Parallelism Refactoring Tools

Calum McCartan
cm4114@columbia.edu
Columbia University, New York, NY

Abstract

Parallel programming has the potential to significantly increase the performance of a vast number of applications, yet it is a technique which is often ignored - the reason being simply that parallel programming is difficult. Parallelism libraries and frameworks provide abstractions of parallel patterns to make this task much easier, however, no framework can inform a developer of where to introduce parallelism or of when it is safe to do so. In this paper I investigate the state of the art parallelism refactoring tools for solving these challenges, and perform an evaluation of one of these tools, ParaFormance[12].

1. Introduction

It has become increasingly clear that parallel programming is required if we want to continue to significantly increase the performance of our software. Processor performance has become limited by fundamental power and heat restrictions that have prevented the rate of performance increase that came before. As other papers have stated, the 'free performance lunch of processors is over'[9] and that 'today's sequential programs are tomorrow's legacy programs, unless they are retrofitted for parallelism'[4]. While tremendous performance improvements have been made possible through multi-core hardware combined with modern parallel programming techniques, the practice has yet to become commonplace.

Assisted parallelism refactoring tools have the potential to introduce a wider audience to making their software parallel, reducing the specialized knowledge that might otherwise be required. Even for parallelism experts, these tools may be able to guide and greatly streamline the refactoring process.

1.1 What will readers learn?

Throughout this paper, I hope to introduce the reader to several state of the art tools that have been developed for retrospectively introducing parallelism to software. In addition, I hope to explain the techniques used to implement these tools, such as parallel patterns, program slicing, and the criterion for safely refactoring a code fragment. I will also introduce some of the libraries and frameworks that are commonly used by these tools. The reader should gain an understanding of what challenges these tools can help overcome, and the extent to which the current state of the art tools manage this.

2. Background

It is common knowledge amongst software developers that parallel programming leads to deadlocks and race conditions, but actually spotting these hazards in a program can be exceedingly difficult. Race conditions and deadlocks are particularly challenging due to their non-deterministic nature, and can introduce very difficult to find and rare bugs. When an application only deadlocks 1% of the time or only on certain hardware, testing becomes extremely difficult.

Parallel programming also comes with many other complexities such as the challenge of thread and lock management, especially when trying to find the most optimal configuration. However, most of these problems can be greatly eased by the use of parallel pattern libraries and frameworks.

2.1 Libraries & Frameworks

As with any common area of software engineering, parallel programming has libraries and frameworks available to abstract over much of the complex/boilerplate code that programmers would otherwise have to write themselves. There are several frameworks for abstracting parallelism into higher level templates and patterns. Some example frameworks for C/C++ include Intel's Thread Building Blocks (TBB), OpenMP, and FastFlow. These libraries can manage threads for the programmer, and can appropriately optimise parameters such as the number of threads to create based on the hardware the program is running on.

The Generic Reusable Parallel Pattern Interface[11] (GrPPI) is an additional layer of abstraction between programmers and other parallelism frameworks such as OpenMP, TBB and FastFlow. After writing a program using GrPPI, it can easily be compiled to any one of the supported frameworks. This makes GrPPI a good choice to use for writing parallel code. One paper[3] takes advantage of this and introduces a tool for refactoring C++ loops into GrPPI code.

What libraries cannot help with however, is the knowledge of *where* to introduce parallelism. Creating new threads can be extremely slow relative to other code, and the time a program spends waiting for locks to be released can cause the overhead of parallel execution to outweigh the benefits. In addition, libraries generally cannot help with determining if code is correct and free of any errors such as race conditions. Although these libraries provide several useful parallel patterns, developers must have a good understanding of the subject in order to select the appropriate one. Refactoring tools often take advantage of the benefits of libraries while aiming to assist with some of the challenges that libraries alone cannot solve.

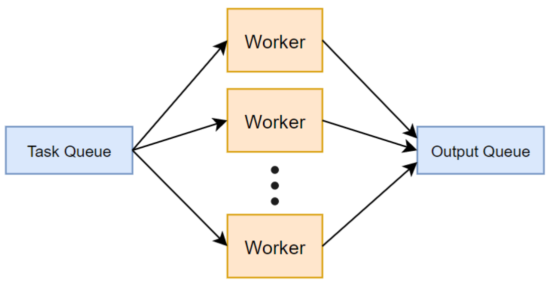
2.2 Parallel Patterns

When writing a parallel program, one or more parallel patterns can often be applied to introduce parallelism in different ways. Some patterns may be applicable to some programs but not others, and sometimes several different patterns may be combined to introduce several types of parallelism to a program.

When parallelizing a program, the problem must be broken down into tasks. These tasks can then be added to one or more input queues. Worker threads will then pull tasks from these queues, process the tasks and store the results. Some of the most common parallel patterns are farms, map/reduce and pipelines.

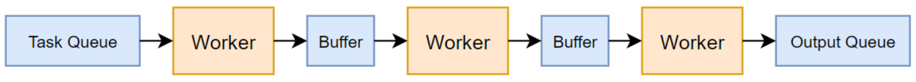
Task farms apply an operation to each task arriving from an input stream. Maps are similar to task farms, but instead take a fixed data structure as an input and perform the operation to each element. Reduce is a pattern which combines the tasks from a data structure into a single output, and is commonly combined with a map to form the map/reduce pattern.

Farm pattern



A pipeline is another example of a streaming pattern, which instead achieves parallelism when a sequence of operations need to be performed on each task where the output from one operation is the input to the next. An example use case would be for a series of loops which each perform some operation on a large array of objects.

Pipeline Pattern



When using a pipeline, the throughput of tasks is limited by the throughput of the slowest stage, as the preceding stages will eventually fill their buffers, while the following stages will eventually be starved with a lack of tasks. One way to solve this is to introduce a task farm at each stage of the pipeline, which allows more workers to be allocated at the slower stages, increasing the throughput of the system as a whole.

3. Tool Objectives

There are several factors to be taken into account when determining the value of a parallelism refactoring tool or technique. These are described here, and the extent to which various tools achieve them is referenced throughout the paper.

Parallelism Discovery: The ability for a tool to discover candidate code fragments for parallelisation is very valuable. It is especially useful when a developer is tasked with improving the performance of a large application made up of many thousands of lines of code, as a tool that can manage this has the potential to greatly reduce the amount of time a developer spends completing this task.

Safety Checking: When a potential site for parallelisation is found (either by a tool or a developer), it is very useful to know whether a refactoring will be safe. This is especially useful for less experienced developers who may have difficulty spotting a potential race condition or deadlock. Safe sites of parallelisation will tend to be free of side effects, and will not read or write global state. Even more useful is when a tool is capable of providing feedback to the user, recommending what parts of a code fragment are causing the problem.

Pattern recommendation: Even when a candidate code fragment is deemed safe, there may be several parallel patterns that could be applied, and so it is useful for a tool to recommend to a developer which pattern will likely be most effective in the given context.

Speedup estimation: The ability for a tool to determine whether or not a proposed refactoring will actually increase the performance of a program is very important - if the overhead is high relative to the actual runtime of the code, it may be best to use the sequential approach.

Runtime: A tool must be able to complete any analysis/profiling in a reasonable amount of time, otherwise it will be too impractical to actually be used. Note that the definition of a 'reasonable' amount of time may vary substantially depending on the tool. For example, you would expect an interactive IDE tool to complete a refactoring within a few seconds, but for a fully automatic tool, it may be acceptable to leave it running overnight.

Accessibility: For a tool to actually be used in practice, it must be easy to install and use. Many of the papers discussed here make themselves available as an IDE plugin (most commonly for Eclipse). The ability for a programmer to highlight a region of code and then click a 'make parallel' button is extremely valuable. Using a refactoring tool is often the easy part however, it can often be more difficult to actually know what you are looking for and then find and install it. For this reason it may be more important for collections of tools to be grouped together and made obtainable as a single package[4].

Readability: After refactoring a code fragment, it is important that the refactored code is both readable and as concise as possible - this is crucial for the maintainability of code. A common way to help tools achieve this is ensuring that any refactored code makes use of parallelism frameworks in order to minimise the amount of code produced. Parallel code is almost certain to be more verbose and difficult to understand than the sequential version, however it is still important that tools produce code which is somewhat comparable to that written by a competent human developer.

4. Parallelism Discovery

When seeking to improve the performance of an application via parallelism, much of the challenge is discovering *where* it is possible to parallelise and what is actually *worth* parallelising. The nature of some code fragments can mean that subtasks are too dependent on each other to be completed in parallel, or the overhead time of thread and lock management can outweigh the time saved by parallel execution. Any tool that is able to make recommendations on candidates for parallelisation are therefore valuable, even if they do not actually perform any code changes. Most tools focus on detecting common patterns[10] which can be translated into parallel patterns, while others check only for a specific structure, such as 'for' loops.

Erlang is a functional language and does not feature any loops, and so refactoring tools must instead search for features such as recursion. PaRTE[8] is a tool created by C. Brown et al. for discovering parallelism in Erlang programs. The tool searches for specific fragments of code such as recursive functions, list function calls, and list compresionshions. However, finding locations in code which repeat many iterations or take a long time is not sufficient by itself. The problem is that these sections of code can have side effects which would break parallel execution, such as mutating a global variable (This is a problem for finding parallelism in any language, not just Erlang). To tackle this, PaRTE records which variables a code fragment has read from, and also the variables to which the fragment writes data. When this information cannot be determined, a special flag is set to mark the uncertainty.

Using this information, PaRTE classifies expressions as 'black' 'grey' or 'white'. Grey are expressions which read global state, black alter the global state, while white does not require any read or write from global state. These classifications can then be used to recommend which code fragments are ideal for parallelisation. Some tools are able to use this information to advise the programmer on which statements are preventing parallelisation.

The strategy of using loop detection followed by analysis of how a code fragment reads and writes state is common amongst several parallelism discovery tools. However, there are differences, which usually occur due to the properties of the language the tool is used for. For example, tools used with C/C++ (such as PPAP[9]) must check for 'break' statements which can cause a loop to exit early, as such loops cannot be parallelized. Exceptions must also be taken into account as they can be used to terminate loops or alter state. Another common prerequisite for parallelism candidates is that they contain no RAW dependencies. Loops containing RAW (read after write) dependencies can cause problems when parallelized, as one thread may write to a variable before another thread writes to it, and so the reading thread gets the old (and incorrect) value.

4.1 Static vs Profiling Discovery Methods

Both static and profiling techniques can be used when detecting parallelism. The profiling approach involves running a program usually over many repetitions, and monitoring which code fragments take longest to execute in order to detect sites which may benefit from parallelism. While this has the major advantage of only discovering slow parts of the program where parallelism will actually be useful, it can be impractical for larger applications due to code coverage difficulties, and the amount of time taken to run a program repeatedly.

5. Parallelism Refactoring Assistants

Once a candidate for parallelisation has been discovered, there are various techniques which can be used to automatically perform the transformation of sequential code into parallelised code. With the exception of highly specific refactorings, code transformation is generally a much more challenging task than detecting potential sites for parallelism.

Some refactoring tools are much more specialized than others, and focus on one particular refactoring type or a feature that is specific to a single language. R. T. Khatchadourian et al.[5] introduces such a technique that focuses only on finding where it is advantageous and safe to replace 'stream()' with 'parallelStream()' in Java 8 programs.

Example of parallelizable Java stream[5]

```
list.stream().filter(x -> x % 2 == 0).map(x -> x * x).sum();
```

Despite the lack of generalisation, these specialised techniques should still be considered valuable, firstly because they may be transferable to other languages (streaming APIs like this are not specific to Java), and second because these focused techniques can still have huge benefits. With the ability to detect where ‘parallelStream()’ could be used, the authors were able to automatically scan over 1 million lines of code and safely introduce parallelization in 116 locations, resulting in an average speedup of 3.49 when tested. The tool makes use of the WALA library in the paper, which is used for performing static analysis, and so is naturally very useful for refactoring software. In addition, they use the SAFE library, another static analysis library which is dependent on WALA. While these are useful libraries for developing refactoring tools, the authors of the paper found that over 80% of their tool’s runtime could be attributed to typestate analysis performed by SAFE. This contributes to the tool's high average runtime of 70.26 minutes per candidate stream.

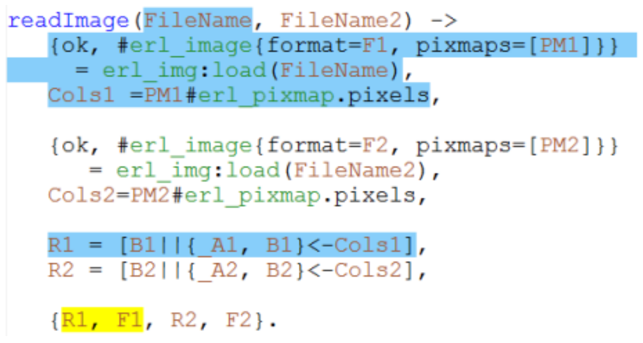
Another specialized tool is Relooper[6] which focuses on introducing Java’s ‘ParallelArray’ to loops which iterate over an array. After the user selects an array, Relooper automatically changes the type to ParallelArray and where it is safe to do so, parallelizes any ‘for’ loops the array is used in. This particular tool uses some strict safety checks. For example, only loops which iterate over all elements in the array are refactored, and only loops that process a single array element are considered. Another safety check that is required is to ensure that each element in the array is unique. If two or more elements in an array are references to the same object then threads may try to access the same object at the same time. This is another tool which makes use of WALA.

Relocker[7] is another Java refactoring tool, which instead focuses on optimising Java’s synchronized blocks. Synchronized blocks in Java are an easy and convenient way to implement thread safe data structures, however Java’s ReentrantLocks and ReentrantReadWriteLocks can have a significantly higher throughput in some scenarios, such as when there are more readers than writer threads. In these scenarios, relocker can be used to transform synchronized blocks into ReentrantReadWriteLocks.

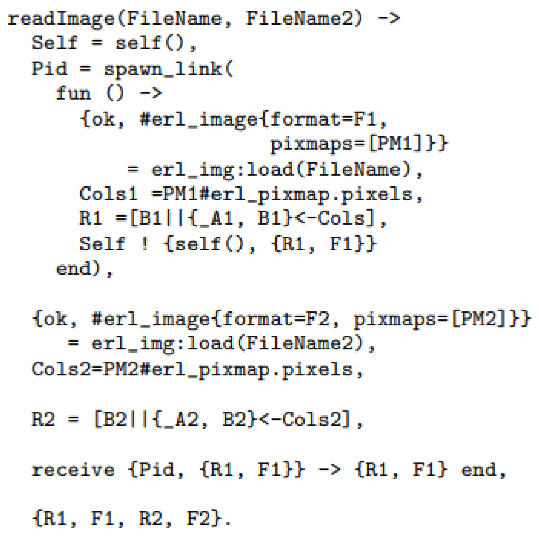
H. Li and S. Thompson[1] introduce methods for parallelising Erlang programs via program slicing. In particular they use slices defined by the portion of a program which can influence the value of a chosen variable. The most general of the three techniques proposed targets tail recursive functions. In a recursive function, an accumulator is defined as a function parameter that influences its own value, does not influence other values, and is not used as a base case for the recursion. The authors are able to use this for refactoring by moving the program slice of the accumulator into a parallel worker.

In another slicing based refactoring from the paper, the developer selects a subset of a function’s return values to be computed separately. In the example from the paper below, outputs R1 and F1 (highlighted in yellow) are selected, and the backwards slice from this selection (highlighted in blue) is moved into a separate process.

Slicing refactor, before (left) and after (right)[1]



```
readImage(FileName, FileName2) ->
    {ok, #erl_image{format=F1, pixmaps=[PM1]}}
      = erl_img:load(FileName),
    Cols1 =PM1#erl_pixmap.pixels,

    {ok, #erl_image{format=F2, pixmaps=[PM2]}}
      = erl_img:load(FileName2),
    Cols2=PM2#erl_pixmap.pixels,

    R1 = [B1||{_A1, B1}<-Cols1],
    R2 = [B2||{_A2, B2}<-Cols2],

    {R1, F1, R2, F2}.
```



```
readImage(FileName, FileName2) ->
  Self = self(),
  Pid = spawn_link(
    fun () ->
      {ok, #erl_image{format=F1,
                      pixmaps=[PM1]}}
          = erl_img:load(FileName),
      Cols1 =PM1#erl_pixmap.pixels,
      R1 =[B1||{_A1, B1}<-Cols],
      Self ! {self(), {R1, F1}}
    end),

  {ok, #erl_image{format=F2, pixmaps=[PM2]}}
     = erl_img:load(FileName2),
  Cols2=PM2#erl_pixmap.pixels,

  R2 = [B2||{_A2, B2}<-Cols2],

  receive {Pid, {R1, F1}} -> {R1, F1} end,

  {R1, F1, R2, F2}.
```

C. Brown et al.[2] propose cost-based methods for determining which pattern is best to use when there are multiple ways in which a sequential program can be parallelized. The cost of a parallel pattern is proportional to the time it takes to execute with an input stream of a given length. For example, the cost of a pipeline pattern is the time taken for the slowest stage of the pipeline to process an input. If a sequential program performs five functions, then the execution time will be the sum of the five functions. However if a five-stage pipeline is introduced to the same program, then the execution time will be reduced to time taken for the slowest of the five functions to execute, plus some minor overhead time. By using this type of mathematical model, the tool is capable of recommending the best pattern to introduce to a selected region of sequential code. While this is much faster than separately profiling each possible variation, it still requires the user to provide some parameters such as sequential execution time in order to work.

Several of the Erlang tools are implemented on top of a tool called Wrangler. Wrangler is a platform for supporting any type of Erlang refactorings, and this is very valuable for improving the accessibility of the tools, and improves the likelihood that the tools will be maintained and adopted by others. This is a philosophy also adopted in [4], where the authors point out the value of a refactoring toolset.

5.1 Code Assistants vs Compiler Refactoring

All of the work presented in this paper has focused on tools which modify source code rather than automatically performing refactorings as a compiler step. The argument for the compiler refactorings is that they would prevent large amounts of auto-generated code being added to projects which are verbose, hard to understand, and difficult to modify and maintain.

While this is a valid point which emphasises the importance of generating readable code, there are several reasons why source code refactoring assistants are a more common area of research. The overarching reason is that fully automated parallelism refactoring techniques are not yet good enough to be trusted. Source code refactoring is transparent and allows developers to confirm that an automated change is doing what they expect it to. Another reason stated in [1] is that introducing parallelism is a significant change that should be reflected in a project's source control repository.

5.2 Language Considerations

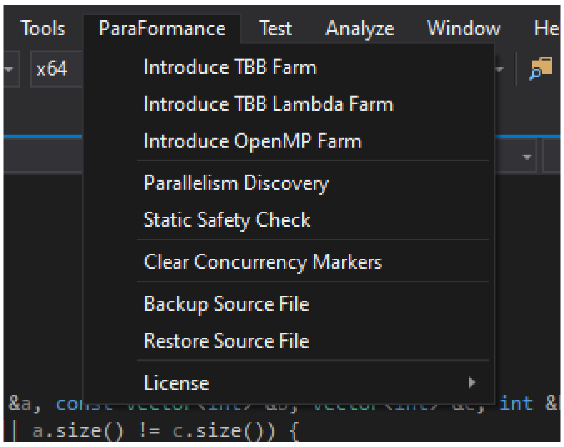
Language tools can have some major differences depending on which programming language they target. The biggest differences occur between functional languages (Erlang and Haskell), and object oriented languages (Java). This is due to the fact that functional languages tend to be more ‘pure’ as functions often have almost no side effects, and lack of side effects is very suitable for code to be parallelized. Generally, once a variable is assigned in a functional language, it cannot be mutated, and instead a new variable must be created. Another reason is that references to objects present in object oriented languages are not suitable due to the way that they are easily accessed and altered across different threads. Furthermore, Erlang actually supports parallelism more than other functional languages as it has built in functionality for easily producing processes and message passing between them. These factors explain why there are a disproportionately high number of research papers which focus on Erlang relative to other more popular languages like Java.

6. Experiment

While parallelism refactoring tools are an active area of research, few tools are available for download, and fewer still are actively maintained or well documented. ParaFormance however, is a tool which has been deemed complete and useful enough to be commercialized, and is advertised with its ability to automatically find and refactor parallelism in C++ applications. ParaFormance was chosen to be evaluated in order to review the current status of state of the art parallelism refactoring tools.

The tool is available as a plugin to both Eclipse and Visual Studio, for this evaluation Visual Studio was used. The tool adds a convenient menu to Visual Studio with several available options, the most important being ‘Parallelism Discovery’, and the three choices used for introducing the farm pattern.

Paraformance menu

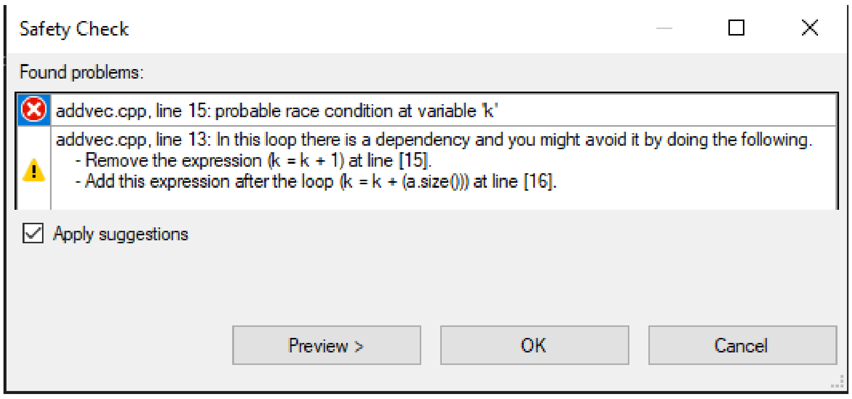


```
&a, const vector<int> &b, vector<int> &c, int &
| a.size() != c.size()) {
```

6.1 Vector Addition

The first task completed was to parallelize a simple function for performing vector addition[12]. Within a couple of seconds the ‘Parallelism Discovery’ option was successfully able to highlight the loop used for actually summing the vectors together. The discovery option uses profiling techniques, and so was highly dependent on the size of the vectors to be combined. After selecting the loop and choosing the ‘Introduce TBB Lambda Farm’ option, the tool warns of a race condition with the counter variable ‘k’. It was then also able to suggest a fix - perform the addition as a single step outside of the loop.

Warning and suggest fix



Safety Check

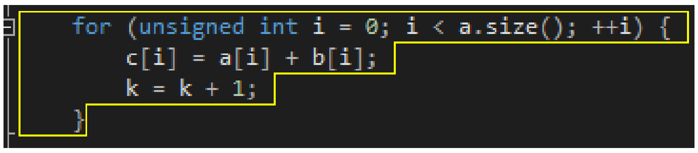
Found problems:

addvec.cpp, line 15: probable race condition at variable 'k'

addvec.cpp, line 13: In this loop there is a dependency and you might avoid it by doing the following.
  - Remove the expression (k = k + 1) at line [15].
  - Add this expression after the loop (k = k + (a.size())) at line [16].

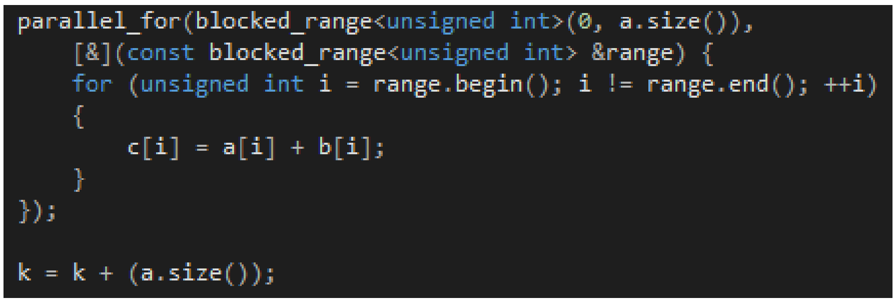
As shown, the code was successfully transformed to use a TBB 'parallel_for' and avoid the race condition, without drastically complicating the code.

Parallelism discovery highlights the loop



```
for (unsigned int i = 0; i < a.size(); ++i) {
    c[i] = a[i] + b[i];
    k = k + 1;
}
```

After the automated refactoring



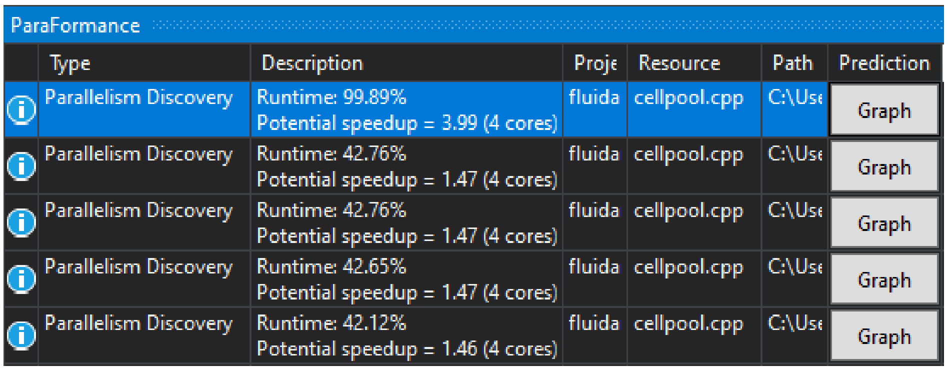
```
parallel_for(blocked_range<unsigned int>(0, a.size()),
    [&](const blocked_range<unsigned int> &range) {
    for (unsigned int i = range.begin(); i != range.end(); ++i)
    {
        c[i] = a[i] + b[i];
    }
});

k = k + (a.size());
```

The tool provides a graph[Appendix 1] for each proposed refactoring, and plots the number of cores against estimated speedup (from 1-512 cores). With six core hardware, the estimated speedup given was 2.1 for the overall runtime of the program. This was put to the test by manually recording runtime before and after the refactoring, over an average of 10 runs. To complete vector addition for a vector containing 100 million elements, the average sequential runtime was 85.48 seconds. After refactoring, the average runtime was reduced to 38.32 seconds, yielding a speedup of 2.22, very close to the predicted value of 2.1.

6.2 Fluid Simulation

To trial ParaFormance against a less trivial and more practical example, the next challenge was to parallelise the fluid simulation tool included as part of the Parsec benchmark[13]. Again, the first action was to try the 'Discover Parallelism' option. Being a more substantial program of a couple thousand lines, this discovery stage took approximately 8 minutes to complete, but this time produced 20 results. The top result claimed to include 99.89% of the runtime and offer a 3.99 speedup (with a 4-core machine). However, this result highlighted the main loop of the program which iterates through each frame of the simulation, which explains the high runtime percentage, but means that this loop is certainly not actually possible to parallelise, as each frame is dependent on the previous. In fact, it turned out that none of the detected loops could be automatically parallelized without introducing race conditions that must be manually solved.

Top 5 results from discover parallelism on the fluid simulator



ParaFormance

	Type	Description	Proje	Resource	Path	Prediction
ⓘ	Parallelism Discovery	Runtime: 99.89% Potential speedup = 3.99 (4 cores)	fluida	cellpool.cpp	C:\Use	Graph
ⓘ	Parallelism Discovery	Runtime: 42.76% Potential speedup = 1.47 (4 cores)	fluida	cellpool.cpp	C:\Use	Graph
ⓘ	Parallelism Discovery	Runtime: 42.76% Potential speedup = 1.47 (4 cores)	fluida	cellpool.cpp	C:\Use	Graph
ⓘ	Parallelism Discovery	Runtime: 42.65% Potential speedup = 1.47 (4 cores)	fluida	cellpool.cpp	C:\Use	Graph
ⓘ	Parallelism Discovery	Runtime: 42.12% Potential speedup = 1.46 (4 cores)	fluida	cellpool.cpp	C:\Use	Graph

The Parsec benchmark includes a manually parallelized version of the fluid simulation which makes use of TBB. Where ParaFormance would simply try to create task farms for heaviest for loops, the manual implementation is far more in depth, and has a completely different structure to the sequential version. The parallel implementation is highly specialized for the program and uses spatial partitioning so that different areas of the simulation are managed by different threads. Although TBB can perform thread and lock management duties, the benchmark implementation uses manual lock placement and thread creation to maximise efficiency. While the benchmark is all but certain to perform faster than an automated refactoring, it would have been interesting to see how significant the difference was, had ParaFormance been able to make the change. Nevertheless, it was still valuable to learn strengths and weaknesses of the tool in its current form.

7. Conclusion

Tools for refactoring parallelism into existing software have made significant progress in recent years, and are effective at locating and safety checking potential sites to introduce parallelism. Generating parallelized code is effective for specialised scenarios, however there is still room for improvement before the tools become more widely used. While automated refactorings will never perform as well as code designed by parallelism specialists, the tools will prove highly beneficial as they become more accessible, and as the need to parallelise applications increases.

7.1 What I learned

During my research I learned about the techniques used to identify parallelism candidates, validate their safety, and automatically replace the sequential code. I also learned how these strategies differ between programming languages, and the ways in which the tools are presented to developers. By getting hands on experience with ParaFormance, I learned the benefits of these tools as well as where they can be improved.

References

[1] H. Li, S. Thompson. “Safe Concurrency Introduction through Slicing” in *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*. 2015. URL: https://kar.kent.ac.uk/46579/1/pepm03-thompsonA.pdf

[2] C. Brown, M. Danelutto, P. L. Kilpatrick, K. Hammond. “Cost-Directed Refactoring for Parallel Erlang Programs” in *International Journal of Parallel Programming*. 2013. URL: https://www.researchgate.net/profile/Christopher_Brown10/publication/236330731_Cost-Directed_Refactoring_for_Parallel_Erlang_Programs/links/004635229a026a95c0000000.pdf

[3] C. Brown, V. Janjic, A. D. Barwell, J. D. Garcia, K. MacKenzie. “Refactoring GrPPI: Generic Refactoring for Generic Parallelism in C++” in *Int J Parallel Prog 48, 603–625*. July 2020. URL: https://link.springer.com/article/10.1007/s10766-020-00667-x

[4] D. Dig. “A Refactoring Approach to Parallelism” in *IEEE Software, vol. 28, no. 01, pp. 17-22*. Feb 2011. URL: https://www.computer.org/csdl/magazine/so/2011/01/mso2011010017/13rRUxNEqNK

[5] R. T. Khatchadourian, Y. Tang, M. Bagherzadeh. “Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams”. 2020. URL: https://academicworks.cuny.edu/hc_pubs/632/

[6] D. Dig, M Tarce, C. Radio, R. Johnson. “Relooper: refactoring for loop parallelism in Java” in *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009*. Oct 2009. URL: https://www.researchgate.net/profile/Ralph-Johnson-3/publication/221320682_Relooper_refactoring_for_loop_parallelism_in_Java/links/00463536fb03523fa9000000/Relooper-refactoring-for-loop-parallelism-in-Java.pdf

[7] M. Schäfer, M. Sridharan, J. Dolby, F. Tip. “Refactoring Java Programs for Flexible Locking” in the *2011 33rd International Conference on Software Engineering (ICSE)*. May 2011. URL: https://manu.sridharan.net/files/icse11.pdf

[8] C. Brown, I. Bozó, V. Fördős, Z. Horvath. “Discovering Parallel Pattern Candidates in Erlang” in *Thirteenth ACM SIGPLAN Erlang Workshop*. Sep 2014. URL: https://www.researchgate.net/publication/263375518_Discovering_Parallel_Pattern_Candidates_in_Erlang

[9] D. R. Astroga et al. “Finding Parallel Patterns through Static Analysis in C++ Applications” in *The International Journal of High Performance Computing Applications*. Mar 2017. URL: https://www.arcos.inf.uc3m.es/jdgarcia/wp-content/uploads/sites/9/2017/02/ijhpca.pdf

[10] K. Molitorisz. “Pattern-Based Refactoring Process of Sequential Source Code” in *17th European Conference on Software Maintenance and Reengineering*. 2013. URL: https://www.semanticscholar.org/paper/Pattern-Based-Refactoring-Process-of-Sequential-Molitorisz/8054186dc016edab48abc56b9cb87c2f6c76b5c3

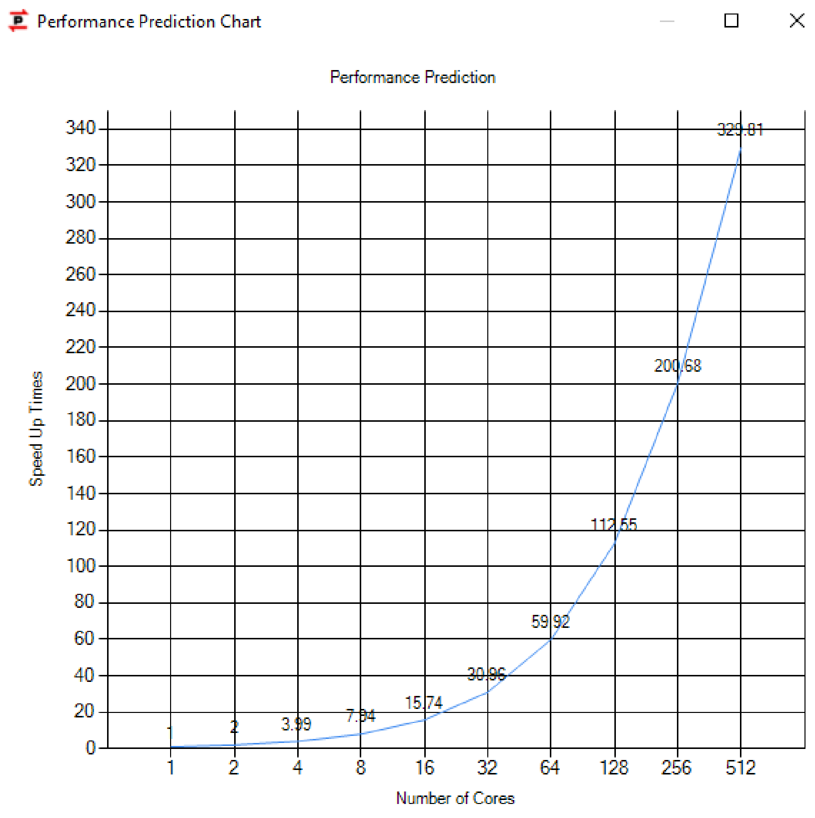
[11] David del Río, Manuel F. Dolz, Javier Fernández, J. Daniel García. “A Generic Parallel Pattern Interface for Stream and Data Processing” in *Concurrency and Computation: Practice and Experience*. ISSN: 1532-0634. DOI: 10.1002/cpe.4175. May 2017. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.4175

[12] Paraformance. ParaFormance Technologies Ltd. URL: http://www.paraformance.com/, https://github.com/Paraformance

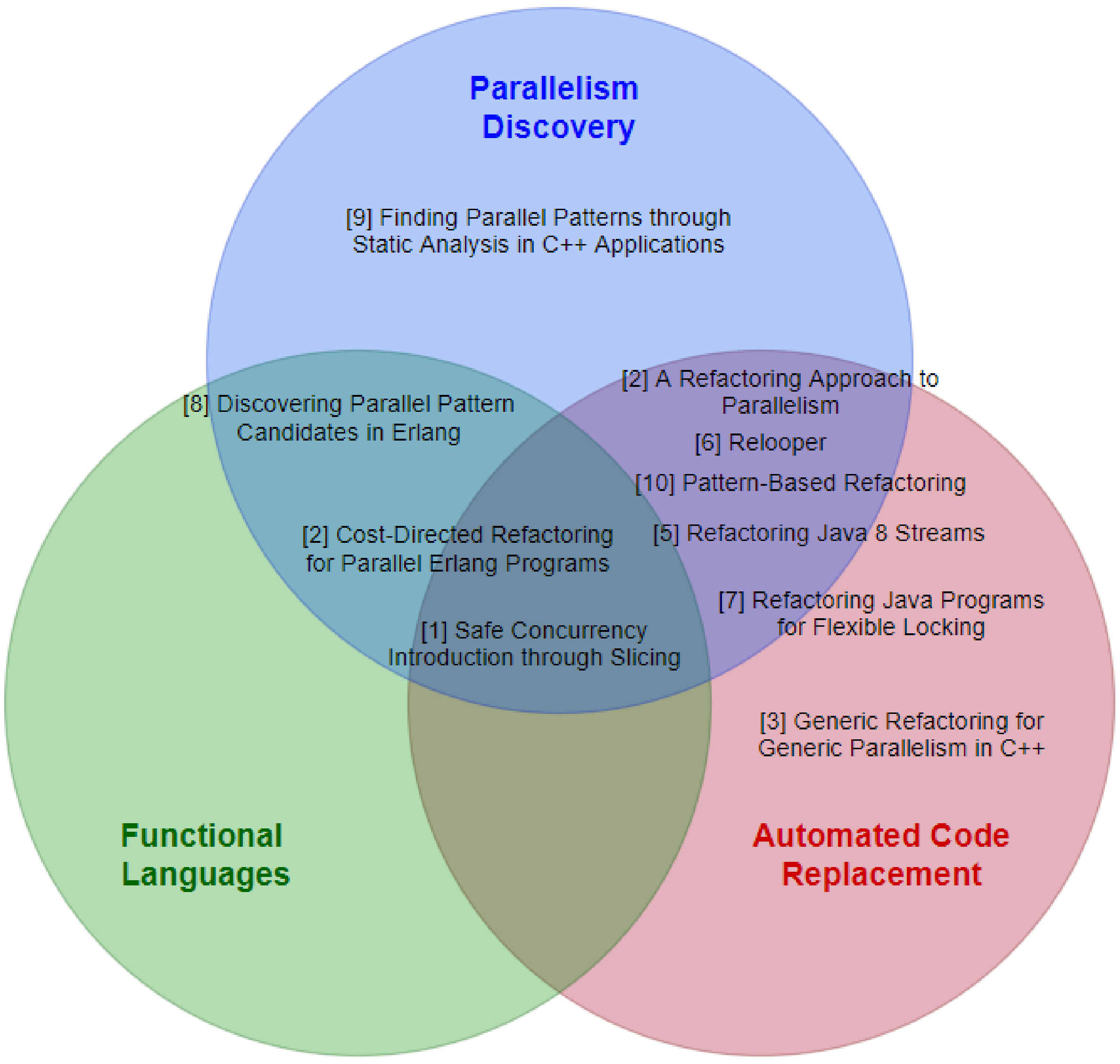
[13] Christian Bienia. “The PARSEC Benchmark Suite” in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques.* Oct 2008. URL: https://parsec.cs.princeton.edu/publications.htm

Appendix

Appendix 1. - A speedup estimation graph from ParaFormance



Venn Diagram
The papers researched are organized into the Venn diagram shown below with three categories. ‘Functional languages’ has replaced ‘IDE Integration’ since the paper outline - I found this to be a more relevant category for determining what sort of techniques are used within the paper.


Parallelism Discovery
[9] Finding Parallel Patterns through Static Analysis in C++ Applications
[8] Discovering Parallel Pattern Candidates in Erlang
[2] A Refactoring Approach to Parallelism
[6] Relooper
[10] Pattern-Based Refactoring
[2] Cost-Directed Refactoring for Parallel Erlang Programs
[5] Refactoring Java 8 Streams
[7] Refactoring Java Programs for Flexible Locking
[1] Safe Concurrency Introduction through Slicing
[3] Generic Refactoring for Generic Parallelism in C++
Functional Languages
Automated Code Replacement