UNIVERSITY OF HERTFORDSHIRE
School of Computer Science

Modular BSc Honours in Computer Science

6COM0282 Computer Science Project

Final Report
April 2013

Work-Stealing Scheduling Techniques Applied to Computing the Mandelbrot Set

M.J.Hawes

Supervised by: Colin Egan

Abstract

Load-balancing of parallel computer programs is a useful means of improving performance. It also serves to remove some of the burden of designing a fairly distributed work-load from the programmer. Work-stealing is a technique used to implicitly balance work-load across multiple processors throughout run-time. In this report the technique is explored and applied to an algorithm to compute a raster plane of the Mandelbrot set.

A Randomised Work-Stealing Algorithm which utilises a non-blocking double ended queue is implemented and successfully applied to a variation of the level-set approach to computing the Mandelbrot Algorithm.

The report concludes with a number of findings which suggest that the work-stealing approach offers significant benefits such as; a performance increase versus an un-balanced parallel approach, good scalability as problem size increases, and a fair approach to choosing where work to be migrated is sourced. In addition a number of weaknesses of the approach are identified.

The investigation also encompasses some exploration into fractals and the Mandelbrot Set as a means of further understanding the problem domain.

Acknowledgements

The author would like to acknowledge Colin Egan for his invaluable guidance throughout this project, and also for his encouragement and great enthusiasm. He has been a resource for both inspiration and the ideas explored in this report, as well as a patron for sound technical and academic advice. Thank you for all the enlightening weekly chats about all things computer science and beyond.

Contents

Chapter 1

Introduction

Multi-processor computers are now commonplace for both consumers and scientists alike, in replacement of uni-processors. This shift in architectural design demands a drastic shift in how programs for such platforms are constructed, from the standpoint of both the programmer and the language designer. Designing effective programs to utilise such hardware, using well established programming tools with sequential roots, is often difficult and riddled with pitfalls. Popular languages such as C, Java, and C++ require a good knowledge of multi-threaded algorithms, as well as a deeper understanding of the problem domain to produce a highly effective design. The programmer may be forced to sacrifice potential performance gain for a more manageable level of design complexity, or worse still; avoid writing parallel code all-together in the interest of robustness.

Techniques to implicitly optimise run-time performance, with a minimum impact on complexity of parallel program design, are useful for more easily producing robust implementations. One aspect which allows for such optimisation is run-time work-load balancing.

This project explores a branch of load balancing techniques known as work-stealing. This is applied to an algorithm which computes an approximation of the infinite mathematical set known as the Mandelbrot set.

Motivation

The author has a strong interest in high performance computing and tools which provide powerful, abstract interfaces with underlying multi-processor architecture. Previous work on the Single Assignment C [9] and S-Net [8] programming languages has served to inspire, and further the desire for knowledge of the subject area.

This project is motivated by the aspiration to better understand how modern computer architectures can be optimally utilised, and communicate these findings in a concise, comprehensive report.

In addition the author wishes to gain valuable experience of the project development and management process.

Aim

To investigate, through research and practical implementation, the effectiveness of work-stealing on a parallel algorithm which computes an approximation of the Mandelbrot set.

Objectives

This section outlines the deliverable items that are presented in this report. Core objectives are primary and more vital, advanced objectives are additional items.

Core Objectives

1. **Background Research**
2. **A Sequential Mandelbrot Set Algorithm**
3. **A Naïve Parallel Mandelbrot Set Algorithm**
4. **A Random Work-Stealing Mandelbrot Set Algorithm**
5. **Analysis of the Implemented Algorithms**

Advanced Objectives

6. Graphical Output
7. A Render Thread Work-Stealing Mandelbrot Set Algorithm
8. Work-Stealing Trace System

Achievements

The following table shows each objective and its completion status at the time this report was produced.

Objective	**Status**
1	Complete
2	Complete
3	Complete
4	Complete
5	Complete
6	Complete
7	Partially Complete
8	Complete

Table 1.1: Table of Achievements

Report Structure

There are five remaining chapters in this report. In chapter two a detailed review of the problem background is presented. Chapter three details the practical implementation, which this report is based on, and the manner in which it was carried out. Chapter four offers an evaluation of the project as a whole. This includes analysis of the implemented software and a comparison of the implemented schemes. Chapter five lists the resources referenced in this report.

In chapter six the appendices are presented. A glossary of terms is presented as Appendix A. This consists of a list describing major concepts surrounding the field of study, and directs the reader to their page of first occurrence. Words emphasised in bold typeface indicate the first appearance of a glossary term. Some example trace output is listed in Appendix B. The source code for all software developed to fulfil the objectives described above is presented as Appendix C.

Chapter 2

Background Research

2.1 Run-Time Scheduling Techniques for Multi-Threaded Computations

This section briefly describes the problems associated with **scheduling multi-threaded algorithms** at run-time and the major paradigms that have surfaced.

To efficiently utilise a parallel computer architecture it is desirable to minimise the amount of time a processor spends idle or performing other logistical tasks, i.e not doing work. When a computation's concurrent sub-tasks (or **threads**) incur a regular cost in processor time, each processor can simply have the same amount of work assigned to them. When the computation has more irregular or dynamically growing sub-tasks, a problem arises resulting in processors becoming idle while others still remain working. The solution to this problem is referred to as **load-balancing** and can be described as a form of dynamic scheduling, that ensures each processor spends approximately the same amount of time working. This means processors generally spend less time idle, however have to deal with scheduling overheads as a trade-off.

When considering the scheduling of multi-threaded computations, two major load balancing techniques have been used. These are **work-sharing** and **work-stealing**.

- **Work-Sharing:** A processor which creates new work attempts to migrate it to another underutilised processor at creation time.
- **Work-Stealing:** A processor which is starved of work attempts to "steal" work from other processors.

Both techniques intend to promote balanced work-load across all processors, however in Work-Stealing the frequency of work migrations is lower. When all processors have a high work-load and no need to "steal" this becomes useful because threads need not get migrated at all. With work-sharing work migration occurs each time new work is created [13]. This also suggests that work-stealing promotes better **locality** and grouping of sub-tasks, as spawned work stays with the same processor until stolen.

2.2 The Mandelbrot Set

The Mandelbrot set is a set of complex numbers which when plotted produce a spectacular and recognisable shape as illustrated in figure 2.1. It is often presented as a colourful and striking image and has been described by some as the most beautiful object in all of mathematics [19, p. 234]. The **complex-numbers** that comprise the set are closely related to julia-sets. In-fact the Mandelbrot set can be described as a catalogue of Julia Sets which, when plotted, all points are connected forming a single unbroken shape [12, p. 177].

The set is named for the mathematician Benoit Mandelbrot, who discovered it in 1980 [12, 25]. He was a pioneer in the study of fractal geometry and also coined the term **fractal**, of which both the Mandelbrot set, and Julia sets are examples of.

In this section I will give a more detailed explanation of the areas mentioned here.

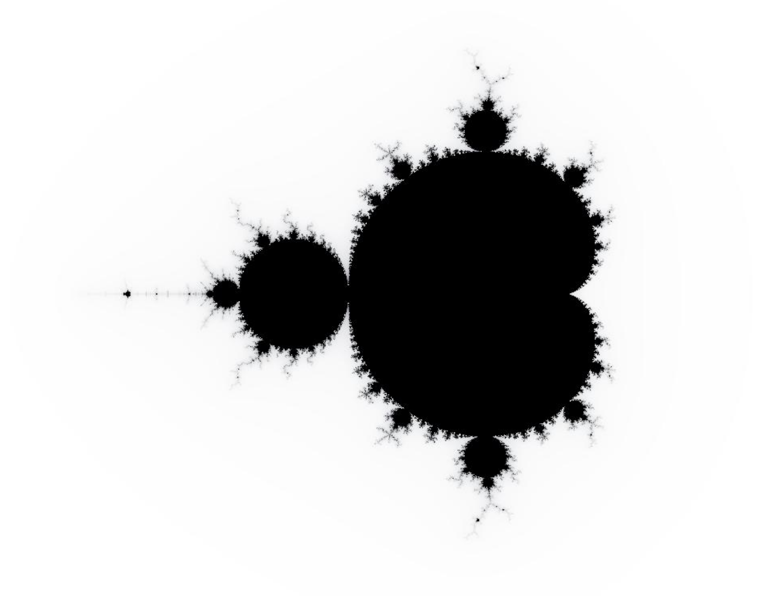


Figure 2.1: A rendering of The Mandelbrot Set generated using the program "fraqtive" [2].

Fractals and Self-Similarity

A fractal is a means of describing shapes which are more complex than a Euclidean shape. The leaves of a pine tree or the forks of a lightning bolt are obvious examples of real things that fractals allow us to more faithfully describe. These **real-world-fractals** are similar to **mathematical-fractals**, of which the Mandelbrot set is an example, but differ in that they do not display the property of **scale-invariance**.

Fractals have a fractional dimension. Unlike shapes with topological dimension, for instance a two dimensional square, a fractals dimension is of a non integer value.

A property of fractals (but not all) is **self-similarity**, where the shape is comprised of smaller "copies" of itself. This is known as **exact self-similarity** and means the shape is identical at any scale. A well-known example of this is the Triadic Koch Snowflake which is a fractal constructed using equilateral triangles. It is important to note here that The Mandelbrot set does not quite show the same property, it is said to be **quasi self-similar**. This means the shape is approximately similar at all scales, in that the shape is replicated but in a slightly distorted form with each "copy".

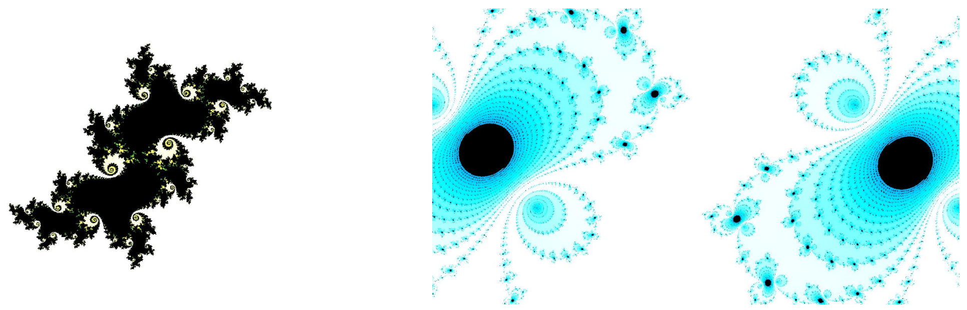
It turns out there are many rather useful applications for fractals. To name a few; computer graphics used in video games and film (notably Star Trek II: The Wrath of Khan [12, p. 8]), military hardware design [17], and measuring the length of a coastline [23].

Julia Sets

To understand the basis of the Mandelbrot set it is first necessary to understand it's relation to julia-sets. The function in equation 2.1 is iterated infinitely where c is fixed. The filled julia-set is comprised of all values of z_0 where the result is bounded and does not tend towards infinity. The julia-set is comprised of those members of the filled julia-set which lie on the boundary [19]. In the interest of keeping this report readable, and because filled Julia Sets are more relevant, filled julia-sets will be referred to simply as julia-sets.

$$f(z) = z^2 + c \tag{2.1}$$

The Mandelbrot Set is related to julia-sets in which the values c and z used are expressed as a complex-number. Figure 2.2 illustrates some examples of such sets.



i: The julia-set where $c = -0.1 + 0.649i$

ii: The julia-set where $c = -0.75 + 0.03i$

Figure 2.2: Two Julia Sets rendered using the program “fraqtive” [2]. Figure 2.2i is a member of the Mandelbrot set, figure 2.2ii is not.

The Mandelbrot Set and How it is Computed

The set is comprised of those julia-sets which are connected. In order to determine whether a Julia Set possesses this property, we need only compute the result for z_0. If this tends towards infinity the value c is not a member of the Mandelbrot Set. If the result is bounded, then it is a member. This is known as the **critical-orbit** and is useful because it means we do not have to compute the entire Julia Set for each value of c.

So the Mandelbrot set can be computed by iterating all possible values of c for the function in equation 2.1 where z is the critical-orbit. Because the set of all possible values of c is infinite, and computers have a finite amount of resources, this set needs to be approximated. This can be done using a raster plane which takes samples of the complex plane at regular intervals.

There are many algorithms including the Level set method [12, p. 188] and Continuous Potential Method [12, p. 191]. The former uses a raster plane to produce an approximation as described above, and the latter produces a smooth surface representation.

Listing 2.1 describes a variation of the level set method in pseudo code. The algorithm is simple and provides a gradient effect of equipotential curves on the output image which is visually pleasing.

```
compute_mandelbrot()
    FOR y = 0 TO height - 1 DO
        c_im := im_min + y * (im_max - im_min)/(height - 1)
        FOR x = 0 TO width - 1 DO
            c_re := re_min + x * (re_max - re_min)/(width - 1)
            output_plane[x][y] := level_set(c_re, c_im, max_iterations)
        END FOR
    END FOR
END

level_set(c_re, c_im, max_iterations)
    z_re := c_re
    z_im := c_im

    FOR i = 0 TO max_iterations DO
        IF( z_re^2 + z_im^2 > 4) THEN
            RETURN i
        END IF

        tmp_im = 2 * z_re * z_im + c_im
        z_re := z_re^2 - z_im^2  + c_re
        z_im := tmp_im

    END FOR
    RETURN max_iterations
END
```

Listing 2.1: A sequential algorithm to compute the Mandelbrot Set

2.3 The Work-Stealing Technique - Described in Depth

As described above, work-stealing is a load-balancing technique which allows work starved processors to acquire scheduled work from other processors.

Each processor has a number of assigned tasks to complete. In general, a processor acquires its work from here. However, once these tasks are exhausted, the processor becomes a **thief** and a **victim** is chosen to steal from. The method used to choose a victim is implementation specific, for instance some implementations adopt a random scheme [10,13,22]. If the processor successfully steals work it relinquishes its thief state and returns to doing work. If the steal attempt is unsuccessful, for instance when the victim has no work or is blocked, the processor tries again until it is determined that there is no work remaining in the entire network.

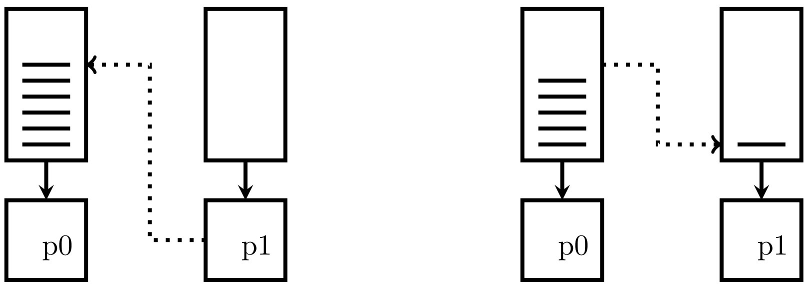
Figure 2.3 illustrates the result of a successful steal operation in which work-starved processor *p1* transfers a piece of work from *p0's* work list to its own.

Research has been conducted to explore its application in programming languages such as Cilk [13], parallel programming libraries such as Hood [14] and Factory [30], and large scale heterogeneous systems such as computational clouds [20].

This section explores some schemes used to implement work-stealing in various settings. It is focused on overall design and techniques presented in related literature.

2.3.1 Blumofe and Leiserson - A Randomized Work-Stealing Algorithm

This scheme is geared towards computation of dynamically growing, fully strict, multi-threaded computations and is applied to the CILK programming language and its runtime system [13].



i: A steal request by the thief processor *p1* on victim *p0*.

ii: The attempt was successful and the work was re-assigned to *p1*.

Figure 2.3: A successful steal operation between a thief and its victim.

Each thread maintains a **ready-deque**; a double-ended queue of work waiting to be processed. Accesses to this queue are made either at the top (for a steal operation), or at the bottom (for a push operation or when the next piece of work is required). A thread becomes a thief when its ready-deque is empty and randomly selects a victim; a thread to attempt to steal work from. If this **steal-operation** is successful it pushes the stolen work onto the bottom of its ready-deque and becomes a worker again. If not it tries, at random, to find another victim.

The ready-deque can be implemented in such a way that a thread need not be stopped in order for a steal operation to occur. This property is known as **non-blocking** and only requires that the top end of the deque has atomic access, while the bottom can freely be accessed by the thread which owns the deque [10]. This is useful because it reduces the overheads of a steal operation in that a working thread generally does not get interrupted. Further still, a non-blocking deque can be made more efficient through use of a **circular array** [15].

Because of the setting this scheme is designed with in mind, the algorithm needs to consider that a piece of work can spawn children dynamically, which it may depend on completing to continue. When computing the Mandelbrot set in a concurrent environment, no such consideration is required as each point can be independently processed.

2.3.2 McGuiness - Render-Thread Algorithm

This scheme is presented as part of McGuiness' Masters Thesis [26], and is suited for parallel computation where each unit of work is independent from any other and can be indexed in a list. Computation of the Mandelbrot set is given as an application of the algorithm.

The algorithm uses a set of **worker-threads** (referred to by McGuiness as render-threads), and a single **monitor-thread**, to control the distribution of work. Each worker-thread is initially given an equal share of the overall work-load before starting.

Each worker-thread maintains an estimated completion time for its assigned work-load. This is initially set to the maximum possible value and is iteratively refined by calculating the average time taken to complete a piece of work. This metric is used as a policy for deciding which thread is the most suitable candidate for the victim of a work-stealing operation.

When a worker-thread completes its assigned work a work-completed signal is generated, its estimated completion time is set to *0*, and the thread is stopped. This thread will be referred to as the thief. When the monitor thread detects this signal, it searches for the worker-thread with the longest estimated completion time, which will be referred to as the victim. The monitor-thread waits for the victim to complete its current piece of work before stopping it. Its workload is then halved, having the other half re-asigned to the thief. Both the victim and the thief are restarted and continue doing work. The monitor-thread returns to waiting for another work-completed signal and the process is repeated until no work remains.

Discussion of the Presented Schemes

The key difference between the Render-Thread Algorithm and the Randomized Work-Stealing Algorithm is that a worker-thread does not maintain a queue of work, but simply has a range of indices assigned. This all but eliminates the overheads associated with initialising and maintaining a potentially costly data-structure, but makes a non-blocking implementation difficult. It also requires a monitor-thread to manage work migration which reduces the maximum number of threads performing work by one. Another limiting factor is that only one thread may perform work-stealing at any given time. The Randomized Work-Stealing Algorithm has no such limitation.

2.4 Tools

This section discusses some of the programming and general tools which were considered for use in the implementation and for the good of the project as a whole. The tools discussed in this section are all viable options for a Linux platform.

Programming Languages

- **C:** A general purpose imperative language which is very widely used. It is statically typed but weakly enforces type errors and allows low level access to memory, which makes the programmer responsible for reclaiming used memory. This allows the programmer a high level of control but reduces type safety. It is not object oriented and lacks many features of more modern high level language features.

  C is well suited for implementing efficient, high performance programs due to its relatively low level of abstraction compared to more modern languages. It is often associated with system programming because of this. A number of parallel programming libraries are available amongst other useful libraries.

  GCC is an open source, industry standard compiler and is freely available. It is well documented and supported.

- **C++:** A general purpose multi-paradigm language inspired by C. C++ adds object oriented programming (amongst other features) to the majority of C's syntax and features. This makes it more suitable for implementing larger application systems with the same benefits of low-level memory manipulation that C boasts.

  Some useful features include inheritance of types, dynamic polymorphism, and templates (including an extensive library of useful template classes). A number of parallel programming libraries are available and are similar if not the same as the equivalent C versions.

  G++ is a variant of the GCC compiler and shares a number of its properties.

- **Java:** Primarily an object oriented language which borrows features from other paradigms. It is highly portable because it targets java bytecode which runs on the java virtual machine, available on most platforms. It is heavily influenced by C and C++ but offers few low-level facilities available in said languages. For instance pointers are not available and a garbage collector is used to move the responsibility of memory management from the programmer, to the run-time system. Although this is a useful tool, the consequence is a considerable run-time overhead.

Java is more suited to application programming than the likes of C and C++. Its extensive API offers concurrent programming classes.

An open source compiler suite is available as well as a propriety version. These are both widely used and well documented.

Parallel Programming Libraries

In order to implement Work-Stealing, support for programming threads with a suitable level of control is required.

- **POSIX Threads (pthreads):**
  Provides low level manipulation of threads for the C programming language [11]. It is a library based on IEEE standard 1003.1. Thread programming is achieved by specifying a function in which to run in parallel. Thread synchronisation is supported through use of a set of functions and data structures provided; such as **mutexes**, **barriers**, and **condition-variables**.

- **Open Multiprocessing (OpenMP):**
  Provides abstract thread programming interfaces for C, C++, and fortran. In general OpenMP only allows coarse grained manipulation of threads through features such as parallel loops [21].

- **Java Threads:**
  An object oriented approach to multi-threaded programming. Java provides a Thread class which the programmer can specialise to perform the desired task. Synchronisation is generally done using implicit locks through use of the synchronised modifier.

Graphical Output

For the purpose of demonstrating that the program correctly generates a raster plane of the Mandelbrot set, a graphical representation of the plane is output.

- **PPM Output File:**
  The simplest option is to output to a Portable Pixel Map (PPM) file. It is text-file based and easy to implement but produces rather large files. The process of outputting to a text file is inherently serial in nature, so with large image resolutions processing takes a long time. There is support for both grey-scale (Portable Grey-scale Map format) and colour images. The advantage of using this method is the portability. No libraries or extras are required and most image viewers will read the file. [7]

- **GNU Plot:**
  A graph plotting package available for multiple operating systems. Supports screen display or file output of both 2d and 3d graphics [5]. There are programming interfaces available for various languages such as C [18], C++ [6], and Java [1]. GNU Plot needs to be installed on the machine in order to use it.

- **OpenGL - glut:**
  Glut is a framework for providing simple, cross-platform, GUI window control in conjunction with OpenGL. Bindings are available for various languages, including C and C++ [4]. A free implementation called “freeglut” is available and can be installed on linux [3]. This method requires OpenGL and an implementation of Glut be available on the machine that the program is run on.

Chapter 3

The Development and Implementation

3.1 General Design and Practices

Development Process Model

A feature driven development (FDD) style approach to development is adopted for this project. FDD is a branch of project management derived from the school of agile process models [27]. It is almost ideal for this project due to the small scale, short product life-cycle, and the flexibility needed when requirements change as the project transpires.

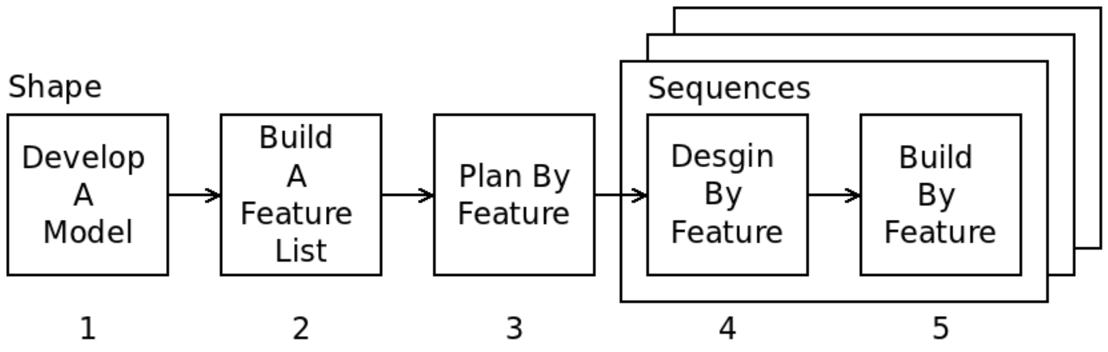


Figure 3.1: The five stages of the feature driven development cycle. The overlapping rectangles behind stages four and five indicate sequences that can be implemented concurrently. The diagram is derived from material presented here [16, p. 90].

Some of the notable features of FDD which make it attractive for such a project are described as follows:

- Short, iterative cycles of development.
- Working results delivered by each iteration.
- Flexibility for developers and clients alike, allowing for easy adaptation to changing requirements.
- Product quality is emphasised at each step.
- Development stages may overlap, i.e. individual features can be produced concurrently.

Major Tool Choices

The implementation is realised using the C programming language, the pthreads library, and PPM files for graphical output. These choices are based on the tools examined in section 2.4.

The C programming language is chosen for its relatively low-level of abstraction and level of control over memory management[1]. It is preferable over C++ and Java because features such as object oriented programming are deemed unnecessary run-time overheads. The implementation is relatively small-scale, so C will suffice in terms of managing the code complexity.

The pthreads library is selected for its low-level control compared to the likes of OpenMP and Java Threads. The implementation of the Randomised Work-Stealing algorithm only requires control of simple thread synchronisation (i.e. mutexes) and nothing more.

The text based PPM file format is chosen for its simplicity and portability. Producing a graphical representation of the Mandelbrot set is not the primary purpose of this project, but is still a desirable feature for verification and demonstration purposes.

Other Tools Used

Listed here are the major utility tools which have been used to make this project of a better over-all quality.

- LaTeX:
  A document preparation mark-up language which produces professional and consistent documents.
- **GNU Make:**
  A tool to aid quick and easy building of a project. It uses a series of rules and dependencies to determine the order in which a project can be built, which are described in an accompanying Makefile.

  This is useful for producing an executable from source code, as well as compiling a LaTeX document in conjunction with bibtex to produce a pdf.
- **Git:**
  A distributed version control system which provides version tracking capabilities. It has the benefit of providing a means of backing up, as-well as maintaining synchronisation of project files across multiple machines. It also serves to document the progress of a project through commit messages.

Coding Conventions

All presented code follows strict conventions to ensure readability. Following is a list of conventions used.

- **Names:**
  All variable and function names are given in lower-case with multiple words separated by an underscore. Function names declared in a module are prefixed with an associated acronym. For instance, functions declared in 'deque.c' start with 'de_'. Type definitions are named in the same fashion as a variable, followed by a trailing '_t', to match the style used by the pthreads library.

[1]This decision is contrary to the choice (i.e. C++) documented in the project proposal.

- **Parentheses:**
  For functions and iteration statements the opening parenthesis is placed on a new line, in any other case it can be placed on the same line. The closing parenthesis is always placed on its own line.

- **Constants and Pre-Processor Directives:**
  Pre-processor directives are declared in the accompanying header file. All names are in upper-case with words separated by an underscore.

- **Commenting**
  Comments are used to describe code sections which are challenging to understand or where complicated constructs are used. Where code is simple enough to be self documenting comments are used sparingly. In general the block style is used (i.e. /* ... */).

Modular Design

The implementation takes the form of a modular design. The module which has the functionality to compute the Mandelbrot set has an interface to which a separate 'scheduling' module can be attached. The Makefile holds a rule for each scheduling module and builds a binary for each. This approach provides several benefits.

- It ensures all scheduling modules use exactly the same scheme to compute the Mandelbrot set, making them more comparable.
- It allows for a common user-interface for each executable, independent of the scheduler back-end.
- It allows for alteration to the mandelbrot module to be made with ease.

3.2 An Algorithm to Compute the Mandelbrot Set

This section describes the Mandelbrot module, which provides an interface for run-time scheduler implementations to compute a raster-plane of the Mandelbrot set line by line. This interface is provided primarily by the compute_line function. The module requires that any attached scheduler module implements the functions ws_initialise_threads and ws_start_threads. The former should be used for any set-up (e.g initialising global variables), and the latter is used to start processing the raster plane.

3.2.1 The Mandelbrot Module

The key functions used in the mandelbrot module are described as follows:

- **compute_line:**
  Iterates a single line of the x axis on the raster plane, specified by a parameter y, and assigns the result to the corresponding members of the plane matrix. It also accepts a 't_id' (thread id) parameter, which is used for visualising regions of work completed per thread. This is used as a point of interface for the attached scheduler module, which is responsible for iteration of the y axis. The implication of this design means that the minimum granularity of a work item is one 'line'.

  The function converts the co-ordinates x and y into a corresponding complex number c using the converte_x_coord and convert_y_coord functions respectively. Should the value of c lie outside the radius of two from the origin, the point is assigned the constant PPM_BLACK. Its thread id is assigned WORKER_COUNT to indicate to indicate that the graphical output should ignore any thread information for this pixel. Otherwise the pixel is assigned the value returned from the is_member function and the thread id is assigned the value of the t_id parameter.

- **is_member:**
  Returns a value between 0 and the constant MAX_ITERATIONS. It accepts a complex number c which corresponds to one point on the complex plane and one pixel of the raster plane. This function determines (approximately) whether point c tends towards infinity.

  The variable z is initially assigned the value of c. The function iterates until either the MAX_ITERATION count is reached or $\sqrt{z_r^2 + z_i^2} > 2$, which is simplified to $z_r * z_r + z_i * z_i > 4$ to avoid the 'sqrt' function for efficiency reasons. The former condition indicates that c is a member of the set and a constant to indicate this is returned. The latter indicates that c is not a member of the set and the number of iterations is returned, which produces a gradient effect on the output image. At the end of each iteration the next value of z is obtained by calling the julia_func function, passing the current value of z and the value of c.

- **julia_func:**
  Computes the function shown in equation 2.1. It accepts two complex numbers; z and c.

  The returned complex number is calculated by the following two expressions; for the imaginary part $res_i = 2 * z_r * z_i + c_i$ and for the real part $res_r = (z_r * z_r) - (z_i * z_i) + c_r$.

- **convert_y_coord** and **convert_x_coord:**
  Collectively converts a coordinate of the raster plane into its corresponding point on the complex plane. The convert_y_coord returns the imaginary part and the convert_y_coord returns the real part. Both return a double type. The two are separated to avoid unnecessary conversion of the y coordinate in the compute_line function, as this only need occur once per 'line'.

The algorithm is based on the level set method [12, p. 188]. It is optimised slightly by checking that the point is not outside the radius of two before calling the is_member function. This is because no member of the set lies outside this radius thus saving us iterating for such a point unnecessarily.

Each pixel of the raster plane maps to a point on the complex plane. Figure 3.2 demonstrates this.

Several parameters are used by the algorithm to construct the raster plane.

- **HEIGHT** and **WIDTH:**
  Constants which define the dimensions of the raster plane i.e the maximum values of y and x respectively.

- **MAX_ITERATIONS:**
  A constant iteration limit which is used to determine if a point lies within the set. This is used by the is_member function.

- **The c_max** and **c_min:**
  Complex numbers which define the limits of the complex plane of which the raster plane samples.

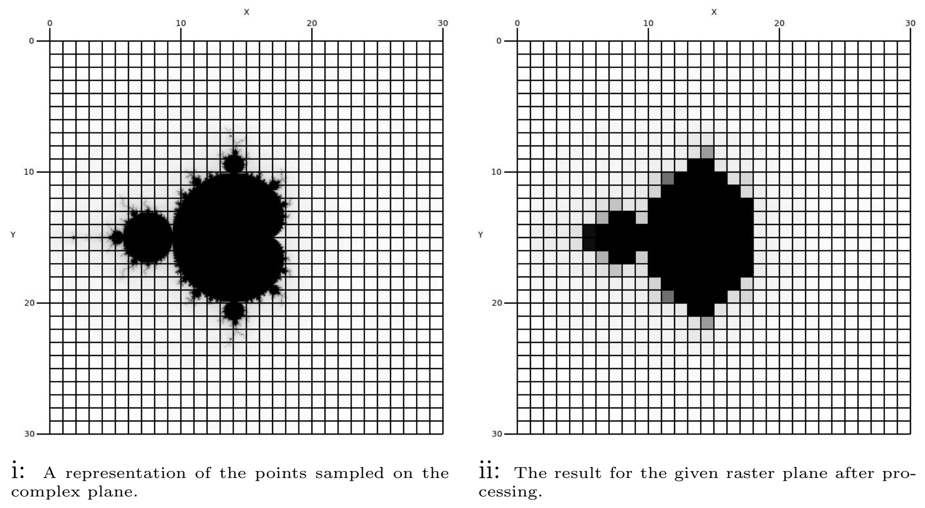


Figure 3.2: A visual representation of an example raster plane where HEIGHT and WIDTH equal thirty and MAX_ITERATIONS equals seventy. Figure 3.2ii shows how the raster plane produces an approximation of the complex plane.

- **The c_factor:**
  A complex number which determines the difference between a sampled point of the complex plane, and an adjacent sample. It is calculated using the dimensions of the raster plane, and the minimum and maximum values of c.

$$cfactor_r = cmax_r - cmin_r/rasterwidth - 1$$

$$cfactor_i = cmax_i - cmin_i/rasterheight - 1$$

3.3 A Randomised Work-Stealing Algorithm

This section describes the implementation of the Randomised Work-Stealing algorithm based on the scheme described in section 2.3.1. This implementation uses four threads, each maintaining its own ready-deque. The implemented ready-deque is based on the scheme presented in [15].

3.3.1 The Deque Implementation

Three key operations are made on a ready-deque. These are de_pop_bottom, de_push_bottom, and de_steal.

- **de_pop_bottom:**
  Accepts a Deque pointer and returns a Line from the front of the deque, or a Line signalling an empty Deque.

  The bottom counter of the Deque passed to the function is decremented regardless of the outcome. If the deque is empty a Line with a status of LINE_EMPTY is returned, expecting the client code to handle this appropriately. If the Deque has more than one item, it may be shrunk and the Line indexed using the bottom counter is returned. If the Deque only holds one item and the top_mutex is locked, then a simultaneous steal operation has claimed the last item. In this case a Line with a status of LINE_EMPTY is returned. Otherwise the remaining bottom item is returned.

- **de_steal:**
  Accepts a Deque pointer and attempts to return a Line from the back of the deque. Otherwise a Line with status LINE_EMPTY or LINE_ABORT is returned.

  If the Deque has only one item remaining a Line with the status LINE_EMPTY is returned. This is tested before the top_mutex is locked in order to avoid unnecessary blocking of the pop_bottom operation. In any other case an attempt to lock the top_mutex is made. If this fails then a simultaneous pop_bottom operation has locked the deque item, and a Line with the status LINE_ABORT is returned. Otherwise the item at the top of the Deque is returned and the top counter is incremented.

- **de_push_bottom:**
  Accepts a Deque pointer and a Line to push onto the bottom of the queue.

  It attempts to re-size the array (if it needs to) and increments the bottom counter whilst placing the Line passed to it on the bottom of the deque.

Both the de_pop_bottom and de_steal operations, in some situations, need to ensure that the thread has exclusive access of the top index of the deque. This is achieved using the pthread_mutex_trylock function, which accepts a mutex and returns *0* should lock be successful. Otherwise, for instance when the mutex is locked by another thread, an error value is returned. This allows the thread to test the state of the mutex, and continue execution without blocking (waiting for the mutex to become un-locked).

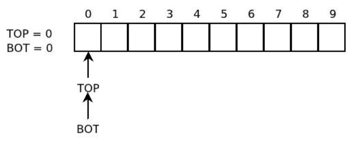
```
if( pthread_mutex_trylock(&d->top_mutex) == 0)
{
    l = d->queue[d->top % d->mem_size];
    d->top++;
    pthread_mutex_unlock (&d->top_mutex);
}
else{
    l = abort_steal;
}
```

Listing 3.1: This excerpt taken from the code for the de_steal operation shows how the pthread_mutex_trylock funciton is used to avoid blocking when the mutex is already locked. If the function returns 0 the mutex is available otherwise the else clause is taken.

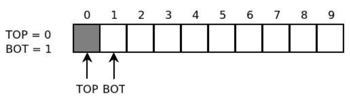
This allows the thread to handle such a situation accordingly; in the case of the steal operation an abort signal, and in the case of the pop operation an empty signal. An empty signal is produced here because the only situation where a de_steal operation will block a de_pop_bottom operation is when there is only one item remaining in the deque, which has already been claimed by the steal operation. The benefit of using this approach, rather than a pthread_mutex_lock based method, is that **dead-lock** is avoided.

The Circular Array

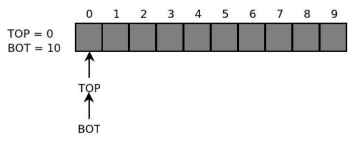
The deque makes use of a circular array which automatically grows and shrinks. This approach is memory efficient and intuitive. It also allows for the top counter to remain unchanged unless a steal operation occurs (in most cases), reducing the frequency of locks occurring. Figure 3.3 illustrates some of its mechanics. For instance figure 3.3v shows that the array is indeed circular, in that it "wraps around" when the final element is reached and there are elements un-used at the start.



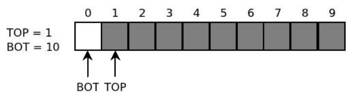
i: The initial state of the circular array.



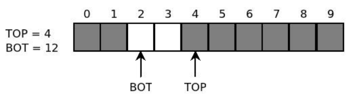
ii: The state after one de_push_bottom operation.



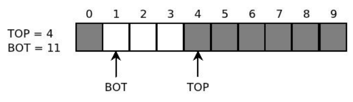
iii: After a further nine de_push_bottom operations.



iv: After one de_steal operation.



v: After three more de_steal operations followed by two de_push_bottom operations.



vi: After one de_pop_bottom operation.

Figure 3.3: A demonstration of the a sequence of operations applied to the circular array data-structure. The 'TOP' field refers to the index of the top element of the deque and the 'BOT' field refers to the index of the bottom element plus one. White squares indicate an un-used element, shaded squares indicate elements which are in use.

The design relies on bottom and top counters (used to index the array), the memory-size (the number of memory slots allocated to the deque), and the size (the number of elements which are currently used). The size is computed by subtracting the top counter from the bottom counter. This is used to detect an empty array, and detect when the array needs to be re-sized. When the array is accessed, the correct element is indexed by computing $realindex = index \bmod memorysize$. For example, let the top counter equal fifteen and the memory-size equal ten, the *realindex* of the top element is five. This allows for the top and bottom counters to exceed the memory size and still point to the correct index, without allowing access to memory outside the bounds of the array. It also maintains that the size can be computed correctly. When the array becomes empty the counters are re-set to zero.

Should the size exceed the memory-size, the array is automatically re-sized by double its current allocation. This is achieved by allocating a new array using malloc, copying the contents of the old array then freeing it, then assigning a pointer to the new array to the deque. Similarly, the array is shrunk by half when its size recedes to half of the memory-size using the same approach. The consequence of this design is that the top_mutex must be locked throughout any re-size operation, blocking any de_steal attempts until the mutex is un-locked. The benefits of this approach are that the deque is more scalable, and only memory that is likely to be used is allocated. Rather than some amount that is defined at compile-time.

3.3.2 The Work-Stealing Mechanism

The design described here utilises the deque implementation to produce an effective work-stealing scheme. The algorithm is so called 'randomised' because of the means to which a victim is selected.

- **ws_worker_thread:**
  The function which is passed to pthread_create. This acts as the point of entry for each thread. It accepts a pointer to the deque which is associated with the given thread. This is passed in the form of a void pointer which has to be cast due to the interface provided by the pthreads library.

  This function makes use of a do-while loop which breaks when there is no work

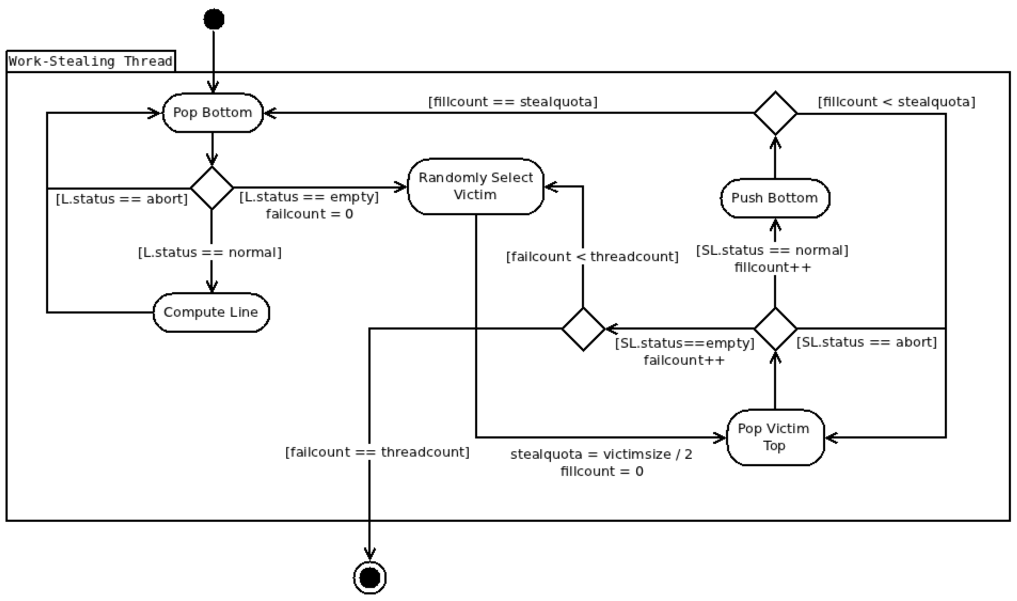


Figure 3.4: An activity diagram to describe the behaviour of a work-stealing thread which utilises the double ended queue outlined in section 3.3.1. L is the line popped from the threads deque. SL (stolen line) is the line popped from the victim's deque. The *victimsize* is the is the amount of work currently in the victim's deque. The *stealquota* is the amount of work items to be stolen. The $fillcount$ is the number of items that have been stolen so far. The $failcount$ is the frequency of failed steal attempts.

remaining in any worker-threads ready-deque. Each iteration of the loop sets the thread to work by calling the ws_compute_deque function, which returns when no work remains in the ready-deque. Then, the thread becomes a thief and must attempt to acquire work from other threads by calling ws_become_thief. If more work is acquired the thread becomes a worker and calls ws_compute_deque again. If no work is found the conditions of the loop are broken and the thread exits.

- **ws_compute_deque:**
  Accepts a Deque pointer, i.e. to the Deque associated with the same thread. This function calls de_pop_bottom until it detects that the ready-deque is empty, at which point it returns the total number of work items completed for this call.

- **ws_become_thief:**
  This operation randomly selects a thread to victimise. If it detects that there is no work available to steal it returns 0, otherwise it returns 1. It accepts a Deque pointer to the Deque associated with the same thread.

  When a victim is selected the ws_victimise function is called.

  In order to determine that no work is available, a set to mark each thread which is unsuccessfully victimised (referred to as the exclude set), as-well as a counter is maintained. The exclude set is initialised by adding the current thread. It is passed to ws_random_deque which uses the standard c rand function to generate a number between zero and the number of threads, used to access an array of deques indexed by thread id. The randomly selected deque is returned.

  The function expects the exclude set to have at-least one element which is not a member.

- **ws_victimise:**
  Accepts deque pointers for the thief and victim threads. This function is responsible for the attempted migration of half the victims workload, to the deque of the thief thread. It returns zero for an unsuccessful victimisation and one for success.

  The initial line is acquired by calling de_steal on the victim. If the line has the status LINE_EMPTY zero is returned. Otherwise the victim has work to be stolen, so its work-count is evaluated and halved to determine how many work items the thief can attempt to steal. The initial line is pushed onto the thief's deque and the next line is stolen, providing the fill_count (the counter which tracks how much work has been stolen) has not exceeded the allotted amount of work to be stolen. Otherwise one is returned. The fill_count is then incremented. The line is checked for the LINE_ABORT status. If this is true de_steal is called again. Otherwise this entire process is repeated until the LINE_EMPTY status is received, or the allotted amount of work is stolen.

Problematic Characteristics

This scheme allows for multiple threads to perform work-stealing operations simultaneously. Due to the random nature of victim selection, some problematic situations may arise at run-time. These situations are likely to be detrimental to the overall balance of work load. They are described as follows:

- **Double Victimisation:**
  Where two threads concurrently victimise the same thread. In this situation more than half of a victims deque is migrated. It can result, under certain circumstances, in almost all of the work in a victims deque being re-distributed in quick succession.

  This is problematic because it can force unnecessary migration of work, which subsequently results in more frequent steal operations.

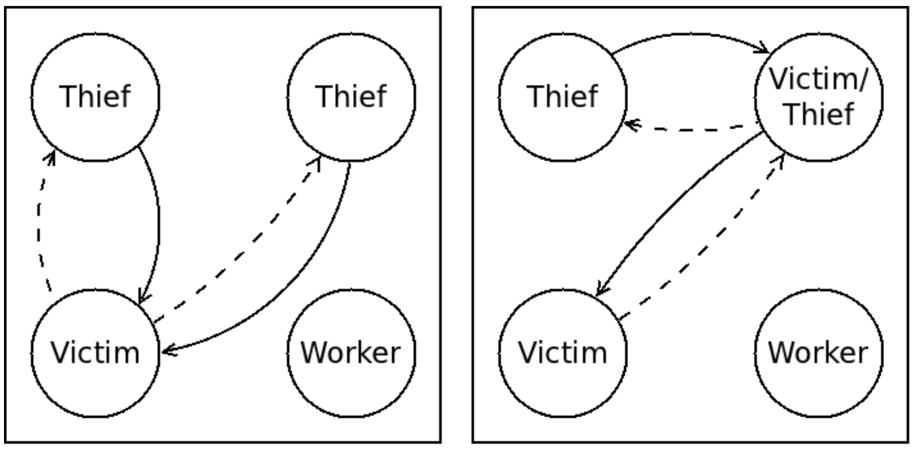
  Figure 3.5i illustrates this phenomenon.

- **Victimisation of a Thief:**
  When a thief is selecting a thread to victimise it does not discriminate between working threads and thief threads. Thus a 'sub-thief' can steal work from a thread which is already stealing work from a victim.

  In this situation work is indirectly moved from a victim to the sub-theif, via another threads deque. The problem is the size of a thief thread's deque, does not equal the total number of stolen items until all items are migrated. As a result the sub-thief effectively steals less than half of the thief's eventual workload, as it evaluates its size before all items are transferred. This could result in an unfair re-distribution of work, and ultimately an unbalanced work load.

  The diagram in figure 3.5ii represents such a situation.

Both problems could potentially be mitigated by making work-stealing operations atomic using mutex locks. In practice this involves implementing a mechanism where a thread can only take the state of either worker, thief, or victim at any given time, where thieves may only victimise workers. This would come at the cost of blocking execution of a thread in certain cases, and would add to the complexity of the algorithm.



i: Double Victimisation ii: Victimisation of a Thief

Figure 3.5: Visual representations of problematic situations. Dashed arrows show the flow of work items.

3.4 Additional Schemes

This section describes three additional modules which make use of different alternative approaches to that described in section 3.3. They interface with the Mandelbrot module in the same fashion as described in section 3.2

These schemes primarily serve as a benchmark for the randomized work stealing algorithm, but also serve to verify the Mandelbrot module.

A Sequential Algorithm

This is the simplest algorithm presented and utilizes no concurrency. It iterates each line (i.e. y index) of the raster plane in sequence, calling compute_line for each value of y. This module effectively implements a variation of the level set algorithm described in listing 2.1 by adding the outer for loop of the compute_mandelbrot function.

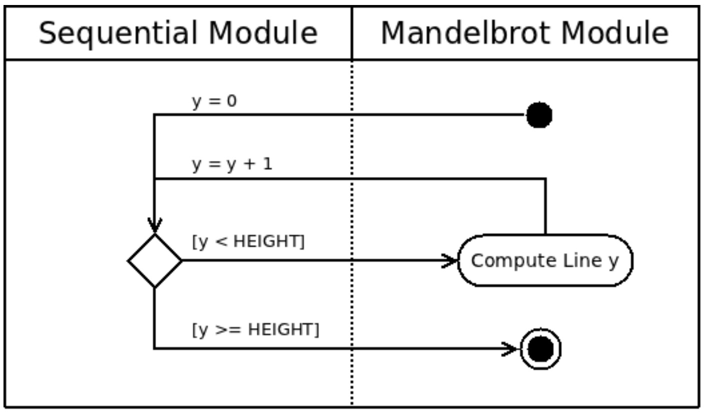


Figure 3.6: An activity diagram which illustrates the simplicity of the sequential scheme.

A Naïve Parallel Algorithm

This scheme is a simple parallel algorithm. It is so called naïve because of its comparable unbalanced work distribution to the a run-time work balanced algorithm. It is an example of a typical scheme used for composing a parallel program, based on a sequential implementation.

The raster plane is divided into equal regions, which each thread computes independently. The regions are comprised of a set of contiguous y lines of the size $height/threadcount$. Each region is assigned as follows where $ystart$ is the first and $yend - 1$ is the last member of the region:

$$regionsize = height/threadcount$$
$$ystart = threadid * regionsize$$
$$yend = (threadid + 1) * regionsize$$

For the region with the thread id equal to $threadcount - 1$ (i.e. the thread which computes the lower region) $yend = height$. This is to ensure that all lines are assigned when $height$ is not integer divisible by $threadcount$. This functionality is omitted from figure 3.7 as not to muddy the simple, demonstrative purpose of the diagram.

Figure 3.7 illustrates how the design of the sequential version, documented in figure 3.6, is composed onto multiple threads to attain a parallel design. This is done by introducing a fork point, which starts the threads, and a join point which returns program control to the main execution flow.

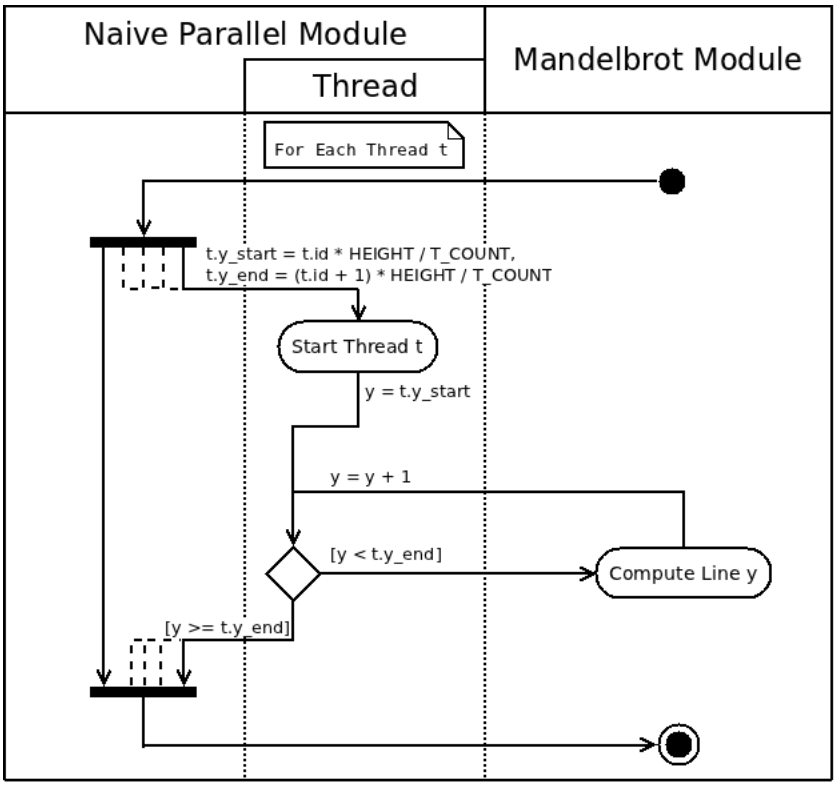


Figure 3.7: An activity diagram which illustrates the use of simple multi-threading employed by this scheme. It is worth noting that the dashed lines near the fork and join bars are added to illustrate that more than one thread executes simultaneously.

A Render-Thread Work-Stealing Algorithm

This scheme implements the approach described in the background research section 2.3.2. One monitor thread and three worker-threads are used.

The monitor thread is delegated the responsibility of controlling steal operations, and synchronising the thief and victim threads in order to complete this operation. Each render thread computes its work-load until it has no more to do. At this point it produces a thief signal which the monitor thread detects. A worker thread can be made a victim while it is working. It is allowed to finish its current line before becoming a victim.

The expected completion time of a render thread is evaluated to determine a victim. This means such a metric needs to be maintained, and regularly updated, in order to make this approach fair. The time taken to complete a work item is multiplied by the amount of work remaining for that thread. This is re-calculated after each line is computed.

Figure 3.8 shows a graphical representation of a typical, happy-day scenario steal operation. It highlights the amount of inter-thread communication which occurs in order to complete such an operation.

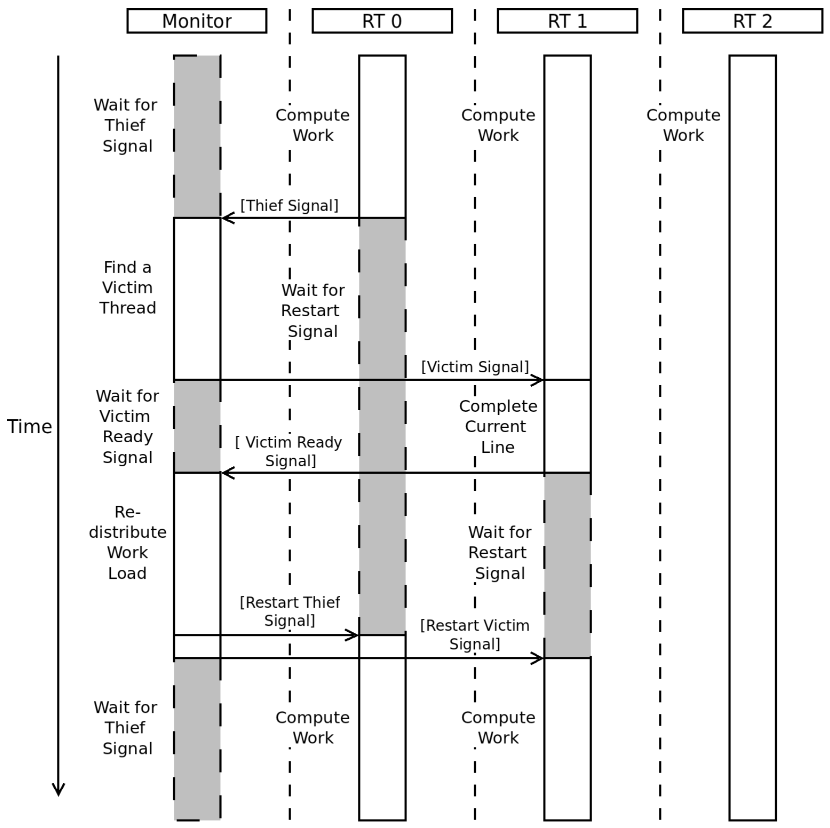


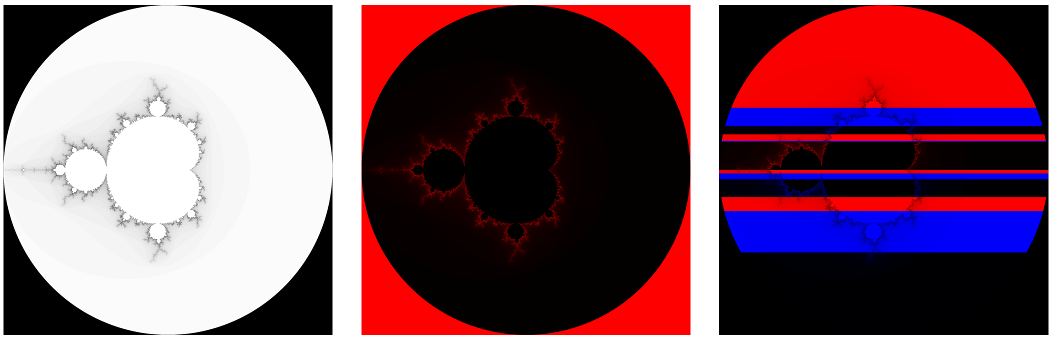
Figure 3.8: A diagram which illustrates a typical steal operation for the Render-Thread Algorithm. It illustrates how steal operations require thread synchronisation. Grey shaded boxes, with dashed borders, represent regions of time in which the corresponding thread must wait for a signal. White boxes indicate that a thread is performing some task. Arrows between threads indicates inter-thread communication.

This approach blocks working threads in order to achieve work-stealing. This comes with the added complication that such a scheme needs to be carefully designed in implementation.

Unfortunately, due to time constraints, there is no fully complete implementation for this scheme in place. It is worth noting that this area alone is a significant body of work, and represents a good portion of the future work this project could lead to.

3.5 Output

The system is capable of outputting PPM image files of the computed raster plane. This can be enabled using command line options described in section 3.7. Output is made optional because the process of producing the file is computationally time consuming, especially for large raster planes. It contributes to the code which cannot be parallelised.



i: greyscale ii: redscale iii: distribution

Figure 3.9: Example images produced using the three implemented output modes. All three are produced using the Randomised Work-Stealing Algorithm detailed in section 3.3.

Output File Format

The specification for the PBM, PGM, and PPM formats is presented here [7].

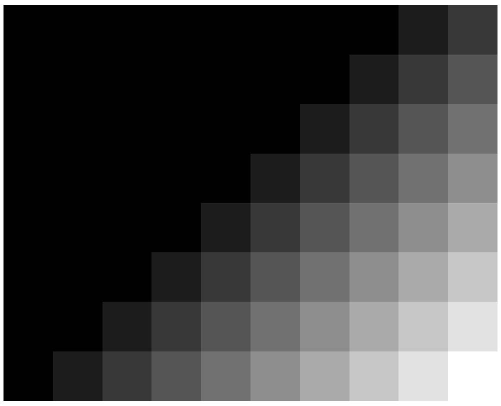
Each file contains data for one image encoded in plain text. This consists of some header information, followed by an entry for each pixel.

The header is comprised of four fields. On line one the 'magic number' determines the colour style used. P1/P4 is used for monochrome (PBM), P2/P5 for grey-scale (PGM), and P3/P6 for RGB colour (PPM). On line two is the width and height as decimal numbers, separated by a space. On line three is the maximum value of a pixel. For the PBM format this line is omitted because it is redundant. Each line is separated with a carriage return.

Pixel data is separated by spaces and arranged in lines ending in a carriage return. Each line has exactly *width* entries and their are exactly *height* lines. For the PPM format each pixel has three entries representing red, green, and blue.

```
 1 P2
 2 10 8
 3 9
 4 0 0 0 0 0 0 0 0 1 2
 5 0 0 0 0 0 0 0 1 2 3
 6 0 0 0 0 0 0 1 2 3 4
 7 0 0 0 0 0 1 2 3 4 5
 8 0 0 0 0 1 2 3 4 5 6
 9 0 0 0 1 2 3 4 5 6 7
10 0 0 1 2 3 4 5 6 7 8
11 0 1 2 3 4 5 6 7 8 9
```

i: Plain text



ii: Image

Figure 3.10: An example of the PGM file format and its corresponding image.

3.6 Run-Time Tracing

A simple tracing mechanism is implemented allowing for more detailed run-time analysis of the program. Any trace related code is not compiled by default, as it has a detrimental effect on performance. To enable this feature an option must be passed to 'make' when the project is built.

Related code is encased in preprocessor directives as follows:

```
/* Tracing for all modes */
#ifdef TRACE
...
trace_event("mode 1: %",123);
#endif

/* Mode specific tracing code */
#if TRACE == 2
...
trace_event("mode 2: %d",456);
#endif
```

The trace_event function prints a time-stamp in microseconds relative to the start of execution, followed by the desired message. A mutex is locked to ensure each output message is completed and written in sequence.

Currently two modes are in place. These can be built by passing the following arguments to 'make':

TRACE=1

A completion message is printed when the scheduling module has completed execution. This serves as a means of measuring the execution time of the program as a timestamp relative to the start in μ seconds.

```
499066: complete
```

TRACE=2

Traces thread activity in scheduler modules. All thread related traces specify a thread id using the T_id field. The following example shows a trace of the naïve parallel algorithm.

```
242: T_id 1 started
...
754652: T_id 1 finished computing 1250 lines
...
754804: complete
```

Tracing the Randomised Work-Stealing Algorithm

The trace format for the Randomised Work Stealing algorithm requires some explanation. It is detailed as follow:

```
1 1316: T_id 0 started
2 ...
3 396334: T_id 0 steal vi: 3, ws: 10
4 ...
5 1948938: T_id 3 ret-work fc: 1
6 ...
7 1960169: T_id 3 finished li: 3237, sc: 4, vc: 4
```

Listing 3.2: Examples of the Four Traced Events for the Randomised Work-Stealing Scheme

- **started:**
  The thread specified by the T_id field has started. Line one of listing 3.2 shows an example.
- **steal:**
  The thread specified by the T_id field has successfully performed a steal operation. The vi (victim) field indicates the thread id of the victim and ws (work stolen) field shows the quantity of work items migrated. Line three of listing 3.2 shows an example.
- **ret-work:**
  The thread specified by the T_id field has finished stealing work and is returning to compute its stolen lines. The fc (fail count) field shows how many times the thread victimised another unsuccessfully. Line five of listing 3.2 shows an example.
- **finished:**
  The thread specified by the T_id field has detected no available work and has fineshed. The li (lines) fields shows the quantity of work items computed, the sc (steal count) field shows how many successful steal operations the thread carried out, and the vc (victimisation count) field indicates how many times the thread was victimised. Line seven of listing 3.2 shows an example.

Some examples of full run-time traces are available in the appendices section B.

3.7 User Interface

A basic command line interface is implemented which offers options to control the output of the program. When the program is given no arguments it computes the mandelbrot set with no output. The available options are described in the following paragraphs.

--outmode=<mode>

Determines the appearance of the output PPM file. The mechanism for producind output is described in section 3.5 and examples of each mode are shown in figure 3.9. There are three possible modes implemented:

- **greyscale** Gradient of black to white.
- **redscale** Gradient of black to red.
- **distribution** Each thread is represented by a different gradient.

This option must be used in order for output to occur.

--outfile=<file>

This option is used to specify the name of the output file. It must be used in conjunction with the 'outmode' option. If this option is omitted the output file takes the default name 'out.ppm'.

--help

Displays a usage message which describes the options detailed above.

Chapter 4

Discussion and Evaluation

4.1 Analysis of the Implemented Algorithms

This section discusses some observations made based on results gathered over a series of program runs. It also serves to asses the validity of the implemented schemes, ensuring that the code does what is expected.

The Test Setup

Here is a brief description of the environment used when gathering the results to support this section.

Test System

- CPU: Quad Core Intel Core I5 750 - 2.67GHz.
- Memory: 8GB DDR-3
- Operating System: Linux Mint 14 - Kernel 3.5.0-17 generic.
- C-Compiler: GCC Version 4.7.2

4.1.1 Performance Analysis

The following observations relate to performance of the implemented schemes. The collected performance data, for both figure 4.1 and 4.2, are calculated using an average time of five runs of the labelled version of the program per plot. These are presented in relation to the sequential version of the program.

The following equation is used to calculate the performance increases shown:

$$RelativePerformanceIncrease_a = \frac{ExecutionTime_a}{ExecutionTime_{sequntial}}$$

- An average speed-up of 3.49 times is achieved for the randomised work-stealing algorithm. The naïve Parallel Algorithm achieves an average speed-up of 2.13 times.

  This observation suggests that by simply adding work-stealing techniques to a parallel scheme which employs no load balancing can increase its performance by an average of 1.66 times.

  Although both schemes make use of four threads, a speed up of four times is unlikely to be possible. This is due to factors such as thread scheduling overheads, portions of code which cannot be parallelised, and imperfect load-balancing. This is associated with Amdahl's law, which states that such a speed-up is limited by the time taken for the sequential portion of the code to execute [28].

- According to figure 4.2, the performance of the Randomised work-stealing algorithm is unaffected by the maximum iterations parameter. The naïve Parallel Algorithm exhibits a performance decrease as the value is increased.

  As the maximum iterations is increased the work required to compute a point in, or close to, the mandelbrot set is amplified. As a result the initially distributed work-load becomes further unbalanced.

  These results suggest that the Randomised Work-Stealing algorithm exhibits good scalability because of its load-balancing capabilities when considering the maximum iterations. This point is also true for the plane size as shown in figure 4.1.

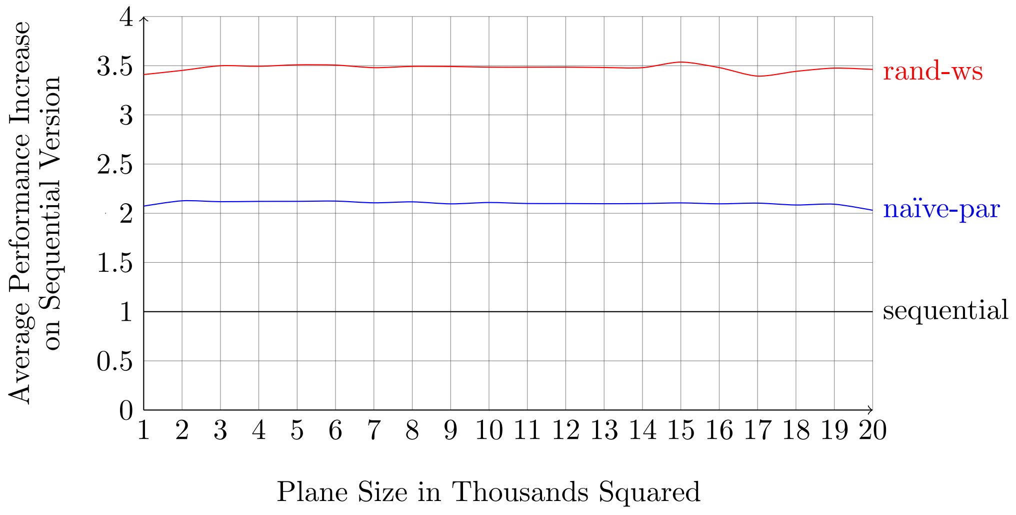


Figure 4.1: Average performance increase, relative to the sequential version of the program, against resolution of the raster-plane. The maximum iteration count is seventy for each sample.

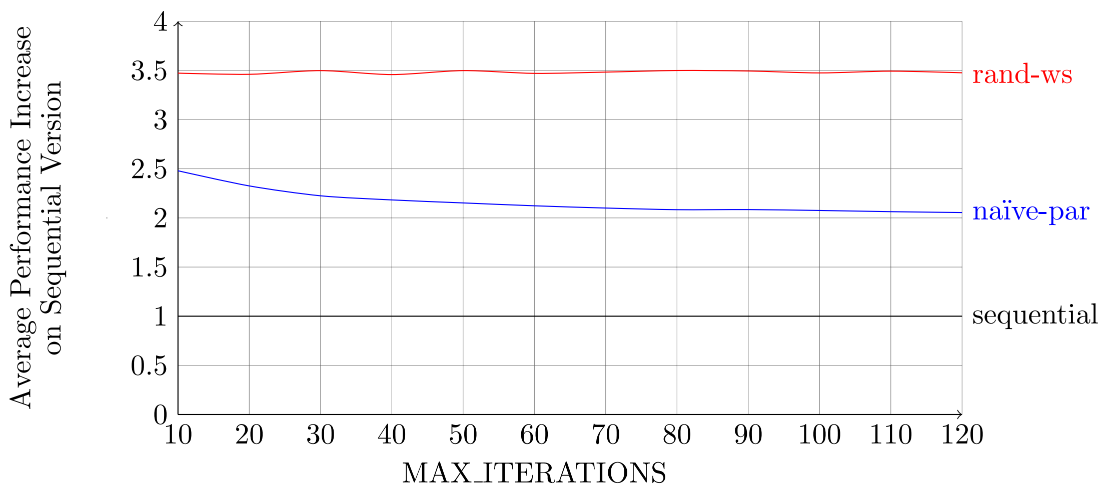


Figure 4.2: Average performance increase, relative to the sequential version of the program, against maximum number of iterations taken to determine the result of a single point. The raster-plane is ten-thousand squared in size for each sample.

4.1.2 Run-Time Analysis of the Randomised Work-Stealing Algorithm

The scatter plot shown in figure 4.3 and the bar charts shown in figure 4.4 bring to light some interesting properties of the algorithm. They also highlight some potential points of improvement.

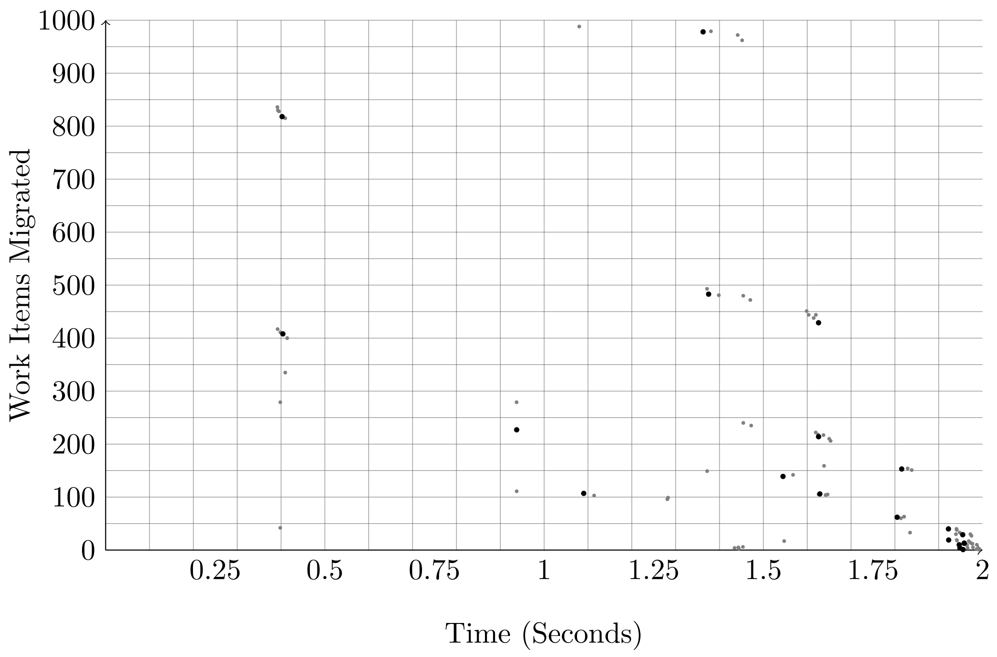


Figure 4.3: A scatter plot where each point is a succesful steal operation. Each point is a mapping of time of occurance, and frequency of work items migrated in the operation. Points marked in black are collected from a single run of the program. Gray points are extracted from an additional four runs. All data is collected using trace mode two documented in section 3.6, where the raster plane is ten-thousand squared in dimension and the maximum iterations is set to seventy.

The following observations are based on the results presented in figure 4.3:

- As less work items are available, steal-operations become more frequent, in which fewer work items are migrated. This trend continues until no work items are available and completion is detected, as is suggested by the data.

  The result is that a high frequency of steal operations, where very small amounts of work are migrated, occur as processing nears termination. The benefit of migrating such small work-loads could be seen to outweigh the cost of allowing the thread to continue computing the given region.

  This could be alleviated by implementing a work migration threshold, which limits a steal operation to a minimum number of steal-able work. Should the projected work-load to be migrated fall under this threshold, the steal attempt would fail.

- A situation where multiple threads perform a steal operation in close time proximity arises often. Five clear examples of this are exposed at approximately 0.4,1.4,1.6,1.8, and 1.9 seconds. This outweighs the three instances in which a single steal operation occurs within 0.1 of a second.

  The following trace excerpt corresponds to the data shown in the plots marked in black. It details the three points plotted at approximately 1.6 seconds.

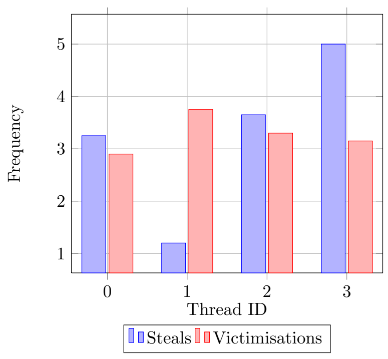
```
...
1625982: T_id 0 steal vi: 1, ws: 429
1626009: T_id 3 steal vi: 1, ws: 214
1628855: T_id 2 steal vi: 3, ws: 106
...
```

  This example highlights some possible instances of the problematic situations detailed in section 3.3.

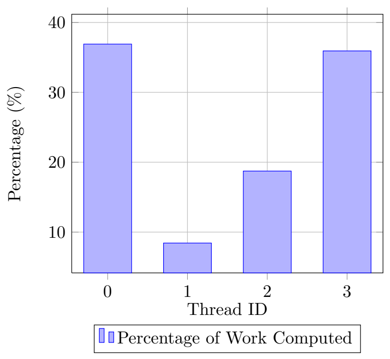
  A double victimisation is present where both thread zero and three steal from thread one. Notably thread three migrates roughly half the work items thread zero does. This suggests that thread one's work load has been reduced by nearly three quarters in quick succession.

  In addition, closely after, thread two victimises thread three. This suggests that victimisation of a thread has occurred, and again the transferred work-load is approximately half that stolen by the victim. It implies that work has indirectly been moved from thread zero, to thread two, via thread three.

  This behaviour is a side effect of using a randomised approach to victimisation. Other approaches, for example evaluating work throughput metrics, could alleviate these problems.



i: Steal count and victimisation count per thread.



ii: Percentage of total work-load computed per thread.

Figure 4.4: Sub-figure 4.4i shows a bar chart to display the average frequency of steal operations and the average frequency of victimisations, per thread id, over twenty test runs. Sub-figure 4.4ii shows a bar chart to show the average percentage of the raster-plane computed by each thread over the same twenty test runs.

The following observations are based on the bar charts presented in figure 4.4:

- Each thread is victimised on average 3.275 times over twenty program runs. The lowest average victimisation count is 2.9 (for thread zero) and the highest is 3.75 (for thead one). These results, which are visualised in figure 4.4i, suggest that randomly selecting a victim thread to steal from is a reasonably fair mechanism.

  It is worth noting that this is dependant on the qualities of the random number generator function. This implementation uses the ‘rand’ function provided the ‘stdlib’ library with a static seed. This is done to produce more repeatable behaviour. A more effective approach may be to produce a random function in which the seed is based on a dynamic system property, for instance the clock time.

- The threads which are initially assigned the upper and lowermost quarters of the raster-plane compute, on average, roughly three times more work items than the inner threads. This demonstrates that the algorithm implicitly balances workload at run-time, and is shown in figure 4.4ii.

  The behaviour exhibited here is expected, as the majority of the computational load lies in the central regions. Put more concisely; when a point is deemed a member of the set, exactly the maximum number of iterations will have occurred, thus involving more work. In general, the closer a point lies to the set, the more costly it becomes to process.

Thread ID	**Completion Time** (μsecs)
0	410922
1	3001814
2	2956266
3	432267

Table 4.1: The time taken from thread start to finish, for each thread in the naïve parallel algorithm to compute its assigned region. The data presented here is sourced from the trace presented in appendices figure B.3.

As the set has symmetry on the horizontal axis it is expected that each half should represent an equal proportion of the work-load. This can be illustrated by analysing a trace of the naïve parallel algorithm. Table 4.1 shows that the central regions take longer to compute when no run-time load balancing is employed.

Split into four initial, equally sized regions the randomised work-stealing algorithm should ideally redistribute work to alleviate the exhibited behaviour of the naïve-parallel algorithm. For the outermost regions (i.e. threads zero and three) this is true, as can be seen in figure 4.4ii, where these ultimately compute approximately the same proportion of the plane. However, a discrepancy arises for the innermost threads (one and two) where thread one, on average, computes much fewer lines than its counterpart two.

This could be attributed to a number of aspects of the design. For instance, the phenomena of double victimisation and victimisation of a thief could be responsible for the unfairly balanced work load. Further investigation is required to determine a more evident cause.

4.2 Reflection

In general, the author feels that this project is a successful one. It has been an enlightening and thoroughly enjoyable experience to investigate the subject area. Here is an evaluation of each objective defined in the introduction section, as well as some aspects of the project which apply in general.

Core Objectives

Background Research

Background research is presented in chapter 2. To start; a broad range of areas related to scheduling parallel algorithms is explored and explained. The work-stealing approach is introduced here, however the discussion is further developed in section 2.3. A concise explanation of the Mandelbrot set, and its surrounding themes, is offered between these. Finally, an overview of considered tools is added to show the extensive range of relevant development resources available.

The studied areas provide the reader with a sufficient overview of the themes explored in the subsequent chapters. To further bolster the readers understanding, a glossary of terms is presented in appendix A. This gives a brief definition for some of the major concepts discussed, which go slightly beyond the scope of this body of work.

A Sequential Mandelbrot Set Algorithm

Although the sequential algorithm described in section 3.4 appears trivial, it relies almost entirely on the mandelbrot module described in section 3.2. This objective encompasses the implementation of this module, as this separation is derived from a sound design decision to encapsulate in order to reduce code complexity. It also improves code re-usability, of which is exploited extensively elsewhere throughout the project.

This body of work also demonstrates good application of the C programming language.

This objective is met with a high degree of quality, and forms a well-grounded foundation for the following objectives to build on.

A Naïve Parallel Mandelbrot Set Algorithm

A basic understanding of composing parallel programs is effectively demonstrated by completion of this objective. The described implementation in section 3.4 shows effective use of the pthreads library and its functionality.

A Random Work-Stealing Mandelbrot Set Algorithm

This objective forms a significant proportion of the work carried out during this project. It is the central focus of the project and exhibits more advanced programming and design techniques. This is described in great depth in section 3.3.

The implemented scheme demonstrates a strong grasp of concepts discovered through research of the subject area (see section 2.3), and utilises these in a proficient and competent fashion. The design is both elegant and simple, as-well as effective in practice. Some advanced techniques demonstrated here include; production of a non-blocking double ended queue, more advanced use of pthreads mutex locks, and design of a sophisticated work-load balancing algorithm.

Constructive criticism of the scheme is also offered, giving commentary on the implications of certain properties and possible remedies for problematic behaviour.

Overall, this portion of the project represents the majority of the technical and theoretical learning achievements made.

Analysis of the Implemented Algorithms

Technical analysis of the end product is offered in section 4.1. Observations made, based on data collected from traced run-time activity, are offered.

These provide some insight into the performance of the end product, as well as some explanation for undesirable behaviour.

This assesses how effective the randomised work-stealing algorithm is and identifies room for improvement. It also serves as validation to show that the product does what is intended, as well as to verify that it functions correctly.

Advanced Objectives

Graphical Output

The program produces output images in various styles. These are detailed in section 3.5. This proves that the product produces a correct approximation of the mandelbrot set (seeing is believing) and, in the case of distribution mode, gives evidence of the work-stealing algorithm's effectiveness.

The image format used is text based, thus inefficient and slow to write. The solution used to achieve this objective is basic, but serves the purpose.

A Render Thread Work-Stealing Mandelbrot Set Algorithm

This objective is incomplete as of the compilation of this report. Areas explored in an attempt to complete this additional objective include a design for the scheme, and a working but unstable implementation.

Work-Stealing Trace System

This objective is completed and provides an effective means for gathering and analysing run-time characteristics of an implemented scheme.

Although the objective is met a number of improvements could be made to allow tracing of a more detailed set of information.

Further Points of Reflection

Project Management

The major objectives for this project where completed comfortably in the time-scale provided. These achievements closely correspond with the planned time allocation described in the Gantt chart, which accompanies the detailed project proposal.

A major factor which streamlined the process is the decision to use a Git repository. It provided a vital means of tracking progress through regular commit messages, and the ability to easily revert back to older revisions. All the work related to this project can be found here: *https://github.com/mhawes/wstealmandel*. The available punch-card graph gives an idea of how frequently used the repository is, as well as an insight into the authors sleeping pattern over the past twelve weeks.

Professional Presentation of Material

All material delivered in a highly professional manner.

In particular, the decision to produce this entire report using LaTeX has been pivotal to the clarity and good structure of this report. It has also provided a useful new skill set, which will be taken forward to future projects.

4.3 Further Work

Although the core project objectives have been completed, as well as the majority of the additional objectives, there are a number of interesting areas in which this work could be extended.

Some of these suggestions could form whole BSc Computer Science projects in their own right. The author hopes they may at least invoke some inspiration for those planning to embark on such a task.

Alternative Work-Stealing Schemes and Policies

To further investigate the work-stealing approach un-explored configurations could be implemented. Areas which could be examined include:

- **Different Victimisation Policies:** Techniques for selecting a victim, for example choosing the thread which has the lowest work-throughput, could prove to more-fairly balance work-load. This would also serve as a point of comparison for the randomised approach.
- **Threshold Stealing:** By enforcing a minimum limit for the amount of work a thief can steal, a number of unnecessary steal operations could be avoided.
- **Alternative Deque Implementations:** The Deques implementation could be re-implemented using a different data structure design. For instance instead of a circular array, a linked list could be put in place.
- **Different Multi-Thread Designs:** Other approaches to parallel algorithm design could applied to produce other, perhaps more intuitive, designs. A fully working implementation of the Render-Thread scheme would be a good starting point for this line of enquiry.

Computing Julia Sets

The existing code in the Mandelbrot module could be fairly easily adapted to support raster-plane generation for Julia sets. The algorithm could accept a parameter to specify the value of z for the function described in equation 2.1. This would expose a huge set of new test cases for the randomised work-stealing algorithm implementation.

Investigate Specific Multi-Processor Architectures

Parallel computing platforms such as cluster computing, cloud computing, cellular architectures, amongst many others geared toward high performance computing, would be a relevant area for further study.

By focusing on a particular class of architecture the algorithm could be optimised, taking into consideration such properties as cache and memory layout, locality of related work-items, and processor locality.

An implementation which utilizes a cluster architecture, using message passing protocols, is of particular interest to the author.

Development of A Parallel Programming Library

The work-stealing techniques explored in this project could be applied to a vast number of programs. Producing a library which provides abstract programming directives, such as parallel data-structure iterators, or support for fully strict multi-threaded computations, could prove a useful general programming tool.

Expansion of the Capabilities of the Tracing Mechanism

A more concise tracing system could be implemented. The current mechanism provides only very limited run-time information of the implemented algorithms. Features to calculate statistics such as work item throughput per thread, detection of detrimental situations, and running tallies of thread status are desirable. These could provide vital debugging tools and could help to identify detrimental properties of the algorithm.

More Efficient Image Generation

Currently the output image takes a noticeably long time to write. This is due to the nature of the file format used.

The program could be adapted to use a dynamic graphics library, such as OpenGL, to display the image as it is generated. In adition, this could be used to illustrate the work-stealing scheme in action. As each thread completes a work item its corresponding line would appear on the screen in its assigned colour gradient. This would show how work items are migrated throughout runtime and serve to demonstrate the workings of the algorithm.

4.4 Conclusion

This study highlights a simple, yet effective, work-stealing scheme for computing a problem which statically exhibits unbalanced work-load when parallelised. It is applied to a parallel program which computes an approximation of the Mandelbrot set. This is compared and contrasted with both a sequential, and naïve parallel approach to the same problem. These deliverables are supported and guided by an extensive review of relevant literature.

The investigation finds that by applying the Randomised Work-Stealing Algorithm, a significant speed-up is made over a similar design which neglects run-time load-balancing techniques. A further benefit is found in that the algorithm scales well with an increase in raster-place size and maximum iterations. Randomisation is also shown to be a fair method for victimisation. The analysis serves to identify some problematic situations in the form of double victimisation and victimisation of a thief, which could have an impact on the fairness of work-load redistribution.

All core objectives and all bar one advanced objectives are complete. The work done towards producing a Render-Thread Work-Stealing Algorithm, although incomplete in the time-scale, still offers some insight into an alternative approach to Work-Stealing.

The project as a whole is reflected upon in great depth and a number of avenues for future work are laid out. Over-all the author considers the project a great success in the respect that the proposed work is complete, and a great deal of valuable knowledge has been acquired. The pursuit of this projects completion has allowed the author to hone his skills in many useful programming tools and techniques. The learning outcomes have not only been technical, good project and time management skills have also been gained.

The main and most valuable contribution is the implementation and design of the Randomised Work-Stealing algorithm. This demonstrates the technical ability of the author, as well as a strong understanding of programming and computer science theory.

Chapter 5

Resources

References

[1] About javaplot. ONLINE. http://gnujavaplot.sourceforge.net/JavaPlot/About.html accessed: 10th Febuary 2013.

[2] Fraqtive - mandelbrot family fractal generator. ONLINE. http://fraqtive.mimec.org/ accessed: 5th Febuary 2013.

[3] The free opengl utility toolkit. ONLINE. http://freeglut.sourceforge.net/ accessed: 4th March 2013.

[4] Glut - the opengl utility toolkit. ONLINE. http://www.opengl.org/resources/libraries/glut/ accessed: 4th March 2013.

[5] Gnuplot homepage. ONLINE. http://www.gnuplot.info/ accessed: 10th Febuary 2013.

[6] Gnuplot-iostream interface - c++. ONLINE. http://www.stahlke.org/dan/gnuplot-iostream/ accessed: 10th Febuary 2013.

[7] Ppm format specification. ONLINE. http://netpbm.sourceforge.net/doc/ppm.html accessed: 10th Febuary 2013.

[8] S-net - home. ONLINE. http://snet-home.org/ accessed: 17th April 2013.

[9] Single assignment c – high productivity meets high-performance. ONLINE. http://www.sac-home.org/ accessed: 17th April 2013.

[10] Nimar S Arora, Robert D Blumofe, and Greg C Plaxton. Thread scheduling for multiprogrammed multiprocessors, 1998.

[11] Blaise Barney. Posix threads programming. ONLINE, 2013. https://computing.llnl.gov/tutorials/pthreads/ accessed: 10th Febuary 2013.

[12] Micheal F Barnsley, Robert L Devany, Benoit B Mandelbrot, Heinz-Otto Peitgen, and Dietmar Saupe Richard F Voss. *The Science of Fractal Images*. Springer-Verlag, 1988.

[13] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[14] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors, 1998.

[15] David Chase and Yossi Lev. Dynamic circular work-stealing deque, 2005.

[16] Peter Coad. *Java modeling in color with UML : enterprise components and process*. Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[17] Nathan Cohen. Fractals new era in military antenna design, 2005.

[18] Nicolas Devillard. Gnuplot interfaces in ansi c. ONLINE. http://ndevilla.free.fr/gnuplot/ accessed: 10th Febuary 2013.

[19] David Feldman. *Chaos and Fractals: An Elementary Introduction.* OUP Oxford, 2012.

[20] Vladimir Janjic. *Load Balancing of Irregular Parallel Applications on Heterogeneous Computing Environments.* PhD thesis, University of St. Andrews, 2012.

[21] Bob Kuhn, Paul Petersen, and Eamonn OToole. Openmp versus threading in c/c++, 2000.

[22] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications, 2012.

[23] Benoit B Mandelbrot. How long is the coast of britain?, 1967.

[24] Benoit B Mandelbrot. *Fractals: Form, Chance and Dimension.* W.H.Freeman and Co Ltd, 1977.

[25] Benoit B Mandelbrot. *The Fractal Geometry of Nature.* W.H.Freeman and Co Ltd, 1983.

[26] Jason McGuiness. Atomic code-generation techniques for micro-threaded risc architectures. Master's thesis, University of Hertfordshire, 2006.

[27] Steve R. Palmer and Mac Felsing. *A Practical Guide to Feature-Driven Development.* Pearson Education, 1st edition, 2001.

[28] David A. Patterson and John L. Hennessy. *Computer Organization and Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[29] Michael M Resch, Alexander Schulz, Matthias S Muller, and Wolfgang E Nagel. *Tools for High Performance Computing 2009.* Springer-Verlag, 2010.

[30] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Factory: An object-oriented parallel programming substrate for deep multiprocessors. *Proceedings of the 2005 International Conference on High Performance Computing and Communications*, pages 233–242, 2005.

[31] Waide B Tristram. Investigating tools and techniques for improving software performance on multiprocessor computer systems. Master's thesis, Rhodes University, 2011.

Chapter 6

Appendices

Appendix A

Glossary of Terms

Glossary

barrier A thread synchronisation technique in which a specified number of threads must reach an explicit point in execution before any may continue. 11

circular array An array in which the element indexed directly after the last element is effectively the first element, giving the effect of the data-structure "wrapping around". 9

complex-numer A number which is expressed in two parts; real and imaginary $(a+b_i)$. 5

condition-variable A thread synchronisation technique which allows conditional lock of a resource dependant on the state of some value. 11

critical-orbit The orbit of point 0 for a set of complex numbers. 7

dead-lock A situation in which two or more threads are waiting for each other to give up exclusive access to a resource required to continue execution. 18

exact self-similarity A property of a shape which is comprised of an exact copy of part of itself. 6

fractal A set which has a fractional dimension. 6

load-balancing A method in which work-load is distributed across multiple resources in an attempt to optimise utilisation of such resources. 5

locality The amount to which a value, or set of related values, is utilised by a resource. 5

mathematical-fractal Fractals which could exhibit the property of infinite scale invariance or quasi-self similarity. 6

monitor-thread A thread which performs administrative tasks such as maintaining worker-threads and controlling synchronisation. 9

multi-threaded algorithm A program which utilises more than one thread to achieve its desired result. 5

mutex In the context of pthreads, a mutex is a mechanism for acquiring exclusive lock on a resource. Short for mutual exclusion. 11

non-blocking A property of a multi-threaded algorithm in which shared resources are not locked, allowing threads to operate without being stopped. 9

quasi self-similarity A property of a shape which is comprised of an approximate copy of part of itself. 6

ready-deque A queue data-structure which allows access to both the top and bottom elements. 9

real-world-fractal Fractals which do not exhibit the property of scale invariance. 6

scale-invariance A property of a shape which is identical at magnifications of a common factor. 6

scheduling The process of determining the order in which parts of a program access system resources. 5

self-similarity A property of a shape which is comprised of part of itself. 6

steal-operation The process of re-distributing work-load from a victim to a thief. 9

thief A thread which has the work-load of some other thread re-assigned to its own. 8

thread A strand of execution which operates independently from the main flow of control in a process. 5

victim A thread which has a portion of its work-load re-assigned to some other thread. 8

work-sharing A processor which creates new work attempts to migrate it to another underutilised processor at creation time. 5

work-stealing A processor which is starved of work attempts to “steal” work from other processors. 5

worker-thread A thread which, in general, performs tasks that work towards achieving the purpose of the program. 9

Appendix B

Trace Output

The trace examples here are produced using the following constants in mandelbrot.h:

- HEIGHT = 10000
- WIDTH = 10000
- MAX_ITERATIONS = 70

```
2997274: complete
```

Listing B.1: An example trace produced by the 'semandelbrot' binary. Trace mode is one.

Other traces for mode one are omitted because they all take the same form.

```
4: Started
6311056: Finished
6311137: complete
```

Listing B.2: An example trace produced by the 'semandelbrot' binary. Trace mode is two.

```
264: T_id 2 started
270: T_id 1 started
292: T_id 0 started
315: T_id 3 started
411214: T_id 0 finished computing 2500 lines
432582: T_id 3 finished computing 2500 lines
2956530: T_id 2 finished computing 2500 lines
3002084: T_id 1 finished computing 2500 lines
3002311: complete
```

Listing B.3: An example trace produced by the 'nsmandelbrot' binary. Trace mode is two.

```
1289: T_id 1 started
1289: T_id 0 started
1318: T_id 2 started
1418: T_id 3 started
402418: T_id 3 steal vi: 2, ws: 818
402426: T_id 3 ret-work fc: 0
404182: T_id 0 steal vi: 2, ws: 408
404185: T_id 0 ret-work fc: 0
```

```
937474: T_id 2 steal vi: 3, ws: 277
937483: T_id 2 ret-work fc: 0
1090359: T_id 0 steal vi: 2, ws: 107
1090368: T_id 0 ret-work fc: 0
1362642: T_id 2 steal vi: 1, ws: 978
1362648: T_id 2 ret-work fc: 0
1375208: T_id 0 steal vi: 2, ws: 483
1375211: T_id 0 ret-work fc: 0
1544923: T_id 0 steal vi: 2, ws: 139
1544931: T_id 0 ret-work fc: 0
1625982: T_id 0 steal vi: 1, ws: 429
1625989: T_id 0 ret-work fc: 0
1626009: T_id 3 steal vi: 1, ws: 214
1626012: T_id 3 ret-work fc: 0
1628855: T_id 2 steal vi: 3, ws: 106
1628857: T_id 2 ret-work fc: 0
1805224: T_id 2 steal vi: 1, ws: 62
1805232: T_id 2 ret-work fc: 0
1815588: T_id 3 steal vi: 0, ws: 153
1815592: T_id 3 ret-work fc: 0
1922202: T_id 1 steal vi: 3, ws: 40
1922211: T_id 1 ret-work fc: 1
1922743: T_id 2 steal vi: 1, ws: 19
1922746: T_id 2 ret-work fc: 0
1946743: T_id 2 steal vi: 3, ws: 10
1946746: T_id 2 ret-work fc: 0
1948303: T_id 1 steal vi: 3, ws: 4
1948305: T_id 1 ret-work fc: 0
1954789: T_id 1 steal vi: 0, ws: 29
1954791: T_id 1 ret-work fc: 0
1955573: T_id 3 steal vi: 2, ws: 1
1955576: T_id 3 ret-work fc: 0
1958113: T_id 3 steal vi: 0, ws: 13
1958114: T_id 3 ret-work fc: 1
1958254: T_id 2 steal vi: 1, ws: 13
1958255: T_id 2 ret-work fc: 0
1978160: T_id 1 finished li: 855, sc: 3, vc: 6
1978439: T_id 0 finished li: 3872, sc: 5, vc: 3
1978493: T_id 2 finished li: 2010, sc: 7, vc: 6
1978682: T_id 3 finished li: 3262, sc: 5, vc: 5
1978762: complete
```

Listing B.4: An example trace produced by the ‘wsmandelbrot’ binary. Trace mode is two.

Appendix C

Source Code