Pthreads and OpenMP

A performance and productivity study

Henrik Swahn

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

Contact Information:
Author(s):
Name: Henrik Swahn
Email: hesw91@gmail.com/h_swahn@hotmail.com

University Advisor:
Name: Daniel Häggander
Department: Software Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Today most computer have a multicore processor and are depending on parallel execution to be able to keep up with the demanding tasks that exist today, that forces developers to write software that can take advantage of multicore systems. There are multiple programming languages and frameworks that makes it possible to execute the code in parallel on different threads, this study looks at the performance and effort required to work with two of the frameworks that are available to the C programming language, POSIX Threads(Pthreads) and OpenMP. The performance is measured by paralleling three algorithms, Matrix multiplication, Quick Sort and calculation of the Mandelbrot set using both Pthreads and OpenMP, and comparing first against a sequential version and then the parallel version against each other. The effort required to modify the sequential program using OpenMP and Pthreads is measured in number of lines the final source code has. The results shows that OpenMP does perform better than Pthreads in Matrix Multiplication and Mandelbrot set calculation but not on Quick Sort because OpenMP has problem with recursion and Pthreads does not. OpenMP wins the effort required on all the tests but because there is a large performance difference between OpenMP and Pthreads on Quick Sort OpenMP cannot be recommended for paralleling Quick Sort or other recursive programs.

Keywords: *OpenMP, Pthreads, Algorithms, Performance, Productivity, Quick Sort, Matrix Multiplication, Mandelbrot Set*

Table of Contents

1 Introduction 6
2 Background and Related Work 8
2.1 Background 8
2.1.1 C 8
2.1.2 Threads 8
2.1.3 POSIX Threads 8
2.1.4 OpenMP 9
2.1.5 Quick Sort 10
2.1.6 Mandelbrot set 10
2.1.7 Matrix Multiplication 11
2.2 Related Work 11
3 Method 13
3.1 Research 13
3.1.1 Questions 13
3.2.2 Design 13
4 Results 15
4.1 Literature Review 15
4.1.1 Literature question answer 16
4.2 Empirical Study Findings 17
4.2.1 OpenMP 17
4.2.2 Pthreads 20
4.2.3 Pthreads vs OpenMP 23
4.2.4 Effort 27
4.2.5 Findings 29
5 Analysis 30
Summary 31
6 Conclusion and Future Work 32
6.1 Conclusion 32
6.2 Future Work 33
References 33
Appendix A. Code 35
Matrix Multiplication 35
Sequential 35
Pthreads 36
OpenMP 37

Quick sort 39
- Sequential 39
- Pthreads 40
- OpenMP 42

Mandelbrot set 43
- Sequential 43
- Pthreads 44
- OpenMP 45

1 Introduction

Ever since the 1970's when the microprocessor that we know today was developed and launched increase in it's performance been achieved by increasing the clock speed. All the way until early 2000 the increase in clock speed was statically, every two years the number of transistors that would fit on a chip would double. This is more commonly known as Moore's Law[1].

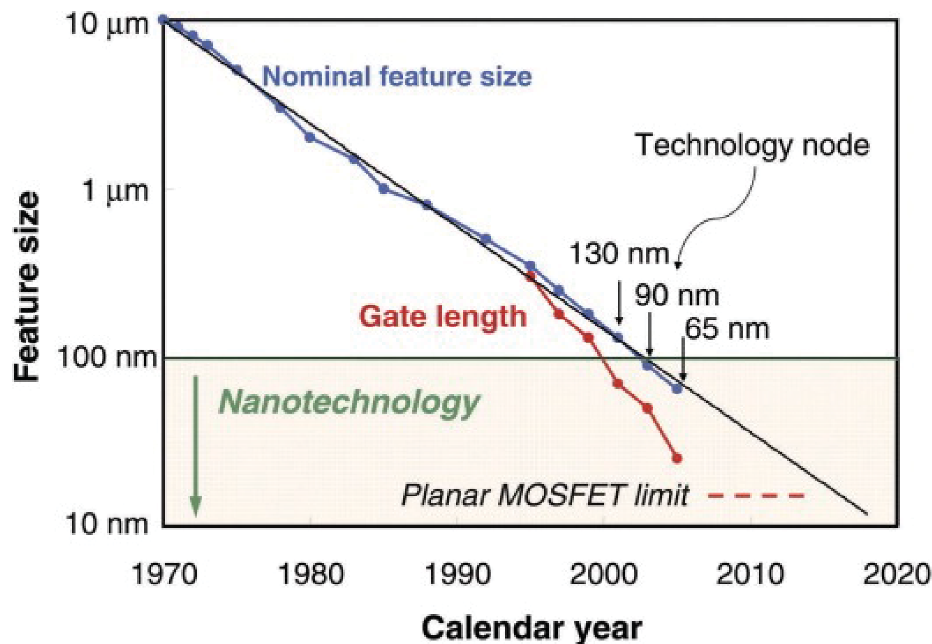


Figure 1 - Transistor size decrease

When increase no longer could be achieved by just increasing the number of transistor on a chip and the clock speed due to high power consumption and heat other means to increase performance had to be developed. One technique that is used to counter the problem is multi-core processors[2].

Today it's very common with a multi-core processors in most computers, laptops, work stations, servers, phones and tablets. Adding just the cores does not means high performance, in some cases it can actually mean worse performance because the individual cores may not have as high clock speed as a single core processor[3]. It's is absolutely vital that the software running on the hardware utilise the cores that the processor offer in order to get enhanced performance.

Developing software that can take advantage of multiple cores can be done in many different programming languages. If the languages does not have a standard library that supports threads there is often a third party library that can be added to allow usage of threads. For all experiments in this paper C was used as main language, reasoning for using C was that C has been and is a very popular programming language and is used in many different areas of development[4], GCC compiler on Linux also has support for the two libraries that was tested. The threading techniques that was investigated in this papers was POSIX Threads and OpenMP.

Today multithreaded computers are everywhere so it's important that the software utilizes this. Pthreads and OpenMP has been around for some time and are both well tested techniques to parallel computing. But Pthreads tends to be more common than OpenMP, so is there a reason for that?

The main focus for the research was the performance offered and effort required to implement a program with Pthreads and OpenMP. The interesting here is that the two models take a very different approach to allow parallel computing, Pthreads takes a more low level approach to threading in the sense that it requires a more tailored program than OpenMP. OpenMP on the other hand is more high level and does not require a very tailored program, it tends to not modify the source code as much as Pthreads does to parallel a piece of software. Both OpenMP and Pthreads require that an library is linked during the compilation but OpenMP also requires that the compiler has built in support for it. Either way, parallelism is achieved and the important aspect is if there is any performance differences. The tests was strictly scientifically, there will not be test on whole application such as web servers etc. The point was to see how the models differentiate in performance, not test what they can be used to build. Another aspect that was investigated were the amount of effort that was required to be able to achieve increased performance using the models.

The focus of the investigation was to investigate the performance that could be achieved with the two threading models. This information can hopefully help developers or companies when they set out to build a system. When they design their system and realise that they are in need of parallelism they should do some research in how different models perform, if they scale good and if they are easy to work with and maintain. This paper will hopefully help with making the right decision early in the development process where it easy and still relatively cheap to make changes by highlighting some situations where the threading models work well and where they does not.

What will come out of this paper is valid arguments for when Pthreads or OpenMP should be used and when they should not. The empirical study will produce results about how the two models perform in different situations.

The expectations for the amount of effort required to implement the parallel version is that OpenMP will be superior to Pthreads, both in the size of the program(number of lines) and the ease that parallelism can be expressed. The question ”Which one will be worth using for my application?” is harder to answer because now the execution time and the effort must be taken into account. Pthreads is expected to be more efficient than OpenMP with a small amount because Pthreads allows the programmer to implement everything, and if done right it should be more effective than OpenMP that does a lot of thing automatically. So when it only is a very small difference in the performance the effort becomes more important and here OpenMP is expected to excel. Expected outcome is that OpenMP will be recommended because it is an easier way to work with threads than Pthreads without sacrificing much performance.

The value comes from showing that OpenMP will be a valid choice when developers needs to parallel parts of a program, they do not get paid to do the most complex implementation but rather they get paid for result.

In the second chapter some background information that can be good to have about some of the techniques that were used during the tests in this paper will be presented and after that some related work. Third chapter is about the method used and the technology that was used, information about C, Threads, Pthreads and OpenMP, the research questions that will be answered by the literature and empirical study will be presented. The last thing that will be explained in the third chapter is the design for the literature and empirical study. In the fourth chapter the findings from the literature study and empirical study will be presented. The results with the sequential, OpenMP and Pthreads version will be presented, after that a more close look how OpenMP and Pthreads perform against each-other and last in this chapter the implementation effort will be presented. In chapter five an analyse about the test and the findings will be presented. Lastly in chapter six conclusion and related work will be presented.

2 Background and Related Work

2.1 Background

2.1.1 C

C is a general-purpose programming language developed in the 1970's by Dennis M. Ritchie. Ever since it's development C has had a central role in the UNIX environment, it started with becoming the main programming language for UNIX when it was re-written[8]. The language is still used today, just looking at the Linux kernel which has around ~97% of it's code base in C[7].

C has many features that appeals to programmers, some of them are:

- C offers low-level features. It is possible to program Assembly in C and access low level features of the operating system[9].
- C is portable, if not using any OS specific libraries C should work on any computer with no modification required[9].
- It offers different datatypes such as char, int, long, float, double etc.[10].
- C is procedural i.e it has support for functions[9].
- C has support for dynamic memory allocation and de-allocation using pointers.

2.1.2 Threads

A thread is a way to make a program execute two or more things in parallel. A thread consist of it's own program counter, it's own stack and a copy of the registers of the CPU, but it shares other things like the code that is executing, heap and some data structures. The main goal with threads is to enhance the program performance and help to efficiently structure a program to performance multiple task at the same time[11].

An example is a web server.

If a web server is not multithreaded and is listening for connections from clients and a connection is found, while handling the connection the web server will be blocked. This means that if another clients tries to connect to the web server he will either be refused or he has to wait, none of the alternatives is an acceptable solution even if the wait time is small.

Instead if the web server is multithreaded what would happen is, when a request is detected the web server would immediately start up a new thread and hand over the request to the new thread. Now the newly created thread can handle the request and the main thread can let the server listen for new connections, by using multithreading the wait time for clients has been eliminated.

2.1.3 POSIX Threads

POSIX Threads is a common threading model when working in C or C++, it is a standardised model to work with threads created by IEEE. POSIX Threads can be use to parallelise task of a program in order to increase the execution speed. The POSIX standard states that threads must share a number of things, for example the threads must share:

- Process ID
- Parant Process ID
- Open FD's (File Descriptors)

POSIX Threads offers a header file and a library that must be included at compilation time or else the program wont compile because functions and data structures wont be found by the compiler. POSIX Threads offers a lot of functions to create, manipulate, synchronise and kill threads in a

program. Starting point for threads in a Pthreads program is a function called pthreads_create, this function takes a pointer to a function that is the starting point for the new thread. Pthread_exit then is used to kill the thread and in the main thread pthread_join is used to wait for the executing threads[5].

2.1.4 OpenMP

OpenMP is a model for parallel programming in shared-memory systems and was developed in the 1990's by SGI. OpenMP is available for three different programming languages, Fortran, C and C++. It consist of a number of compiler directives that can be used in the source code[6], these directives need functions to run and those functions is specified in a library that must be included at compile time. It works by letting the compiler translate the directives in to functions that is found in the OpenMP library, it does this at compile time. That means that the compiler must have support for OpenMP, if it does not have support for OpenMP it wont know what to do with the directives. In most cases this wont crash the application, that is a sub goal of OpenMP. It should be portable between different systems, even systems that does not support it. Of course these system wont have the increased performance from parallel computing but they should still be able to run the program[12].

To create the threads when an parallel directive is encountered OpenMP uses a model called fork-join described in *Figure 2 - Fork-Join model*. What this means is that when a executing thread encounters a parallel construct it will create a team of threads(fork) and become the master thread of the team. Then the team executes the code assigned to it and before the initial or master thread continue to execute the code after the parallel construct all the threads in the team are terminated(join).

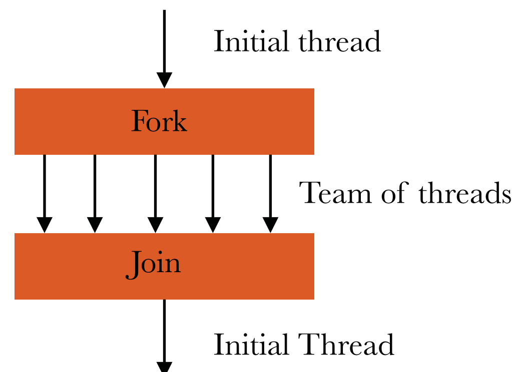


Figure 2 - Fork-Join model

When the team of threads are created tasks that each thread can perform are generated and assigned, the task assigned to a specific thread is always executed by just that thread. So for example if a for loop that will iterate over an array containing 20 elements would be parallelised and the system has a processor with 4 cores, OpenMP will be used to create four threads and execute one thread on each core. The for loop will be wrapped in a parallel directive where the limit of threads is four. Then when the initial thread encounters this construct a fork would occur, four tasks would be created and the goal for each task is to iterate over a subset of the array. This will increase the performance by four times, sometimes more if it benefits from super-linear speedup[23]. Very shortly what super-linear speed up means is that sometimes a program can execute faster than the theoretical speed up i.e. if the system has fours processors and the recorded speed up is greater than four. When the threads are done they terminate and when all of them are terminated the initial thread can continue.

2.1.5 Quick Sort

Quick Sort[18] is sort algorithm that was developed by Tony Hoare in 1959. It works by dividing the original array into sub-arrays. The first sub-array contains all the elements that is less than an chosen pivot element and the second sub-array contains all the elements that is larger than the pivot element. Now both of the arrays can be sorted separately, but before this is done they are partitioned again. To do this a new pivot is chosen in each of the sub arrays and then they are portioned into even smaller sub-arrays. This whole process continues until we have one element sub arrays that does not need to be sorted. In this partitioning process the array is sorted. In it's worst case scenario it has a computational complexity of O(n^2) and an average of O(n log n).

2.1.6 Mandelbrot set

A Mandelbrot[20] set is a special set of fractal values, it marks the set of points in the complex plane[21] such that the corresponding Julia set[22] is connected but not computable. A Mandelbrot set can be visualised by sampling complex numbers and determining if it's results tends towards infinity if particular math operation is iterated on it. A visualisation of the Mandelbrot set is shown in *Figure 3*.

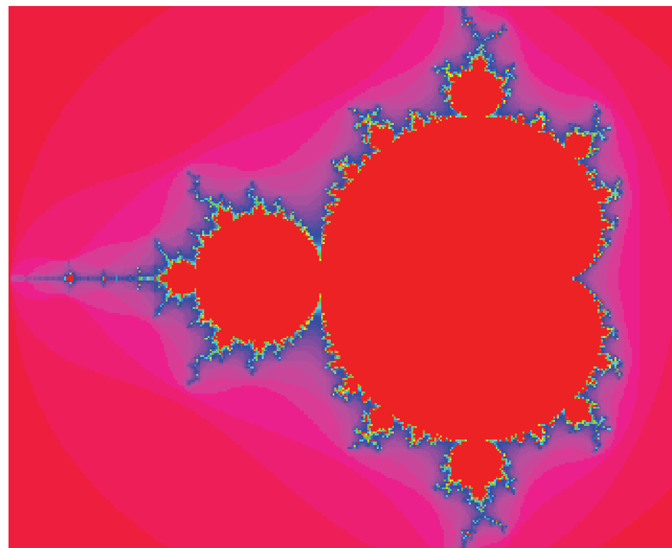


Figure 3 - Visualisation of a Mandelbrot set

2.1.7 Matrix Multiplication

Matrix Multiplication[19] is the process of either multiplying a scalar with a matrix or multiplying two matrices together, the focus will be on the second one. It works by multiplying the rows of matrix A with the column of matrix B. A11 will be multiplied with B11, then A12 with B21, the sum of these multiplications will end up in C11. The next step we multiply A11 with B12, then A12 with B22. The sum of these multiplications will ned up in C12. This will continue for all rows and columns in the matrices. Step by step example of matrix multiplication is shown in *Figure 4*.

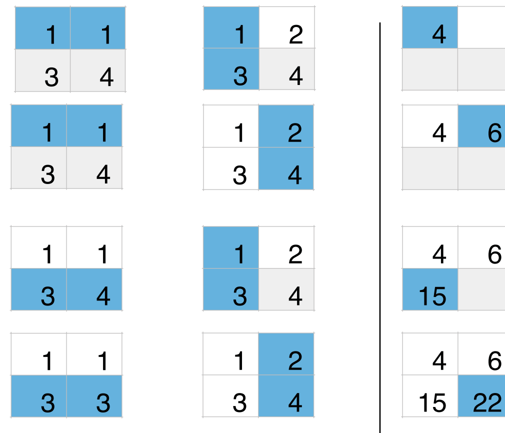


Figure 4 - A visualisation of a Matrix multiplication

2.2 Related Work

OpenMP versus Threading in C/C++[13], Bob Kuhn and Paul Petersen did a paper where they implemented two parallel version of the same program, one with OpenMP and one with Pthreads. Their aim was to see if it was possible to get the same performance from OpenMP compared to a hand threaded library such as Pthreads. They implemented parallelism in a software called Gene-hunter which is a program that analyses DNA from family trees. It's purpose is to find the gene or genes that are likely to cause a decease. The rationale they had to implement parallelism was because the inheritance vectors that needs to be investigated grows exponentially with the family tree size, parallelism would benefit the execution speed a lot. What they found was that they could build the two parallel version of the program and get the same speed in the program. But the interesting finding in their work was that the OpenMP version was to prefer over Pthreads because a number of things. OpenMP was much easier to apply to sequential code, Pthreads need's a much more tailored program towards threading. It was easier to maintain a program using OpenMP and easier to express the parallelism in the program than with Pthreads.

Scientific computations on multi-core systems using different programming frameworks[14], Panagiotis D. Michailidis, Konstantinos G. Margaritis did a paper which goals was to test different threading models in C/C++. Not just the execution speed but also the ease parallelism could be expressed with the different frameworks. Their paper looks at Pthreads, OpenMP, Intel Cilk Plus, Intel TBB, SWARM and FastFlow. Their aim for the paper was strictly scientific, they wanted to look at how the frameworks performs doing calculations such as matrix operations. They concluded a number of different things:

- Some of the models have a single-core overhead compared to a strictly sequential program(Pthreads, Cilk Plus, TBB and Fastflow), but has little impact when the set size grows.
- The execution time reduces significantly as the number of cores increases when the amount of parallel work is high.
- The best performance as a function to the set size and number of cores was performed by SWARM and OpenMP.

- The high level models i.e OpenMP, Cilk Plus, TBB and SWARM support parallel patterns through ready built constructs and only need small amount of manipulation of the source code to create parallelism. Low effort.
- The more low level models i.e Pthreads and FastFlow does not support parallel patterns directly and need to manipulate the source code a lot to create the required parallelism. High effort.

This is close to the focus of my research but the focus will only be on Pthreads and OpenMP libraries.

A User's Experience with Parallel Sorting and OpenMP[15], Michael Süß and Claudia Leopold did a paper where they took a closer look at OpenMP and some problems that can occur using it. It is suppose to be a simple and high level way of interacting with thread creation but it has some drawbacks because of how it works. They looked at recursion and busy waiting because OpenMP has problem with these two areas. Quick sort was the algorithm that was used to test the performance, they chose quick sort because it is a common algorithm that is relative simple and depends on recursion. What they came up with was different way to address or work around the problems. For example removing the recursion from quick sort and make it iterative instead.

Parallelization of General Matrix Multiply Routines Using OpenMP[24] Jonathan L. Bentz and Ricky A.Kendall did a paper where they used OpenMP to test different ways of matrix multiplication. They tested Basic Linear Algebra Subprograms(BLAS) and Double precision general matrix multiply(DGEMM). They used OpenMP utilise data parallelism and spread the work evenly among they available threads.

Parallelizing Parallel Rollout Algorithm for Solving Markov Decision Processes[25], Seon Wook Kim and Hyeong Soo Chang did a paper where they used OpenMP and MPI to introduce parallelism to Parallel rollout algorithm. It is a method of combining multiple heuristic policies avail to a sequential decision maker in the framework of Markov Decision Processes(MDPs). They found that OpenMP was more effective than MPI at paralleling the algorithm because of the way OpenMP handle data synchronisation.

High-Scalability Parallelization of a Molecular Modeling Application: Performance and Productivity Comparison Between OpenMP and MPI Implementations[26],Russell Brown and Ilya Sharapov did a paper about paralleling a molecular modelling application using OpenMP and MPI. They were interested in both performance and productivity that could be achieved with OpenMP and MPI. Their result showed that OpenMP had better performance in all their test cases but did not scale better than MPI in all cases, but the productivity had been best working with OpenMP in all cases because it required much less effort to implement the algorithm than MPI required.

Something that all the research that was examined has together is that they all find that OpenMP is very simple to work with and does not require that much effort. Not much research was found that put Pthreads and OpenMP against each other but the few that was examined [13, 14] all find that OpenMP require much less effort than Pthreads. [13] gets the same execution speed from Pthreads and OpenMP but find that OpenMP is to prefer because it is much easier to work with. [14] finds that OpenMP has very good performance in relation to the set size and number of processors available and that Pthreads need's a ver tailored program to work well.

The other research that compares OpenMP against other threading techniques find that OpenMP is superior to them in effort but not always in speed and how well it scales. [15] took a close look at OpenMP and some problems that it has and finds that it does not works well with recursion at all. This is a problem since recursion can be a very powerful way of solving some problems.

3 Method

3.1 Research

3.1.1 Questions

1. What results has other research found about comparison between OpenMP and Pthreads?
2. Can OpenMP help computation intensive algorithms perform better on multi-core system?
3. Can Pthreads help computation intensive algorithms perform better on multi-core system?
4. How does the efficiency differ between OpenMP and Pthreads for computation intensive algorithms on multi-core systems?
5. How much efforts is required of the developer to implement algorithms in OpenMP and Pthreads?

Question one is basis for the literature study, the other questions is part of the empirical study. To answer some of the questions related work will be examined, especially for the first question because here it's interesting to investigate what results has been found and what conclusions has been made. Best case scenario is if there are related work that touches this exact topic and has measurements with both OpenMP and Pthreads, then their research can be examined and maybe compared to my research to see if the same conclusion as they made can be made from this research. Question two, three, four and five is part of the empirical study and will be answered through experiments and measurements.

3.2.2 Design

3.2.2.1 Literature study

To find the information to answer the questions two different scientific databases was used, Google Scholar, Summon. But when doing the literature study there was a limited selection of work available that touches this subject. Also turned to google directly just to see what was available on the web, it was a bit easier to find some information here.

3.2.2.2 Empirical study

3.2.2.2.1 Experiments

To investigate the performance of the models a series of test will be performed on common algorithms that can be found in different applications. Algorithms to be used:

- Quick Sort
- Mandelbrot set
- Matrix Multiplication

For each of the three algorithms three programs was created, one sequential and two using parallelism. Own version of the algorithms was implemented due to that i wanted to explore how the different frameworks was to work with, just using code from the internet would not gain me any understanding how they work and how they are to work with. The sequential one is used as a point of reference to measure how much the models increase the performance from a sequential execution. Before each experiment is run and timed a set of data was randomised, it's a function that used rand() from C standard library to fill the set with random values, this set was then used in the program. To make sure that the measurements are correct the experiments was run on different occasions if spikes in the measurement was found, this helped to make sure that the measurements was correct. Each of the programs will be developed in C programming language.

Measurements was taken on the sequential version of the programs, then on the POSIX version and OpenMP version. Measurements was taken on different input set's to see how the parallel version scale depending on the input set's size. Then a comparison between the models was made to be able to make a prediction on which model perform best and under which circumstances.

The last thing that was taken into an account is the implementation effort, the effort required to parallel the sequential version. This was measured in number of lines required to parallel the sequential program using Pthreads and OpenMP.

3.2.2.2.2 Measurements
To be able to measure the part of the program that was interesting a small library that has two functions to help with the time measurements was created. There is a function called diff in the library that takes two timespec[17] structs, one that represents the start of the measurement, and one that represent the end of the measurement. The function then calculates the difference between the two structs and returns this value. The measurements was taken right before the function that needs to be parallel and just after it was done. This gave a more accurate time on how the threading library performed instead of measuring the whole program. Each measurement was done ten times to get a good amount of data before calculating the average. This helped to round of any eventual spikes in the execution time.

When taking the measurements a few things had to be considered, the executing process will not always be on the processor because the processor switches between the actives processes(context switching). This means that if the realtime was taken this will include time when the process is waiting and not executing. The other approach is to just count the time when the process is on the processor and exclude the waiting time, the problem with that is that when comparing the sequential version of the program with the multithreaded version the time outputted is misrepresentative of how the execution time is perceived by the user. The result is misrepresentative because the sequential version will only execute on one core while the multithreaded version will execute on all available cores. The time will be counted for all cores and which means that even if the perceived time is lower by multithreaded version the outputted time from the time function will be greater than the sequential. So when comparing the sequential and parallel version the first approach will be used, in the time library from the standard C library there is something called CLOCK_REALTIME[16] that will count the all the time. The other approach will be used when just looking at Pthreads and OpenMP and how they perform against each other, to measure this the standard C library has something called CLOCK_PROCESS_CPUTIME_ID[16].

The test environment was a multicore environment because i wanted to investigate how the frameworks could perform benefiting from true parallelism i.e. execution on different cpu's or cores. This removes some overhead that exist when using threads on a single core environment because the processor has to schedule the processes and switch between them. The test computer has eight cpu's so to get max efficiency from the threading the number of threads was limited to the number of CPU's. This was possible on almost all of the tests, there is a scenario with OpenMP and recursion when it was hard to limit the number of threads.

The effort to implement one of the algorithm in sequential, Pthreads or OpenMP was measured with the number of lines. Lowest number of lines is the best. To count the number of lines Linux wc command is used with the count lines flag. The structure of all the program is the same i.e the same space between each function and lines of code is the same, so even if that command counts empty lines it does not matter because the code structure is the same for all the programs. The number of lines was used as measurement unit because it gave a number on the effort that could be used to measure the different implementations against each other.

For the tests the cache had to be examined multiple times, it interesting to look at the cache because it can give an explanation why performance can be bad sometimes. To look at the cache the program will be run through a program called perf stat[28] with a flag to capture the number of cache misses.

All the experiments was done on a computer that has two Quad Core Intel® Xeon® E5335, 2x4MB Cache, 2.0GHz processors and 6 gb of ram running on 667mhz.

4 Results

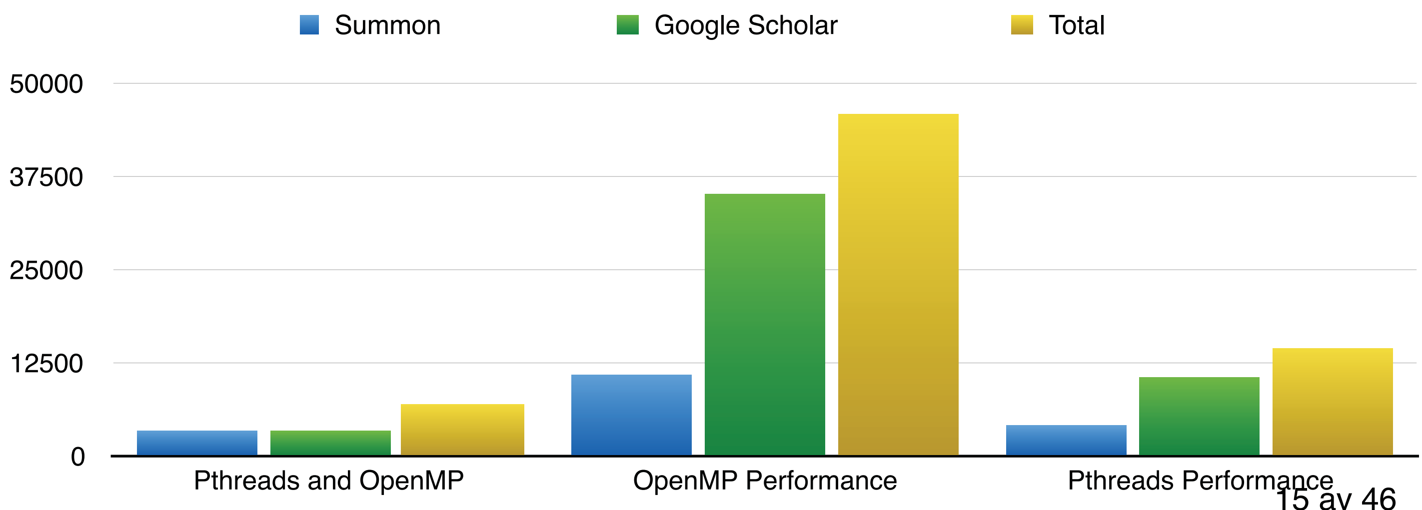
4.1 Literature Review

Searching in the databases for papers that was about Pthreads and OpenMP and performance comparisons there was a total of 6748 hits between the search engines. After reading a total of 50 abstracts and found that only 3 would be useful for my research[13,14,27]. The others that was read were not important for my study because none of them put Pthreads and OpenMP against each other in performance tests.

Searching for just OpenMP there was a lot of hits in the databases, 45409, from all hits 30 abstracts was read to see if what they were doing in the research could be connected to my research, not very much that could be compared to my researched was found. The focus many papers had was to evaluate some algorithm or program using OpenMP, not evaluating how well OpenMP performed. Four papers that was useful used even though they are not comparing OpenMP against Pthreads was found[15,24,25,26].

Searching for Pthreads there was fewer hits than for OpenMP, 14442 results were found between the search engines. Many papers about Pthreads had the same focus as many of the papers for OpenMP, they tested some algorithm using Pthreads but actual focus was not on Pthreads or to test Pthreads against OpenMP. Found 2 papers useful even though they are not comparing Pthreads against OpenMP[27, 28]

Searching in the databases a number of different search phrases was used, they all resolve around Pthreads or OpenMP. The words Pthreads and OpenMP was combined with words like performance, effort, versus, productivity, implementation effort etc. Papers that don't evaluate the technique is excluded from the pool of results that i chose from, with that i means papers that only mentioned that they used one of the framework but don't really talk about the performance or effort in some way.



Results in the databases

4.1.1 Literature question answer

There was not much research on performance comparison, effort comparison or any comparison between OpenMP and Pthreads, only found three papers that brings up this subject and out of those three only two that focus on just OpenMP and Pthreads.

[14] test six threading frameworks and puts them against each other, Pthreads, Fastflow, OpenMP, Intel Cilk Plus, Intel TBB and SWARM. The focus is not only on OpenMP and Pthreads in the research but they did find some interesting things about Pthreads and OpenMP as they did their research. Pthreads has an overhead when executing on single-core environments while OpenMP handles this better. In their tests OpenMP performed better than Pthreads and that OpenMP does require very little programming effort compared to Pthreads which requires advanced programming knowledge.

[13] tested Pthreads and OpenMP against each other by paralleling a program called Genehunter. Their versions of the program achieves equal performance and they also look at the effort required and easy of maintaining the code. They find that OpenMP is the one that is easier to work with and that Pthreads requires a more tailored program.

[27] does two implementations of a program to calculate protein structure prediction using OpenMP and Pthreads, their result is that Pthreads offer best performance which contradicts what [14] concludes. This may be that the program that [27] built did benefit from Pthreads low level api and the ability to keep threads alive and pick them from a thread pool, this eliminates overhead from creating new threads and terminating them. [27] were not able to achieve the same thing in OpenMP, [14] program maybe did not depend on this as much as [27] program which may have resulted in that [14] found that OpenMP was the best.

Findings:

- OpenMP requires much less effort to implement a program compared to Pthreads[13,14]
- There are more alternatives than just Pthread and OpenMP. There are three additional that can be grouped with OpenMP as high level frameworks: Intel Cilk Plus, Intel TBB[14] and SWARM. There is one additional that can be grouped with Pthreads as a low level framework: FastFlow[14].
- Pthreads has a single core overhead that can make a program run slower on some environments[14].
- OpenMP program is easier to maintain and understand[13].
- Not possible to create a thread pool in OpenMP and re-use threads, this can affect performance because there is an overhead in thread creation[27].

4.2 Empirical Study Findings

4.2.1 OpenMP

4.2.1.1 Matrix multiplication

The result that OpenMP achieves when put against a sequential version of matrix multiplication is that it does not outperform it in the beginning. On the small matrices the sequential version is faster than OpenMP, first at matrices with a size of 128 x 128 or bigger OpenMP starts to get ahead. After 128 x 128 matrices the gap just keeps growing for each step in OpenMP's favour. Comparing the execution time after 128 x 128 matrices we can clearly see that there is a performance gain from paralleling the algorithm. OpenMP starts to outperform the sequential version at 128 x 128 matrices and only takes ~71% of the execution time of the sequential version. It continues to outperform the sequential version through all the tests, 256 x 256 matrices took ~33% of the execution time of the sequential version, 512 x 512 took ~15% of the execution time of the sequential version, 1024 x 1024 took ~21% of the execution time of the sequential version, 2048 x 2048 took ~23% of the execution time of the sequential version. Overall there is a good increase from paralleling matrix multiplication with OpenMP especially on larger matrices. Working on small matrices(<128 x 128) it is not worth paralleling the algorithm with OpenMP, it actually makes the program execute slower. *Figure 5* shows the test results:

	Sequential(sec)	OpenMP(sec)
32x32	0,002	0,008
64x64	0,004	0,008
128x128	0,021	0,015
256x256	0,156	0,053
512x512	1,447	0,222
1024x1024	61,530	12,894
2048x2048	553,660	126,442

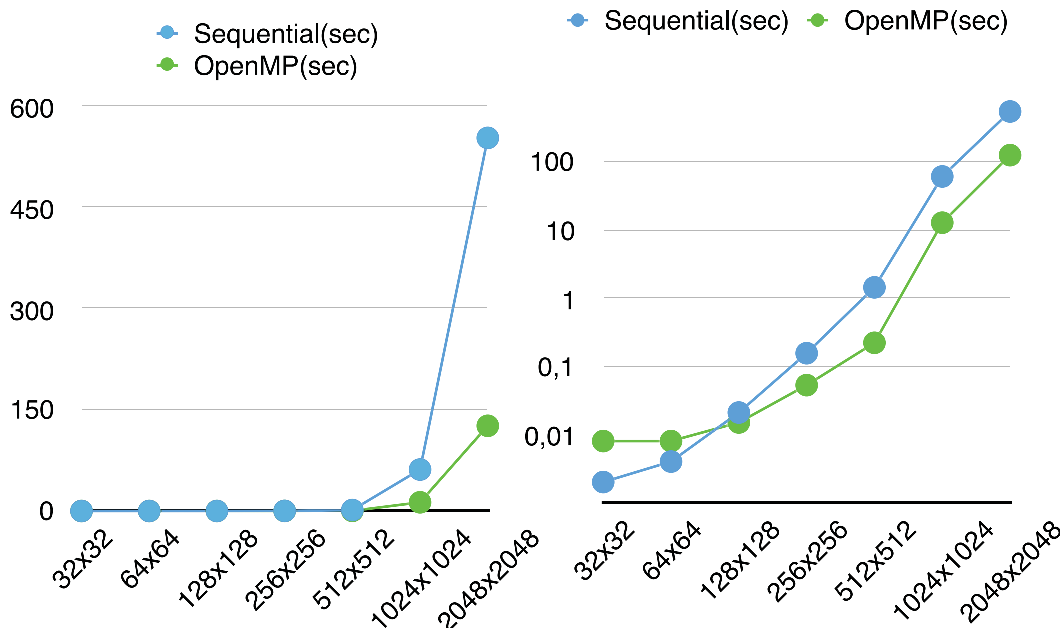


Figure 5 - OpenMP Matrix Multiplication
Left: Linear graph, Right Logarithmic graph

4.2.1.2 Quick Sort

Analysing the result that OpenMP's version of Quick Sort gets against the sequential version of Quick Sort it is obvious that paralleling an algorithm is not always beneficial. From the start OpenMP is outperformed by the sequential version. The reason why it is outperformed throughout the test is because it is hard to express parallelism in recursion with OpenMP, it's one of the drawbacks with OpenMP's high level approach to threading. Because it is hard to express parallelism it was hard to control the number of threads created, it exceeded the number of available cores with a lot. The number of threads that was created when running a test on 10 million elements almost 10 millions threads were created, a lot of the execution time is spent on creating threads. Analysing the final measurement on 100 million element the sequential version takes ~45,5 sec, OpenMP takes ~71,7 sec. OpenMP takes ~57% longer to execute than the sequential version, it is very much slower. It is not worth paralleling Quick Sort even on the smaller sets, a fix could be to not use a recursive Quick Sort but rather make it iterative, this could make it easier to express the parallelism. *Figure 6* shows the test results for OpenMP Quick Sort vs Sequential Quick Sort:

	Sequential(sec)	OpenMP(sec)
10000000	3,9990122873	7,3846038776
20000000	8,3467466488	15,2761299154
30000000	12,7043459658	21,1537746987
40000000	17,266797704	31,7263201437
50000000	21,9652739951	40,5338859016
60000000	26,3655543779	42,1267760366
70000000	31,1980906666	50,4931627025
80000000	35,8601403092	62,8212976466
90000000	40,5653292126	66,0237372302
100000000	45,5124670877	71,7466770492

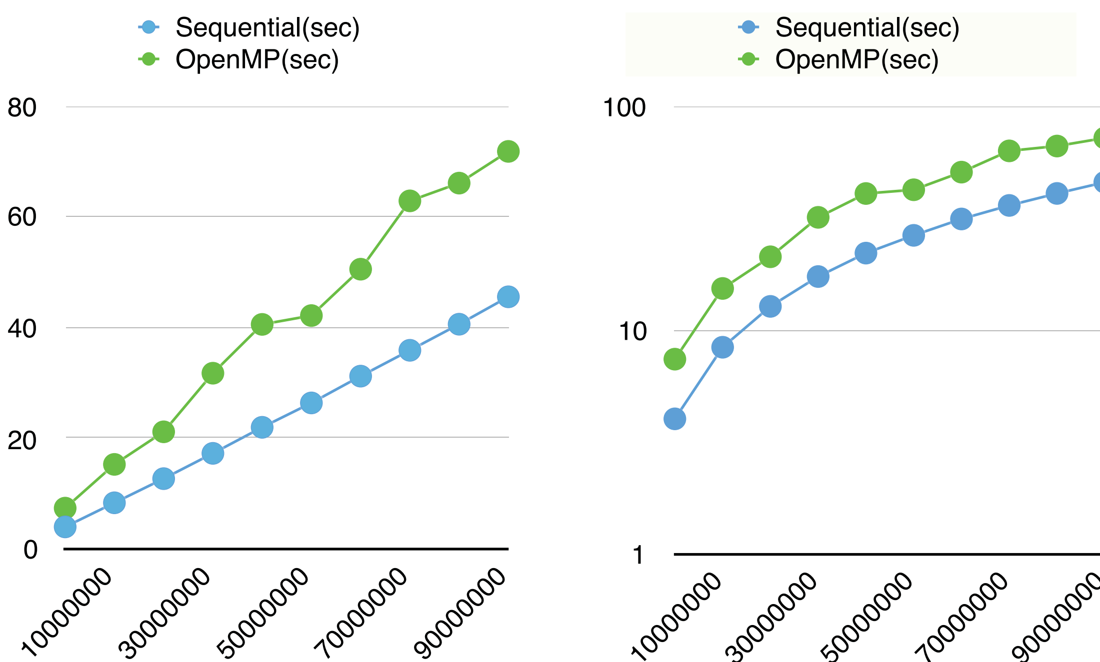


Figure 6- OpenMP Quicksort

Left: Linear graph, Right Logarithmic graph

4.2.1.3 Mandelbrot set

OpenMP performed very good on calculating the Mandelbrot Set, at the start of the test at a resolution of 1000 x 1000 it outperformed the sequential version with over two seconds. The way the calculations are made when calculating the set made it very suitable for paralleling with OpenMP. It is noticeable that OpenMP is designed and optimised for these types of calculations and problems. OpenMP's version of the calculation was ahead of the sequential on all the tests that was run between the them. At 1000 x 1000 OpenMP takes ~34% of the execution time that the sequential version needs, at 1500 x 1500 OpenMP takes ~35% of the execution time that the sequential version needs and finally on 2000 x 2000 it takes around ~34% of the execution time of the sequential. On these test performed OpenMP is around 65-66% faster than the sequential version of the calculation. OpenMP does offer a good increase in performance compared to the sequential version. *Figure 7* shows the test results:

	Sequential(sec)	OpenMP(sec)
1000	4,2569970239	1,4870022712
1100	5,1462506484	1,8097723319
1200	6,1271844849	2,1347673174
1300	7,1905930092	2,5219422388
1400	8,3393718567	2,955996154
1500	9,5733589769	3,3503613675
1600	10,8916757102	3,8258463875
1700	12,2946977356	4,2979192986
1800	13,7826372207	4,8078042232
1900	15,3587830872	5,3596429662
2000	17,183182009	5,9201818945

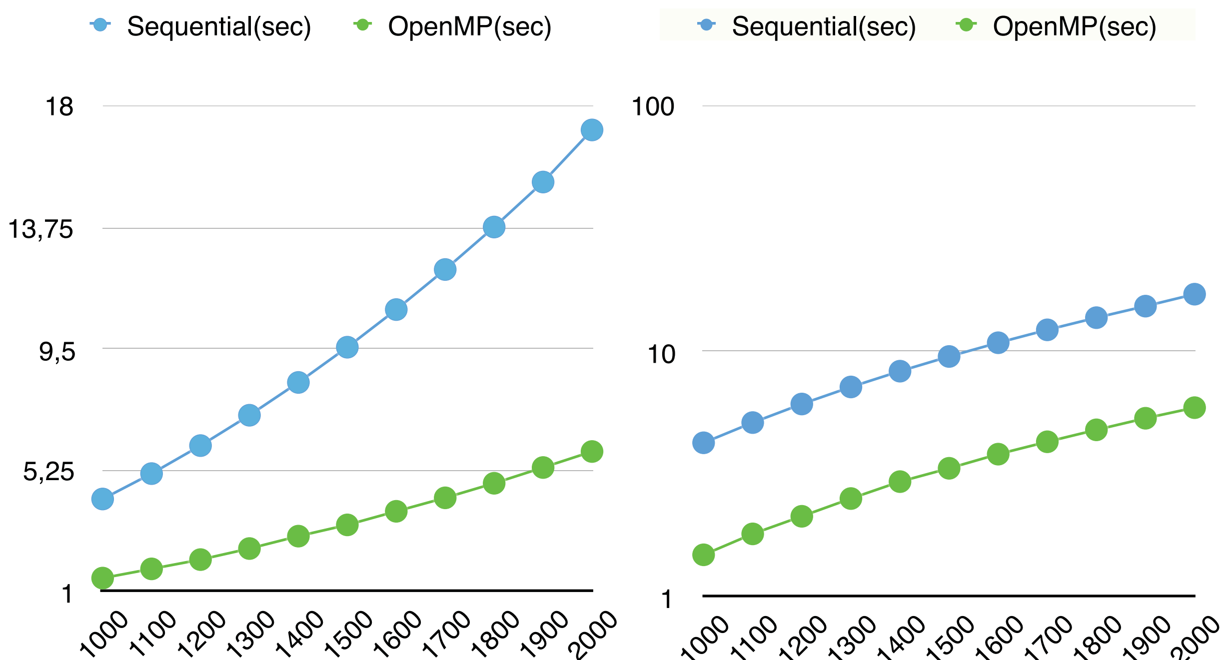


Figure7 - OpenMP Mandelbrot set
Left: Linear graph, Right Logarithmic graph

4.2.2 Pthreads

4.2.2.1 Matrix multiplication

Pthreads does outperform the sequential version on most of the test. On the smallest matrix it is outperformed by the sequential version and on the 64 x 64 matrix it has the same execution time. After that the Pthread version start to perform better than the sequential version. Then the gap keeps increasing through all the tests, and after 128 x 128 matrices a benefit is gained from paralleling the algorithm. It continues to out perform the sequential version through all the tests, 256 x 256 matrices is ~25% of the execution time of the sequential version, 512 x 512 took ~14% of the execution time of the sequential version, 1024 x 1024 took ~21% of the execution time of the sequential version, 2048 x 2048 took ~23% of the execution time of the sequential version. Overall there is a good increase from paralleling matrix multiplication with Pthreads especially on larger matrices. Working on small matrices(<128 x 128) it is not worth paralleling the algorithm with Pthreads, it adds unnecessary complexity to the code. *Figure 8* shows the test results:

	Sequential(sec)	Pthreads(sec)
32x32	0,002	0,003
64x64	0,004	0,004
128x128	0,021	0,010
256x256	0,156	0,040
512x512	1,447	0,209
1024x1024	61,530	12,958
2048x2048	553,660	127,525

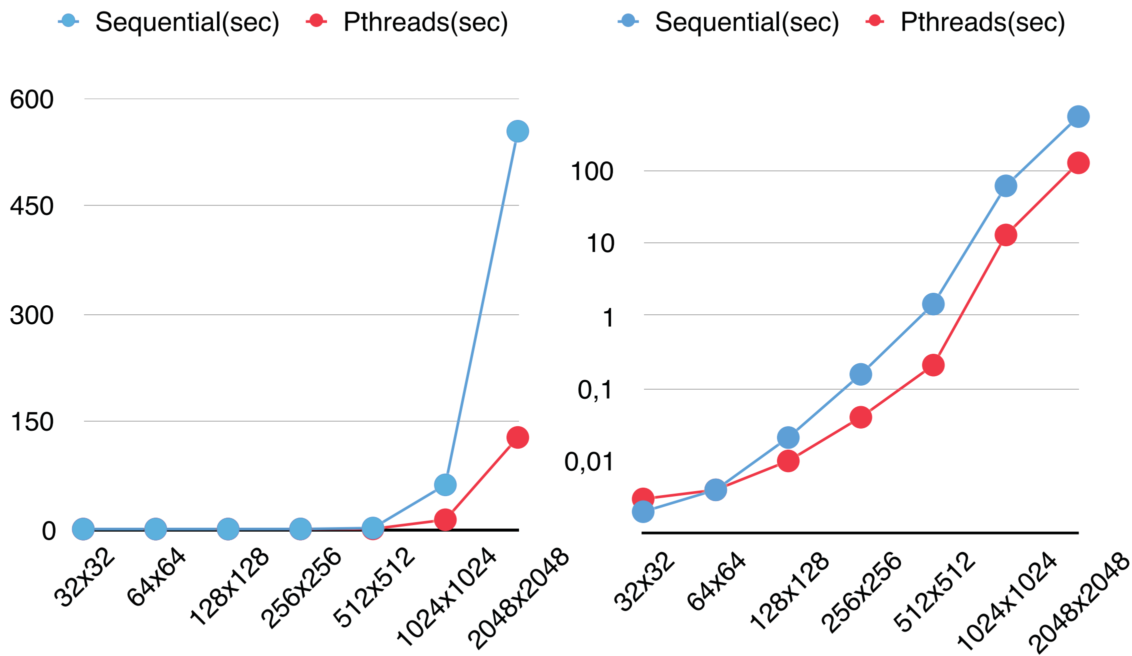


Figure 8 - Pthreads Matrix Multiplication

Left: Linear graph, Right Logarithmic graph

4.2.2.1 Quick Sort

Pthreads is performing better than the sequential version of Quick Sort right from the start of the tests at 10 million elements to the very end at 100 million elements. It was easy to create the right amount of threads and spread the work of the algorithm on the threads with Pthreads. The fact that Pthreads is pretty low level when it comes to the implementation that the programer has to do contributed a lot to the result, the implementation has to be tailored around parallelism and this shows in the results. Analysing the final measurement on 100 million element it took the sequential version ~45,5 sec, Pthread took ~8,8. Pthreads takes only ~20% of the execution time of the sequential version, it is 5 times faster. Quick Sort can really benefit from the parallelism that can be done with Pthreads, it is worth to add a bit of extra complexity to the code because it performs very well. *Figure 9* shows the test results:

	Sequential(sec)	Pthreads(sec)
10000000	3,9990122873	0,8410523927
20000000	8,3467466488	1,6802041384
30000000	12,7043459658	2,5476288502
40000000	17,266797704	3,4196113761
50000000	21,9652739951	4,3006941404
60000000	26,3655543779	5,1781134339
70000000	31,1980906666	6,4609506059
80000000	35,8601403092	7,1895013202
90000000	40,5653292126	7,8763219666
100000000	45,5124670877	8,7696634233

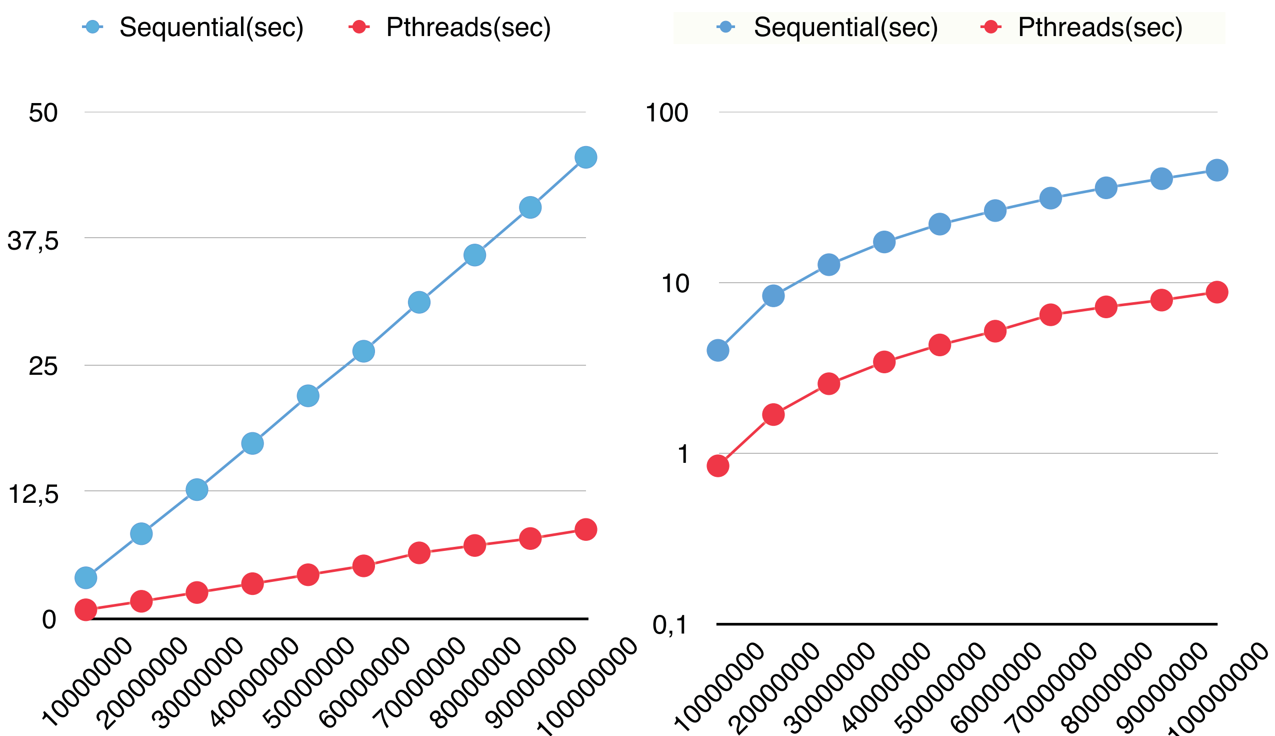


Figure 9 - Pthreads Quick Sort

Left: Linear graph, Right Logarithmic graph

4.2.1.2 Mandelbrot set

Pthreads performed very well calculating the Mandelbrot Set, at the start of the test at a resolution of 1000 x 1000 it outperformed the sequential version with over two seconds. The nature of how algorithm work made it a bit tricky to get good performance from the Pthreads implementation, splitting the data set in a good way without impacting performance to much can be a bit tricky. Pthread was faster than the sequential on all the tests that was run between the two, at 1000 x 1000 Pthreads takes ~37% of the execution time that the sequential version, at 1500 x 1500 Pthreads takes ~38% of the execution time that the sequential version and finally on 2000 x 2000 Pthreads takes around ~40% of the execution time of the sequential. On these test performed Pthreads is around 60-63% faster than the sequential version of the calculation. Pthreads did offer a good increase in performance but it also added a fair bit of complexity to the code because the division of the data set between the threads must be handled. *Figure 10* shows the test results:

	Sequential(sec)	Pthreads(sec)
1000	4,2569970239	1,5862005617
1100	5,1462506484	1,9122037006
1200	6,1271844849	2,2883103047
1300	7,1905930092	2,6691168442
1400	8,3393718567	3,6007798799
1500	9,5733589769	3,7306604849
1600	10,8916757102	4,408943543
1700	12,2946977356	5,0357689317
1800	13,7826372207	5,3826312744
1900	15,3587830872	5,7391984368
2000	17,183182009	6,9268621319

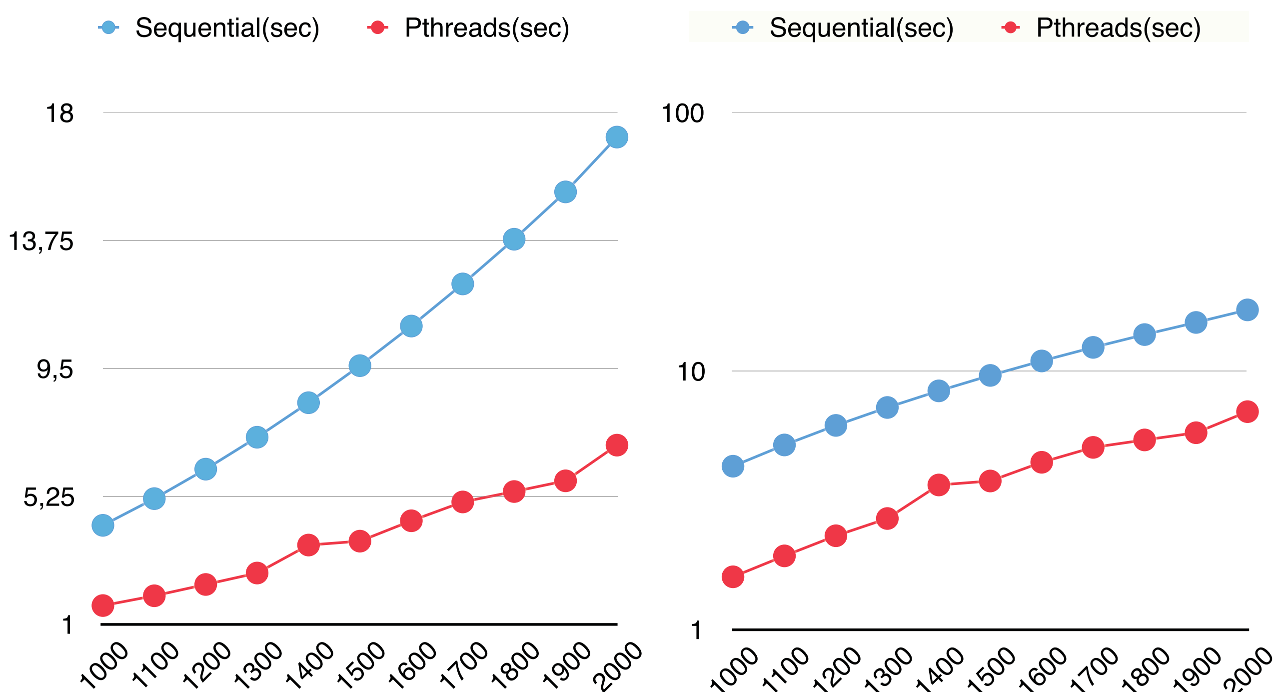


Figure 10 - Pthreads Mandelbrot set

Left: Linear graph, Right Logarithmic graph

4.2.3 Pthreads vs OpenMP

4.2.3.1 Matrix multiplication

Removing the sequential execution and just looking at OpenMP and Pthreads for matrix multiplication we can see from the first tests that on matrices with size of 1000 x 1000 elements all the way up to 2000 x 2000 matrices we don't see any major differences in execution time. Overall OpenMP is a little bit faster with a total execution time of 618,9435939603 seconds against Pthreads 626,2167442654 seconds. Each time the matrices was computed i also looked at the cache and how many cache misses occurred during each execution. In these test that was run on matrices from 1000 x 1000 to 2000 x 2000 with 100 increment after each test OpenMP had and average of 207292614 cache misses and Pthreads had an average of 212195293 cache misses, so what we can see here is that Pthreads had a little bit worst hit ratio in the cache and also a bit slower execution. *Figure 11* shows the test results:

	Pthreads(sec)	OpenMP(sec)
1000	13,4347022827	13,2202978136
1100	20,6380034526	20,7396684497
1200	23,1231403528	21,7244684168
1300	34,844475488	34,8447272505
1400	38,0107711316	36,8224829735
1500	53,6393276871	53,7207894295
1600	68,8443288695	65,0814884921
1700	78,6901331437	78,5463518172
1800	81,7559504154	81,5283362744
1900	103,999956759	103,886546986
2000	109,235954683	108,828436057

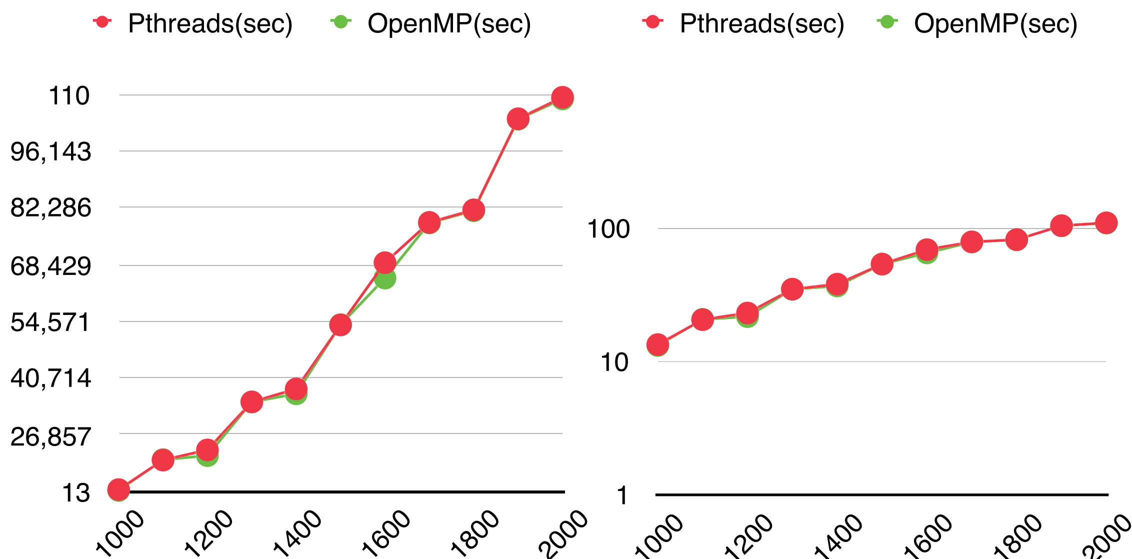


Figure 11 - Matrix Multiplication, OpenMP vs Pthreads - Medium size Matrices

Left: Linear Graph, Right: Logarithmic Graph

Since it was a very small difference between OpenMP and Pthreads on the previous test it was interesting to scale up the tests and run on them on larger matrices. This time the tests starts from 5000 x 5000 matrices and goes up to 6000 x 6000 matrices with 200 increment after each run. We can se right from the start that OpenMP has increased it's lead over Pthreads and keeps it through all the tests. The gap between OpenMP and Pthreads has become much larger than it was on the smaller matrices, now the total execution time from OpenMP is 14369,73309242 seconds and the total execution time from Pthreads is 14697,70712948, over 300 seconds slower than OpenMP. Examining the cache this time we can see just like before that OpenMP has fewer cache misses than Pthreads, OpenMP had an average of 12724779825 cache misses during the tests and Pthread had an average of 12821448229 cache misses. Here it is even more obvious that the fewer cache misses contributed to that OpenMP was a bit faster.

On 5000 x 5000 matrices OpenMP is ~2% faster than Pthreads. On the next test with 5200 x 5200 matrices OpenMP is ~2,4% faster than Pthreads. On 5400 x 5400 matrices OpenMP is ~3,2% faster than Pthreads. On 5600 x 5600 matrices OpenMP is ~3,5 % faster than Pthreads. On 5800 x 5800 matrices OpenMP is ~1,4% faster than Pthreads and lastly on 6000 x 6000 matrices OpenMP outperforms Pthreads by ~1,9%. *Figure 12* shows the test results:

	Pthreads(sec)	**OpenMP(sec)**
5000	1770,39659526	1749,70841587
5200	2033,92943583	1986,80021136
5400	2250,721892	2180,82662043
5600	2663,99373341	2570,2253889
5800	2793,27989453	2756,64751597
6000	3185,38557845	3125,52493989

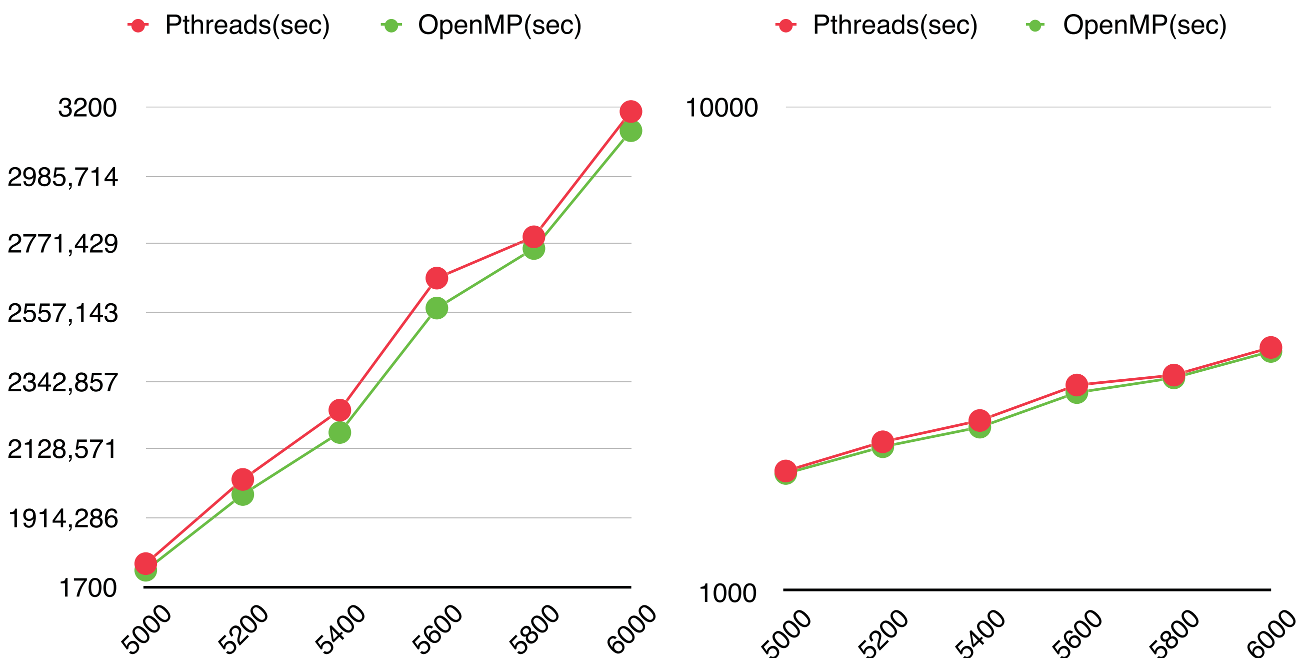


Figure 12 - Matrix Multiplication, OpenMP vs Pthreads - Large Matrices
Left: Linear Graph, Right: Logarithmic Graph

4.2.3.2 Quick sort

When looking at how Pthreads and OpenMP performed against the sequential version it was clear that Pthreads outperformed OpenMP that was also outperformed by the sequential version. When looking at why Pthreads outperformed OpenMP with so much the cache needed to be examined. There was a noticeable difference in cache misses between Pthreads and OpenMP, on average through the tests Pthreads had 6938696 cache misses while OpenMP had 27057618 cache misses. Here it is also clear that the number of cache misses has an impact on the execution time, OpenMP had around 20 million more cache misses on average. Since it was so large difference in execution time between OpenMP and Pthreads it could not be just the cache. When writing the Pthread version of Quick sort everything could be controlled, where the parallel sections is, what functions and data that should be parallelised and how. This gives the programmer a lot of control and he can really fine-tune the code, this also means that the number of threads that are spawned for a certain task can be controlled. OpenMP also has this ability to control what should be parallelised and how the data should be divided and how many threads etc. The problem with OpenMP is that it has problem to express parallelism within recursion and in an algorithm like a Quick Sort this becomes a problem since it depends on recursion heavily. The number of threads that the program created using OpenMP was outputted so it could be examined and on some runs the number of threads that was created exceeded the number of elements to be sorted. It was obvious that this was the problem, the overhead when creating a threads stack up and makes the algorithm run much slower than the Pthreads version and the Sequential version. *Figure 13* shows the test results:

	Pthreads(sec)	OpenMP(sec)
10000000	0,8410523927	7.3846038776
20000000	1,6802041384	15.2761299154
30000000	2,5476288502	21.1537746987
40000000	3,4196113761	31.7263201437
50000000	4,3006941404	40.5338859016
60000000	5,1781134339	42.1267760366
70000000	6,4609506059	50.4931627025
80000000	7,1895013202	62.8212976466
90000000	7,8763219666	66.0237372302
100000000	8,7696634233	71.7466770492

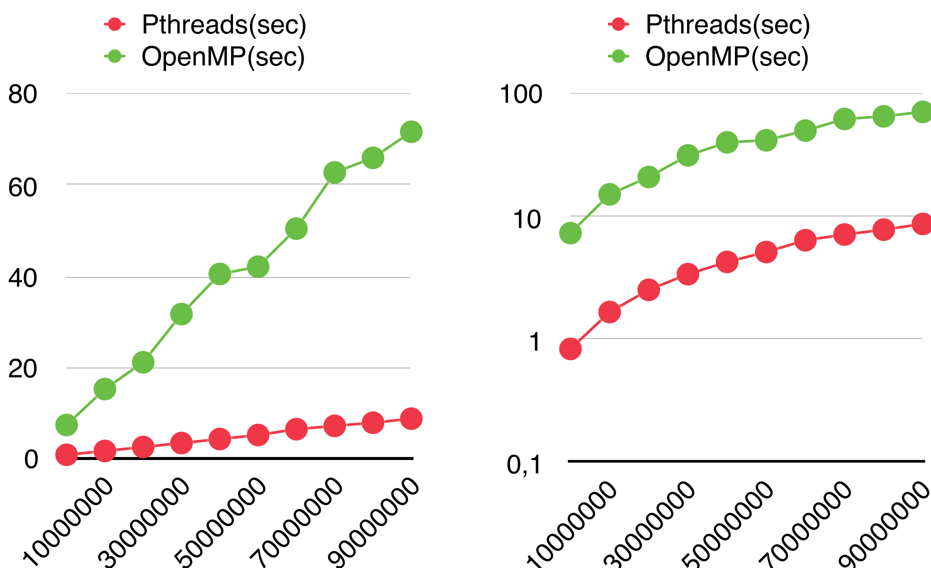


Figure 13 - Quick sort Pthreads and OpenMP

Left: Linear graph, Right: Logarithmic graph

4.2.3.2 Mandelbrot set

Both OpenMP and Pthreads performed better then the sequential version when calculating the Mandelbrot set. Mandelbrot set was a perfect candidate for parallelism because the work to calculate small parts of the set could easily be spread among the available threads. Pthreads were a bit more work to do because the division of the data set need's to be done manually whilst OpenMP does it automatically. On the test result below the Mandelbrot set is calculated for a resolution of 1000 x 1000 up to 2000 x 2000 and overall OpenMP performed better than Pthreads with a total execution time of 38,4712364508 seconds against Pthreads 43,2806760938 seconds. Measuring cache misses this time OpenMP had an average of 100114 cache misses overall and Pthreads had 107139 cache misses. It is also visible here that OpenMP had lower cache misses than Pthreads and also achieved a lower execution time than Pthreads just as it was visible on Matrix Multiplication and Quick Sort, the one with lower amount of cache misses had a faster execution time. On this rather small size OpenMP is faster but only with a small amount, ~7% faster at the first test and ~15% on the last test, OpenMP did increase it's lead when larger set was tested. *Figure 14* shows the test results:

	OpenMP(sec)	Pthreads(sec)
1000	1,4870022712	1,5862005617
1100	1,8097723319	1,9122037006
1200	2,1347673174	2,2883103047
1300	2,5219422388	2,6691168442
1400	2,955996154	3,6007798799
1500	3,3503613675	3,7306604849
1600	3,8258463875	4,408943543
1700	4,2979192986	5,0357689317
1800	4,8078042232	5,3826312744
1900	5,3596429662	5,7391984368
2000	5,9201818945	6,9268621319

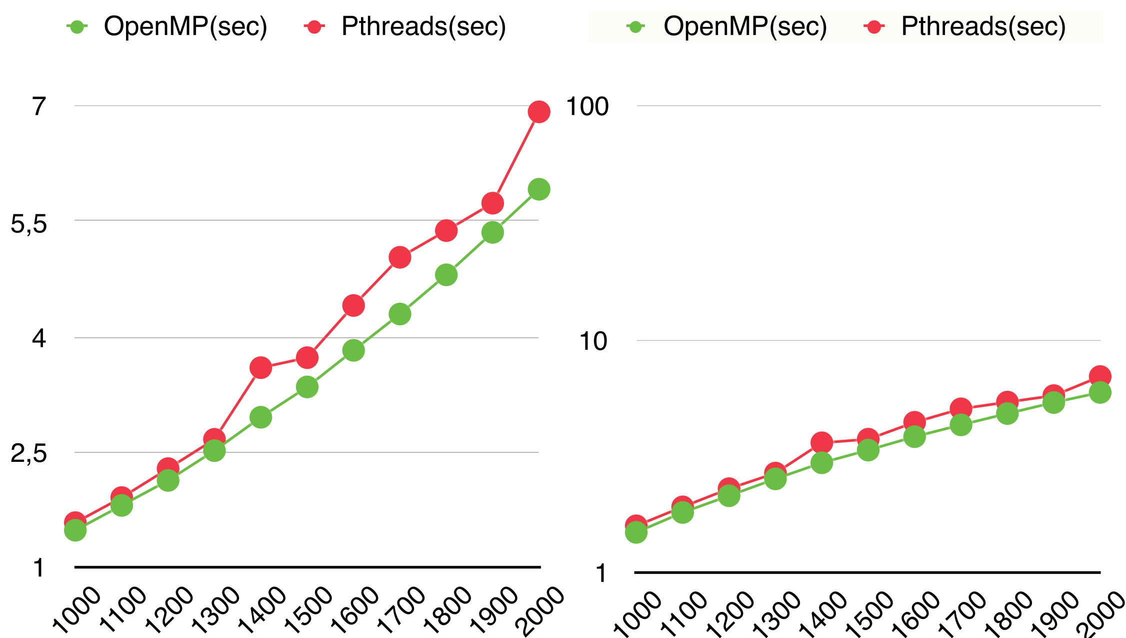


Figure 14 - Mandelbrot Pthreads and OpenMP

Left: Linear graph, Right: Logarithmic graph

4.2.4 Effort

Implementing the sequential version of matrix multiplication required 89 lines, this includes the whole program from start to finish. Introducing parallelism to the code increased the required amount of lines and added a complexity to the source code. OpenMP is much easier to work with and only required five additional lines to parallel the program, Pthreads required a much more tailored version to be able to work. Analysing this result and the result when comparing the performance between OpenMP and Pthreads we see that OpenMP requires much less effort to implement a Matrix Multiplication algorithm and achieves better performance than Pthreads. *Figure 15* shows the test results:

Sequential	Pthreads	OpenMP
89	121	94

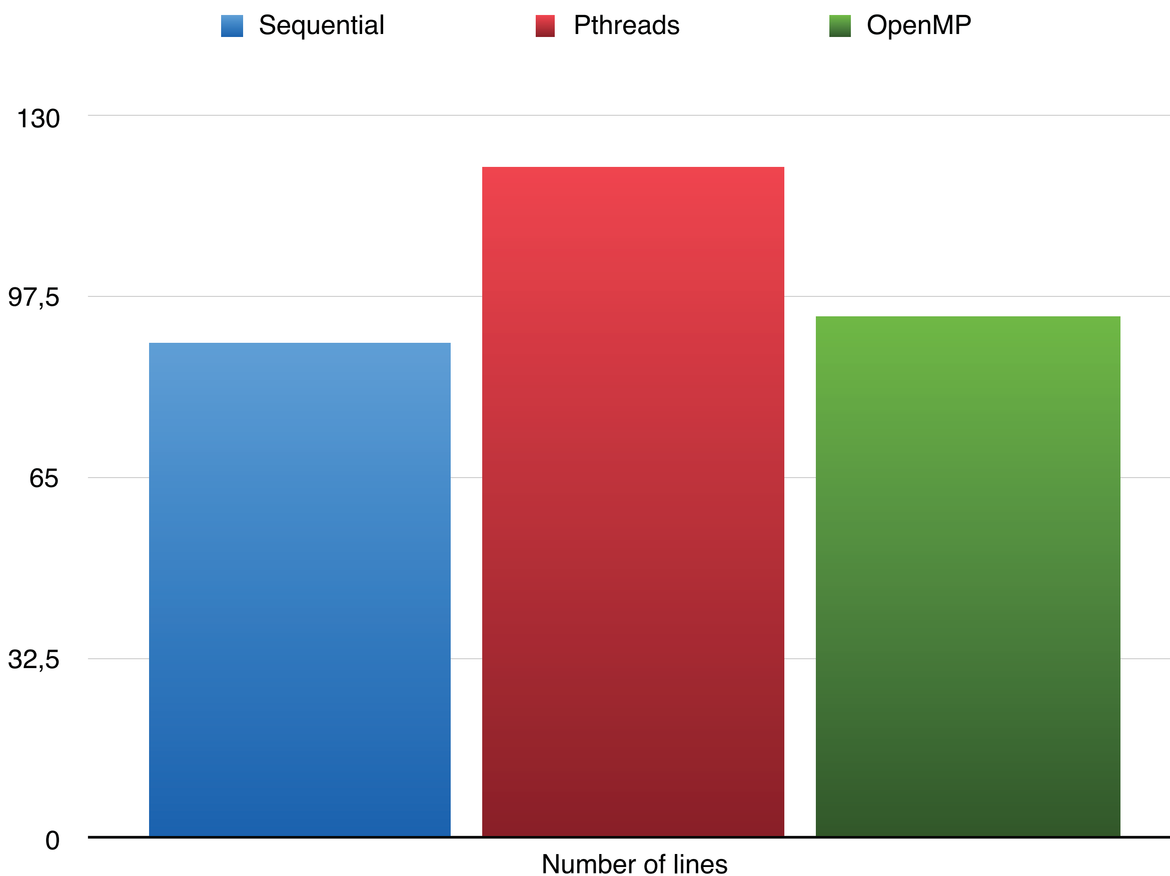


Figure 15- Matrix Multiplication Effort

The sequential version of Quick Sort required 76 lines of code to implement, introducing parallelism to this program required a bit more modifications of the sequential source code than Matrix Multiplication required. Building the parallel version using Pthreads required 210 lines of code, OpenMP required only 91. For Quick Sort OpenMP performed much worse than Pthreads, so even though Pthreads requires a much more tailored program and a bigger program Pthreads is to recommend because it achieves much better performance, so much that the extra effort to implement is worth it. OpenMP is not worth using since it requires more effort than the sequential

version but performance worse than a sequential Quick Sort. The high level nature of OpenMP did not let me to express parallelism in an effective way in recursion. *Figure 16* shows the test results:

Sequential	Pthreads	OpenMP
76	210	91

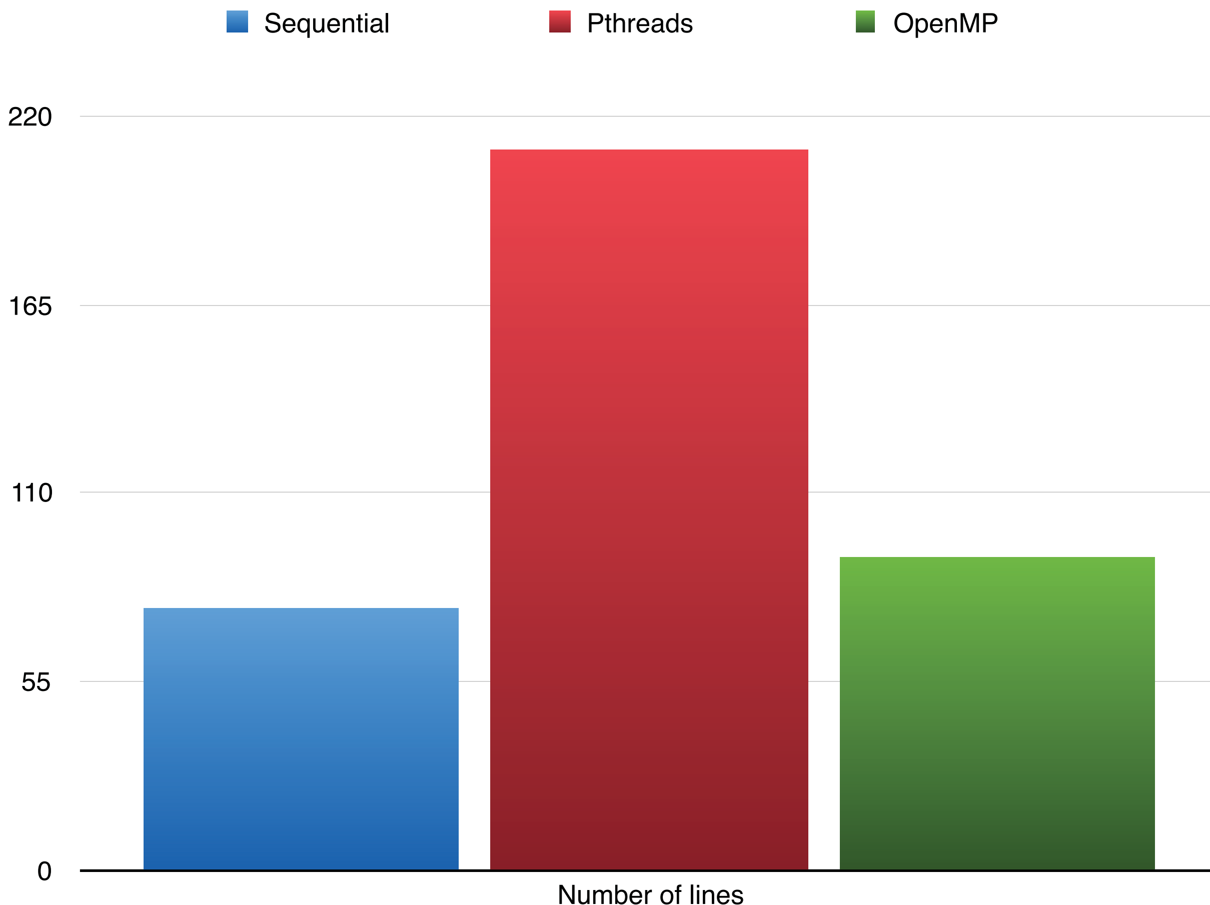


Figure 16 - Quick Sort Effort

The sequential version of calculating the Mandelbrot set required 62 lines of code, introducing parallelism to this program using Pthreads required a total of 109 lines of code, OpenMP only required a total of 68 lines. OpenMP was not limited like it was on Quick Sort and we see that it is performing better than Pthreads and requires less modification of the sequential source code. For programs that is working with calculation like this it is more effective to use OpenMP than Pthreads. Much like in Matrix Multiplication Pthreads requires so much work of the programmer to distribute the workload between the threads, OpenMP handles this automatically and is more optimised than most would be able to do using Pthreads in a simple implementation. *Figure 17* shows the test results:

Sequential	Pthreads	OpenMP
62	109	68

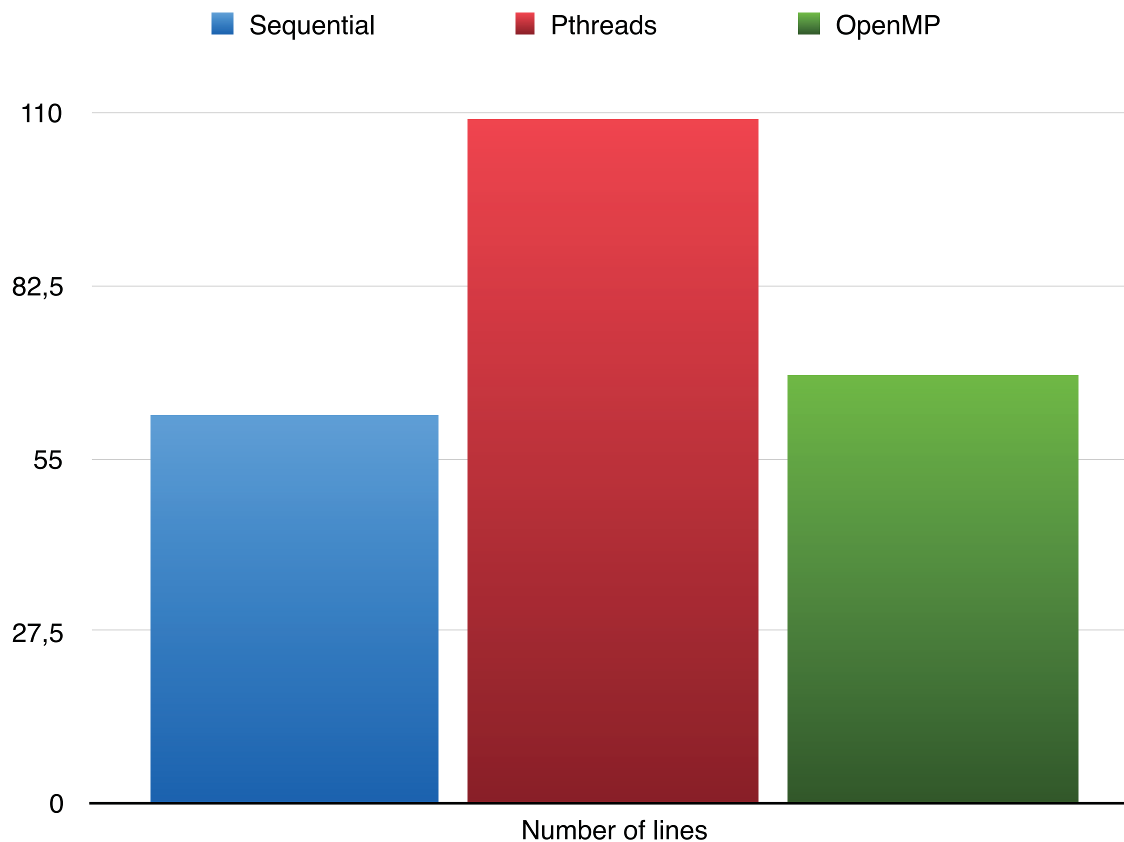


Figure 17 - Mandelbrot Set Effort

4.2.5 Findings

- OpenMP is very capable to help computation intensive algorithms to take advantage of a multi-core architecture, in most cases. It does not perform well when working with recursion, it has problem with expressing clear instructions for a recursive problem. It increased the execution speed for two of the three tested algorithms compared to the sequential version.
- Pthreads is very capable to help computation intensive algorithms to take advantage of a multi-core architecture. It increased the execution speed for all tested algorithms compared to the sequential version.
- The OpenMP version of Matrix Multiplication and Mandelbrot set calculations outperformed the Pthreads versions. Pthreads outperformed OpenMP on the recursive Quick Sort.
- Using OpenMP resulted in fewer cache-misses than with Pthreads.
- It is not worth paralleling Matrix Multiplication for small input sets, adds unnecessary complexity to the source code and can actually decrease performance.
- Pthreads required the most effort to implement, OpenMP required only small modifications to the source code while Pthreads required a totally tailored source code.

5 Analysis

Calculating matrix multiplications OpenMP did perform better than Pthreads and i think that besides that it had lower amount of cache misses than Pthreads version OpenMP also had lower overhead dividing the work among the threads and thread synchronisation. [25] did a paper using OpenMP against MPI to calculate a version of matrix multiplication and one of the reasons they found that made OpenMP faster than MPI was the thread synchronisation and task dividing. This does not prove that OpenMP is better than Pthreads at this but it does prove that OpenMP is very good at it. Both OpenMP and Pthreads were effective at improving a sequential algorithm making it parallel, OpenMP do make the work a whole lot easier because it does not require that you tailor the program around parallel execution, Pthreads do. I though this made the whole experience working with Pthreads much worse than the experience i had working with OpenMP, Pthreads requires much more effort and did not perform better than OpenMP. A note here is that both the OpenMP version and Pthreads version could probably be optimised to perform better but this is what i managed to do with the available time.

Looking at the calculations of the Mandelbrot set we see similar results that we see in the Matrix Multiplication, OpenMP is a bit faster than Pthreads overall and has fewer cache misses. Just as on the Matrix Multiplication i think that OpenMP benefits from good data division and thread synchronisation, better than what i was able to implement in Pthreads and this together with the fewer cache misses results in that OpenMP is a bit faster. It did not require much effort at all to implement using OpenMP and just as in the other algorithms Pthreads require the program to be tailored towards threading. I think that this is a major selling point of OpenMP that should convince a lot of people to use it, lower implementation effort and better performance is a very interesting combination. [13] experiments shows that they get similar results using OpenMP and Pthreads to implement the same algorithm but that OpenMP's version is much easier to read, understand, implement and maintain. This somewhat contradicts my result and also confirms it, i got a little better effort on two of the three algorithms i tested, not the same result, but just as they concluded the OpenMP versions of the programs is much easier to work with.

When testing Quick Sort got a surprising result, i thought that Pthreads would be faster than OpenMP because Pthreads is a more low level API towards threading, I actually thought that it would be on all the tests, but i did not think that OpenMP version of Quick Sort would lose against both Pthreads and the Sequential version of the algorithm. [15] found that it was hard to work with OpenMP and recursion because it is hard to express parallelism in it, because i chose a Quick Sort that used recursion this was a problem. The implementation i used did not perform well at all, it had a major flaw that each time the recursive function would be called a new thread is spawned that handles the new level of recursion. This means that the algorithm tries to start threads for each partition which results in almost as many threads as there is elements to be sorted. I made the same conclusion as [15] that OpenMP is not suitable for such problems and if it must be used the problem needs to be redesigned to not use recursion. I would use Pthreads for a problem like this because it let's the programmer express parallelism in recursion very clearly, which OpenMP does not.

All the OpenMP programs required lesser effort than Pthreads did and the measurements for effort was the line of code required. I chose this kind of measurement because i feel that it gives a number on the effort, else effort can be hard to define because it can be highly individual. [13,14,15] all touches the subject effort and productivity and talk about it, they don't say it directly that they measure it in lines of code but i understood that in their context it was the amount of lines in the code. That's why i think my method to measure the effort is a valid method and can be used.

A behaviour that was noticed during the tests was that on some sizes of the input set the algorithm did not scale as well on some other sizes. It could not be proven but it seems to be connected to the set sizes and the number of threads available, the test machine has eight cores and therefor the input set is sometimes divided by eight. If the input set divisible by eight the algorithm seems to scale just fine, but if it is not divisible by eight, if the result of the division is a float, the algorithm did not scale as well, this behaviour was noticed on both OpenMP and Pthreads.

Summary

RQ1: What results has other research found about comparison between OpenMP and Pthreads?

The three papers i found that actually made a comparison between the two frameworks, found that OpenMP is much easier to work with and maintain and does not give lower performance, actually it outperformed Pthreads in some cases that [14] did test but in some other cases it does not just as [27] and my results proves, so the type of application is very important when choosing the right framework.

There was not very much earlier work on just performance measurements between OpenMP and Pthreads so this can make it harder to prove that my result is correct. The results that i got during the tests is for a specific machine with a specific code, results may vary on other environments.

RQ2: Can OpenMP help computation intensive algorithms perform better on multi-core system?

OpenMP were very efficient at increasing the performance when paralleling the algorithms compared to a sequential version, in most cases. OpenMP did not perform well at all when paralleling the recursive Quick Sort. The way OpenMP is build makes it hard to express parallelism in recursion using OpenMP and because of this the performance was affected. [15] talks about this problem in their paper and do suggest to make the algorithm sequential as a solution.

RQ3: Can Pthreads help computation intensive algorithms perform better on multi-core system?

Pthreads were very effective at increasing the performance when paralleling the algorithms compared to a sequential version. It added a lot of complexity to the source code but also increased the performance with a lot.

RQ4: How does the efficiency differ between OpenMP and Pthreads for computation intensive algorithms on multi-core systems?

OpenMP did perform a bit better than Pthreads on two of the algorithms, Matrix Multiplication and calculating the Mandelbrot set. On the recursive Quick Sort Pthreads was a clear winner because it was much easier to control Pthreads for this algorithm than OpenMP was. OpenMP spun up to many threads and with the overhead of creating threads the execution time was affected a lot.

RQ2-RQ4

The environment where the test were performed is a computer shared among many students that can use it at the same time, this means there is no guarantee that when my tests was run i had access to 100% of the computers resources. I tried to counter this by running many tests at different times and take an average of the results.

RQ5: How much efforts is required of the developer to implement algorithms in OpenMP and Pthreads?

OpenMP did require much less effort to implement the three algorithms and do perform better on two of them. OpenMP versions of the algorithms only need to add 4-5 lines of code while Pthreads

version adds at least 50+ lines. This makes me think which one should be used, and it does look like OpenMP is the winner. But since OpenMP did perform very badly at the recursive Quick Sort Pthreads should be used when working with recursive code that is executing parallel.

The work that was found that talks about OpenMP and productivity does not talk about how they measure the productivity, just that OpenMP does not require much effort to achieve parallelism. Productivity with Pthreads is not much described either, only a few describes that it is a low level framework that requires a very specific program. The lack of other work with effort and productivity between Pthreads and OpenMP can make my result harder to validate because the number of lines is not necessarily the best way of measuring effort. I chose it because it give a number on the effort that can be used to measure the amount of modification of the source code that was needed.

6 Conclusion and Future Work

6.1 Conclusion

This paper has talked about OpenMP and Pthreads performance computational intensive algorithms in the C programming language. Tests has been performed on parallel execution versus sequential execution to see how the frameworks scale and also on sequential versus parallel execution to see how the frameworks perform against each other. The test consisted of three different algorithms, Quick Sort, Matrix Multiplication and calculation of the Mandelbrot set and all the algorithms was implemented in three versions, sequential, Pthreads and OpenMP. The different algorithms was tested with datasets of different sizes containing randomly generated data, the different set sizes was not random but controlled, it started on a size and after each test run the set size was incremented with a value. This gave multiple data points that could be used to show how the algorithm performed over a collection of sets in a graph. The effort that was required to implement the algorithms using the two different frameworks was also compared, the effort was measured in the number of lines in the final source code. This was then used together with the performance to decide which of the frameworks is suitable in what situation.

The tests show that for small set it is not worth paralleling at all because the sequential version of the algorithms was faster, it just adds unnecessary complexity to the code. On larger sets the results show that OpenMP is performing better than Pthreads on Matrix Multiplication and calculation of the Mandelbrot set, but not on Quick Sort. On Quick Sort OpenMP was performing really bad due to problems was recursion and parallel execution in OpenMP, it did not even perform better than the sequential version of the algorithm, Pthreads on the other hand did perform very well on Quick Sort. On all three of the algorithms OpenMP was the winner when looking at just the effort but it was only the winner on two of the algorithms when looking at the performance and effort combined. On Matrix Multiplication and Mandelbrot set OpenMP was both better in performance and required less effort than Pthreads, but on Quick Sort OpenMP could not be recommended since it performed so bad. For Quick Sort Pthreads was the winner even though it required much modification of the source code.

Overall OpenMP is to recommend over Pthreads because it offer better performance in most of the cases that was examined in this study. OpenMP's high level approach to threading contributed to that it required less effort to implement the algorithms and maintenance of the code was much easier, this is also a major reason why i recommend to use it over Pthreads. The only time i recommend Pthreads over OpenMP is when the parallel part will require Pthreads.

6.2 Future Work

An area that could be explored within comparisons between OpenMP and Pthreads is to test what they are capable of building, i mean try to build real world applications like web-servers, chat applications etc. There are two aspects to check when doing that comparisons, first if they are capable of being used to build that application and if they are the performance that is possible to achieve.

I would have liked to test Pthreads and OpenMP against the threads implementation that is available in the new version of C++. This would also test how Pthreads and OpenMP is able to function in an object oriented environment. Test could also be performed on the same program written in C and then in C++ using Pthreads or OpenMP to do even further investigation if the object oriented environment impact the performance of the libraries.

References

1. Moore's law: the future of Si microelectronics, Scott E. Thompson, Srivatsan Parthasarathy, June 2006
   http://ac.els-cdn.com/S1369702106715395/1-s2.0-S1369702106715395-main.pdf?_tid=2c23d742-dee4-11e5-a061-00000aab0f6b&acdnat=1456750907_9c115ce84ccc9526d08983102e58c78c
2. Balaji Venu. Multi-core processors - An overview.
   http://arxiv.org/pdf/1110.3535.pdf
3. The Free Lunch is Over, A Fundamental turn towards Concurrency in Software, Herb Sutter
   http://mondrian.die.udec.cl/~mmedina/Clases/ProgPar/Sutter%20-%20The%20Free%20Lunch%20is%20Over.pdf
4. Tiobe Software. 2016. Top programming language of 2015.
   http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
5. Linux Manual. 2015. PThreads, POSIX Threads Manual.
   http://man7.org/linux/man-pages/man7/pthreads.7.html
6. OpenMP Architecture Review Board. 2013. OpenMP API Version 4.0.
   http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
7. torvalds/linux
   https://github.com/torvalds/linux
8. History of C Programming language, 2014, Myriam Grandchamp, Saraswathi Kaja, Sourav Chandra, Oscar Serrano Serrano
   http://www.peoi.org/Courses/Coursesen/cprog/fram1.html
9. Features of C Programming Language
   www.c4learn.com/c-programming/c-features
10. C - Data Type, Tutorialspoint
    www.tutorialspoint.com/cprogramming/c_data_types.html
11. PThreads Programming: A POSIX Standard for Better Multiprocessing, Bradford Nichols,Dick Buttlar,Jacqueline Farrell, 1996, pages.1-26
    https://books.google.se/books?hl=sv&lr=&id=oMtCFSnvwmoC&oi=fnd&pg=PR5&dq=Pthreads&ots=QNaTaoulU7&sig=PgVf_phIX3gxNo7OtBpvowCqVyE&redir_esc=y#v=onepage&q&f=false
12. Parallel Programming in OpenMP, Rohit Chandra Leonardo Dagum Dave Kohr Dror Maydan Jeff McDonald Ramesh Menon, 2001, ISBN 1-55860-671-8
13. OpenMP versus Threading in C/C++, Bob Kuhn, Paul Petersen
    http://www.cs.colostate.edu/~cs675/OpenMPvsThreads.pdf
14. Scientific computations on multi-core systems using different programming frameworks, Panagiotis D. Michailidis, Konstantinos G. Margaritis
    http://www.sciencedirect.com.miman.bib.bth.se/science/article/pii/S016892741400213X

15. Michael Su¨ß and Claudia Leopold, A User's Experience with Parallel Sorting and OpenMP http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.5003&rep=rep1&type=pdf
16. Linux Manual. 2015. Time
http://man7.org/linux/man-pages/man2/clock_gettime.2.html
17. The Single UNIX ® Specification, Version 2, The Open Group
http://pubs.opengroup.org/onlinepubs/007908775/xsh/time.h.html
18. Data Structures and Algorithms in C++, Adam Drozdek, Cengage Learning 2013, p.512-518 ISBN-13: 978-1-133-61305-3, ISBN-10: 1-133-61305-5
19. Matrix Multiplication, WolframMathWorld,
http://mathworld.wolfram.com/MatrixMultiplication.html
20. Mandelbrot set, WolframMathWorld,
http://mathworld.wolfram.com/MandelbrotSet.html
21. Complex Plane, WolframMathWorld,
http://mathworld.wolfram.com/ComplexPlane.html
22. Julia Set, WolframMathWorld
http://mathworld.wolfram.com/JuliaSet.html
23. Superlinear Speedup in Parallel Computation, Jing Shan,
http://www.ccs.neu.edu/course/com3620/projects/scalable/jshan/final1.pdf
24. Jonathan L. Bentz, Ricky A. Kendall Parallelization of General Matrix Multiply Routines Using OpenMP, Shared Memory Parallel Programming with OpenMP, May 2004, pp.1-12
25. Seon Wook Kim, Hyeong Soo Chang, Parallelizing Parallel Rollout Algorithm for Solving Markov Decision Processes, Shared Memory Parallel Programming with OpenMP,
26. Russell Brown, Ilya Sharapova, High-Scalability Parallelization of a Molecular Modeling Application: Performance and Productivity Comparison Between OpenMP and MPI Implementations Nov 2006
http://uv3sv3ds3g.search.serialssolutions.com/?ctx_ver=Z39.88-2004&ctx_enc=info%3Aofi%2Fenc%3AUTF-8&rfr_id=info:sid/summon.serialssolutions.com&rft_val_fmt=info:ofi/fmt:kev:mtx:journal&rft.genre=article&rft.atitle=High-Scalability+Parallelization+of+a+Molecular+Modeling+Application%3A+Performance+and+Productivity+Comparison+Between+OpenMP+and+MPI+Implementations&rft.jtitle=International+Journal+of+Parallel+Programming&rft.au=Brown%2C+Russell&rft.au=Sharapov%2C+Ilya&rft.date=2007-09-10&rft.issn=0885-7458&rft.eissn=1573-7640&rft.volume=35&rft.issue=5&rft.spage=441&rft.epage=458&rft_id=info:doi/10.1007%2Fs10766-007-0057-y&rft.externalDBID=n%2Fa&rft.externalDocID=10_1007_s10766_007_0057_y¶mdict=en-US
27. Wei Zhong, Gulsah Altun, Xinmin Tian, Robert Harrison, Phang C. Tai, Yi Pan, Parallel protein secondary structure prediction schemes using Pthread and OpenMP over hyper-threading technology, 2007
http://uv3sv3ds3g.search.serialssolutions.com/?ctx_ver=Z39.88-2004&ctx_enc=info%3Aofi%2Fenc%3AUTF-8&rfr_id=info:sid/summon.serialssolutions.com&rft_val_fmt=info:ofi/fmt:kev:mtx:journal&rft.genre=article&rft.atitle=Parallel+protein+secondary+structure+prediction+schemes+using+Pthread+and+OpenMP+over+hyper-threading+technology&rft.jtitle=The+Journal+of+Supercomputing&rft.au=Zhong%2C+Wei&rft.au=Altun%2C+Gulsah&rft.au=Tian%2C+Xinmin&rft.au=Harrison%2C+Robert&rft.date=2007-07-01&rft.pub=Kluwer+Academic+Publishers-Plenum+Publishers&rft.issn=0920-8542&rft.eissn=1573-0484&rft.volume=41&rft.issue=1&rft.spage=1&rft.epage=16&rft_id=info:doi/10.1007%2Fs11227-007-0100-1&rft.externalDBID=n%2Fa&rft.externalDocID=2007_11227_41_1_100¶mdict=en-US
28. Linux Man Page, Perf stat,
http://linux.die.net/man/1/perf-stat

Appendix A. Code

Matrix Multiplication

Sequential

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 32

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

void init_matrix(void) {

    int i, j;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++) {
               a[i][j] = rand() % 10;
               b[i][j] = rand() % 10;
        }
}

void matmul_seq() {

    int i, j, k;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            c[i][j] = 0.0;
            for (k = 0; k < SIZE; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}

void print_matrix(void) {

    int i, j;

    printf("Matrix A\n");
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf(" %7.2f", a[i][j]);
        }
        printf("\n");
    }

    printf("Matrix B\n");
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf(" %7.2f", b[i][j]);
        }
        printf("\n");
    }

    printf("Matrix C\n");
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf(" %7.2f", c[i][j]);
        }
        printf("\n");
    }
}

int main(int argc, char **argv) {

    srand((unsigned) time(NULL));
    struct timespec initStart, initEnd;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &initStart);
    init_matrix();
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &initEnd);
    printTimespec(initStart, initEnd, "initialize");

    removeResultFile();

    int i = 0;
    for(; i < 100; ++i) {
        struct timespec mulStart, mulEnd;
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &mulStart);
        matmul_seq();
```

```
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &mulEnd);
        printTimespec(mulStart, mulEnd, "matrix multiply");
        logResult(mulStart,mulEnd);
    }

    if(SIZE <= 16) {
        print_matrix();
    }
}
```

Pthreads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define SIZE 6000
#define MAX_THREADS 8

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

typedef struct {
        int startX;
}Args;

pthread_t tid[MAX_THREADS];

void * worker(void *);

static void init_matrix(void) {

    int i, j;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
                a[i][j] = rand() % 10;
                b[i][j] = rand() % 10;
        }
    }
}

static void matmul_par() {

        int i = 0;
        int xCounter = 0;
        int yCounter = 0;
        int threadCounter = 0;

        for(; i < MAX_THREADS; ++i) {
                Args * args = malloc(sizeof(* args));
                args->startX = xCounter;
                xCounter += SIZE/MAX_THREADS;
                pthread_create(&tid[threadCounter++], 0, worker, args);
        }

        int j = 0;
        for(; j < MAX_THREADS; ++j) {
                pthread_join(tid[j], NULL);
        }
}

void* worker(void * args) {

        Args * arg = args;
        int i,j,k;
        int endX = arg->startX + SIZE/MAX_THREADS;

        for(i = arg->startX; i < endX; ++i) {
                for(j = 0; j < SIZE; ++j) {
                        c[i][j] = 0.0;
                        for(k = 0; k < SIZE; ++k) {
                                c[i][j] = c[i][j] + a[i][k] * b[k][j];
                        }
                }
        }
        free(arg);
        pthread_exit(0);
}

void print_matrix(void) {

    int i, j;

    printf("Matrix A\n");
    for (i = 0; i < SIZE; i++) {
```

```
        for (j = 0; j < SIZE; j++) {
            printf(" %7.2f", a[i][j]);
        }
        printf("\n");
    }
    printf("\n");

    printf("Matrix B\n");
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf(" %7.2f", b[i][j]);
        }
        printf("\n");
    }
    printf("\n");

    printf("Matrix C\n");
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf(" %7.2f", c[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

int main(int argc, char **argv) {

    srand((unsigned) time(NULL));
    init_matrix();

    removeResultFile();

    int i = 0;
    for(; i < 10; ++i) {
        struct timespec mulStart, mulEnd;
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &mulStart);
        matmul_par();
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &mulEnd);
        printTimespec(mulStart, mulEnd, "matrix multiply");
        logResult(mulStart, mulEnd);
    }

    if(SIZE <= 16) {
        print_matrix();
    }
}
```

OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#define SIZE 10000

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

void init_matrix(void) {

    int i, j;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++) {
               a[i][j] = rand() % 10;
               b[i][j] = rand() % 10;
        }
}

void matmul_seq() {

    int i, j, k;

    #pragma omp parallel shared(a,b,c) private(i,j,k) num_threads(8)
    {
        #pragma omp for schedule (static)
        for (i = 0; i < SIZE; i++) {
            for (j = 0; j < SIZE; j++) {
                c[i][j] = 0.0;
                for (k = 0; k < SIZE; k++)
                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }
```

```c
}

void print_matrix(void) {

    int i, j;

    printf("Matrix A\n");
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf(" %7.2f", a[i][j]);
        }
        printf("\n");
    }

    printf("Matrix B\n");
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf(" %7.2f", b[i][j]);
        }
        printf("\n");
    }

    printf("Matrix C\n");
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf(" %7.2f", c[i][j]);
        }
        printf("\n");
    }
}

int main(int argc, char **argv) {

    srand((unsigned) time(NULL));
    init_matrix();
    removeResultFile();

    int i = 0;
    for(;i < 10; ++i) {
        struct timespec mulStart, mulEnd;
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &mulStart);
        matmul_seq();
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &mulEnd);
        printTimespec(mulStart, mulEnd, "matrix multiply");
        logResult(mulStart, mulEnd);
    }

    if(SIZE <= 16) {
        print_matrix();
    }
}
```

Quick sort

Sequential

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_ITEMS 100
#define swap(v, a, b) {unsigned tmp; tmp=v[a]; v[a]=v[b]; v[b]=tmp;}

static int *array;

static void print() {
    int i;

    for (i = 0; i < MAX_ITEMS; i++) {
        printf("%d\n", array[i]);
    }
    printf("\n");
}

static void init() {
    int i;

    array = (int *) malloc(MAX_ITEMS*sizeof(int));
    for (i = 0; i < MAX_ITEMS; i++) {
        array[i] = rand();
         }
}

static unsigned partition(int *array, unsigned low, unsigned high, unsigned pivot_index) {

    if (pivot_index != low)
        swap(array, low, pivot_index);

    pivot_index = low;
    low++;

    while (low <= high) {
        if (array[low] <= array[pivot_index])
            low++;
        else if (array[high] > array[pivot_index])
            high--;
        else
            swap(array, low, high);
    }

    if (high != pivot_index)
        swap(array, pivot_index, high);
    return high;
}

static void quick_sort(int *array, unsigned low, unsigned high) {

    unsigned pivot;

    if (low >= high)
        return;
    pivot = (low+high)/2;
    pivot = partition(array, low, high, pivot);

    if (low < pivot)
        quick_sort(array, low, pivot-1);
    if (pivot < high)
        quick_sort(array, pivot+1, high);
}

int main(int argc, char **argv) {

    int i = 0;
    removeResultFile();
    for(; i < 10; i++) {
         init();
         struct timespec quickStart, quickEnd;
        clock_gettime(CLOCK_REALTIME, &quickStart);
         quick_sort(array, 0, MAX_ITEMS-1);
         clock_gettime(CLOCK_REALTIME, &quickEnd);
        printTimespec(quickStart, quickEnd, "quicksorting");
        logResult(quickStart,quickEnd);
    }
         logOrder(array, MAX_ITEMS);
}
```

Pthreads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX_ITEMS 100000000
#define MAX_THREADS 8
#define SET_SIZE (MAX_ITEMS/MAX_THREADS)
#define swap(v, a, b) {unsigned tmp; tmp=v[a]; v[a]=v[b]; v[b]=tmp;}

static int *v;
pthread_t tid[MAX_THREADS];

typedef struct {
        int end;
        int start;
} Args;

typedef struct {
        unsigned leftStart;
        unsigned rightStart;
        unsigned setSize;
} MergeArgs;

static void print_array(void) {
    int i;

        for(i = 0; i < MAX_ITEMS; i++) {
                printf("%d\n", v[i]);
        }
    printf("\n");
}

static void init_array(void) {
        int i;

        v = (int *) malloc (MAX_ITEMS*sizeof(int));
        for (i = 0; i < MAX_ITEMS; i++) {
                v[i] = rand();
        }
}

static unsigned partition(int *v, unsigned low, unsigned high, unsigned pivot_index) {
    if (pivot_index != low)
        swap(v, low, pivot_index);

    pivot_index = low;
    low++;

    while (low <= high) {
        if (v[low] <= v[pivot_index])
            low++;
        else if (v[high] > v[pivot_index])
            high--;
        else
            swap(v, low, high);
    }

    if (high != pivot_index)
        swap(v, pivot_index, high);
    return high;
}

static void quick_sort(int *v, unsigned low, unsigned high) {
    unsigned pivot_index;
    if (low >= high)
        return;

    pivot_index = (low+high)/2;
    pivot_index = partition(v, low, high, pivot_index);

    if (low < pivot_index)
        quick_sort(v, low, pivot_index-1);
    if (pivot_index < high)
        quick_sort(v, pivot_index+1, high);
}

void * sortWorker(void * arg) {

        Args args = *((Args *) arg);
        quick_sort(v, args.start, args.end-1);
        free(arg);
        pthread_exit(0);
}

void * mergeWorker(void * arg) {

        MergeArgs args = *((MergeArgs *) arg);
```

```
        int * left = v + args.leftStart;
        int * right = v + args.rightStart;
        int * temp = malloc((args.setSize * 2)*sizeof(int));
        int size = args.setSize;
        int i,j,k;
        i = j = k = 0;
        while(i < size && j < size) {
                if(left[i] <= right[j]) {
                        temp[k++] = left[i++];
                }
                else {
                        temp[k++] = right[j++];
                }
        }
        while(i < size)
                temp[k++] = left[i++];
        while(j < size)
                temp[k++] = right[j++];

        i = 0;
        int s = args.leftStart;
        for(; i < args.setSize * 2; i++) {
                v[s++] = temp[i];
        }
        free(arg);
        free(temp);
        pthread_exit(0);
}

int main(int argc, char **argv) {

    int j = 0;
    removeResultFile();
         for(; j < 10; j++) {
            init_array();
                struct timespec quickStart, quickEnd;
        clock_gettime(CLOCK_REALTIME, &quickStart);
                int i = 0;
                for(; i < MAX_THREADS; i++) {
                        Args * args = malloc(sizeof(* args));
                        args->start = SET_SIZE * (i+1) - SET_SIZE;
                        args->end = SET_SIZE * (i+1);
                        pthread_create(&tid[i], 0, sortWorker, args);
                }
                i = 0;
                for(; i < MAX_THREADS; i++) {
                        pthread_join(tid[i], NULL);
                }

                i = 0;
                for(; i < MAX_THREADS/2; i++) {
                        if(i == 0) {
                                MergeArgs * args = malloc(sizeof(* args));
                   args->leftStart = 0;
                   args->rightStart = SET_SIZE;
                   args->setSize = SET_SIZE;
                                pthread_create(&tid[i], 0, mergeWorker, args);

                        }
                        else if(i == 1) {
                                MergeArgs * args = malloc(sizeof(* args));
                                args->leftStart = SET_SIZE * 2;
                                args->rightStart = SET_SIZE * 3;
                                args->setSize = SET_SIZE;
                                pthread_create(&tid[i], 0, mergeWorker, args);
                        }
                        else if(i == 2) {
                                MergeArgs * args = malloc(sizeof(* args));
                                args->leftStart = SET_SIZE * 4;
                                args->rightStart = SET_SIZE * 5;
                                args->setSize = SET_SIZE;
                                pthread_create(&tid[i], 0, mergeWorker, args);
                        }
                        else {
                                MergeArgs * args = malloc(sizeof(* args));
                                args->leftStart = SET_SIZE * 6;
                                args->rightStart = SET_SIZE * 7;
                                args->setSize = SET_SIZE;
                                pthread_create(&tid[i], 0, mergeWorker, args);
                        }
                }

                i = 0;
                for(; i < MAX_THREADS/4; i++) {
                        pthread_join(tid[i], NULL);
                }

                i = 0;
                for(; i < MAX_THREADS/4; i++) {
                        if(i==0) {
                                MergeArgs * args = malloc(sizeof(* args));
```

```
                                args->leftStart = 0;
                                args->rightStart = SET_SIZE * 2;
                                args->setSize = SET_SIZE * 2;
                                pthread_create(&tid[i], 0, mergeWorker, args);
                        }
                        else {
                                MergeArgs * args = malloc(sizeof(* args));
                                args->leftStart = SET_SIZE * 4;
                                args->rightStart = SET_SIZE * 6;
                                args->setSize = SET_SIZE * 2;
                                pthread_create(&tid[i], 0, mergeWorker, args);
                        }

                }
                i = 0;
                for(; i < MAX_THREADS/4; i++) {
                        pthread_join(tid[i], NULL);
                }

                i = 0;
                MergeArgs * args = malloc(sizeof(* args));
                args->leftStart = 0;
                args->rightStart = SET_SIZE * 4;
                args->setSize = SET_SIZE * 4;
                pthread_create(&tid[0], 0, mergeWorker, args);
                pthread_join(tid[0], NULL);
                clock_gettime(CLOCK_REALTIME, &quickEnd);
                printTimespec(quickStart, quickEnd, "quicksorting");
                logResult(quickStart,quickEnd);
        }
        logOrder(v, MAX_ITEMS);
}
```

OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#define MAX_ITEMS 10000000
#define swap(v, a, b) {unsigned tmp; tmp=v[a]; v[a]=v[b]; v[b]=tmp;}
static int counter = 0;
static int *array;

static void print() {
    int i;

    for (i = 0; i < MAX_ITEMS; i++) {
        printf("%d\n", array[i]);
    }
    printf("\n");
}

static void init() {
    int i;

    array = (int *) malloc(MAX_ITEMS*sizeof(int));
    for (i = 0; i < MAX_ITEMS; i++) {
        array[i] = rand();
         }
}

static unsigned partition(int *array, unsigned low, unsigned high, unsigned pivot_index) {

    if (pivot_index != low)
        swap(array, low, pivot_index);

    pivot_index = low;
    low++;

    while (low <= high) {
        if (array[low] <= array[pivot_index])
            low++;
        else if (array[high] > array[pivot_index])
            high--;
        else
            swap(array, low, high);
    }

    if (high != pivot_index)
        swap(array, pivot_index, high);
    return high;
}

static void quick_sort(int *array, unsigned low, unsigned high) {
```

```
    unsigned pivot;

    if (low >= high)
        return;
    pivot = (low+high)/2;
    pivot = partition(array, low, high, pivot);

    #pragma omp parallel sections num_threads(8)
    {
         #pragma omp section
         {
             if (low < pivot) {
                quick_sort(array, low, pivot-1);
             }
         }
         #pragma omp section
         {
             if (pivot < high) {
                 quick_sort(array, pivot+1, high);
             }
        }
    }
}

int main(int argc, char **argv) {
    int i = 0;
    removeResultFile();
    for(; i < 10; i++) {
         init();
         struct timespec quickStart, quickEnd;
        clock_gettime(CLOCK_REALTIME, &quickStart);
         quick_sort(array, 0, MAX_ITEMS-1);
         clock_gettime(CLOCK_REALTIME, &quickEnd);
        printTimespec(quickStart, quickEnd, "quicksorting");
        logResult(quickStart,quickEnd);
         free(array);
    }
}
```

Mandelbrot set
Sequential

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define L 1000
#define HEIGHT L
#define WIDTH L
#define MAX 1500

static unsigned int* map;

void mandelbrot() {

        int i, j;
        double x_min = -1.6f;
        double x_max = 1.6f;
        double y_min = -1.6f;
        double y_max = 1.6f;

        for (i = 0; i < HEIGHT; i++) {
                for (j = 0; j < WIDTH; j++) {
                        double b = x_min + j * (x_max - x_min) / WIDTH;
                        double a = y_min + i * (y_max - y_min) / HEIGHT;

                        double sx = 0.0f;
                        double sy = 0.0f;
                        int iterations = 0;

                        while (sx + sy <= 64.0f) {
                                float xn = sx * sx - sy * sy + b;
                                float yn = 2 * sx * sy + a;
                                sx = xn;
                                sy = yn;
                                iterations++;
                                if (iterations == MAX)    {
                                        break;
                                }
                        }

                        if (iterations == MAX)    {
                                map[j + i * WIDTH] = 0;
                        }
```

```
                        else {
                                map[j + i * WIDTH] = 0xffffff;
                        }
                }
        }
}

int main() {
        int l;
        for(l = 0; l < 10; l++) {
        map = malloc(WIDTH * HEIGHT * sizeof(int));
        struct timespec brotStart, brotEnd;
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &brotStart);
        mandelbrot();
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &brotEnd);
        printTimespec(brotStart, brotEnd, "mandelbrot set");
        free(map);
        }
}
```

Pthreads

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <pthread.h>

#define L 2000
#define HEIGHT L
#define WIDTH L
#define MAX 1500
#define MAX_THREADS 8
#define T HEIGHT/MAX_THREADS

typedef struct {
        int x;
        int y;
        double h;
        double w;
} Args;

static unsigned int* map;

pthread_t tid[MAX_THREADS];

void mandelbrot(int x, int y, double w, double h) {

        int i, j;
        double width = x + w, height = y + h;
        double x_min = -1.6f;
        double x_max = 1.6f;
        double y_min = -1.6f;
        double y_max = 1.6f;

        for (i = y; i < height; i++) {
                for (j = x; j < width; j++) {
                        double b = x_min + j * (x_max - x_min) / WIDTH;
                        double a = y_min + i * (y_max - y_min) / HEIGHT;

                        double sx = 0.0f;
                        double sy = 0.0f;
                        int iterations = 0;

                        while (sx + sy <= 64.0f) {
                                float xn = sx * sx - sy * sy + b;
                                float yn = 2 * sx * sy + a;
                                sx = xn;
                                sy = yn;
                                iterations++;
                                if (iterations == MAX)    {
                                        break;
                                }
                        }

                        if (iterations == MAX)    {
                                map[(int)(j + i * WIDTH)] = 0;
                        }
                        else {
                                map[(int)(j + i * WIDTH)] = 0xffffff;
                        }
                }
        }
}

void * worker(void * args) {
        Args * arg = args;
```

```
        mandelbrot(arg->x, arg->y, arg->w, arg->h);
        free(arg);
}

Args * create_args(int x, int y, double w, double h) {
        Args * args = malloc(sizeof(Args));
        args->w = w;
        args->h = h;
        args->x = x;
        args->y = y;
        return args;
}

int main() {
        int l;
        for(l = 0; l < 10; l++) {
                map = malloc(WIDTH * HEIGHT * sizeof(int));
                struct timespec brotStart, brotEnd;
                clock_gettime(CLOCK_REALTIME, &brotStart);
                int i, counter = 0;
                double x = 0, y = 0;
                for(i = 0.0; i < MAX_THREADS*2; i++) {
                        pthread_create(&tid[counter++], NULL, worker, create_args(x,y, T, HEIGHT/2));
                        if(counter == MAX_THREADS) {
                                counter = 0;
                                x = 0;
                                y += HEIGHT/2;
                                int j;
                                for(j = 0; j < MAX_THREADS; j++) {
                                        pthread_join(tid[j], NULL);
                                }
                        }
                        else {
                                x += T;
                        }
                }
                clock_gettime(CLOCK_REALTIME, &brotEnd);
                printTimespec(brotStart, brotEnd, "mandelbrot set");
                free(map);
        }
}
```

OpenMP

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

#define L 2000
#define HEIGHT L
#define WIDTH L
#define MAX 1500

static unsigned int* map;

void mandelbrot() {

    int i, j;
    double x_min = -1.6f;
    double x_max = 1.6f;
    double y_min = -1.6f;
    double y_max = 1.6f;

    #pragma omp parallel shared(map,x_min, x_max, y_min, y_max) private(i,j) num_threads(8)
    {
        #pragma omp for schedule(static)
        for (i = 0; i < HEIGHT; i++) {
            for (j = 0; j < WIDTH; j++) {
                double b = x_min + j * (x_max - x_min) / WIDTH;
                double a = y_min + i * (y_max - y_min) / HEIGHT;

                double sx = 0.0f;
                double sy = 0.0f;
                int iterations = 0;

                while (sx + sy <= 64.0f) {
                    float xn = sx * sx - sy * sy + b;
                    float yn = 2 * sx * sy + a;
                    sx = xn;
                    sy = yn;
                    iterations++;
                    if (iterations == MAX)  {
                        break;
                    }
```

```
                }

                if (iterations == MAX)  {
                    map[j + i * WIDTH] = 0;
                }
                else {
                    map[j + i * WIDTH] = 0xffffff;
                }
            }
        }
    }
}

int main() {
        removeResultFile();
    int l;
    for(l = 0; l < 10; l++) {
        map = malloc(WIDTH * HEIGHT * sizeof(int));
        struct timespec brotStart, brotEnd;
        clock_gettime(CLOCK_REALTIME, &brotStart);
        mandelbrot();
        clock_gettime(CLOCK_REALTIME, &brotEnd);
        printTimespec(brotStart, brotEnd, "mandelbrot set");
        free(map);
    }
}
```