UMEÅ UNIVERSITY

CONGESTION-CONTROLLED AUTOTUNING OF OPENMP PROGRAMS

Klas af Geijerstam Unger

Bachelor Thesis, 15 credits
Bachelor Of Science Programme in Computing Science

2019

Abstract

Parallelisation is becoming more and more important as the single core performance increase is stagnating while the amount of cores is increasing with every new generation of hardware. The traditional approach of manual parallelisation has an alternative in parallel frameworks, such as **OpenMP**, which can simplify the creation of parallel code. Optimising this code can, however, be cumbersome and difficult. Automating the optimisation or tuning of parallel code and computations is a very interesting alternative to manually optimising algorithms and programs. Previous work has shown that intricate systems can effectively autotune parallel programs with potentially the same effectiveness as human experts. This study suggests using an approach with the main algorithm used inspired from the congestion control algorithms from computer networks, namely **AIMD**. By applying the algorithm on top of an OpenMP program the parallel parameters such as grain size can be controlled. The simplified algorithm is shown to be able to achieve a 19% speedup compared to a naive static parallel implementation.

Acknowledgements

I want to thank my friends and family for making the last three years as amazing as they have been. I especially want to thank Jakub, Max and Desireé, you are simply the best.

Contents

1 Introduction

The need for parallelisation of performance critical applications is always increasing as the rate at which single-core performance is increasing is stagnating [13, 6]. Traditionally, parallelisation usually means that an algorithm or program is either created or modified to split the work onto threads or cores using a threading library such as **pthreads** [11][1] or using a message passing framework like **MPI** [7]. The traditional approach of manual parallelisation of programs and algorithms has an alternative in frameworks, such as **OpenMP**[3]. The frameworks abstract much of the complicated implementation details of parallelisation of loops and the creation of thread pools.

The potential speedup of parallelising a program is highly dependent on the nature and characteristics of the problem and how it is parallelised. Problems and computations with high dependencies of different stages of the workload are inherently difficult to parallelise, whilst problems with few dependencies that can be cleanly split into independent parts (often called embarrassingly parallel) can even achieve **superlinear** speedup. Superlinear speedup is when the achieved speedup exceeds the amount of parallel cores. Apart from the characteristics of the problem, there are many parameters that can impact the performance of the parallel implementation; such parameters include the **grain size** and what parts of the program that is parallelised. The grain size is the granularity of the parts a computation or problem is split into. As optimising these parameters are potentially time-consuming and difficult, the ability to allow for the automatic tuning of the algorithms is a very attractive approach.

Some problems have near-optimal static parameters and some problems have dynamic computation wide parameters. Previous work suggests that there are performance gains in autotuning of parallel applications, but the nature of problems that can benefit from autotuning and which algorithms that can provide autotuning data need further exploration [12, 9].

The main parameter that this study focuses on is the grain size, which corresponds to how granular a problem is split. The grain size is the size of the individual parts a parallel algorithm is split into. Optimising the parameters of a parallel application can be very time-consuming. The parallel frameworks often have built in algorithms that can be activated to automatically tune some parallel parameters, which often is the grain size. This allows for better usage of resources with little effort or requirement of knowledge from the programmer to tune the application, which could result in better running times. In the OpenMP framework, the traditional parallel for construct supports a scheduling option which allows the runtime to change the grain size dynamically but other parts of the framework lacks this ability.

The extra overhead introduced by autotuning differ greatly depending on the implementation. Previously suggested methods of autotuning range from compiler optimisation to machine learning, both which require preprocessing of the parallel programs [9, 10, 12].

To investigate whether or not **AIMD** has a valid use case as a parallel control algorithm a test implementation was created. A simple skeleton code is presented, with the aim of autotuning the grain size of a parallel program using a common computer network algorithm, **AIMD**. The presented approach for autotuning is shown to be able to achieve a 19% speedup.

[1]From here on, words or acronyms in bold will either have a full clarification following, be referenced, or be followed by a short description with a more complete description further down in the text.

2 Related work

Many approaches to autotune have been previously suggested and evaluated. Nogina et al.[12] suggested and attempted autotuning during runtime using an approach similar to that of memoization from the dynamic programming paradigm. By caching results of previous runs and grain sizes the algorithm could optimize and turn off the parallel sections of the tested problems. Autotuning as performed by the previous work can be classified into two different categories. The categories are the live category and the offline category.

The live category includes the work of Nogina et al. [12], which focus on the optimization of the parallel parameters during production runs of the algorithm. Category two are offline methods which change the parameters either by using dummy or test runs and or by static analysis of the program before live runs of the program. Previous work that fall under this category are the work by Collins et al. [9] who presented the **MaSiF** framework for autotuning parallel applications using machine learning. The MaSiF tool relies on training a model on autotuning different algorithms and problems, to later use the pre-trained tool to autotune the parameters of other programs and algorithms. Another offline approach is proposed by Cuenca et al. [10] where multiple compilers are used to autotune the algorithm as the different implementations optimize some code more optimally than others.

The proposed approach of autotuning of this study resembles the 'oracle' in the work by Nogina et al. [12], which is an extra piece of software that the parallel program consults on parallel parameters before entering a new parallel section. The oracle caches previous runs of similar parallel sections. The oracle starts with a low level of parallelisation and gradually increases the amount of parallelism for a section as long as it yields a performance increase. If the initial attempt at parallelisation fails, the oracle turns of all parallelisation for the current section.

This study focuses on the online category. More specifically as the **congestion control** algorithm proposed as the autotuning algorithm is designed to adapt during runtime.

3 Theoretical background

Parallel frameworks, such as the OpenMP framework, often have multiple ways of achieving parallelisation. Work can be split amongst threads using parallel loop constructs or by a task-oriented approach. A task is a separate piece of logic or computation that can be performed independently of other tasks and the rest of the program. The tasks are often executed by a **thread pool** of one or more threads. A thread pool consists of threads and a task queue; when a thread lacks work it picks a new task from the queue and executes it. When the queue is empty and thread lack work the threads are called starved. The tasks in the task queue are free to produce more tasks which results in subtasks. An OpenMP parallel section with task or taskloop constructs reflect a thread pool.

The OpenMP 4.0 framework introduced the concept of tasks. Where the parallel for loop splits the iterations of a loop onto different cores, the task construct creates independent tasks to be scheduled and executed by the runtime. OpenMP supports the creation of individual task of arbitrary size but also since version 4.5 a **taskloop** construct that splits the iterations of a loop into tasks. The taskloop construct is an OpenMP feature or pragma that generates OpenMP tasks from a for loop. The taskloop construct, however, lacks the support for dynamic control of the grain size by the runtime system and only supports user-controlled grain size. This means that the responsibility of optimisation is entirely left up to the developer.

As mentioned earlier, grain size is a very important concept in parallel computations. The grain size of a parallel computation is the granularity in how a problem is split. Computing the maximum value of a list using two parallel units by splitting the list in two would mean that the grain size is $N/2$ where N is the length of the list. Selecting a grain size is a balance between parallelism and overhead; a smaller grain size could result in higher levels of parallelism but at the cost of overhead, represented visually in Figure 1.

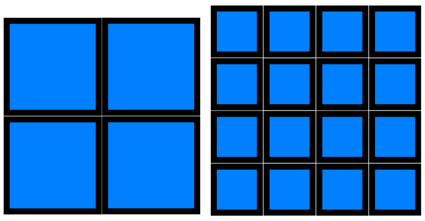

Figure 1: An illustration of how a lower grain size result in more overhead. The inner (blue) squares represent the useful work and the borders the overhead; the blue areas are equal of both the left and right figure.

When comparing frameworks for parallel computations the frameworks can be categorised into two different categories; the first is the shared memory and the other is the separate memory model. The OpenMP framework is a shared memory model, which is understandable since it is based off **pthreads** [11], which is a shared memory framework. This means that every parallel unit, i.e a thread or core has access to the memory of the other parallel units, this has both advantages and drawbacks. An advantage of the shared memory model is that communication is simple and fast, a core can simply read or write to the memory used by other cores. This comes with a drawback, as **race conditions** can create bugs or unwanted behaviour. A race condition is whenever the outcome of a statement depends on the order at which different operations are performed, and that order is not deterministic.

Congestion in computer networks represent, just as in traffic, the speed at which information or (in traffic terms vehicles) flow at. A highly congested network or street means a high load, and a slow throughput of information. Congestion control algorithms are used to adapt he speed at which information is sent to prevent overflowing buffers, thus limiting the amount of congestion in the network[5]. Congestion control comes into this study as a way of balancing grain-size and congestion, as the algorithms are created to work on a continuous stream of data, much like a long-running computation.

Additive increase and multiplicative decrease (AIMD)[5] is an algorithm used to dynamically change the amount of data than can be sent each time frame during a TCP connection. The algorithm tries to additively increase the congestion window as long as possible and as soon as congestion is detected the congestion window is multiplicatively decreased.

4 Method

There are two aspects of the method. The first aspect is to evaluate the performance of the suggested implementation using AIMD and if it has a valid use case as a control algorithm for grain size in parallel programs. The second aspect is the comparison of the implementation compared to the results of the previously tried control algorithms.

Two approaches of implementing the test program were considered, both with their respective strengths and weaknesses. The first approach was to modify the OpenMP framework and add an extra option to the taskloop construct that allowed OpenMP to autotune the grain size. The second approach was to create the autotuning code on top of the OpenMP **pragmas**. A pragma is a control structure or compiler directive added to the code to allow the OpenMP framework to parallelise the code [8]. The two approaches balance between ease of implementation and performance, the extension of the OpenMP framework would likely result in much less overhead than the latter approach; another benefit is the visuals of the parallel code, where code using pure OpenMP pragmas enjoy parallelisation without major changes to existing code.

In terms of speed, the most optimal approach would have been to modify the existing functions in the OpenMP framework, but this was deemed too complex and out of scope for this study. Therefore, the simpler but less efficient approach was chosen. The results will however contain a test with the autotuning algorithm disabled to allow for a comparison.

4.1 Choice of Method

The suggested method is in theory very simple. A cache of timestamps is maintained by the threads of the parallel section where the exit times of the tasks are saved in the cached with the thread id as key. After a set interval or phase in the computation the master thread compares the maximum difference in exit times of the threads in the cache. If the maximum difference is less than a given threshold the grain size is decreased additively, else the grain size is increased multiplicatively. Note that this is the inverse of the of the AIMD algorithm, which is additive increse and multiplicative decrease. The reason why this was changed was that the algorithm could achieve a higer level of parallelism quicker.

The test system had to conform to a couple of criteria that were recognized as important. The two criteria that were created to ensure that the test system was:

1. Easy to implement, as in little code required to implement.
2. Minimized overhead, with only a limited amount of operations required to autotune the **taskloop**.

The selected approach conforms to the above criteria well, especially the second criteria as it is very lean on operations. A pseudocode representation of the algorithm can be seen in Figure 1 and has two important sections. Before entering a parallel section the cache of timestamps is cleared. During task execution the threads store the time at which they stop executing a task in the timestamp cache. Thus, when all tasks are finished the difference between the cached time and the current time is the time a thread was starved and the autotuning algorithm can

```
# Begin parallel section, initialize cache to 0
time_cache = [0 for _ in range(THREADS)]
congestion_window = THREADS
# Executed by master thread
if MASTER:
    while work:

        # Creates tasks to be executed by threadpool
        create_n_tasks(congestion_window)

        task_wait()

        total_starve = 0
        for t in time_cache:
            total_starve += time() - t

        if t >= INCREASE_THRESHOLD:
            congestion_window *= INCREASE_FACTOR
        else:
            congestion_window -= DECREASE_TERM
```

Listing 1: Listing shows a pseudocode version of the control algorithm. The work in the tasks is omitted for brevity.

act accordingly.

The **taskloop** construct of the OpenMP framework will be the base of the optimisation as this allows for a simple task-oriented approach and importantly lacks grain size control in the framework. The method was chosen as it required only a small amount of code to create the test environment and that it allowed for testing with many kinds of problems.

As a test problem to evaluate the method a variation of computing the Mandelbrot set was selected. Computing the Mandelbrot set is an embarassingly parallel and has a simple implementation, which makes is a good candidate for testing the control algorithm.

The setup used by Collins et.al. [9] where multiple runs must be performed and a system for machine learning must be created, is a much more complex system than the approach suggested by this study. Along with the previous prerequisites the 'MaSiF' tool presented by Collins et.al. [9] relies on computing eigenvectors and also requires the collection of parameters from similar problems. This might improve the performance of the autotuning, but it requires a much more complicated setup than the time-stamp based solution proposed as well as much extra pre-deployment overhead.

As there are only three parameters that can be changed and the effectiveness of the autotuning should be predictable, i.e., when a certain increase in size proves to be less optimal a further increase would prove the same. The three parameters are the decrease term, the increase factor and the threshold for when to perform an increase or a decrease.

Phasing or synchronization must be artificially enforced into the algorithms that lack a natural dependency between stages, as it is required for the suggested setup to be able to control

Table 1 Table shows the first test sequence where the increase factor is adjusted.

Increase factor	2	3	9	18	100
Decrease term	-1	-1	-1	-1	-1
Threshold	0.01	0.01	0.01	0.01	0.01
Runtime (Seconds)	18.66	24.18	24.2	18.1	18.12

Table 2 Table shows the second test sequence where the threshold is adjusted.

Increase factor	2	2	2	2	2
Decrease term	-1	-1	-1	-1	-1
Threshold	0.09	0.01	0.001	0.0001	0.00001
Runtime (Seconds)	22.91	18.66	18.05	18.04	18.06

the grain size properly. The for the algorithm unneeded synchronization would most likely induce much overhead compared to an unsynchronized version. Because of this expected overhead three categories of runs are performed. One were any synchronization and autotuning is turned off. Two additional types with added synchronization where one version has the autotuning enabled and another with the autotuning disabled. The focus on the analysis is the difference between the synchronized versions but the runs without synchronization and autotuning is kept for reference comparisons.

4.2 Data Collection

The main metric that will be examined is the running time of the program. There are three main parameters that can be controlled in the proposed setup. The threshold of whether to increase or increase the grain size, and the two parameters controlling the speed of increase and decrease of grain size. Execution time for both the autotuning and the total running time is to be measured by the built in **wtime** function in OpenMP [3]. To improve the accuracy of the measured values all recorded times are to be averaged over multiple (10) runs.

5 Results

All test were run over a modified version of a computation of the mandelbrot set. An artificial synchronization point was added between the rows of the matrix. Tests were first performed to determine whether or not the values of the fixed parameters had an impact on the performance of the algorithm. The tests have linearly increasing problem intensity.

The tests focused on adjusting one parameter at a time. When no further increase in performance was deemed possible the direction of change in the parameter value was changed or the test stopped. One test with the autotuning algorithm turned off was performed, which

Table 3 Table shows the first test sequence where the decrease term is adjusted.

Increase factor	2	2	2	2	2
Decrease term	-1	-2	-3	-9	-18
Threshold	0.01	0.01	0.01	0.01	0.01
Runtime (Seconds)	18.66	17.41	17.36	17.38	17.36

Table 4 Table shows the second test sequence with a static task count, relative to the amount of available threads on the test hardware.

Task count ($T = 4$)	T	$2T$	$3T$	$9T$	$18T$
Runtime (Seconds)	21.54	12.87	11.28	8.43	7.58

can be seen in Table 4. A final test using the most optimal parameters from the three first tests where also performed, with the resulting runtime of 17.61 seconds. The results was poorer than the best result from previous tests, which where 17.36. This was unexpected as the expectation was that individually optimising the parameters would result in an optimal combined value, even if a global optimisation not can be guaranteed.

The results show that the values of the parameters have a noticeable impact on the performance. The largest difference in runtime in the autotuning algorithms favour is a 19.4% speedup; this is of course the best possible result from the test runs, where the best result of the non-autotuned version is a 66.9% speedup. The first test, seen in Table 1 showed a surprising result as the runtime decreased after two increments without a better yield, breaking the expected trend of the previous parameters compared to the runtime. The expected behaviour was noted in test two and three, where if a change already introduced worse results, a further similar change would not improve the runtime. The results of tests two and three can be seen in Table 2 and Table 3 respectively. The performance of the combined best parameters is still good compared to the runtime of most of the test runs, only over performed by some of the runs in the third test run.

6 Discussion

The initial expectation was that the proposed autotuning algorithm would perform better than an unguided solution but worse than that of the related work since the solution was less complex and considered fewer metrics. The results showed that the tested autotuning approach could yield increase performances compared to a non-autotuned version, but with only minor adjustment of the parameters the non-autotuned code performed much better.

The test runs indicate that it is better to saturate the OpenMP task queue as the samples with autotuning turned off far outperformed the samples with autotuning under almost all the tested parameters. This could be explained by the fact that the OpenMP framework limits the maximum number of tasks in the task queue at any given time, and exceeding this value prevents any new tasks from being spawned until the queue is freed up [4]. The speedup achieved by the autotuning version of the program compared to the initial tests with the autotuning turned off can be explained by the autotuning version allowing for a higher amount of parallelism. This is supported by the fact that the non-autotuned version surpassed the autotuned version in performance as soon as the amount of tasks generated was increased.

One strength of the method is its simplicity, which also is a weakness. The AIMD algorithm is applied in a very rough manner, with even small values for the parameters causing a drastic change in the way the program is parallelised. A potentially better approach could be to design the control structure in a way that finer adjustments are possible. This could, and most likely would, result in better performance of the algorithm as compared to the approaches of the related work, as their control structures allow for a much more granular tuning of the algorithm. A benefit of the presented approach is its memory complexity, even if memory often is a commodity the amount of data that needs to be stored is only a few bytes per thread, compared to the more complex approaches where more parameters or even data from previous runs must be cached.

The scope of the study did not leave room for data collection and analysis using tools such as **extrae** [1] and **paraver** [2] which can provide a visual representation and insight into a multitude of metrics such as task execution and thread efficiency. Given more time, expanding the results with such measurements would both be interesting and most likely give more insight into the specifics of the performance difference between the different parameter values.

One unreliable component of the test system is the OpenMP **wtime** function as this is not guaranteed to be consistent across threads. This could introduce errors into the autotuning algorithm if the inconsistency across the threads make the total time difference exceed the set threshold. This was deemed a necessary evil as creating a manually synchronized function for the retrieval of timestamps would introduce much overhead as this function sees heavy usage in the autotuning algorithm.

6.1 Problem characteristics

The characteristics of the problem is very important on the benefits of this type of autotuning. Other approaches can, for example, disable parallelisation if they detect that parallelisation does not yield better performance, however the method in this study can not. Another important characteristic of the problem for the autotuning to be useful is that the optimal parameters of the program must be dynamic over the computation. If the optimal parameters are static, there is no need for online autotuning and the parameters can be predetermined. The test problem is dynamic in the way that the operation complexity is increased linearly. Given a more dynamic problem it is possible that the suggested autotuning could perform better, which is partly supported by some of the related work [12].

7 Future work

There are many parts of the study that are highly interesting to examine further, especially autotuning in general. Future work of this study could be to implement it as part of a parallel framework and to investigate possible modifications of the algorithm to improve the performance of it. Another extension is to perform a study using the **extrae** and **paraver** tools mentioned in the discussion as this could give a better understanding of how the tasks are both executed and distributed over the threads.

Different kinds of problems are going to benefit from different types of autotuning, and it should be possible to create criteria to classify a problem to decide what kind of autotuning to perform. Autotuning is, as the related work shown, not limited to online dynamic autotuning, but some algorithms can be successfully be tuned using static analysis based on previous data. Combining the different kinds of autotuning into one framework could prove to be very useful, as the framework could do everything from compiler optimisation to online dynamic autotuning if deemed possible. To be able to create such a system the criteria to use for the classification, especially if the autotuning is to be performed automatically, very clearly defined. Defining these criteria would not only allow for the creation of the aforementioned framework, but also give a deeper insight into when and where autotuning is useful.

References

[1] BSC extrae. https://tools.bsc.es/extrae. [Online; accessed 6-mar-2019].

[2] BSC paraver. https://tools.bsc.es/paraver. [Online; accessed 6-mar-2019].

[3] Openmp 4.5. https://www.openmp.org/resources/refguides/. [Online; accessed 6-mar-2019].

[4] Openmp 4.5, task queue size. https://github.com/llvm-mirror/openmp/blob/aebe831622403337fabe44206f5d497e066c761e/runtime/src/kmp.h#L2397. [Online; accessed 6-mar-2019].

[5] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock. Host-to-host congestion control for TCP. *IEEE Communications surveys & tutorials*, 12(3):304–342, 2010.

[6] K. Asanović, R. Bodik, B. Catanzaro Christopher, J. Gebis Joseph, P. Husbands, K. Keutzer, D A. Patterson, W. Plishker Lester, J. Shalf, S. Williams Webb, and K A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[7] B. Barker. Message passing interface (MPI). In *Workshop: High Performance Computing on Stampede*, volume 262, 2015.

[8] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. M. Kaufmann, 2001.

[9] A. Collins, C. Fensch, H. Leather, and M. Cole. MaSiF: Machine learning guided autotuning of parallel skeletons. In *20th Annual International Conference on High Performance Computing*, pages 186–195, Dec 2013.

[10] J. Cuenca, L P. García, and G. Domingo. A proposal for autotuning linear algebra routines on multicore platforms. *Procedia Computer Science*, 1(1):515 – 523, 2010. ICCS 2010.

[11] B. Nichols, D. Buttlar, J. Farrell, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[12] S. Nogina, K. Unterweger, and T. Weinzierl. Autotuning of adaptive mesh refinement pde solvers on shared memory architectures. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, editors, *Parallel Processing and Applied Mathematics*, pages 671–680, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

UMEÅ UNIVERSITY