Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Marko Rasa

Instrumentation of OpenMP task scheduling

Master's Thesis
Espoo, April 27, 2015

Supervisor:	Assoc. Prof. Keijo Heljanko
Instructor:	D.Sc.(Tech.) Vesa Hirvisalo

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author:	Marko Rasa		
Title: Instrumentation of OpenMP task scheduling			
Date:	April 27, 2015	**Pages:**	58
Major:	Software Technology	**Code:**	T-110
Supervisor:	Assoc. Prof. Keijo Heljanko		
Instructor:	D.Sc.(Tech.) Vesa Hirvisalo		

Parallel computing models, such as tasking, are increasingly important as the modern processors are scaled mostly by adding processor cores instead of increasing clock speed. Scheduling the tasks requires computational resources and can be implemented with either software or hardware.

The goal of the thesis has been to develop a measurement system that is able to yield the basic tasking performance metrics of parallel program execution. The main challenge in developing such system is the overhead caused by the measuring, especially when tasks are extremely fine grained. An additional aim for the thesis has been understanding the scheduling properties of OpenMP task parallelism model.

We selected version 4.9 of the GCC compiler and its runtime (GOMP) as the concrete target of our study. The parts of the OpenMP 4.0 specification that are essential for the study are implemented by the compiler. The task scheduling operates by assigning tasks to worker threads. In doing so, it handles dependence structures and task queues, which require synchronisation to ensure correct operation. The studied GOMP implementation uses locking for that purpose.

We designed and implemented a measuring system using software performance counters by instrumenting the runtime system (GOMP) and hardware counters by calling PAPI (Performance Application Programming Interface) functions inside the runtime system. We tested the measurement system by implementing a task parallel version of AES (Advanced Encryption Standard), which yields small granularity tasks. In our tests we observed parallel scaling up to four threads, at which point lock congestion rose rapidly.

To understand the performance of our measurement systems, we compared the performance of instrumented and non-instrumented versions of the runtime system. The developed measurement system yields low overhead.

Keywords:	Parallel, Task parallelism, Runtime system, Measuring, OpenMP
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Marko Rasa		
Työn nimi: OpenMP tehtävävuoronnuksen instrumentointi			
Päiväys:	27. huhtikuuta 2015	**Sivumäärä:**	58
Pääaine:	Ohjelmistotekniikka	**Koodi:**	T-110
Valvoja:	Professori Keijo Heljanko		
Ohjaaja:	TkT Vesa Hirvisalo		

Rinnakkaislaskentamallit, kuten tehtävärinnakkaisuus, ovat yhä tärkeämpiä modernien suorittimien skaalautuessa enimmäkseen suoritinytimien lisäämisellä kellotaajuuden kasvattamisen sijaan. Tehtävien vuoronnus vaatii laskentaresursseja ja voidaan toteuttaa joko ohjelmistolla tai laitteistolla.

Tämän työn tavoite on ollut kehittää mittausjärjestelmä, joka antaa perusmittareita tehtävärinnakkaisuuden tehokkuudesta. Päähaaste järjestelmän kehittämisessä on mittauksen aiheuttama kuormitus, erityisesti kun tehtävät ovat erittäin hienojakoisia. Lisätavoite työlle on ollut ymmärtää vuoronnuksen ominaisuuksia OpenMP:n tehtävärinnakkaismallissa.

Valitsimme version 4.9 GCC kääntäjästä ja sen ajonaikaisen järjestelmän (GOMP) konkreettiseksi kohteeksi tutkimukselle. Kääntäjä toteuttaa työlle oleelliset osat OpenMP 4.0 määritelmästä. Tehtävien vuoronnus toimii jakamalla tehtäviä työsäikeille. Tämän mahdollistamiseksi sen täytyy käsitellä riippuvuusrakenteita ja tehtäväjonoja, jotka vaativat koordinointia. Tutkittu GOMP toteutus käyttää lukkoja tähän tarkoitukseen.

Suunnittelimme ja toteutimme mittausjärjestelmän käyttämällä ohjelmistotehokkuuslaskureita instrumentoimalla ajonaikaisen järjestelmän (GOMP) ja laitteistotehokkuuslaskureita kutsumalla PAPI (Performance Application Programming Interface) funktioita ajonaikaisen järjestelmän sisällä. Testasimme mittausjärjestelmän toteutteuttamalla tehtävärinnakkaisen version AES:stä (Advanced Encryption Standard), joka tuottaa pienijakoisia tehtäviä. Testeissä havaitsimme rinnakkaista skaalautumista neljään säikeeseen asti, jonka jälkeen lukon ruuhkautuminen kasvoi nopeasti.

Mittausjärjestelmämme suorituskyvyn ymmärtämiseksi vertasimme instrumentoidun ja instrumentoimattoman ajonaikaisen järjestelmän suorituskykyä. Kehitetyllä mittausjärjestelmällä on matala mittauskuormitus.

Asiasanat:	Rinnakkaisuus, Tehtävärinnakkaisuus, Ajonaikainen järjestelmä, Mittaus, OpenMP
Kieli:	Englanti

Acknowledgements

I would like to thank my supervisor, Associate Professor Keijo Heljanko, for his supervising and ideas for improving the thesis.

I would also like to thank my instructor, Vesa Hirvisalo, for the possibility to write the thesis and his frequent help with the writing.

Finally, I would like to thank my parents, siblings and friends for their support during the writing.

Espoo, April 27, 2015

Marko Rasa

Abbreviations and Acronyms

AES	Advanced Encryption Standard
API	Application Programming Interface
CBC	Cipher Block Chaining
DRAM	Dynamic random-access memory
ECB	Electronic Codebook
GDB	The GNU Project Debugger
GOMP	GNU OpenMP
GPU	Graphical Processing Unit
HMAC	Hash-based Message Authentication Code
MCA	Multicore Association
NIST	National Institute of Standards and Technology
OpenEM	Open Event Machine
OpenMP	Open Multi-Processing
PAPI	Performance Application Programming Interface
RAPL	Running Average Power Limit
RFC	Request For Comments
SHA	Secure Hash Algorithm
TLS	Transport Layer Security

Contents

Chapter 1

Introduction

In this thesis, we have studied mechanisms to measure behavior of OpenMP task scheduling.

The world has been shifting towards parallel computing due to improvements on clock speed not keeping up with the demand. One way to implement parallel computing is by dividing the work to tasks and schedule the tasks for execution. The scheduling itself requires time, effort and space, which are away from the execution of the payload of the tasks. The scheduling can be supported by the hardware or various software mechanisms.

The more tasks there are in an execution of a program the more they burden the system, as additional resources for handling the tasks are required, which is away from the execution of the payload. Measuring an OpenMP runtime helps to detect where the runtime and possibly the associated specification could be improved.

1.1 Problem

The goal has been to develop a measurement system that is able to yield the basic tasking performance metrics of parallel program executions. The main challenge in developing such a measurement system is the overhead caused by the measuring as tasks can be quite small. In addition to mere measuring, the aim has been to understand the scheduling properties of the current implementations of OpenMP task parallelism model.

1.2 Method

We selected version 4.9 of the gcc compiler and its runtime (gomp) as the concrete target of our study. The compiler implements essential parts of the

OpenMP 4.0 specification, including tasking with task dependencies. The task scheduling operates by assigning tasks to worker threads. In doing so, it handles dependence structures and task queues, which require synchronization to ensure correct operation. The studied gomp implementation uses locking for that purpose.

We designed a measuring system based on using both software and hardware performance counters. The software performance counters were implemented by instrumenting the runtime systems (gomp). The hardware performance counters were implemented by using PAPI (Performance Application Programming Interface) inside the runtime system. We tested the measurement system by implementing a task parallel version of AES (Advanced Encryption Standard), which yields small granularity tasks.

1.3 Results

We measured performance qualities of OpenMP task scheduling model by developing an embedded monitoring system in GOMP. The monitoring system was tested and evaluated by implementing a parallel cryptographic algorithm with couple variations. The main variation of the workload has very high granularity tasks.

Overhead caused by the monitoring system stayed on reasonable levels and didn't show signs of growing with more threads. The runtime performance was measured by executing the workload with the custom runtime. These results were also compared to the non-instrumented runtime.

In our tests we observed that the runtime was able to yield parallel scaling up to four threads in the fine grained workload. At this tipping point, we observed lock congestion to rise rapidly.

1.4 Structure of the Thesis

The thesis is structured into background chapters and contribution chapters. Chapters 2, 3 and 4 give background on parallel computing, runtimes for tasking and OpenMP runtimes respectively. Chapter 5 talks about monitoring GOMP, the OpenMP runtime chosen for the concrete study of this thesis. Chapter 6 describes The implemented test workload for the measurements and Chapter 7 provides the results of the measurements. Chapter 8 concludes the thesis.

Chapter 2

Background

2.1 Parallel Computers

Parallel execution is increasingly important for development of computer hardware. This change requires attention to computer architecture details such as memory hierarchy, different parallelism levels, multicore processors and synchronisation of the processor cores. This section largely cites Hennessy and Patterson [35].

Parallel processors have multiple separate processor cores, which can execute different program code at the same time. If the cores execute program codes that are related to each other, for example parts of the same program, they and their L1 caches need enough synchronisation that the program runs correctly. Normally only one program thread can be in execution at a time within a single processor core. However, if the processor implements simultaneous multithreading, such as Hyper-Threading, a processor core can have instructions from two or more, depending on the architecture, threads in its pipeline at the same time.

Memory hierarchy consists of L1 cache that is closest to a processor core, fastest to access, but small in size. Each processor core typically has its own L1 cache. If the processor has also both, L2 and L3 caches, L2 cache might be shared between pair of processor cores and L3 shared by the whole processor. If a required data is not found in L1 cache, it will be looked for from L2 cache and from L3 cache if still not found. The next step is to look from the DRAM memory, which is located outside of the processor and thus has a large increase in latency.

Hennessy and Patterson divide parallelism into Data-Level Parallelism and Task-Level Parallelism. Data-Level Parallelism has processor cores executing the exact same code, but on different data, which is typical on GPUs

(Graphics Processing Units). Task-Level Parallelism contains both thread level parallel model and task parallel model, which is also called tasking model. Thread parallel model has multiple threads executing at the same time with possibly different program code on each. Task parallel model can be seen to be implemented on top of the thread parallel model. Task parallel model is further explained in Section 2.3.

In addition to the above, processors have Instruction-Level Parallelism, where multiple instructions are in execution simultaneously in so called instruction pipeline. The instruction execution is divided into multiple steps and a processor core can have an instruction in each step, but the instructions cannot depend on each other. For that, the compiler will typically attempt to reorder instructions of a program in such a way that Instruction-Level Parallelism is as effective as possible while not changing the behaviour of the program.

New processors can have high amount of processor cores, such as Intel's Knights Landing [28] with 72 cores. Utilising all available performance from such processors will require large amount of parallel tasks.

2.2 Parallel Computing

Processors used to calculate everything in serial, but eventually the increase in clock speed could not keep up with the demand for faster computation [35]. Solution for this was found from parallel computation, in which each processor has multiple cores, each computing different code at the same time. Thus, the calculation power of processors was increased through having more than one core in the processor.

With the increase of multicore processors, the importance of concurrent and parallel programming also increased. Concurrent programming [30] concentrates on utilising coordination for correctness of the execution, which means that the result should be the same as with a serial counterpart of the program and the program should not lock due to behaviours of threaded execution. Concurrent programming is useful even on a single core machine, as it divides the execution time between threads, which can for example make interface of a program more responsive.

Parallel programming [47] concentrates on utilising all the performance from the processor cores by executing the program in multiple threads at the same time. This requires the program to have parts that are not too dependent on each other.

2.3 Task Parallelism

Task parallelism models [46] consider parallel execution as entities consisting of code to be executed and the related data. The task system gives such entities the handles for execution. In this respect, tasks for parallel execution resemble procedural closures. Individual tasks may be executed in parts, but a natural way is to execute them unsuspended and leave the synchronisation up to resolving the inter-task dependencies.

This contrasts with the traditional thread parallelism [35] that bases synchronisation on placing primitives within the code. Thus, instead of abstracting the parallelism with execution handles, thread parallelism exposes the hardware program counters in a raw manner to the programmer.

Inter-task dependencies are unrestricted in task parallel programming models. Whereas in data parallelism [35], the dependencies should follow the structure of the underlying data. Task parallelism assumes no replicated structures in the underlying data to directly indicate the structure of the parallelism.

2.4 Runtime System for Parallel Programs

Runtime system works as an interface between compiler and operating system. Operating system provides resources [38], such as memory allocation/deallocation, call stack and threads to the runtime and the runtime works as an abstraction to these resources for the compiler to utilise. A program using the runtime can then be compiled by the compiler and can use the resources during execution through the runtime library.

Runtimes for parallel programs are especially interested in threads. To get any performance gains, the underlying hardware needs to provide hardware threads. In multicore environments, each processor core provides at least one hardware thread. In case of simultaneous multithreading, each core provides multiple threads. However, the performance gain from using both or all of the threads provided by simultaneous multithreading vary and may be quite small in some cases.

The operating system running on top of the hardware will have software threads. The operating system can assign the software threads to run on the hardware threads. If there are not enough hardware threads, the rest of the software threads will wait their turn.

These operating system threads are further provided for usage by the runtime library, which allows the usage of threads or some abstraction built on top of the threads to the program using the runtime.

Some programming languages are designed to run their binaries in a virtual machine of the language [36]. The virtual machine in this case is basically a runtime environment for running the code compiled from the language in question.

Different task parallel runtimes include [31], but is not limited to: MPI, OpenMP [23], Cilk [2], TBB [17], Java Threads, Fork/Join framework, TPL, CAF, UPC, Fortress, Chapel, X10, CnC, Parallel Haskells, Erlang, Manticore, Offload, SkePU and P3L.

Chapter 3

Runtime System Support for Tasking

Runtime systems for parallelisation provide userland programs tools for parallel execution. The runtime acquires the necessary resources from underlying operating system.

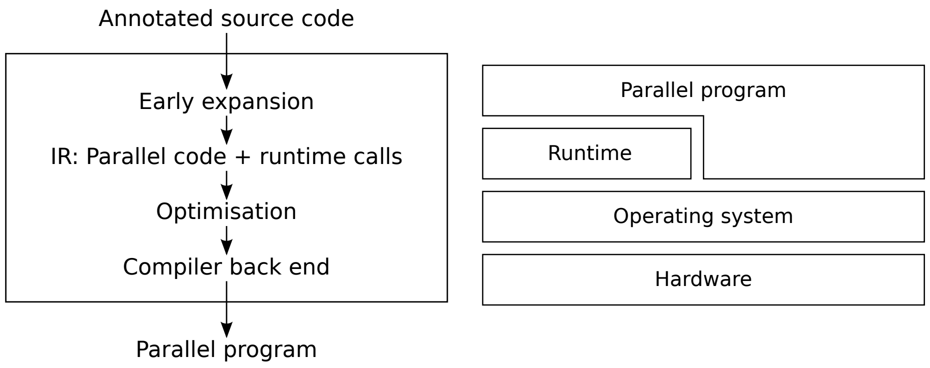

Figure 3.1: Compilation process for parallel program code on left and platform stack on right

The left side of Figure 3.1 pictures a currently typical compilation process for parallel program as explained by Pop et al. [45]. The process starts with source code annotated with parallelisation directives. The annotations from the code are first expanded into runtime calls and other code necessary for the runtime structures. This result is fed into optimisation passes and finally to the compiler backend which converts the optimised intermediate representation into executable program.

The right side of Figure 3.1 shows how hardware, operating system, runtime and a parallel program are built on top of each other. Operating system has direct access to the hardware and provides abstractions of the hardware resources to the layers on top. Parallelisation runtime uses the abstractions provided by the operating system and offers tools for creating parallel programs. The parallel program built on top of the runtime uses the tools provided by the runtime. The program also accesses the abstractions provided by the operating system, but likely only for other needs and leaves all the parallelism needs for the runtime.

Runtimes for task parallelisation interact with compiler and operating system in similar manner as other parallelisation runtimes. Annotations and function calls to the runtime are different, but compiler and operating system handling of tasking annotations is similar to the handling of other parallelisation annotations. More about task parallelism with runtimes on Sections 3.1 to 3.3.

Section 3.4 concentrates on how the functioning of runtime can be monitored and Section 3.5 talks more about the scheduling aspect of task parallelism runtimes.

3.1 Task Models

Task models are parallel computation models where computation is divided into tasks. The runtime for a task model contains a pool of tasks, which usually can be created during the execution, and pool of worker threads, which execute those tasks in parallel. The details of the tasks and pools vary between different runtimes, but the main characteristics stay the same. This section mainly references Belikov et al. [31].

Task parallel models differ from the thread parallel models in the management of the threads. In task parallel models the worker threads process multiple tasks during their lifetime, whereas in thread parallel models the thread is terminated after its workload is done. Thread parallel models tend to favour larger workloads to avoid overhead from thread creation, whereas creating tasks in task parallel models is comparatively light operation. Task parallel models also have static number of worker threads, whereas in thread parallel models the programmer need to pay attention to the number of threads in the program. Synchronisation in thread parallel models is done inside the code executed by the thread, which makes the relation of different threads less apparent. Whereas, depending how well the task parallel model supports dependencies between tasks, most of the synchronisation of the tasks can be done in the handling of the tasks with the task dependencies making the task relations more apparent.

Task parallel runtimes have differences between each other, even though the main idea behind each of them is the same. As classified by Belikov et al. [31], the abstraction level of the runtime can vary from low level, where the programmer is exposed to most coordination issues, to high level, which abstracts over most of the coordination issues. Various different memory models are used in different parallelism runtimes. Some runtimes have such a programming model that they guarantee the parallel program to have the same result as a serial counterpart. For other runtimes the programmer needs to pay attention to make the result same when it is desirable. Embedding the runtime directives to the host language can also be achieved in multiple different ways ranging from just a library to an entirely new language. Embedding parallelisation as a library has a drawback of being restricted to the optimisations available in the host language, whereas a new language has all the problems of any new language.

Event models, such as OpenEM [13], organise the tasks with events, execution objects and event queues. Events hold the data to process. Execution object contain the code to execute for given data. Event queue contains the events waiting to be processed by execution objects. Event models are more restricted and straightforward than other task models, which helps their efficient and scalable scheduling with hardware scheduler. OpenMP 4.0, which is the focus of this thesis, on the other hand has much more complex and dynamic tasks. More on the OpenMP task model in Section 3.2.

3.2 OpenMP Task Model

Tasks were introduced into OpenMP in version 3.0 [20] of the specification and extended in OpenMP 4.0 with, for example, dependencies between tasks [22]. Original focus of OpenMP was in parallel for-loops. Multiple hardware vendors contribute to the OpenMP specification in order to have more standard directives for parallelism [39]. OpenMP specification specifies a set of compiler directives, library routines and environment variables that can be used to create parallel C, C++ or Fortran code [14].

OpenMP is designed around shared memory model [23], which means that all the threads of the program have access to the same memory, which needs to be kept consistent between threads using the memory. In addition, depending on the implementation, threads may have their own view of the memory working as a cache.

OpenMP tasks contain block of code to execute and data to operate on. If a task is created inside another task, it will have the other task as its parent task and every other task created directly inside that parent task as its sibling tasks.

OpenMP 4.0 added possibility for dependencies between sibling tasks. The task can be specified to have in, out or inout dependencies to variables, which correspond to reading, writing and doing both to those variables. If a task has out dependency on a variable, the task will require all sibling tasks depending in any way to the same variable and created before this task to be finished before this task can be started. With an in dependency, the task will require all previously created sibling tasks with out dependency to the same variable to be finished.

If not otherwise specified, OpenMP tasks are tied [39], which means that whichever thread starts the execution of a task will keep the task until it is executed. A thread may switch which task is currently being executed at a task scheduling point. However, if starting a new tied task, the task must be descendant of all the tasks already tied to the thread. More about task scheduling in Section 3.5.

3.2.1 Threads

At the parallel directive, OpenMP creates a group of threads, which in OpenMP is called a team [23]. Each thread of the created team will then start to execute the contents of the parallel section.

A useful construct with the task model is to use a single directive directly inside the parallel directive and create initial tasks in the single directive. In this case, one of the threads will execute the single contents of the single directive and the rest will skip the single and continue to the end of the parallel section. The threads will hit an implicit barrier at the end of the parallel block, which will cause them to wait for either the last thread or new tasks to execute.

3.2.2 Tasks

The task directive was added to OpenMP specification in the version 3.0. At a task directive, OpenMP creates a new task. In the case of libgomp, the new task is placed to the end of the task queue and the current task continues execution.

Any thread waiting in a barrier will pick the first task from the queue when there are tasks available. This will in effect result in breadth first traversal of the tasks. When a task gets finished, the thread will return to the barrier and pick another task, if available.

3.2.3 Tied Versus Untied Tasks

OpenMP tasks can be either tied or untied [23]. When a thread picks a tied task for execution, the task will be tied to the thread and no other thread can work on the task. In comparison, untied task does not have such restriction. If untied task is not in execution at the moment, any free thread can pick the task to continue its execution. By default, OpenMP creates tied tasks, but untied task can be created by adding a clause to the task directive.

Duran et. al. [34] have experimented with different schedulers for OpenMP 3.0. They tested breadth first and work first schedulers with different work stealing strategies. They concluded that work first schedulers are better in general, but tied tasks, which are default in OpenMP, severely restrict the performance of the work first schedulers. And that in the default circumstances for OpenMP, the breadth first schedulers outperform the work first schedulers.

3.2.4 Dependencies

OpenMP 4.0 adds the possibility to define dependencies between tasks. When a task has input dependencies, all sibling tasks created before the task in question, which have a requested input as their output dependency, need to be finished before executing the task in question [23].

In libgomp, the tasks depending on other tasks will not get placed to the task queue for execution. Instead, they will get added into a structure for tasks with dependencies. A task's counter for unfulfilled dependencies will be reduced every time its dependency task gets executed. When the counter reaches zero, the task is ready for execution and will be added to the end of the task queue.

3.2.5 Task Scheduling

In task scheduling, OpenMP threads are used as worker threads. The threads themselves are scheduled by the underlying operating system, but the thread scheduling has no significant impact to the task scheduling. A task is executed by a thread until the task is either finished or a task scheduling point is reached in the code of the task. Task scheduling points include creation of new task, taskwait directive and barriers. At a task scheduling point, the thread may switch to another task or continue executing the current task. [23]

Because the task scheduling points do not force any change, an OpenMP implementation might ignore some of them. For example, current implementation of GOMP fully ignores taskyield directive, which would add task scheduling point, but won't break anything when ignored. Other task scheduling points, such as barriers and taskwaits, must be implemented to avoid deadlocks.

3.3 Runtime Support of Tasking

Multicore platforms are computation platforms with multiple cores. They are designed for parallel computation. To support parallel computations, the multicore platform needs to have some level of communication between different cores or threads, resource management for shared memory and task management for threads and tasks. This section considers the model used by Multicore Association (MCA) [11].

Over the years the computational platforms have shifted towards multicore processing leading to the growing importance of multicore platforms. Unlike single core platforms, multicore platforms require attention to communication between cores, managing resource usage between different tasks and managing the tasks themselves.

3.3.1 Communication

During some operations, the cores need to synchronise their data. Some of the communication is typically done automatically by the platform, for example if a core has executed all its work, the platform might automatically give it more work from another core. Other kinds of communication might require the programmer to mark when it should be done. For example, if the cores need access to the same shared memory, the programmer might need to mark when to synchronise that shared memory with the core's own view of the memory.

Communication can be done with either blocking or non-blocking operations. Blocking operations will pause the execution until the operation is ready and returns. For example, a blocking read operation will wait until it has something to read and then returns the data which was read. In comparison, non-blocking operations will return immediately and if the requirements for the operation were not fulfilled, the operation will return a value informing of failure. In the case of reading, the operation would succeed, if a buffer for the operation had something and fail if the buffer was empty.

Multicore platforms have some overhead and latency compared to singlecore platforms due to synchronisation of the cores. However, the overhead is smaller than the gain from the parallelisation. In addition to differences on overhead and latency, multicore platforms may also differ on how well they scale when more cores are added to the system.

3.3.2 Resource Management

Multicore platforms need to handle both thread private and shared memory. Thread private memory is only accessible by the specific thread and as such can be handled easily. For shared memory, attention needs to be paid to latency and concurrent access.

Some basic synchronisation is required within the resource management of multicore platforms to handle concurrent access to resources. This could be, for example, a basic lock to let only one thread to use a resource at a time.

3.3.3 Task Management

Task management in multicore platforms need to handle the tasks from their creation to when they are finished and destroyed. The created tasks need to be stored in some form until a thread picks them for execution. One simple way to store a task is storing its code as a function and the arguments for that function as data. The data can then be placed in a queue or other data structure to wait for its turn.

After the task is created, a task scheduler of the multicore platform will choose when and in which thread or core the task will be executed. Tasks can have different priorities, which define the order in which the tasks get executed. Also, the tasks may have dependencies between each other, which add further restriction on the execution order.

Different platforms may have different requirements on the decision of which thread or core a task is executed on. The typical goal is to achieve good load balance, so that each available core is efficiently used. On some architectures, the distance between task origin and a core may have a role on the performance [40]. And on other situations, the scheduling may aim to keep the core temperatures even between cores [44]. Olivier et.al. [42] compared OpenMP 3.0 task scheduling on a platform with importance on cache locality.

3.4 Runtime Monitoring

Runtime monitoring can be done in various different ways. Lowest level approach is to read the performance monitor counters manually with an instruction for reading the counters. Another approach for monitoring is to get some performance information from Linux kernel. The kernel tracks various statistics related to the performance of programs. Tools for debugging used for programming also include measuring performance. Of course some external programs exist for specifically measuring performance of other programs. And finally, with access to source code, runtime or some workload for it can be instrumented to get more fine grained or specific measurements.

Modern computer processors have performance counters, which can be used to read performance metrics of the processor. The performance counters are special registers in the processor that store performance metrics of the processor's activity. The available counters and how many of them can be read at once vary between different processors.

Linux kernel tracks some performance statistics of programs through its own events. Application performance and its kernel event trace can be tracked for example with KProbes [9], which is intended as a debugging mechanism for the Linux kernel, or LTTng [10], which is a toolkit for tracing Linux systems and applications and an acronym for Linux Trace Toolkit: next generation.

Tools for debugging, such as GDB [6] and gprof [7] can be used for measuring the performance statistics of a program. Part of program development is optimising its performance, which requires measurements for the performance to be optimised. Thus, the programming tools can be also used for measuring the performance and tracing the execution.

Performance measuring also has its own tools, such as perf [16]. These tools measure the performance of another program. The tools work by reading the performance counters mentioned earlier in this section and make the measurement of the counters much easier by creating an abstraction on the low level access to the processor registers.

With access to the source code of the runtime to be monitored or a workload program using the runtime, the source code can be instrumented for measuring the performance of the runtime. Some performance metrics can be acquired by instrumenting just the workload, but other details require instrumentation code inside the runtime source. In addition to basic time measurements, performance measurements can be made by accessing hardware performance counters. PAPI [15], Performance Application Programming Interface, provides an abstraction as a library for reading the performance counters.

Approach for runtime monitoring for this thesis is to instrument runtime and workload source code with PAPI and time measurements. In addition, the runtime will be modified to print some of its internal information about the program execution.

3.5 Task Scheduling

Task parallel systems use threads as worker threads. However, thread scheduling has only a minor role on task scheduling. Task scheduling is more concerned on the worker threads switching tasks on specific places in the program code and in which order the tasks are taken from the pool of available tasks. Work-first [41] scheduling switches the executing thread to execute the created task, whereas help-first [41] scheduling continues executing the parent task resembling breadth first execution. Cilk and OpenMP will be mentioned as examples on work-first and help-first scheduling approaches.

Worker threads are typically acquired from the underlying operating system. Also, the scheduling of those threads is handled by the operating system. However, the thread scheduling has very little effect on task scheduling, as task scheduling can be done inside each worker thread when they need to look for more work to do.

Ideally the runtime would have as many worker threads as the machine has hardware threads available and each of the threads would be executing tasks all the time. In this case, all the available resources for calculations would be utilised. However, in a less ideal situation some worker thread may get stuck either with a task waiting for a resource or without a task. The possibility of this of course depends on the situation and the runtime being used, as different runtimes have varying amount of freedom for choosing the next task to execute.

Work-first [41] approach for task scheduling is used in Cilk. With work-first approach, whenever a worker thread creates a new task, the parent task is suspended and the thread continues execution to the newly created task. This causes the execution order to slightly resemble that of the sequential execution. If sequential execution has good temporal locality, the parallel execution will have good temporal locality as well. Olivier et.al. have looked into Cilk and OpenMP 3.0 performance on unbalanced task graphs [43].

To function, a runtime with work-first scheduling approach must be able to migrate partially executed tasks between threads, because all the free tasks will be partially executed. Help-first [41] approach for task scheduling on the other hand will continue the parent task when a new task is created. The help-first approach can be implemented without such migration, as the new tasks are placed into queue without starting their execution.

If not mentioned otherwise on task creation on OpenMP, the task will be tied meaning it can't be migrated to any other thread after starting its execution. Untied tasks would be allowed to migrate, but tasks are only untied if mentioned so and OpenMP implementations are not required to implement untied tasks. Thus, OpenMP runtimes have mainly help-first or similarly behaving scheduler.

Chapter 4

An OpenMP Runtime System

There are three parts required for the OpenMP: the OpenMP specification [23], a runtime and a compiler which is able to convert the OpenMP directives into calls to the runtime. OpenMP specification specifies the directives and other details that are required from a compliant OpenMP runtime. Runtime is the part which contains the implementation of the OpenMP. Thus, runtime is the only one of these active during program execution.

4.1 Platform Selection

Two choices of OpenMP runtimes were considered: GNU implementation called GOMP [27] and Intel implementation called iomp5 [8]. At the time of the selection, GOMP worked with GCC and iomp5 worked with a branch from Clang called OpenMP/Clang [29], which was being cleaned for inclusion into the main branch. Iomp5 combined with GCC produced wrong behaviour with dependencies between tasks.

GCC 4.9 adds support for OpenMP 4.0 [26]. As expected, this works correctly with libgomp. However, with libiomp5, the task dependencies were tested to not work even with the update of 4th September 2014 [37]. Another correctly functioning combination that was tested was a branch for Clang [29] with libiomp5.

The combination of GCC and GOMP was chosen for this thesis. GCC had had the support for OpenMP 4.0 for longer time suggesting more stability and less bugs in the implementation.

4.2 OpenMP as a Multicore Platform

This section will give a bit background on how OpenMP functions as a multicore platform. The subsections will divide the subject into coordination in Section 4.2.1, memory management in Section 4.2.2 and thread and task management in Section 4.2.3.

4.2.1 Coordination

OpenMP abstracts away most communication and synchronisation between threads. For example, for mutual exclusion, the OpenMP specification [23] has a simple directive for marking a structured block as a critical section. The specification does also have locks for lower level synchronisation.

OpenMP specification [23] states that the `parallel` directive will start the parallel execution. At this point, the OpenMP runtime will create a team of threads, which will execute the contents of the parallel region. Normal code in the parallel section get executed by each of the threads. Some directives, such as `single` and `master`, get executed by only a single thread and some directives, such as `for`, will be cooperated on by all of the threads together.

If a thread encounters a `task` directive [23], it will create a task that will be executed at a later time. Such task will be scheduled for execution when any thread hits an implicit or explicit barrier. An explicit barrier is marked by the user of the API with a `barrier` directive and an implicit barrier is implied by some other directive. For example, at the end of a parallel region is an implicit barrier. At any barrier, the threads that get there need to execute the accumulated tasks and wait for all other threads of the team to get to the barrier.

4.2.2 Memory Management

Managing the physical memory is still left to the operating system, but an OpenMP API needs to handle the distinction of variables shared between threads and variables private to each thread [23].

In the case of variables private to each thread, each of the threads have their own private version of the variable. The initial value for this variable may be copied from the original variable at the beginning of a parallel section or the private variable may be initialised with the default value for that variable type.

For shared variables, each thread has their own temporary view of the variable in addition to the global state. The thread's view can be made consistent with the global state with an implicit or explicit flush. The flush will write the variable to global state, if there were any changes to the variable, and the next time the variable is read, the read is done from the global state.

4.2.3 Thread and Task Management

The parallel directive of OpenMP forks the execution into multiple threads and joins them back at the end of the structured block of the parallel directive [23]. These threads are then used to execute the parallel block and any tasks spawned from the block. In GOMP, the scheduling of threads is left to the operating system and the scheduling of tasks simply chooses the next task to execute in certain locations explained in more detail in the following paragraphs.

In GOMP, a task execution can start in four different places: creation of the task, an implicit or explicit barrier, a taskwait directive and at the end of taskgroup directive. A task will be executed immediately at the creation only if we want to execute it as if it were just sequential code. The other three locations will form the scheduling of the tasks.

In a barrier, GOMP thread will pick any free task created by the team of threads. The scheduling made in a barrier will pick the first task that was added in the queue, which results in first in first out order. The threads will execute any new tasks until all threads have reached the barrier and all created tasks are executed.

At the taskwait directive, the thread will pick a child of the current task. The children of a task will behave like a stack, resulting in a first in last out order. Already completed child tasks will be put into the bottom of the stack and cleared at a later time. If all the children are already being executed, the thread will wait for them to be finished. This waiting may result in inefficiency, as the thread will not pick anything else to do while waiting, even though the OpenMP specification would allow the thread to pick any descendant task for execution and not just immediate children.

4.3 Compiler

Compiler converts OpenMP directives into code containing calls to runtime functions. In case of GCC, the functions to use are taken from libgomp [27]. To use another runtime with the compiler, the other runtime must either work with the same function calls or the compiler needs to be modified. Code construct that jumps away from parallel region are not allowed, such as break and goto. However, the OpenMP specification [23] defines that a compiler may assume users to write code conforming to the specification. So, it is up to the compiler if the code conformity is checked or not.

The compiler converts structured blocks affected by parallel and task directives into functions. This makes it easier for the parallel directive to start new threads and the execution of a task directive to be postponed and makes executing tasks on different threads simpler.

4.4 Operating System

Operating system offers some necessary resources for the OpenMP runtime to use. For the case of GOMP, the required services are memory and threads.

Like any other program, OpenMP runtime won't use physical memory directly. Instead, the program utilising memory will get a virtual memory address space from the operating system and the runtime will use the program's virtual memory address space.

When creating a team of threads with the parallel directive, OpenMP asks for threads from the operating system. In the case of libgomp, the threads are acquired through the pthread library [33].

4.5 Software Structure of GOMP

The source code of GOMP resides in the repository of GCC [5] in a directory named libgomp. The directory contains various c-files from which `task.c` is the most important for task parallelism and contains functions related to tasks. Other interesting files include: `team.c` for team related functions and `parallel.c` for functions related to parallel directive.

Some essential source files are further in the directory structure, as they are platform specific. So, in the case of Linux system, `mutex.c`, `mutex.h` and `bar.c` can be found under `libgomp/config/linux` among with less relevant files for the purpose of this thesis. Mutex files contain mutex lock and unlock functions, which are needed for a `task_lock` every time task queues and dependencies are handled. `bar.c` contain the code for when a barrier is reached, which also is one location for starting the execution of tasks.

`libgomp.h` is used as an internal header file for most c-files and contain structure definitions, enumerations and function declarations that are needed in multiple places. Notably for task parallelism, `libgomp.h` contains definitions for task, thread and team structures.

GOMP is compiled along with GCC compilation [4]. That is, by getting the GCC source, configuring to a build directory with the provided configuration script and building with make. This should result in GOMP shared object files in `build/<architecture>/libgomp/.libs/`. Setting environment variable `LD_LIBRARY_PATH` to point to this directory will cause programs using GOMP runtime to use the newly compiled version instead of the systemwide version.

The GOMP used in this thesis is from the GCC version 4.9 release tag from the GCC repository [5].

4.6 Interfaces

GOMP interfaces to multiple directions. On user side, the interface is with directives, which are implemented as pragmas in C, and API calls. On compiler side, these directives are converted to runtime calls and surrounding code, such as converting task content into a function. Interfacing with operating system is done mainly through pthread and standard C libraries. Though, few system specific header files are also used inside the system specific components in the `libgomp/config/` directory.

4.7 Data Structures

Basic data structure definitions can be found from `libgomp.h`. For tasking, the `gomp_task`, `gomp_taskgroup`, `gomp_thread` and `gomp_team` structures are most relevant. Task dependency structure has its hash table and other definitions in `hashtab.h`. Task group is handled in a similar fashion as child task relation, but was not used in the experiment of this thesis and not studied in depth.

`gomp_task` structure holds data of the task and a pointer to a function representing the program code of the function. In addition to these, the task structure holds variables for its state, queue for child tasks and pointers forward and backward for each queue the task is in. The state variables tell if the task is tied, waiting in taskwait directive and if the task is final task, meaning any new tasks need to be executed immediately.

`gomp_thread` structure holds information about a thread, such as a function pointer and data to the function the thread is to run upon launch, its current task, pointer to a thread pool, a place where the thread is bound to and state of its current thread team. The function of the thread is the function resulting from converting the structured block of the current parallel region into function during preprocessing of the source code. Current task of the thread can be either explicit, created with a task directive, or implicit, created by the runtime without a task directive. The thread pool is the pthread thread pool in which this thread belongs to. The place of the thread is zero if the thread is not bound anywhere and the bind position plus one if the thread is bound. Finally, the thread has a state of its current team as `gomp_team_state` structure. The thread can access the team it belongs to either through the team state structure or through the thread pool.

`gomp_team` structure contain notably the number of threads in the team, a task lock, queue of all tasks ready to be run and various counts of tasks. The team contains the number of threads in the team, but no obvious way to access the team. However, the team is always accessed through the current thread, which can access the thread pool. The task lock of the team is used to synchronise the access to the tasks of the team. Any time a task that is not completely isolated is touched, the task lock is locked during the operation. The head for the queue of all tasks of the team is located in the team structure. The task counts stored in the team structure include number of waiting or tied tasks in the team, number of waiting tasks in the team and number of tasks being directly run from a barrier, which is always at most the number of threads in the team.

4.8 Static Operation

During compile time, the compiler converts OpenMP pragmas into code. The conversion is mostly a simple matter of converting the pragma into a function call to a corresponding function in the runtime and possibly adding some surrounding code or converting the structured block after the pragma into a function, like task directive does.

The function calls resulting from the conversion of pragmas are the entry points to the runtime for the program.

Because of the early conversion of pragmas into typical serial code in the preprocessor of the compiler, parallel aware optimisation is not possible. Also, at least most of the pragmas result in code that work as a barrier for typical optimisations.

4.9 Dynamic Operation

This section explains what happens inside the GOMP runtime during program execution.

4.9.1 Objects

Parallel directive will start the parallel execution and create a `gomp_team` structure. Creating and starting the team is implemented in `team.c` in functions `gomp_new_team` and `gomp_team_start`. Team is the structure which holds the parallelism together by representing the collection of threads and containing queue for pending tasks.

When a team is created, number of threads are created. The original thread starts the new threads from the `gomp_team_start` function and the new threads start to execute `gomp_thread_start` function. This function directs them to execute the body code of the parallel section. In a simple tasking setup, one thread will create tasks in `single` section and the rest will hit the barrier without doing much anything. The threads will wait in a futex for the last thread and pick any tasks that get created for execution.

Tasks are created when there is a task directive in the code. The structured block following the directive is converted into a function in the compilation process and the resulting task will call that function when run. The creation is handled in `task.c` in `GOMP_task` function.

Task creation has four conditions for when a task would be executed immediately instead of put into a queue waiting for later execution. Firstly, if a task is created without having a team, it needs to be executed immediately, as we don't have any parallel context. Secondly, if a user defined if clause for the task creation tells to execute immediately, the task will be executed immediately. Third, if the task is created inside a final task, it is executed immediately by the definition of final tasks [23]. Finally, if there are already too many tasks waiting, any new tasks will be run immediately if able, where too many is defined as $64 \times numberofthreads$.

If however the task had some unfulfilled dependencies, it is not possible to execute immediately even if the above reasons would say so. In such case, an entry to a dependency hash table will be added for the task. The entry of the new task is marked to any task the new task depends on and a counter for the number of those links added to the entry. Whenever a task gets finished, it'll reduce the counter of each linked entries by one. If such counter reaches zero, it means that all dependencies for that task are fulfilled and it will be moved from the dependency hash table to the task queues.

If a task didn't have any unfulfilled dependencies or its dependencies got fulfilled, it will be added to task queues. A task can exist in multiple of the three available queues at the same time. First, it'll be in a task queue under team, which behaves in a first-in first-out manner. Whenever a thread picks a task from implicit or explicit barrier, this queue is used. Second, the task will be in a child queue of its parent task, if the task has a parent. This queue will be used when a thread picks a task from a taskwait directive and behaves in first-in last-out manner. Additionally, in the case of GOMP, a thread is only able to pick a direct child of the current task when in taskwait. Specification allows and other runtimes implement picking any descendant. Lastly, the task will be in a task group children queue if it is created from inside a task group. This group behaves similarly to the children queue of a parent task.

Chapter 5

Runtime Monitoring

In this chapter, we describe how the chosen OpenMP runtime, libgomp, was instrumented for monitoring. The runtime was monitored for three different aspects during the thesis: dependency graph of the tasks in the workload, task lock congestion and time spent in the runtime.

There would be existing profiling tools for OpenMP, such as ompP [12]. However, using a ready made tool lacks the flexibility of an embedded monitor created for the exact need, which was deemed important together with gaining insight of the system while developing the monitor.

5.1 Current Operation of GOMP

Tasks are created with OpenMP task directive. This directive will result in a call to `GOMP_task`, which will create a structure for the new task and either add it to dependency structures, add it to task queues or execute it immediately. The case of executing task immediately is chosen when the number of existing tasks is getting too high or a condition defined by the user specifies that the task needs to be executed immediately. However, if the task has unfulfilled dependencies, it will never be executed immediately and will always go to a structure for tasks with dependencies. If the task is neither executed immediately nor depend on other tasks, it will go to normal task queues to wait for execution.

As defined in the OpenMP specification [23], task depends on all previously created sibling tasks which have a variable as an output dependency that the new task has as an input dependency. Tasks with dependencies are stored into a hash table. Every time a dependency is fulfilled from a task, its dependee count is decreased. When the count reaches zero, the task will be added to the task queues the same way that tasks without dependencies were added during their creation.

Preprocessing for the tasks is done before the task is executed. The preprocessing removes the task from task queues, marks the task to be in the execution and checks if the parallel execution has been cancelled. After the task has finished execution, some postprocessing is required to keep the structures in correct state. Postprocessing checks if any tasks has their dependencies finished requiring moving to the task queues and clears structures that are not needed anymore.

Libgomp is mostly a run to completion system. At barrier, taskwait and taskgroup end, the running thread will leave its current task to the stack and pick another task to process. When the other task is finished, the original task will be continued if possible. Additionally, taskwait and taskgroup end will only pick tasks which are their immediate children and if all of them are being executed on other threads, the thread will idle until they are finished. This is not the case with libiomp5, which will pick also other tasks when in those situations.

Barriers should not be used inside tasks, as threads continue from a barrier when all threads have reached the barrier and the number of tasks might not be divisible with the number of threads. Misusing the barriers by placing them inside tasks may cause deadlocks.

Some synchronisation, like critical directive and locks, cause the thread to wait for its turn. Libgomp will let the thread idle while waiting. Being inside or outside of task will have no effect for this case. Having other threads which the operating system can run while one thread is waiting can increase the efficiency of CPU usage.

5.2 Monitoring System Overview

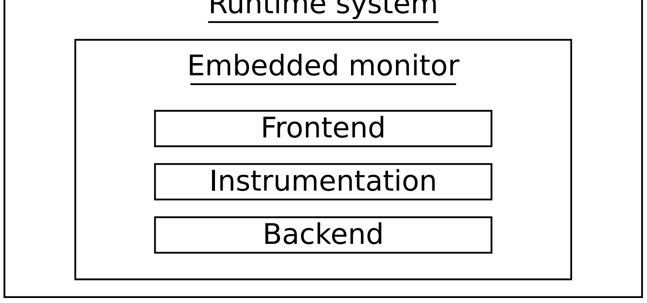

Figure 5.1: Structure of the embedded monitor.

The structure of the implemented embedded monitor is pictured in Figure 5.1. The monitor is embedded into the runtime system under study, libgomp. The embedded monitor consists of three parts: frontend, instrumentation and backend. Frontend initialises PAPI and sets any added variables to their initial values. Instrumentation does the actual measurements and cooperates with backend to store the result into variables working as intermediate storage for the results. Latter half of the backend is run at the end of the measured execution and outputs the results from their intermediate storage.

5.2.1 Frontend

The frontend of the monitoring system is responsible for initialising PAPI and setting initial values for variables added to the runtime for monitoring. As the structure of the workload permits, the frontend is located at the team initialisation of GOMP activated by parallel directive of OpenMP.

5.2.2 Backend

Backend is responsible for collecting the data of the measurements. First half of the backend cooperates with the instrumentation and stores the results into intermediate variables.

The latter half of the backend is run at the end of the parallel section when all the work of the section is done and the runtime is uninitialising the team of threads. Embedding the latter half of the backend in this location was possible due to the workload using only a single parallel block and doing all the actual work in that parallel section.

5.2.3 Monitoring Techniques

Measuring the duration of the whole execution in real time is done with high resolution clock provided by C++11 standard library [3]. In Linux environment this gives the current time in nanoseconds. The time is measured before and after the workload, which gives us fairly accurate measurement of the total time used for the workload.

PAPI [15] is used for reading the values of performance monitor counters. PAPI supports a long list of performance counters. PAPI library was used by initialising the library and setting each thread to measure a set of counters, running the workload in parallel and finally reading the resulting counter values and summing them together. However, performance monitor counters ended up with only minor role on the final measurements.

5.2.4 Implementation

Finding the locations for instrumenting had three different approaches: start from `task.c`, start from assembly conversion of the workload and start from the implementation fo the OpenMP parallel pragma. `task.c` is the most relevant file for task scheduling in GOMP, which makes it a good start for finding where to place instrumentation code and understanding the task scheduling. `config/linux/bar.c` contains code for barrier handling, which might not be obvious when reading `task.c`, but is related to the task scheduling, as the barrier is one location for executing remaining tasks.

Starting from the assembly conversion provides easily all the entry points from the workload to the runtime by searching for GOMP functions. Finally, starting from the implementation of the parallel pragma provides a view in the order of execution and how the structures are initialised.

5.3 Runtime Instrumentation

This section concentrates on the instrumentation part of the structure described in the Section 5.2. Multiple versions of the instrumentation was created during the work. Main focuses of the versions were: dependency graph collection, task lock performance and time spent in runtime. In addition, the task lock performance measurements took three different approaches to the measuring: basic counting of lock usage, measuring instructions spent in the lock and measuring time spent in the lock. However, measuring instructions was noticed to be inefficient and less interesting than the other two approaches and was not followed further.

5.3.1 Instrumenting for Dependency Graph

Task creation was instrumented to collect and output a graph of all tasks in the workload and the relationships of those tasks.

A dependency graph of an execution was acquired with a modified libgomp. The modifications assign each task an unique id and stores to an array the task id, its start time, the parent task, thread that executed the task and the memory addresses the task depends on. At the end of execution, this array is written out for processing and visualising. Taskwaits were not visualised in the result.

5.3.2 Instrumenting the Task Lock

The function responsible for locking the task lock was found from `mutex.h` in `config/linux/` directory and the related function for waiting in the lock from `mutex.c` from the same directory. After locating these functions, they were instrumented to count the number of times the lock is locked, number of times we need to wait for the lock and the amount of time we spend waiting.

The naive way of instrumenting the lock by incrementing a shared counter with an atomic incrementation provided by GCC was noticed to be highly inefficient. After the incrementation was done to a per thread variable and summed together at barrier, the overhead was lowered to a reasonable level. Similarly, measuring the waiting time in the lock was first done by reading spent instructions from a performance counter, but the overhead was too high. Changing the time measurement to be done in wall clock time with `PAPI_get_real_cyc` again lowered the overhead to reasonable levels.

5.3.3 Instrumenting Runtime Entry and Exit

Entry and exit points of the runtime were instrumented to measure the time spent in the runtime and time spent in the workload. Entry and exit points relevant for the experiment were found by reading the runtime source and assembly conversion of the workload. The entry points can be divided into two types: calling runtime function and returning from call to task payload. The exit points likewise divide into returning from the call to runtime function and calling the task payload.

Entry points to the runtime were instrumented by taking current time with `PAPI_get_real_cyc` and counting the preceding time towards time in user code. The exit points were instrumented similarly, but the preceding time was counted for time in the runtime. The code for instrumentation was placed right at the beginning and right before the return from the called runtime function and immediately around the call to the task payload. The values were first stored per thread and summarised on a barrier, like was done with the task lock instrumentation.

Chapter 6

OpenMP Test Workload

The experiment of the thesis aims to measure the overhead caused by mutual exclusion towards the task management structures in GOMP. Managing these structures is the main part of scheduling tasks.

The measurement was done by instrumenting the runtime, implementing a workload using OpenMP tasks and running the workload with the said instrumented runtime.

AES (Advanced Encryption Standard) is a symmetric block cipher based on Rijndael algorithm. AES repeats rounds of operations on a block of 128 bits to encrypt the data. An encryption key can be 128, 192 or 256 bits. Though, the implementation for the thesis only accepts 128 bit key. To encrypt more data than just one block, AES needs to be used with some block cipher mode, which defines how a stream of blocks can be encrypted. Section 6.2 describes AES and some block cipher modes in more detail.

If the goal of the workload would not be to create large amount of dependent tasks, the AES encryption could be done more efficiently on modern computer with a processor instruction made especially for AES encryption [21].

6.1 Overview

The experiment was run on an x86 computer with 10 gigabytes of memory and Intel Core i7 970 with 6 cores at 3.20 GHz. The processor has hyper-threading, so each core has two hardware threads. There is 32 kilobytes of both L1 data and L1 instruction cache and 256 kilobytes of L2 cache.

Software environment for running the experiment consisted of Ubuntu Linux version 12.04 with Linux kernel version 3.2.0 and without starting any window managers or desktop environments. Instrumented GOMP was built from GCC release 4.9 found from GCC repository [5]. The workload was compiled with GCC 4.9.2 with no optimisations enabled. PAPI version 5.3.2 was used for measurement in the instrumentation.

AES in CBC (Cipher-Block Chaining) mode with fragmentation and message authentication code was implemented for the experiment. This corresponds to one of the options in TLS (Transport Layer Security) [19] for encrypting data. Some details are different, but the operations are the same on high abstraction level.

The AES rounds were rearranged by moving the key adding step from the end of a round to the beginning of the next round. This allows us to wait for the dependency right when we need it. The OpenMP directives for parallel computation was added to convert the rounds into tasks in two different ways, which are described in more detail further in this section.

Monitoring the execution was done with PAPI [15] and C++11 high performance clock. PAPI offers measuring of performance counters. Separate of the performance monitoring, a graph of task dependencies was created by modifying libgomp to store the dependencies of an execution.

Loosely following on the functioning of TLS, an AES implementation was created with CBC mode and fragmentation of message and calculating message authentication code was added around the AES implementation. Message authentication was done using a library call, because the rest already provided chances of parallelisation and hashing the message for the message authentication code would not parallelise well. The message authentication code was included, because it provided one large task as a whole. The rest was self-implemented to be able to add parallelisation with OpenMP to them.

The nature of CBC prevents effective parallelisation, but most of the CBC can be calculated parallel with message authentication code, which will be added to the end of the message and encrypted at the end of CBC. The message authentication code will be in the last two blocks of the message. Thus, the last two blocks depend on both the message authentication code and earlier part of CBC.

Fragmentation provides easy source of flat parallelism, as the encryption of the fragments don't depend on each other. To compare the effectiveness, the workload was compiled with and without the parallelisation of fragments. With parallelisation each fragment created its own task and without parallelisation there was a taskwait after each fragment, so that all created tasks were completed before starting the next fragment.

Two parallelisations of the AES itself was tried. The first parallelisation had each round of encryption and each round of key expansion be their own tasks resulting in 21 tasks per block encrypted with AES. The second parallelisation tried to reduce the amount of tasks by grouping the rounds to tasks.

The first encryption task had 1 round, the second 3 and the last 7. Similarly the two key expansion tasks had 3 and 7 rounds. This resulted in 5 tasks per encrypted block. The amounts of rounds per tasks were loosely measured so that the key expansion should be slightly faster than the encryption task that is executed at the same time. The second parallelisation was used in the experiment to keep the amount of tasks slightly more reasonable level.

6.2 Advanced Encryption Standard

AES (Advanced Encryption Standard) is a symmetric block cipher based on Rijndael algorithm, which won a competition by National Institute of Standards and Technology (NIST) in 2001 [1].

AES has three different key sizes: 128 bits, 192 bits and 256 bits. The key size will determine how many rounds of the algorithm will be run. The number of rounds is 10, 12 or 14 respectively for the key sizes. The key will be expanded and each round will use 128 bits from the expanded key.

One round contains four steps: S-Box, Shift Rows, Mix Columns and Add Round Key steps. In addition, there is an initial round with only Add Round Key step and a final round without Mix Columns step. However, for parallelising, the Add Round Key should be at the beginning of the round, so that we'll wait when we need the key we are depending on instead of waiting at the beginning of the original round structure. So, the round borders were shifted by one step resulting in empty initial task and final task with two Add Round Key steps.

A block cipher by itself is just for encrypting a single block, for example 128 bits of data. To do more than that, some mode of operation needs to be used with the block cipher, such as ECB, CBC or CTR, which are explained shortly below.

6.2.1 Block Cipher by Itself, ECB Mode

Just using the block cipher separately for each block of the message would be exactly what is done in ECB (Electronic Codebook). This approach causes security issues. For example, encrypting a bitmap image in ECB mode will result in original shapes showing clearly in the encrypted data.

6.2.2 CBC Mode

CBC (Cipher-Block Chaining) mode takes an exclusive or from the first block and an initialisation vector, which can be random bits, and encrypts that with the block cipher to get the first block of the encrypted data. All successive blocks after the first block will depend on the previous block, as an exclusive or will be taken from the new block of plain text and the encrypted previous block.

After encryption, both the encrypted message and the initialisation vector will be sent to the recipient and together with the key, the original message can be acquired.

CBC mode doesn't parallelise that well, but together with hash-based message authentication code offers a bit of parallelism and a bit of interesting dependencies between tasks.

6.2.3 CTR Mode

CTR (Counter) mode relies on encrypting an ever increasing counter together with a static nonce and taking exclusive or with the result and the message to be sent. The nonce should not be reused for different messages and the counter should be long enough to not overflow. This ensures that no two blocks given to the block cipher are the same and that the result is unpredictable and seemingly random.

The functionality of CTR mode resembles that of a one-time-pad, but the pad in CTR mode is generated from the relatively short key, which gives it an obvious theoretical weakness when compared to one-time-pad, but that is true for any algorithm besides the one-time-pad.

CTR mode would allow every block to be encrypted in parallel, but as the thesis is also interested in dependencies between tasks, CBC mode is more suited for the purpose.

6.2.4 Fragmentation

TLS protocol is defined to fragment messages which are too long into smaller pieces and encrypt and send them separately [19]. As the fragments are handled separately, the fragmentation allows for large amounts of parallelisation. Although, without dependencies between the tasks. One fragment can contain at most 2^{14} bytes of data according to RFC5246 [19].

6.2.5 HMAC-SHA1

HMAC-SHA1 is a hash based message authentication code, which is calculated with SHA1 algorithm. The message is hashed with the cryptographic hashing algorithm and the resulting hash is appended to the end of the message. After the MAC (Message Authentication Code) is appended, the message is encrypted and sent. However, as the MAC is only needed at the end of the encryption, most of the message can be encrypted in parallel with the HMAC calculation [18].

6.2.6 Parallelism

On the highest level is the fragmentation, which provides almost arbitrarily high parallelisation for large messages. If the TLS specification for fragment size is ignored, the fragment size can be used to control the amount of flat parallelism in the workload.

The next level of parallelism is provided by the combination of CBC and HMAC, in this case HMAC-SHA1, but the hash algorithm has no effect. As mentioned in earlier section, most of CBC and HMAC can be calculated in parallel and only the end of the CBC requires the results of both of them. If we make a rough guess that HMAC takes 50% of the time that CBC takes, this level of parallelism would offer work for 1.5 threads and couple simple dependencies. In reality, the HMAC takes less time than estimated here.

Parallelism between AES rounds and key expansion would be next in the scale. Each round depends on the previous round and the round key for that round. Round key for a round only depends on the round key of the previous round. If we make another guess that the key expansion takes 50% of the time that the rounds take, this offers again work for 1.5 threads, but this time there are lots of dependencies.

For finer granularity of tasks, AES rounds could be further partitioned to tasks in the S-Box and Mix Columns steps [24], which would allow fork-join kind of parallelism to four threads, but without hardware implementation, the workload would very likely be too fine grained to have any benefit.

Chapter 7

Results and Discussion

The plots in this chapter show the performance of three variants of parallelisation and that of the serial execution. The AES only parallel variant (AES o.p. in the plots) has AES block encryption parallelised, but waits after each fragment to complete before starting the next fragment. This variant has only little room for parallel execution and with more than couple threads, the threads are waiting for tasks most of the time.

The fully parallel variant (fully p. in the plots) has fragments parallelised in addition to the AES block encryption. This results in a mix of dependencies between tasks due to the AES blocks being parallelised and large number of potential for parallelisation due to the possibility to calculate the fragments in parallel. Both, this and the AES only parallel variant, have the problem of there being roughly 16 million very light tasks to execute for the 50 megabytes of data to encrypt that was used as load for the performance measurements.

The fragment only parallel variant (frag.o.p. in the plots) has only the fragments parallelised and the contents of one fragment are executed in serial by the thread that picked that fragment. This avoids the problem of having too many tasks and retains the benefit of potential parallelisation through the fragments, but does not have any dependencies and does not represent what we wanted to test. However, this variant gives perspective of how the runtime behaves with more favourable properties of tasks.

7.1 Dependency Graph

Dependency graph shown in Figure 7.1 was derived by instrumenting libgomp for the purpose and running the workload with the instrumented runtime. The figure shows only an excerpt from the end of the full recorded graph

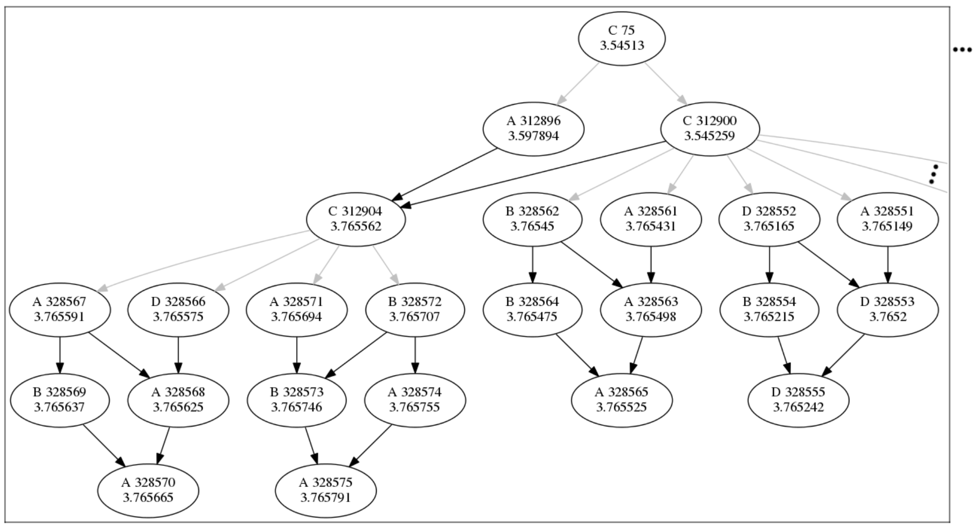

Figure 7.1: Excerpt of a dependency graph of the workload.

due to size constraints. The recording of the dependency graph was highly disruptive for the performance of the runtime and the instrumentation for the dependency graph was thus kept separate from the performance measuring instrumentations. The dependency graph recording also encrypted only one megabyte of data whereas the performance measuring worked on 50 megabytes. This was to reduce the size of the resulting dependency graph. The resulting dependency graph should be similar in form to the one with larger workload data. The only difference with the dependency graphs of smaller and larger input data is the number of dependency trees representing fragments in the encryption.

Each node in the graph represents a task. The letter in the node tells which thread executed the task in question and range from A to D, as the measurement was done with four threads. The number after the letter is an incrementing id for the task. Each task was assigned unique incrementing id. The task at the bottom left is the last task of the execution, which tells us that the encryption of one megabyte used 328575 tasks. The last number in the node is the time of when the task was started in seconds relative to the start of the program.

Edges in the dependency graph show the dependencies between different tasks. Black arrow between two nodes means that the pointed task depends on the pointing task because of an OpenMP depend clause. Gray arrow between two nodes means that the pointing task is a parent task to the pointed task.

The execution has a taskwait at the end of each five task blocks branching from the four CBC related tasks, due to the next block in CBC requiring the result of the previous one. This restricts the parallelism of the workload. However, because of fragmentation, there are many more of these trees one of which is partially pictured in the Figure 7.1.

The topmost task with id 75 is a task generated to encrypt one fragment. The left one of its children starts to calculate a hash for the message. The hash calculation is not parallelised in any way, as it seemed to be short in relation to the encryption and not necessary for the topic of this thesis. The right one, with id 312900, starts to encrypt most of the load of the fragment. There are 128 blocks with five tasks under this task, most of which are not pictured in the excerpt. After both of the children of the top task have finished, the task 312904, that depends on both of them, can be started and it will encrypt the two blocks of hash that was generated to the end of the message.

There might be bias in the excerpt concerning how often the threads switch between tasks of the same tree, because the part shown in the excerpt is at the end of the execution and all the other work is likely already done.

7.2 Challenges with Runtime Instrumentation

Figure 7.2 shows the execution time of different attempts for instrumenting the runtime for measurements with uninstrumented execution and serial execution for comparison. Parallel 1 in the plot is an attempt to instrument the task lock by reading performance counter each time it is locked. Parallel 2 in the plot is instrumentation using `PAPI_get_real_cyc()`. The measurements are from the fully parallel variant of the workload.

The parallel 1 line in Figure 7.2 shows an execution with reading the count of executed instructions on both sides of locking the GOMP task lock. Overhead is clearly unreasonable with anything more than two threads and the resulting graph doesn't resemble that of the uninstrumented execution.

The parallel 2 line shows an execution with a different approach. This version uses `PAPI_get_real_cyc()` to measure time in the lock instead of using a performance counter for the job. This also has the benefit of counting time spent in the lock instead of counting instructions used, thus getting the overhead even if the threads are not doing anything while waiting.

Getting the locking counts also had a similar performance issue for using a sequentially consistent atomic access to a global variable for storing the count. This was solved by storing the values first to the current thread and summing the results in barriers. However, the performance issue with this only showed on a development machine and not on the machine where final

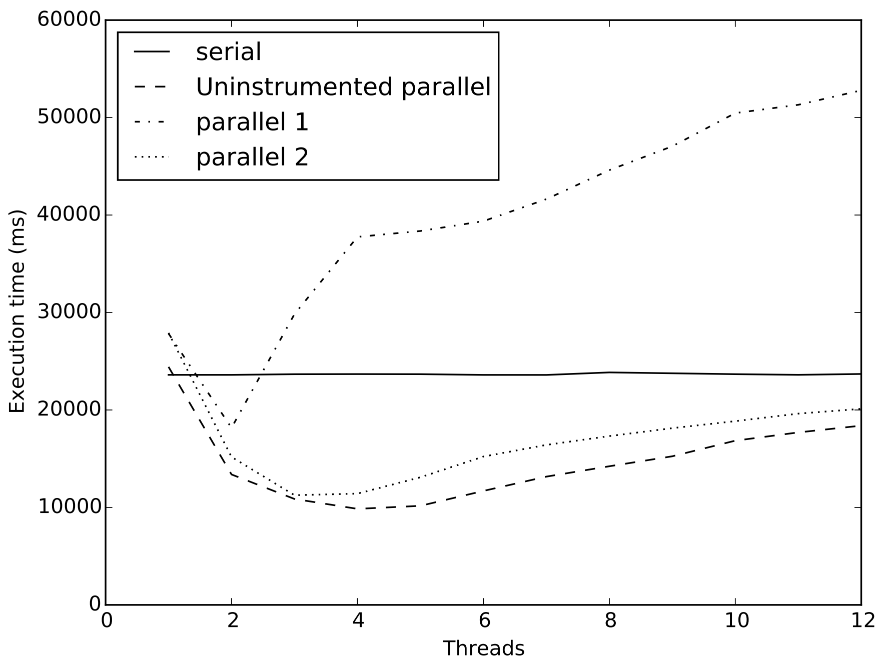

Figure 7.2: Execution time of two different attempts of instrumentation with uninstrumented and serial executions for reference.

measurements were taken.

The main difficulty of measuring the task lock is the huge amount of tasks in the execution of the chosen workload. At best, the runtime executes the 16 million tasks of the workload in roughly 10 seconds.

It would have been possible to further improve the instrumentation performance by implementing some kind of sampling and only take measurements from part of the times the lock is requested. However, the overhead was already in reasonable level and this was not deemed necessary.

7.3 Speedup of the Workload

Speedup of the workload was measured by measuring the time around the only OpenMP parallel section in the workload which does the fragmentation and encryption. Allocating the memory and preparing data to encrypt was done before starting the measurement. Average measurement from the serial version of the workload was then divided with a time measurement to acquire the speedup. Figure 7.3 shows the speedup for different variations of the

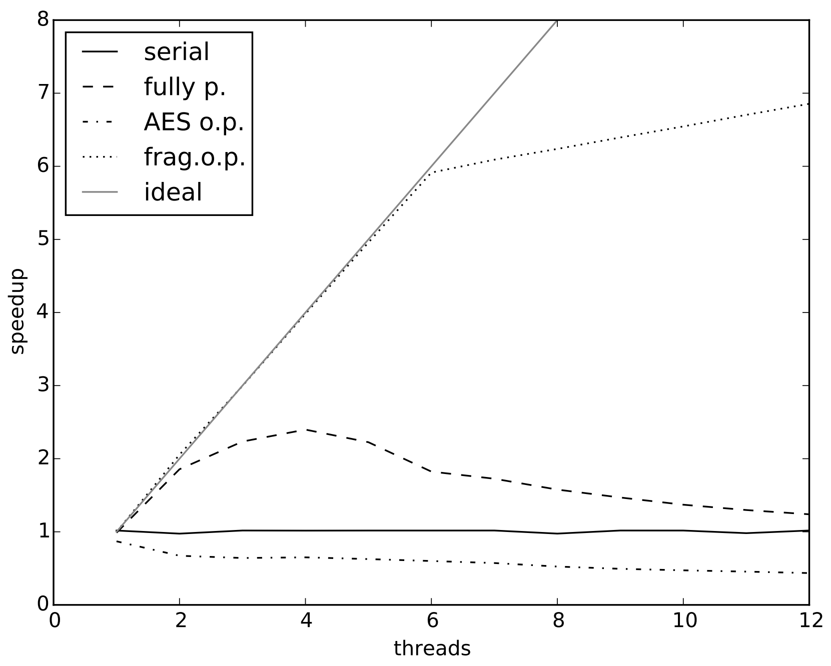

Figure 7.3: Speedup of the different variants of the workload.

workload and different number of threads.

As seen in Figure 7.3, the fully parallel variant of the workload speeds up to four threads after which the performance starts to deteriorate. The peak performance for the fully parallel variant of the workload is roughly double that of the serial version. The deterioration is likely due to the high amount of tasks generated during the execution, which is roughly 16 million tasks during 10 seconds of execution at the performance peak of the fully parallel variant of the workload.

The fragment parallel variant of the workload scales really well up to 6 threads after which the performance gain lessens creating an angle instead of smooth deterioration. The point with six threads coincides with the amount of actual cores in the measurement machine. More than 6 threads are achieved with hyper threading, which allows the number of hardware threads to be up to double the number of actual cores. Fragment size was 16 kilobytes, as mentioned in RFC about TLS [19], which was coincidentally half of the L1 cache, so that two fragments would not fit into the L1 cache at the same time, but reducing or increasing the fragment size did not affect the forming of the angle. Likewise, measuring L1 cache misses showed no unusual behaviour around six threads.

The performance of AES only parallel variant stays below the serial, keeps its performance about the same for two to four threads and starts to deteriorate more when the number of threads is further increased. As the AES only parallel variant has shortage of tasks and too many threads, the threads need to compete for the tasks causing increasing amount of overhead.

7.4 Task Lock Congestion

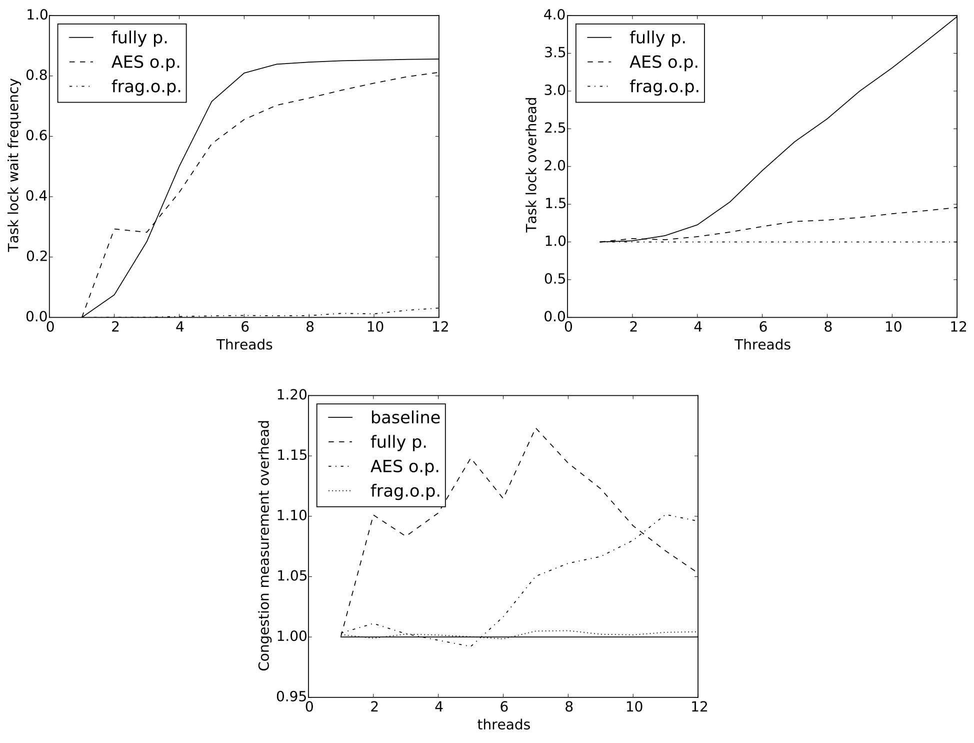

Figure 7.4: Task lock wait frequency, task lock wait time and the overhead from measuring these.

Task lock congestion was measured by instrumenting a lock function used to lock the task lock. A count for the number of lockings was increased before locking the lock and a count for the number of longer waits was increased if the lock was not acquired immediately. These counts were first increased to per thread counter and syncronised to a global counter when encountering a barrier, which there was only couple in the workload implementation. The time for waiting in the lock was measured with `PAPI_get_real_cyc()`, a

PAPI function for getting current real time in clock cycles. The time was measured before and after the wait and the difference stored. Time spent outside of the lock wait was gained by taking the difference of the start wait measurement of the current lock wait and end wait measurement of the previous time the thread was waiting for the lock.

The Figure 7.4 shows measurement results from measuring the congestion of the task lock. The first image of the figure shows the frequency of how often threads need to wait for the lock instead of acquiring the lock immediately when needed. The fully parallel variant can be seen to rapidly increase to roughly 0.8, which means that 80% of the time when the lock is needed, some other thread has the lock locked and the current thread needs to wait to acquire the lock. The AES only parallel variant increases slightly slower, which could be partly because of the lower tasks per second rate. Finally, the fragment only parallel variant stays close to zero, as most of the time is spent in the task workload. Slight increase to the wait frequency can be seen at the end side of the scale.

The second image of the Figure 7.4 shows the overhead from the time spent waiting for the lock. The overhead is plotted so that value 1.0 has no overhead and anything above that is extra time used. The plot shows that the overhead from waiting for the task lock in fully parallel variant increases slower at the beginning and starts to increase steeper at around 4 threads. At 4 threads the overhead is 23% and at 6 threads it is 95%. The AES only parallel variant waits for the task lock considerably less and is still only 46% with 12 threads. However, this is likely due to the lower task throughput of the AES only parallel variant and the threads are spending their time in some other overhead. Most likely they are sleeping in a condition variable for new tasks to appear. The fragment only parallel variant is again close to the optimal value due to having larger tasks and thus less frequent locking for the task lock. Time spent for waiting the lock in this variant is around 8 millionths for a measurement with 12 threads.

The third image of the Figure 7.4 shows the disruptiveness of the measurement by showing the overhead caused by the measurement itself. This was done by comparing the execution times of instrumented and non-instrumented versions of the runtime. Scale shows the execution time relative to the execution time of the non-instrumented version. The image shows that the fully parallel variant of the workload has roughly 15% overhead from the measurement and a bit less with small or large thread counts. The AES only parallel variant has no noticeable overhead from the measurement until 6 threads from where the overhead from the measurement starts to increase to the 10% overhead at 12 threads. The measurements for the fragment only parallel variant shows overhead from the measurement to be close to the baseline, which means there was next to no interference from the measurement for the fragment only variant. The interference for the other variants is noticeable, but still on a reasonable level.

7.5 Time Spent in Runtime

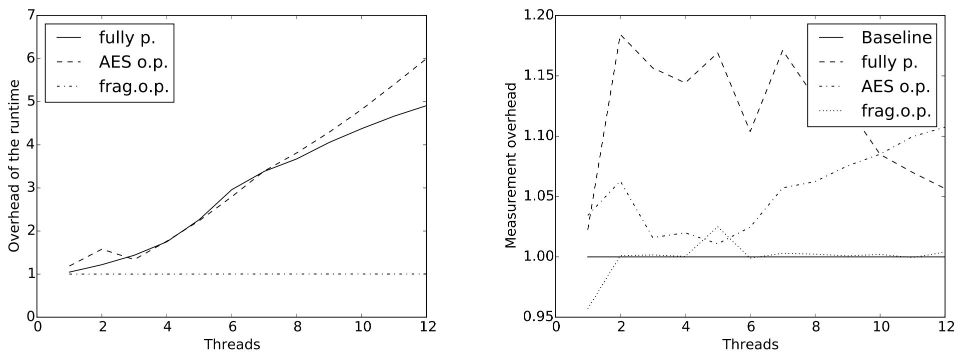

Figure 7.5: Overhead from the time spent in runtime code.

Time spent in the runtime and in the user code of the workload was measured by instrumenting locations in the runtime code where the execution either leaves or enters the runtime code. When the execution leaves the runtime code, the time since last measurement is added to a variable holding the total runtime time. Likewise, when the execution enters the runtime code, the time is added to total usercode time. The time measurement is done with the `PAPI_get_real_cyc()` function like with the measurement with the lock. Also similarly, the variables are stored first per thread and summed together on barriers.

Assembler conversion of the workload was used to find the locations for instrumenting the runtime. As the OpenMP pragmas have relatively simple conversion into code and include a function call to the runtime, the entry points were easily tracked by finding those calls to runtime functions. Such functions would then have an entry point at the beginning and exit point at thee end. The rest entry and exit points in the workload in question were executing task workloads. These were found by reading `task.c` from GOMP source. Task function call has an exit point before the call and an entry point after the call is complete.

In Figure 7.5 is pictured overhead from the time spent in the runtime and overhead from measuring the runtime and usercode times. The first image of the figure shows the overhead from time spent in the runtime. The fully parallel variant and the AES only parallel variant both behave in a similar manner. Both of them increase steadily and with five threads use over half of the time in the runtime. If this graph is compared to the results in Section 7.4, we notice that most of the time in runtime for the fully parallel variant is spent waiting for the task lock. AES only variant on the other hand can be explained by the threads waiting for available tasks. Fragment only variant of the workload stays close to the baseline meaning almost all time was spent in the usercode.

In the second image of Figure 7.5 we can see the overhead from the measurement. The overhead of this measurement is on similar level to the overhead of the task lock measurement. So, the overhead is around 15% for the fully parallel variant of the workload, increases to 10% on 12 threads for the AES only parallel variant and stays close to the baseline for the fragment only parallel variant.

7.6 Discussion

This section talks about what future research could provide interesting results. These ideas are basically: comparing time spent on dependencies and time spent on queues, modifying the task scheduling, experimenting with work stealing and dependencies, modifying the GOMP taskwait behaviour and creating dedicated scheduling thread.

Measuring time spent on dependency tree and time on task queues could tell which of them is more responsible for the task lock congestion.

Another possibility for future work would be modifying the task scheduling and to see how that affects the performance of the runtime. Modifying thread scheduling could have performance effect as well, but the thread scheduling is done in the kernel of the operating system instead of the OpenMP runtime and is not as such related to the task scheduling.

Creating a queue of tasks for each thread and implementing work stealing between the queues could be done and is done in some other runtimes. Figuring out how to parallelise the task dependency handling and how well it would work with work stealing could yield interesting results. The result of such modification could then be compared to the performance of the current behaviour.

Comparing tied and untied tasks would require implementation of the untied tasks. Experimenting with taskyield would also make more sense with untied tasks implemented. However, the GOMP habit of only taking a direct children for executing in taskwait could be experimented with and figured if it can affect any real world application. A specifically built test program that performs work between creation of tasks and the taskwait will show this effect.

As the task lock gets heavily congested with small tasks and high number of threads, a thread dedicated for scheduling could ease the burden on the task lock or remove the need altogether. The idea may deviate somewhat from OpenMP specification and makes sense only for task parallelism, but if the scheduling won't take too much time or can partly be delegated to others, it could improve the performance with small tasks. However, with small tasks the danger is that the scheduling does take too much time.

Chapter 8

Conclusion

The thesis focuses on the performance properties of the OpenMP task scheduling model. The original contribution of the thesis is in designing and evaluating an embedded monitoring system for an OpenMP runtime.

We measured performance qualities of OpenMP task scheduling model by developing an embedded monitoring system in GOMP, which is described in more detail in Chapter 5. The monitoring system was tested and evaluated by implementing a parallel cryptographic algorithm with couple variants, which is explained in Chapter 6. The main variant of the workload has really fine grained tasks, but still gained some speedup from the parallelisation.

Overhead caused by the monitor was found to be on a reasonably low level. The overhead from the measurement was around 15% and didn't show signs of growing with more threads. This makes the monitoring system we developed to be feasible for measuring the runtime performance. This finding is made in two different variations of the embedded monitor in Chapter 7.

Task lock that GOMP uses to protect task related data structures was found to be heavily congested when the individual tasks are tiny. In our measurements, the runtime spent roughly half the time waiting for the task lock when the amount of threads was increased to six for the main variant of the workload. Task lock congestion is further explained in Section 7.4.

One approach for continuing the research would be modifying the embedded monitoring system to measure in which parts of the runtime the time is spent while holding the task lock. First interest on this approach could be comparing time spent for managing dependency structures and time spent for managing task queues.

Another approach would be to modify the runtime. Some discussion of possible modifications can be found in Section 7.6. Additionally, using a lockless data structures could be possible, but would require deep understanding of how to implement such data structures.

The contribution of the thesis, designing and evaluating an embedded monitoring system for an OpenMP runtime, is significant for future research, as it gives a base to build up on and shows how experimental changes to the runtime can be evaluated and compared.

Bibliography

[1] Advanced encryption standard. URL http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[2] Intel Cilk Plus. URL https://software.intel.com/en-us/intel-cilk-plus.

[3] High resolution clock. URL http://en.cppreference.com/w/cpp/chrono/high_resolution_clock.

[4] Installing GCC, . URL https://gcc.gnu.org/install/.

[5] GCC: Anonymous read-only SVN access, . URL https://gcc.gnu.org/svn.html.

[6] GDB: The GNU project debugger. URL http://www.gnu.org/software/gdb/.

[7] GNU gprof. URL https://sourceware.org/binutils/docs/gprof/.

[8] Intel OpenMP runtime library. URL https://www.openmprtl.org/.

[9] An introduction to kprobes. URL http://lwn.net/Articles/132196/.

[10] LTTng. URL http://lttng.org/features/.

[11] The multicore association. URL http://www.multicore-association.org/.

[12] ompP / MADAME. URL http://www.ompp-tool.com/.

[13] OpenEM.org, . URL https://www.openem.org/.

[14] OpenMP frequently asked questions, . URL http://openmp.org/openmp-faq.html.

[15] Performance application programming interface. URL http://icl.cs.utk.edu/papi/.

[16] perf: Linux profiling with performance counters. URL https://perf.wiki.kernel.org/index.php/Main_Page.

[17] Thread Building Blocks. URL https://www.threadingbuildingblocks.org/.

[18] Enhancing security performance with parallel crypto operations in SSL bulk data transfer phase, 2007. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4448620&tag=1.

[19] The transport layer security (TLS) protocol version 1.2, 2008. URL http://www.ietf.org/rfc/rfc5246.txt.

[20] OpenMP Application Program Interface - version 3.0, May 2008. URL http://www.openmp.org/mp-documents/spec30.pdf.

[21] Intel Advanced Encryption Standard (AES) Instruction Set - Rev 3.01, August 2012. URL https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set.

[22] OpenMP 4.0 specification released, July 2013. URL http://openmp.org/wp/2013/07/openmp-40/.

[23] OpenMP application program interface - version 4.0, July 2013. URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[24] Parallel AES encryption with modified mix-columns for many core processor arrays, 2014. URL http://www.ijesit.com/Volume%203/Issue%203/IJESIT201403_23.pdf.

[25] GCC 4.9 release series, July 2014. URL https://gcc.gnu.org/gcc-4.9/.

[26] GCC wiki / OpenMP, July 2014. URL https://gcc.gnu.org/wiki/openmp.

[27] GNU libgomp documentation, 2014. URL https://gcc.gnu.org/onlinedocs/libgomp/index.html.

[28] Intel's "Knights Landing" Xeon Phi Coprocessor Detailed, June 2014. URL http://www.anandtech.com/show/8217/intels-knights-landing-coprocessor-detailed.

[29] OpenMP / Clang, September 2014. URL `http://clang-omp.github.io/`.

[30] Gregory R Andrews. *Concurrent programming: principles and practice.* Benjamin/Cummings Publishing Company, 1991.

[31] Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. A survey of high-level parallel programming models. Technical report, Technical Report HW-MACS-TR-0103, Heriot-Watt University, 2013.

[32] Mordechai Ben-Ari. *Principles of concurrent and distributed programming.* Pearson Education, 2006.

[33] Lawrence Livermore National Laboratory Blaise Barney. POSIX Threads Programming. URL `https://computing.llnl.gov/tutorials/pthreads/`.

[34] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP task scheduling strategies. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 100–110, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-79560-X, 978-3-540-79560-5. doi: 10.1007/978-3-540-79561-2_9.

[35] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2012.

[36] Smith Jim Jr, Ravi Nair, James E Smith, and Heath Potter. Virtual machines: Versatile platforms for systems and processes, publisher morgan kaufmann publishers, may 2005. Technical report, ISBN 1-55860-910-5.

[37] Patrick Kennedy. GCC 4.9 OpenMP code cannot be linked with Intel OpenMP runtime, September 2014. URL `https://software.intel.com/en-us/articles/gcc-49-openmp-code-cannot-be-linked-with-intel-openmp-runtime`.

[38] Monica Lam, Ravi Sethi, JD Ullman, and AV Aho. Compilers: Principles, techniques, and tools, 2006.

[39] Yuan Lin, Guansong Zhang, et al. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009. doi: 10.1109/TPDS.2008.105.

[40] Ananya Muddukrishna. Exploiting locality in OpenMP task scheduling. page 76, 2010.

[41] Stephen L Olivier. Design issues in the semantics and scheduling of asynchronous tasks. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2013.

[42] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. OpenMP task scheduling strategies for multicore NUMA systems. *International Journal of High Performance Computing Applications*, page 1094342011434065, 2012. doi: 10.1177/1094342011434065.

[43] StephenL. Olivier and JanF. Prins. Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*, 38(5-6):341–360, 2010. ISSN 0885-7458. doi: 10.1007/s10766-010-0140-7.

[44] Artur Podobas. Thermal-aware scheduling in OpenMP. page 90, 2010.

[45] Antoniu Pop and Albert Cohen. Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, Vienna, Austria, July 2010. URL `https://hal.inria.fr/inria-00551518`.

[46] Jaspal Subhlok, James M Stichnoth, David R O'hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Notices*, volume 28, pages 13–22. ACM, 1993. doi: 10.1145/173284.155334.

[47] Barry Wilkinson and Michael Allen. *Parallel programming*, volume 999. Prentice hall New Jersey, 1999.