2018-06-09

Analyzing OpenMP Parallelization Capabilities and Finding Thread Handling Optimums

Simon Olofsson Emrik Olsson

DEGREE PROJECT
Computer Engineering
Bachelor level G2E, 15 hec

Department of Engineering Science, University West, Sweden

EXAMENSARBETE

Analyzing OpenMP Parallelization Capabilities and Finding Thread Handling Optimums

Sammanfattning

Utmaningar i modern processortillverkning begränsar klockfrekvensen för enkeltrådiga applikationer, vilket har resulterat i utvecklingen av flerkärniga processorer. Dessa processorer tillåter flertrådig exekvering och ökar därmed prestandan.

För att undersöka möjligheterna med parallell exekvering används en Fast Fourier Transform algoritm där trådprestanda mäts för olika skapade tester med varierande problemstorlekar. Dessa tester körs på tre testsystem och använder olika sökalgoritmer för att dynamiskt justera antalet trådar vid exekvering. Denna prestanda jämförs sedan med den högsta möjliga prestanda som kan fås genom Brute-Forcing. Testerna använder OpenMP-instruktioner för att specificera antalet trådar som finns tillgängliga för programexekvering.

För mindre problemstorlekar resulterar färre antal trådar i högre prestanda. Motsatsen gäller för större problemstorlekar, där många trådar föredras istället. Denna rapport visar att användning av alla tillgängliga trådar för ett system inte är optimalt i alla lägen då det finns en tydlig koppling mellan problemstorlek och det optimala antalet trådar för maximal prestanda. Detta gäller för alla tre testsystem som omfattas av rapporten. Metodiken som har använts för att skapa testerna har gjort det möjligt att dynamiskt kunna justera antalet trådar vid exekvering. Rapporten visar också att dynamisk justering av antalet trådar inte passar för alla typer av applikationer.

Datum:	2018-06-09
Författare:	Simon Olofsson, Emrik Olsson
Examinator:	Thomas Lundqvist
Handledare:	Andreas de Blanche
Program:	Nätverkstekniskt påbyggnadsår, 60 hp
Huvudområde:	Datateknik
Utbildningsnivå:	Grundnivå
Kurskod:	EXD500, 15 hp
Nyckelord:	*OpenMP, Parallell exekvering, Dynamisk trådhantering, Prestanda, Optimum*
Utgivare:	Högskolan Väst, Institutionen för ingenjörsvetenskap 461 86 Trollhättan Tel: 0520-22 30 00 Fax: 0520-22 32 99, www.hv.se

DEGREE PROJECT

Analyzing OpenMP Parallelization Capabilities and Finding Thread Handling Optimums

Summary

As physical limitations limit the clock frequencies available for a single thread, processor vendors increasingly build multi-core systems with support for dividing processes across multiple threads for increased overall processing power.

To examine parallelization capabilities, a fast fourier transform algorithm is used to benchmark parallel execution and compare brute-forced optimum with results from various search algorithms and scenarios across three different testbed systems. These algorithms use OpenMP instructions to directly specify number of threads available for program execution.

For smaller problem sizes the tests heavily favour fewer threads, whereas the larger problems favour the native 'maximum' thread count. Several algorithms were used to compare ways of searching for the optimum thread values at runtime.

We showed that running at maximum threads is not always the most optimum choice as there is a clear relationship between the problem size and the optimal thread-count in the experimental setup across all three machines. The methods used also made it possible to identify a way to dynamically adjust the thread-count during runtime of the benchmark, however it is not certain all applications would be suitable for this type of dynamic thread assignment.

Date:	June 09, 2018
Author:	Simon Olofsson, Emrik Olsson
Examiner:	Thomas Lundqvist
Advisor:	Andreas de Blanche
Programme:	Network Technology, Bachelor Year, 180 HE credits
Main field of study:	Computer Engineering
Education level:	First cycle
Course code:	EXD500, 15 HE credits
Keywords:	*OpenMP, Parallelization Capabilities, Dynamic, Thread Handling, Performance*
Publisher:	University West, Department of Engineering Science SE-461 86 Trollhättan, Sweden Phone: + 46 520 22 30 00, Fax: + 46 520 22 32 99, www.hv.se

Preface

The authors would like to thank Andreas de Blanche at University West for excellent mentoring, patience and great insights in this topic.

The different parts in this report have been written by both authors and have been evenly divided between them.

Analyzing OpenMP Parallelization Capabilities and Finding Thread Handling Optimums

Simon Olofsson, *Student, University West,* Emrik Olsson, *Student, University West,*

***Abstract*—As physical limitations limit the clock frequencies available for a single thread, processor vendors increasingly build multi-core systems with support for dividing processes across multiple threads for increased overall processing power. To examine parallelization capabilities, a fast fourier transform algorithm is used to benchmark parallel execution and compare brute-forced optimum with results from various search algorithms and scenarios across three different testbed systems. These algorithms use OpenMP instructions to directly specify number of threads available for program execution. For smaller problem sizes the tests heavily favour fewer threads, whereas the larger problems favour the native 'maximum' thread count. Several algorithms were used to compare ways of searching for the optimum thread values at runtime. We showed that running at maximum threads is not always the most optimum choice as there is a clear relationship between the problem size and the optimal thread-count in the experimental setup across all three machines. The methods used also made it possible to identify a way to dynamically adjust the thread-count during runtime of the benchmark, however it is not certain all applications would be suitable for this type of dynamic thread assignment.**

I. Introduction

STREAMLINING software performance is a continuous part of both hardware and software development. Challenges regarding power dissipation in modern processor chips have caused a trade-off between continued increase in clock frequencies or an increased core count [1]. Processor vendors like Intel and AMD also have the capability to handle two threads per core, called Simultaneous Multithreading [2]. When software is engineered in such way that it can be executed in parallel, performance is heavily increased on multi-core systems [1]. On shared memory computer systems, OpenMP [3] can be deployed to use parallelism. OpenMP is a set of compiler directives and runtime library routines.

In this bachelor thesis, *FFT_OPENMP* [4], a program utilizing OpenMP parallelism capabilities is used for testing on three shared memory computer systems. It computes fast fourier transforms [5] using different problem sizes. It is desirable to maximize performance for a computer system running this software, but it is uncertain that utilizing all available cores on a specific system will constantly result in better performance, especially regarding varying problem sizes. Instead, we believe that a dynamic thread count will be more suitable for those workloads.

The purpose of this bachelor thesis is to implement dynamic adjustment mechanisms in the *FFT_OPENMP* software in order to investigate whether thread handling optimums differ, and if so, find an optimal thread count at runtime. It focuses on maximizing the performance on three computer systems and the following problem description forms the basis for this thesis:

1) Does *FFT_OPENMP* thread optimum differ between different problem sizes and different amount of threads?
2) How can a dynamic adjustment of threads be achieved during program execution?

II. Background and related work

OpenMP is a set of compiler directives and runtime library routines [3]. It is designed to support implementation of parallel programs for shared memory architectures and employs a fork-join execution model [6]. An OpenMP program begins execution as a single process, called the master thread. The master thread executes sequentially until the first parallel construct is encountered. Then it creates a team of threads, including itself as part of that team. Upon completion of the parallel construct the threads in the team synchronize and only the master thread continues execution. There are numerous directives and constructs that can be used during the parallel execution, including work sharing constructs, synchronization and data sharing. Reduction operations, such as summation, can also be used [6].

There are several papers and articles investigating benefits, challenges and characteristics using the OpenMP framework. In [7], a benchmark using the OpenMP tasking features integrated in OpenMP version 3.0 is created and evaluated. The benchmark includes a set of applications exploiting regular and irregular parallelism, based on tasks instead of loops [7] and also presents other benchmarks that can be used to evaluate performance of using OpenMP. In [8], a OpenMP library is built to lower energy consumption for iteratively recurring imbalance patterns for Intel Haswell processors, being able to lower the energy consumption with up to 20% with minimal performance loss.

However, there are no published papers found that investigates whether it is possible to dynamically adjust thread count at runtime in order to get optimal performance in OpenMP enabled software. The possibility given today is to set thread count before executing the software. If no specific choice is made, native maximum thread count is chosen. This bachelor thesis aims to prove that thread handling optimum differs for the program *FFT_OPENMP* for different problem sizes. Then, an implementation and evaluation of different thread altering mechanisms is done, with the purpose of increasing execution performance.

III. Experimental setup

In this bachelor thesis, two test scenarios are constructed to accurately test if thread optimum differs between problem sizes and different thread counts. Three testbed computers are used for both scenarios to conduct performance tests. They are of different age, architectural design and consist of both Intel and AMD processors in order to get general and architecture-independent results. The computers are named T7500, Kraken and Ymca, with their hardware characteristics shown in Table I. All computers have CentOS Linux - v7 installed as the operating system.

TABLE I: Testbeds

T7500	24-core Intel Xeon (E5645 @ 2.4GHz) 2 threads / core, 6 cores / socket, 2 sockets
Kraken	64-core AMD Opteron (6284 SE @ 2.7GHz) 2 threads / core, 8 cores / socket, 4 sockets
Ymca	24-core Intel Xeon (E5-2650v4 @ 2.2GHz) 2 threads / core, 12 cores / socket, 1 socket

The base software used for testing is *FFT_OPENMP* [4], written by John Burkardt. It demonstrates an implementation of the Fast Fourier Transform of a complex data vector using OpenMP for parallel execution [4]. More specifically, the program performs matrix calculations. The matrix is quadratic and the size being calculated can be configured in accordance with a desired problem size. The sizes chosen for this thesis range from n=2 to n=4194304 as shown in Table II. These n values correspond to the number of rows and columns for the matrix.

Since the problem size for *FFT_OPENMP* can be configured, the program is considered suitable for performance testing. The total throughput recieived is measured in megaflops (million floating-point operations per second). Conducted performance tests in this thesis consist of the base software *FFT_OPENMP* and modified versions of it.

Tests are created for two scenarios. Scenario one aims to maximize execution performance in *FFT_OPENMP* for a stand-alone computer system. Scenario two also aims to maximize execution performance in *FFT_OPENMP*, but with two distinctions. Firstly, *FFT_OPENMP* is modified to include five areas of code being executed in parallel to demonstrate a real world situation having varying problem sizes. Secondly, this scenario aims to show that if thread count can be altered dynamically for a program including more than one area being executed in parallel, it would be feasable to implement the same thread altering mechanisms in any given software using OpenMP.

The tests presented for each scenario begins execution by running a configured number of inner iterations. These iterations correspond to the execution of matrix calculations. When these calculations are done the test has successfully executed one round of measurements, called an outer iteration, and outputs a throughput value. The test continues to do measurements by starting the next outer iteration. This is repeated until all outer iterations are done executing. The test is then finished and outputs an aggregated throughput value for the entire testrun. Thread count can be altered for each outer iteration. Table II shows problem size and iteration count for both scenarios.

To set the thread count that the program will use, the source code of *FFT_OPENMP* is modified by changing a pragma instruction before the function in the program that does the

TABLE II: Problem size and iteration counts

	Scenario One		Scenario Two	
	Inner iterations	Outer iterations	Inner iterations	Outer iterations
n=2	1000	100	100	48091924
n=128	1000	100	100	3635224
n=4096	1000	100	100	114775
n=32768	1000	100	100	98806
n=4194304	1000	100	100	133

actual matrix calculations. This way, the number of threads available for use is set to a specific amount like the example below:

```
int threads = 32;
# pragma omp parallel for num_threads(threads)
for ( j = 0; j < lj; j++ ) {
  code block
 }
```

This code divides the code block work into multiple threads, which are run simultaneously. In this example the code will never exceed 32 simultaneous running threads, as that is the limit set by the num_threads() argument.

A. Scenario One

Scenario one aims to maximize execution performance in *FFT_OPENMP* for a stand-alone computer system. The software contains one area that is executed in parallel. Except from benchmarking *FFT_OPENMP* itself, additional tests are built and based on the same program. These additional tests integrate different types of *HillClimber* [9] and *BinarySearch* [10] algorithms to dynamically find thread optimums for the specific problem sizes.

1) MaxThreads: Original software is executed with default settings resulting in execution using all available processor threads. The results given are used as a performance baseline for relative comparison with other variants of the software.

2) Optimum: A modified version of the software is executed running one problem size at a time for every thread count. It ranges from one thread to all available threads. This way, thread runtime optimum is found by brute-forcing every problem size and thus outputs the maximum throughput possible as a result.

3) HillClimberLow: Begins execution with one thread. It then increments thread count with one and compares current performance with the value given in the last outer iteration. As long as performance increases, so does the thread count.

4) HillClimberMeanLow: Begins execution with one thread. Thread count is then increased by one, like *HillClimber Low*, but with the distinction that performance of the first two outer iterations are saved and a mean value is generated. The mean value is then compared to the third iteration. As long as performance is greater than the mean value of the last two outer iterations, execution continues and future results are used for mean value calculation. This results in a built-in inertia, with the aim of passing local maximums. For a more detailed explanation with pseudocode, see Appendix B.

5) HillClimberHigh: Begins execution with all available threads. It then decrements thread count with one and compares current performance with the value given in the last outer iteration. As long as performance increases, thread count is decreased.

6) HillClimberMeanHigh: Begins execution with all available threads. Thread count is then decreased by one, like *HillClimber High*, but with the distinction that performance of the first two outer iterations are saved and a mean value is generated. The mean value is then compared to the third iteration. As long as performance is greater than the mean value of the last two outer iterations, execution continues and future results are used for mean value calculation. For a more detailed explanation with pseudocode, see Appendix B.

7) BinarySearch: Attempts to find the optimal thread count by using a span defined as the distance between the thread count in *min_threads* and *max_threads*. Each iteration measurement is taken at the min/max value of this span. From the min/max values, the change factor is compared to established thresholds and results in one of three actions taken:

- The min value is shifted halfway towards max.
- The max value is shifted halfway towards min.
- If neither treshold is triggered, elect the lowest thread value as the winner.

This effectively finds which half of the span holds the highest MFLOPS value and sets the new span to cover this half. This process is repeated until min and max have adjacent thread count, and the thread with the highest MFLOPS is selected as the winner. The remaining jobs are all run with this thread count. The threshold for the upward or downward movement was set to 1% for these runs. For a more detailed explanation with pseudo code, see Appendix A.

Performance testing in Scenario one begins with running the brute-forcing test *Optimum* to investigate how thread optimums differ on T7500, Kraken and Ymca. The results are shown as performance scaling results, presented in IV-A. After *Optimum* has finished, all other tests are executed one by one with their respective problem size and thread altering mechanism. The throughput performance recieved from these tests are presented in IV-A, sorted by testbed computer.

B. Scenario Two

Scenario two aims to maximize execution performance in *FFT_OPENMP* for a stand-alone computer system, but with two distinctions. Firstly, *FFT_OPENMP* is modified to include five areas of code being executed in parallel to demonstrate a real world situation having varying problem sizes within that software. Secondly, this scenario aims to show that if thread count can be altered dynamically for a program including more than one area being executed in parallel, it would be feasable to implement the same thread altering mechanisms in any given software using OpenMP.

The modified versions of *FFT_OPENMP* created and used in this scenario are only executed on the Ymca testbed. The performance baseline is based on the results from the test *OneThread*. The baseline can then easily be compared with the other conducted tests *TwelveThreads*, *AllThreads* and *BinarySearch*. The aim is to visualize performance differences linked to thread count and problem sizes by showing execution time spent in each parallel region and the aggregated execution time for all five regions. Results are shown in section IV-B.

1) OneThread: The baseline test is tuned so that every parallel region has a native execution time of one hour, using one thread for execution. This was done by measuring each execution of the problem calculation and running until each problem size had reached 60 minutes of processing time, totaling five hours. The number of iterations required to reach the goal time for each problem is saved for the remaining tests.

2) TwelveThreads: Uses the iteration count data acquired from the *OneThread* test as a goal, instead of time as with the previous test. For the entirety of the execution time, the number of threads used are forced to 12 and each problem calculation repeats until the target iteration count is reached. Again, the total time spent in these loops is aggregated.

3) AllThreads: Uses an identical iteration count and recording of time like the *TwelveThreads* test, but instead forces the threads used to what is considered maximum by the function call *omp_get_max_threads()* as defined by the OpenMP library.

4) BinarySearch: Uses an identical iteration count and recording of time like *TwelveThreads* and *AllThreads*. However, instead of forcing the thread count to a specific value - it uses a binary search algorithm described in Scenario One to dynamically choose the best performing thread count for each problem size. This value is then kept throughout the remaining execution time of the program until the iteration count has been reached.

IV. Results

This section presents results for the conducted tests related to Scenario one and Scenario two, which are explained in section III.

A. Scenario One

Results from the brute-forcing test *Maximum* show that thread optimums differ for varying problem sizes and thread counts. This applies to all three testbeds and the results can be viewed as performance scaling patterns, shown in figure 1, 2, 3, 4 and 5. When looking at the performance scaling results when n=2, results show a clear advantage of running those tests with one thread for all testbed computers. When n=128, results show that both T7500 and Kraken performs best with one thread, whereas Ymca gives better performance with two threads. For the middle-sized problem n=4096, results show that optimal thread count is increased for each testbed computer. The performance scaling is most noticeable on Ymca, which prefers seven threads for best performance. When n=32768, T7500 prefers eight threads, Ymca prefers 20 threads and Kraken prefers nine threads. For the largest problem size n=4194304, T7500 prefers 17 threads and Kraken 33 threads (thread 31-64 not visible in these figures). When looking at Ymca, performance scaling is barely noticable for this problem size. Ten threads is found as the optimal thread count.

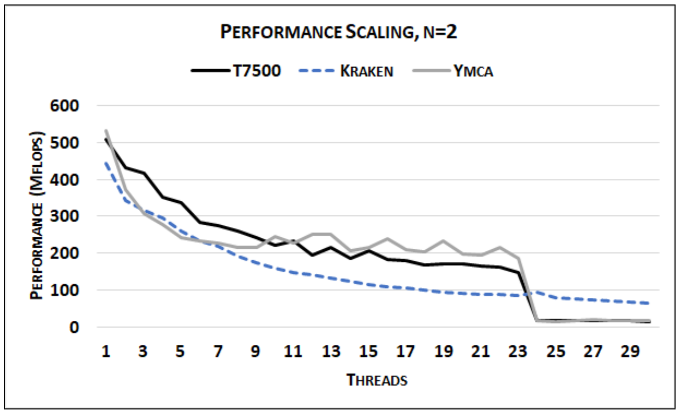

Fig. 1: Performance scaling results for all testbed computers when n=2. Higher values are better.

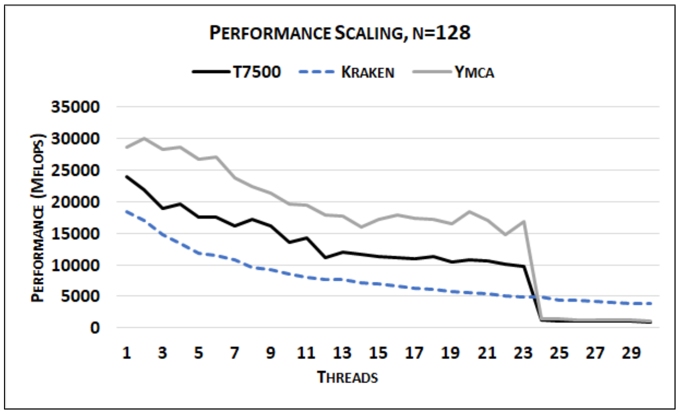

Fig. 2: Performance scaling results for all testbed computers when n=128. Higher values are better.

Results for the other conducted performance tests on T7500 shown in figure 6 indicate large performance differences between them. Looking at the results for smaller problem sizes, *HillClimberLow* and *BinarySearch* give a significant increase in performance. *HillClimberLow* gives the best result, by a performance factor of 13.66 over *MaxThreads*. For larger problem sizes on the T7500, results show that utilizing more threads gives better performance. Here, *BinarySearch* is best, with results on par with *Optimum*, giving roughly a 7% increase over *MaxThreads*.

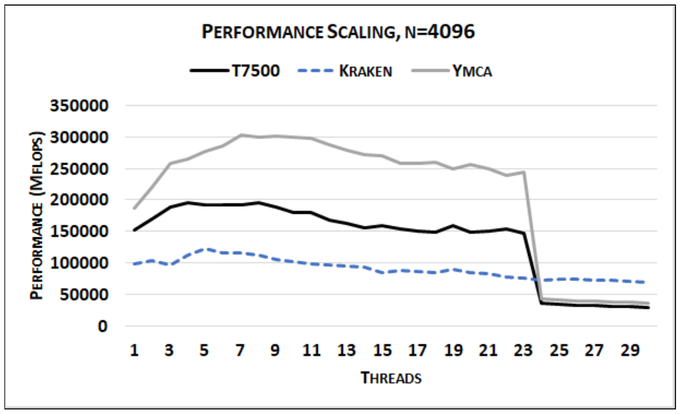

Fig. 3: Performance scaling results for all testbed computers when n=4096. Higher values are better.

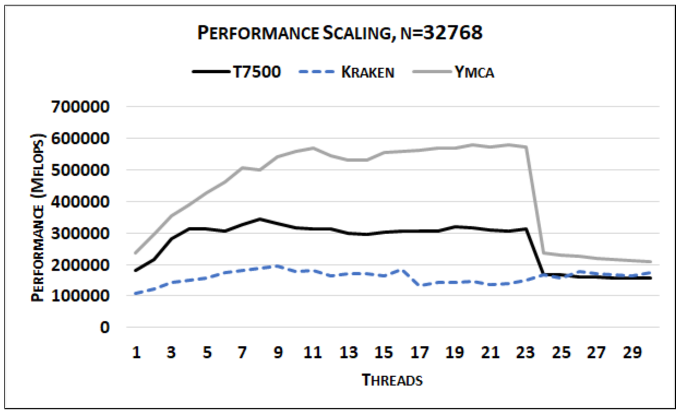

Fig. 4: Performance scaling results for all testbed computers when n=32768. Higher values are better.

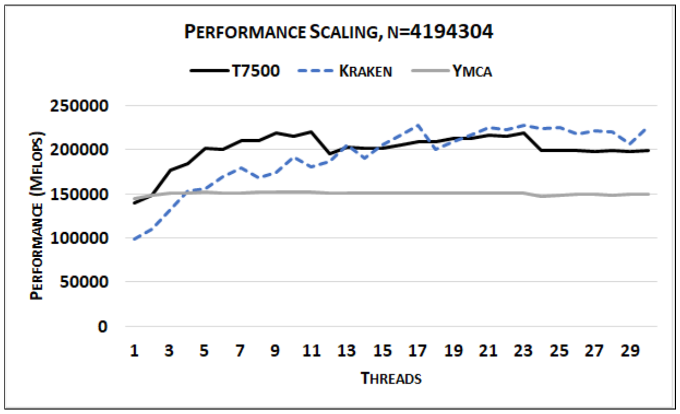

Fig. 5: Performance scaling results for all testbed computers when n=4194304. Higher values are better.

When tests are conducted on the second testbed computer, Kraken, a similar performance pattern is observed as the one found on the T7500. Results show a large spread, with *HillClimberLow* and *BinarySearch* giving a significant increase in performance for the smallest problem size, shown in figure 7.

But the results also reveal a noticeable difference which is visible when observing the relative performance gain in total. For Kraken, the difference between *MaxThreads* and the best

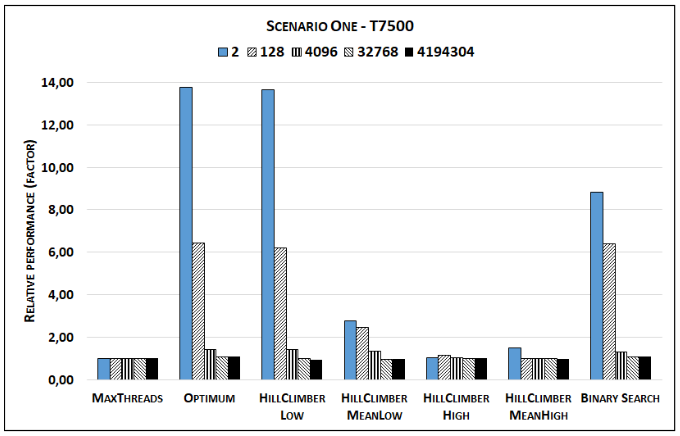

Fig. 6: Performance results for T7500 with altering problem sizes and thread count. Higher values are better.

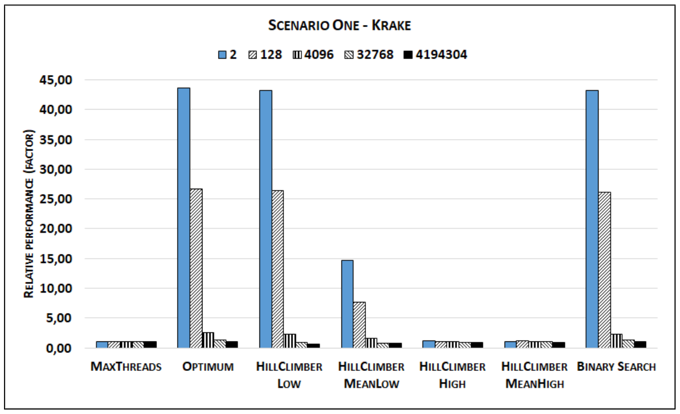

Fig. 7: Performance results for Kraken with altering problem sizes and thread count. Higher values are better.

performing test is by a factor of 43.28 when n=2. For the largest problem size the performance gain is more modest with a 5% increase, with *BinarySearch* edging ahead. Another difference viewing the results on Kraken is that *BinarySearch* performs noticeably better when n=2, being on par with both *HillClimberLow* and *Optimum*. Even *HillClimberMeanLow* performs noticeably better when n=2.

The results received from Ymca show similar performance patterns as for T7500 and Kraken, shown in figure 8. The performance gain between *MaxThreads* and *BinarySearch*, which performs best for the smallest problem sizes, is by a factor of 10.01 for n=2. This is lower compared to the other testbed computers, especially Kraken. Results also show a positive performance gain for *HillClimberMeanLow* when n=128. For larger problem sizes, *BinarySearch* performs best, being on par with *Optimum*.

B. Scenario Two

Throughout this subsection, the test *OneThread* is used as the performance baseline for the measurement of execution time. For the *AllThreads* test - running at 24 threads - the execution time for the n=2 problem size increased by a factor of 10.00. A similar trend was seen with n=128, where the

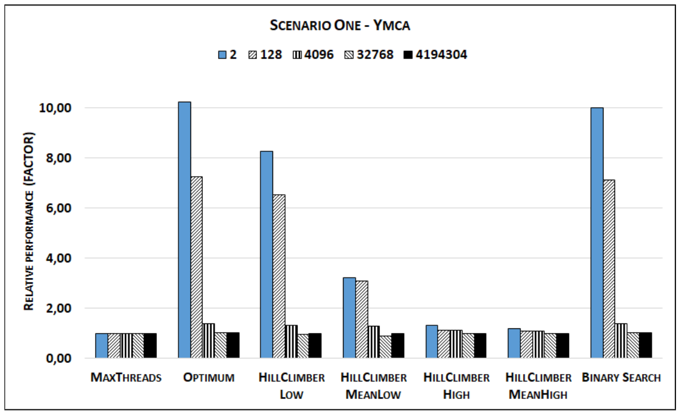

Fig. 8: Performance results for Ymca with altering problem sizes and thread count. Higher values are better.

execution time increased by a factor of 7.00. For n=4096, 32768 and 4194304 execution time was reduced by 32%, 75% and 77% respectively.

In the *TwelveThreads* test, the execution time for n=2 increased by a factor of 7.00. The result for n=128 also show an increase by a factor of 4.00. For n=4096, 32768 and 4194304 execution time is reduced by 45%, 76% and 77% respectively.

In the *BinarySearch* test, execution time for n=2 is reduced by 12%. For n=128, 4096, 32768 and 4194304 the execution time is reduced by 16%, 16%, 75% and 77% respectively.

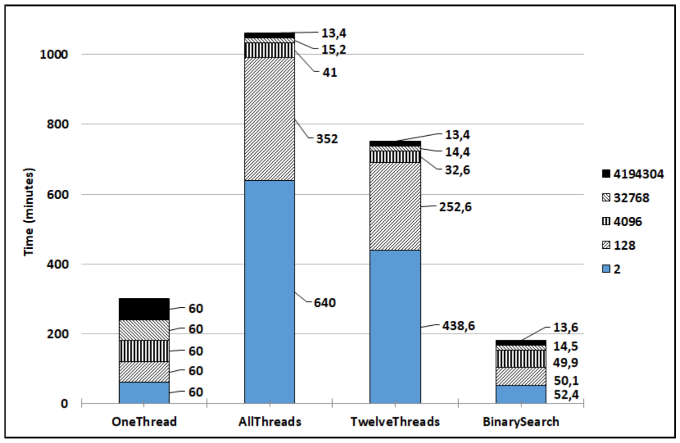

Fig. 9: Performance comparison in time. *OneThread* is the baseline. *AllThreads*, *TwelveThreads* and *BinarySearch* use same problem size.

Despite the reduced total execution time for problem sizes n=2 and n=128 for the binary search method this does not mean *BinarySearch* found a better thread value. Table III shows the thread optimums found. As the baseline and binary search both found the same thread value, the difference comes from performance variation during execution. These data points come from brute-forcing a search to find the optimum thread count as well as the results from the binary search algorithm across all problem sizes.

TABLE III: Thread optimums found

Problem sizes	2	128	4096	32768	4194304
Optimum (brute-forced)	1	1	8	23	11
BinarySearch	1	1	12	21	12

V. Discussion

The purpose of this bachelor thesis was to investigate if thread handling optimums differ, and if so, find an optimal thread count at runtime. When running the brute-forcing test *Maximum*, getting performance scaling results, they show that optimal thread count clearly differs between different problem sizes. In order to find those thread optimums dynamically at runtime, hill climber and binary search algorithms were an inspiration and an aid for the tests being created for that purpose. The reason for that choice is based on the fact that the algorithms are comparatively efficient in finding an optimum value and fairly easy to integrate.

However, the implementation of thread altering mechanisms became a challenge since the whole program needed to be understood and analyzed thoroughly to know which functions ran the matrix calculations and how they worked. Another challenge was to know where and how OpenMP function calls best would be placed to alter the thread count at runtime. For the searching algorithms, the challenge was to find a method that could find the optimum thread value in x steps where x < max threads as the brute-force test *Maximum* simply cycles through threads from one to max and will find the optimum value based on the returned MFLOPS. The hill climbing methods allowed for a limited search as any local maximum will prevent the search from finding any true maximum.

While a binary search allows us to reach a target value relatively quickly, there is also the chance to miss potential optimum values as the first step discards one half of the total available thread spectrum. We also opted not to continually re-check the optimum thread value which could potentially be another problem if available resources varied greatly during execution time.

When looking at the performance results from the other conducted tests, they show that performance can be largely increased, specifically regarding smaller problem sizes when few threads are used for execution. In those cases, performance increases by a factor of at least 10.01 and 6.00 respectively, depending on testbed computer. The performance gain is measured by looking at the difference between *MaxThreads* and the best performing test. For Kraken, having the oldest computer architecture, performance is increased by a factor of 43.28. When looking at the newest system Ymca, performance is increased by a factor of 10.01 which is significantly lower. This indicates that processor vendors have improved multi-core performance. Regarding the larger problem sizes, performance was not heavily increased.

VI. Conclusion and future work

We have shown that employing parallelizing techniques in a program with default settings, using all available threads for exectuion, is not always optimal in terms of performance. The relationship between problem size and thread usage shows better performance when using fewer threads for smaller problems and increasing the thread count as the calculation complexity increases. Attempts to dynamically adjust the thread count at runtime were successful, with the *Binary Search* algorithm being the winner and giving a good approximation of the optimal thread value. Attempts to dynamically adjust the thread count in five parallel areas were also successful. This states that the thread altering mechanisms used in this thesis is feasible to use in any given software using OpenMP.

For future work, a more robust search algorithm could be implemented allowing for monitoring and re-acquiring the optimum values during continuous execution. Examining parallelization options for production software is another possibility, with a focus on different types of operations and possible bottlenecks with the goal of finding individual thread optimums for each iteration loop.

Acknowledgment

The authors would like to thank Andreas de Blanche at University West for excellent mentoring, patience and great insights in this topic.

References

[1] B. Venu, (2011), *Multi-core processors - An overview*
[2] S.J. Eggers, J.S. Emer et. al, *Simultaneous multithreading: a platform for next-generation processors*, IEEE Micro, vol.17, no.5, pp.12-19, 1997.
[3] OpenMP, *What is OpenMP* http://www.openmp.org/about/openmp-faq/ [Accessed: June 1st, 2018]
[4] J. Burkhardt, *Fast Fourier Transform using OpenMP* https://github.com/futurecore/jburkardt-c/tree/master/fft_openmp [Accessed: June 1st, 2018]
[5] Wolfram Research Inc., *Fast Fourier Transform* http://mathworld.wolfram.com/FastFourierTransform.html [Accessed: June 1st, 2018]
[6] Jin, H & MA, Frumkin (2000), *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*
[7] A. Duran, X. Teruel, R. Ferrer et. al, *Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP*, 2009 International Conference on Parallel Processing, pp.124-131, 2009.
[8] B. Wang, D. Schmidl, M.S Müller, *Evaluating the Energy Consumption of OpenMP Applications on Haswell Processors*, 11th International Workshop on OpenMP, IWOMP 2015 Aachen, pp.233-246, 2015.
[9] GeeksForGeeks, *Introduction to Hill Climbing | Artificial Intelligence* https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/ [Accessed: June 1st, 2018]
[10] Revolvy, *Binary Search* https://www.revolvy.com/main/index.php?s=Binary+search [Accessed: June 1st, 2018]

APPENDIX A
BINARYSEARCH PSEUDOCODE

```
while target iterations not reached do
  run problem with min thread value
  run problem with max thread value
  get change factor from min to max
  if change factor < negative threshold then
    if thread difference == 1 then
      stop searching
      set optimum to min
    else
      adjust span downwards
    end if
  else if change factor > positive threshold then
    if thread difference == 1 then
      stop searching
      set optimum to max
    else
      adjust span upwards
    end if
  else
    lock thread count to min
    stop searching
  end if
  increment iteration counter
end while
```

APPENDIX B
HILLCLIMBERMEAN PSEUDOCODE

```
while target iterations not reached do
  run problem with test-specific thread count
  save current result
  if optimum == 0 then
    if target iterations == 0 then
      set value A to result
      increase/decrease thread count
    else if target iterations == 1 then
      set value B to result
      increase/decrease thread count
    else
      calculate mean value
      if result » mean value then
        set value A as value B
        set value B as result
        increase/decrease thread count
      else
        set optimum to 1
      end if
    end if
  end if
end while
```