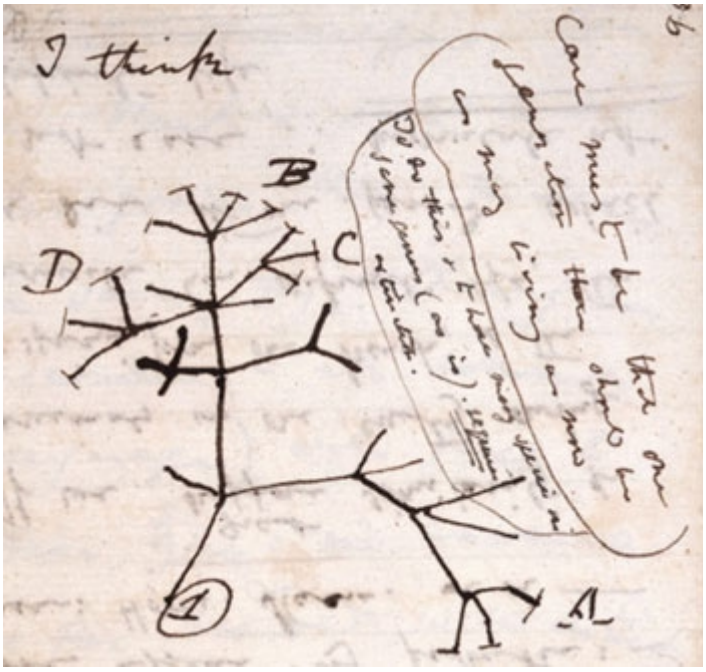
Speed Optimisation of "Trees of Life"

Leslie MacMillan

M.Sc. Computer Science Summer Project Report, 2014

Supervisor: Christophe Dessimoz



Charles Darwin's illustration of a tree of life, 1837

Abstract

The purpose of this project was to decrease the run time of a pre-existing scientific-computing algorithm, “Trees of Life,” which draws phylogenetic trees based on genetic distance. Project goals consisted in enhanced speed of the program and improvement in the author's computing skills.

The increase in speed was achieved largely through refactoring. Other avenues of speed increase were pursued, such as vectorisation, replacing function calls with direct calculations, and architecture tuning. This project required research into concepts from biology, low-level computer architecture, linear algebra, and numerical analysis. It developed the author's skills in Linux command-line environments, C++, refactoring, and speed optimisation.

The final speedup was 1.40X, representing a significant speed improvement. Learning objectives were also successfully attained.

Contents

Key Terms

Vectorisation: A method of single instruction multiple data parallelism. An example would be if a set of many numbers needed to be added, then the addition could be performed on several of them at once. (Clements, 2006)

Profiling: Analysis of certain aspects of a program such as time consumption.

Compiler: The computer program that translates high-level source code into low level assembly code.

Compiler Flag: An instruction to the compiler.

Optimisation: Changing a program to improve its efficiency.

Weight: Degree of confidence.

Pragma: A comment directed at the compiler.

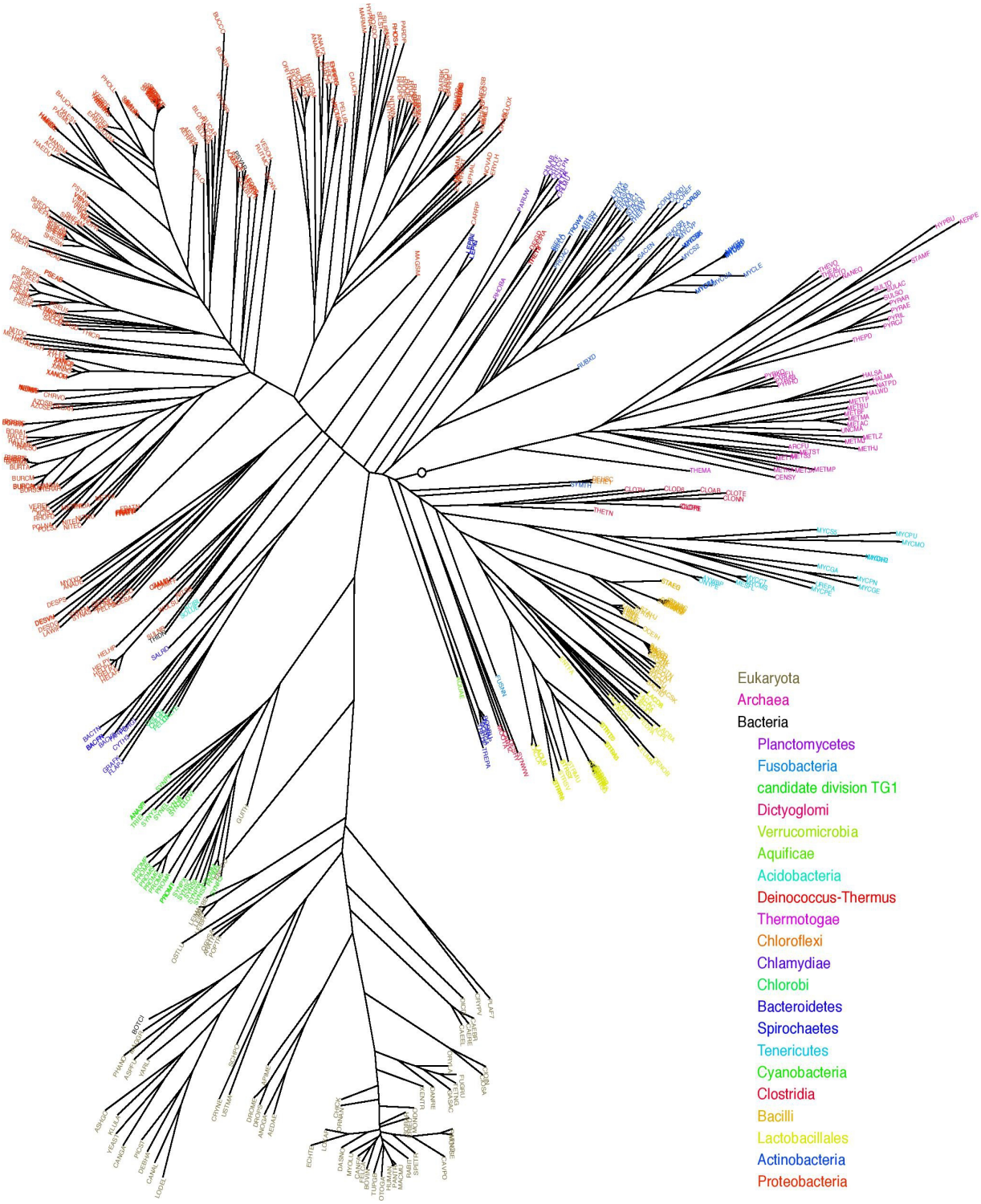


Figure 1: Phylogenetic tree created with Trees of Life program.

Chapter 1
Introduction

1.1 Statement of the Problem

Phylogenetic trees are graphical structures used in biology to explore and represent evolutionary relationships among organisms. Trees are generally drawn to show the branching of various species as they became distinct from one another over time; however, the Trees of Life algorithm draws unique phylogenetic trees for each gene, rather than an average based on whole-genome information. The purpose of this M.Sc. Project was to improve this existing algorithm by reducing the run time.

In this report, we will first examine the pre-existing code and its purpose, then discuss profiling of the original code. Following this, we will discuss the research done on optimisation theory and the changes that were made to the code based on this information. Then we will discuss the project conclusions and related concepts, such as possibilities for extension, and finish with reflections on the project process.

1.2 Background

For the non-biologist, the Trees of Life project can be thought of as a scientific computing algorithm which does many calculations on potentially large data sets, including many matrix computations. This is of course computationally expensive and the goal of this project was to optimise the existing code for enhanced speed, which potentially could lead to wider adoption by the scientific community and thus enhanced value.

1.2.1 On Tree Drawing and Trees of Life

Delineating relationships between species is relevant to the history of life on this planet and can help us trace the paths of derivation from ancient creatures to modern life. A "phylogenetic tree is a representation of the genealogical relationships among species, among genes, among populations, or even among individuals" (Yang, 2014; p. 70). It is a type of graph structure where there are "nodes for vertexes and branches for edges... the tips, leaves, or external nodes, represent present-day species while the internal nodes usually represent extinct ancestors" (Yang, 2014; p. 70).

The Tree Collection algorithm uses pairwise distance information (stored in a distance matrix) as the input to draw phylogenetic trees, as depicted in Figure 1. A pairwise distance is a metric for the similarity of two genes, found by performing a sequence alignment, and counting the degree of mismatch between two species using a certain formula. The value found by this pairwise comparison is treated as the observed distance (Yang, 2014). A distance matrix is a rectangular matrix which denotes the distance between two input points at the position in the matrix where they intersect. For example, if the distance between species 2 and species 1 is 12 units, the position (2,1) in the table would contain the value 12 units ("Distance Matrices in Phylogeny", 2014).

Working with this distance information, a tree is proposed which attempts to match all the pairwise distances for the leaves it contains as closely as possible. Some distances will be stretched and some shortened in order to create a tree that can most closely account for all the data at once. In this new tree, the branch lengths of given pairs of leaves are treated as the estimated pairwise distances. The Trees of Life algorithm uses a Sum of Weighted Least Squares formula to calculate a tree quality score based on how closely the estimated tree can match all the observed pairwise data for all species in the tree. It then selects the tree which has the best (lowest) such score. It does this iteratively on many small subtrees, eventually creating an over-arching tree for all given species.

Most phylogenetic trees today seek to establish the correct overall relatedness among species. They accumulate genetic information and use it in the aggregate. What that model does not account for, is that individual genes evolve at different rates. A gene necessary for metabolism may evolve very slowly, while one that is not as crucial may evolve very quickly. To the geneticist, individual gene evolution is useful information which is not represented in whole-genome average trees.

OMA (Orthologous MAtrix) is a database and associated online tools developed partially at UCL that gives information about orthologous groups of genes. Orthologs are genes that are present in two different species, which are related by heredity. That is, if one gene was present in an ancestor and a speciation event occurred, the corresponding genes in each resulting species would be orthologs. These similar genes in several species would constitute an orthologous group. The Trees of Life project seeks to draw phylogenetic trees among species based on new genomic information, and organizes these trees by orthologous gene groupings. Its contribution to the field lies in

drawing one topology (i.e. arrangement of branches) per dataset, yet having a variation on that tree for each orthologous group: so that the same diagram of species relatedness is modified (through varying branch lengths) to reflect the evolutionary distance by gene group. That is, the topology which reflects the species relationships is set, but because different genes evolve at different rates, the tree is redrawn as many times as there are genes, with variation in branch lengths reflecting the evolutionary distances of that gene among different species. Thus, Trees of Life is a distinct way of representing relatedness than the previous, singular Tree of Life model. (Although the same gene is rarely present in two species, in this paper orthologous gene group is used interchangeably with the term "gene" for simplicity).

1.2.2 On the Pre-existing Code

The full pre-existing code is available in the enclosed USB drive, in the folder marked "Original Code." It is instructive to examine this code to understand the opacity of the pre-existing code and its impact on the difficulty of the project. It consists of several files. The bulk of the program is contained in MinSqTree.cc. The code is already somewhat optimised, with "low-hanging fruit" optimisations already having been implemented. Often, these optimisations can cause code to be more difficult to read.

The project of speed optimisation has concentrated on a few of the most time-consuming functions within MinSqTree.cc, listed here:
FiveOptimCollection345() -This function is likely time-consuming because it calls functions which call FourSubtree()
FourSubtree() - This function focuses on fitting a four pronged tree, with areas of the tree extending beyond those four branches collapsed into the four nodes of the four pronged tree. It then accumulates weights for the distances of this new sub-tree. In other functions in MinSqTree, this sub-tree is combined with other sub-trees to draw a full tree.
FitQuartet() - Sets lengths between a similarly collapsed four-pronged tree.

The output of the code is two files, one ending in .nwk, which is a representation of the tree (in the Newick tree format), and one ending in .txt, which contains the final distance fit value. While the program is running, some updates are printed to the console. See Appendix for examples of the output. The newick tree file can be used by other programs (not used in this project) to draw the tree as seen in Figure 1.

Working with pre-existing code is often difficult, because its meaning can be difficult to discern by reading alone. Some projects attempt to ameliorate this with heavy documentation. In-program commenting can be an easy and direct way to explain code, however code can be made legible without comments if methods are extracted and named appropriately. Choosing meaningful variable and function names is also an important step in this process (Fowler, 2000; p. 88). The existing Tree Collection code contained relatively few comments. Method and variable names were relevant but sometimes in themselves insufficient to explain the code's meaning without prior

knowledge of the execution path. In this case, much of the meaning of the program was discerned through personal interviews with the project advisor who is also one of the authors of the code. This was combined with examination of the call graph information from gprof and careful close-reading of the source code while tracing control flow mentally, to create the call graph in Figure 2.

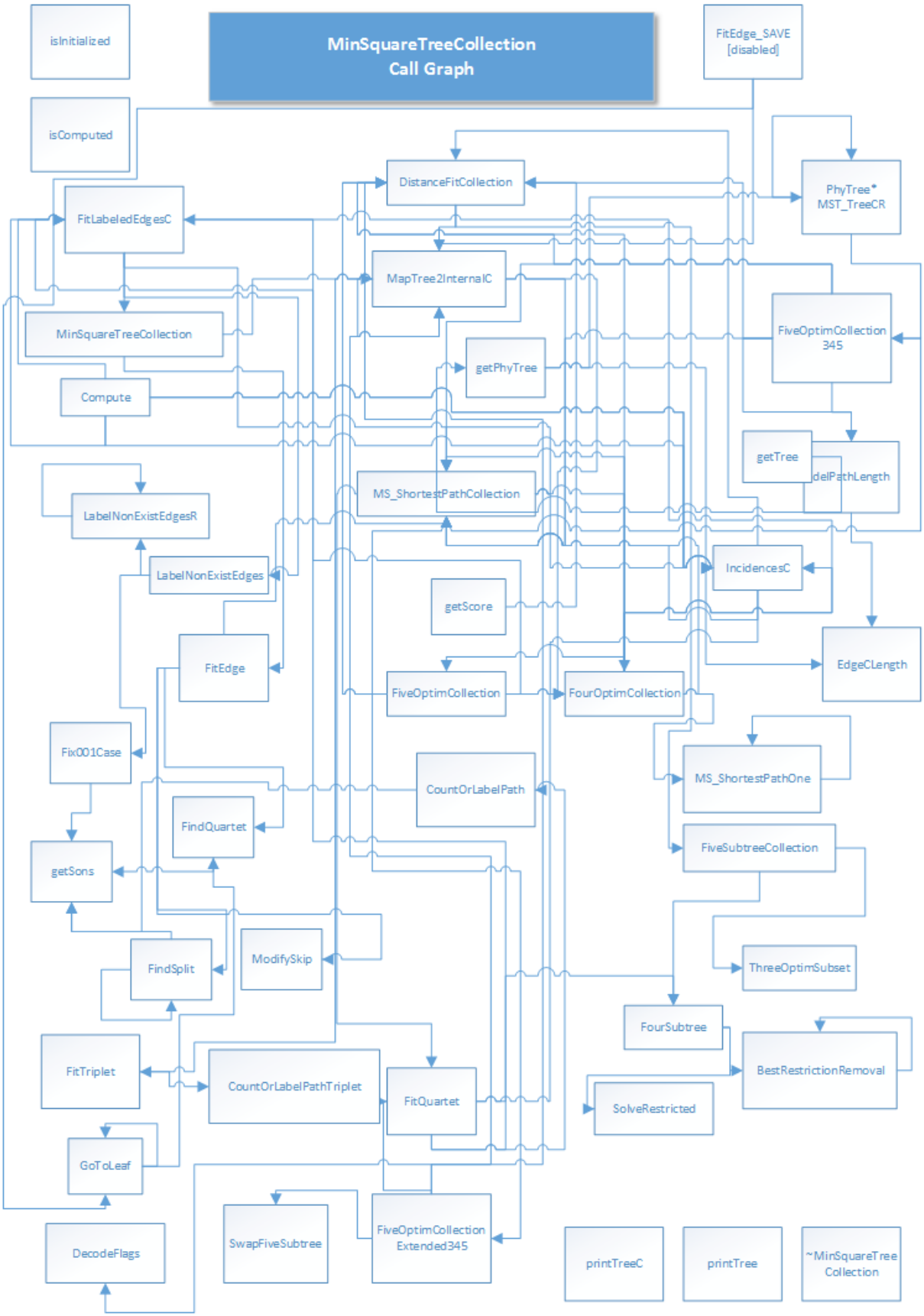


Figure 2: Call graph for MinSquareTreeCollection

1.3 Motivation and Main Achievements

The main contribution of this M.Sc. project to the Trees of Life project was a speedup of 1.40X (calculated based on a benchmark dataset described in section A.1). Another main output of this project was accomplishment of personal learning goals. I set out to learn about version control, Linux systems, terminal user interfaces, networking, low-level optimisation, and increase familiarity with working with existing code (as this is a common issue in the working world) and best practices for collaboratively working on code and programming in general. In this project I especially hoped to understand how to work on UNIX-like systems on a command line, which ended up being one of the largest challenges of the project. Another product of this project is the distillation of research on phylogenetic tree-drawing, linear algebra, computer architecture, and vectorisation.

Time of execution is an important factor in the user-friendliness of a product. The end-users of this product are scientists who may be limited in the amount of time they have to access computing resources. They may have extremely large datasets, covering thousands of species and hundreds of orthologous genes. By reducing the time demand of this program, I hope to increase its adoption by the scientific community, and therefore increase the impact of individual gene evolution on phylogenetic representations.

Chapter 2
Results

The results of this project are explained in this chapter in two parts. The first part covers profiling techniques and findings. The second part covers speedup research, design and implementation.

Results Part 1: Profiling

One strong message from optimising experts is that it is best to profile before optimisation (Amarasinghe, 2010). Thus, the first step in this project was to profile the program, that is, to analyze it with several different tools to find hotspots. Hotspots are areas of code which use a large proportion of run time. Hotspots are often found in tight loops, short pieces of code that are executed many times in a loop (Hot spot(computer programming), 2014); however, hotspots can be located in surprising places, and identifying these by intuition is often not as effective as using empirical profiling.

2.1.1 Description of test input data set

All tests shown here use the same sample input data set, which we will call data set one (DS1). It consists of data regarding about 50 orthologous groups in about 500 species. Some of the groups do not contribute any information for a given gene group, because that gene group is not present in that species. This dataset has a somewhat typical degree of this type of sparsity, with about 25% of species not contributing information for a given gene group.

2.1.2 Tools

There are many software tools which measure the actual time taken by the program. There are several different ways in which tools do this. They vary in what they measure: such as CPU time or elapsed total time; and the methodology of this measurement: some use sampling and others event counts. A combination of various tools can give a more complete picture of what is actually happening in the program. This project uses gprof, perf and Cachegrind to achieve profiling results.

Gprof

Gprof is the GNU profiler. It comes prepackaged with most Linux systems and is a subroutine-level profiler (Dowd, 1998; p. 110). It creates a flat profile, which gives time consumed by each function, and a call graph, which gives some indication of the hierarchy of control flow. For this and all other profiling results, please see appendix for in-depth output. Gprof was used to time various iterations of the code and to identify which functions take the most time.

Perf

Perf uses the data accumulated in performance counter registers. It also has an option perf_annotate which "annotates assembly or source code with event counts" (Perf Wiki 2014). In this project it was used for a feature which can provide annotated disassembly code; however, it was found that when compared with assembly code obtained through .s dump files, the disassembly view is simplified and not always accurately reflective of the actual assembly code being used to run the program. .S files contain actual machine code, and comparing them to the perf-annotate output shows a mismatch, as shown in following figures 3 & 4.

```
.L537:
.LBE31940:
.LBE31938:
.LBE31876:
.LBB31875:
.LBB31935:
.LBB31934:
    .loc 223 265 0
    cmpq    %r9, %r8
    jge .L539
    movq    %r10, %r15
    movq    %r8, %rax
.LVL855:
    .p2align 4,,10
    .p2align 3
.L540:
.LBB31887:
.LBB31888:
.LBB31889:
.LBB31890:
    .loc 6 908 0
    movsd   -8(%r15), %xmm2
    movhpd  (%r15), %xmm2
.LBE31890:
.LBE31889:
.LBE31888:
.LBE31887:
    .loc 223 265 0
    addq    $16, %r15
.LVL856:
.LBB31894:
.LBB31893:
.LBB31892:
.LBB31891:
    .loc 6 908 0
    movapd  %xmm2, %xmm1
.LVL857:
.LBE31891:
.LBE31892:
.LBE31893:
```

Figure 3: Sample .s file portion, assembly code output.

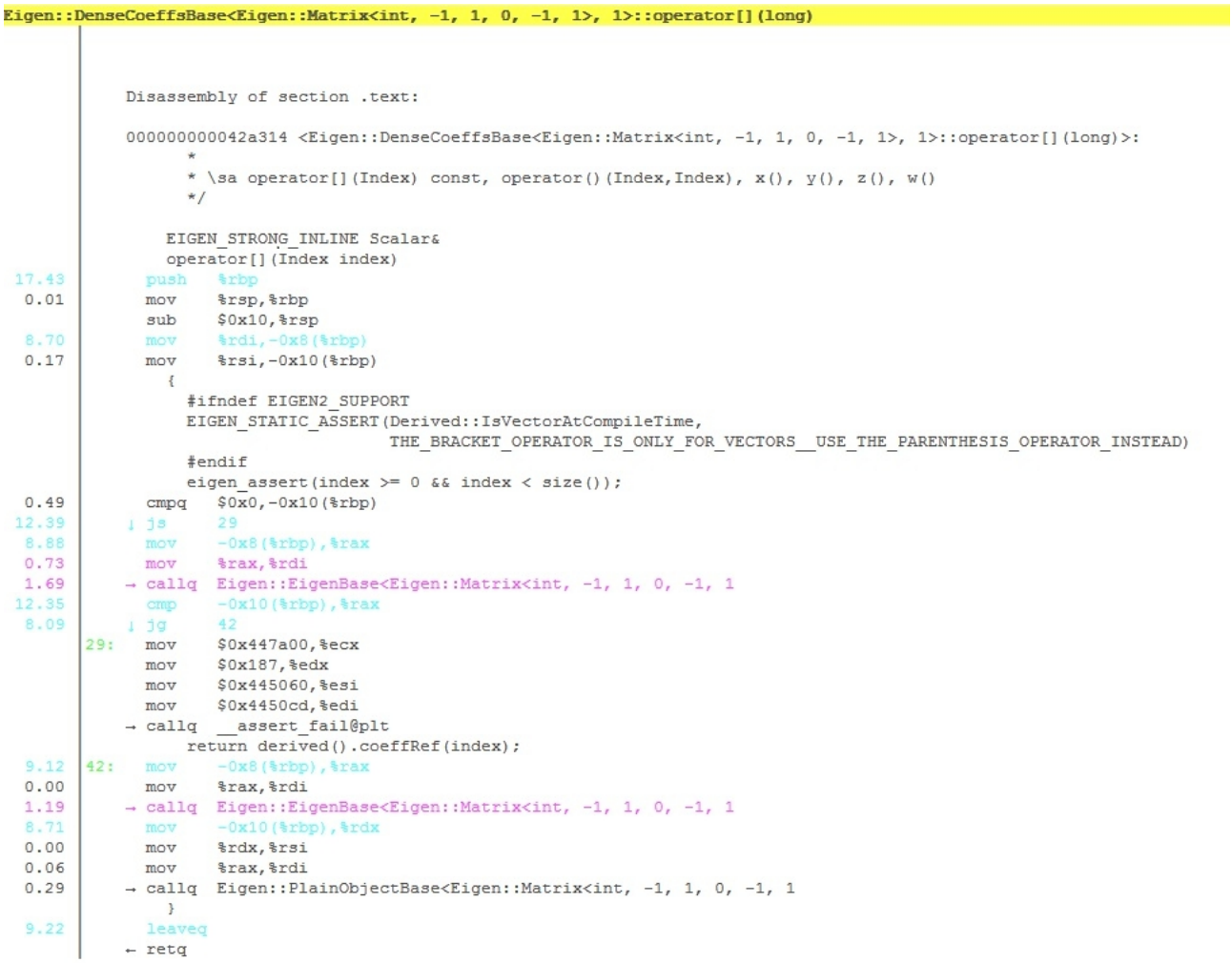


Figure 4: Sample output of perf_annotate. Disassembly, with percentage of event samples occurring per line noted to the left of the line.

Disassembly provides a reverse translation from machine code to assembly code, and is "not an exact science" (Disassembler, 2014). Figure 3 shows the machine code, which may contain the same instruction many times, if it is executed many times, so the complete .s file is longer by several times. The perf_annotate output is readable in comparison, but this sacrifices the detail necessary to understand what is actually being executed. Thus while perf_annotate was initially considered as a tool to verify vectorisation, it was not used to draw conclusions for this project.

Cachegrind

Cachegrind, part of the Valgrind tool suite, is designed to measure numbers and types of cache accesses. It can also provide information on branching and branch misprediction. The cg_annotate option annotates the source code to show the number of instructions actually executed per line, which proved to be an effective way to identify hotspots for this project.

2.1.3 The Importance of Compiler Flags

Optimisation relies on good communication with the compiler and the correct use and meaning of compiler flags need to be learned and implemented. If aspects of optimisation are left to the compiler, this takes advantage of possible future improvements to the compiler, and makes code more portable than hard-coding optimisations (Amarasinghe, 2010).

The optimisation modes for the g++ compiler used in the project are O0, O1, O2 & O3. These indicate to the compiler the degree of optimisation it should undertake. O3 is maximum optimisation and is likely the flag to be used on the final manifestation of a program; however it is not always appropriate to use during the implementation and testing phase. For example, in debugging, the error "value optimised out" may appear when trying to examine a variable value. This high optimisation level enables vectorisation and other memory tricks which may not follow a straight path through the source code, and this may be confusing when debugging. In this case it may be better to use O0 or O1. However, this will significantly slow down the program, and may give an inaccurate representation of the final behaviour of the code, which will likely be using O3 optimisation. Thus proper understanding and use of these flags is key (Gentoo Wiki, 2014).

2.1.4 Profiling Results

Using profiling tools leads to insights either unattainable or not easily attainable through intuition alone. In this section, we will examine the findings determined through use of the above profiling tools.

2.1.4.1 Time by function

The first step in profiling was to establish a ranking of functions by time, to prioritize speedup efforts. Using gprof, the program was profiled to see which functions were taking the most time. The "flat profile" section of the gprof report ranks the most-timing consuming functions, and lists the absolute and relative time taken by each. The below figure shows time consumed by various functions at various levels of compiler auto-optimisation. This information was used to prioritize optimisation efforts. Although many of the highest time-consuming functions took place in the Eigen library, modifying this library was considered beyond the scope of this project. The highest-time-consuming functions in MinSquareTreeCollection were FiveOptimCollection345 (abbreviated in this report as FOC345) and FitQuartet.

Function	Time spent in self O0 (s)	Time spent in self O3 (s)	Time spent in self O3 mtune=core2 (s)
Eigen::DenseCoeffsBase<Eigen::Matrix	434.91	--	--

<int, -1, 1, 0, -1, 1>, 1>::operator[](long)			
MinSquareTreeCollection::FiveOptim Collection345	337.36	172.13	174.27
Eigen::EigenBase<Eigen::Matrix<int, -1, 1, 0, -1, 1> >::size() const	262.34	--	--
Eigen::DenseStorage<int, -1, -1, 1, 0>::rows() const	258.75	--	--
Eigen::EigenBase<Eigen::Matrix<int, -1, 1, 0, -1, 1> >::rows() const	234.43	--	--
MinSquareTreeCollection::FitQuartet	229.08	99.60	99.91
Eigen::DenseCoeffsBase<Eigen::Matrix <int, -1, -1, 0, -1, -1>, 1>::operator() (long, long)	216.26	92.09	91.57
Eigen::DenseCoeffsBase<Eigen::Matrix <double, -1, -1, 0, -1, -1>, 1>::operator() (long, long)	121.69	47.15	46.65
Eigen::DenseBase<Eigen::Matrix<doub le, -1, -1, 0, -1, -1> >::Zero(long, long)	116.41	20.74	22.27

Figure 5: Information from gprof's call graph section- time of various functions

2.1.4.2 Hotspot locations

Gprof showed that FOC345() and FitQuartet() were taking a large portion of the time of the program, and so these functions were investigated further. A breakthrough came when the source code of FitQuartet was annotated using Cachegrind to show the number of instructions executed per line: one troublesome line in particular was causing 90.6 billion instructions, out of a total of approximately 2 trillion for the whole program. After profiling identified this hotspot, the code was refactored as described in section 2.2.1.

2.1.4.3 Architecture specification

When compiling, it is possible to specify to the compiler the architecture of the processor which will run the code. This prompts the compiler to tailor the machine code to the particularities of the actual processor. This project used the g++ compiler. The "-march=" flag specifies the architecture and the "-mtune=" flag specifies which architecture to tune for; that is which it should run best on (although it would still be functional when run on similar processors)(GCC team, 2014).

As shown in the below table, the mtune flag on the compiler-optimised original code yielded a small effect. It was not possible to compare the effect on unoptimised code, because architecture tuning is an optimisation.

	O3 no mtune	O3 mtune=core2	Oo no mtune
Overall Time (seconds)	502.34	496.15	6984.95

Figure 6: Time difference from architecture specification

The use of a different compiler flag, -march=native, was investigated, this time on the final modified code. This flag prompts the compiler to determine the CPU type at compile time and optimise for that type automatically. Using this flag, the total time of execution is 354.8 seconds, compared to 366.54 without it. So specifying the architecture for which to tune, either through "-mtune=[CPU type]" or "-march=native" may shave a few seconds off run time. However, this needs to be done during compile time, so if binaries are being created for distribution, several versions would need to be created for each architecture, and this might be confusing to the end-user.

2.1.4.4 Compiler optimisation

Several findings from this project demonstrate the optimisation power of the compiler. From the behaviour of the program versus the nature of the source code, it appeared that advanced optimisations were taking place. Profiling showed that certain changes expected to make a large impact on an unoptimised program often had little actual impact.

For example, several of the most time-consuming functions originate from the Eigen library, which is actually used in only two lines of the MinSqTree.cc source code. One line (see MinSqTree.cc line 61 in the original code) was mistakenly placed inside a loop in the original code, which was an error, but did not affect the semantics of the program (because it was calculating, and then overwriting, a value for each iteration when only the final value was needed). Yet, taking this line out of the loop sped up the program by only .5% when maximum compiler optimisation was enabled. Comparing unoptimised versions, there was a 22% reduction (1.136x speedup) by taking the expensive line out of the loop. The compiler appears to be correcting for some problems in efficiency automatically, and thus the impact of manual optimisation is sometimes minimal.

Results Part 2: Speedup Efforts

In addition to profiling before optimisation, other philosophies of optimisation include to do so iteratively, to combine profiling with intuition, and to use but also assist the compiler. Hints or pragmas can be used to suggest optimisations to the compiler, and assembly code can be inspected to verify that these optimisations have been implemented (Amarasinghe, 2010).

Once profiling identified targets for improvement, the task remained to implement changes in those areas. This section will cover refactoring, initial research into vectorisation, efforts to replace a function call with a hard-coded formula, testing, and final speedup.

2.2.1 Refactoring

Refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" (Fowler, 2000).

Sometimes what looks like sleeker code actually results in more instructions being executed after compilation. As mentioned in section 2.1.5.2, profiling showed that a large proportion of the instructions of the program originated from one particularly troublesome line of code, located in a loop within the FitQuartet function.

The line (MinSqTree.cc line 737 in the original code) is a conditional statement, specifically:

"if (ShortestLabel[si]!=ShortestLabel[sj])"

To understand what effect this line has, it is necessary to go over the semantics of the code at this point in the program. In context, this line is operating on a structure as in figure 7:

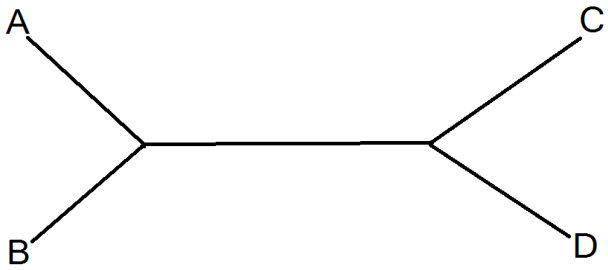


Figure 7: Four-leaved subtree

The above illustration actually shows the complete tree which has been reduced to a simple four leaf tree by collapsing the various species data into four groups: a, b, c, and d. The above line comes from a section of code that is working with the distance

between the gene found in species “si” with the one in “sj”. ShortestLabel[species number] outputs the leaf containing the input species, so the effect of the line is to check if the two species of interest are in the same leaf. If these species are in the same leaf, the distance between them is not relevant to finding the branch lengths of the simplified four leaf tree, and the code which follows can be skipped. Ironically this time-saving step takes a long time. Conditional statements introduce branching in the code, which can slow down execution due to branch misprediction. Branching also prevents vectorisation (Chesebrough, 2013). Instead of treating the label as unknown, progressing in a methodical manner removes the need to check for this case.

The code was changed so that each species (for which there is information) is placed into an array. The number of representatives of each leaf is counted. The array is populated such that species are sorted in order by leaf. Then several for loops are set so that a species is compared only to others not in its leaf. The code was also modified so that before a species can go into this new array, it is tested to see if it does indeed contain that gene. Because the array only contains species for which there is information, there is no need to check for “empty” species while within the loop. The modified code can be seen in the MinSqTree.cc file in the Final Code folder from about lines 730-820.

20% of the run time was reduced after these modifications. An additional 5% of run time was reduced by applying a similar change to another section of the code, in FOC345, which had originally used this same conditional statement.

The following figure illustrates the time of execution of the total program on DS1, comparing the original code, code after refactoring the loop in FitQuartet, and after refactoring the similar loop in FiveOptimCollection345:

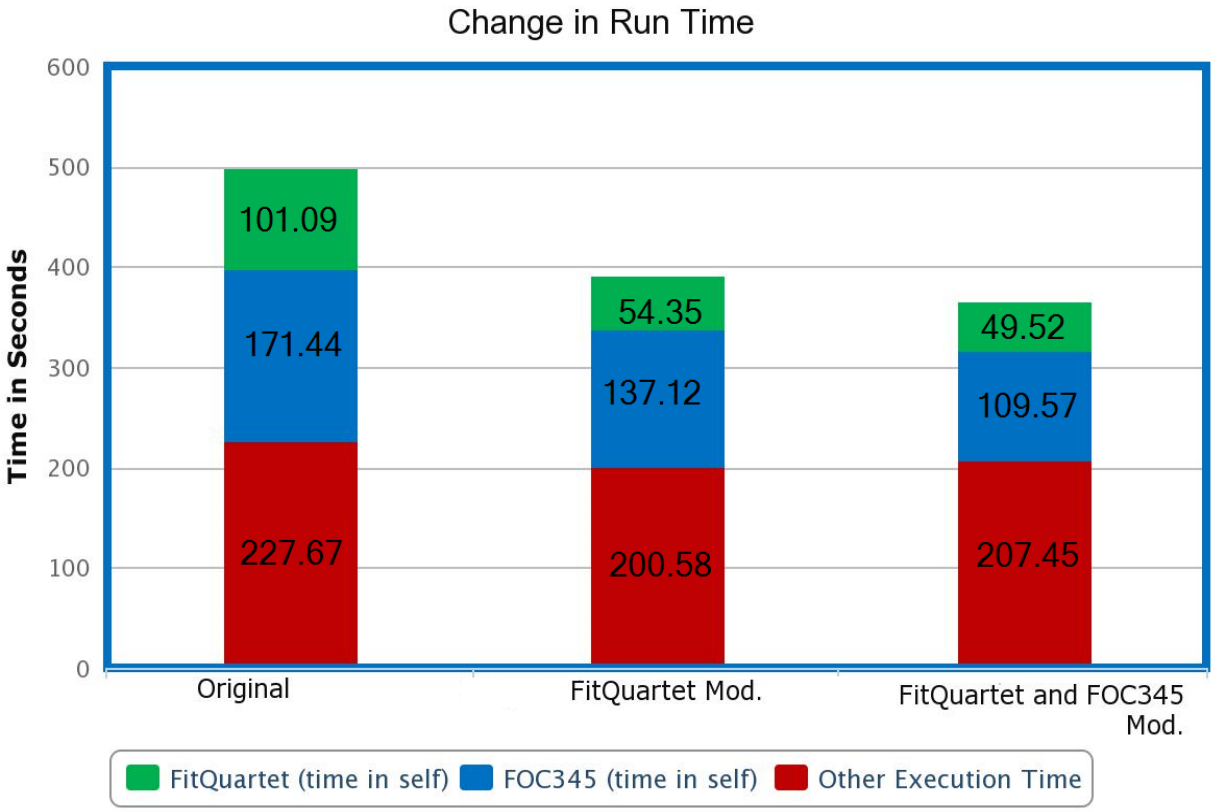


Figure 8: Graph of run time before, and after FitQuartet refactoring, and following additional refactoring of FOC345.

FOC345 was reduced when FitQuartet was reduced, which is logical as FOC345 calls functions which in turn call FitQuartet. After application of the same conditional-statement-to-for-loop change in FOC345, the reduction in time was not as dramatic, partially because FOC345 was already reduced when FitQuartet was reduced. Also, FOC345 is a longer function, with more occurring outside the for loop than FitQuartet. FOC345 is called 6,981 times, while FitQuartet is called 284,874 times, in both the original and modified versions. Thus the original loop in FitQuartet was causing a large time drain, and so the reduction in loop time would not be as significant in FOC345. Overall, this refactoring demonstrates that conditional statements in loops are costly, and that expanding source code can often shorten its run time.

2.2.2 Vectorisation

This section will cover the concepts behind vectorisation, testing vectorisation by examining machine code, and different approaches to vectorisation. At the outset of the project, the code's previous handlers indicated that likely targets for speed improvement would be vectorisation through the use of SIMD instructions, and inlining some functions. By the time this project was scheduled to end, vectorisation

had not been implemented, due to time constraints. Therefore any future fixes to the code would be wise to consider vectorisation as an avenue to speedup; hopefully, the following research will be instructive to those efforts.

2.2.2.1 Background on vectorisation concepts

Vectorisation is "the unrolling of a loop combined with the generation of packed SIMD instructions by the compiler" (Chesebrough, 2013). SIMD means “single instruction, multiple data” which is one of several forms of data parallelism.(Multiple instruction multiple data vectorisation also exists but is not widely used) (Vector Processor, 2014). Other forms of parallelism include loop parallelism, reduction for data aggregation, scheduling and data sharing. Parallelism is important, especially as processor technology advances. As the number of cores in the average processor increases over time, code which has a high degree of parallelism, even higher than that used at the time of its creation, will be most effective (Amarasinghe, 2010).

To understand how vectorisation works, it is necessary to consider the architecture of a processor chip. In the processor, there are several sets of registers, or spaces to store data of a certain size. Some of these are designated as data registers, others as instruction registers. During vectorisation, instructions are carried out on multiple data registers in parallel. In fact, on most modern CPU's, certain registers are designated vector registers (Vector Processor, 2014).

Vectorisation improves serial performance of CPU cores. Each core has its own set of registers and can do vectorisation in each core (Amarasinghe, 2010). Vectorisation operates on loop structures. Loops are vectorisable if they are: countable, single entry and single exit (a basic block), non-branching, innermost loops, without function calls for non-inlined functions. Non-contiguous memory access and data dependencies can prevent vectorisation from being implemented. Thus it is a good idea to practice data alignment, and use caution with pointers and loop indexing. Before vectorising, a developer should know the architecture of the target core, and the size of its vector registers. The length of a vector is usually the size of the vector registers divided by the size of the data type (This paragraph: Chesebrough, 2013.)

2.2.2.2 Examining assembly code

Because vectorisation is implemented on a low level, it is not visible by reading source code alone, and machine code should be checked. Examining .s files with actual assembly code proved to be too detailed to be useful within the time constraints of this project. See figure 3 for an example of this detailed code. However, to the trained eye this is a direct way to ensure that vectorisation is taking place.

2.2.2.3 Approaches to vectorisation

Research was done on the best way to vectorise. Two options were seriously considered (among many possibilities): 1) using SSE instructions and 2) using a concurrency

platform, such as OpenMP. One advantage of OpenMP is that it can be used to implement data and task parallelism, whereas vectorisation is only data parallel (Amarasinghe, 2010).

An alternative option would have been profile-guided optimisation, which is an option in Visual Studio. This is good for improving performance with minimal developer-involvement. Because this project is oriented towards developer involvement, it was inappropriate in this instance.

One vectorisation design choice is between explicit and auto-vectorisation. Auto-vectorisation is done by the compiler, and depends on the compiler being able to recognize certain patterns. The developer can nudge the compiler to vectorise by shaping their code to fit these patterns. Explicit vectorisation can take the shape of inserting SSE instructions into code.

2.2.2.4 Design choices for vectorisation

Optimisation occurs on a continuum from the lowest level of binary code to the highest level of fully automated compiler optimisation. Lower level optimisations can be architecture-specific and also specific to the current release of the software in which they were implemented. They are less portable and can be hard to maintain. Approaches to vectorisation, in order from highest to lowest programmer control are: assembler code, vector intrinsic, SIMD intrinsic, explicit vector programming, compiler auto-vectorisation with hints, and compiler auto-vectorisation (Amarasinghe, 2010).

Higher level optimisations may be more robust over changes to software and hardware, however they may not give the highest possible speedup. At the time of this writing, the most strongly considered option was OpenMP, a linguistic extension to C/C++, which exists as compiler pragmas. Concurrency platforms appear to be the best option in this instance, because, as Amarasinghe notes, "a concurrency platform abstracts processor cores, handles synchronization and communication protocols and performs load balancing" as "programming directly on processor cores is painful and error-prone" (Amarasinghe, 2010).

2.2.2.5 Ftree vectorizer report and analysis

The ftree-vectorizer tool provides a detailed report of which sections of the source code (including any libraries used) were vectorised by the compiler. If examining machine code is not practical, this tool provides another option for validation. Figure contains a small section of sample output.

```
...
MinSqTree.cc:690: note: not vectorized: too many BBs in loop.
MinSqTree.cc:690: note: not vectorized: Bad inner loop.
/usr/include/eigen3/Eigen/src/Core/DenseCoeffsBase.h:392: note: not vectorized: too
```

```
many BBs in loop.
MinSqTree.cc:687: note: vect_model_store_cost: inside_cost = 1, outside_cost = 1 .
MinSqTree.cc:687: note: vect_model_store_cost: inside_cost = 1, outside_cost = 1 .
MinSqTree.cc:687: note: Cost model analysis:
  Vector inside of loop cost: 2
  Vector outside of loop cost: 4
  Scalar iteration cost: 2
  Scalar outside cost: 0
  prologue iterations: 0
  epilogue iterations: 1
  Calculated minimum iters for profitability: 4

MinSqTree.cc:687: note:   Profitability threshold = 3

MinSqTree.cc:687: note: Vectorization may not be profitable.
MinSqTree.cc:687: note: LOOP VECTORIZED.
MinSqTree.cc:676: note: vectorized 1 loops in function.
...
```

Figure 9: Example section of ftree-vectorizer verbose=6 report for original code. BB denotes "basic block."

Most of the loops that were vectorised in the initial MinSqTree.cc were also vectorised in the final version, and also almost all that were unvectorised remained unvectorised. One main difference the reports show is that the newer code makes more use of the Assign.h file of the Eigen library, whereas the original code used GeneralBlockPanelKernel.h more often. It is unclear why this change was made or what it means, so it would merit further study.

The report provides only very short error codes to explain why some loops were not vectorised. The most common errors are "Too many BB's," meaning too many basic blocks, "bad inner loop" and "unhandled data reference". Most of these relate to branch-prediction, a key idea in auto-vectorisation. A basic block is a defined section of code with one point of entry and one exit. When there are multiple paths through a section of code, it is difficult for the compiler to predict its behaviour. In many sections of the code, there are nested loops, and many conditional statements. These conditional statements create branch prediction issues for the compiler (Chesebrough, 2013).

Vectorisation remains an excellent target for future speedup efforts for this program, and the ftree-vectorizer tool is a useful testing tool in this area. It would be beneficial to use its output to guide reduction of the use of conditional statements, and nested loops.

2.2.3 From general functions to direct calculations

As seen in the initial profiling in Figure 5, functions in the Eigen library account for a large proportion of the run time of the program. However, the Eigen library is only used for one thing: to provide an LU decomposition of a matrix, and solve it for a given vector. This action occurs in the line: "L=ZZ.lu().solve(BB);" which occurs twice in the source code (MinSqTree.cc line 61 and 1622). In this line, the matrix ZZ is decomposed into its LU form. Assuming the vector BB is the answer for the multiplication of ZZ by a different vector L, the system of linear equations is solved for L, using numerical values for ZZ and BB.

This formula is a generic one, yet the specific qualities of this system may reduce the number of steps needed to solve the system. In the semantics of the program at that point, the task is to minimize the sum of weighted least squares. The formula for the sum of weighted least squares is:

$$S=\sum_{i->j} W_{ij}(d_{ij}-\sigma_{ij})^2$$

where W is the weight, and d represents the observed data for the distance from I to j, and sigma is the unknown distance- in our case, the branch lengths. ZZ represents a matrix of combinations of weights. The vector B is a vector of combinations of products of the inverses of variances and distances.

One of the co-authors of the original code provided a direct formula for solving the system, via Gaussian elimination (Manuel Gil, personal communication, July 30, 2014). This method took advantage of the fixity of the matrix to rewrite the members of the L vector in terms of the combinations of the ZZ and BB terms to which they would be equivalent. This direct method was substituted for the Eigen call and tested. In the original output (again using DS1), about 8 fits are printed to the console, with a final error value of about 1.9x10^9. In the program, swaps are when rearrangements of the tree occur in an attempt to reduce the error. The program is designed to stop after a certain number of iterations with no swaps. After substituting the direct formula, the error values were much larger and the number of swaps did not tend to decrease. After the substitution the termination condition was not reached after several hours and the program was terminated by the developer as it appeared a problem had been introduced.

Numerical stability is a principle of numerical analysis which may explain the unexpected behaviour of the direct formula. Thought of simply, rounding errors accumulate during operations involving fixed-representation numbers which vary widely in value (Hosking, 1996).

Because numerical instability was suspected as the cause of the increased error, an attempt was made to create a matrix which would be, effectively, sorted. Sorting branch lengths was investigated as a proxy for sorting the matrix. Rearranging the tree so that branch lengths are sorted in order from small to large prior to solving the matrix did not solve the problem, nor did sorting them in order from large to small. There remained

high error values, and non-decreasing swapping.

Numerical analysis is relevant to much of scientific computing. Numerical stability is a subtle concept, but it should be understood to understand where exactly this substitution effort went wrong, and may ultimately be corrected.

Further research proved that partial pivoting was likely the key step which preserved the numerical stability in the original code.
Rounding (or chopping) errors are inherent in floating point arithmetic (Hosking, 1996; p. 14). Gaussian elimination involves "(n^3+3n^2-n)/3 multiplications/divisions and (2n^3+3n^2-5n)/6 additions/subtractions" for a system which has n unknowns. (Hosking, 1996; p. 52)" therefore rounding errors add up quickly.

Partial pivoting rearranges a matrix to be used in Gaussian elimination so that the largest element in each column is the pivot element (on the diagonal). As the system of equations is solved, each equation is multiplied or divided and added or subtracted to the one above it. As each equation's solution eliminates a variable, the focus of arithmetic moves in a diagonal fashion. If the pivot element (the diagonal one) is the largest, its relative error dominates, which is acceptable if other values are small. Hosking (1996, p. 48) puts it best:

> If a pivot element is small compared with the elements in its column which have to be eliminated, the corresponding multipliers used at that stage will be greater than one in magnitude. The use of large multipliers in the elimination and back-substitution processes leads to magnification of round-off errors, and this can be avoided by using partial pivoting.

The Eigen lu() function creates an LU decomposition form of the input matrix, and the solve function used performs partial pivoting prior to its Gaussian elimination. The substituted code did not account for partial pivoting. (Eigen, 2010).

Sorting via branches had no effect on the arrangement of pivot elements within the matrix, and so the attempt at branch rearrangement was irrelevant to the numerical stability of the system.

At the time of this writing, the original Eigen method is in place. Although it takes significat time, it remains accurate. Future speedup efforts may reexamine this issue.

2.2.4 Testing

2.2.4.1 Output

The program was tested by determining if the same output files and console status statements were produced before and after modification. In keeping with iterative design philosophy, the program was tested after each significant change to the code.

Please see the appendix for output of the original and modified code (which are the same).

Below is a table of distance fit values for various iterations of the code

Code Version	Original	After substituting direct formula (sampled after 20 iterations)	Final Code
Distance Fit	1.89816e+09	2.23e+09	1.89816e+09

Figure 10: Distance Fit values for different versions of the code.

2.2.4.2 Speedup

Speedup is a measurement of the effect changes made to the program have on the time of execution, and is an important metric for the success of speed optimisation.

The general formula for speedup is (Amarasinghe, 2010):

Speedup=(time of original program)/(time of modified program)

2.2.4.3 Time

The total program time as measured by gprof was used as the standard measurement of run time. Tests were run on the same computer, Hepatitis in the Darwin Building.

2.2.5 Final Speedup

The main purpose of this project was to decrease the run time of the program. Following all changes, the total speedup accomplished in this project is:

Speedup=(500.20s/356.70s)=1.40X

Chapter 4
Conclusions and Things Learned

This project was challenging because I previously had little experience in this type of computer science, in general, and no knowledge of vectorisation or optimisation techniques in particular. I had a brief overview of assembler code in a Computer Architecture and Hardware course, but have not worked extensively on a low level before, so had much to learn. Furthermore, it required knowledge of other areas, such as biology (which I had studied before), and linear algebra and numerical analysis (which I had not).

As stated in the introduction, one main aim of the project was to accomplish learning goals. The code is written in C++, which I had covered outside of UCL many years ago, so review was necessary (for working with pointers and namespaces especially). I also learned some more about assembly language. Profiling as a concept was new to me before this project and I think I now understand its importance, methodology and current tools. My understanding (and appreciation) of compilers was also expanded during this project. Many other best practices and skills were added to my toolkit, such as gnu screen, working with TUIs, networking and setting permissions, and understanding the importance of setting up a functional and comfortable environment.

Much of the work of this project consisted in learning underlying existing systems: for example, from working with the git version control system (and version control philosophies in general) to understanding the semantics of a several-thousand-line codebase. I also learned a lot about working on the Linux terminal, through much trial and error. For example, it took a day to realize the path the Eigen library didn't match the local path, find what to set it to and find how to reset it. One particular challenge was the research into linear algebra required to understand the least squares calculations. However, it appears linear algebra and numerical analysis are important to

much of scientific computing and I am glad that this project has given me exposure to these disciplines.

Certain things were not implemented in this project, but the research I did on them has increased my knowledge in computer science, such as: data alignment, loop unrolling, and other approaches to low-level optimisation such as bit hacks.

While optimising, one approach is to make a change and then profile again, and measure any potential speedup. This can be done somewhat blind, in terms of not knowing ahead of time the effect on speedup; any effect on the code semantics of course must be known, or the changes should be avoided. Understanding the execution path of code you did not write is time-consuming but often the only way to begin to improve existing code. However, if one works on functions independently so that the input and output remains the same, it may not be necessary to fully understand the code base as a whole. This approach was generally not used in this project, but is an option for exploration in future speedup efforts. In general, undertaking the painstaking process of teasing out the control flow and semantics of the code remains the only way to refactor without breaking the code.

Overall, the efforts expended this summer did lead to meaningful speedup. Of course, there is still ample room for improvement as well. However, given the time limitations and the difficulty in working in a new environment on a large piece of unfamiliar code, I believe the 1.40x speedup represents a significant accomplishment. Furthermore, the learning objectives attained represent a significant accomplishment in themselves.

Chapter 5
Possibilities for Extension and Other Reflections

There are many things that could be done with this code base, both in terms of increasing the speed and increasing the usability of the finished project. To increase the speed, resolution of the numerical stability issue and correct implementation of the direct formula replacement would likely yield good results. Also, implementation of vectorisation would be a great next step. To begin vectorisation, it would be useful to examine the ftree-vectorization report, and then examine the data dependencies of the code in-depth. Then vectorisation could be implemented either with SSE intrinsics or a concurrency platform.

Following the changes made by refactoring FitQuartet and FOC345, a surprising finding was that there is a time reduction of 20.22 seconds which is not accounted for in the two altered functions. This would be an interesting area for further exploration. Perhaps there was a gain in overall branch prediction. Perhaps some loops outside the functions are now being vectorised, and so it may be elucidating to look at the vectorisation reports in more detail.

I did lose some time trying to investigate the vectorisation status of the Eigen library. The results of perf_annotate were somewhat misleading because they included some confusing Eigen source code, which caused the loss of a fair amount of time trying to fix something that wasn't truly a problem. Some of the source code seemed to indicate that Eigen was not vectorising because it contained the keyword scalar ("EIGEN_STRONG_INLINE Scalar&" [Guennebaud, 2010]). However the results of ftree-vectorizer, show that vectorisation was being implemented by the compiler even on the original code (given that it was enabled through the use of an -O2 or higher flag).

An area of refactoring which may be productive to explore is bit-hacking. Bit-hacking involves using bit-wise operations to avoid memory accesses, which have a high overhead. For example "(r=y^((x^y)&-(x<y));" is all done in the CPU. A compare instruction using if would have changed the program counter which is expensive (Amarasinghe, 2010). If statements implemented as masked assignments avoid branching (Chesebrough, 2013). Some bit-hacks had been applied to this program prior to this project, and additional implementations would be a productive area of improvement in the future.

Many times in the program, subtrees are proposed and rejected or accepted based on their error scores. In the early stages of fitting, the errors can be quite large. One possible way to increase the speed of the program could be to decrease the precision (size of representation) of the values used in early calculations. To preserve accuracy of the program, once an approximation was achieved, the error should be recalculated on higher precision values to match the original behaviour of the program. To determine when larger precision is necessary, the difference between the current and previous topology errors could be stored. A ratio of this difference to the precision of the numbers involved would be calculated and compared to a standard ratio. If below it, a separate function could be called using the higher precision numbers, with the newly produced rough topology as a parameter to provide the beginning framework, and then the output of this could maintain the precision of the original code in its final results.

To increase the usability of the project, it would be useful to post binaries on a publicly accessible website with instructions for use (if further low-level optimisation is implemented, including architecture specification flagging, it may be necessary to create multiple files for various common chip architectures). If these files were placed on the web, it would be natural to create FAQ and help pages for the software. As an alternative to downloadable copies of the project, a web-hosted interface may be more likely to be used as the barrier to use would be lower (no need to download).

I learned a lot from this project and hope the speed improvements gained will help to contribute to the wider use of the larger Trees of Life project.

Bibliography

Albert, D. *Understanding C by learning assembly*. Retrieved June 24, 2014, from https://www.hackerschool.com/blog/7-understanding-c-by-learning-assembly

Amarasinghe, Saman, and Charles Leiserson. *6.172 Performance Engineering of Software Systems, Fall 2010*. (MIT OpenCourseWare: Massachusetts Institute of Technology), http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2010 (Accessed 18 Jun, 2014). License: Creative Commons BY-NC-SA

Clements, A. (2006) *Principles of Computer Hardware*. Oxford, United Kingdom: Oxford University Press.

Chesebrough, B. (2013) *Performance Essentials OpenMP 4 Vectorization.* Retrieved June 20, 2014, from https://software.intel.com/en-us/articles/performance-essentials-with-openmp-40-vectorization

Disassembler. (2014, July 28). In *Wikipedia, The Free Encyclopedia*. Retrieved 12:52, August 22, 2014, from http://en.wikipedia.org/w/index.php?title=Disassembler&oldid=618792250

Distance matrices in phylogeny. (2014, May 31). In *Wikipedia, The Free Encyclopedia*. Retrieved 12:26, August 29, 2014, from http://en.wikipedia.org/w/index.php?title=Distance_matrices_in_phylogeny&oldid=610970877

Dowd, Kevin & Severance, Charles R. (1998) *High Performance Computing, Second Edition*. Sebastopol, United States of America: O'Reilly & Associates, Inc.

Fowler, Martin. (2000) *Refactoring: Improving the Design of Existing Code*. Boston, United States of America: Addison-Wesley.

Gentoo Wiki Editors. *(*29 July 2014) *GCC optimization.* Retrieved,14 August 2014, from http://wiki.gentoo.org/wiki/GCC_optimization

GCC Team (2014-06-29.)*Auto-vectorization in GCC*. Retrieved Aug 30, 2014 from https://gcc.gnu.org/projects/tree-ssa/vectorization.html

Graham, S, Kessler, P and Mckusick, M. 1982. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*(SIGPLAN '82). ACM, New York, NY, USA, 120-126. DOI=10.1145/800230.806987 http://doi.acm.org/10.1145/800230.806987

Guennebaud, G., Jacob, B. and others, (2010). Eigen v3.

Hosking, RJ, Joe, S, Joyce, DC, and Turner, JC. (1996) *First Steps in Numerical Analysis*. London, United Kingdom: Arnold.

Hot spot (computer programming). (2014, February 2). In *Wikipedia, The Free Encyclopedia*. Retrieved 14:43, August 26, 2014, from http://en.wikipedia.org/w/index.php?title=Hot_spot_(computer_programming)&oldid=593529943

Morse, H. Stephen. (1994) *Practical Parallel Computing*. Cambridge, United States of America: AP Professional.

Perf Wiki Main Page. (2014) Retrieved August 19, 2014 from https://perf.wiki.kernel.org/index.php/Main_Page

StefanLohn (2014, May 28) Vectorize Source-Code. Retrieved July 18, 2014, from https://twiki.cern.ch/twiki/bin/view/LHCb/VectorizeSource-code

Strang, Gilbert. *18.06 Linear Algebra, Spring 2010*. (MIT OpenCourseWare: Massachusetts Institute of Technology), http://ocw.mit.edu/courses/mathematics/18-06-linear-algebra-spring-2010 (Accessed 16 Jul, 2014). License: Creative Commons BY-NC-SA

Valgrind Authors. Valgrind Documentation, Release 3.8.0 10 August 2012 Copyright © 2000-2012.

Yang, Ziheng. (2014) *Molecular Evolution: A Statistical Approach*. Oxford, United Kingdom: Oxford University Press.