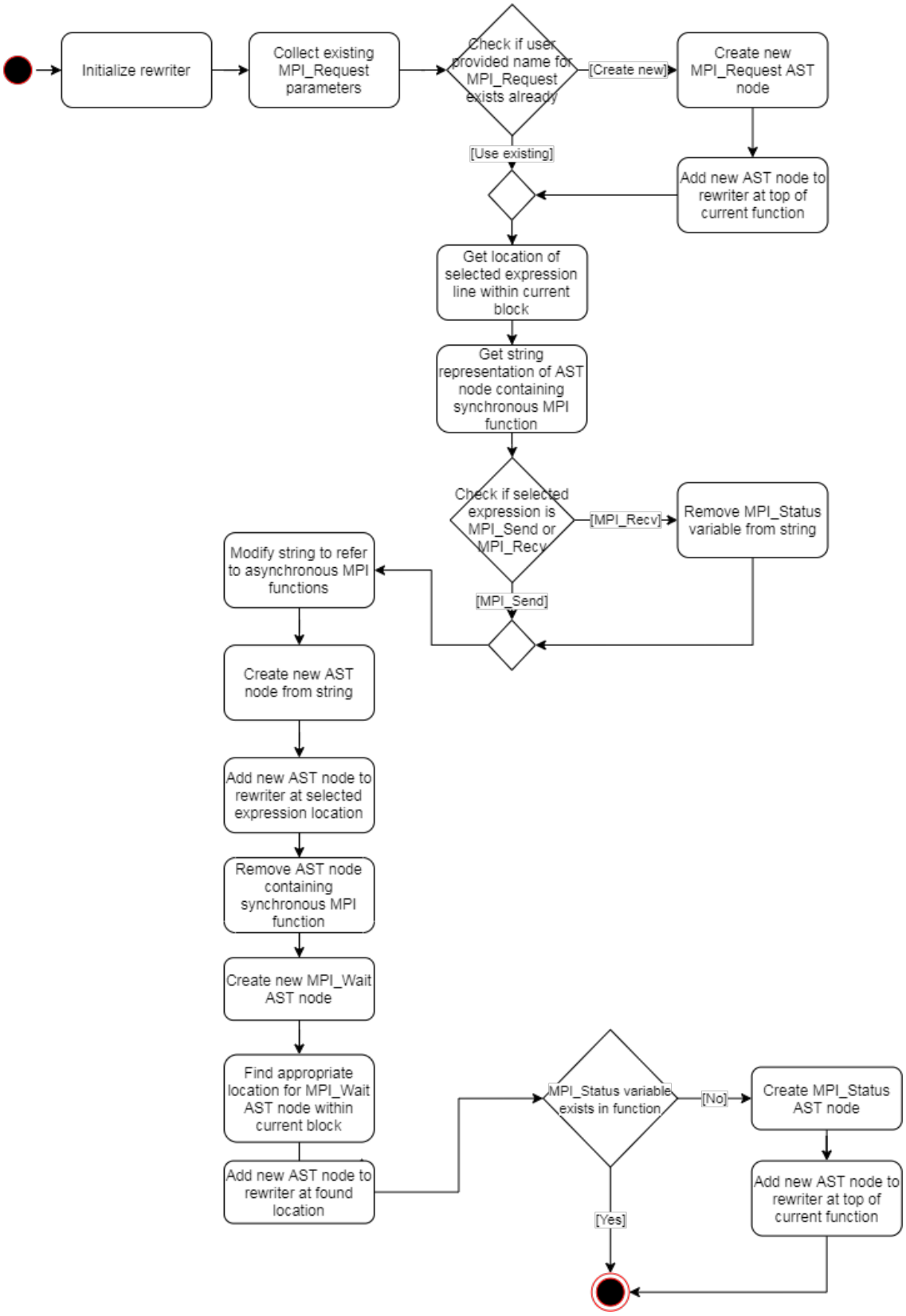

Figure 5.2: Calculate changes UML activity diagram

be added to the *rewriter* at the top of the current function. The next step searches for the location of the selected expression to use for later. The AST node containing the full `MPI_Send` or `MPI_Recv` function call is transformed to a string, the string is modified and a new AST node is created using the modified string. The new AST node is added to the rewriter at the previously found location and the AST node containing the synchronous MPI function is removed instead. A new AST node containing the `MPI_Wait` call is created and by using the AST to search for message buffer dependencies, a location for the new AST node is determined. This search does not differentiate if the message buffer is being read or modified, it searches for the first use after the initial use within the MPI function call. The new AST node is added to the *rewriter* at the found location. Finally, if an `MPI_Status` variable does not exist in the current function, a new AST node is created and added to the *rewriter* the same way the `MPI_Request` AST node is. This diagram is described in relation to the implemented method `gatherModifications` in Section 5.7.2.

5.5 Implementation structure

The implementation is separated into two Eclipse plug-in projects as can be seen in Figure 5.3. The first project contains the refactoring plug-in, both what is related to the User Interface (UI) and the methods behind the refactoring calculations and changes. This plug-in is described in sections 5.6 and 5.7. The second project contains unit testing done for the refactoring plug-in. It is kept in a separate project to allow shipping of the refactoring plug-in to end users without the test code dependencies. The unit testing is further described in Section 5.8.

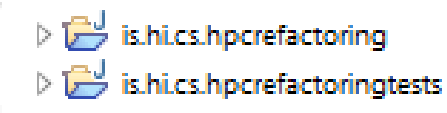

Figure 5.3: The two Eclipse projects

5.6 Refactoring plug-in structure

The refactoring implementation is an Eclipse plug-in that depends on CDT. As said it currently contains one refactoring which is included in the following Java package:

- `is.hi.cs.hpcrefactoring.synctoasync` – This package contains the implementation of the *Synchronous to asynchronous communication* refactoring.

Figure 5.4 shows the structure of the refactoring plug-in.

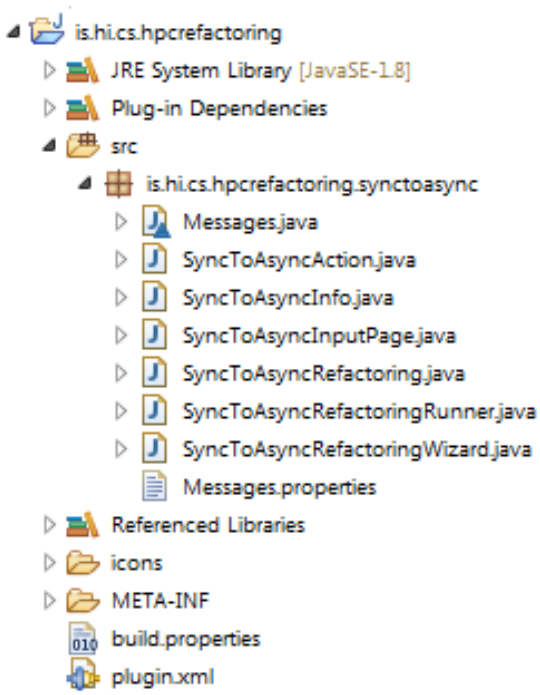

Figure 5.4: The plug-in structure

As can be seen in Figure 5.4 the `is.hi.cs.hpcrefactoring.syntoasync` package contains several Java files along with one property file that make up the plug-in:

- `Messages.java`: Extends the `org.eclipse.osgi.util.NLS` class that supports translating text shown in the user interface (UI language localisation). It initializes the messages that the plug-in can show the user in the interface.

- `SyncToAsyncAction.java`: Is the class that launches a SyncToAsync refactoring, activated by the Eclipse UI framework when the *Sync to Async* menu entry is selected by the user. It extends the `org.eclipse.cdt.ui.refactoring.actions.RefactoringAction` class.

- `SyncToAsyncInfo.java`: contains variables (filled by the user via the input page) and methods used for configuring the refactoring, i.e. provide the needed parameters.

- `SyncToAsyncInputPage.java`: Extends the `org.eclipse.ltk.ui.refactoring.UserInputWizardPage` class and creates the input page. It also contains methods to verify the user input as the user types, to make sure that the user does not use names that are actually C/C++ identifiers or names that are

already used by other variables in the code.

- `SyncToAsyncRefactoring.java`: Extends the `org.eclipse.cdt.internal.ui.refactoring.CRefactoring` class and contains all the logic behind the refactoring itself, i.e. the actual AST and source code transformation. This class will be further described in Section 5.7 along with UML activity diagrams for the functionality. Listings 5.1 and 5.2 (see Section 5.7.2) show source code of two methods in this class.

- `SyncToAsyncRefactoringRunner.java`: Extends the `org.eclipse.cdt.internal.ui.refactoring.RefactoringRunner` class and initializes the `SyncToAsyncRefactoring` and `SyncToAsyncRefactoringWizard` classes and runs the refactoring.

- `SyncToAsyncRefactoringWizard.java`: Extends the `org.eclipse.ltk.ui.refactoring.RefactoringWizard` class and contains the wizard page for the SyncToAsync refactoring and creates the UI pages.

Figure 5.4 also shows the *plugin.xml* file, which is the plug-in manifest file [17]. Information on where to display the new plug-in in the UI is contained in the manifest file. The SyncToAsync refactoring menu entry is added to the UI of the Eclipse IDE at one location, within the CDT refactoring menu, as shown in Figure 5.5. When the *Refactor* main menu item is selected a drop-down menu appears and at the bottom there is the *HPC* option. When the user hovers over the *HPC* option there is a sub-menu which is intended to contain all refactorings for HPC. Currently, there is one entry in this menu, *Sync to Async*.

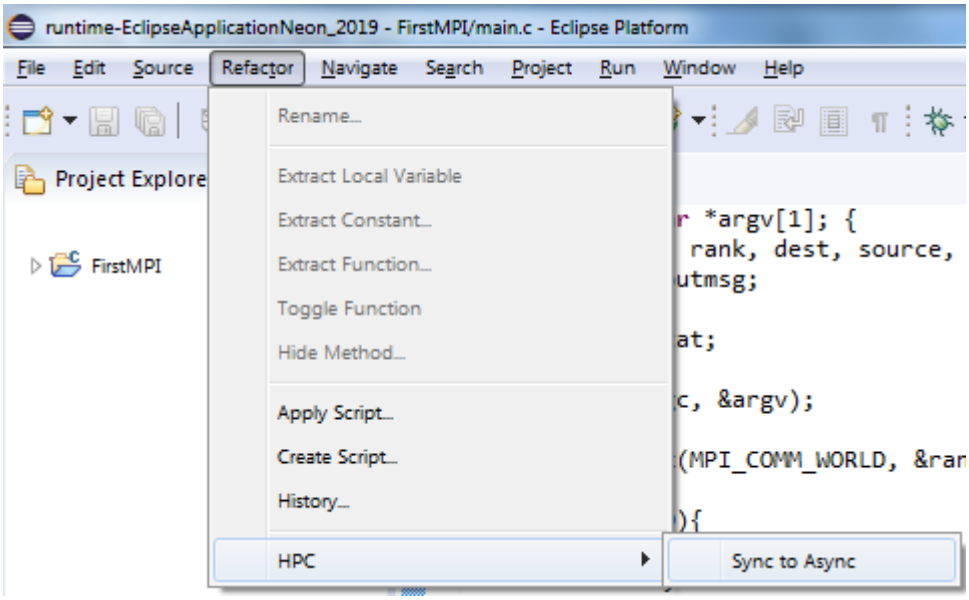

Figure 5.5: HPC refactoring menu entry displayed in the UI

5.7 SyncToAsyncRefactoring class

The `SyncToAsyncRefactoring` class contains most of the functionality that is not directly related to the UI but to the AST and source code transformations that the refactoring makes. It contains two larger public methods along with several helper methods in addition to inheriting methods from the CDT `CRefactoring` class. The larger public methods will be further described in the next two sections.

5.7.1 checkInitialConditions

The `checkInitialConditions` method shown in Listing 5.1 mainly makes sure that the expression selected by the user is an allowed one. The UI flow starts with a user selecting source code (source code must be highlighted, placing the cursor only is not sufficient) and selecting the *Sync to Async* refactoring menu entry, shown in Figure 5.5. Before the input page is shown, this action starts a check to see if the user-selected location is appropriate for the application of the refactoring. The user should select an expression that is syntactically a C/C++ function call and if not an error message is returned, as shown in Figure 5.6. In addition, the selected

function name needs to be either `MPI_Send` or `MPI_Recv` for the refactoring to work. If another function is chosen an error message stating so is returned, as shown in Figure 5.7.

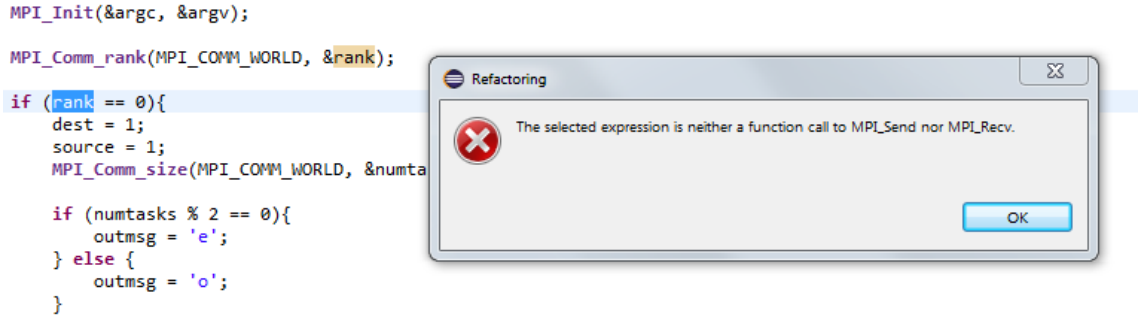

Figure 5.6: Error message when a non-function is selected

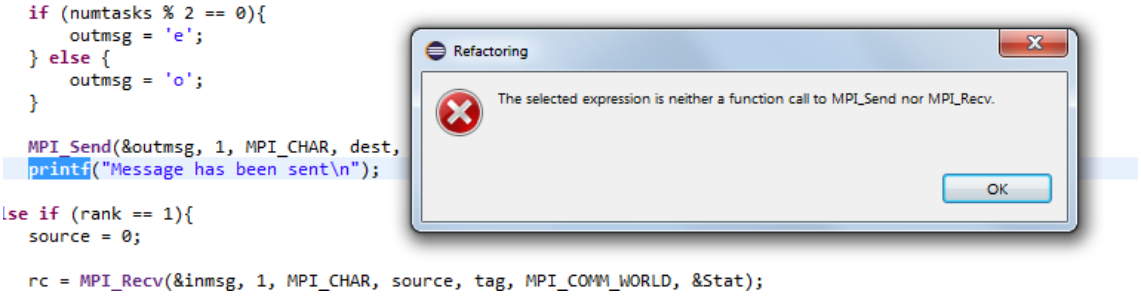

Figure 5.7: Error message when the function type selected is not allowed

As can be seen in Listing 5.1 the method returns an object of type `RefactoringStatus`[4] to indicate if the conditions have been met and that the refactoring should be allowed to continue. Line 14 finds the selected expression with the help of an internal method and line 15 returns the string format of said expression. The variables `parent` and `propertyInParent` contain the string format of the parent node of the selected expression and string format of the selected expression's property in the parent node, respectively. These strings are used for lookup in lines 24–33. Lines 24–27 contain the first check, whether the selected expression is a function, using values found for functions in an AST. If this check fails the status is returned with an error message which is displayed in an error dialog as seen in Figure 5.6 and the process

[4] `org.eclipse.ltk.core.refactoring.RefactoringStatus`

is terminated. The second check is done in lines 30–33 where the string format of the selected expression is checked to be equal to either `MPI_Send` or `MPI_Recv` and if not the process is terminated after displaying the returned error message in an error dialog shown in Figure 5.7. Line 35 sets the global variable *target* as the selected expression so that it can be used throughout the class. Lines 37–40 set a suggestion of a name for the `MPI_Request` parameter, a default name *request* if such a parameter does not exist already, and the first result of a lookup (of `MPI_Request` parameters within the current function) if one or more exist. To be able to use the name of an existing `MPI_Request` parameter, lines 42–52 look for a given name within all possible `MPI_Request` names and marks them as "not used" so they are not blocked from being used again.

Listing 5.1: `checkInitialConditions` method

```
1  public RefactoringStatus
       checkInitialConditions(IProgressMonitor monitor)
       throws CoreException, OperationCanceledException {
2    SubMonitor subMonitor = SubMonitor.convert(monitor,
       12);
3
4    RefactoringStatus status =
       super.checkInitialConditions(subMonitor.split(8));
5    if (status.hasError()){
6      return status;
7    }
8
9    if (selectedRegion == null){
10     status.addFatalError("No selected region");
11     return status;
12   }
13
14   IASTExpression selectedExpression =
       findSelectedExpression(subMonitor.split(1), tu);
15   String expressionString =
       ASTStringUtil.getExpressionString(selectedExpression);
16
17   String parent = "";
18   String propertyInParent = "";
19   if(!expressionString.isEmpty()) {
20     parent = selectedExpression.getParent().toString();
21     propertyInParent =
         selectedExpression.getPropertyInParent().toString();
22   }
23
```

```
24    if (!parent.contains("FunctionCallExpression") ||
          !propertyInParent.contains(
25          "IASTFunctionCallExpression.FUNCTION_NAME")) {
26      status.addFatalError(Messages.NonMPISendRecvSelected);
27      return status;
28    }
29
30    if (!expressionString.equals("MPI_Send") &&
          !expressionString.equals("MPI_Recv")){
31      status.addFatalError(Messages.NonMPISendRecvSelected);
32      return status;
33    }
34
35    target = selectedExpression;
36
37    ArrayList<String> MPIRequestStrings =
          getMPIVariableNames("MPI_Request", true);
38    if (info.getName().isEmpty()) {
39      info.setName(MPIRequestStrings.get(0));
40    }
41
42    info.setMethodContext(NodeHelper.findMethodContext(target,
          refactoringContext, subMonitor.split(1)));
43    subMonitor.split(1);
44    IScope containingScope =
          CVisitor.getContainingScope(target);
45    IASTTranslationUnit ast = target.getTranslationUnit();
46    info.setNameUsedChecker((String name) -> {
47      if (MPIRequestStrings.contains(name))
48        return false;
49
50      IBinding[] bindingsForName =
            containingScope.find(name, ast);
51      return bindingsForName.length != 0;
52    });
53    return status;
54  }
```

5.7.2 gatherModifications

The `gatherModifications` method shown in Listing 5.2 initializes a new *rewriter* and retrieves the AST for the code of the expression selected by the user in the opened Eclipse editor. It then creates each modification of the AST and adds it to the *rewriter* variable that collects all AST modifications.

As with all LTK-based refactorings the modifications will either be applied immediately to the source code if the user selects the *OK* button right away or is displayed first in a *Preview* window for the user to review the changes and confirm or cancel (this functionality is provided by LTK itself (see Section 2.3.2)).

The `gatherModification` method shown in Listing 5.2 is corresponding to the UML activity diagram shown in Figure 5.2. All changes are added to the *rewriter* (described in Section 5.3) which is initialized in line 6 of the `gatherModification` method and the first step of the diagram. This *rewriter* is built from an input parameter of type `ModificationCollector`[5] and the AST itself and as said before keeps track of all changes without actually applying them to the AST.

In lines 9–13 in the code, which correspond to the next part of the diagram, the user has a choice between using an existing `MPI_Request` parameter or creating a new one. If an `MPI_Request` parameter exists it is put as a placeholder in the text field at the input page, giving the user the option to use an existing one. Line 18 in the code corresponds to the part of the diagram from the *Get string representation of AST node containing synchronous MPI function* action to the *Create new AST node from string*. It has a slightly different functionality depending on if the refactoring is dealing with `MPI_Send` or `MPI_Recv`. This comes from the fact that `MPI_Recv` has one additional parameter to `MPI_Send`, namely `MPI_Status`, that needs to be removed before other changes from the synchronous communications to asynchronous ones are applied. Lines 19 and 20 correspond to the next two actions in the diagram, first inserting the newly created declaration at location before the original one and then removing the original one so the new one takes its place. Lines 23–25 in the `gatherModifications` method correspond to actions *Create new MPI_ Wait AST node* to *Add new AST node to rewriter at found location* respectively. The `MPI_Wait` expression that needs to be created has two input variables of types `MPI_Request` (the same one as is used in the asynchronous function) and `MPI_Status`. If a variable of type `MPI_Status` is found within the current function it is used as an input. If not a new name is used and a global variable set indicating that a new variable of type `MPI_Status` has to be declared. How the location for the new `MPI_Wait` is found is described in Section 5.9.2 (note that the implementation does not differentiate between read or write access to the message buffer, it searches for the first use after the initial use within the MPI function call). Line 27 in the code corresponds to the

[5] `org.eclipse.cdt.internal.ui.refactoring.ModificationCollector`

last decision in the diagram. In this line a global variable tells the program if an `MPI_Status` variable should be added or not having checked in the previous step if such a variable exists in the code. If it exists a new one is not added else it is added at the top of the function, along with other declarations.

Listing 5.2: `gatherModifications` method

```
1  public void gatherModifications(IProgressMonitor pm,
       ModificationCollector collector) throws CoreException
       {
2    IASTTranslationUnit ast = target.getTranslationUnit();
3    String constName = info.getName();
4
5    IASTDeclaration nodes;
6    ASTRewrite rewriter =
       collector.rewriterForTranslationUnit(ast);
7    TextEditGroup editGroup = new
       TextEditGroup("SyncToAsync");
8
9    ArrayList<String> MPIRequestNames =
       getMPIVariableNames("MPI_Request", false);
10   if (!MPIRequestNames.contains(constName)) {
11     nodes = createMPIVariableDeclaration("MPI_Request",
         constName);
12     rewriter.insertBefore(findCurrentFunctionNode(target),
         findCurrentFunctionNode(target).getChildren()[0],
         nodes, editGroup);
13   }
14
15   IASTNode correctLocationNode = getCorrectParentNode();
16
17   //Change send/recv to isend/irecv
18   nodes = syncToAsync(constName);
19   rewriter.insertBefore(correctLocationNode.getParent(),
       correctLocationNode, nodes, editGroup);
20   rewriter.remove(correctLocationNode, editGroup);
21
22   //Add MPI_Wait
23   nodes = createWaitString(constName, ast);
24   IASTNode waitLocation =
       findWaitLocationNode(correctLocationNode, ast);
25   rewriter.insertBefore(correctLocationNode.getParent(),
       waitLocation, nodes, editGroup);
26
```

```
27    if (addStatusVariable != null) {
28      nodes = createMPIVariableDeclaration("MPI_Status",
           addStatusVariable);
29      rewriter.insertBefore(findCurrentFunctionNode(target),
           findCurrentFunctionNode(target).getChildren()[0],
           nodes, editGroup);
30    }
31  }
```

Listing 5.3 shows the method referenced in Line 11 from the `gatherModifications` method described above. It shows an example of how a CDT node factory is used to create a new AST node for a declaration (lines 9–11) to later add to the *rewriter*. With this method a new `MPI_Request` or `MPI_Status` variable is created.

Listing 5.3: `createMPIVariableDeclaration` method

```
1  private IASTSimpleDeclaration
      createMPIVariableDeclaration(String variableType,
      String newName) {
2    //Declares a new MPI parameter to be used in the
        asynchronous call
3    String MPIVariable = variableType + " " + newName;
4
5    ICPPNodeFactory factory =
        ASTNodeFactoryFactory.getDefaultCPPNodeFactory();
6    IASTName RequestName =
        factory.newName(MPIVariable.toCharArray());
7    IASTDeclarator declarator =
        factory.newDeclarator(RequestName);
8    IASTSimpleDeclSpecifier declSpec =
        factory.newSimpleDeclSpecifier();
9    IASTSimpleDeclaration simple =
        factory.newSimpleDeclaration(declSpec);
10
11   simple.addDeclarator(declarator);
12   return simple;
13   }
```

5.8 Unit testing plug-in

This section discusses the unit tests created to test the implementation of the refactoring plug-in. All testing is done using a special PDE JUnit 4 Plug-in test runner. A separate plug-in testing project is used for the unit tests to avoid the refactoring plug-in having a dependency on JUnit to minimize dependencies.

5.8.1 Structure

Figure 5.8 shows the structure of the test project containing the unit tests for the refactoring. It consists of Java files containing the code of the unit tests that call the refactoring implementation under test and test data files, i.e. C files used as input of the refactoring under test and as expected results to compare the actual result with. There are two separate *before* files, one contains an existing `MPI_Request` variable (file `beforeWithRequest.c`) and the other does not (file `beforeWithoutRequest.c`), i.e. it is a test case where a new `MPI_Request` variable has to be created. Each of these two *before* files tested has three variants of further input parameters of the refactoring under test. Altogether, the project includes six different tests cases which are as follows:

- Test A: Refactor `MPI_Send` to `MPI_Isend` using an existing `MPI_Request` variable.
- Test B: Refactor `MPI_Send` to `MPI_Isend` creating a new `MPI_Request` variable when another one exists.
- Test C: Refactor `MPI_Send` to `MPI_Isend` creating a new `MPI_Request` variable when none exists.
- Test D: Refactor `MPI_Recv` to `MPI_Irecv` using an existing `MPI_Request` variable.
- Test E: Refactor `MPI_Recv` to `MPI_Irecv` creating a new `MPI_Request` variable when another one exists.
- Test F: Refactor `MPI_Recv` to `MPI_Irecv` creating a new `MPI_Request` variable when none exists.

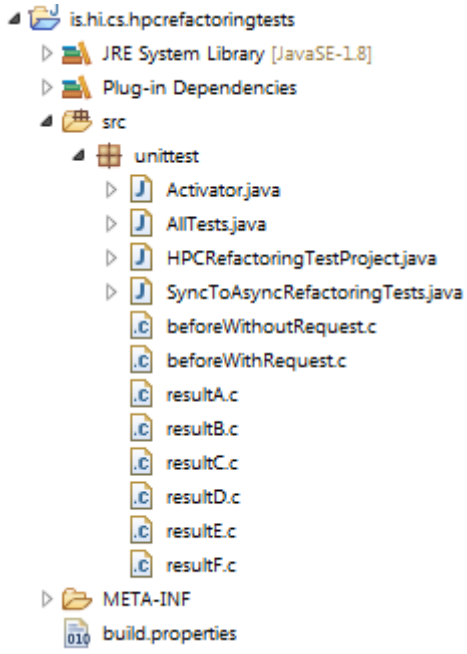

Figure 5.8: Test project structure

In addition to modifying the synchronous calls to be asynchronous in the correct way, for tests A and D a correct implementation needs to take care not to add an additional `MPI_Request` variable while tests B, C, E and F need to do so. Each test uses a *result* file marked with a corresponding letter to compare with the result of the refactoring execution using the corresponding *before* file as input, e.g. *Test A* uses file *resultA.c* for comparison.

`AllTests.java` calls a parameterized test method with the six different test input files and further parameters specific to each test case. The parameters for each test case are as follows:

- *filePath* is the path for the appropriate *before* file, to open in the editor.
- *projectName* is a name for the test case, e.g. *testProjectA*.
- *MPIType* is either `MPI_Send` or `MPI_Recv`, depending on the test case.
- *requestString* is a name for the `MPI_Request` parameter that may or may not be newly created, depending on the test case.

- *beforeFilePath* is an absolute path to the *before* file within the test project workspace (only available at runtime).
- *afterFilePath* is an absolute path to the *result* file within the test project workspace (only available at runtime).
- *selectedNodeOffset* is the offset for `MPI_Send` or `MPI_Recv` within the *before* file open in the editor, used to set the selected expression manually.
- *errorMessage* is a message to display if the test case fails.

5.8.2 testSyncToAsyncRefactoring

The unit testing is done only for the `gatherModifications` (Listing 5.2) method since that is the method performing all the transformations that are actually applied to the code. The method `testSyncToAsyncRefactoring` contains the actual unit test cases. It is called for each of the six unit tests with different parameters.

The parameterized unit test case is shown in Listing 5.4 using the parameters described in previous section. The unit test first creates a new test project (line 6) that will contain the file with C code to be refactored. When a new `HPCRefactoringTestProject` is created a new Eclipse workspace is created and filled with the files from the directory `is.hi.cs.hpcrefactoringtests/src/unittest` that can be seen in Figure 5.8. Lines 7–13 use the newly created project to open a *before* file in an editor in the IDE. At that point the opened file can be used to create a translation unit as can be seen in line 14. The translation unit is one of the global parameters that need to exist when applying the `gatherModifications` method. Line 18 creates a new instance of the `SyncToAsyncRefactoring` class and lines 19–28 set necessary parameters for the refactoring such as the `MPI_Request` parameter name. The AST for the test file is created[6] in line 21 and is used to set the selected node from an offset input in line 32. When this preparation is done the `gatherModification` method is called at line 36 and the next two lines apply the gathered modifications to the file in the editor. The *result* file is now ready and is compared to the *expected* file. The implementation passes the test if the files are equal.

Listing 5.4: `testSyncToAsyncRefactoring` function

```
1 public void testSyncToAsyncRefactoring(String filePath,
    String projectName, String MPIType, String
```

[6]The function `getAST` returns using the Singleton design pattern a shared AST when it exists. If a shared one does not exist a new one is created.

```
    requestString, String beforeFilePath, String
    afterFilePath, int selectedNodeOffset, String
    errorMessage) throws Exception {
2
3   //Initialize testproject
4   testProject = new HPCRefactoringTestProject(
5       "/HPCRefactoringTests/src/unittest/", projectName);
6   IPath path = new Path(filePath);
7   IFile file = testProject.project.getFile(path);
8   IResource resource =
      testProject.project.findMember(path);
9   iproject = resource.getProject();
10  ICProject icProject =
      CoreModel.getDefault().create(iproject);
11  IWorkbenchPage page = PlatformUI.getWorkbench()
12      .getActiveWorkbenchWindow().getActivePage();
13  IEditorPart edPart = IDE.openEditor(page, file);
14  ITranslationUnit tu = (ITranslationUnit)
      CDTUITools.getEditorInputCElement(edPart.getEditorInput());
15  ICElement element =
      CoreModel.getDefault().create(path);
16
17  //Initialize the refactoring and set necessary
      parameters
18  SyncToAsyncRefactoring refactoring = new
      SyncToAsyncRefactoring(element, null, icProject);
19  CRefactoringContext refactoringContext = new
      CRefactoringContext(refactoring);
20  refactoring.setContext(refactoringContext);
21  IASTTranslationUnit testast = refactoring.getAST(tu);
22  refactoring.setInfoName(requestString);
23  CheckConditionsContext context= new
      CheckConditionsContext();
24  context.add(new
      ValidateEditChecker(refactoring.getValidationContext()));
25  ResourceChangeChecker resourceChecker = new
      ResourceChangeChecker();
26  IResourceChangeDescriptionFactory deltaFactory =
      resourceChecker.getDeltaFactory();
27  context.add(resourceChecker);
28  ModificationCollector modificationCollector = new
      ModificationCollector(deltaFactory);
29
30  //Find "selected" node from offset and set as target
```

```
31    IASTNodeSelector nodeSelector =
          testast.getNodeSelector(null);
32    IASTNode selectedNode =
          nodeSelector.findNode(selectedNodeOffset, 8);
33    refactoring.target = (IASTExpression)
          selectedNode.getParent();
34
35    //Collect and apply the changes
36    refactoring.gatherModifications(new
          NullProgressMonitor(), modificationCollector);
37    CCompositeChange change =
          modificationCollector.createFinalChange();
38    change.perform(new NullProgressMonitor());
39
40    //Get the before/after files from comparison
41    String expected = new String
          (Files.readAllBytes(Paths.get(afterFilePath)));
42    String result = new String
          (Files.readAllBytes(Paths.get(beforeFilePath)));
43
44    //Compare result to expected file
45    assertEquals(errorMessage, result, expected);
46  }
```

5.9 Implementation challenges

This section discusses the challenges that occurred during the implementation of a refactoring plug-in within the Eclipse environment along with a solution description where applicable.

5.9.1 Documentation

A common challenge that developers often face is lack of documentation and this project is no exception in that matter. Eclipse has a lot of general documentation but often lacks examples and explanations. Same goes with CDT where there is sometimes need to dig through a lot of source code for a better understanding when a Google search has not been helpful. This often makes the implementation very time consuming when a lot of different possible solutions need to be systematically

tried instead of documentation (with or without examples) leading the way.

An example of lack of documentation is the placement of the new refactoring menu entry within the UI of the IDE. The aim was to place the new refactoring in the same main menu as for the existing refactorings. Without proper documentation the only question/answer addressing this problem found with a Google search was around a decade old and things had in fact changed since then. In the end this problem was solved with trial and error, taking much longer than it could have.

5.9.2 Finding the correct place for MPI_Wait

The refactoring tries to push the `MPI_Wait` function as far away as possible from its matching asynchronous MPI communication call, although it will always stay within the same block. It has to be placed before the message buffer in the asynchronous call is used again. The solution here is using the `find`[7] and `getReferences`[8] methods from CDT to search for the message buffer parameter from the asynchronous call until the end of the block to see if it is used. The `MPI_Wait` is placed either the line before the message buffer is used again or at the end of the block, if it is not used again within the block.

Note that the implementation does not differentiate between if the message buffer is being accessed with read or write for different placements depending on `MPI_Send` or `MPI_Recv` (as described in the mechanics in Section 4.3.1), it only checks if the message buffer is being used at all, whether it is read or write.

5.9.3 Finding all instances of a type within a function

To be able to offer the user the possibility of using an existing `MPI_Request` variable the refactoring plug-in needs to be able to find instances of that type. The method described in Section 5.9.2 is not guaranteed to work since technically it searches for names and not types (although this works in some cases). As a fallback each location in the AST is checked for this parameter and then cross-referenced if found to see if it is located within the current function.

[7] `org.eclipse.cdt.core.dom.ast.IScope.find`
[8] `org.eclipse.cdt.core.dom.ast.IASTTranslationUnit.getReferences`

5.9.4 AST and unit testing

For unit testing the `gatherModification` method (shown in Listing 5.2 of Section 5.7.2) the AST has to be created somehow, as a translation unit does not "automatically" exist as when Eclipse is opened normally. The solution, described in Section 5.8.2, is to enforce CDT to build the AST by opening the desired C/C++ file in an editor in Eclipse and working from there.

5.9.5 AST and indentation

For the refactoring to indent newly inserted lines properly, the indentation the user is using in their code must match the configured indentation within Eclipse. An informal testing of refactorings in CDT showed this indentation issue as well. With the implementation relying on the CDT approach of using the AST *rewriter* this was not investigated further, but rather the indentation was corrected within the editor. A quick fix to correct the indentation for the code in the editor, if it does not match the configured indentation, is by a right click somewhere in the editor and finding *Source→Correct Indentation*, as shown in Figure 6.1. The same action can also be invoked by pressing *CTRL+A*, followed by *CTRL+I*. If the user wants to configure the preferred indentation within Eclipse rather than changing the source code in the editor, it can be done via the *Code Style preference* panel [12].

5.10 Outlook on further implementation

The refactoring that was implemented as a proof-of-concept and is described in this chapter can be used as a foundation for implementing further refactoring such as the other four refactorings described in the catalogue in Chapter 4. Most changes would have to be made to `SyncToAsyncRefactoring.java` as it contains the logic behind the refactoring itself, i.e. it needs to reflect the respective refactoring mechanics.

Three refactorings in the catalogue aim to replace multiple `MPI_Send` and `MPI_Recv` calls with one collective communication operation instead, as described in sections 4.3.2, 4.3.3, and 4.3.4. The functional structure underlying these three refactorings should be fairly similar so once one of these three is implemented the implementation of the other two should need even less effort.

While the UI of the implemented *Synchronous communication to asynchronous communication* refactoring is based on the user selecting either a single `MPI_Send` or `MPI_Recv` to be refactored, as described in Section 4.3.1, the three refactorings in-

volving multiple `MPI_Send` and `MPI_Recv` would involve a UI complication: either the user would have to select the correct `MPI_Send` and `MPI_Recv` pair (however the Eclipse UI does currently not support multiple selections) or if the user selects either only `MPI_Send` or `MPI_Recv` call the refactoring itself has to locate the other call.

The fifth refactoring, described in Section 4.3.5, replaces multiple `MPI_Bcast` calls with a `MPI_Allgather` call. In a similar manner as the three aforementioned refactoring, it aims to replace multiple calls with one. In that way the functionality structure is similar to the other three above refactorings.

The implemented refactoring does not differentiate between read or write access to the message buffer when looking for the correct placement for `MPI_Wait` (see description on `MPI_Wait` placement in the mechanics in Section 4.3.1). It would be a good opportunity for improvement for the existing implementation to take this into account.

6 Case study

In this chapter the implemented *Synchronous communication to asynchronous communication* refactoring is applied to a code example to confirm that the functionality is indeed correct according to the refactoring description in Section 4.3.1, in particular with respect to the mechanics. Section 6.1 describes the chosen code example before any refactoring has been applied. In Section 6.2 the application of the refactoring is shown visually step by step and Section 6.3 shows the final result after all the changes have been applied to the code by the implemented refactoring tool.

6.1 Before

This section describes the original code example. Listing 6.1 shows the code before any refactoring is done. It is a simple example, written in C, of message passing between two processors. Lines 15 and 28 check which one of two processors is currently running, the processor with rank 0 being the source or sender and the other processor with rank 1 being the receiver. Line 25 contains the send communication call for the sender processor and line 31 the receive communication call for the receiving processor. Note that both send and receive actually return an integer but the functions can be called with or without using this return integer. Line 25 does not include this return integer but line 31 does: Using the return parameter obviously adds some elements to the AST (a simple function call AST node versus an assignment node with a function call on the right hand side) and therefore for testing purposes these two lines were kept different so both cases appear.

Note that for the refactoring to indent new lines correctly, the configured indentation for the Eclipse editor must match the indentation of the code (see description in Section 5.9.5 and Figure 6.1).

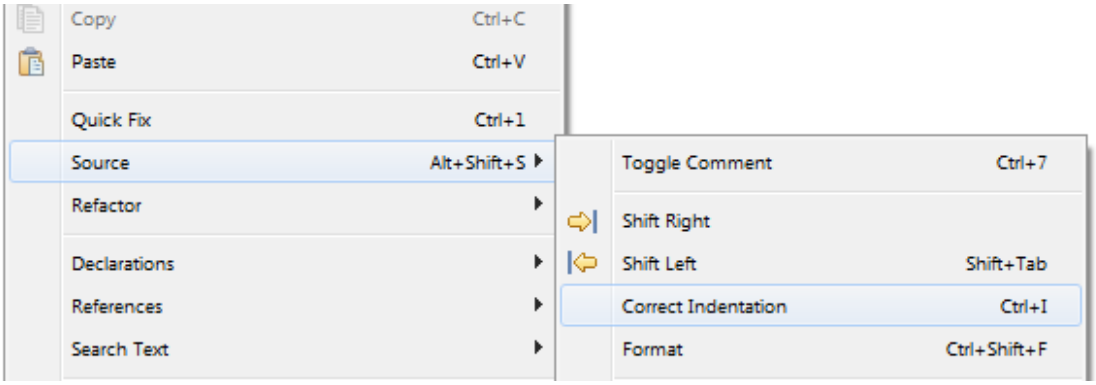

Figure 6.1: Correcting the indentation in an Eclipse editor

Listing 6.1: Before – Simple C code example

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(argc, argv)
5    int argc; char *argv[1]; {
6    int numtasks, rank, dest, source, rc, tag=1;
7    char inmsg, outmsg;
8
9    MPI_Status Stat;
10
11   MPI_Init(&argc, &argv);
12
13   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14
15   if (rank == 0){
16     dest = 1;
17     source = 1;
18     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
19
20     if (numtasks % 2 == 0){
21       outmsg = 'e';
22     } else {
23       outmsg = 'o';
24     }
25     MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
```

```
        MPI_COMM_WORLD);
26      printf("Message has been sent\n");
27    }
28    else if (rank == 1){
29      dest = 0;
30      source = 0;
31      rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
           MPI_COMM_WORLD, &Stat);
32      printf("Message has been received\n");
33      if (inmsg == 'e'){
34        printf("The number of processors is even");
35      } else {
36        printf("The number of processors is odd");
37      }
38    }
39
40    MPI_Finalize();
41  }
```

6.2 Applying the refactoring

This section covers the steps of the refactoring application to the code example described in Section 6.1. Figures 6.2–6.8 show the different steps of applying the refactoring twice: first to transform the synchronous send into an asynchronous send and then to transform the synchronous receive to an asynchronous receive.

Figure 6.2 shows where the plug-in adds an entry in the CDT refactoring menu to allow the user to select the *Sync to Async* refactoring. In this first case the user has the expression `MPI_Send` highlighted in the editor window before selecting the aforementioned refactoring.

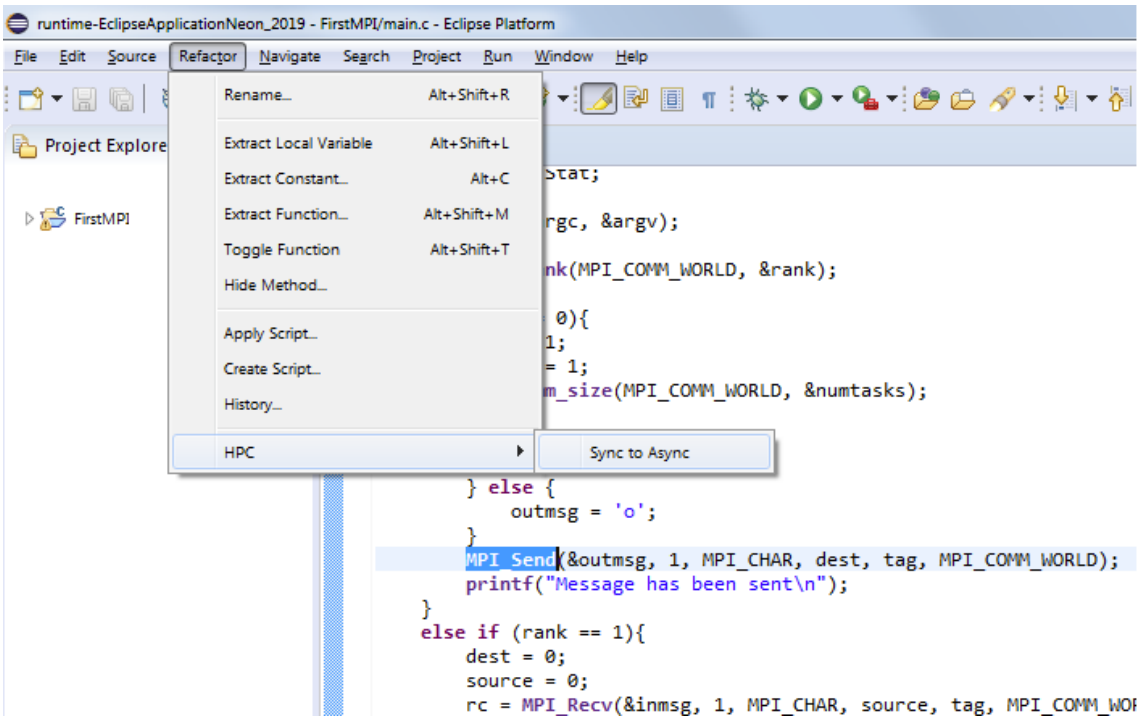

Figure 6.2: Selecting the SyncToAsync refactoring option

After the *Sync to Async* refactoring entry from Figure 6.2 has been selected the window in Figure 6.3 appears. It allows the user to choose a name for the `MPI_Request` parameter which can be a name of an `MPI_Request` parameter that already exists within the same function as the chosen `MPI_Send`/`MPI_Recv` or an entirely new name. The default value is the existing parameter name if one exists, otherwise a default name, *request*, is provided. In this case study the user chooses to create a new name for the `MPI_Request` parameter, `newRequest`.

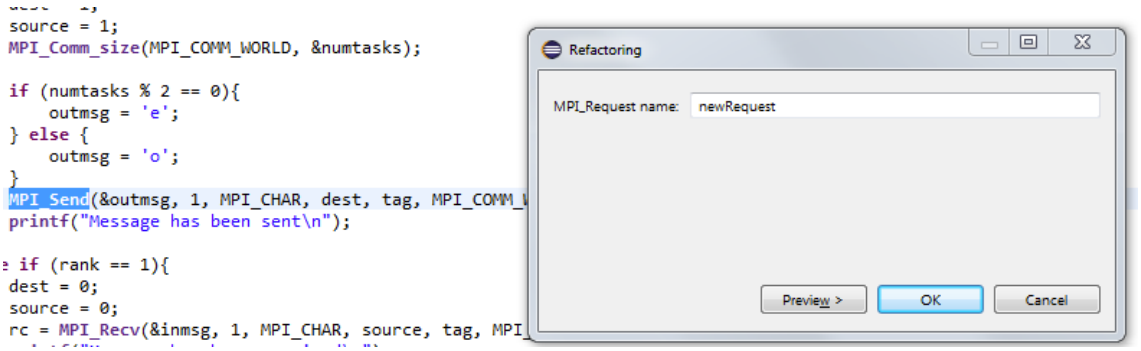

Figure 6.3: Refactoring user input window for MPI_Send

By clicking the *Preview* button the user is presented with the *Preview* window shown in Figure 6.4. This window allows the user to look at the changes without actually applying them to the original code. Here it can be seen that a new declaration of a parameter of type `MPI_Request` is added to the top, using the name input from Figure 6.3. It also shows the changes made to the `MPI_Send` line and where the `MPI_Wait` call will be placed. The message buffer `outmsg` is not used again within the respective code block and therefore the `MPI_Wait` call is placed at the very end of that block.

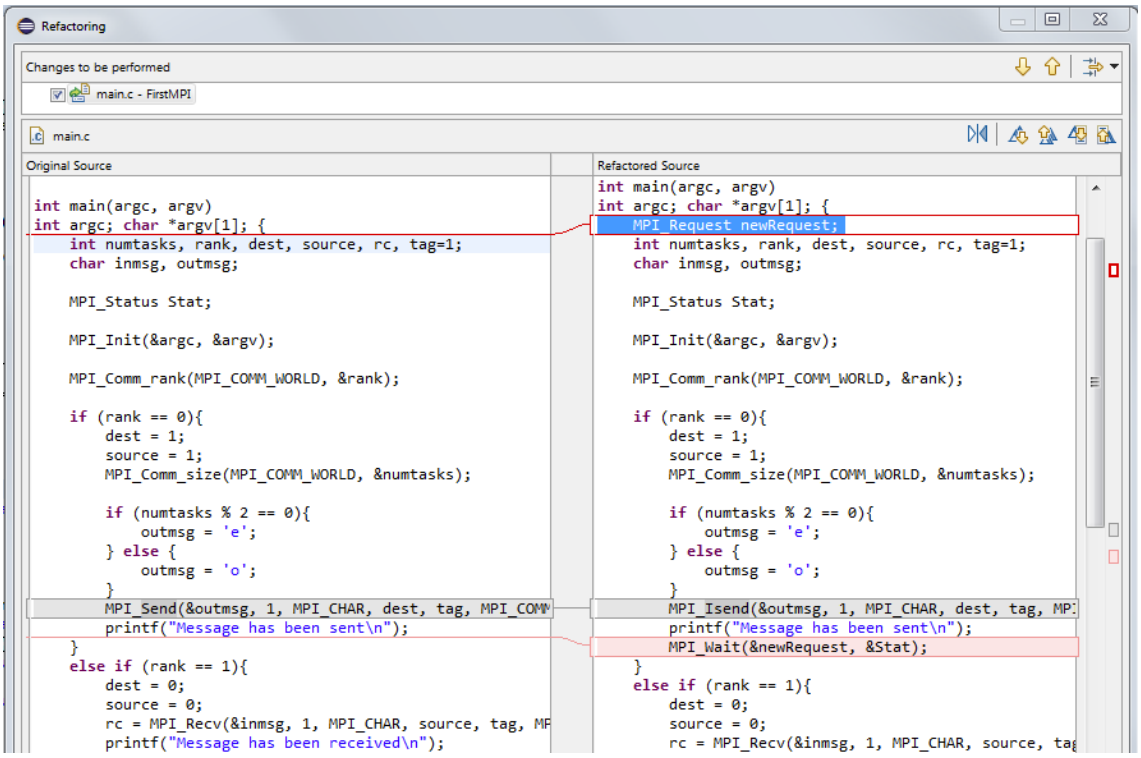

Figure 6.4: Preview window for MPI_Send change

If the user accepts the changes seen in the *Preview* window in Figure 6.4 they press the *OK* button and the refactoring is applied. This also closes the refactoring wizard and displays the Eclipse C/C++ Editor once again, now with the code changes, as shown in Figure 6.5.

```
int main(argc, argv)
int argc; char *argv[1]; {
    MPI_Request newRequest;
    int numtasks, rank, dest, source, rc, tag=1;
    char inmsg, outmsg;

    MPI_Status Stat;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0){
        dest = 1;
        source = 1;
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

        if (numtasks % 2 == 0){
            outmsg = 'e';
        } else {
            outmsg = 'o';
        }
        MPI_Isend(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD, &newRequest);
        printf("Message has been sent\n");
        MPI_Wait(&newRequest, &Stat);
    }
```

Figure 6.5: MPI_Send to MPI_Isend change applied

Now, assume the user will apply the refactoring again, however, this time to the `MPI_Recv` call: again the *Sync to Async* refactoring entry from the CDT refactoring menu shown in Figure 6.2 is selected, but now the `MPI_Recv` expression is highlighted. The user input window opens once again as seen in Figure 6.6 and now the suggested name for the `MPI_Request` parameter displayed for the user is `newRequest` as that name was found in the code (as it was added by the previous refactoring application). In this case, the user decides to keep the default name and clicks the *Preview* button.

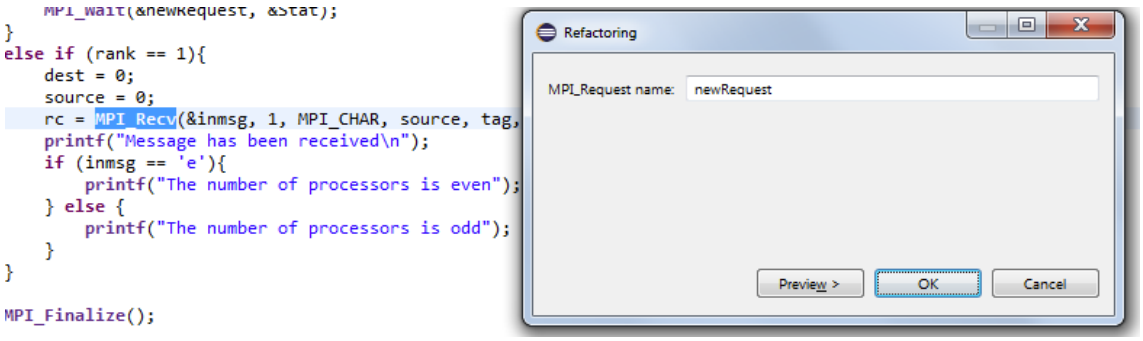

Figure 6.6: Refactoring user input window for MPI_ Recv

The *Preview* window for this `MPI_Recv` to `MPI_Irecv` change is shown in Figure 6.7. No additional parameter of type `MPI_Request` is declared because an existing one was used. The changes to the `MPI_Recv` line are shown as well as the placement of the `MPI_Wait` call. The `MPI_Wait` call is now automatically placed in the middle of the respective code block right before the message buffer `inmsg` is used again.

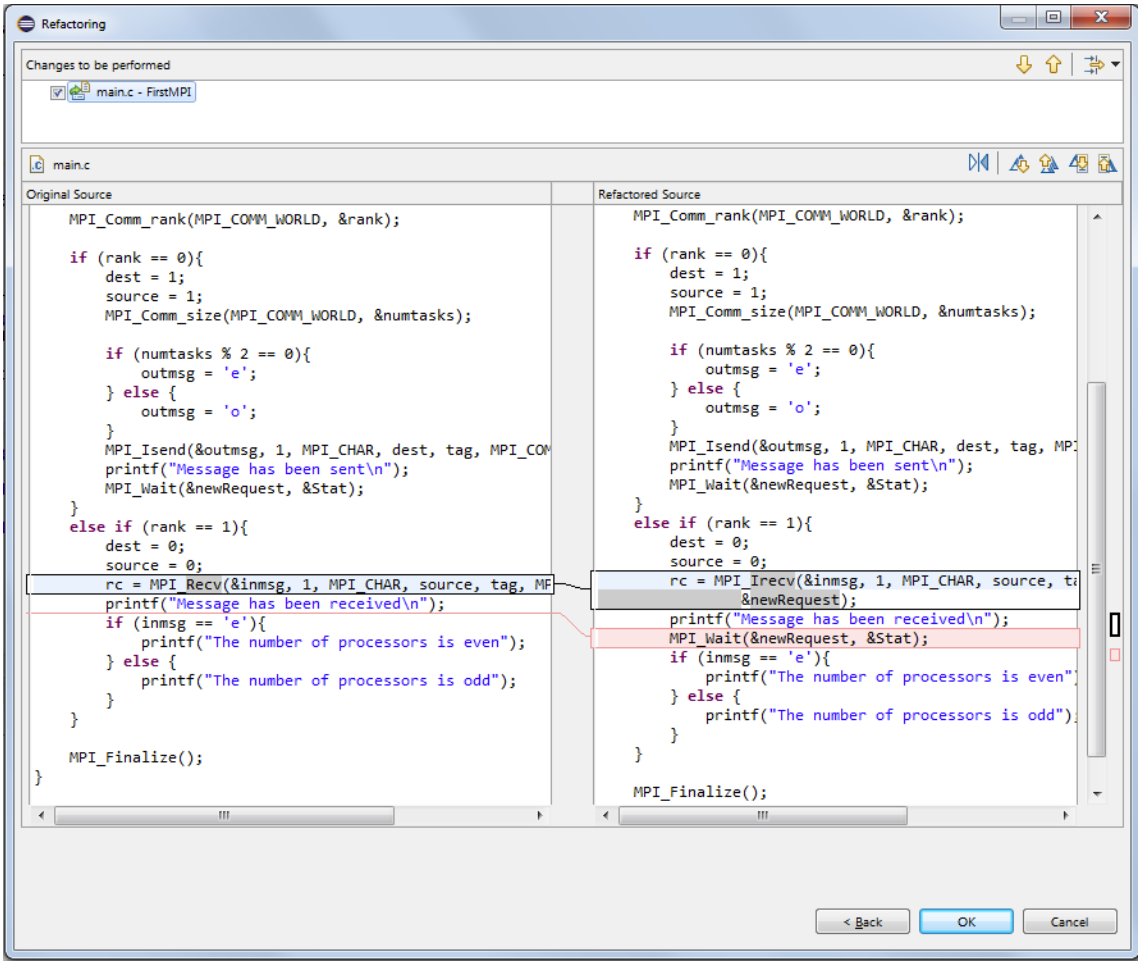

Figure 6.7: Preview window for MPI_Recv change

If the user is satisfied with the changes shown in the preview (Figure 6.7) they click the *OK* button. This applies the changes to the code and closes the refactoring wizard. The Eclipse C/C++ Editor is displayed to the user once again as seen in Figure 6.8 and at this point the refactoring has been applied twice, in different locations.

```
if (rank == 0){
    dest = 1;
    source = 1;
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    if (numtasks % 2 == 0){
        outmsg = 'e';
    } else {
        outmsg = 'o';
    }
    MPI_Isend(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD, &newRequest);
    printf("Message has been sent\n");
    MPI_Wait(&newRequest, &Stat);
}
else if (rank == 1){
    dest = 0;
    source = 0;
    rc = MPI_Irecv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
            &newRequest);
    printf("Message has been received\n");
    MPI_Wait(&newRequest, &Stat);
    if (inmsg == 'e'){
        printf("The number of processors is even");
    } else {
        printf("The number of processors is odd");
    }
}
```

Figure 6.8: MPI_Recv to MPI_Irecv change applied

6.3 After

Listing 6.2 shows the resulting code after the refactoring is applied twice as described above in Section 6.2.

The first application of the refactoring is to the synchronous sending communication. Line 6 shows the added parameter of type `MPI_Request` which is then used in both transformations. The transformation from `MPI_Send` to `MPI_Isend` can be seen in line 26 where the newly created `newRequest` parameter is added to the asynchronous communication call (other input parameters stay the same). Line 28 shows where the `MPI_Wait` call matching the `MPI_Isend` call is added at the end of the code block as the message buffer `outmsg` is not used again within the block.

The second application of the refactoring is to the synchronous receive communication. Line 33 shows the transformation from `MPI_Recv` to `MPI_Irecv` where the already existing `newRequest` parameter is added to the asynchronous communication call. In this case there is also one input parameter which is removed, the last input parameter from the `MPI_Recv` call as seen in line 31 in Listing 6.1 (the `Stat` parameter). Line 35 shows where the `MPI_Wait` call matching the `MPI_Irecv` call is added in the middle of the code block. As seen in line 36 the message buffer `inmsg` is used again while still within the code block and therefore the `MPI_Wait` call needs to be located before that line.

Listing 6.2: After – Simple C code example

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(argc, argv)
5    int argc; char *argv[1]; {
6    MPI_Request newRequest;
7    int numtasks, rank, dest, source, rc, tag=1;
8    char inmsg, outmsg;
9
10   MPI_Status Stat;
11
12   MPI_Init(&argc, &argv);
13
14   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15
16   if (rank == 0){
17     dest = 1;
18     source = 1;
```

```
19      MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
20
21      if (numtasks % 2 == 0){
22        outmsg = 'e';
23      } else {
24        outmsg = 'o';
25      }
26      MPI_Isend(&outmsg, 1, MPI_CHAR, dest, tag,
           MPI_COMM_WORLD, &newRequest);
27      printf("Message has been sent\n");
28      MPI_Wait(&newRequest, &Stat);
29    }
30    else if (rank == 1){
31      dest = 0;
32      source = 0;
33      rc = MPI_Irecv(&inmsg, 1, MPI_CHAR, source, tag,
           MPI_COMM_WORLD, &newRequest);
34      printf("Message has been received\n");
35      MPI_Wait(&newRequest, &Stat);
36      if (inmsg == 'e'){
37        printf("The number of processors is even");
38      } else {
39        printf("The number of processors is odd");
40      }
41    }
42
43    MPI_Finalize();
44  }
```

6.4 Case study conclusion

This chapter demonstrated the semi-automated application (i.e. refactoring location and parameters need to be manually provided, but the actual transformation is then fully automated) of the *Synchronous to Asynchronous communication* refactoring implementation described in Chapter 5 that is based on the corresponding mechanics provided in Section 4.3.1. Using the given code example as input and applying the refactoring to two different locations, the results were as expected. The transformations from `MPI_Send` to `MPI_Isend` on one hand and from `MPI_Recv` to `MPI_Irecv` on the other along with declaring a new parameter of type `MPI_Request` when appropriate gave the expected results as well as the chosen location of the cor-

responding `MPI_Wait` calls. Therefore this case study gives (together with the unit tests covering further combinations of special cases) confidence that the refactoring implementation is correct in its functionality.

7 Conclusion

This chapter concludes this thesis by providing in Section 7.1 a summary and in Section 7.2 an outlook.

7.1 Summary

Software refactoring is widely used and can do a great difference when software aging becomes a problem. The best case is when these refactorings can be automatically applied to code, lowering the risk of human error, reducing the effort of applying them, and taking away the fear of applying it. As described in this thesis the original definition of software refactoring focuses on code readability and maintainability. When looking at HPC the focus becomes different as developers strive to get as much performance as possible. These three aspects (readability, maintainability, and performance) usually cannot be achieved at the same time, e.g. if improving performance means lines need to be added to the code it typically lowers readability and maintainability as well.

The already existing work on HPC refactoring provides different definitions of refactoring. As conventional refactorings can also be used on HPC code to achieve higher readability and maintainability this thesis introduces a new definition for refactorings specific to HPC that adds performance and portability to the original definition of refactoring by Fowler [21] .

Since the existing refactorings already cover readability and maintainability along with HPC-specific portability, this thesis focuses only on optimizing performance. With this focus an HPC refactoring catalogue that follows a format inspired by Fowler [21] is introduced, where each refactoring is described with a name, a description, a motivation, mechanics to apply the refactoring, and code examples. The catalogue currently contains five HPC refactorings:

- Synchronous communication to asynchronous communication
- Multiple send to broadcast

- Multiple send to scatter
- Multiple receive to gather
- Multiple broadcast to all-gather

As part of this thesis, one of the refactorings from the created catalogue has been implemented as a refactoring tool. This is the *Synchronous communication to asynchronous communication* refactoring. The developed refactoring tool is based on the Eclipse Platform and is a plug-in for the C/C++ Development Tooling (CDT). It can be used to semi-automatically apply the refactoring on HPC code written in C/C++. In addition to the implementation, this thesis also describes unit tests for assessing the correctness of the implementation of the automated refactoring. The implementation is open source and can be found at `https://doi.org/10.5281/zenodo.3753576`. Finally, a case study with a simple code example is presented to evaluate the applicability of the automated refactoring.

7.2 Outlook

Currently, the HPC refactoring catalogue is just a start and leaves opportunity to keep searching for further HPC refactorings that should be included. The provided case study is also rather informal and only for the sake of showing that the implemented refactoring's functionality is correct. A larger case study with more complex code could be done showing that it actually makes a difference in performance of the HPC code and that it helps developers, e.g. reduces errors made and time spent in comparison with applying the refactoring manually.

For the implemented refactoring the developer still has to detect the opportunity for the refactoring. While Fowler states "no set of metrics rivals informed human intuition" [21] it could be of great value for the developer to be able to automatically get suggestions for possible opportunities for refactoring. This could be solved by making the search for this *code smell* automatic [40].

The fact that only one refactoring is implemented as proof of concept opens up the opportunity for implementing more refactorings related to HPC, both from the catalogue in this thesis and others that have yet to find their way there.

All in all HPC refactoring is a relevant and current topic in the ever-growing HPC community and worth putting the effort in to automate as much as possible, along with improving IDE support for HPC in general.

Bibliography

[1] Brandon Barker. Message passing interface (MPI). In *Workshop: High Performance Computing on Stampede*, volume 262, 2015.

[2] Richard S. Barr and Betty L. Hickman. Reporting computational experiments with parallel algorithms: Issues, measures, and experts' opinions. *ORSA Journal on Computing*, 5(1):2–18, 1993.

[3] Victor R. Basili, Daniela Cruzes, Jeffrey C. Carver, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Marvin V. Zelkowitz, and Forrest Shull. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE software*, 25(4):29–36, 2008.

[4] Kostadin Damevski and Madhan Muralimanohar. A refactoring tool to extract GPU kernels. In *WRT '11: Proceedings of the 4th Workshop on Refactoring Tools*, pages 29–32, May 2011.

[5] Class ASTRewrite. `https://www.cct.lsu.edu/~rguidry/eclipse-doc36/org/eclipse/cdt/core/dom/rewrite/ASTRewrite.html`. Accessed on 2020-05-15.

[6] Ryusuke Egawa, Kazuhiko Komatsu, and Hiroyuki Kobayashi. Designing an HPC refactoring catalog toward the exa-scale computing era. In *Sustained Simulation Performance 2014*, pages 91–98. Springer, 2015.

[7] Ryusuke Egawa, Kazuhiko Komatsu, and Hiroyuki Takizawa. Designing an open database of system-aware code optimizations. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pages 369–374, Nov 2017.

[8] Message Passing Interface Forum. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015.

[9] Eclipse Foundation. About the Eclipse foundation. `https://www.eclipse.org/org/`. Accessed on 2020-05-09.

[10] Eclipse Foundation. About the Eclipse project. `https://www.eclipse.org/eclipse/`. Accessed on 2020-05-09.

[11] Eclipse Foundation. CDT overview. https://help.eclipse.org/2019-09/index.jsp?topic=%2Forg.eclipse.cdt.doc.user%2Fconcepts%2Fcdt_c_over_cdt.htm. Accessed on 2020-05-16.

[12] Eclipse Foundation. Code Style preferences. https://help.eclipse.org/2020-03/index.jsp?topic=%2Forg.eclipse.cdt.debug.application.doc%2Freference%2Fcdt_u_c_code_style_pref.htm. Accessed on 2020-05-16.

[13] Eclipse Foundation. Eclipse CDT? https://www.eclipse.org/cdt/. Accessed on 2020-05-09.

[14] Eclipse Foundation. Eclipse Java development tools (JDT). https://www.eclipse.org/jdt/. Accessed on 2020-05-09.

[15] Eclipse Foundation. Eclipse PTP. https://www.eclipse.org/ptp/. Accessed on 2020-05-09.

[16] Eclipse Foundation. FAQ what is LTK? https://wiki.eclipse.org/FAQ_What_is_LTK%3F. Accessed on 2020-05-09.

[17] Eclipse Foundation. FAQ What is the plug-in manifest file (plugin.xml)? https://wiki.eclipse.org/FAQ_What_is_the_plug-in_manifest_file_(plugin.xml)%3F. Accessed on 2020-05-16.

[18] Eclipse Foundation. PDE. https://www.eclipse.org/pde/. Accessed on 2020-05-09.

[19] Eclipse Foundation. Platform architecture. https://help.eclipse.org/2019-12/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Farch.htm. Accessed on 2020-05-16.

[20] Eclipse Foundation. The Workbench. https://help.eclipse.org/2020-03/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2FgettingStarted%2Fqs-02a.htm&cp%3D0_1_0_0. Accessed on 2020-05-28.

[21] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[22] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[23] Leif Frenzel. The language toolkit: An API for automated refactorings in eclipse-based IDEs. https://www.eclipse.org/articles/Article-LTK/ltk.html. Accessed on 2020-05-09.

[24] J. Daniel Garcia. Refactoring software to heterogeneous parallel platforms. *The Journal of Supercomputing*, 75(8):3997–4000, 2019.

[25] Björn-Soren Gigler, Alberto Casorati, and Arnold Verbeek. Financing the future of supercomputing: How to increase investment in high performance computing in europe. `https://www.eib.org/attachments/pj/financing_the_future_of_supercomputing_en.pdf`. Accessed on 2020-05-09.

[26] Emanuel Graf, Guido Zgraggen, and Peter Sommerlad. Refactoring support for the C++ development tooling. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 781–782, 2007.

[27] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2011.

[28] Fredrik B. Kjolstad, Danny Dig, and Marc Snir. Bringing the HPC programmer's IDE into the 21st century through refactoring. In *SPLASH 2010 Workshop on Concurrency for the Application Programmer (CAP'10). Association for Computing Machinery (ACM)*, 2010.

[29] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 4 1965.

[30] MPI forum. `https://www.mpi-forum.org/`. Accessed on 2020-05-09.

[31] MPICH. MPI_Allgather. `https://www.mpich.org/static/docs/v3.1.x/www3/MPI_Allgather.html`. Accessed on 2020-05-09.

[32] MPICH. MPI_Bcast. `https://www.mpich.org/static/docs/v3.1.x/www3/MPI_Bcast.html`. Accessed on 2020-05-09.

[33] MPICH. MPI_Gather. `https://www.mpich.org/static/docs/v3.1.x/www3/MPI_Gather.html`. Accessed on 2020-05-09.

[34] MPICH. MPI_Irecv. `https://www.mpich.org/static/docs/v3.1.x/www3/MPI_Irecv.html`. Accessed on 2020-05-09.

[35] MPICH. MPI_Isend. `https://www.mpich.org/static/docs/v3.1.x/www3/MPI_Isend.html`. Accessed on 2020-05-09.

[36] MPICH. MPI_Recv. `https://www.mpich.org/static/docs/v3.1.x/www3/MPI_Recv.html`. Accessed on 2020-04-10.

[37] MPICH. MPI_Scatter. `https://www.mpich.org/static/docs/v3.1.x/www3/MPI_Scatter.html`. Accessed on 2020-05-09.

[38] MPICH. MPI_Send. `https://www.mpich.org/static/docs/v3.1.x/www3/MPI_Send.html`. Accessed on 2020-05-09.

[39] MPICH. MPI_Wait. `https://www.mpich.org/static/docs/v3.1.x/www3/MPI_Wait.html`. Accessed on 2020-05-09.

[40] Helmut Neukirchen and Martin Bisanz. Utilising code smells to detect quality problems in TTCN-3 test suites. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science (LNCS)*, pages 228–243. Springer, Berlin, Heidelberg, 2007.

[41] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[42] Behrooz Parhami. *Introduction to parallel processing: algorithms and architectures*. Springer, Boston, MA, 2002.

[43] David L. Parnas. Software aging. In *Proceedings of 16th International Conference on Software Engineering*, pages 279–287, May 1994.

[44] Morris Riedel. personal communication, 2017.

[45] Morris Riedel. High performance computing: Lecture 9 performance optimization & tools. `http://morrisriedel.de/wp-content/uploads/2018/03/HPC-Lecture-9-HPC-Performance-Optimization-and-Tools-v1.pdf`, October 2017. lecture slides.

[46] Michael Rüegg. Eclipse CDT refactoring overview and internals. In *Parallel Tools Platform (PTP) Workshop, Chicago*, 2012.

[47] Karl Rupp. 42 Years of Microprocessor Trend Data. `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`. Accessed on 2020-06-01.

[48] Doug Schaefer. 10 things you probably didn't know about the CDT. `https://www.eclipse.org/community/eclipse_newsletter/2013/october/article1.php`. Accessed on 2020-05-09.

[49] Markus Schorn. Using CDT APIs to programmatically introspect C/C++ code. `https://wiki.eclipse.org/images/c/c7/CDT_APIs_for_code_introspection.pdf`. Accessed on 2020-05-09.

[50] Jan Verschelde. Introduction to supercomputing. `http://homepages.math.uic.edu/~jan/mcs572/MCS572.pdf`, Nov 2016. lecture notes.

[51] Wes Kendall. MPI Scatter, Gather, and Allgather. `https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/`. Accessed on 2020-05-18.