University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Developing and speeding up bell ringing composition-finding software using parallelisation

Alistair Miller

12th April 2022

Abstract

The process of exhaustively searching for change-ringing composition sequences using software is very computationally intensive, and enthusiasts in the field have been working on trying to optimise these searches over the last couple of decades. However, the use of parallelisation as a means of speeding up composition-finding software has not been explored in depth, and it is thought that this could lead to an improvement in what can be found. In this project, we develop a composition-finding program in Java, and parallelise it by utilising multithreading. We ultimately obtain a speedup of around 8 times when running our program across 24 processors, rather than just one.

Acknowledgements

Firstly, I'd like to thank Professor Simon Gay for introducing me to the world of change-ringing (the rabbit hole of which goes down deeper than I ever could've imagined), and providing me with a great deal of knowledge and guidance over the course of this project.

I also owe my gratitude to Dr Ciaran McCreesh and Dr Blair Archibald for dealing with my basic questions that I frankly should have been able to figure out myself. There would be no results section of this dissertation without them.

Finally, I'd like to more informally thank, in an arbitrary order, Andrew and Jenny for seeing me through these last four years, my mother for her unwavering support and continual pestering about deadlines and suchlike (even when it seems futile to do so), and the people at Monster Energy for everything they've done for me over the years.

Education Use Consent

Contents

1 Introduction

This chapter will introduce the style of bell ringing known as change-ringing, and define some relevant bell-ringing terminology.

1.1 Introduction to change-ringing

Change-ringing is the art of ringing a group of bells in different permutations by continually changing the order in which they are rung. Each of these permutations is known as a **row**, and between each row being rung, some (or all) of the bells will swap places with a bell in an adjacent position. This transition between each row is called a **change**. Constantly changing the order of the bells in this manner means that the bells can be continually rang in different permutations.

1.1.1 Method-ringing

This project will be focused entirely on **method-ringing**, a form of change-ringing where the changes that are applied to the bells are given by a **method**. A method is a specific sequence of changes, and defines which bell *positions* switch at each change.

Figure 1.1 (Sturman 2016) shows a representation of the method *Plain Bob Minimus*, which has the bells move to the front and back of the order by alternating between swapping both pairs of bells, and swapping the inner two bells.

This collection of rows obtained is referred to as a **lead**.

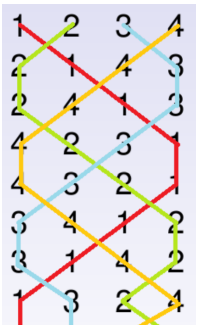

Figure 1.1: *A method for 4 bells*

1.1.2 Calls

Between leads, a method will typically instruct only the first and second bells to stay in place while the other pairs switch places, or only the first and last bells to stay in place while the other pairs switch places. However, continuing to ring leads following the same method puts a hard limit on the number of permutations that can be rung, as the changes will eventually lead the bells back to the initial row.

To be able to reach more of these permutations, a **call** can be issued at the end of a lead to perform a slightly different transition into the next lead. For the scope of this project, we only consider the **bob** call, which results in an additional cycle of three bells away from the row that would have been rang without the call (known as a **plain**).

The positioning of these calls between leads forms the basis for creating different method compositions.

1.1.3 Compositions

It is a common goal in change-ringing to ring as many different rows as possible, and as such, compositions almost always aim to find a path from **rounds**, the row consisting of all bells in order from highest to lowest pitch (eg. 12345678), through some number of leads, and ending back at rounds, all while only ringing unique rows with no duplicates. This row uniqueness property is referred to as **truth**, and a composition that fulfils this is known as a **true composition**. The composition itself is can be considered as the specification of calls and where they occur between the rows.

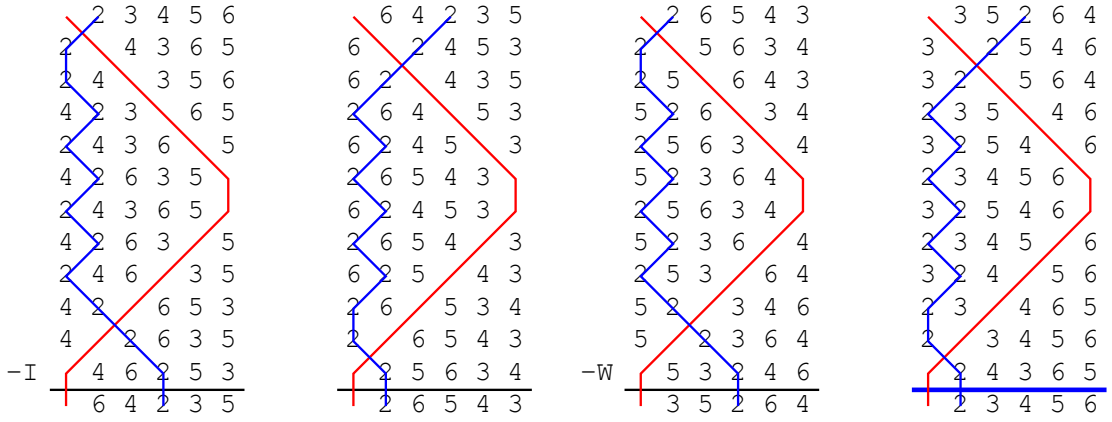

Figure 1.2: *A representation of a true composition (generated by* `complib.org`*)*

Figure 1.2 shows a 4 lead composition of *Double Oxford Bob Minor* (Church 2021), a method for 6 bells. This method defines a change at the end of the lead where only the first and second position bells stay in the same place, as discussed before. This is written as (12) in **place notation**, which shows only the bells that stay in place during a change.

Following that change repeatedly (a **plain** after every lead) would result in ending back at rounds after 5 leads, the path of which is known as the **plain course**.

For this method, calling a bob makes this an inter-lead change of (14), where the first and fourth position bells stay in the same place while the other two pairs of bells switch places. (note how the highlighted bells 1 and 2 stay in the same place at the end of each lead) Calling a bob at the end of the first and third leads (indicated by the -I and -W) [1] results in a different path can be taken to end back at rounds after just 4 leads. This is a true composition as all 48 rows are unique.

Compositions can be categorised into lengths depending on the number of rows they produce, a common one of which is called a **peal**, which contains at least 5000 rows. There are others such as a **quarter peal** containing at least 1250 rows, or an **extent** which contains every possible permutation of the set of bells, but this project will focus on peal-length compositions.

[1] the I and W stand for **In** and **Wrong** respectively, which are categorisations of a bob call, and depend on the position the **tenor bell** (the lowest pitch bell, in this case bell 6) is in after the bob.

1.2 Motivation

Method-ringing dates back to the 17th century, (Sanderson 1987) and as such, producing compositions originally had to be done by hand. This was the case for a long period of time, until the development of, and increasing availability of computers (particularly from the mid-20th century onwards) meant that compositions could be checked, and eventually discovered by software. (Papworth 1960)

However, exhaustively searching for compositions is computationally intensive, and although enthusiasts in the field have been working on trying to optimise such searches, the use of parallelisation to speed up composition-finding software does not seem to have been explored much.

Since the turn of the 21st century, multi-core processors have gone from being a concept (IBM 2012) to the standard - all modern computers support parallelism in hardware through at least one parallel feature. (McCool et al. 2012)

This shift means that parallel programming has become essential to take advantage of what computers are capable of, especially if the aim is to reduce the length of time required to compute the solution to a problem.

While the usage of parallel computing does not revolutionise solving computationally complex problems (for example *NP-hard* or other intractable problems), being theoretically able to reduce the running time of programs by orders of magnitude means that previously unfeasibly large instances of certain problems approach or become feasible.

As discussed by (PITAC 2005), "the practical difference this makes is substantial, as it qualitatively changes the range of studies that can be conducted".

So while this project only explores the utilisation of parallelism within a niche topic, the methodology is applicable to many areas of research.

1.3 Outline

The remainder of this dissertation is structured as follows:

- **Chapter 2** discusses existing composition-finding software and past work on speed optimisation.
- **Chapter 3** outlines the functional requirements of our composition search and the parallelisation of it.
- **Chapter 4** discusses the abstract design of how this project is going to be approached.
- **Chapter 5** details the implementation of our solution.
- **Chapter 6** discusses the results of the project and overall effectiveness.
- **Chapter 7** summarises the project as a whole.

2 Background

2.1 Similar composition-finding software

2.1.1 Elf

Elf (Davies 2002) is a composition-finding program written in Java.

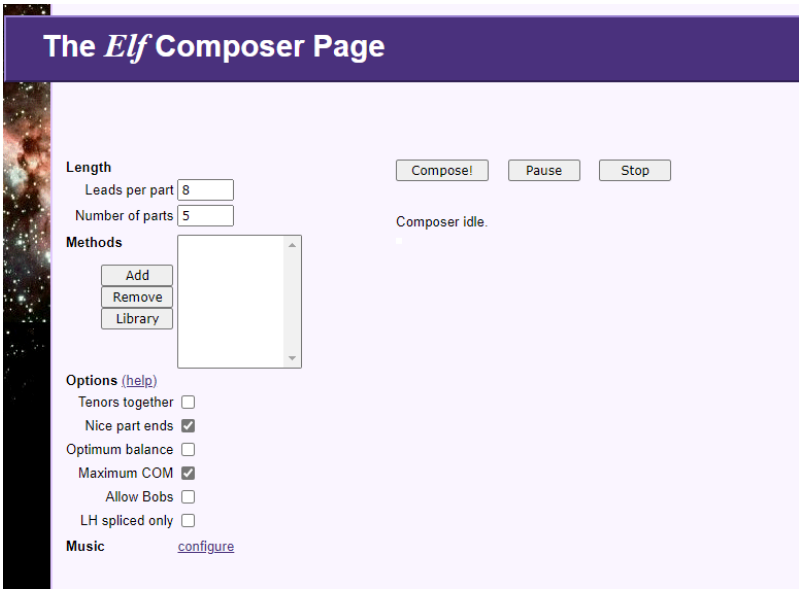

Figure 2.1: *The user interface of the web version of Elf (Davies 2002)*

Advantages

- Elf allows for many types of composition to be found, such as searching for **spliced** compositions (using multiple methods) or using half-leads as the smallest "block".
- Elf utilises different ways to improve the running time of the search, such as allowing for multipart searches, or keeping the tenors together throughout the composition.

Disdvantages

- Elf uses a directed graph of nodes (Davies 2002) (containing different information depending on the type of search) to implement the search - and has an initial phase of generating the full node table of all of these, which can be a considerable overhead, depending on the search criteria.
- Allowing for searches such as spliced significantly increase the complexity of the directed graph - because of this, Elf can only support finding compositions up to quarter-peal length.

It seems to be the case that Elf cannot support searching for peal-length compositions due to either offering too many features (particularly spliced composition searches, as any implementation would get exponentially longer with each increase of the number of possible methods to splice), or that the program has to initially calculate the full vertex table before performing the search.

There exist many other programs that implement composition searching, but it seems that any software similar to what this project aims to produce are either defunct (such as BYROC, which allowed for a search for peal-length compositions with a single method), or have no documentation available to analyse how their implementation works (such as Compositeur, which allows a search for single method compositions up to 12 bells, with lengths up to extents).

2.2 Speeding up composition-finding software

2.2.1 Full Monty Search benchmark

In 1995, Graham John completed the first exhaustive search of all the compositions of the method Cambridge Suprise Major (with the tenors together condition), which took 9 days to complete using a search program developed in a 16-bit assembly language. Then in 1998, the same search was repeated with an updated, 32-bit Assembler searching program, which took 25 hours. (John 2002) Since then, this exhaustive search of Cambridge has become a form of benchmark for the speed of composition-finding software (although the aforementioned 32-bit Assembler program *SMC32* seems to still be the fastest program, and the benchmark is more dependent on the CPU that it using it).

While this exhaustive search includes **single** calls, which were intended to be out of the scope of this project, these can also be implemented to see how our program compares to others using this benchmark search.

3 Analysis

This chapter will discuss the requirements of the software and the priority of various other aspects that could be implemented.

3.1 Functional requirements

The potential features of the system were categorised into different levels based on the MoSCoW method of prioritisation (Clegg and Barker 1994), detailed as follows.

3.1.1 Composition search

Firstly, we need to develop a working search progam for finding compositions. This comes with its own set of potential features.

Must have

- The program must be able to search different paths of bobs and plain calls for compositions.
- The program must have some internal representation of a method, and be able to apply it to rows.
- The program must be able to search for only true compositions, so must be able to check that all rows are unique.

Should have

- The program should be generalised to support searching for compositions of different methods.
- The program should have some form of output for valid compositions.

Could have

- The program could output compositions in a compact tabular form, such as in Figure 3.1, which condenses a composition down to its bob calls to make it easier to read and analyse.
- The search could be adapted to work with finding multipart compositions.
- The search could be adapted to work with finding spliced compositions, where multiple methods can be used simultaneously.

23456	M	W	H
35264	2	2	3
56342	2	2	3
64523	2	2	
42635	2	2	3
23456	2	2	3

Figure 3.1: *Tabular form for compositions*

3.1.2 Parallelisation

Once a working search program has been developed, we can then try to parallelise it.

Must have

- The search program must be able to be divided up into smaller searches.
- These smaller searches must be able to be run independently, and simultaneously.

Should have

- **(Hardware)** - to be able to take advantage of the speedup gained via parallelisation, the program should be run on a powerful multi-core processor capable of running over many threads.

Could have

- The code could be made to be easily parallelised at different levels (eg running on 4 cores, running on 16 cores etc)

4 Design

This chapter will discuss the possible abstract designs that both the composition-finding software and the parallelisation of it could follow.

4.1 Graph representations

As change-ringing is fundamentally built on the transitions between rows, the usage of graph vertices and edges are a fitting data structure to use as our representation.

4.1.1 Graphs of rows

We first consider creating a graph where each vertex represents a different permutation of bells, and connecting these by edges that represent every possible change between two rows. A visual representation of the graph obtained for 4 bells is shown in Figure 4.1, where the edges are also categorised into colour by which type of change they represent (for example, a green edge represents both pairs of bells swapping positions, and a purple edge represents the inner two bells swapping positions).

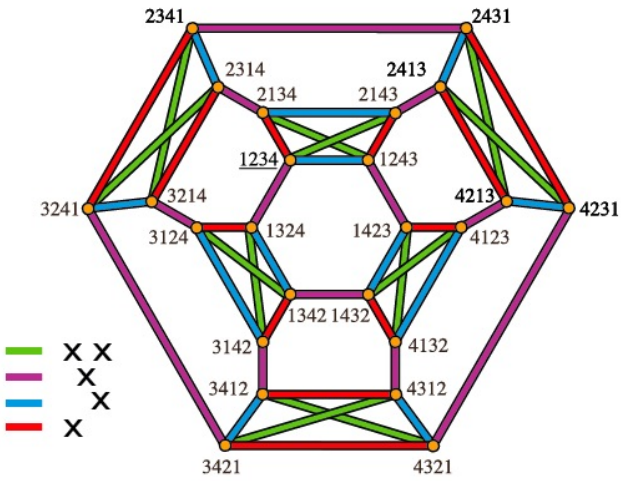

Figure 4.1: *Graph showing all possible changes on 4 bells (Polster and Ross 2009)*

Each true composition can now be represented as a cycle in this graph, a path starting and finishing at the **rounds** vertex that visits any other vertex at most once. This also means that searching for an **extent** is the equivalent of searching for a Hamiltonian cycle in the graph, a path that starts and finishes at the **rounds** vertex that visits every other vertex *exactly* once.

The graphical representation of one such extent is shown in Figure 4.2, the path that corresponds to the **plain course** of Plain Bob Minimus. (introduced in Figure 1.1)

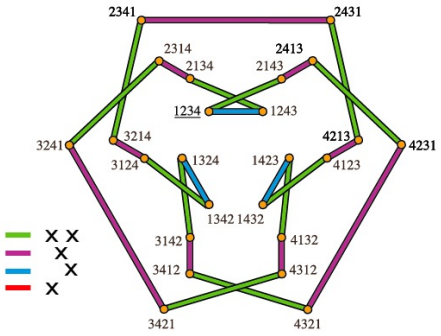

Figure 4.2: *The plain course of Plain Bob Minimus, shown on the graph of rows (Polster and Ross 2009)*

As discussed in Bellringing and Maths (St Martin's Guild SMGCBR 2021), we could also use a directed graph rather than undirected edges, where each row vertex points only to the others that can be reached.

4.1.2 Graph of leads

For all method-ringing in the scope of this project, it is assumed that all compositions are made up of full leads, and calls are only made between leads. This means that each lead can be considered as one cohesive unit, and it makes more sense to consider these the building blocks of our compositions.

Therefore, we now consider creating a graph that has each vertex represent a whole lead. As we are only considering calls of bobs or plains after each lead, we can assign the leads obtained from each of these calls as child vertices.

Using this structure, and also the fact that our graph does not need to be initialised with every possible vertex, we arrive at a binary tree design, a diagrammatic example of which is shown in Figure 4.3.

When searching for compositions, we can implement a depth-first search of this tree.

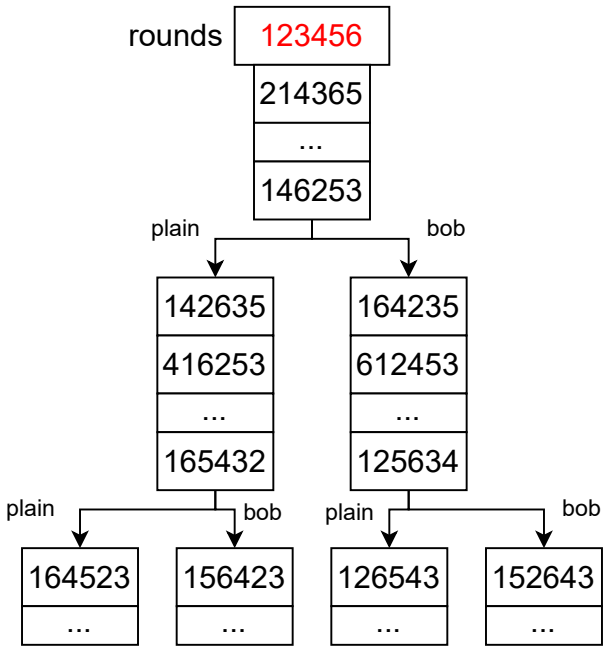

Figure 4.3: *An example binary tree of leads, starting from rounds*

4.2 Parallelisation

The fundamental idea of parallelisation is to divide tasks into some number of smaller sub-tasks which can be run independently and simultaneously, and then combining the results from each of these.

This is often done by repeatedly halving a task into two sub-tasks until the desired level of division is reached. An example of this is shown in figure 4.4. For our composition search using the binary tree of leads (as detailed in section 4.1.2), we can see that each level creates two sub-trees of equal size [1] so we can follow a similar process of creating sub-tasks by creating our tree down to a certain level, and then, treating each leaf node as its own root, assign searching each sub-tree from this point as its own task.

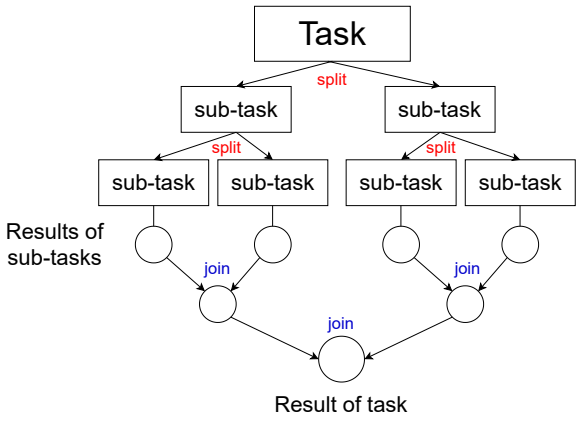

Figure 4.4: *An abstract view of parallelisation*

[1] Due to criteria we use to narrow the search space down (for example, checking for internal falseness), these two sub-trees will most likely not actually be of the same size or shape, but we will treat this split functionally as a halving.

5 Implementation

This chapter will explain the process of implementing the composition searches, and the parallelisation of them.

5.1 Coursing order search

Before starting to implement a search for full compositions, we first tried to implement a search on just the **coursing order** of a method. This was done mainly to gauge the feasibility of using a directed graph of rows (discussed in Section 4.1) to eventually search for compositions.

The coursing order is a way of describing how the bells follow each other around in a method, and in the case of this preliminary exercise, can be thought of as an underlying index of a lead.

When calling a bob at the end of a lead, the coursing order changes by a 3-cycle of bells, so implementing a search that starts and finishes at the coursing order for rounds functions very similarly to how the equivalent search would work with leads.

In a method such as *Cambridge Surprise Major*, a very common method for 8 bells, the **treble bell** (bell 1) and the **tenor bells** (bells 7 and 8) are in fixed positions in the coursing order [1] - only bells 2 through 6 are affected by these 3-cycles.

This allows us to test the directed graph implementation on a much smaller scale as we only need to create vertices for each permutation of the 5 bells that change position in the coursing order (giving us 5! = 120 vertices) rather than all 8 bells (which would give us 8! = 40320 vertices).

We implemented this search by first initialising a graph with a vertex for each coursing order (permutation of bells 2-6). We then iterate through each vertex and create its adjacency list - an array of vertices of the coursing orders obtained by calling each of three types of bobs - each of which cycle a different three bells.

We can then call a depth-first search from the vertex containing '53246' - the corresponding coursing order for rounds, back to itself.

While executing this search found several billion paths and did function as intended, we determined that using this graph of rows would not scale well to searching for full compositions and was not efficient overall - we therefore had to consider to a different design.

5.2 Composition search

We now implement a composition-finding search using the binary tree design as detailed in Section 4.1.2, which we hypothesised to be much more efficient for searching for compositions.

[1] Coursing orders are *cyclic* so can be rotated such that they stay in the same positions. This also assumes the **tenors together** limitation as previously discussed - which we can assume as this will also apply when searching for compositions.

5.2.1 Representations

We first need to define some data representations of different aspects of bell-ringing, to be able to work with them programmatically.

Methods

It was decided to represent a method by a nested array of integers, where each bell is given its own array, indicating which position it is in for each row.

Algorithm 1: Creating the representation of a method

```
Data: the first lead of a method for n bells, as an array of rows
Result: the method, represented as an int[][]
begin
    method ⟵ new int[n]
    for row ∈ lead do
        for i from 0 to (n − 1) do
            bell ⟵ row[i]
            append i to method[bell − 1]
        end
    end
end
```

Storing the positions generalises our representation such that we can reverse this process to generate a lead from a given lead head.

Internal falseness

To check for internal falseness, we need some structure to store which rows have been rung. This was initially done by storing an array of these rows as strings, although this very quickly became too inefficient. We therefore decided to use a BitSet, a Java implementation of an array of bits to indicate whether a row has been rung or not.

Lehmer codes (Diallo and Zopf 2020) provide a way of encoding a permutation of objects as an integer, and so we implemented the following algorithm to serve as a hashing function to use as indices for the BitSet.

Algorithm 2: Permutation hashing function rowToInt()

```
Data: a permutation of n bells
Result: an integer i in the range 0 ≤ i ≤ n! − 1
begin
    subtract 1 from all digits
    value ⟵ 0
    for digit ∈ row do
        value ⟵ value + (digit ∗ (numberofdigitstotheright)!)
        if anydigitstotherightarelargerthanthisone then
            subtract1fromthem
        end
    end
end
```

5.2.2 Initial depth-first search

We first implemented a composition search using vertices indexed by lead heads that stored their corresponding lead, a BitSet storing the rows that had been rung (as discussed above), an array of

lead heads followed on that path, and the corresponding sequence of calls (plain or bob).

We initialise a root node corresponding to the lead starting at rounds, and recursively create new child vertices (representing a plain or bob at the end of each lead) while performing the depth-first search.

Upon visiting a vertex, we set it as visited and generate the lead that corresponds to its lead head.

To reduce the number of vertices that have to be visited in the search altogether, there are some other criteria we use here to also set certain sub-trees as 'visited', as we know they cannot produce any compositions that fit our criteria.

We use the 'rung' BitSet to determine if any of the rows in this lead have already been rang. Continuing through this lead would make our composition false, so we can stop the search here and backtrack.

The other main criterion is we want the tenors to stay together throughout the composition. We use the method detailed above to determine whether this vertex's lead head has the tenors together, and if not, we can also stop the search here and backtrack.

To indicate either of these criteria being broken, we simply visit all successor nodes as having been 'visited', as the search would continue downwards through any 'unvisited' vertices. Therefore, any remaining nodes marked as unvisited are those that could produce a true composition, and so we check if either of these successor nodes have a lead head of rounds. If so, we have found a true composition, and we finally check that the path is long enough at this point to be of peal-length (consisting of between 5000 and 5600 rows, inclusive), and if so, we can finally output the composition.

Otherwise, the depth-first search continues down any unvisited child vertices.

This implementation worked as intended and returned the correct compositions, but was slow due to storing many (millions of) copies of data. Namely, having each vertex store its own copy of the rung rows BitSet, the path of lead heads taken, and the sequence of calls was very inefficient, as these data stored in these are only slightly different compared to those in the vertices above and below them.

5.2.3 Dynamic depth-first search

To try and alleviate this issue, we refactored the classes such that most of this data being stored in the vertices that was very similar between parent-child nodes was moved to be stored in the class containing the whole tree.

We can then continually update the data in the tree class by adding and removing elements as we traverse the tree as before, comparable to 'simulating' the original depth-first search with the tree class containing that data that would have been stored in the current vertex.

The rest of the search still functions using the same logic as before of setting vertices as visited and so on, we just perform this using much more lightweight nodes.

This is much more efficient as we are only storing one copy of the BitSet of rung rows, the current lead, the sequence of calls and the path of lead heads, as opposed to millions, as we had before.

Making these changes to the search program made the execution time much more reasonable.

5.2.4 Multipart composition search

Based on the single-part search as detailed above, we adapted the program to search for **multipart compositions**.

This was done by refactoring our vertices to store multiple leads at the same time, but this multipart search program was ultimately not used.

5.3 Parallelisation

As detailed in Section 4.2, we opted to implement a two-stage process for parallelising the search program - firstly creating the tree down to a certain level, and then treating the search from each leaf node as its own task.

5.3.1 Generating starting data

We perform the first stage by creating the data corresponding to each node at a certain level down from the root.

We can generate all possible starting sequences of plains and bobs by considering them as bits (0 and 1, respectively), and using the n-bit binary representations of all integers from 0 to 2^n, where n is our desired length of input paths.

We then take each of these call sequences, and using the same code that creates successor vertices in our search program, create the BitSet of rows, and array of lead heads taken corresponding to this input path.

5.3.2 Runnable/Thread

To utilise multithreading with our search programs, we implement the Java interface Runnable [2], which provides a run() method, which can be executed on a single thread. We use the same functions from the single part search in this Runnable class, and have the run() method start the depth-first search from the root of the sub-tree that this thread has been assigned to.

We add a constructor that stores the starting data so we can pass this in when instantiating each Runnable object.

We then create a separate class that performs the data generation in the first stage, creates the Runnable objects with this starting data, and invokes Threads to carry out each Runnable task.

5.3.3 ForkJoinPool

In order to limit the number of processors that are used simultaneously, we use a ForkJoinPool [3] which has a *parallelism* parameter, which sets the maximum number of processors that can be used (equivalently, the number of threads that can be concurrently running). We then submit all of the Runnable tasks into this pool, and the ForkJoinPool will take tasks from this pool and start them as processors become available.

[2] see `https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html`

[3] see `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html`

6 Evaluation

6.1 Program speedup

To gain a noticeable speedup running our search programs in parallel, we ran all searches on one node of the University's FATA cluster, a computer cluster consisting of nodes with dual Intel Xeon E5-2697A v4 CPUs. (McCreesh n.d.) There were 24 cores available to be used for multithreading [1].

Throughout this evaluation, we only consider the search for peal-length true compositions of Cambridge Surprise Major, as detailed over the course of this dissertation. This is because it has more **false course heads** than most other methods. (this reduces the running time of our search program in general, as we can consider that the likelihood of a branch of our tree ending the search at that point is higher than other methods) So performing searches of Cambridge provides a reasonable length of execution time to improve and measure.

6.1.1 Limiting the number of cores

Due to our implementation of generating starting data also checks if the paths are valid before passing them into threads, we can only have a limited number of paths at input. The table below shows the possible sizes of sets of starting data we can choose from for our Cambridge searches.

length of calls	# valid paths
0-2	1
3	2
4-6	4
7-9	7
10	13
11-13	24
14-16	41

Table 6.1: *Possible amounts of valid input paths*

Given this limitation, we first consider running a search with n valid search across n cores, and observing the speedup gain. Performing the search this way will always assign one sub-tree search to one thread.

We first obtain a base speed by running the search on one core, and then calulate the speedup for n cores by dividing the length of time it took to execute on one core by the length of time it took to execute the search on n cores.

Table 6.2 shows the running times of the search over each number of cores. As shown, we ran each test 3 times and used the means to determine the time for each number.

We use these means to calculate the base speed on one core and the speedup for each number of cores. Figure 6.1 shows the plot of this speedup, including error bars for the fastest and slowest of the 3 trials for each number of cores.

[1] according to Java's Runtime.getRuntime().availableProcessors()

	1	2	4	7	13	24
run 1	73619	36205	22085	22046	13571	9625
run 2	79585	37921	22166	22496	13514	9622
run 3	79550	39137	24195	22231	14198	9953
mean	77585	37754	22815	22257	13761	9733

Table 6.2: *The length of time (in milliseconds) for each run of the search program over n cores*

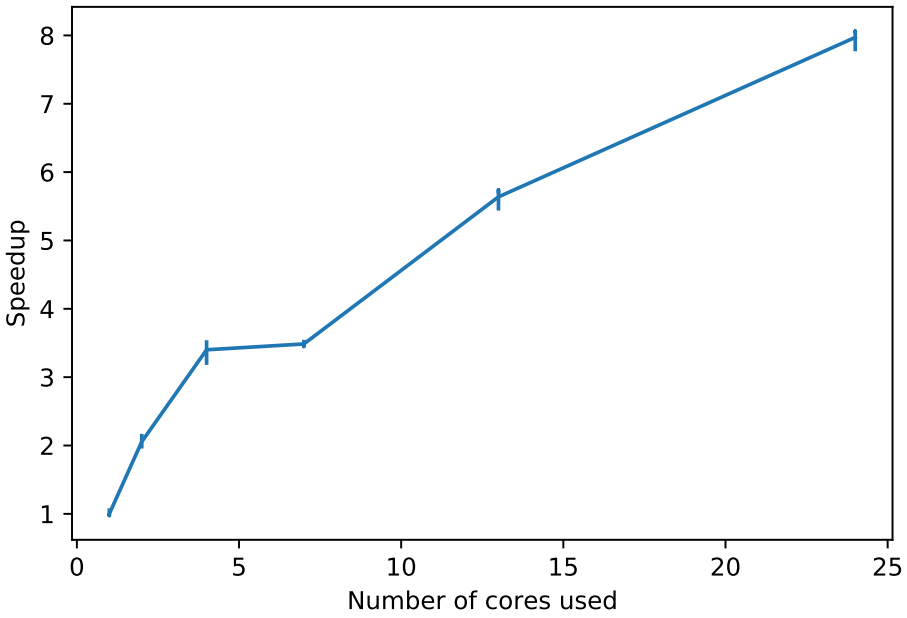

Figure 6.1: *Speedup for number of cores = number of input paths*

We obtain a graph with a shape similar to that which follows Amdahl's law, which gives us that the speedup does not increase linearly with the number of workers executing a parallel process, due to the serial (unparallelised) part of the program. (McCool et al. 2012) We instead get that running our program across 24 cores (with each thread handling exactly one starting path) grants us a speedup of roughly 8 times. (7.97x, using the means of the three trials).

We also see that there is almost no difference in the speedup going from 4 cores to 7 cores.

```
length 4 -> 4 paths
thread 3 finished:  5 compositions found
thread 4 finished:  5 compositions found
thread 2 finished:  5 compositions found
thread 1 finished:  8 compositions found
```

```
length 7 -> 7 paths
thread 4 finished:  5 compositions found
thread 5 finished:  0 compositions found
thread 3 finished:  4 compositions found
thread 7 finished:  0 compositions found
thread 6 finished:  5 compositions found
thread 2 finished:  1 compositions found
thread 1 finished:  8 compositions found
```

Observing the output from the program as shown above, we see that in both cases, thread 1 is assigned to search over a sub-tree containing 8 compositions. As this is the last thread to finish executing in both cases, we conclude that the speed of the search is limited by this one thread, hence there being no improvement despite using more cores.

6.1.2 Increasing the number of paths

We now try submitting more tasks to the ForkJoinPool than we have cores carrying out the search. For this test, we ran the search over 8, 16, and 24 cores, for each of 5 possible input sizes.

Table 6.3 shows the results for each trial. Again, we ran each search three times to take a mean execution time for each combination of paths and cores.

path length (# paths)		# cores 8	16	24
7(7)	run 1	21411	21261	21319
	run 2	20013	20737	21183
	run 3	21562	21081	21271
10(13)	run 1	11373	11886	11946
	run 2	11555	11495	11587
	run 3	11403	11935	12052
11(24)	run 1	12399	9259	9133
	run 2	13058	9330	8983
	run 3	12915	9120	9016
14(41)	run 1	12839	9746	9535
	run 2	12589	10398	9513
	run 3	12710	10093	9498
17(72)	run 1	9727	8049	7091
	run 2	9653	7531	7213
	run 3	9592	7767	7277
18(117)	run 1	11438	8675	8976
	run 2	11689	8528	9155
	run 3	11093	8602	9091

Table 6.3: *The length of time (in milliseconds) for each run of the search program over 8, 16, and 24 cores for varying input sizes*

Figure 6.1 shows the plot of these tests, including error bars out to the fastest and slowest of the 3 trials for each test.

As we can see, all three numbers of cores take approximately the same length of time to execute the search with 7 and 13 paths. The 8 core search keeps up with the larger numbers of cores as only 6 sub-trees out of the 13 contain compositions (see output below), and by quickly finishing threads with no compositions in them, the pool of size 8 can perform similarly to that of size 16 and 24.

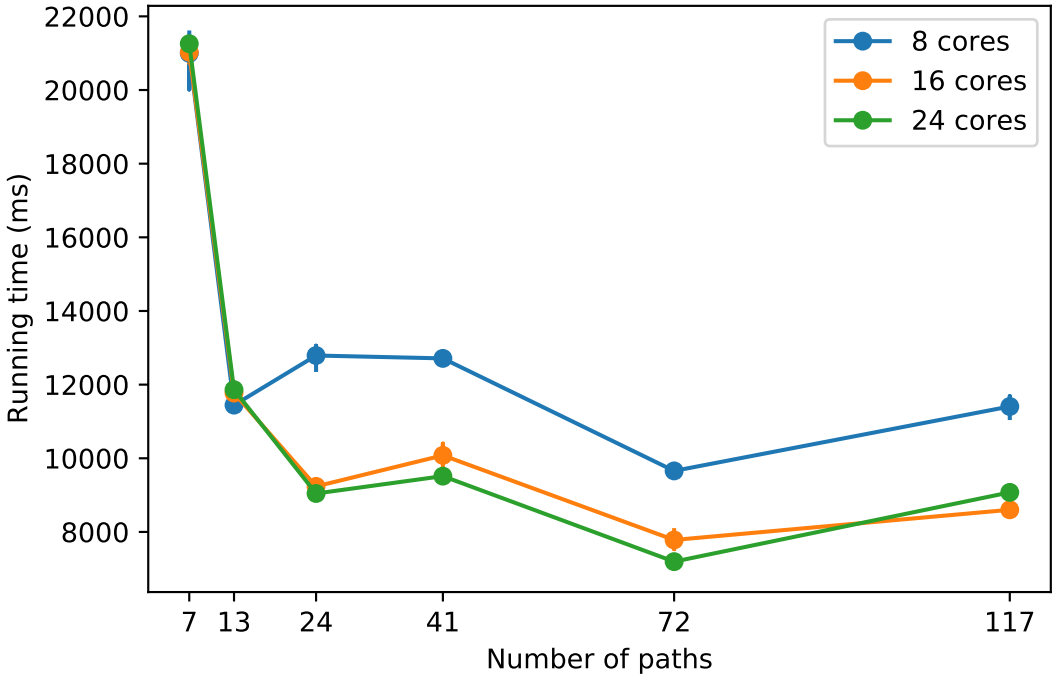

Figure 6.2: *Running time (in ms) for each number of cores, for each number of paths*

```
length 10 -> 13 paths
thread 9 finished:  0 compositions found
thread 6 finished:  0 compositions found
thread 13 finished:  0 compositions found
thread 3 finished:  0 compositions found
thread 11 finished:  0 compositions found
thread 8 finished:  0 compositions found
thread 10 finished:  5 compositions found
thread 5 finished:  4 compositions found
thread 7 finished:  5 compositions found
thread 12 finished:  0 compositions found
thread 2 finished:  4 compositions found
thread 4 finished:  1 compositions found
thread 1 finished:  4 compositions found
```

6.2 Full Monty Search benchmark

We added a 'single' call to our search program such that at the end of each lead of Cambridge Surprise Major, we either have a plain (12 in place notation), a bob (14 in place notation), or a single (1234 in place notation). This means that each vertex now has 3 children, making the search take longer. However, this addition made the search so much more complex that the program was unable to find any peal-length compositions after 45 minutes of running over 22 threads. (It did find over half a million compositions shorter than 5000 rows, but the FMS benchmark only considers peals.)

7 Conclusion

7.1 Summary

In this project, we sought to try and parallelise software for finding bell-ringing compositions. While many programs exist that perform this type of search, the usage of parallelisation did not seem to have been explored, despite composition-finding software being computationally intensive. We created a program using Java for finding these compositions by using a binary tree to represent leads, and carried out the search by implementing a depth-first traversal of this tree, that dynamically changes a set of data to determine when compositions have been found. We then parallelised this search by means of multithreading, where we adapted our search to be executable in a single thread, using Java's Runnable interface. Then by invoking new threads with these Runnable tasks, we concurrently ran our search program.

Ultimately, we were able to achieve a speedup of approximately 8 times when running our program across 24 processors compared to just one.

7.2 Reflection

The project can be considered a success - we were able to parallelise the search for bell-ringing compositions. Ultimately, most of the time spent on this project ended up being spent creating and debugging the composition-finding software itself. Using this knowledge retrospectively, since there are already many programs available that do the same thing, (in most cases better), we believe it would have been a better use of time to take some existing software (such as Elf, as discussed) and parallelise that, rather than also creating a search from scratch. This would have allowed much more time to focus on the parallelisation aspect of the project, which was the main aim due to being largely unexplored.

7.3 Future work

Given more time to progress further with this project, evaluating the current parallelisation implementation could be interesting to see for different categories of search - perhaps seeing what speedup would be obtained for searching for multipart or spliced compositions. There certainly could be more work done on the base search itself - the BitSet method used of determining internal falseness is overall very inefficient compared to other ways of determining which leads would make the composition false.

As noted in the evaluation, the naive method of splitting the trees into two to generate new tasks did not scale necessarily well to composition-finding - due to the distributions of where the compositions are within the trees (often, very few sub-trees contain most of the compositions, and the majority of sub-trees contained none of them), more time could be spent developing a smarter way of splitting up the search - perhaps carrying out an initial phase of determining which branches contain the most compositions and splitting *these* sub-trees down further into more tasks.

A Appendices

Bibliography

Church, P. (2021), '48 Double Oxford Bob Minor'.
URL: *https://complib.org/composition/84260/*

Clegg, D. and Barker, R. (1994), *Case Method Fast-Track: A RAD Approach*, Addison-Wesley Longman Publishing Co., Inc.

Davies, M. (2002), 'Elf: Under the hood'.
URL: *http://bronze-age.com/elf/underthehood.html*

Diallo, A. and Zopf, M. (2020), 'Permutation Learning via Lehmer Codes', *Santiago de Compostela* p. 8.

IBM (2012), 'IBM100 - Power 4 : The First Multi-Core, 1GHz Processor'. Publisher: IBM Corporation.
URL: *http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/power4/*

John, G. (2002), 'Cambridge Major Tenors Together - The Full Monty'.
URL: *https://changeringing.co.uk/cambridgefm.htm*

McCool, M., Reinders, J. and Robison, A. (2012), *Structured Parallel Programming: Patterns for Efficient Computation*, Elsevier Science.
URL: *https://books.google.co.uk/books?id=zpaHa5cjLwwC*

McCreesh, C. (n.d.), 'fatanode Usage Instructions'.
URL: *https://ciaranm.github.io/fatanodes.html*

Papworth, D. G. (1960), 'Computers and Change-Ringing', *The Computer Journal* **3**(1), 47–50.
URL: *https://doi.org/10.1093/comjnl/3.1.47*

PITAC (2005), 'Computational science: Ensuring america's competitiveness'.

Polster, B. and Ross, M. (2009), 'Ringing the changes'.
URL: *https://plus.maths.org/content/ringing-changes*

Sanderson, J. (1987), *Change ringing: The history of an English art*.

St Martin's Guild SMGCBR (2021), 'Bellringing and Maths'.
URL: *https://www.youtube.com/watch?v=EHRZswpnnxU*

Sturman, R. (2016), 'Mathematics of bell-ringing', `http://www1.maths.leeds.ac.uk/~rsturman/talks/gravity_fields.pdf`.