

UNIVERSITY OF ST ANDREWS

CS5099 INDIVIDUAL COMPUTER SCIENCE DISSERTATION

Java Parallel Refactorer

Michael Lynch
24th August 2018

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is NN,NNN* (FIX THIS LATER) words long, including project specification and plan. In submitting this project report to the university of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Abstract

abstract here

Contents

1. Introduction

With almost all devices today using multicore processors, parallelism provides developers with the opportunity to take advantage of more of the resources on their system and speed up the execution of computationally intensive algorithms. A simple tool for introducing parallelism into applications can become a powerful asset to a developer who wants to work more efficiently. The use of a parallel refactoring tool allows developers to reduce time spend on producing parallel patterns for embarrassingly parallel problems and can instead focus on more complex areas of their project.

Java is consistently ranked in the top 3 of the most popular programming languages in the world[?] and is present on almost every platform including android. With such popularity, creating tools for the Java platform is likely to reach a wide audience.

In this project, I have created a console based application which allows Java developers to introduce a parallel approach to their programs with the goal of improving their software performance. This refactorer allows the developer to choose for loops from within their code which they would like to refactor into a parallel version. The refactorer performs safety checks to ensure a thread safe parallel version of the developer's code can be generated.

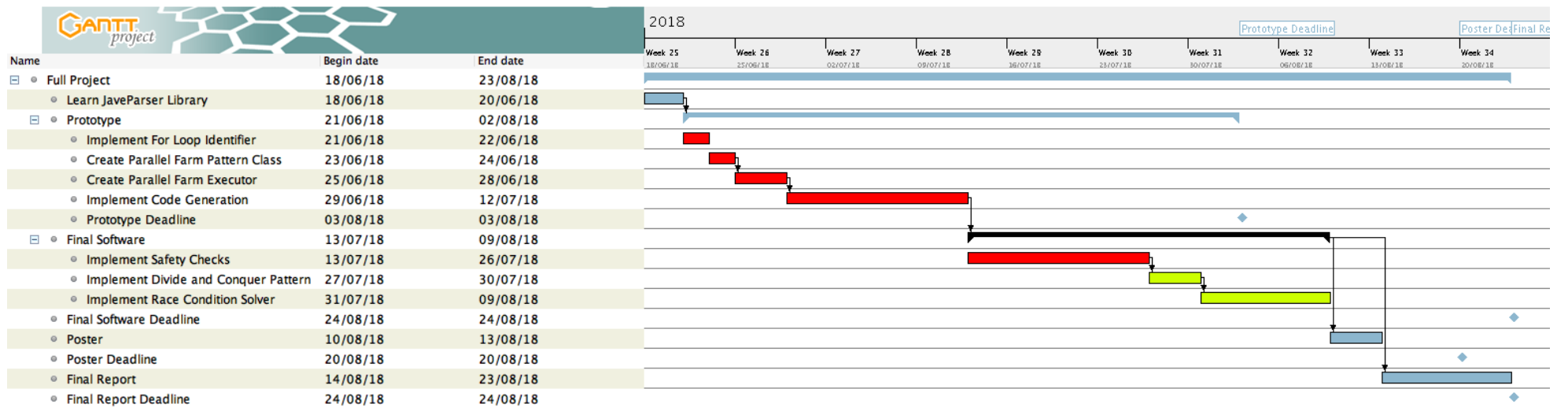
The developer can then select important options for the implementation of the parallel version, such as the number of chunks to split the loop into as well as the number of threads to parallelise over.

To realise this project, several existing technologies are used alongside newly created technology designed specifically for use within this project.

As refactoring is an important aspect of the project, code syntax and semantic analysis is necessary to perform this task, the Java Parser[?] along with its counterpart library Java Symbol Solver provide the groundwork to providing this. these libraries provide the ability to represent the source code as an Abstract Syntax Tree (AST) as well as give a limited semantic understanding of the types of the variables being used within the program. Built on top of this technology, several additional semantic analysers have been created within this project to specifically help in developing parallel code.

Java's standard threading technology (`java.lang.Thread`) is also used to develop an algorithmic skeleton and task execution parallel library. The refactorer generates code that uses this newly developed library making it the underpinning of all parallelism within programs in which the refactoring tool has been used on. This library provides, along with handling the execution of parallel tasks, protection from some deadlock conditions through its deadlock detection and resolving capability.

2. Plan



3. Context Survey

Refactoring code (restructuring the source code of a program while retaining its original behavior) has a history almost as long as computer science itself. Refactoring is traditionally used to improve the readability of the source code making it easier to maintain. As software projects become larger, refactoring to restructure the code base becomes a necessity in order to allow programmers to continue.

As the practice of refactoring has become such a necessary part of software development, many IDEs have incorporated refactoring tools to make the more common and mundane forms of refactoring a more simple process. The Eclipse IDE offers automatic refactoring to rename variables and classes as well as extracting functions from blocks of code to make large methods more readable to the programmer.

Although nearly all major IDEs as well as some text editors offer these simple refactoring techniques as an automated process, very few offer any more complex refactoring processes such as the introduction of design patterns. Some work has been carried out to create automatic design pattern tools such as the work of Mel O’Cinneide[?] as well as the Korean Advanced Institute of Science and Technology[?]. Despite this work being carried out almost two decades ago, these types of tools have not seemed to have made their way into the mainstream software development process.

Over the last decade there have been a number of projects for converting serial programs to parallel using automated refactoring tools, moving away from the traditional use of refactoring as restructuring code for readability to refactoring for functional benefit. Refactoring tools for parallelism have already been demonstrated for Java in *A Refactoring Approach to Parallelism*[?] as well as in C++ and Erlang in the ParaPhrase project[?].

With the modern approach to improving computational power coming through increases in the number of cores on chips, Developers are looking to parallelism to provide a powerful boost in their program’s performance.

Parallelising an originally serial program is not a trivial process however, making this area of research very important for improving developer workflows.

Danny Dig’s Java refactorer[?] shows the capability of such tools for programmers in increasing their productivity as well as reducing the number of errors present in their program code. When Dig’s refactorer was tested in making serial code thread-safe against manual refactorings by open-source developers, it was found that the refactoring tool was able to produce thread-safe code more quickly with fewer errors[?].

A problem noted by Danny Dig with the refactoring tool created is that, unlike typical code restructuring refactor operations, refactoring for parallelism often increases the complexity of the source code rather than decreasing it. A solution suggested in the paper introducing the tool is to provide two versions of the code which can be maintained by an IDE allowing the programmer to switch between the sequential, and thus more understandable code, and the real code[?].

Although this solution holds merit, it can lock programmers to a specific IDE and in some ways

obfuscates the code further by providing multiple versions.

Work on the ParaPhrasing project[?] provides a different approach to refactoring programs to work in parallel which somewhat mitigates the issues of code base obfuscation.

Rather than making structural changes to the code as is done with Dig's refactoring, the ParaPhrase refactoring allows the programmer to introduce a parallel pattern to their code. These pattern calls, although more complicated than the original code, are often still relatively simple to read with some examples incorporating almost identical code to the original in the pattern skeleton.

Original Code

```
1 int main() {
2
3     int iArray[1000];
4
5     for(int i = 0; i < 1000; i++) {
6         iArray[i] = 10;
7     }
8
9     return 0;
10 }
```

Refactored using ParaFormance (The successor to ParaPhrase)

```
1 class Closure {
2     int *iArray;
3
4 public:
5     void operator()(const blocked_range<int> &range) const {
6         for (int i = range.begin(); i != range.end(); ++i) {
7             iArray[i] = 10;
8         }
9     }
10
11     Closure(int * &iArray) :
12         iArray(iArray) {
13     }
14 };
15
16 int main() {
17
18     int iArray[1000];
19
20     parallel_for(blocked_range<int>(0, 1000), Closure(iArray));
21
22     return 0;
23 }
```

As can be seen, the code on lines 5 to 7 in the original code which was chosen for refactoring is very similar to the refactored code on lines 5 to 7. The refactored code is still relatively easy to understand.

A major feature of the ParaFormance refactoring which succeeded the ParaPhrase research[?] is the ability to perform safety checks to ensure the code highlighted by the programmer is thread safe or if it needs to be rewritten in preparation for parallelisation. By performing these checks, the programmer can be informed if potential race-conditions as well as other problems introduced in parallel programs.

Although a great strength of the ParaFormance platform, the safety checker can produce warnings on parallelisable code if this code cannot be parallelised using the patterns available to the tool. This

could mislead some programmers into believing a computationally intensive area of their program cannot be parallelised. Due to these kind of issues present in parallel refactoring tools, it is clear that they must currently only be used to supplement the workflow of a programmer experienced in parallelism, not a novice. This can cause significant problems for attempting to bring these tools into mainstream software development. The benefit for IDEs such as eclipse to only offer very simple refactoring tools such as symbol renaming and method extraction is that it will work almost every time as the programmer expected without errors or warnings for them to handle. An area which has been quite heavily researched for parallelism refactorers, is their capability of detecting areas of a sequential program that are a good candidate for running in parallel. This feature can become an invaluable tool to developers working on projects with very large code bases as attempting to manually search for areas of the project that can be sped up through parallelism is (ironically) very time consuming.

Introduces in *Finding Parallel Pearls*[?] a method of detecting areas of a programmer's code that may be well suited for introducing a parallel approach, allowing for an increase in speed of execution of this section of the program. This process of detecting parallelisable code uses anti-unification to find the similarities between the developer's code and an already known algorithm which has a parallel equivalent algorithm. Having been matched to this algorithm with a parallel version, the code can be sped up using this parallel pattern with relative ease.

the detection of parallelisable code within a developers project and transforming it into a relevant parallel version is only part of the problem involved with introducing parallelism to a program. Often it is not advisable to introduce parallelism into every area of code in which it can be as a situation may be created where many more threads have been produced than cores on the machine leading to constant context switching and a reduction in performance.

Cost Directed Refactoring for Parallel Erlang Programs[?] gives developers the opportunity to make empowered decisions on where to use parallelism, and where not to, by informing the developer of the sort of performance they can expect for parallelising different sections of code. With this knowledge the code sections that do not appear to give good parallel performance can be ignored in favour of more effective areas.

Another way in which developers can make more informed decisions on where in their program to use a parallel algorithm is by knowing how much of a program's execution time is spent on a particular section of code. This is a feature used in the previously mentioned ParaFormance refactoring

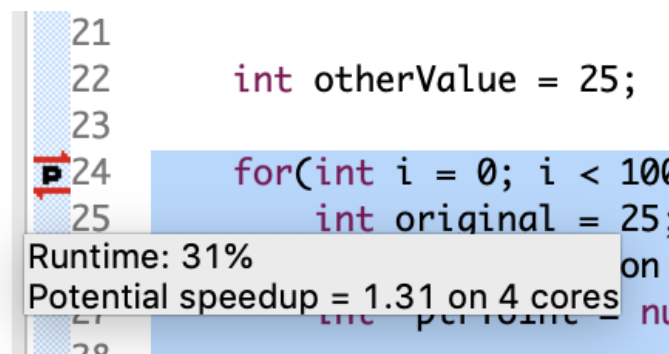


Figure 3.1: Runtime label shows the percentage of the programs execution time is spent on the selected section of code

This give a percentage measurement of how much time is being spent by the program on this

section meaning that the developer can spot the areas of the code that are the most computationally intensive.

Refactoring for parallel is becoming an important area of research in the current age of multi-core devices. There is however, still a long way to go to ensure that such tools can be made accessible to more novice parallel developers.

4. Objectives

4.1 Primary Objectives

- P1 The refactorer will allow for the developer to choose a section of the code, and then perform refactoring to create an instance of a task farm which can carry out that section in parallel.
- P2 The refactorer will take as argument a Java source file which is the target of refactoring along with a path to the developer's project to allow the refactorer to understand the source file in its wider context.
- P3 The refactorer will perform limited safety checks on the code to ensure its parallelisability. The refactorer will consider some race conditions as well as detecting the use of data streams within a for loop which may be written/read from in the incorrect order when moved into parallel.
- P4 The source code file will be rewritten, replacing the highlighted code with a call to a task farm. The refactoring will preserve the lexical formatting of the source code.
- P5 When calling the task farm, the developer will be able to specify the number of threads they wish the farm to run on.

4.2 Secondary Objectives

- O1 The refactorer may provide an estimation of how much time is saved from parallelising the chosen piece of code.
- O2 The Divide & Conquer parallel pattern may be added to enable the refactoring of a greater number of parallelisable problems.
- O3 Race Condition Solver: For specific race conditions found in the developer's code, the program may be able to rewrite the code in a way which removes race conditions from a section of code that is intended to be parallelised. This is likely dependent on objective O2.

5. Ethics

There are no ethical considerations for this project. Ethics form is available within the appendices section 9.4

6. Design and Implementation

6.1 Code Structure

This project follows a modular structural architecture within multiple layers of the project. The project is split into two main code bases which are responsible for the refactorer software itself along with the parallel library in which the refactorer generates code for. Given the radically different focuses of these two sections of the project, it was natural to fully separate them from each other. With this structure, the refactorer can be modified in its entirety without effecting the functionality of the parallel library. Much of the parallel library can also be modified without effecting the functionality of the refactorer however, if the method for implementing algorithms in the parallel library changes, the refactorer will need to be updated to ensure that the code it generates is correct.

Parallel Library

The parallel library can be used entirely apart from the refactorer as a way for developers to introduce parallelism to their programs manually should they wish.

The library uses algorithmic skeletons to easily introduce parallelism to a program allowing the user to implement built in parallel algorithms (in the case of this library, the parallel for algorithm) with minimal effort. The library implements the more complex systems involved in parallelism in the background, simplifying the process for the developer.

As with the overall architecture of the project, the library follows a similar modular approach. Although the parallel library only implements the parallel for loop algorithm, the modular nature of the libraries code means that new task farm algorithms can very easily be introduced to the library at a later date.

Refactorer

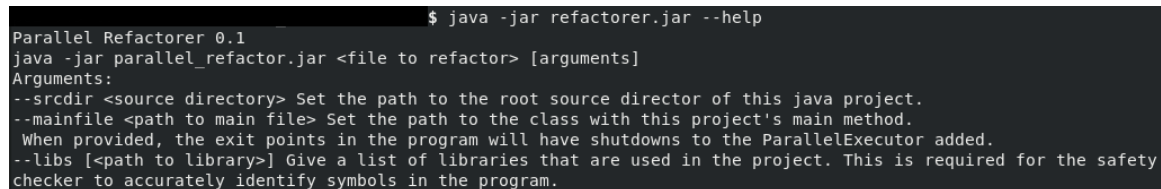
The modular design philosophy is also extended to the source code of the refactorer. There are three main java packages that make up the refactorer java code: Code generation, Safety and Discovery. The Code generation package, as the name suggests is charged with taking the original source code of the developer and converting it into a parallel equivalent source. The safety package is used to check the section of code being refactored for compatibility with parallelism. It mostly detects potential race conditions and warns the developer when such an event could occur within their code. Finally, the Discovery package, is a helper package to be used mostly by the other two packages. Discovery contains many of the more general semantic analysers as well as classes to ease in navigation of the source code's AST. Although both Code generation and Safety depend upon the Discovery package, They are completely separate from each other meaning development of them can be performed safely without them breaking one another.

In the execution of the refactoring program itself, a pipeline software structure is used. Infor-

mation is taken from the user of the program as well as semantic understanding of the source code and is together sent to the code generator. The code generator creates the new parallelised program from the input source code and outputs it back into the source file.

Command Interface

The refactorer is a terminal based application. In order to maintain a good user experience, standard unix formatting of program arguments is used.



```
$ java -jar refactorer.jar --help
Parallel Refactorer 0.1
java -jar parallel_refactor.jar <file to refactor> [arguments]
Arguments:
--srcdir <source directory> Set the path to the root source director of this java project.
--mainfile <path to main file> Set the path to the class with this project's main method.
When provided, the exit points in the program will have shutdowns to the ParallelExecutor added.
--libs [<path to library>] Give a list of libraries that are used in the project. This is required for the safety
checker to accurately identify symbols in the program.
```

Figure 6.1: The command line printout of the help argument

6.2 Refactorer

6.2.1 Discovery

Locating Valid For Loops

With the refactorer designed to parallelise the user's for loops, the first task of the program is to locate all the for loops present in the user's source code and list them for the user to choose the loop they would like to be parallelised. The JavaParser library makes this relatively simple to achieve as nodes in the AST can be searched for all sub-nodes of a specific type, in this case a standard for loop.

Once the loops have been listed, the user can choose a loop based on its position in the source file by line, and column if necessary where multiple for loops are present on the same line.

With a loop chosen, the refactorer holds information on the position of the loop in the user's code as well as its contents in preparation for its transformation into a parallel version.

Shared Variables

An important element to preparing a for loop to be parallelised is to establish which variables exist outside of the loop and which are localised to a single iteration of the loop.

The refactored structure of the code moves the contents of the for loop to a new Object in the code for execution in parallel. This means that the parallel code is running in a different variable environment to the original loop. In order to ensure that the parallel code is able to make changes to the variables in the original environment, the program produces copies of the variables that the loop requires access to. These are the shared variables of the loop.

Understanding which variables are shared and which are not is also important in checking for race conditions as will be shown later in Safety.

Detecting Shared Variables

Checking for the shared nature of a variable is a relatively simple process of establishing where a variable was first declared in the source code.

A variable declared within the loop is local to that loop, as such, when a variable declaration is found within

the loop, the variable being declared is not shared. If no such declaration is found then the variable is shared. Unfortunately simply checking whether a variable has a declaration within the loop is not enough information to show that it is not shared data. In java, variables that are declared within a block statement are local to that block only.

```
1  if(true) {  
2      int a = 10;  
3  
4      System.out.println(a);  
5  }  
6  
7  //The a in this context does not refer to the a declared on line 2  
8  System.out.println(a);
```

The variable declaration for a variable reference can be determined by only moving vertically up the abstract syntax tree of the source until the declaration for the variable is found. If the declaration is found to be within the loop that is to be refactored then this means that the variable in question is local to an iteration of the loop and is not shared with either other iterations of the loop or the rest of the program. As the purpose of the traversal of the AST is just to determine whether or not the variable is shared, once the search in the AST moves outside the refactored loop, this shows that the declaration exists somewhere outside the loop proving that the variable is shared. The search for the variable's declaration can now be cut short.

The the refactorer's source code, checks for shared data are performed in the `SharedDataDetector` class within the discovery package. The tree search for the variable's declaration is performed within the `findVarDecInTreeTraversal` method. As the detection of shared variables is important in both the for loop to be refactored as well as methods that may be called within the loop, The `SharedDataDetection` class is abstract to allow for changes to the process of traversing the tree for these different situations. When searching for shared variables within the refactored for loop, the final node to check before ending the search is the loop itself. When searching for shared variables within methods the final node is the method declaration. These changes to the way the tree is traversed is given using the `addTreeTraversalSpecialCase` method which takes as parameters, a piece of code to run on a specific node type along with the corresponding node type.

By designing the shared data detector in this way, it can very easily be extended to check for shared data in other AST structures in the future.

6.2.2 Safety

(Have you said this somewhere in your context review??) Transforming a sequential program into one that takes advantage of parallelism is often not a trivial process. With parallel programs, an added layer of complexity is added as multiple threads may be attempting to access and change the same piece of data at the same time.

An important part of the refactorer I have produced is its ability to warn the user of potential pitfalls that may exist within their code, whether that be through race conditions or changes being made in one iteration of a loop that may effect another.

Loop Structure Safety

This refactorer works specifically with one type of loop, a Fortran style loop where the number of iterations is set prior to starting the loop and does not change during its execution. This limits the acceptable types of for loops from what is available within the Java programming language.

Under Java, the execution of a loop can be prematurely ended through two main means. First, the `break` statement allows for a loop to immediately be stopped and all later iterations to not take place. This poses a problem for transforming the loop into parallel, as multiple iterations can be

taking place at the same time the meaning of a break statement becomes somewhat ambiguous. Does the break statement mean that this specific iteration should be stopped? Or does it mean that all other iterations within the for loop should stop? What about iterations that have a lower iterator number than the iteration that called the break statement?

Considering these major problems with the break statement it's clear that allowing it to be used in a parallel system does not make sense.

The first safety check made by the refactorer is to find these kinds of problematic break statements and inform the user that such a loop cannot be successfully refactored until such a statement is removed. As with finding variable declarations for variable references, finding problematic break statements is similarly not simply a boolean check to see if a break statement exists anywhere within the loop. A break statement is acceptable within the loop as long as it is being used to break out of some other statement that exists within the code such as a different loop or a switch statement. A similar approach to finding variable declarations for shared variables is employed to check for the statement being broken by a break statement. The search continues up the AST until a structure is found that can be broken out of and this is considered to be what the break statement is referring to. If this statement is the loop to be refactored then this is a problematic break statement and the refactorer should discontinue the attempt to parallelise the loop.

A notably missing edge case with regards to break statements is present in the refactorer. The Java programming language allows for the labeling of for loops which can then be referenced by a break statement in an area of the program where it would not typically break out from. This is an area of the safety checker which can be improved given more development time.

Variable Safety

Accessing variables in multiple iterations of a parallel for loop can lead to race conditions in many cases. This is the other purpose of the SharedDataDetector. Shared variables are inherently unsafe to use in parallel as they may be being used by multiple threads at the same time. This means that the use of shared variables must strictly be read only. Checking for writes to a variable is relatively simple. Searching the AST for assignment expressions produces a list of all the potential problems with variable safety that may be found. If the assignment expression is assigning to a variable that has been found to be shared, then this is given as a warning to the user that there is a race condition.

There is one condition however when writing to a shared variable is acceptable and this exception is identified by the safety checker. Consider the following loop:

```
1  int a = 0;
2
3  for (int i = 0; i < 100; i++) {
4      a = 10;
5  }
```

This loop will obviously set the shared variable 'a' to 10 in every iteration of the for loop on line 3 in this case the contents of 'a' is certain and therefore multiple writes from different iterations in the loop even in parallel will never give a differing result. As long as 'a' is never read from or written to again in the loop this is an acceptable structure which will not produce a safety warning.

When writing directly to a variable, evaluating for if the variable will cause a race condition is simply an act of checking whether the variable being assigned to exists outside of the refactored loop's environment or not. When making writes to sub-variables however, the process becomes more complex. Java uses a referencing system to handle objects which allows for multiple different variables to reference the same object. This causes problems for assessing the shared nature of a variable when a variable that has been declared within the refactored loop references an object that was defined outside of the loop.


```

1 Foo a = new Foo();
2
3 //Loop to be refactored
4 for(int i = 0; i < 100; i++) {
5     Foo b = a;
6
7     //Although this assignment is to the non-shared variable b
8     //It references the shared variable a and therefore can cause a race condition
9     b.bar = i;
10 }

```

In this situation, data-flow analysis is required to determine the safety of an assignment even if the variable being edited is not shared. The challenges involved with fixing this problem will be covered more thoroughly later.

Array Access Safety

Checking for the safety of an array access produces a number of challenges not reflected in a typical variable write. Where writing to a shared variable is always considered to be a race condition, writing to an element within an array is not in all cases. As long as a write to a specific array element is not attempted across multiple iterations of the loop, then the write to the element is safe in parallel. Although this cannot always be determined, there are common situations in which it can be.

A parallel for loop contains an iterator which is an integer unique to every iteration of the loop. When using the iterator as the index to an array, it is guaranteed that the element of that array being accessed is unique to every other iteration within the loop.

```

1 int[] a = new int[100];
2 //i is the iterator for this loop
3 for(int i = 0; i < 100; i++) {
4     //Valid access to the array 'a'
5     a[i] = i;
6
7     //Invalid access to the array 'a'
8     a[2] = i;
9 }

```

Although this in itself is useful for many applications such as initialising a large array or copying data between arrays, functionality can be improved further by making other observations about the use of the iterator as an index.

Considering that applying the iterator as the index guarantees that the element being accessed is unique to that iteration, it must also hold that offsets and multiples of the iterator must also be valid.

```

1 int[] a = new int[100];
2
3 for(int i = 0; i < 100; i++) {
4     //Valid access to the array 'a'
5     a[(2*i)+2] = i;
6 }

```

Allowing for this kind of functionality however comes with additional checks that need to be taken to ensure safety. Unlike simply using the iterator as the only valid index, this system allows for many possible valid array access indexes, but not within the same loop.

```

1 int[] a = new int[101];
2 for(int i = 0; i < 100; i++) {

```

```

3   a[i] = i*5;
4   a[i+1] = i*5
5 }

```

Taken separately, the array writes on line 4 and 5 appear to be valid, however they need to be taken in context of the rest of the loop. A write on line 5 causes a race condition as the next iteration in the loop will also write to the same element when it executes its assignment on line 4.

Multiple element writes to the same array need to be checked to ensure that the writes are always going to the same element. A simple string or AST comparison of the index expression is not enough to establish whether or not this is the case however (index $i+1$ and $1+i$, although equivalent, are not the same structurally in the AST), the expression itself needs to be evaluated to check if the two expressions are truly equal.

In order to check if two index expressions are truly equal, the expression is treated as a mathematical equation to be evaluated. Using the `SimplifiedEquation` class, all index equations are simplified into a standardised format which then allows them to be compared against all other index equations.

This standard form is simply a count of the number of iterators in the expression ($i*2$ counts as two iterators) as well as the number of constant values there are ($i+2$ counts as one iterator and two as constant value). Some assumptions on the format of the index expression are made in order to simplify:

- The only identifier that may be present in the index expression is the for loop's iterator identifier
- All other terminating nodes within the expression must be numerical literals
- There must be no powers of the iterator (e.g. $i*i$)

If the expression fails any of these assumptions the simplifier fails and assumes that the given index expression is unsafe.

In order to find the standard simplified equation the `SimplifiedEquation.createSimplifiedEquation(...)` method is used. This recursively works through the sub-nodes of the expression, counting up the iterators and constant values of each and then performs the appropriate operations to merge them together

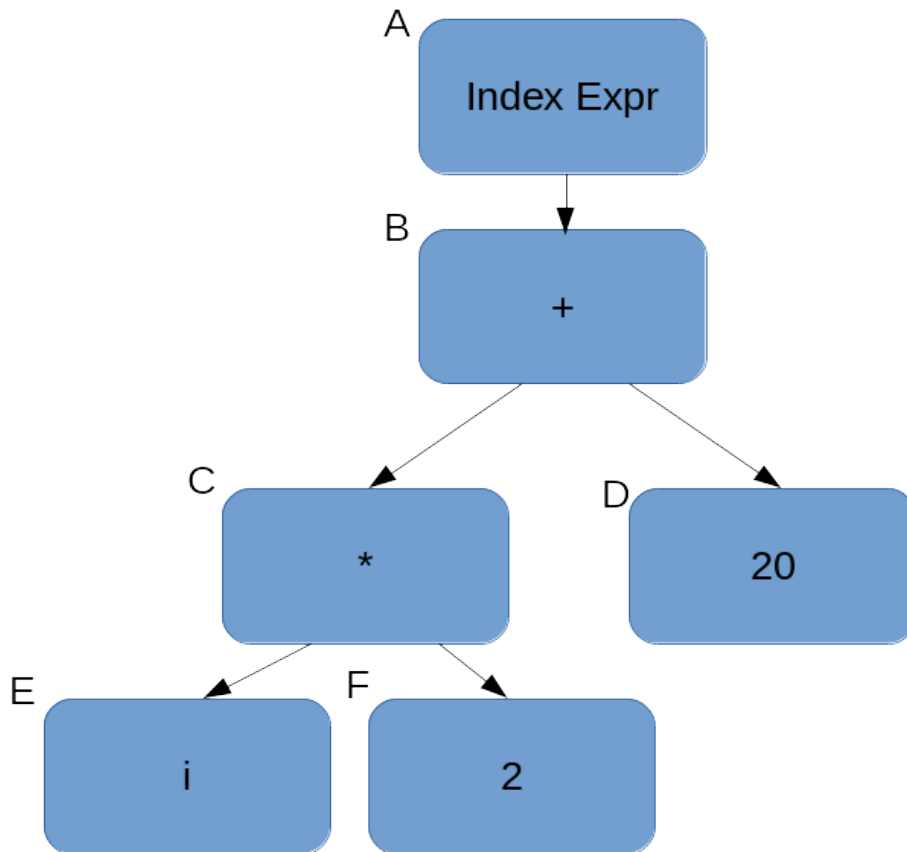


Figure 6.2: Index Expression AST

In figure 8.1 the given expression would be evaluated by starting at the bottom of the AST and working its way up. The evaluator would evaluate node E to be worth 1 i(terator) and 0 c(onstant). Node F would be worth 0i and 1c. Nodes E and F are multiplied. In a multiplication, the node with only constants has its constant value multiplied against the other node's iterator value. This means node C is worth 2i and 0c. Finally node B is an addition of node C (2i) and node D (20c). An addition operation simply adds the i and c values of the two nodes making B worth 2i and 20c which is the simplified equivalent to the expression $i*2 + 20$. This can then be compared against different expressions such as $i*(1+1) + 2*10$ and give the same result. This gives the required proof that two index expressions point to the same element in an array and can therefore appear in the parallel for loop without causing a race condition.

Method Call Safety

Where calls to methods exist within the refactored for loop, checks need to be made to ensure a race condition is not being created. Methods can be a potentially unsafe area of the loop as some of them contain side effects which modify the data within the object during its execution. If a method with side effects is called within the for loop on an object shared between multiple iterations, this may cause a race condition in parallel. Checks on method calls are made in the `SafetyChecks.checkMethodCallSafety(...)` method. If a method is static, it is assumed to be race condition. This is a simplification as a static method can exist which does not have side effects and therefore be safe, however the checks made on other methods allow for certain circumstances where a side effect method is still considered safe where a static method would not be. Because analysis of side effects in a method requires access to the source code, only methods that exist within the user's source code can be analysed. Methods that are called from either the standard Java libraries or other libraries that have been added to the project will produce a warning informing the user that the method cannot be analysed.

Analysing methods for side effects is very similar to the checks made against for loops to be parallelised. Share variables (which in this case are fields of the method's object) are found within the method and checks are made to ensure that they are not being used safely. As is the case with for loops, a `SharedDataDetector` class has been produced to find shared variables within method declarations (`MethodDeclarationSharedDataDetector`). This class slightly modifies the tree traversal of the data detector to ensure that it stops searching for variable declarations once it reached the top of the method node as this will mean the variable is shared.

As the same checks are being made on method declarations as are being checked on for loops, method calls embedded within the method declaration are checked for safety making the system recursive in nature.

Data-flow Analysis

Earlier in the Variable Safety section of this document, a piece of code was shown in which simply checking whether or not given variables were shared or not between iterations was not enough information to evaluate its safety:

```
1 Foo a = new Foo();
2
3 //Loop to be refactored
4 for(int i = 0; i < 100; i++) {
5     Foo b = a;
6
7     //Although this assignment is to the non-shared variable b
8     //It references the shared variable a and therefore can cause a race condition
9     b.bar = i;
10 }
```

The non-shared variable `b` is referencing the shared variable `a` and then editing the contents of `a` indirectly on line 9. In this case, data-flow analysis is required to work out that the data within variable `b` is shared among other iterations.

Backward Data-flow Analysis

A modified of the backwards data-flow analysis algorithm[?] is used to find the type of data being used within variables. From the position where the variable in question is used (line 9 in this case), the program works backwards to find the last assignment to this variable. Once found (on line 5), the data that is being assigned to the variable is checked to see whether or not it is shared data

or not. In this case, `b` is being assigned to the shared variable `a` meaning that the analysis can be stopped as it is now assured that the data contained within variable `b` is shared and that a race condition warning should be produced for the edit of the object on line 9.

Although in this example the question of whether or not the data was shared or not was resolved relatively simply, the analysis becomes more difficult for more complicated pieces of code.

Listing 6.1: Complex Dataflow

```
1 Foo a = new Object();
2
3 int main() {
4     Foo c = new Foo();
5
6     for(int i = 0; i < 100; i++) {
7         Foo b;
8
9         b = c;
10
11        if(i < 50) {
12            b = new Foo();
13        } else {
14            b = new Foo();
15        }
16
17        b.bar = 10;
18
19        b = getObject();
20
21        if(i < 50) {
22            b = new Foo();
23        }
24
25        b.bar = 25;
26    }
27 }
28
29 Foo getObject(int i) {
30
31     Foo d = new Foo();
32
33     if(i < 25) {
34         return d;
35     }
36
37     return a;
38 }
```

Listing 6.1 requires more complicated data-flow analysis in order to detect shared data use.

The first potential problem in this code arises at line 17. With the contents of variable of `b` being modified rather than written over, this is a potential race condition for non-shared variables. The assignment of `b` for this point needs to be found through data-flow analysis. Unlike the previous example however, the assignment to `b` for line 17 is ambiguous. There are multiple paths to line 18 flowing either through the if statement on line 11 or the else statement on line 13. Due to this ambiguous, it is treated as if the program flowed through either meaning the assignments that effect line 17 is both the assignment on line 12 and on line 14 and both are checked for shared data. In this case, both assignments are assigning newly instantiated objects to `b` meaning the data within the variable is safe for use in parallel. Although `b` was assigned the shared data `c` on line 9, this is found to be dead code by the analysis as both paths on the line 11 if statement have their own

assignments to `b`. The analysis will not go back further than is necessary to find assignments in this case.

As with if statements, where other statements create branching paths in the code (try blocks and switch statements), the algorithm will not go beyond the branching statement if all branches covered with assignments to the problem variable. This ensures that the shared data detection is as accurate as possible, not giving warnings for assignments that cannot actually effect the code. The next problem code appears at line 21 with another modification of variable `b`. This is another case where the assignment of `b` for this point is ambiguous. In this case the two possible assignments is `b = getObject()` on line 19 and `b = new Foo()` on line 22.

Although the assignment on line 22 is contained within an if statement, it is unlike the line 11 if statement in that the analysis much continue up the source code to find other assignments as not all paths in the if statement have an assignment to `b` (if `i` is greater than 49 then the if statement code would not be executed). This means the `b` must go back to the point where the contents of `b` can concretely be assured (line 19).

Line 19 assigns the return value of the method `getObject(...)` to `b`. Analysis of methods is also possible using this algorithm. The `getObject(...)` method has its own ambiguity due to its multiple return values (line 34 and line 37) so both return statements are checked for shared data. On line 34 `d` is being returned which as found in line 31 is both a non-shared variables and contains non-shared data within it so this causes no problems. The return value on line 37 references the shared variable `a`. The unsafe nature of the line 37 return statement overrides the safe statement on line 34. This makes the assignment on line 19 unsafe and therefore the code on line 25 unsafe also (The unsafe assignment on line 19 overrides the safe line 22 assignment).

6.2.3 Code Generation

Naturally code generation is an important element of refactoring. There are two main elements to code generation within the Java Parallel Refactorer, implementing the parallel pattern as well as the code to replace the for loop that was selected for parallelising. There is also minor house keeping code to be generated to ensure that the the parallel library being used is correctly shut down once the program has finished.

Parallel Pattern Implementation

The Java Parallel Refactorer takes an algorithmic skeleton[?] approach to parallelising code. A parallel library that is separate from the main refactorer code is used which implements the algorithm used for parallelising in this project, a parallel for loop. This is implemented in the class `ParallelForFarmTask`. To implement this algorithm the refactorer generates a class that extends this one complete with code that specialises it to fit to the user's code. To implement the parallel for algorithm, the code generator needs to override the `operation(rangeStart, rangeEnd)` method as well as create a custom constructor for passing in shared variables that are to be used within the for loop.

Listing 6.2: Generated Implementation of Parallel For

```
1 private static class Closure extends ParallelForFarmTask {
2
3     public int fibNumber;
4
5     public Closure(int rangeStart, int rangeEnd, int noOfChunks, int fibNumber) {
6         super(rangeStart, rangeEnd, noOfChunks);
7         this.fibNumber = fibNumber;
```

```

8     }
9
10    @Override()
11    public void operation(int rangeStart, int rangeEnd) {
12        for (int i = rangeStart; i < rangeEnd; i++) {
13            fibonacciNumber(fibNumber);
14        }
15    }
16 }

```

Listing 6.2 shows an example of a class generated by the refactorer for parallelisation from the original loop:

Listing 6.3: Original Code

```

1  for(int i = 0; i < forLoopIterations; i++) {
2      fibonacciNumber(fibNumber);
3  }

```

Some obvious similarities can be seen between the original for loop and the operation method in the generated class. The main difference between these two pieces of code is the iterator initialisation and the compare expressions for breaking out of the for loop. In the parallel for algorithm, chunks of the for loop are run at the same time (One thread may run iterations 0 to 100 as the next thread runs from iterations 101 to 200). The generated code uses the parameters `rangeStart` and `rangeEnd` for the range to iterate over inside the operation method to allow for chunking the for loop into smaller sections rather than the original loop which iterates over the entire start (0) to finish (`forLoopIterations`) sequentially.

The code on line 2 in the original code is copied over to line 13 in the generated code. This is the case for all refactorings, the body of the loop to be refactored is copied over to the body of the loop within the operation method as the code to be executed for each iteration of the loop.

The code in the original loop calls a local method `fibonacciNumber`. To ensure that methods in the original class are callable from inside the newly generated class, the generated class is embedded into the class from which the original for loop came from, embedded classes are able to access the methods of their parent class in Java.

Also present in the original for loop code is the reference to the variable `fibNumber`. This variable has not been declared within the for loop and is therefore an example of a shared variable. All shared variables within a for loop have fields within the new class generated for them as equivalents for the new parallel loop to access (As seen on line 3 in listing 6.2 for `fibNumber`). The constructor generated on line 5 sets the clone `fibNumber` field from its parameter which is used in the code which replaces the original loop.

Replacement Code

Listing 6.4: Replacement Code For Listing 6.3

```

1  {
2      Closure returnData = ParallelExecutor.executeParallel(new Closure(0,
3          forLoopIterations, 20, fibNumber), 4);
4      fibNumber = returnData.fibNumber;
5  };

```

The original loop from Listing 6.3 has been replaced here with a call to a parallel algorithm runner method `ParallelExecutor.executeParallel`. In the Java Parallel Library, this method is used to execute parallel algorithms based on the object that is used in its first parameter (The

farmTask parameter). The next parameter represents the number of threads that the parallel algorithm is to be used on. Normally the number of threads would match the number of cores that are present on the target machine however it can be set to however many threads the user wishes in the refactoring options.

Listing 6.5: Java Parallel Refactorer Output for Refactoring Example

```
1 Parallel Refactorer 0.1
2 Found For Loops:
3 for loop at line 12 column 9
4 for loop at line 15 column 13
5 for loop at line 27 column 9
6 for loop at line 30 column 13
7
8 Choose for loop (line): 30
9 Performing safety checks.
10 WARNING Line 31 Column 17 in class App: The method 'fibonacciNumber(fibNumber)' is
    static, this may cause a race condition.
11 WARNING Line 30 Column 28: Iterator variable is compared to a non-constant value.
    Note that the parallel for loop will run the same number of iterations regardless
    of changes to this value during its execution
12 Safety checks complete.
13 Give name for newly created class: Closure
14 How many chunks should the for loop be split into? 20
15 How many threads should the parallel program run on? 4
```

In this example the for loop is being chunked into 20 tasks to be run across 4 threads as chosen by the user on lines 14 and 15 in the refactorer output. This is reflected in line 2 of the replacement code with the number of threads parameter being set to 4. The chunks variable forms part of the parameters to the parallel for algorithm and is therefore fed in as the third parameter of the implemented parallel for loop object Closure (Also named by the user on line 13 of the refactorer output). Notable also to the Closure constructor is the first two parameters 0 and forLoopIterations. These are the start and finish iterations of the original for loop informing the parallel for loop algorithm of the range to iterate over. The final parameter of the constructor is shared data to be used within the loop which will be discussed later. As the library takes the algorithmic skeleton approach to design, the real implementation of parallelism is obfuscated from the user within the ParallelExecutor class and super classes of the generated Closure class.

Shared Variables

Unlike languages such as C++ and C, Java does not support pointers. This makes the use of shared variables within the generated parallel for loop more challenging. The shared variable being used within this example is fibNumber which is a variable, of int type, local to the method which the original for loop was defined in. In order to run the parallel for loop, the program needs to move out of this original method into a new variable environment which does not contain this local variable. In order to maintain access to the fibNumber variable, it is copied over into the constructor of the implemented parallel for class Closure. The Closure class contains an equivalent variable that represents this shared variable during the execution of the parallel loop (The field fibNumber on line 3 of Listing 6.2 in this case). With the variable now available with its initial data input, the parallel loop can execute correctly. Having finished the loop any changes to the fibNumber variable made will be made to the fibNumber field within the Closure object. Such changes will not be reflected in the original fibNumber local variable however. Where C or C++ would be able to use a pointer to directly modify the value of the original variable, Java does not have such functionality. This means that shared variables that have fields within the generated class to represent them, must

have the contents of the field copied back into them after the parallel loop finishes.

The replacement code contains a variable `returnData` which is of the same type as the class that is generated by the refactorer. Where as the first parameter of the `executeParallel` method represents the starting environment of the loop to be executed, the `returnData` represents the ending environment. This means that the contents of shared variables after the execution of the original loop reflect the contents of the shared variable representation fields in the `returnData` object. All that is left to do is copy back the contents of these fields back to the original variables that they represent. In this example the only variable that is shared among all iterations is the `fibNumber`. As such, the corresponding `fibNumber` field within the `returnData` object is copied back into it on line 3 of the replacement code 6.4. These copy backs are used by the refactorer for all shared variables of all types. Copying back the values of variables is not strictly necessary for all variable types but the compromise of copying back all variables was required due to the technologies being used. In Java objects may be referenced by multiple variables and when modified effect the contents of both variables. This means that if a shared variable is of an object type and the corresponding field within the parallel for class simply modifies the contents of the object in some way, the copy back after loop execution will not be required as both variables are already referencing the same object (causing the copy back to be useless overhead within the program). There are situations however, where the field inside the generated class can become dereferenced from its corresponding original variable, such as assigning a new object to it. Such an assignment will not be reflected in the original variable and as such the copy back is required. Finding the situations where the copy back is not required is a complex process of data-flow analysis that is beyond the scope of this project. One improvement to this system that could be relatively easily implemented is the not using variable copies for fields in the original class. As mentioned in the Parallel Pattern Implementation section. The generated class is actually embedded into the class that the original loop came from. This means that although the generated class does not have access to share variables that are local to the original method, it does have access to fields within the original class. With direct access to the original fields, the use of copy backs or even corresponding fields in the generated class are not required. This could remove quite considerable overhead if the number of shared variables that are fields used in the loop is high.

As it stands with the current implementation however, it is up to the user to remove copy backs on variables where it is not required if they want to reduce overhead.

A final feature of the replacement code is that it is surrounded by curly braces to make a block statement in the AST. This serves more than an aesthetic purpose. By making this section of code its own block, variables declared within the block are local to only the block. The `returnData` variable defined in this block is not accessible afterwards. This means that if the user wants to parallelise multiple loops in the same method, the `returnData` identifier can be reused without Java throwing the variable redefined compiler error.

6.3 Parallel Library

In producing parallel code for the user, the refactorer uses a library to implement the parallel code. This library provides an algorithmic skeleton for the parallel farm pattern as well as a class used for execution of the given algorithm (the `ParallelExecutor` class).

6.3.1 Parallel For Pattern

the parallel for pattern is implemented in this library in the `ParallelForFarmTask9.1` class. A user of the library (In this case the refactorer program) needs to simply implement this class (As seen in Listing 6.2), by providing the operation method and any shared variables that are required, and the parallel library will do the rest of the work in transforming the sequential loop into parallel.

FarmTask class

The `ParallelForFarmTask` itself extends the `FarmTask` class. The `FarmTask` is designed to be the super class of all parallel algorithms that are based on the task farm parallel algorithm.

A task farm takes a job to be completed and splits it up into smaller pieces, called tasks, to be given to multiple workers which execute in parallel[?]. In this case, the workers are the threads in the thread allocator (which is discussed in more detail later) and the tasks are separate executions of an instantiated `FarmTask` object. The `FarmTask` implements the `Runnable` interface from the standard Java library which allows it to be run on standard Java threads giving the program its parallelism. To implement `Runnable` the `run()` method must be implemented.

Listing 6.6: FarmTask run method

```
1  @Override
2  public void run() {
3      Object [] inputValues;
4      synchronized(this) {
5          inputValues = readInputData();
6      }
7      dataInputUsed = true;
8      synchronized(dataInputMonitor) {
9          dataInputMonitor.notifyAll();
10     }
11     operation(inputValues);
12 }
```

Rather than tasks within a farm being instantiated as separate objects, each run of the `FarmTask` is given separate input values in which it will run its operation method (implemented by more specific algorithms as well as the library user). It is designed this way in order to reduce the overhead of instantiating a potentially very high number of objects for every task in a farm. There are downsides however to such an approach. With Java threads, it is not possible to pass information onto specific threads. The thread will simply execute the `run()` method of the object so to pass input values into individual threads, the data needs to be passed in on fields within the object. This poses a problem as fields within the `FarmTask` object are shared with all other tasks that are executing in parallel. So for the purposes of feeding in input data, the `run()` method in the `FarmTask` is designed to safely access the input data present in the class fields. To do this, the class makes the assumption that the system does not attempt to create new tasks for the farm until the previously allocated task picks up the input data present in the `FarmTask` object's fields. This is done on line 5 of the run method where the input data is copied into the variable local to the individual task, `inputValues`. Lines 7 to 10 are then used to notify the task creator that the next task can now be safely fed into the task allocator without it effecting the currently running tasks, and line 11 starts the actual task with a call to the `operation` method.

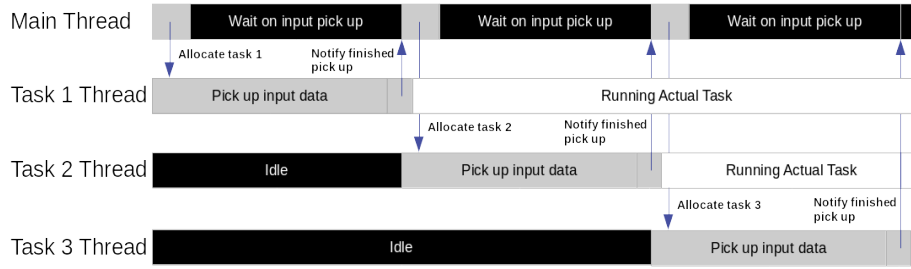


Figure 6.3: Data Input Overhead

As can be seen in Figure 6.3, this synchronising between threads to allow them to pick up input data leaves quite a significant amount of overhead in the system with the farm tasks having exaggerated staggered starts. As will be shown later in evaluation, large overhead in the parallelisation system have caused significant issues in regards to the effectiveness of the speed ups produced in this project.

This use of staggered starts on farm tasks for getting input data was originally intended as a compromise to not require the system to create new objects for every task within a farm job. Although it is still present within the current system, it is in fact no longer required. This system was developed at a time when the thread allocator ran each `FarmTask` directly on to standard Java threads for which this issue of not being able to pass in data directly was a problem. The current thread allocation system however runs tasks on a custom `TaskThread`. With little modification, the `TaskThread` would be able to have input data passed directly to it and thus this overhead produced by the compromise could be removed relatively easily.

ParallelForFarmTask class

As a subclass of `FarmTask`, the `ParallelForFarmTask 9.1` class is designed as the implementation of the parallel for loop (a form of farm). In implementing this algorithm, it must override the `allocateTasks` method to inform which tasks should be completed within the task farm. The tasks to be completed in a parallel for loop are chunks of iterations within the loop. The number of chunks to split the for loop into is set in the class' constructor's `noOfChunks` variable.

6.3.2 Thread Allocator

To give parallelism, tasks produced by task farm need to be run on a number of threads which run at the same time. The `ThreadAllocator` class is responsible for managing these multiple threads as well as queuing up tasks to be executed on them.

The user is able to set the number of threads that are available in the thread allocator and this would typically set to the number of cores available on the target machine to ensure context switching between multiple threads is kept to a minimum. In order to provide tasks to be run, the `execute` method is used. This method will put the given task onto the outstanding tasks queue until there is a free thread for it to execute on.

The process of assigning tasks to threads is actually handled by a thread which is separate to the thread in which the `execute` method was called. The task assignment thread is defined by the run code within the `TaskPlanter` class. This thread continuously monitors the outstanding task list as well as threads to immediately assign tasks to threads once they become available. The threads themselves run their own code defined by the `run` method of the `TaskThread` class.

A standard Java thread executes the `run` method of a `Runnable` class and then stops. The `TaskThread` however runs a task, and then notifies that it is free for a new task to be run on it. This ensures that new threads do not need to continuously be created for every new task and instead the currently running threads can be used.

The number of threads used in the thread allocator is limited to the number set by the user upon calling the `executeParallel` method. As mentioned previously, the system is designed this way to ensure that the number of threads can match the number of cores on a machine ensuring that context switching is kept to a minimum. This however can produce a deadlocking problem. Typically when tasks in a farm outnumber the threads in the thread allocator, they will be queued until the tasks that are currently executing on the threads finish and free up a thread. This can cause problems in systems where the tasks that are currently executing in the threads themselves call their own tasks.

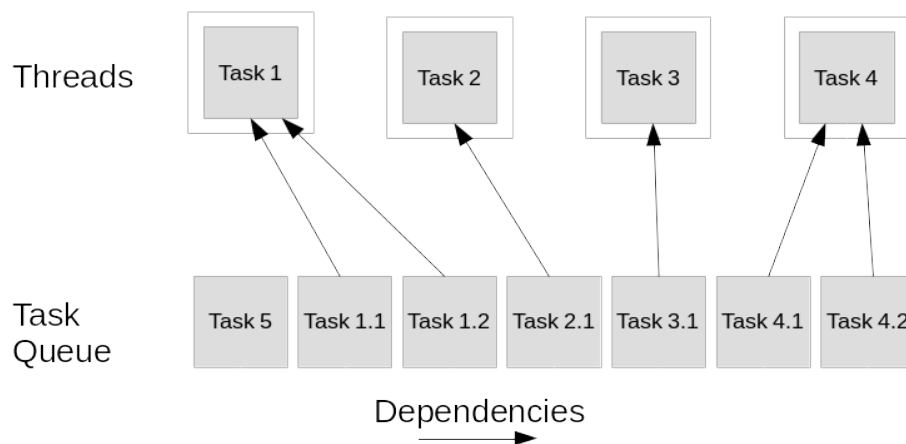


Figure 6.4: Dependencies from active tasks in threads are blocked by inactive tasks in queue

Dependencies on tasks in the outstanding task queue by tasks that are currently executing causes a deadlock situation where the parent tasks are blocking the ability of the tasks they depend on from executing.

Deadlock Prevention

Deadlocks have the capability to bring a program completely to a halt. This parallel library however, is able to detect deadlock situations such as the one previously described and resolve them as required. It should be noted that the deadlock resolver is just used as last resort to stop programs from completely halting and if a program is needing to use the resolver to run then any of the speed up that may be gained in parallelism is probably being lost in this situation.

Deadlock Detection

In this kind of deadlock situation, all of the threads in the thread allocator are stuck on their respective tasks and need these sub-tasks to complete before they can continue.

The detection of this problem is handled by the `TaskPlanter` class. The `TaskPlanter` manages the assignment of tasks to threads. This class will identify threads that are free to take up new

tasks and then assigns one from the outstanding task queue. If the `TaskPlanter` find that there are no free threads however, it performs a check to see if there are any sub-tasks present in the task queue that currently executing tasks depend upon. In order to check if a task in the queue is a dependency of a currently executing task, the thread in which the `executeParallel` method for this tasks was called from must be known. All `FarmTask??` hold a field, `ownedThread`, which identifies which thread created it and placed it in the thread allocator. The `TaskPlanter` checks this information to see if any of the tasks in the queue have an owning thread which comes from the thread allocator, if it does, this would mean that whatever task is running on that thread at the moment is dependent on this task in the queue. This could lead to a deadlock situation and therefore needs to be immediately rectified.

Deadlock Resolving

Similar to detection, the resolving of deadlocks is also handled by the `TaskPlanter` class. When confronted with a deadlock, the deadlock resolver will temporarily create a new thread which the sub-tasks can run on before returning to the initial pool of threads to let the original task finish. This new temporary thread is called a sub-thread (of which the parent of the sub-thread is the thread which originally called the sub-tasks for execution). This is part of the reason why deadlock resolving should be avoided, There is an overhead cost of starting up a new thread for the purposes of running these tasks that could be avoided if the number of initial threads in the thread allocator is high enough. The other reason for avoiding deadlock resolving will become clear later.

With the sub-thread ready, the `TaskPlanter` will place any sub-tasks in the outstanding task queue which was created by a parent thread on to the sub-thread for execution. A maximum of one sub-thread is created for every thread that has created sub-tasks. This is because the number of threads in the thread allocator is expected to be around the same as the number of cores that are present on the target machine. Creating more threads than this would mean context switching between the threads would occur causing additional overhead and no gain in speed through parallelism. Because only one sub-thread is being created for these sub-tasks, these tasks will in fact run sequentially one after another until all sub-tasks are completed. This is the other reason why deadlock resolving should be kept to a minimum, there is no actual parallelism during the process and instead the program is slowed down.

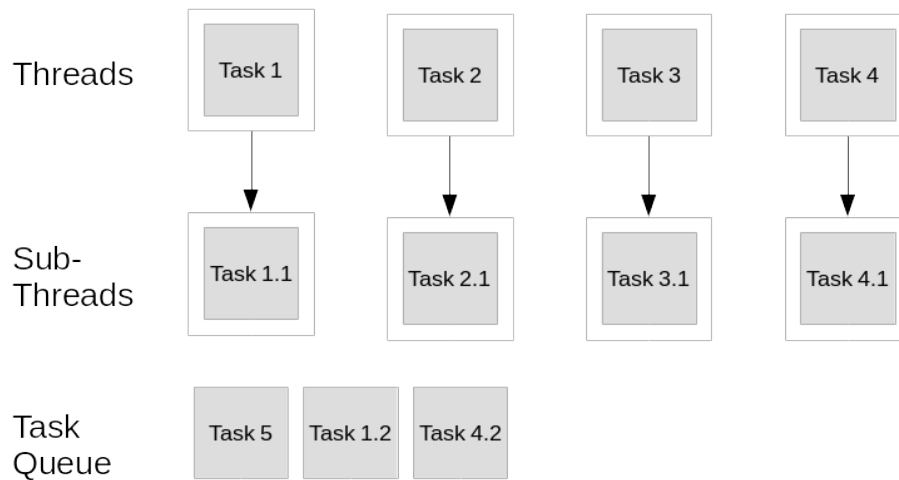


Figure 6.5: Sub-tasks execute on their respective sub-threads until a main thread becomes free to continue normal execution

The deadlock resolver is also capable of fixing dependency deadlocks that are multiple layers deep, if a sub-task is created from a sub-thread then this itself will start its own subthread to ensure deadlocking does not occur.

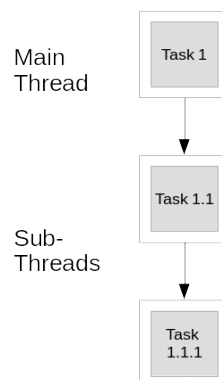


Figure 6.6: Sub-tasks of sub-tasks are run down the chain of sub-threads

7. Evaluation

The success of this project is mixed. Although solid progress in the refactorer program has been made, the usefulness of the system as a whole falters somewhat due to problems with the functionality of the parallel library that is being used to create the parallelism within the system.

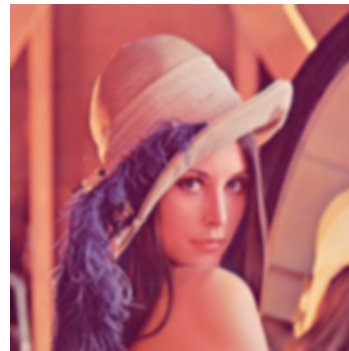
7.1 Speed Ups

An important element of any system that is designed to introduce parallelism to a program is of course, the extend of which it is capable of speeding up the program's processing. To test the speed up of the parallel library being used in this project I made tests on parallelising a sequential image convolution program[?]. Image convolution[?] is an ideal candidate for parallelism as each pixel in the image is independent of all other pixels being written to during the process. When applying parallelism using this system, the image was effectively split up into smaller chunks with the sub-images processing pixels in parallel.

Tests involved running a 5 x 5 Gaussian blur[?] on images of varying size while measuring the time taken for programs with different parallel and sequential set ups to complete the task.



(a) Input



(b) Image Convolution Output

Figure 7.1: Test Image for Image Convolution

Tests were performed on three versions of the image convolution program. Firstly, for assessing a baseline speed, the sequential version of the program was executed. The parallel library implemented in this project went through several design iterations before the currently used system was implemented. The next version of the image convolution program comes from an old implementation of the parallel library and is being used to check how the performance of the parallel library

has progressed (or regressed) over the course of its development (Further information on this old implementation can be found in appendices). Lastly, the refactored result of the image convolution program is tested to find the speed improvements made under the current system. All versions of the program were run several times to ensure background work on test machines did not adversely effect the test results to any great extend.

7.1.1 Small Scale Test

The first test machine is a 4 core MacBook Pro laptop. Tests were run on a range of image resolutions to see how extensive processing effects the capability of the parallelism.

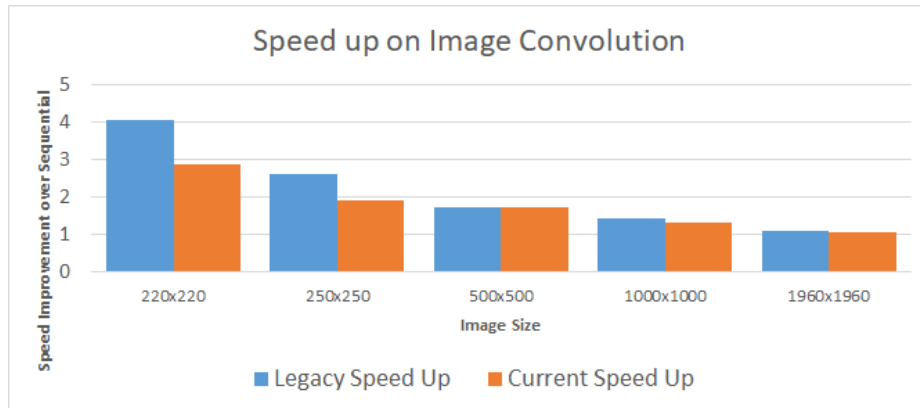


Figure 7.2: Speed Tests on Multiple Image Sizes on 4 Core Laptop

As can be seen in Figure 7.2, Although the speed gains on smaller images were quite significant for the current system with a 2.87x speed up on the 220 x 220 image, speeds of the current system were consistently slower than the old parallelism implementation and speed improvements degrades as the resolution of the image grew.

The reduction in performance compared to the legacy parallel system, although unfortunate, is not entirely unsurprising due to many of the more complicated features involved in the new parallel system such as the ability to use more tasks within a task farm than threads in the system, as well as the more complicated deadlock resolving system. Less formal tests on these parallel systems found that once a task farm uses around 200 to 500 tasks, the performance of the legacy parallel system started to lag behind that of the current.

The reductions of performance on larger images is more problematic. As a system that is designed to speed up sequential programs its inability to effectively provide speed up on more process intensive programs means that the programs that should be benefiting from this technology the most are not accommodated. The reason for such slow down on larger images is unclear and as is shown in the large scale test, does not appear to be a problem universal to all target machines.

7.1.2 Large Scale Test

The second test machine is a 56 core server provided by the University of St Andrews called Cor-ryvreckan.

For this machine, due to to extensive number of cpu cores, tests were performed varying the number of cores used as well as the image resolutions.

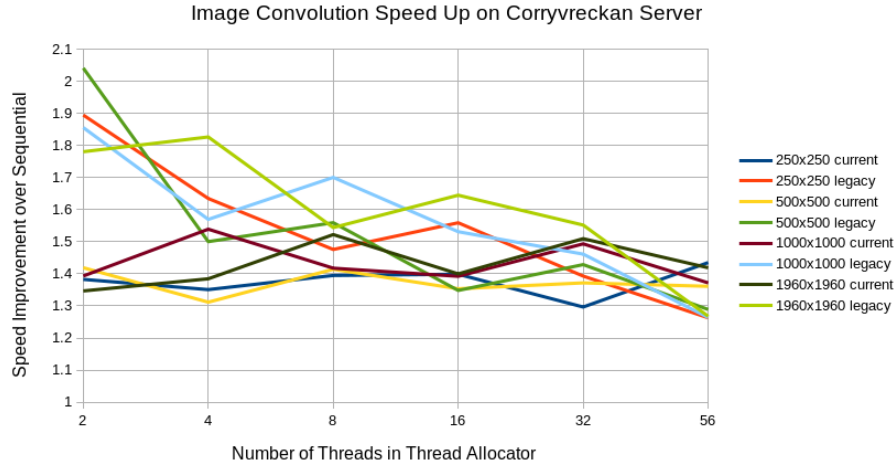


Figure 7.3: Speed Tests on Multiple Image Sizes for up to 56 Threads

As can be seen in Figure 7.3 finding a consistent pattern within the results on the large scale test is more difficult. There does appear to be a somewhat downward trend in performance as the number of threads in the thread allocator is increased, especially within the legacy parallel system. The size of the image in this case does not appear to negatively effect the performance of the program. Despite the slight increases in speed (on average 1.4x speed up using the parallel system), overall the results show that the parallel library system does have problems it needs to overcome to be a useful tool for developers. With effectively no improvement in speeds regardless of the number of cores used in the program, this suggests that thread contention has become an issue within the library. As new threads are introduced to the system, any gain in speed from running in parallel is lost again as the thread management attempts to use shared data with yet another thread.

As the standard Java threads library does not give the programmer low level access to how threads are being executed, it may also be possible that despite the high number of cores available on the server, the program is not actually using them and instead running these multiple threads on a smaller number of cores with context switching. In this case, an increase in the number of threads would not improve overall performance.

7.2 Safety Detection Capability

The potential safety problems created in transforming a sequential program into a parallel one are so numerous that creating a tool for detecting all such problems is not really possible. All a safety checking system can hope to do is to find a reasonable subset of safety errors, without producing too many false positives on areas of code that is actually safe.

The safety detection system used in this refactorer focuses mainly on detection of race conditions within the user's code. The functionality of the safety checking is somewhat hindered by problems with the symbol solver library being used in this project. Although the symbol solver and parser is capable of finding methods and generating ASTs of for other files within the user's project, symbol solving for variables within these files does not seem to work. With the limited documentation available on the Java Symbol Solver library[?], it is difficult to fix such a problem if indeed this is not actually a bug within the library itself. This problems makes it nearly impossible for methods

called from outside classes to be analysed, which unfortunately, is an important element to many object oriented Java projects.

Due to this problem being present in the final version of the refactorer, I have opted against performing any evaluative tests on code samples. As most code samples need to be able to evaluate these methods across different files, they cannot give a reasonable result on the actual capability of the safety checker. To produce my own code samples for testing would likely bias the result.

In tests I have done where method evaluation has been successful, there is one notable problem that does become apparent on some programs. Where recursion takes place within a method, the system used to check methods for side effects within the refactorer falls into an infinite loop. Further work on the refactorer should be able to fix this relatively easily by applying dynamic programming on side effects checking. Overall, despite this setback, I believe that a solid groundwork has been laid for the safety checker. The system can effectively give warnings for:

- Methods that exist within the refactored file,
- Shared data usage whether this be directly from shared variables or indirectly from non-shared variables that reference such data,
- Accessing elements in arrays from multiple iterations,

Among other warning unrelated to race conditions. Once problems with the symbol solver have been fixed, and minor changes in the code to allow the evaluation of static methods, I see no reason to believe that the safety checker used in this refactorer, will cover a relatively large subsection of problems related to parallelising programs.

Evaluating on the basis of comparing the end result software to the objectives given in the project's proposal (and mirrored in the objectives section of this document), I believe a success has been made in realising the original vision. All primary objectives have been met to varying degrees:

- Objective P1 has been implemented in full.
- The functionality of objective P2 cannot be fully realised. The user is able to directly pick which file they would like to refactor however the refactorer cannot fully understand the code within the wider context of the project due to what is assumed to be a bug in the Java Symbol Solver.
- Unfortunately the detection of data streams has not been possible due to capability issues within the Java Parser/Symbol Solver, however I believe safety checking is functioning within the software more effectively than originally envisioned in objective P3. The data-flow analysis technique being used within the safety checking as well as checks being made on method calls bring this feature beyond the "limited" functionality which was originally intended.
- Objective P4 has been mostly implemented. Code is successfully refactored with most of the user's code being lexically preserved. The code that needs to be copied from the original loop to the newly generated class however does not maintain its lexical format. This is an area that could be improved upon further development
- The parallel library developed within this project more than implements the P5 objective. The user is able to set the number of threads they wish to execute their task farm on as well as set other options on how their parallel for loop runs, such as setting the number of chunks to split the loop into. The parallel library also implements a deadlock detection and resolving system that was not specified in the original objectives.

The secondary objectives of this project have not been met. During development I decided that improving the functionality of currently implemented features (with deadlock resolving within the parallel library and data-flow analysis within the safety checker) was preferable to introducing these new features which may not have been able to meet the same level of quality as the rest of the project.

8. Conclusion

In this project I have developed a refactoring tool capable of converting a developers sequential program into a parallel equivalent. The refactorer contains powerful safety checking capability to find race conditions within the developer's code through the use of data-flow analysis techniques and general semantic analysis. To produce the parallelism used within refactored programs, a parallel algorithmic skeleton library has been produced. This library makes the implementation of the parallel for loop algorithm very simple and features a anti-deadlocking system to reduce the chance of parallel programs falling into a situation where it continues to run forever.

A modular design philosophy has been employed across the project with the code base being split into two distinct parts separating the refactor from the parallel library it implements. These separate sections of the code similarly make use of this modular design. Future development on this project benefits from the modularity ensuring that newly implemented features and modifications of current features can be introduced with minimal chance of breaking the current system.

The functionality of the software produces is somewhat compromised through large levels of overhead within the parallel library as well as problems with making use of the Java Symbol Solver library limiting the effectiveness of method evaluation within the race condition safety checker.

8.1 Future Work

Various improvements in the capability of this refactoring system can be made in both the short term as well as considerations on a more long term basis.

- The obsolete process for providing input data to tasks in the parallel library can be replaced reducing the overhead of the parallelism system. This improvement can be introduced quickly with small modifications to the `FarmTask` and parallel algorithm task allocation methods (implemented `allocateTasks` methods).
- Dynamic programming can be used in analysing methods to ensure that recursive methods can be served without the refactorer falling into an infinite loop. This feature is relatively involved needing quite considerable modification of the `SafetyChecks` class.
- The current limitation of evaluation of static methods can be removed almost immediately giving the safety checking system greater functionality. A version of the refactorer which removed this limitation during the development of this project however it was removed from the final version due to there not being enough time to test the feature. With greater development time on the project, adequate testing could be done to introduce this feature very quickly.
- Once a work around for the problems in the Java Symbol Solver regarding analysing source files unrelated to the file being refactored can be found, the effectiveness of the safety checker

can be drastically improved. The checks for race conditions across object oriented projects can be made making the refactorer an effective tool for the typical Java developer.

- Considering the dissapointing speed ups produced by the provided the parallel library, it may be preferable to change the target library of the refactorer to a third party's more established algorithmic skeleton library. This can improve parallelism performance for programs being refactored allowing them to take advantage of multi of many core processing power. The ease in which the refactorer's code generator means that replacing the library being used for parallelism should be possible to implement in a relatively short period of time.

9. Appendix

9.1 Testing

9.2 User Manual

9.3 Code Listings

Listing 9.1: ParallelForFarmTask Class Implementing the Parallel For Algorithm

```
1 package parallel;
2
3 import parallel.threadallocation.ThreadAllocator;
4
5 /**
6  * Skeleton code to be extended to run a for loop in parallel.
7  * @author michaellynch
8  *
9  */
10 public abstract class ParallelForFarmTask extends FarmTask {
11     private int rangeStart;
12     private int rangeEnd;
13
14     private int totalLoopRangeStart;
15     private int totalLoopRangeEnd;
16     private int noOfChunks;
17
18     @Override
19     protected void operation(Object[] inputValues) {
20         operation((int)inputValues[0], (int)inputValues[1]);
21     }
22
23     /**
24      * Creates a new parallel for loop with start iteration rangeStart and end
25      * iteration rangeEnd.
26      * The loop is split into the given number of chunks to be executed as separate
27      * tasks in the parallel farm.
28      * @param rangeStart Iteration start point.
29      * @param rangeEnd Iteration end point.
30      * @param noOfChunks The number of chunks to split the for loop into which will
31      * run in parallel.
32      */
33     public ParallelForFarmTask(int rangeStart, int rangeEnd, int noOfChunks) {
34         totalLoopRangeStart = rangeStart;
35         totalLoopRangeEnd = rangeEnd;
36     }
37 }
```

```

33         this.noOfChunks = noOfChunks;
34     }
35
36     /**
37     * Overridden with for loop operation from rangeStart to rangeEnd
38     * @param rangeStart Start of loop iterations for this farm
39     * @param rangeEnd End of loop iterations for this farm
40     */
41     protected abstract void operation(int rangeStart, int rangeEnd);
42
43     @Override
44     protected Object[] readInputData() {
45         Object[] inputData = new Object[2];
46         inputData[0] = rangeStart;
47         inputData[1] = rangeEnd;
48
49         return inputData;
50     }
51
52     /**
53     * Set the start and end ranges of the next farm to be started.
54     * @param rangeStart Start of loop iterations for the next farm
55     * @param rangeEnd End of loop iterations for the next farm
56     */
57     public void setRange(int rangeStart, int rangeEnd) {
58         this.rangeStart = rangeStart;
59         this.rangeEnd = rangeEnd;
60         dataInputUsed = false;
61     }
62
63     @Override
64     protected void allocateTasks(ThreadAllocator ta) {
65         int total = totalLoopRangeEnd - totalLoopRangeStart;
66         int range = total / noOfChunks;
67         int leftover = total % noOfChunks;
68
69         for(int i = 0; i < noOfChunks; i++) {
70             if(i == noOfChunks - 1) {
71                 setRange(totalLoopRangeStart + i*range, totalLoopRangeStart + ((i+1)*
72 range) + leftover);
73             } else {
74                 setRange(totalLoopRangeStart + i*range, totalLoopRangeStart + ((i+1)*
75 range));
76             }
77
78             ta.execute(this);
79
80             synchronized(getDataInputMonitor()) {
81                 while(!isDataInputUsed()) {
82                     try {
83                         getDataInputMonitor().wait();
84                     } catch (InterruptedException e) {
85                         e.printStackTrace();
86                     }
87                 }
88             }
89         }
90     }

```

Listing 9.2: FarmTask Class, The base class for all task farm algorithms

```

1 package parallel;
2
3 import parallel.threadallocation.ThreadAllocator;
4
5 /**
6  * A superclass to algorithms that implement the parallel farm algorithm.
7  * @author michaellynch
8  *
9  */
10 public abstract class FarmTask implements Runnable {
11
12     private Thread ownedThread;
13     protected volatile boolean dataInputUsed;
14     private Object dataInputMonitor = new Object();
15
16     /**
17      * operation to be performed in each farm
18      * @param inputValues the unshared objects for each farm
19      */
20     protected abstract void operation(Object [] inputValues);
21
22     /**
23      * Used to place tasks onto the ThreadAllocator in the format required for this
24      * kind of farm operation.
25      * @param ta the ThreadAllocator in which the farm's tasks are to be run on.
26      */
27     protected abstract void allocateTasks(ThreadAllocator ta);
28
29     /**
30      * Create an array of objects that is non-shared data between each farm
31      * @return unshared data of each farm
32      */
33     protected abstract Object [] readInputData();
34
35     @Override
36     public void run() {
37         Object [] inputValues;
38         synchronized(this) {
39             inputValues = readInputData();
40         }
41         dataInputUsed = true;
42         synchronized(dataInputMonitor) {
43             dataInputMonitor.notifyAll();
44         }
45         operation(inputValues);
46     }
47
48     public Object getDataInputMonitor() {
49         return dataInputMonitor;
50     }
51
52     /**
53      * Checks to ensure the previously inputted data has been taken up by the latest
54      * farm started.
55      * @return true if the inputted has been read by the latest farm
56      */
57     public boolean isDataInputUsed() {
58         return dataInputUsed;
59     }
60 }

```



```
58
59     public FarmTask() {
60         ownedThread = null;
61     }
62
63     public void setOwnedThread() {
64         this.ownedThread = Thread.currentThread();
65     }
66
67     public Thread getOwnedThread() {
68         return ownedThread;
69     }
70 }
```

9.4 Ethics Form

Ethics form follows on next page.

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.

Tick one box

<input type="checkbox"/>
<input checked="" type="checkbox"/>
<input type="checkbox"/>

Staff Project

Postgraduate Project

Undergraduate Project

Title of project

Java Parallel Refactorer

Name of researcher(s)

Michael Lynch

Name of supervisor (for student research)

Professor Kevin Hammond and Dr Chris Brown

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted YES ☒ NO ☐

There are no ethical issues raised by this project

Signature Student or Researcher



Print Name

MICHAEL LYNCH

Date

07/06/2018

Signature Lead Researcher or Supervisor



Print Name

DR CHRIS BROWN

Date

07/06/2018 .

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

Computer Science Preliminary Ethics Self-Assessment Form

Research with human subjects

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

YES ☐ NO ☒

If YES, full ethics review required

For example:

Will you be surveying, observing or interviewing human subjects?

Will you be analysing secondary data that could significantly affect human subjects?

Does your research have the potential to have a significant negative effect on people in the study area?

Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

YES ☐ NO ☒

If YES, full ethics review required

For example:

Is there any potential that there could be physical harm for anyone involved in the research?

Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

Conflicts of interest

Do any conflicts of interest arise?

YES ☐ NO ☒

If YES, full ethics review required

For example:

Might research objectivity be compromised by sponsorship?

Might any issues of intellectual property or roles in research be raised?

Funding

Is your research funded externally?

YES ☐ NO ☒

If YES, does the funder appear on the 'currently automatically approved' list on the UTREC website?

YES ☐ NO ☒

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

Research with animals

Does your research involve the use of living animals?

YES ☐ NO ☒

If YES, your proposal must be referred to the University's Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages

<http://www.st-andrews.ac.uk/utrec/>

Bibliography

- [1] Danny Dig
A Refactoring Approach to Parallelism
<https://www.semanticscholar.org/paper/A-Refactoring-Approach-to-Parallelism-Dig/3336edd22e52ecdba602fce62ea29bc5a3624c3a>
Page 2: Improving Thread Safety
Page 4: Refactorings for Thread Safety

- [2] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner, Tino Breddin
Paraphrasing: Generating Parallel Programs Using Refactoring
<https://www.semanticscholar.org/paper/Paraphrasing%3A-Generating-Parallel-Programs-Using-Brown-Hammond/elb7e75d5039dd4e7058eb0a069816e90506cf99>

- [3] ParaFormance
paraformance.com

- [4] Mel O'Cinneide
Automated Application of Design Patterns: A Refactoring Approach
<http://csserver.ucd.ie/~meloc/thesis/thesis.pdf>

- [5] An automated refactoring approach to design pattern-based program transformations in Java programs
<https://ieeexplore.ieee.org/document/1183003/>

- [6] ARCHER
Parallel Programming Patterns
Page 13: Task farm https://www.archer.ac.uk/training/course-material/2016/07/intro_epcc/slides/L04_ParallelProgramming.pdf

- [7] Rosetta Code
Image Convolution in Java
https://rosettacode.org/wiki/Image_convolution#Java

- [8] Wikipedia
Kernel Image Processing
[https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing))
Gaussian blue 5 x 5

- [9] Wikipedia
Algorithmic Skeleton
https://en.wikipedia.org/wiki/Algorithmic_skeleton
- [10] Wikipedia
Backwards Data-flow Analysis
https://en.wikipedia.org/wiki/Data-flow_analysis#Backward_Analysis
- [11] The University of Texas Computer Science
Java Coding Samples
<https://www.cs.utexas.edu/~scottm/cs307/codingSamples.htm>
- [12] Java Parser and Java Symbol Solver
javaparser.org
- [13] TIOBE Index 2018
<https://www.toibe.com/tiobe-index>
- [14] Adam D. Barwell, Christopher Brown, Kevin Hammond
Finding parallel functional pearls: Automatic parallel
recursionscheme detection in Haskell functions via anti-unification
<https://www.sciencedirect.com/science/article/pii/S0167739X17315200>
- [15] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick, Archibald Elliott
Cost-Directed Refactoring for Parallel Erlang Programs
<https://link.springer.com/article/10.1007/s10766-013-0266-5>