Investigating the Performance and Code Characteristics of Three Parallel Programming Models for C++

Waide Tristram[1] and Karen Bradshaw[2]
Department of Computer Science
Rhodes University, P. O. Box 94, Grahamstown 6140
E-Mail: g05t1067@campus.ru.ac.za[1]; k.bradshaw@ru.ac.za[2]

***Abstract*—We investigate and compare three parallel programming models for implementing parallel programs in C++ on multi-core computer systems. The models under consideration include the tried and tested OpenMP, Intel®'s Thread Building Blocks (TBB) and a do-it-yourself approach using Pthreads and Boost.Thread. For demonstration purposes, we create multiple parallel implementations of an algorithm suitable for parallelisation using the above models. The implementations are then compared on their performance characteristics and coding effort required. Additionally, the performance of the GNU C++(G++) and Intel® C++(ICC) compilers are compared. It is shown that OpenMP requires the least coding effort, while still providing good performance. Pthreads, Boost.Thread and TBB all require significant changes to the structure of the program. The Pthreads and Boost.Thread implementations are more complex than that of TBB, but provide more flexibility, coupled with increased risk. However, TBB promotes a better programming style, abstracts away thread management and has respectable performance. Performance measurements reveal that the SSE optimised Pthreads implementation compiled using ICC performs the best. However, without SSE, our OpenMP program using G++ outperforms the other implementations.**

***Index Terms*—Parallel programming, Parallelizing compilers, Shared memory systems**

I. Introduction

MULTIPROCESSOR computer systems are now widely available, with the majority of processors shipped by manufacturers such as Intel® and AMD being multi-core based designs [1]. The current trend towards more processor cores instead of higher clockspeeds can be attributed to the physical limitations of modern processor designs [2], [3]. This has serious implications for the design and development of applications as writing and debugging parallel software is difficult and requires more expertise than sequential programming [3]. However, it is vital that programmers become proficient at writing parallel code so that we can harness the parallel power available in our multi-core computer systems.

Various tools and libraries which help programmers make the transition from sequential to parallel programming are available. These include the compiler directive based OpenMP and Thread Building Blocks, which is a high-level library providing parallel templates and optimised parallel algorithms [4], [1], [2], [5]. Additionally, lower-level libraries such as Pthreads and Boost.Thread provide more control, but require the programmer to manage threads explicitly [6], [7].

The work reported in this paper was undertaken in the Telkom Centre of Excellence in Distributed Multimedia at Rhodes University, with financial support from Telkom, Comverse, Tellabs, Stortech, Amatole Telecom Services, Bright Ideas 39 and THRIP.

Other parallel and performance tools include Cilk/Cilk++, FastFlow, Intel® Integrated Performance Primitives (IPP) and Intel® Math Kernel Library (MKL).

This paper aims to evaluate the performance and code characteristics of the above parallel libraries for the parallelisation of an existing sequential Mandelbrot Set algorithm. The various parallel implementations are compared on performance factors such as program elapsed time, memory usage and speedup, using both the GNU C++(G++) and Intel® C++(ICC) compilers. Code characteristics such as effort required (measured in lines of code added or changed) and total number of lines will be compared.

The intention is to harness the experience gained through the use of these tools and techniques to improve the performance and throughput of transaction based services, such as the Asterisk Voice over IP based PBX system. This will involve improving and extending any existing parallelism as well as looking at other possible bottlenecks such as the encoding and decoding of audio.

II. Related Work

Utilising all the available CPU cores of a computer system requires that software be able to spread its workload across processors [3]. On shared memory multiprocessor systems, such as multi-core CPUs, this is typically achieved using multi-threading, although other techniques such as message passing can be employed [3]. There are limits to how far one can parallelise a program, which affects the performance and scaling of software on parallel computers [2], [8].

A. Amdahl's Law

The problem lies with Amdahl's law, defined in (1).

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

where S represents the speedup, P the proportion of the program that executes in parallel and N the number of processors [8]. This law states that both speedup and scaling are dictated by the sequential portion of the program. This means that given a P of 0.9 and N of 10, the speedup is only $\approx$ 5.26x, which is far from ideal.

It is therefore crucial to make every effort to reduce the sequential portion to increase speedup [8]. However, code that works well with four processors may not scale with eight or even sixteen processor cores [2].

Concurrent code can also have bottlenecks, such as inefficient locks and inappropriate lock granularity, which can cause a drastic decrease in performance under load [8], [9].

B. Parallel Programming Concepts

1) Tasks and Threads: A *thread* can be seen as a distinct execution path within a process, with its own *instruction pointer* and *stack*, and the ability to access shared memory between other threads within the process [5]. For the purposes of this paper, a *task* will be defined as a chunk of work that needs to be processed and is not explicitly associated with any particular thread until run-time.

2) Locks and Critical Sections: A *lock* or *mutex* is a special shared object that simplistically has two states: locked and unlocked. When a thread attempts to lock a particular mutex, the state of the mutex is checked. If the mutex is currently locked (the lock is held by another thread), the calling thread will typically block until the lock becomes available, otherwise it will immediately acquire the lock and continue execution. The unlock method simply releases the lock on the mutex, making it available to other threads [8].

A *critical section* is a section of code that can only be executed by a single thread at a time. This section is usually controlled by locking a mutex at the start and then unlocking the mutex at the end of section. This prevents other threads from entering the critical section as they will block while the mutex is locked. This forms the basis of mutual exclusion and synchronisation in parallel programming [8], [5].

3) Race Conditions: A *race condition* typically occurs when two or more threads attempt to update the same variable. The *race condition* becomes apparent when a thread overwrites the changes made by another thread, thus losing the original change and affecting program correctness [5].

4) Deadlock: Deadlock occurs when at least two threads block indefinitely because they are each waiting for locks to be released that are currently held by each other. This deadlocked state results in the program never running to completion unless the deadlock is forcibly broken [5].

C. Parallel Programming Models

Identifying the appropriate sections of code to optimise requires profiling the target program with a profiler tool such as OProfile or Intel® VTune™. Although optimising hotspots can be achieved by manually implementing multi-threading, the use of parallel abstractions such as OpenMP and Thread Building Blocks is recommended [2], [10].

1) Thread Building Blocks: Thread Building Blocks (TBB) is a C++ library that provides high-level, task-based multithreading features targeted at improving multi-core performance [5]. It makes use of common C++ templates and coding style to facilitate multi-threaded programming. The library also provides a set of highly-efficient parallel algorithm templates, such as *parallel_for* and *pipeline*, which further assist the programmer [11]. Since TBB is a run-time library, it can be used with any suitable C++ compiler [5].

TBB has a sophisticated thread management system that provides for a higher-level parallel programming abstraction as creation, termination and scheduling of threads is hidden from the programmer, allowing them to focus on the algorithm instead of thread management. The TBB scheduler assigns tasks to threads at run-time and can re-assign tasks through its "task stealing" mechanism to load-balance threads [5], [11].

2) OpenMP: The OpenMP API augments C/C++ and Fortran by extending the languages with parallel directives [4]. These directives can be added to specific loops or sections of code by the programmer to instruct the compiler to parallelise the specified code [4]. OpenMP is targeted at shared memory multiprocessors such as the multi-core CPUs found in modern computer systems [4], [12].

OpenMP focuses on data parallelism, however, with OpenMP 3.0 came improved support for nested parallelism and the introduction of the *omp task* directive. The *omp task* directive allows the programmer to allocate statements to tasks, which can be executed in parallel within an *omp parallel* section. Task barriers have also been introduced through the *omp taskwait* and *omp taskgroup* directives [12].

3) Do-It-Yourself Threading: The do-it-yourself approach is characterised by the low-level, explicit use of threads, mutexes and work scheduling algorithms. The Pthreads library enables this kind of low-level threading approach by providing methods and data-structures for the creation and termination of threads and mutexes. It also provides methods for locking and unlocking mutexes and provides conditional variables, which allow for signalling [6]. However, Pthreads is more suited to use in pure C programs as that was its intended purpose. While it is still possible use pure Pthreads in C++ programs, it does not conform with the C++ object-orientated programming style and care must be taken when implementing such a solution [6], [7].

The Boost C++ libraries have a wealth of functionality, including Boost.Thread library, which is a C++ threading library based on Pthreads. As with Pthreads, Boost.Thread enables low-level threading, with explicit creation and termination of threads and mutexes. However, Boost.Thread expands upon the capabilities of Pthreads by supporting multiple lock types, mutex types and lock functions, as well supporting barriers and more advanced conditional variables. Unlike Pthreads, Boost.Thread is designed for object-orientated programming and therefore promotes a better programming style [7].

D. Unsuitable Algorithms

While only the Mandelbrot program is discussed in this paper, parallelisation of the N-Body problem was also attempted. However, this proved futile as the nature of the N-Body algorithm is such that performance improvements cannot be easily achieved using parallelism without introducing race conditions and affecting the correctness of the program as confirmed by Reed [13]. This serves as a warning that not all algorithms can be improved with parallelism.

III. Research Design

The aims of this research require that various tools and optimisations be evaluated to gauge their effectiveness at improving software performance on multiprocessor systems. Accordingly, the approach taken focuses on experimentation and benchmarking of optimisations and tools. The Mandelbrot algorithm was chosen for this simple demonstation of the tools as it provides a best-case scenario for parallelisation. As the workload is easily distributed, any overheads introduced by the parallel tools should be made apparent when comparing the performance of the tools.

The first step of our approach involves identifying the sections of code in our sequential Mandelbrot program that

are consuming the most CPU time. This is achieved by executing the program and analysing its execution using an online code profiler. Finding the "hotspots" in our program provides a good starting point for the investigation into possible parallelisation approaches. Even though the chosen program has relatively obvious parallelisation targets, code profiling is still necessary to highlight any unforeseen CPU time usage [14].

After the parallelisation targets have been identified through the code profiling process, the program is modified or re-implemented to execute in parallel using each of the following parallel programming models: OpenMP, TBB, Pthreads and Boost.Thread. In some cases, more than one implementation for a specific programming model can be investigated. While use of the Intel® MKL and IPP libraries was considered, the functions provided were not appropriate for the algorithms being considered in this paper.

Once the various implementations have been developed and tested locally for compilation errors and glaring run-time issues, they are added to the automated benchmarking system described in Section VI-A. The results of the benchmarking process are then analysed and used to tweak the implementations where necessary.

Extensive performance testing of the implementations is then performed and the results are summarised along with an analysis of code characteristics in Section VI.

IV. Code Profiling

As per the approach described in Section III, the target program is compiled with debugging information and executed while being monitored by a software profiler. The profiling data is then examined and used to determine which sections of code are using the majority of the CPU cycles.

The OProfile software profiler was used to analyse the execution of the Mandelbrot program. OProfile is a system-wide profiler for Linux released under the GNU General Public License (GPL). It makes use of a kernel driver and a daemon to collect sample data from the hardware performance counters of the CPU, and supplies several tools for analysing the resulting sample data [15]. For the purposes of our profiling sessions, only the *CPU_CLK_UNHALTED* event was monitored, which provides the number of unhalted cycles used by the program during execution.

```
        : for (int y=0; y < dimension; ++y){
        :   C = complex<double>(-1.5, \
                ((2.0*double(y)) / dimension)-1.0);
 0.8011%:    for (int x=0; x < dimension; ++x){
 1.9536%:      C = complex<double>(((2.0*double(x) \
                  / dimension) - 1.5), C.imag());
 5.1812%:      for (int i=1; i < iter && norm; ++i){
77.8985%:        ...
        : } } }
13.5937%: /* from file: ".../include/g++-v4/complex" */
```

Listing 1. Mandelbrot sourcecode annotated with cycle count information.

In profiling the Mandelbrot program, a *dimension* of 8000, was used, which is then scaled accordingly by the algorithm. The OProfile tool, *opannotate*, was used to annotate the original source code with per-line cycle counts and total cycles per method. The basic structure of the Mandelbrot algorithm and associated profiling information (percentage of CPU cycles) can be seen in Listing 1.

From the profiling, it is evident that the majority of the CPU cycles are being expended executing the inner-most loop. However, this workload can be distributed by parallelising the outer loops instead of trying to distribute the work within the inner-most loop itself. It also appears that the GNU C++ complex numbers library is taking up a noticeable proportion of the CPU cycles. This could indicate an opportunity to decrease execution time by optimising the mathematical calculations.

V. Mandelbrot Parallelisation

After an examination of the code profiling information from Section IV, it is clear that the bulk of the work in the Mandelbrot program is carried out within the nested *for* loops. As such, the primary focus of the parallel optimisation effort is on making the loop iterations run in parallel. However, one has to be careful and ensure that it is safe to do so and that no race conditions are introduced in the parallel implementation of the algorithm.

It is evident from the structure of the *for* loops that the outermost *for* loop is the safest to run in parallel as there are no data dependencies between loop iterations, unlike the inner loops, which have several dependencies. This allows for a lock-free implementation, which is desirable as threads do not have to waste any time waiting for locks to be released and there is no chance of deadlock [8]. A large, shared array is used to store the results of each loop iteration, which may lead one to believe that it would need to be in a critical section. However, each loop iteration writes to a unique element in the array and no read operations are performed on the array until after all the worker threads have stopped. Therefore, we have deemed it safe to perform write operations on the array elements outside of a critical section.

The first parallel implementation was devised using OpenMP. Thereafter, TBB, Pthreads and Boost.Thread implementations were devised and tested along with OpenMP. In some cases, more than one implementation is demonstrated and discussed for a particular threading tool.

A. OpenMP

OpenMP makes use of simple pragma directives placed above the appropriate section of code. In this instance, the *omp parallel for* directive has been used above the outermost *for* statement as shown in Listing 2. The *schedule(static)* clause of the *omp parallel for* directive splits the loop iterations evenly between the threads for each processor [11].

```
void mandelbrot(...){
  ...
  #pragma omp parallel for default(shared) schedule(static)
  for (int y = 0; y < dimension; ++y){
    ...
    for (int x = 0; x < dimension; ++x){...}
    ...
  } /* end of omp parallel for */
} /* end of void mandelbrot(...) */
```

Listing 2. OpenMP parallel for with static scheduling.

However, the workload of the Mandelbrot algorithm is unbalanced, which can result in some processors sitting idle. Therefore, a second OpenMP implementation makes use of dynamic scheduling with a *chunk* size based on the input dimension divided by 64.

B. TBB

Unlike OpenMP, Thread Building Blocks requires more extensive modifications to the original code to implement parallelism. As stated in Section II-C1, TBB makes use

of C++ templates, which promotes an object-orientated approach [5]. As such, the Mandelbrot algorithm code has been moved into its own class and placed within an overloaded *()* operator.

The only change that needs to be made to the algorithm itself involves modifying the loop bounds for the outer *for* loop such that it uses *blocked_range.begin()* for the initial value and *blocked_range.end()* for the loop termination boundary [5]. The call to the Mandelbrot function is then replaced with a call to the TBB *parallel_for* template.

C. Do-it-yourself

One easy method for implementing parallelism using Pthreads or Boost.Thread involves spawning a thread for each available processor. The iteration space is then distributed evenly between threads, much like OpenMP's static scheduling as described in Section V-A.

As with the TBB implementation in Section V-B, the loop boundaries of the outer-most loop need to be modified. The Boost.Thread implementation requires the creation of a new class with an overloaded *()* operator, much like TBB [7]. The workload distribution occurs within the *main* method for both the Pthreads and Boost.Thread implementations.

As with the OpenMP and TBB implementations, a dynamic scheduling solution is also possible with Boost.Thread and Pthreads. The implementation is somewhat more complex than the static scheduling version above. Our implementation makes use of a very simple queue structure that stores tasks, which are created by dividing the iteration space into chunks of work. The dynamic scheduling versions of the Pthreads and Boost.Thread implementations, differ from their static counterparts in that the loop bounds are taken from the the thread's currently assigned task.

A thread-pool is then created, but instead of giving each thread and equal segment of the work, smaller segments of work are added to a task queue. Each thread then requests a new task once it has completed its current task until the task queue is empty.

D. SSE Optimisation

During the profiling process in Section IV, it was noted that it may be possible to improve the algorithm further by optimising the mathematical calculations. So, as an additional optimisation, the mathematical calculations of the Mandelbrot algorithm were completely rewritten to use Streaming SIMD Extensions(SSE) intrinsic functions instead of the complex numbers library. The SSE intrinsic functions allow access to additional registers and operations supported by most modern x86 CPUs. These registers and operations allow for more sophisticated and optimised solutions as multiple values can be packed into a single variable representing multiple registers and acted upon simultaneously by the SSE extensions on the CPU [14].

While the conversion to SSE has affected the readability of the algorithm, the SSE version is able to perform calculations for two inner-loop iterations simultaneously. This essentially introduces further parallelism into the algorithm as a single processor is able to perform multiple calculations at the same time [14]. G++ offers readability benefits over other compilers, such as ICC, in that it provides overloaded arithmetic operators for the intrinsic functions. However, this reduces the portability of the code.

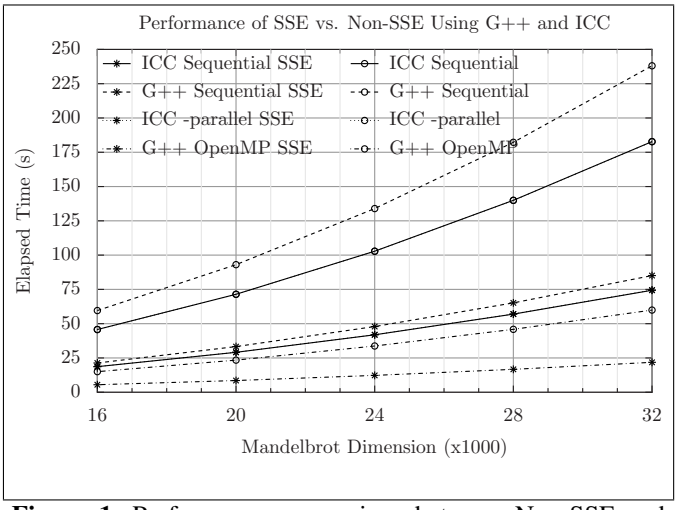

Figure 1: Performance comparison between Non-SSE and SSE optimised implementations.

VI. Results

To compare OpenMP, TBB, Pthreads and Boost.Thread, we developed multiple implementations of the Mandelbrot algorithm using each of the above libraries. We also developed an additional set of implementations making use of SSE intrinsics. We then benchmarked these implementations using the tool described in Section VI-A.

Testing was performed on a quad-core (Intel® Q9400, 2.66GHz) system with 4GB DDR2 800MHz RAM, running Gentoo Linux x64. Software versions are as follows: GCC 4.4.3, ICC 11.1.056, TBB 2.2 Update 1, Boost 1.42 and OpenMP 3.0. To ensure the consistency of the measurements, all non-essential services and programs were halted, including the X Window System.

A. Automated Benchmarking Tool

Owing to the repetitive testing and benchmarking of changes, as well as the number of different test cases, an automated benchmarking tool was used. This ensures that the benchmarking process is consistent and less time-consuming.

Instead of developing the benchmarking tool from scratch, an existing tool was modified. The Computer Language Benchmarks Game [16] makes use of a tool developed in Python by Isaac Gouy. The sourcecode for the benchmark tool is available under the 3-clause BSD license, thus allowing it to be downloaded and freely modified to meet additional requirements related to result logging.

The tool performs repeated measurements of program CPU time, elapsed time, resident memory usage and CPU load while the program is running, and summarises these measurements for each test program implementation [16]. To ensure that each program implementation executes correctly, the results of each run are compared and any inconsistencies are flagged causing the run to be excluded.

B. Mandelbrot Performance Analysis

Each implementation was run with multiple values for *dimension*, more specifically: 16000, 20000, 24000, 28000 and 32000. Each of these runs was repeated five times and the results averaged to produce the final measurements.

The first performance comparison of interest is the difference between the original algorithm and the SSE optimised version seen in Figure 1. It is quite interesting to note that

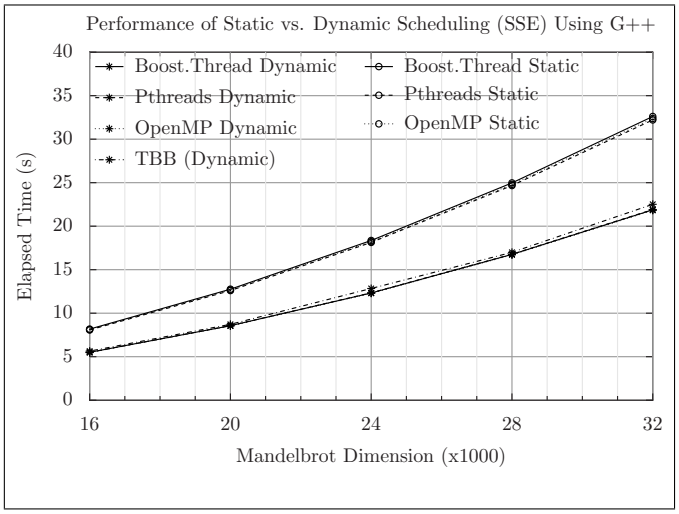

Figure 2: Performance comparison between static and dynamic scheduling.

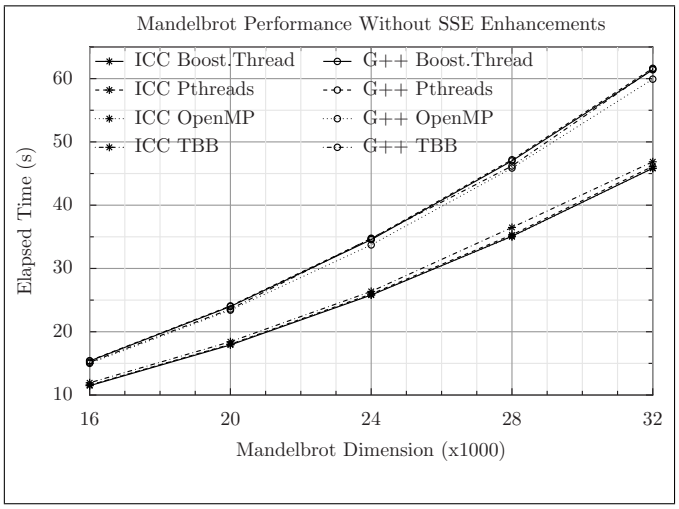

Figure 3: Elapsed time for non-SSE optimised parallel implementations.

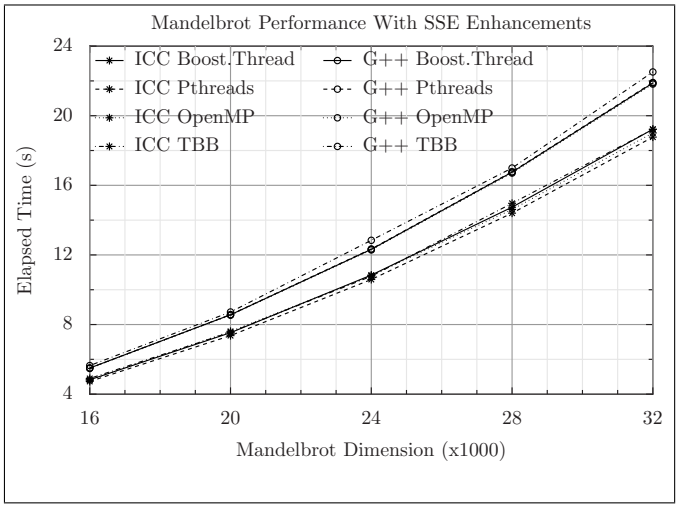

Figure 4: Elapsed time for SSE optimised parallel implementations.

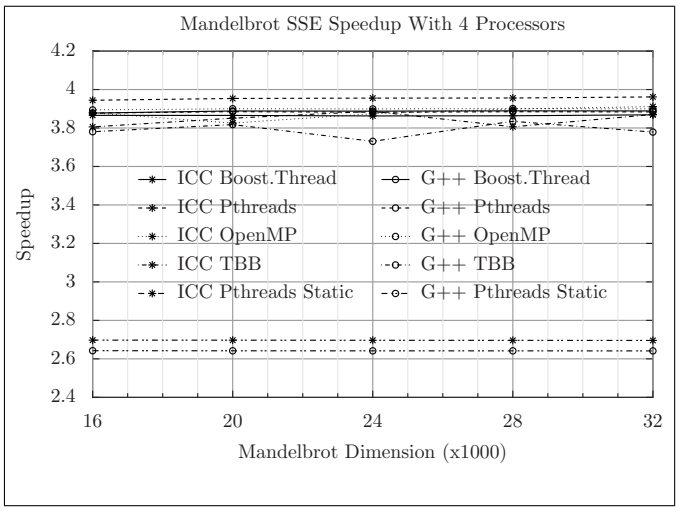

Figure 5: Speedup of SSE optimised parallel implementations.

the non-SSE OpenMP implementation is not that much faster than the sequential SSE versions. ICC features an auto-parallelisation flag, which is meant to analyse the supplied sourcecode and attempt to parallelise loops. However, in this case it has been unable to resolve the dependency issues and performs exactly the same as without the flag.

Another clear-cut comparison can be seen between the performance of static and dynamic scheduling. Our measurements are summarised in Figure 2, which show that dynamic scheduling easily outperforms implementations using static scheduling as alluded to in Section V-A. Unlike static scheduling, dynamic scheduling is able to keep the CPU busy at all times with uneven workloads.

From Figure 3, it is clear that OpenMP outperforms the other tools with non-SSE optimised implementations. Beyond this, we can see that Boost and Pthreads are basically equivalent on G++ and TBB fluctuates between OpenMP and Pthreads. All of the ICC implementations appear to have roughly equivalent performance, except for the TBB version, which performs the worst. Additional maths optimisation flags for the G++ and ICC compilers were also tested. However, these optimisations affect the accuracy of floating point operations, which lead to incorrect results. The performance was unaffected when using only the safer maths optimisations. Therefore, these results have been excluded.

Figure 4 shows that ICC notably outperforms G++ for SSE optimised implementations. The performance difference between parallel implementations seem to vary somewhat with the compiler being used. With ICC, Pthreads executes the fastest, followed by the rest, which perform very similarly. Whereas TBB is noticeably slower with G++.

The speedups of the various implementations, as seen in Figure 5, confirm previous observations that a SSE optimised, Pthreads implementation compiled with ICC exhibits the best performance, followed by OpenMP and Boost. All implementations, except TBB, exhibit a speedup of over 3.8x, with Pthreads achieving near-linear speedup of over 3.9x. The speedups of the static scheduling Pthreads versions are included and it is quite visible how poorly they scale when compared to dynamic scheduling.

While it may appear that TBB performs poorly when compared to OpenMP, Pthreads and Boost, it must be noted that our Mandelbrot algorithm is an ”embarrassingly parallel” data problem that is more suited to OpenMP’s data-parallel nature. Whereas TBB is better suited to task-parallel problems and object-orientated programs [5] and may prove easier to implement and perform better when faced with task-parallel problems. Due to the uneven workload, it ap-

pears that the use of mutexes in the dynamically scheduled Pthreads and Boost.Thread implementations has had little noticeable effect on performance as it would be rare for multiple threads to reach the critical section simultaneously.

Measurements show that memory usage for each implementation is essentially the same and as such, a detailed analysis of this performance aspect has been excluded.

C. Mandelbrot Code Analysis

For code characteristic comparisons, we record the number of lines of code for each implementation as well as the number of modified lines over the original sequential version.

Table I
CODE CHARACTERISTICS

Implementation	Lines	Modified
Sequential	77	-
SSE Sequential	93	38
Boost Static	119	48
Boost Dynamic	210	139
OpenMP	78	1
Pthreads Static	125	57
Pthreads Dynamic	202	134
Thread Building Blocks	101	34

It is immediately apparent that OpenMP requires the least effort to implement, only requiring a single additional line. This makes it well suited to the augmentation of existing software [11]. TBB requires significantly more code modification, but not nearly as much as Pthreads or Boost since the TBB library handles thread management. TBB also promotes a strong object-orientated C++ programming style and is thus well suited to the development of new projects [11].

While it is possible to develop simple Pthreads and Boost.Thread implementations without modifying existing code too much, the peformance is poor compared to the more complex implementations, which require significant modification to the code. These modifications need to be implemented carefully to avoid common parallel programming errors and therefore require more effort and expertise. It must also be noted that a custom task-queue was implemented for this experiment. However it should be possible to re-use this code in further threading endeavours, or make use of existing thread-safe queue structures. Regarding programming style, Boost.Thread promotes a similar object-orientated programming style to TBB, unlike Pthreads, which is a C library that is not recommended for use in C++ programs. The SSE implementation requires a number of code changes and decreases readability somewhat, but as seen in Section VI-B, the performance benefits are well worth the effort.

VII. CONCLUSIONS

OpenMP's trivially simple implementation and good performance make it a clear choice for implementing parallelism in many existing programs. However, it lacks the flexibility of other parallel libraries such as TBB and is best suited to tackling data-parallel problems. Although, with the addition of the *omp task* directive, OpenMP is in a better position to handle task-parallel problems. Meanwhile, TBB requires significantly more effort to implement, but is more flexible and is better equipped to handle other problems such as task-parallel processes, while still maintaining respectable performance when using *parallel_for* on data-parallel loops. It also fosters good programming style for new projects [11].

Pthreads and Boost.Thread require the most effort and expertise to implement effectively. However, they provide the programmer with significantly more control over how to implement and schedule threads, which can lead to very fast implementations given enough time and effort. They also expose the programmer to a number of threading pitfalls in that threads have to be managed very carefully to avoid deadlock and other parallel errors. Unlike OpenMP and TBB, which are vulnerable to a lesser extent as the libraries perform thread management on behalf of the programmer.

Regarding the run-time performance for the chosen algorithm, it appears that our SSE optimised, dynamically scheduled Pthreads implementation compiled using ICC produces the best results, followed by the dynamically scheduled OpenMP and Boost.Thread SSE implementations and finally TBB. However, when using G++ without SSE changes, our OpenMP implementation outperforms the other implementations followed by TBB.

REFERENCES

[1] R. Merritt. (2008, Apr. 3,) Chip industry confronts 'software gap' between multicore, programming. EETimes. [Online]. Available: http://www.eetimes.com/news/semi/showArticle.jhtml?articleID=207001633

[2] K. Carlson. (2008, Mar. 7,) SD West: Parallel or Bust. Dr. Dobb's Journal. [Online]. Available: http://www.ddj.com/hpc-high-performance-computing/206902441

[3] B. Hayes, "Computing in a Parallel Universe," *American Scientist*, vol. 95, pp. 476–480, 2007.

[4] M. Sato, "OpenMP: parallel programming api for shared memory multiprocessors and on-chip multiprocessors," in *ISSS '02: Proceedings of the 15th International Symposium on System Synthesis*. New York, NY, USA: ACM, 2002, pp. 109–111.

[5] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[6] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[7] B. Kempf. (2002, May 1,) The Boost.Threads Library. Dr. Dobb's Journal. [Online]. Available: http://www.drdobbs.com/cpp/184401518

[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[9] B. Goetz. (2001, Sept., 5) Threading lightly, Part 2: Reducing contention. [Online]. Available: http://www.ibm.com/developerworks/java/library/j-threads2.html?ca=drs-

[10] T. Mattson and M. Wrinn, "Parallel programming: can we PLEASE get it right this time?" in *DAC '08: Proceedings of the 45th annual conference on Design automation*. New York, NY, USA: ACM, 2008, pp. 7–11.

[11] P. Kegel, M. Schellmann, and S. Gorlatch, "Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores," in *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 654–665.

[12] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, 3rd ed., May 2008. [Online]. Available: http://www.openmp.org

[13] R. Reed. (2010, Feb. 23,) n-bodies: a parallel TBB solution: parallel code: first runs fatal flaw. Intel Corporation. [Online]. Available: http://software.intel.com/en-us/blogs/2010/02/23/n-bodies-a-parallel-tbb-solution-parallel-code-first-runs-fatal-flaw/

[14] A. Fog. (2009, May 3,) Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. Copenhagen University College of Engineering. [Online]. Available: http://www.agner.org/optimize/optimizing_cpp.pdf

[15] J. Levon. (2009, November) OProfile. OProfile Project. [Online]. Available: http://oprofile.sourceforge.net/about/

[16] I. Gouy. (2009, May) Computer Language Benchmarks Game. [Online]. Available: http://shootout.alioth.debian.org/help.php

Waide Tristram completed his Honours in Computer Science in 2008 under the guidance of Dr Karen Bradshaw in the Department of Computer Science at Rhodes University. Waide is now in his second year of MSc in Computer Science and working for the Centre of Excellence (CoE) at Rhodes University.