Enabling Moldability in OpenMP

Master's thesis in Computer science and engineering

Pontus Sundqvist
Simon Sundqvist

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Enabling Moldability in OpenMP

Pontus Sundqvist
Simon Sundqvist

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Enabling Moldability in OpenMP
Pontus Sundqvist, Simon Sundqvist

© Pontus Sundqvist, Simon Sundqvist, 2023.

Supervisor: Nikela Papadopoulou, Computer Science and Engineering
Examiner: Miquel Pericàs, Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Enabling Moldability in OpenMP
Pontus Sundqvist, Simon Sundqvist
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

OpenMP has long been a ubiquitous technology in High-Performance Computing (HPC), making parallel programs simple to reason about and portable to many different systems. When an OpenMP runtime decides which threads should run tasks, it often uses a simple work-stealing scheduler as they evenly distribute tasks among cores. This is the method used by LLVMs OpenMP runtime. But today, HPC systems often consist of multiple sockets, each with many cores and non-uniform memory access (NUMA). This creates a complicated memory hierarchy which isn't accounted for by simple work-stealing schedulers. Another feature not supported well by simple work-stealing schedulers is nested parallelism, where each task runs multiple threads in parallel. It isn't clear how many threads each task should be allocated i.e. the width of the task. If it's too high there will be over-subscription while if it's too low there will be load imbalance. This can be solved by supporting moldable tasks, which are tasks where the scheduler decides each task's width. We extend LLVM's OpenMP runtime with support for moldable tasks scheduled using a locality-aware scheduler.

Keywords: Adaptive Scheduling, OpenMP, NUMA-aware scheduling, Moldable

Acknowledgements

We want to thank Nikela Papadopoulou for supervising our project. This project would not have been possible without her discussions and guidance.

Pontus Sundqvist, Gothenburg, 2023-07-11

Simon Sundqvist, Gothenburg, 2023-07-11

Contents

List of Figures xi

1 Introduction 1

2 Background 3
- 2.1 LLVM . . . 3
- 2.2 OpenMP . . . 3
- 2.3 Affinity . . . 4
- 2.4 Locality and availability . . . 5
- 2.5 Scheduling of moldable tasks . . . 6
- 2.6 Previous work . . . 7
- 2.7 Alternatives to moldable tasks . . . 7
  - 2.7.1 Teams . . . 7
  - 2.7.2 Taskloop . . . 7
  - 2.7.3 Free-agent threads . . . 8

3 Implementation 9
- 3.1 Moldable task teams . . . 9
- 3.2 Front end . . . 10
- 3.3 Threading . . . 10
- 3.4 Execution flow . . . 11
- 3.5 Executing a moldable task . . . 11
- 3.6 Scheduling moldable tasks . . . 12
- 3.7 Work-stealing . . . 13

4 Results 15
- 4.1 Hardware . . . 15
- 4.2 SparseLU . . . 15
- 4.3 Synthetic benchmark . . . 18

5 Conclusion 21

Bibliography 23

A Appendix 1: Implementation Details I
- A.1 Front end . . . I

A.2 Back end I
A.2.1 Moldable task team hierarchy I
A.2.2 Steal order lists II
A.2.3 Work-stealing II
A.2.4 Executing a moldable task II
A.2.5 Scheduling moldable tasks III
A.2.6 Timetable III

List of Figures

3.1 Hierarchy of moldable task teams 10

4.1 SparseLU benchmark runtime 17
4.2 Speedup of SparseLU using moldable tasks 17
4.3 Execution time of synthetic benchmark 19

A.1 Example visualisation using `timetable.py` III

1 Introduction

Today most systems are multicore systems, which require programs to be parallel if they want to fully utilize the available resources. Many programs can be parallelized by dividing the program into tasks, small parts of the program which are scheduled at runtime on different threads and then executed in parallel. Each type of task will have different performance characteristics and may depend on the result of previous tasks.

A popular method to make parallel programs is by using the OpenMP API. OpenMP is a generic API that can be used in several different programming languages and there are several runtimes that implement its features. While originally focused on parallelizing loops, support for the creation of tasks was added in version 3.0.

Different OpenMP implementations can have different methods to schedule tasks, but a common one is work-stealing: each thread has a queue of tasks to execute and whenever a thread runs out of tasks to execute, it steals a task from another queue [1]. This method has the advantage of achieving high parallelism, as threads will always execute tasks if available, but there are some disadvantages.

One disadvantage is the need for the programmer to consider the granularity of tasks; i.e. how many tasks the program should be split into. One can have very fine granularity of tasks, where each task is very small and can be quickly executed. This would make it easy to achieve high parallelism as every thread will have a task to execute, but it has the downside of adding more overhead, as each task has to be scheduled, moved to a thread, and begin executing.

To reduce overhead, one may want to use a more coarse granularity of tasks, grouping many small tasks into larger tasks. For example, if we repeatedly called a function containing a loop that spawned many small tasks, we may instead decide to make the loop itself into a task. This would reduce time spent switching between tasks but can also create new problems. If each task has a high variability in its execution time then the “fast” threads will have to wait for the “slow” threads. This problem is called under-utilization.

One possible solution would be to allow the task scheduler to choose the task granularity. This would be possible if the tasks themselves were parallelizable. The scheduler would then be able to choose the number of threads to allocate to each task and in this way control the task granularity. A task that can be scheduled on different amounts of threads, i.e. different widths, is called a moldable task.

Another problem with work-stealing is that it assumes that other threads and the cores they are executed on are interchangeable when stealing tasks. This assumption is not true if the system has a complicated memory hierarchy, with Non-Uniform Memory Access (NUMA). On NUMA systems it can be useful to consider the locality of tasks when scheduling them. There already exists partial support for this in OpenMP through thread affinity (manually controlling thread placement) but there is no support for task affinity yet. Previous work related to this was done by Alexandersson & Nilsson who made a locality-aware scheduler for OpenMP tasks, optimized for energy usage on heterogeneous systems [2]. It seems natural to use a locality-aware scheduler for moldable tasks, as the threads used to execute a moldable task should run faster if they share caches with each other.

In this thesis, we have added moldable tasks to OpenMP which are scheduled using a locality-aware scheduler. We will explain some background related to OpenMP and related work in Chapter 2, then describe our implementation in Chapter 3. Results from benchmarks of our implementation are then presented in Chapter 4, which is followed by a conclusion and ideas for future work in Chapter 5.

2

Background

2.1 LLVM

LLVM is a collection of compiler technologies and APIs. We are interested in two parts of LLVM: (i) Clang, a C/C++ compiler with support for OpenMP and (ii) the LLVM OpenMP runtime, a dynamic library that implements OpenMP 4.5. When Clang compiles a C++ program, it recognizes OpenMP constructs using the syntax `#pragma omp <CONSTRUCT> <CLAUSES>`. The construct is then converted into various OpenMP runtime calls, which may depend on the surrounding context and various clauses. When the compiled program is executed, it calls functions exposed by the LLVM OpenMP runtime. This separation of the runtime into a dynamic library allows it to be used by compilers other than Clang, e.g. the Fortran compiler Flang.

2.2 OpenMP

OpenMP is an API used for shared memory multiprocessing, for C, C++ and Fortran. It features a wide range of constructs used in parallel programs and there are several independent runtime implementations of the API. This makes programs that use OpenMP portable and allows easy experimentation by changing the underlying mechanisms, such as different schedulers.

The OpenMP specification describes every feature of the API, but its flexibility can make it hard to describe what actually happens when running an OpenMP program. For example, there are several constructs that have the purpose of parallelizing a program but allow the runtime to execute it serially. To make things more concrete we will describe OpenMP constructs as they are implemented in the LLVM OpenMP runtime, ignoring other runtimes.

To give an overview of OpenMP we'll examine a simple example of an OpenMP program, presented in Listing 1. It features three different OpenMP constructs: `parallel`, `single` and `task`. The `parallel` construct on row 1 is a common start to an OpenMP program.

```
1  #pragma omp parallel
2  #pragma omp single
3  {
4      #pragma omp task
5      #pragma omp parallel
6      a();
7
8      #pragma omp task
9      #pragma omp parallel
10     b();
11 }
```

Listing 1: Subsection of a simple OpenMP program

A `parallel` construct creates a team of threads, equal to the number of threads on the machine. One of these threads will be the master thread, usually the thread which encountered the construct. It then uses these threads to execute the following block; all threads will execute the work-stealing algorithm described in Chapter 1. At least, that is what would happen if there wasn't a `single` construct on row 2, as it interrupts the execution of the threads in the team. Instead, only a single thread will execute the code contained in the block, while the rest of the threads will act as if they encountered a barrier.

When the single thread reaches the `task` at row 4, it will spawn a new task and place it in its own task queue, without executing it. Meanwhile, the other threads in the team will be busy work-stealing. When one of these threads inspects the single threads task queue, it will steal the task and execute it. The single thread will at the same time continue to the next task at row 8 and repeat the process.

For the tasks themselves, each one contains a `parallel` and then a function call to `a` or `b`, respectively. This is an example of nested parallelism, which is not supported by OpenMP by default but can be activated by setting the environment variable `OMP_MAX_ACTIVE_LEVELS` to one's preferred nesting level. The problem, in this case, is that the nested `parallel` regions would cause over-subscription as it would create double the number of threads as the number of logical cores on the system.

The number of threads could be reduced by adding a `num_threads` clause on row 5 and 9. This would limit the number of threads in the new team, but only statically. In a more complicated program, one may not know how many threads to allocate to each task. It may even change during the execution of the program. Moldable tasks would solve this as it would allow the scheduler to choose the number of threads used by each task when scheduling it.

2.3 Affinity

Thread affinity is the assignment of threads to specific logical processors. This is mostly done automatically by the operating system but a program can also specify

thread affinity manually. This can be done in OpenMP by setting the environment variable `OMP_PLACES`. Manually setting thread affinity can be useful if the processors of our system aren't interchangeable, such as when the system has two sockets with four cores each. Then our example in Listing 1 may perform better if we always scheduled `a` on one socket and `b` on the other. This could be achieved by setting `OMP_PLACES="{0,1,2,3},{4,5,6,7}"` and using the code in Listing 2.

```
#pragma omp parallel num_threads(2)
#pragma omp single
{
    #pragma omp task
    #pragma omp parallel num_threads(4) proc_bind(master)
    a();

    #pragma omp task
    #pragma omp parallel num_threads(4) proc_bind(master)
    b();
}
```

Listing 2: Improved nested parallelism

The outer parallel region will now only have two threads, which will be assigned to separate sockets. The use of `proc_bind(master)` on row 5 and 9 ensures that all 4 threads of the nested parallel region are assigned to the same place as the thread which started executing the task. This code manages to separate the tasks, but there are still some problems with it. For one, the different tasks will be scheduled on an arbitrary socket, without any care for the difference between the tasks. If one socket has much faster memory than the other, then we would prefer the more memory-intensive task to be scheduled there. This isn't possible to do with just thread affinity in OpenMP.

One solution would be to add task affinity to OpenMP as a variant of thread affinity. There has been research by Terboven et al. to implement this in OpenMP [3]. Without task affinity, they saw inconsistent performance of tasks, especially on NUMA architectures, and tried implementing various versions of task affinity to mitigate it. Their methods improved performance up to 40% and decreased runtime variation.

Another problem with the code in Listing 2 is the risk of underutilization caused by static scheduling. If one of the tasks is much faster, then the 4 threads allocated to it would go to waste. If one used a dynamic scheduler with moldable tasks then the imbalance could be detected automatically and mitigated.

2.4 Locality and availability

Today's multicore systems have a deeper memory hierarchy than just dividing the cores on two sockets. For example, a system could have logical processors with

multiple levels of shared caches, multiple NUMA nodes and multiple sockets. In that case, it would be a good idea to keep memory-intensive tasks that touch the same memory on logical processors that share some parts of the memory hierarchy. This is called locality-aware task scheduling. In the case of moldable tasks, this means that threads executing the same moldable task should be executing on logical processors with shared caches, the same NUMA node or the same socket. This is achieved by setting the affinity of threads to match parts of the memory hierarchy. For example, there could be one team of threads that covers each L3 cache and one for every NUMA node and so on. This ensures that threads executing a moldable task take locality into account.

One way to further take advantage of this deep memory hierarchy is by using the concept of availability. For a memory-intensive task, one might want to give it exclusive access to some cache so that memory accesses have lower latency. For this to work one would have to separate the number of threads used to execute a task and the number of logical processors reserved for execution. This has been explored by Miquel Pericàs in his work on Elastic Places [4].

2.5 Scheduling of moldable tasks

There has also been some theoretical work on scheduling moldable tasks. Because the number of tasks in OpenMP is unknown until the program is run it is classified as an online scheduling problem, also known as a dynamic scheduling problem. Furthermore, because splitting up tasks and task switching should be avoided for the best performance, only non-preemptive scheduling will be considered. There is also a distinction between moldable tasks where all threads start and stop the execution of the task synchronously at the same time, and moldable tasks where threads can start and stop execution asynchronously. These are called gang tasks and fork-join tasks respectively. In this thesis, we focus on gang tasks.

When scheduling tasks there are also different assumptions one can make regarding the execution time of tasks. One can assume that cores are either homogeneous or heterogeneous. In our case we want to handle NUMA architectures where the placement of tasks can affect the runtime so we assume heterogeneous cores. There are also different speedup models one can assume. A speedup model here refers to how the runtime of a task is assumed to change when the width of the task is changed. Some examples of speedup models are the roofline model [5], the communication model [6] and Amdahls model [7]. In our case, we will be assuming that the execution times with different widths are uncorrelated. This increases our flexibility in handling different types of tasks but increases the number of tasks needed before we have good estimates of task execution time.

Hikida et al. studied a setting very similar to ours with online scheduling of moldable gang tasks, though they assumed homogeneous cores while we make no such assumption and accept heterogeneous cores [8]. They tested different heuristics for scheduling moldable tasks and they also gave an excellent overview of related work. Benoit et al. created an algorithm for online task scheduling and derived competitive

ratios under different speedup models achieving state-of-the-art ratios [9].

2.6 Previous work

The main inspiration for this thesis is Abuljabbar et al's implementation of moldable tasks in the research runtime XiTAO [10]. Some key differences between their work and our work are that in the XiTAO runtime, they use an explicit construction of a DAG to describe the dependencies between tasks in a parallel program while OpenMP programs usually have a more implicit description of task dependencies so we do not have access to a DAG of dependencies. They also have information regarding the data locality of tasks which we do not.

2.7 Alternatives to moldable tasks

Some of the problems which we try to address using moldable tasks could be solved using other methods. We will list some of the advantages and disadvantages of these other methods but will not focus on them much more in the rest of the thesis.

2.7.1 Teams

The `teams` construct was added in OpenMP 4.0 to model a type of nested parallelism which is used on GPUs. While a `parallel` construct can be used to create a single team with one master thread, a `teams` construct is used to spawn multiple separate teams with their own master threads. If this code is offloaded to a GPU, each team will be executed in a way that accounts for the thread group hierarchy of GPUs. This allows one to efficiently distribute loops to a GPU but does not work if one wants to use tasks.

2.7.2 Taskloop

A `taskloop` construct is used to split a loop into tasks. The naive method to spawn tasks in a loop would be to either spawn a new task for every iteration or to treat the whole loop as a single task. Both of these have problems as they represent two extremes of task granularity. Using `taskloop` allows the scheduler to group iterations into larger tasks in a dynamic way. One can either use the `grainsize` clause to decide the number of iterations per task or the `num_tasks` clause to set the number of tasks created. In the LLVM OpenMP runtime, the default is to create 10 times as many tasks as the number of threads in the current team. This could be done in a more dynamic way, and could then be considered as an alternative to moldable tasks. This could work well but be less flexible than allowing the creation of arbitrary moldable tasks.

2.7.3 Free-agent threads

A limitation of moldable tasks is that they must decide the number of threads used for execution, i.e. their width, before they start executing. There exists a more general concept called malleable tasks, which can change their width during execution. This can remove the risk of imbalance that occurs when threads are waiting for other tasks to finish. There have been proposals to add malleable tasks to OpenMP by adding "free-agent" threads, which are threads not part of any specific team. Free-agent threads would instead always be available to add more threads to a team whenever there were unused resources on the system. This would mitigate load imbalance in a program but would have some limitations. It would for example not care about locality, as free-agent threads are free to jump between teams [11, 12].

3

Implementation

We have extended OpenMP with a built-in concept of moldable tasks. To handle them, the scheduler was modified to schedule moldable tasks according to the system's memory hierarchy, while the scheduling of normal tasks was mostly left unchanged. We have omitted specific code changes and some technical details, which can be found in Appendix A

To make a task moldable, the user only has to add `moldable` to a `task` construct, as seen in Listing 3.

```
1  #pragma omp parallel           // create threads.
2  #pragma omp single             // only let one thread create tasks.
3  {
4      #pragma omp task moldable // create and schedule a moldable task
5      #pragma omp parallel
6      a();
7
8      #pragma omp task moldable
9      #pragma omp parallel
10     b();
11 }                              // wait until threads are done executing tasks.
```

Listing 3: Using our implementation

Marking a task as moldable is similar to enclosing a region in a `team` construct, as any enclosed parallel region will be limited to running using the tasks team.

3.1 Moldable task teams

Normal tasks are executed by a single thread, but moldable tasks are executed using so-called moldable task teams. These teams are created automatically and represent groups of logical processors which share some part of the memory hierarchy. See Figure 3.1 for an example. Each moldable task team consist of a task queue and a mask which specifies the logical processors to use when executing a task using that team. This design means that the scheduler doesn't have to decide the width or locality of a moldable task directly, but only which moldable task team to schedule the task to.

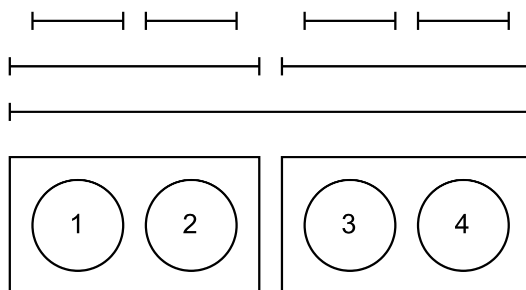


Figure 3.1: Diagram of a hierarchy of moldable task teams. Each circle is a core and each box is an L3 cache. Each moldable tasks team is drawn above the core(s) they cover.

Each moldable task team has a master thread assigned to it which can start using the team after it's done executing regular tasks. A thread can be the master thread for several moldable teams, and will then have multiple queues. We also added a configuration to allow for multiple master threads for each team, which is activated by default. Every thread can then act as a master thread for any moldable team in which its contained, which allows them to execute moldable tasks using that team.

3.2 Front end

The changes to Clang, the front end, were kept to a minimum. We only added the ability to add a `moldable` clause to `task` constructs, following the same logic used for other clauses, e.g. `untied`. At compile time, the front end converts all clauses of a task into a parameter which is then used at runtime when a thread encounters the `task` construct and spawns a new task.

We could have implemented moldable tasks without changing the front end by treating all tasks as moldable tasks. As moldable tasks are a superset of normal tasks, there would theoretically be no loss of functionality. This would have the advantage of allowing our runtime to be used with other front ends without any work and allow us to simplify our implementation as we wouldn't need to handle two different types of tasks. This sounds good in theory, but there are many reasons one would want to keep normal tasks. Implementation-wise, our moldable tasks aren't a superset of normal tasks as there are some features we don't support, e.g. untied moldable tasks. The user may also want to avoid the overhead caused by moldable tasks when they only need small normal tasks. During development, we also found it useful to compare moldable tasks and normal tasks in the same program to find bugs.

3.3 Threading

In a typical OpenMP program with normal tasks, a parallel construct is used to create enough threads to saturate the system. The number of threads can also be customized using a `num_threads` clause. The implementation of moldable tasks uses

these threads to create and execute moldable tasks exactly as it would for normal tasks. The difference from normal task execution comes when it executes a task with a width larger than 1 i.e. a parallel or non-serialized moldable task. Then it uses one or more worker threads to execute the task. Before execution of the moldable task starts the worker threads are taken from a free thread pool or, if not enough threads are available, new threads are created. These worker threads are different from the ones created for the initial parallel region. This means that about twice the number of threads are created when executing moldable tasks, which is why threads that overlap with a team of a currently executing task are suspended, and woken up when the task is finished.

This implementation is certainly unsatisfactory. It could be improved by having threads transition from the task execution loop to being a worker thread. This would remove the need to suspend and resume threads. Removing the need to take worker threads from the free threads pool would also avoid the need to take a global lock when starting and finishing a parallel moldable task.

3.4 Execution flow

All threads in a `parallel` construct follow the same execution flow, except when encountering a work-distribution construct, e.g. a `single` construct or a `for` construct. A detailed description of the execution flow can be found in the specification [13]. The parts we have added are the scheduling and execution of moldable tasks. When a moldable task is encountered, it is scheduled by choosing a moldable task team and pushing the task to the team's task queue.

A moldable task team will not start executing a moldable task from its queue if there is an overlapping team already executing a moldable task. The status of which logical processors are busy executing is encoded in a mask behind a global lock.

We have made no changes to how teams and threads are reused when executing moldable tasks so we rely on existing mechanisms in the runtime such as the free threads pool, free teams pool and hot teams pool which are used to reuse threads and data structures wherever possible.

For every moldable task we store an estimated execution time for every moldable task team it could be executed on. Moldable tasks are identified by the task's source location. These estimations are initialized to zero so that every type of task is run at least once on every team. These statistics are used for scheduling and updated using a running exponential average. The amount of smoothing of the exponential average can be tuned with the environment variable `KMP_MOLDABILITY_EXP_AVERAGE`.

3.5 Executing a moldable task

Before starting the execution it is assumed that the logical processors that will be used to run the moldable task have been set to true in the global mask. First,

the moldable task is set to “started” and the current task of the current thread is updated. Then the thread affinity is set, both on the system level and in the runtime where the mask is saved so that any worker threads can copy the mask later. After that, the routine of the moldable task is executed as the routine of a `teams` region, with the number of threads set to the width of the moldable task team. After the task has been executed, any threads which were suspended because of the execution will be woken up and have them cleared in the global mask so that they can execute tasks again. Finally, the moldable task is marked as completed and the state of the thread is restored.

3.6 Scheduling moldable tasks

When a thread reaches a task-generating construct, such as a normal `task` construct or a `taskloop` construct, it is called an encountering thread. Normal tasks are scheduled by pushing tasks to the queue of the encountering thread and then other threads use work-stealing to steal tasks from that queue.

For moldable tasks, we have chosen to let the encountering thread schedule a task by choosing which moldable task queue to push the task to. This is done by choosing the team which minimizes the following equation:

$$C(T,i) = \overbrace{R(T,i) * |T|}^{\text{Cost}} + \overbrace{s\left(\frac{1}{|T|}\sum_{t\in T} W(t) - W_{min}\right) + \epsilon}^{\text{Load Balancing}}$$

where the variables are

- T, the set of threads of a given team.
- i, the task.
- $R(T,i)$, the estimated runtime of the task for the team, calculated using an exponential running average of previous runtimes.
- s, a scaling term.
- $W(t)$, the estimated work scheduled to a given thread. We assume the work of a team is distributed equally among its threads.
- $W_{min} = \min_{t\in T} W(t)$, the minimum amount of work any thread has been scheduled.
- ϵ, a randomness term which causes us to schedule randomly when warming up, as all estimated costs are initialized to zero.

When scheduling tasks we are trying to reduce the runtime of our program. This is not the same as reducing the runtime of an individual task because that would mean using more cores, which could be used to execute other tasks. So when there are enough moldable tasks to saturate our system we would like to minimize cost. On the other hand when there are not enough tasks to saturate the system we would like to increase parallelism to minimize the runtime of a task.

We achieve this goal by minimizing cost when we are pushing new tasks to moldable task queues, and minimizing runtime when stealing tasks. This would in theory create 2 stages of task execution. One where we have enough tasks to saturate the system so we run tasks on smaller teams, and one where we are not saturated so we run tasks on larger teams. In practice though it is likely that the first few tasks will be stolen which leads to them being executed on larger teams than necessary.

3.7 Work-stealing

When executing normal tasks the victim to steal from is chosen randomly. This works well because it is likely that there is only one thread with tasks in its queue to steal from because normal tasks are always pushed to the queue of the encountering thread. When encountering moldable tasks they are sent to different queues taking the cost of execution and load balancing into account. This means that there will be many more queues with tasks to steal from than with normal tasks. This allows us to focus on stealing from threads that have similar performance characteristics when it comes to executing tasks. Therefore work-stealing of moldable tasks tries to steal from nearby threads first. A thread is considered closer to another thread if they share a smaller moldable task team.

One trade-off is that with normal tasks where most of the tasks are put in the same queue, one can keep taking tasks from the same queue without searching for a new victim. This is still done for moldable tasks but is not as effective because the tasks are spread out over more queues.

Just like work-stealing with normal tasks, work-stealing with moldable tasks is greedy so it will always execute a task if possible. It will try to execute moldable tasks on the team with the lowest estimated runtime as long as there are no overlapping teams currently executing tasks. This could most likely be improved by implementing a scheduler that allows some slack. This would allow threads to wait until other tasks are done in order to run moldable tasks on wider teams, possibly improving performance.

One interesting detail is that unlike with normal tasks, it is possible for a thread to steal a moldable task from itself. This is in case a task that is scheduled to be executed using a moldable task team cannot execute right now because it's overlapping with another currently executing team. In that case, it can steal the moldable task from itself to run it on a different team.

4

Results

4.1 Hardware

We tested our implementation on the Dardel HPC system, an HPE Cray EX supercomputer. We ran each benchmark on separate nodes, where every node has 2 AMD EPYC™ Zen2 2.25 GHz processors with 64 processors each. Furthermore, each core has simultaneous multithreading with 2 threads, resulting in a total of 256 logical processors. The cores are also separated into NUMA nodes with 4 nodes per socket. With such a deep memory hierarchy, locality-aware methods should have an advantage. For the purpose of creating moldable task teams, we treat this as a memory hierarchy of depth 5 with moldable task teams being created for every socket, NUMA node, L3 cache, core and logical processor. There is no team which contains both sockets, so there's no team which contains all cores.

We ran all our programs in containers to ease development across machines. We used Singularity/Apptainer as it has been shown to have low overhead, especially compared with technologies such as Docker [14]. Even if there was a performance impact, we expect the relative speed-up between implementations to be unaffected.

4.2 SparseLU

We tested our implementation on the BOTS benchmark, specifically the SparseLU benchmark [15]. It computes an LU decomposition of a sparse $n \times n$ matrix, with $m \times m$ sub-matrices. This is a very memory-intensive task, which should highlight the advantages of moldable tasks. Another advantage for moldable tasks when it comes to SparseLU is that moldable tasks should be able to adapt to the decreasing number of tasks between barriers during the execution of the program. Algorithm 3 shows an overview of the benchmark.

When compiling the runtime it was configured to run efficiently on up to 256 threads by increasing the task queue size from the default of 256 to 1024. Otherwise, there may not have been enough tasks in queues for work-stealing to work well. This is because during normal task scheduling the tasks are executed immediately if the encountering thread's queue is full causing the task-creating thread to be busy.

```
Data: n > 0, m > 0, M (n × n matrix)
for k ← 0 to n − 1 do
    for i ← k + 1 to n − 1 do
        M ← A(M, k, i)
    end
    for i ← k + 1 to n − 1 do
        M ← B(M, k, i)
    end
    <WAIT FOR TASKS TO COMPLETE>
    for i ← k + 1 to n − 1 do
        for j ← k + 1 to n − 1 do
            M ← C(M, k, i, j)
        end
    end
    <WAIT FOR TASKS TO COMPLETE>
end
```

Algorithm 1: LU decomposition. A, B and C are subroutines which spawn tasks.

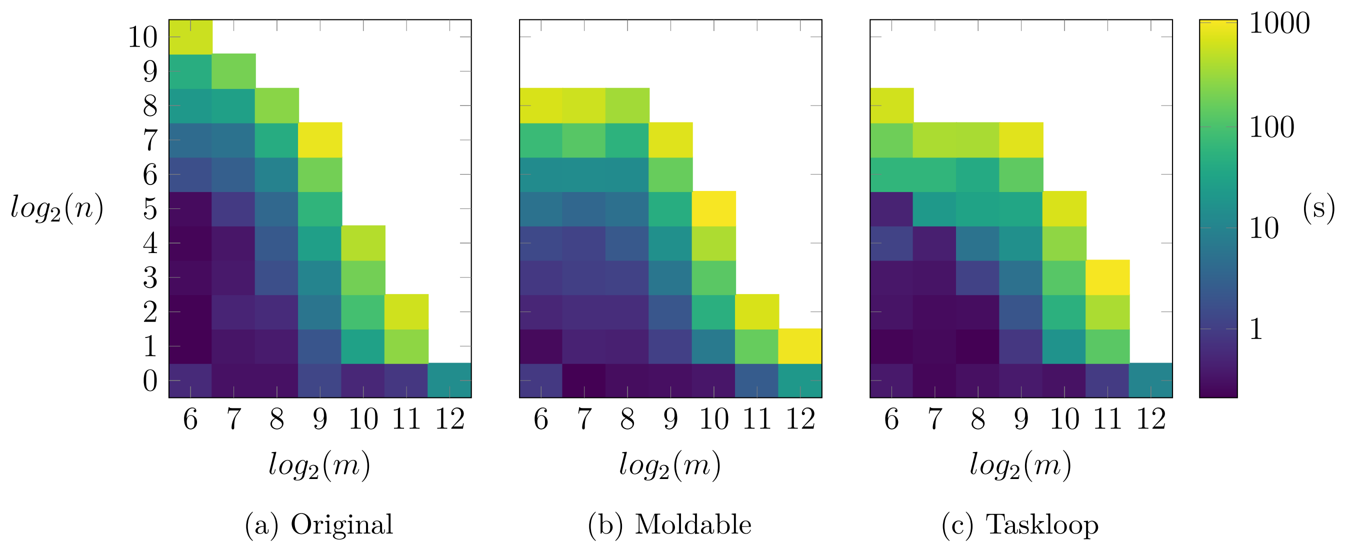
We created three different variants of the benchmark to evaluate our implementation.

1. In the **original** variant, the subroutines A, B and C each spawn a normal task, which performs $\mathcal{O}(m^3)$ work. We changed the original code slightly by removing the use of an untied task which spawned all other tasks, instead making a single thread spawn all tasks in the outer parallel region. This was done to avoid stack overflows.

2. The **moldable** variant is similar to the original variant, but uses moldable tasks instead of normal tasks. The only change to the program was to add `moldable` to every `task` construct and to parallelize the tasks using `parallel` constructs.

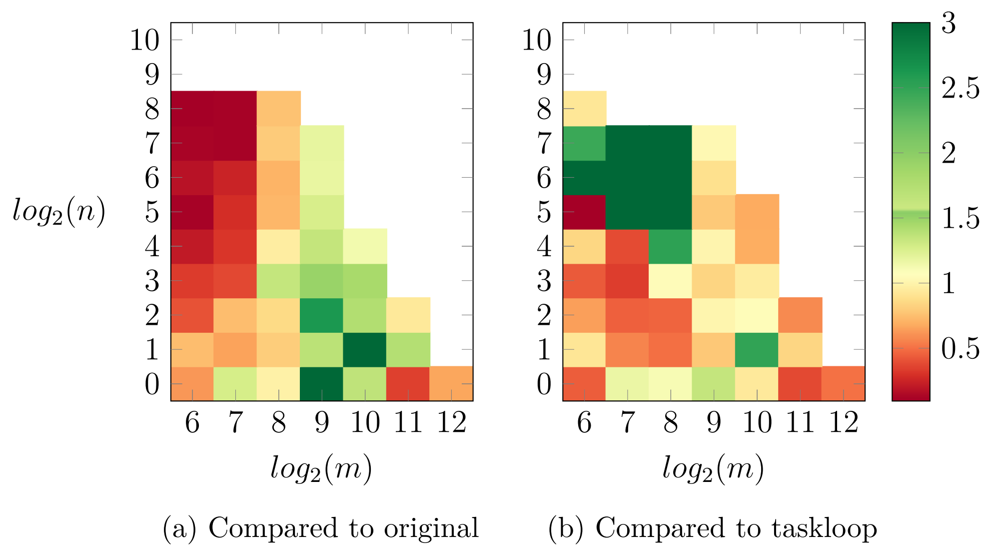
3. The **taskloop** variant once again uses normal tasks but uses a `taskloop` construct in C, spawning $\mathcal{O}(m)$ tasks with $\mathcal{O}(m^2)$ work each, giving it finer task granularity. See Section 2.7.2 for a description of the `taskloop` construct.

We then executed each variant, sweeping through values for n and m. The runtime of each variant can be seen in Figure 4.1.



(a) Original (b) Moldable (c) Taskloop

Figure 4.1: Runtime of each variant of the SparseLU benchmark depending on n, the size of the matrix, and m, the size of the sub-matrices. Cells for benchmarks that ran for more than 1000 seconds are left blank.



(a) Compared to original (b) Compared to taskloop

Figure 4.2: Speedup of SparseLU using moldable tasks, depending on n, the size of the matrix, and m, the size of the sub-matrices. Cells where either variant took more than 1000 seconds to execute are left blank.

If we compare the moldable and original variants, we find a speed-up with moldable tasks when m is large; i.e. with more work per task. This can be seen in Figure 4.2a. The worse performance at smaller m is to be expected as moldable tasks have a larger overhead than normal tasks.

In both comparisons, we can see that when n is small, which means we don't have that many tasks in our benchmark, the variance is higher because we don't have enough tasks to estimate the execution times so the scheduling becomes worse. For higher values of n, we find that moldable tasks strike a middle ground between coarse-grained tasks and fine-grained tasks, exploiting parallelism better than the original variant as long as the tasks are large enough and handling small tasks better than the taskloop variant. The performance when the work per task is small is worse than expected compared to the original variant and is most likely from how we handle threads and various global locks that cause contention in our implementation.

4.3 Synthetic benchmark

We also tested our implementation on a synthetic benchmark to more precisely measure the overhead of moldable tasks. The algorithm is described in Algorithm 2. It spawns a number of tasks, where each task executes a busy-wait loop for a set amount of wait time. This can be seen as simulating a very compute-intensive program with uniform tasks. This benchmark allows us to investigate the overhead of moldable tasks. When we decrease the wait time for each task, the overhead created by scheduling, work stealing and task execution becomes a larger part of the overall runtime. As we can see in Figure 4.3, moldable tasks perform much worse than normal tasks in this scenario.

Data: $n > 0$, $m > 0$, $p > 0$, $s > 0$
for $i \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** m **do**
        $B(p, s)$
    end
    <WAIT FOR TASKS TO COMPLETE>
end

Algorithm 2: Synthetic benchmark. $B(p, s)$ spawns a task which runs a busy wait loop p times, where each iteration waits for s microseconds (µs). When testing moldable tasks, we add a `parallel for` in B. This allows one task to run for only s ms if scheduled on a team of width p.

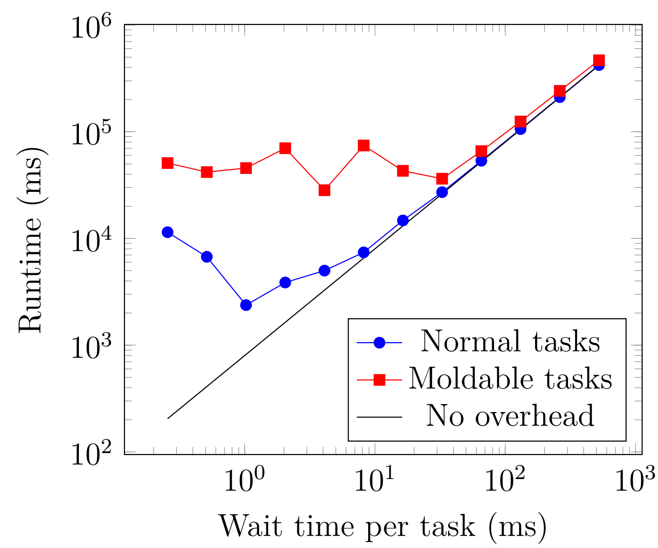


Figure 4.3: Execution time of synthetic benchmark, where $n = 100$, $m = 2048$ and $p = 256$.

5

Conclusion

We have extended the LLVM OpenMP runtime with the ability to schedule and execute moldable tasks. Our code is flexible and leaves room for further experimentation. In the benchmarks, we find an improvement in performance compared to normal tasks in some cases, especially when each task is large. The worse performance when tasks are small is most likely caused by the increased overhead from moldable tasks.

There are several parts of the implementation which could be improved. One part is the scheduling, both during initial task creation and during work-stealing. Our implementation uses simple heuristics in both cases and could be improved by using methods that incorporate the amount of work a thread has in its queue. It would most likely also be a good idea to unify the scheduling done at initial task creation and when work-stealing, as work-stealing happens even when we are still creating new tasks.

Related to scheduling, our execution time estimation could be improved. We treat all teams as unrelated when estimating execution times which means we need more samples than necessary. One could try incorporating a speedup model to model correlations between the runtimes of different teams. This would reduce the warmup time when we schedule to all teams to find out their performance. One drawback of using a speedup model is that they often assume the system only has homogeneous cores.

Lastly, to reduce the overhead from moldable tasks, the thread management could be improved. Our implementation uses separate worker threads to execute moldable tasks, instead of the existing task execution threads which we suspend when execution starts and resume when the task is done. This causes unwanted delays and synchronization costs.

Bibliography

[1] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999, ISSN: 0004-5411. DOI: 10.1145/324133.324234.

[2] T. Alexandersson and O. Nilsson, "Implementing online performance modelling in OpenMP Adding an energy aware scheduler for increased energy efficiency," eng, 2022. [Online]. Available: https://hdl.handle.net/20.500.12380/304695 (visited on 06/07/2023).

[3] C. Terboven *et al.*, "Approaches for task affinity in OpenMP," in *OpenMP: Memory, Devices, and Tasks*, N. Maruyama, B. R. de Supinski, and M. Wahib, Eds., Cham: Springer International Publishing, 2016, pp. 102–115, ISBN: 978-3-319-45550-1.

[4] M. Pericàs, "Elastic Places: An Adaptive Resource Manager for Scalable and Portable Performance," en, *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 2, pp. 1–26, Jun. 2018, ISSN: 1544-3566, 1544-3973. DOI: 10.1145/3185458. [Online]. Available: https://dl.acm.org/doi/10.1145/3185458 (visited on 06/07/2023).

[5] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. [Online]. Available: https://doi.org/10.1145/1498765.1498785 (visited on 06/06/2023).

[6] J. T. Havill and W. Mao, "Competitive online scheduling of perfectly malleable jobs with setup times," en, *European Journal of Operational Research*, vol. 187, no. 3, pp. 1126–1142, Jun. 2008, ISSN: 0377-2217. DOI: 10.1016/j.ejor.2006.06.064. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221706008265 (visited on 06/06/2023).

[7] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ser. AFIPS '67 (Spring), New York, NY, USA: Association for Computing Machinery, Apr. 1967, pp. 483–485, ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. [Online]. Available: https://dl.acm.org/doi/10.1145/1465482.1465560 (visited on 06/06/2023).

[8] T. Hikida, H. Nishikawa, and H. Tomiyama, "Heuristic algorithms for dynamic scheduling of moldable tasks in multicore embedded systems," en, *International Journal of Reconfigurable and Embedded Systems (IJRES)*, vol. 10, no. 3, pp. 157–167, Nov. 2021, ISSN: 2722-2608. DOI: 10.11591/ijres.v10.

i3.pp157-167. [Online]. Available: https://ijres.iaescore.com/index.php/IJRES/article/view/20362 (visited on 06/06/2023).

[9] A. Benoit, L. Perotin, Y. Robert, and H. Sun, "Online Scheduling of Moldable Task Graphs under Common Speedup Models," en, Aug. 2022. [Online]. Available: https://inria.hal.science/hal-03778405 (visited on 06/05/2023).

[10] M. Abduljabbar, M. Eljammaly, and M. Pericas, *Mitigating inefficient task mappings with an Adaptive Resource-Moldable Scheduler (ARMS)*, Dec. 2021. DOI: 10.48550/arXiv.2112.09509.

[11] V. Lopez, J. Criado, R. Peñacoba, R. Ferrer, X. Teruel, and M. Garcia-Gasulla, "An OpenMP free agent threads implementation," in *OpenMP: Enabling Massive Node-Level Parallelism*, S. McIntosh-Smith, B. R. de Supinski, and J. Klinkenberg, Eds., Cham: Springer International Publishing, 2021, pp. 211–225, ISBN: 978-3-030-85262-7.

[12] J. Criado, V. Lopez, J. Vinyals-Ylla-Catala, G. Ramirez-Miranda, X. Teruel, and M. Garcia-Gasulla, "Exploiting OpenMP malleability with free agent threads and DLB," in *High Performance Computing. ISC High Performance 2022 International Workshops*, H. Anzt, A. Bienz, P. Luszczek, and M. Baboulin, Eds., Cham: Springer International Publishing, 2022, pp. 162–175, ISBN: 978-3-031-23220-6.

[13] "Execution Model." (2021), [Online]. Available: https://www.openmp.org/spec-html/5.2/openmpse3.html (visited on 06/05/2023).

[14] C. Arango, R. Dernat, and J. Sanabria, *Performance Evaluation of Container-based Virtualization for High Performance Computing Environments*, Sep. 2017. DOI: 10.48550/arXiv.1709.10140. [Online]. Available: http://arxiv.org/abs/1709.10140 (visited on 06/05/2023).

[15] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *2009 International Conference on Parallel Processing*, 2009, pp. 124–131. DOI: 10.1109/ICPP.2009.64.

[16] D. Computadors, V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," *WoTUG-18*, vol. 44, Mar. 1995.

A

Appendix 1: Implementation Details

We have in the rest of the thesis avoided discussing any specific code changes and some of the more technical details of our work. We will here include some of those details, to make our work more reproducible. Our changes were done to the LLVM repository at commit `4c273cd0 (2023-02-13)`. We added approximately 1600 lines of code to the project.[1]

A.1 Front end

For the front end, we changed some files in the `clang/` folder. We already knew that `untied` was a clause for `task` constructs, so we searched for any occurrence of `untied` and copied the same functionality for `moldable`. We then changed the function `emitTaskInit` to detect if a `moldable` clause is present and pass it through when generating a call to `__kmpc_omp_task_alloc`, which is what causes the runtime to spawn a new task.

A.2 Back end

All files we changed for the back end were in the `openmp/runtime/src` folder.

A.2.1 Moldable task team hierarchy

The moldable task teams are stored in structs of type `kmp_base_thread_data` which is defined in `kmp.h`. They are generated in `__kmp_realloc_task_threads_data`, which is used to allocate a new task team or reuse an existing one. Each moldable task team is assigned a mask which represents which cores it consists of, but the mask doesn't have any topology information. To generate the masks, we iterate through all cores and get their corresponding "topology ID". This ID is used by the existing affinity mechanisms and contains information about where the core is in the memory hierarchy. Each ID contains 12 numbers, where each number is some part of the memory hierarchy (e.g. socket, NUMA node, core) in order of granularity.

[1]Our code can be found at github.com/Zinfour/llvm-project.

Every core in a moldable task team has a prefix of its ID in common with the other cores in the same team.

A.2.2 Steal order lists

After we've generated the moldable task teams, we generate a steal order list for every thread. This is used when work-stealing to steal from "close" threads first. For each thread, we execute Algorithm 3.

```
Data: T: All threads, M: Maximum number of teams per thread, t_c: The
      current thread
begin
    A ← []
    for i ← 0 to M − 1 do
        for t ∈ T do
            core ← GetMask(t, i)
            if t_c ∉ mask then
                continue
            end
            B ← []
            for core ∈ mask do
                if core ∉ A ∪ B then
                    AddItem (B, core)
                end
            end
            B ← FisherYates(B)
            A ← A + B
        end
    end
    return A
end
```

Algorithm 3: Steal order list creation

A.2.3 Work-stealing

Work-stealing was implemented in `__kmp_execute_tasks_template`. We wrote `__kmp_steal_moldable_task` to steal from moldable task queues and then implemented logic to choose which moldable task team to execute the task on.

A.2.4 Executing a moldable task

To execute moldable tasks we added the function `__kmp_execute_moldable_task` to `kmp_tasking.cpp` and call it from `__kmp_execute_tasks_template`. In `__kmp_execute_moldable_task` we want to create a teams region with the task as its inner region so we need a way to convert the tasks `kmp_routine_entry_t` to a `microtask_t`. This is done using `__kmp_invoke_task_dummy2`. `__kmp_execute_moldable_task` also sets the thread affinity of the master

thread which is then copied by the worker threads in `__kmp_fork_barrier` in `kmp_barrier.cpp`

A.2.5 Scheduling moldable tasks

Scheduling was implemented in `__kmp_push_task`. Estimates for task execution times are stored in `__kmp_task_stats_list` which is initialized in `kmp_global.cpp` and freed in `kmp_runtime.cpp` in the function `__kmp_internal_end`.

A.2.6 Timetable

When debugging our implementation, we found it useful to visualize when cores were executing moldable tasks. We attempted to use Paraver [16], but did not manage to get it to work. We printed (in debug builds) information about each moldable task whenever we finished executing it, and then parsed the output using a script called `timetable.py`. An example of the output can be found in Figure A.1.

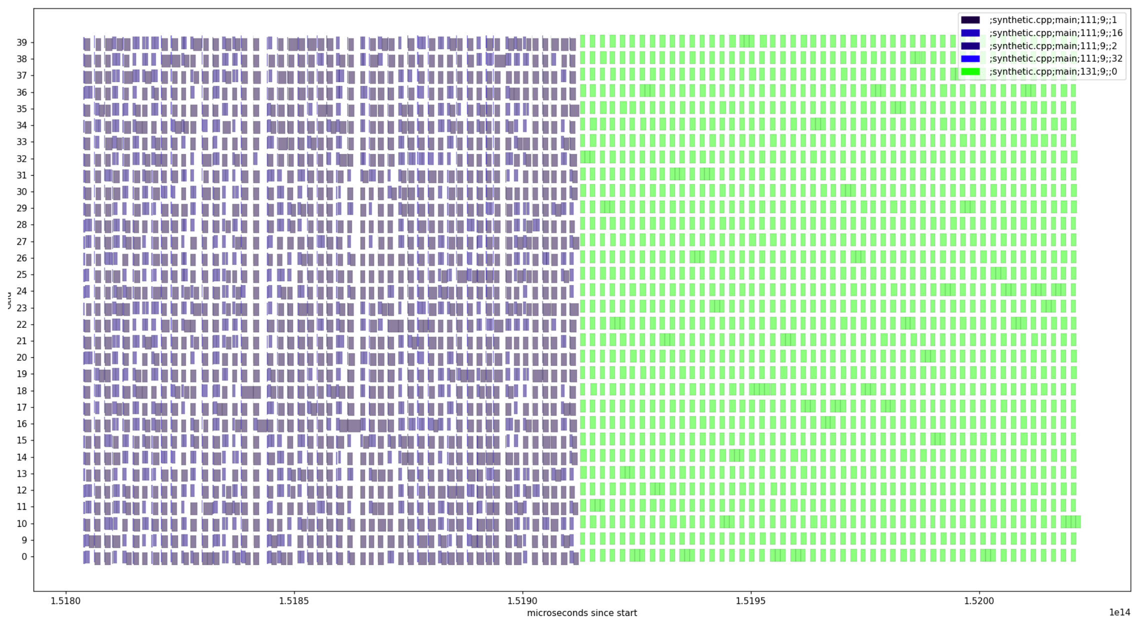


Figure A.1: Example visualisation using `timetable.py`