Task Self-Scheduling in OpenMP

Jonathan Giger

June 2021

Abstract

The OpenMP standard allows for programs using loops to be parallelized either as loop iterations or as tasks. This project analyzes the possible performance gains which can be achieved when using tasks. The current LLVM runtime library uses a static steal method to schedule tasks for execution. In this project we modify the runtime to steal a chunk of tasks of a given size in order to reduce the time spent on overhead. The results show a promising reduction in parallel execution time when stealing chunks of tasks for programs that have a large number of tasks.

Submitted by:

Jonathan Giger

Supervisors:

Jonas H. Müller Korndörfer

Dr. Ali Omar Abdelazim Mohammed

Examiner:

Prof. Dr. Florina M. Ciorba

Contents

1 Introduction **3**
- 1.1 OpenMP Tasks . . . 3
- 1.2 History of OpenMP Tasking . . . 3
- 1.3 Converting applications from parallel loop to tasking . . . 3
  - 1.3.1 Puretask Single . . . 4
  - 1.3.2 Puretask Parallel . . . 4
  - 1.3.3 Taskloop Linear . . . 5
  - 1.3.4 Taskloop Recursive . . . 6
- 1.4 Scheduling Tasks . . . 7
  - 1.4.1 Task Execution Logic . . . 7
  - 1.4.2 Task Stealing Logic . . . 8

2 Related Work **8**

3 Methods **9**
- 3.1 Visualizing task execution . . . 9
  - 3.1.1 Enabling Task IDs . . . 9
  - 3.1.2 Exporting Task Timestamps . . . 10
- 3.2 Experiment Setup . . . 10
  - 3.2.1 Generating the task timeline visualization . . . 10
  - 3.2.2 Taskloop Guided Scheduling Experiment . . . 11
  - 3.2.3 Loop Shifting Technique . . . 12

4 Results **13**
- 4.1 Puretask Single . . . 13
- 4.2 Puretask Parallel . . . 14
- 4.3 Task time distribution . . . 15
- 4.4 Linear Taskloop with Guided scheduling . . . 15
- 4.5 Standard LLVM and Vector task stealing . . . 16

5 Discussion **17**
- 5.1 Overhead, affinity, and load imbalance . . . 17
- 5.2 Increasing task affinity without downsides . . . 17
- 5.3 Reduced variance in program execution time . . . 18
- 5.4 Overhead and load imbalance bathtub curve . . . 18

6 Conclusion **19**

1 Introduction

1.1 OpenMP Tasks

When using OpenMP to parallelize code, traditionally the code would be in the form of a for loop and a construct would be used to split this loop into iterations. With the advent of tasking, any given block of code can be labeled a task.

One use-case of tasking is to simply create a task inside of a for loop for code that doesn't not have any intraloop dependencies. Another use case is to create tasks in a recursive manner that depend on their parent task, for example when solving a Sudoku puzzle. In this paper, we will only discuss tasks without dependencies.

Previous work has shown that a program parallelized using tasks can perform better than its parallel loop equivalent in some cases. [4] The goal of the project is to find the balance between overhead caused by the sheer number of tasks in a tasking program and overhead caused by load imbalance due to having tasks that are too large in a program's execution. During the course of the experiment and according to the related work, we also found that the task-to-data affinity could also be a secondary benefit from this work.

In this paper we compare and contrast the LLVM tasking implementation with some other implementations discussed in research papers. We then make changes the the nature in which tasks are scheduled, and perform benchmarks on the new implementation. The results will then be analyzed by comparing the program execution times and the task affinity. We then explore the potential for further research on this topic, for instance by continuing the experiment with more scheduling algorithms and a larger variety of parallel programs.

1.2 History of OpenMP Tasking

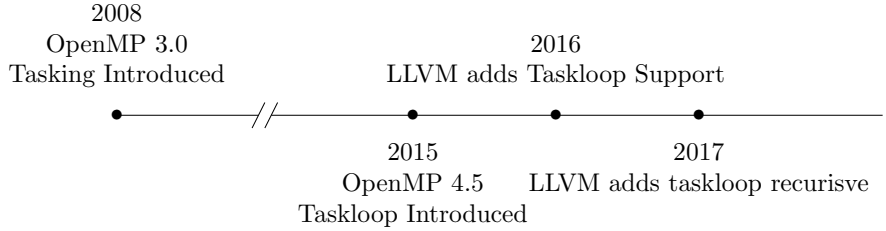

Figure 1: Timeline

1.3 Converting applications from parallel loop to tasking

As seen in the timeline, there are several methods that a programmer can use to convert a program parallelized with loops into a program parallelized with tasking. The methods that use the task construct will be referred to as puretask

methods and the methods that use the taskloop construct will be referred to as taskloop methods in this paper. While neither method is considered to be the correct way, some are considered to be more NUMA friendly, as they distribute the workload between cores in different ways.

1.3.1 Puretask Single

This is the most common implementation used when converting a loop into tasks without using taskloop. This implementation is known to be problematic on NUMA architectures.[6]

1.3.2 Puretask Parallel

This method is similar to the previous method, but the generation of tasks is split evenly among the threads. The resulting distribution of tasks depends on the scheduling method used to parallelize the for loop. If a simple static schedule is used, which is the default, each thread will create the same number of tasks, and they will be in sequential order.

1.3.3 Taskloop Linear

After the introduction of tasking in OpenMP 3.0 it has become commonplace to parallelize a for loop using tasks, by simply defining the entire contents of a for loop as a task. In 2015 the taskloop construct was added with the release of OpenMP 4.5 which automates this behavior. The exact behavior regarding how many tasks each thread creates is left up to the runtime to decide. In LLVM, the task generation is done through a combination of taskloop linear and taskloop recursive. While the taskloop construct is specified in the OpenMP standard, the method used to determine the order and which thread the tasks will be created on is left up to the implementation to decide. The desired grain size or the number of tasks that should be created is also configurable by the user.[3]

This is the simplest implementation of the taskloop construct. First, a task is generated with a lower bound which is the lowest value of the index variable and an upper bound which is the highest value of the index variable. This main task is then split into new tasks with new lower and upper bound values based on the grain size or num_tasks parameter. The extra iterations that do not divide into the grain size are distributed evenly. This implementation results in very high task affinity for the thread that is creating the tasks, but all the tasks will be created on one thread.

Listing 1: Task-parallel Single

```
#pragma omp parallel
{
#pragma omp single
{
for (int i = 0; i < N; i++)
#pragma omp task
{
    do_work();
}
}
}
```

Listing 2: Taskloop

```
#pragma omp taskloop
for (int i = 0; i < N; i++)
{
    do_work();
}
```

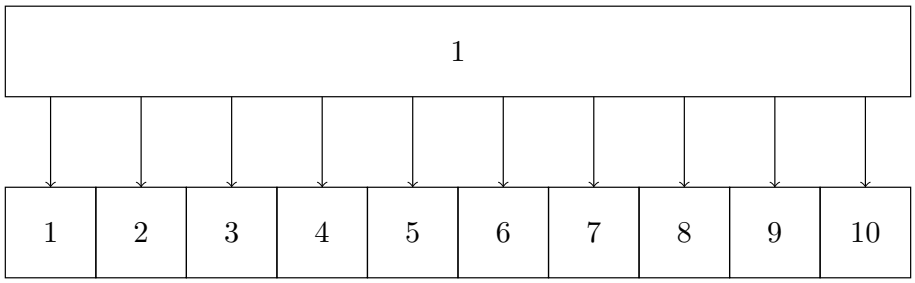

Figure 2: Taskloop Linear

1.3.4 Taskloop Recursive

Since the method used to generate the tasks when using taskloop is left up to the implementation, it is not documented by the OpenMP runtime. The recursive taskloop function also differs between each OpenMP implementation, with some implementations not using it at all. The description in this paper refers to the current LLVM implementation. The first task is generated with the entire upper and lower bound just like in a linear taskloop, it is then split into two even halves. These halves are then split again until a threshold is reached. Once the threshold is reached, the resulting tasks are then split in a linear manner until the desired grain size is reached. This implementation was created with the goals to make the taskloop construct more NUMA friendly.

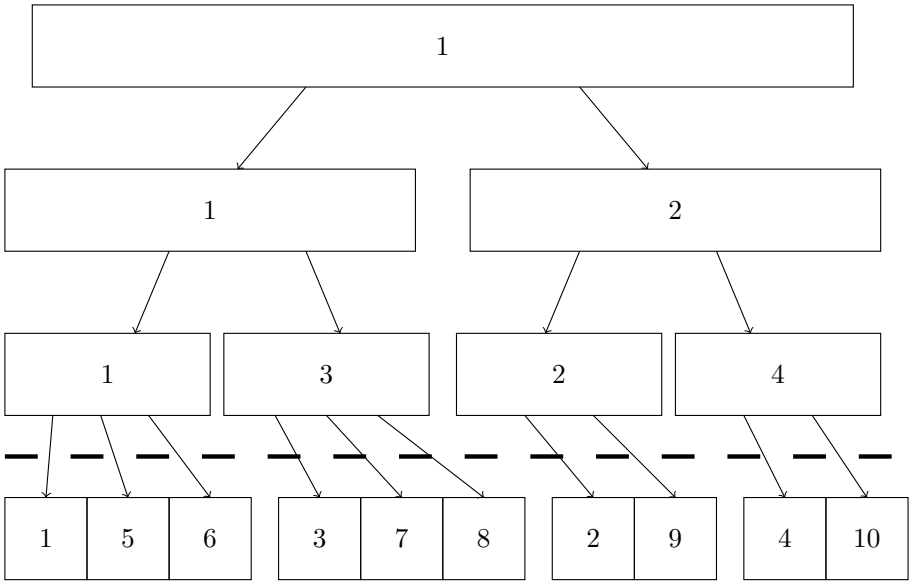

Figure 3: Taskloop Recursive (Threshold shown as dotted line)

1.4 Scheduling Tasks

1.4.1 Task Execution Logic

This diagram explains the logic that a thread uses to determine which tasks to execute, whether they are from the current thread or whether they have to be stolen from another note. Note how simple the logic is, a thread simply saves the thread number after it has successfully stolen a task from another thread, or otherwise it simply picks a random thread number. This behavior is changed in the task-to-data affinity related work, in which they change the logic in the LLVM runtime to prefer threads from the same node.

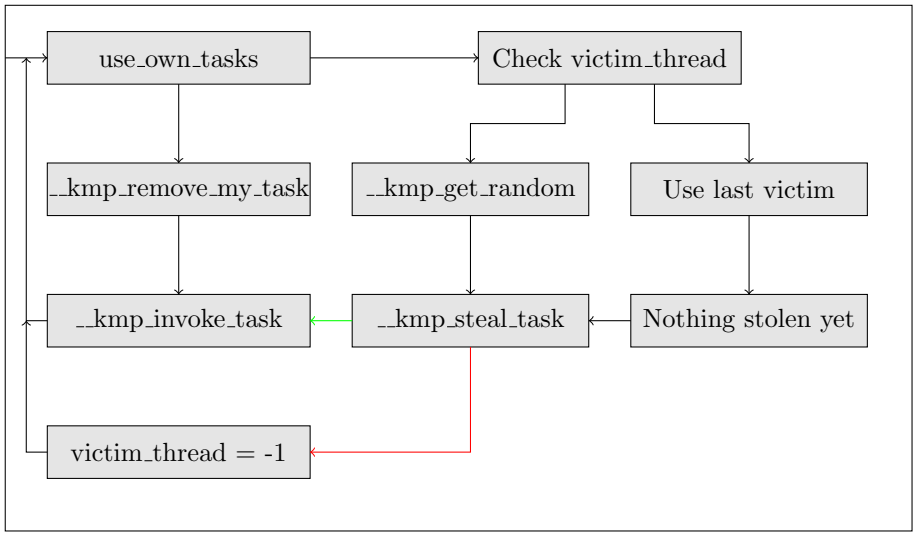

Figure 4: Task Execution Logic

1.4.2 Task Stealing Logic

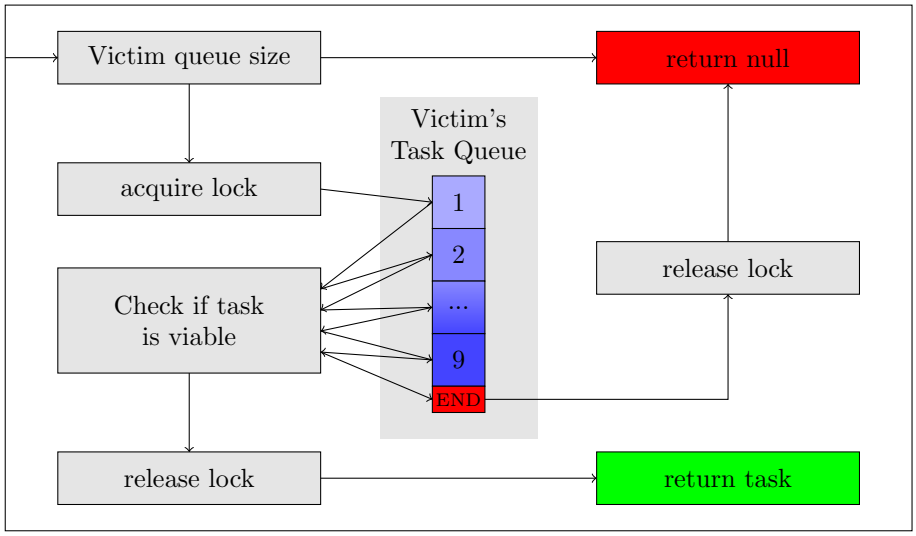

Figure 5: Task Stealing Logic

2 Related Work

Since this paper analyzes the LLVM implementation of OpenMP tasking, it is important to also analyze what other research exists for this implementation and others as well. Since this paper focuses on the overhead caused by scheduling tasks, the exact amount of time it takes for one thread to take a task from the queue would be of particular interest. Past research has measured the exact amount of time that it takes for a thread to steal a task from another thread (LLVM) or from a central queue (GOMP) and found it to be approximately 35 microseconds when using 20 threads on LLVM[1]. This number matches up with the results in our experiment. For example in Figure 12, reducing the grain size from 50 to 1 introduces 2054769 more tasks. Of these, 60-80% of the tasks are stolen by another thread. This number can be multiplied by the time it takes to steal one task, but one single thread only affects the total program execution time by 1/20 of this, so the result must be divided by 20. This would result in an expected total increase in program execution time of 2.2 to 2.9 seconds, which is in the range of the results from this experiment. Also of note is that they acknowledge that "it is a common pattern to create tasks with a single or master construct" and that "it might be useful to develop a specialized implementation for this use-case". A possible implementation for this use-case would be the taskvector stealing as described in this paper.

Another possible area of research when it comes to improving the perfor-

mance of parallel OpenMP applications using tasks is to implement scheduling algorithms to determine which tasks to schedule on which threads. Experiments implementing these scheduling algorithms have been done using the OpenUH runtime library[5]. They use the OpenUH runtime because it is highly customizable when it comes to tasking. The queue and task stealing behavior can be customized with options set at runtime, including which end of the queue tasks should be added to, which end of the queue tasks should be stolen from, and how many tasks should be stolen at once.

When it comes to scheduling tasks, overhead is usually the first thing to be considered, but the task-to-data affinity should not be overlooked either. Experiments have shown that speedup can be improved by up to 4x when using the new "affinity" parameter in OpenMP 5.0 when using the taskloop clause[2]. There is a possible overlap between the affinity gained when using the affinity parameter and the improved affinity when stealing multiple tasks in row using vectors of tasks. They also introduce the concept of "NUMA-aware task stealing" which modifies the logic in Figure 4 to prefer tasks from the same node.

3 Methods

3.1 Visualizing task execution

In order to better understand how tasks are scheduled and executed in the LLVM OpenMP runtime, we needed a way to understand how a program using tasks has executed in an efficient manner for a variety of programs. By assigning ID numbers to each of the tasks, printing the start and end times of the tasks executions, and visualizing the printed time in a timeline we were able to quickly evaluate the execution of different programs.

3.1.1 Enabling Task IDs

By default, the LLVM runtime does not create task ID numbers when new tasks are created. The runtime always refers to tasks by using a pointer to the task data's memory address. It does have debugging functionality built-in which allows for the creation of task ID numbers. In this project we do not turn on the entire debugging functionality, but we do enable the generation of task ID numbers. This is done by calling the function to generate a task ID when a new task is allocated.

Since both implicit and explicit tasks both call this function to generate task ID numbers, they both share the same counter. In all of the experiments that were conducted in this project, all implicit tasks were created before any explicit tasks were created. When running experiments using 20 threads there were approximately 82 implicit tasks created, which resulted in the task ID numbers for the explicit tasks starting at approximately 83. In order to improve readability and prevent lapses in the task ID numbers, the KMP_GEN_TASK_ID was duplicated and each was set to a separate counter. This ensured that the

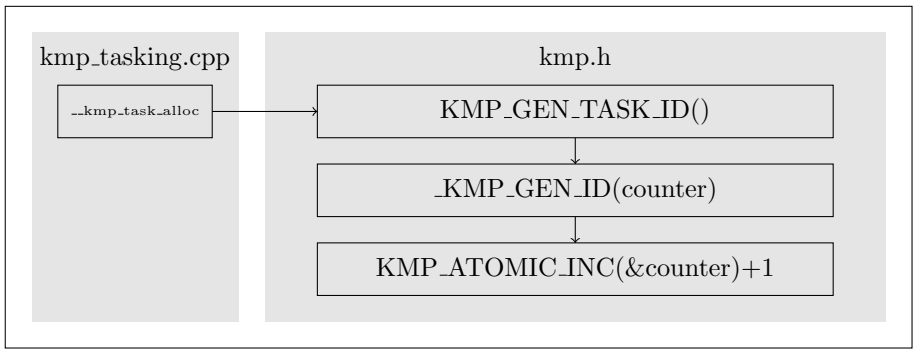

Figure 6: Task ID Generation

task ID numbers for explicit tasks always started at one and always incremented by one.

3.1.2 Exporting Task Timestamps

The simplest way to measure the start and end time of a task would be to simply add a print statement in the task start and task end functions in the runtime. In order to reduce the overhead caused by the large amount of print statements being buffered, these values are instead saved to a global array. At the end of the program execution, the buffer is printed to a CSV file named taskTimeOutput. A sample of the file can be seen in Table 1.

Timestamp	**Executing Thread**	**Task ID**	**Creating Thread**	**1=Start 0=Stop**
5708049.64039087	14	3	0	1
5708049.64044245	6	2	0	1
5708049.64046568	14	3	0	0
5708049.64051432	14	1	0	1
5708049.64057035	6	2	0	0
5708049.64063098	14	1	0	0

Table 1: Sample task time output

3.2 Experiment Setup

3.2.1 Generating the task timeline visualization

When executing a program using the LB4OMP library with the tasking file from this project installed, the feature to export task times has to be turned on using environment variables. Simply add these environment variables into your sbatch script.

Listing 3: Setup environment variables

```
EXPORT OMP_EXPORT_TASK_TIMES 1
EXPORT OMP_EXPORT_CHUNK_BOUNDS 1
EXPORT OMP_PRINT_ENV_VARIABLES 1
```

After execution, the task start and end times will be printed into a file named taskTimeOutput.csv in the source directory of the program. The taskTimeOutput.csv file can then be visualized using the provided Python tool. Simply run the python program with the CSV file as an argument. A PDF file with the visualization will be generated in the same directory.

The plot in the top right shows each task as a blue box. The X axis is time and the Y axis is the thread number that executed the task. The plot on the is the same plot but zoomed in on the first few milliseconds on runtime. The histogram with the task ID number on the X axis and the task's execution time on the Y axis.

3.2.2 Taskloop Guided Scheduling Experiment

The first experiment conducted in this project was to apply a scheduling algorithm to the linear taskloop function. Since the linear taskloop function simply uses a for loop to create the tasks with a given size, the size values were modified so that they were generated from a function. What resulted is a series of tasks that start with a large size and become smaller. Upon running the experiment it was found that the grain size was too large, any additional modifications to the grain size would restrict the use cases of the solution, and we pivoted the project to only using adaptive scheduling techniques.

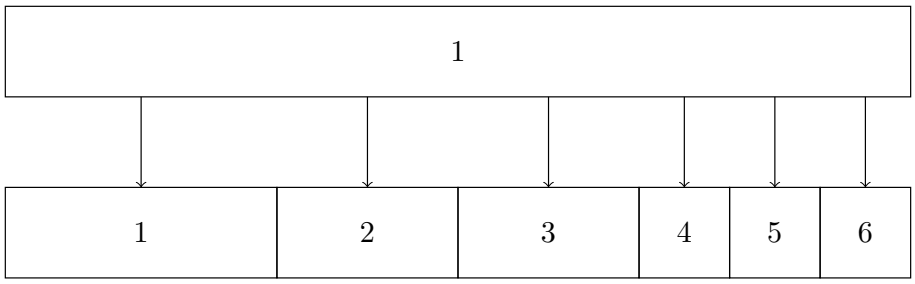

Figure 7: Taskloop Linear using Guided scheduling

3.2.3 Loop Shifting Technique

After evaluating the linear taskloop with scheduling technique and before implementing the taskvector stealing technique, the concept of rearranging the order of the tasks was explored. The idea was that if the small tasks could be moved to the end of the runtime and the large tasks could be executed immediately, the load imbalance could be reduced without affecting the total task switching overhead at all. This was achieved by simply modifying the loop index iterator to start and end at a number other than zero.

Listing 4: Without shift

```
#pragma omp parallel
{
#pragma omp single
{
for (int i = 0; i < N; i++)
#pragma omp task
{
    do_work(i);
}
}
}
```

Listing 5: With shift value

```
#pragma omp parallel
{
#pragma omp single
{
for (int i = 0; i < N; i++)
#pragma omp task
{
    q=((i+(N/2))%N)+shift;
    do_work(q);
}
}
}
```

Changing the starting index of the loop did decrease the total execution time with some shift values, and increase it with other shift values, however the shift values that resulted in the decrease were too specific to this specific application that the experiment was not useful in the general case. Since the goal of this project is to increase the performance for all programs using the LLVM runtime and not just certain specific applications, this concept was not further evaluated.

4 Results

4.1 Puretask Single

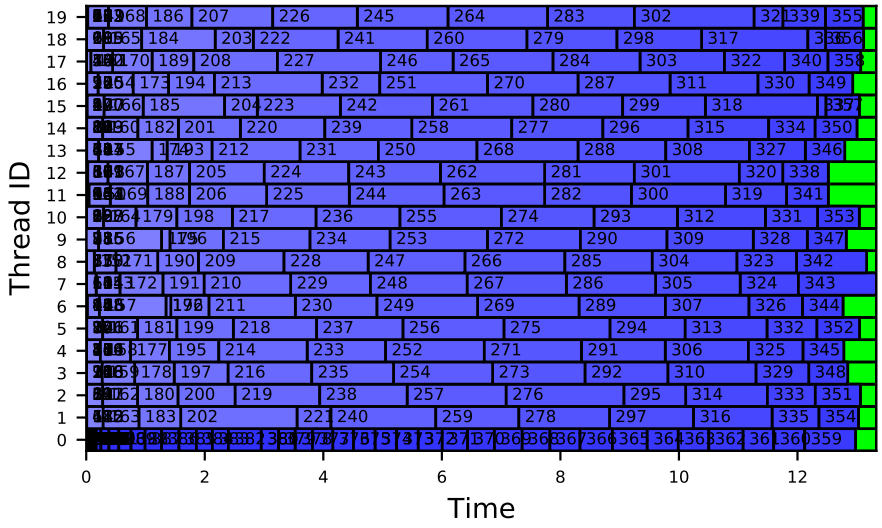

Figure 8: Task Distribution Puretask Single

This visualization is generated by the visualize.py python script. The red area shows time where a thread is sitting idle while there are still tasks left to be computed, this can be because the tasks are either not generated yet or the thread is currently switching from one task to another. The blue area shows time that a thread is completing a task, the darker the shade of blue, the higher the thread ID. The green area shows time that a thread is idle due to load imbalance.

In this visualization, all tasks are created by one thread. Since a "single" construct was used instead of a "master" construct, any thread could have been the one creating the tasks. By looking at the thread ID numbers, which are decreasing for thread 0 and increasing for all other threads, it is clear that thread 0 is the one creating tasks in this execution of the program.

Another point to note from this graph is that the task ID numbers are sequential for thread 0, which means that they are generated in a sequential order, however they skip numbers on all the other threads. This is the basis for the concept of improving task-affinity in this paper.

4.2 Puretask Parallel

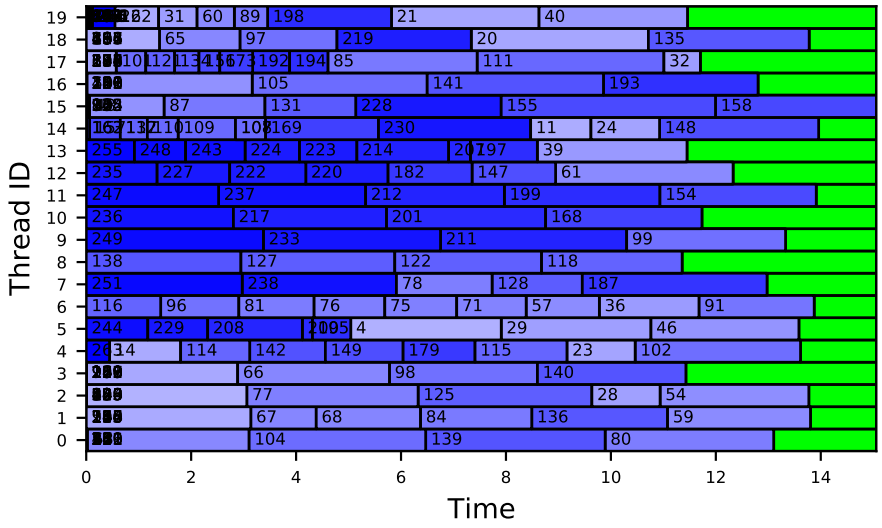

Figure 9: Task Distribution Puretask Parallel

This visualization works in the same way as the previous one, the only difference is that instead of using a "single" construct to generate the tasks, a "parallel for" is used. Logically, this would mean the the scheduler for the for loop distributes the iterations of the for loop, and each thread then simply converts its for loop iterations into tasks.

While in the previous visualization the task ID numbers correlated with the location of the pixel in the Mandelbrot program, this was simply a coincidence because there was only one thread accessing the atomic iterator that generates the task ID numbers, and the iterator was being accessed in the order of the original for loop. In this case here with the parallel task generation, multiple threads are accessing this atomic iterator with different parts of the original for loop, essentially creating a random ordering of the task ID numbers.

Even though this is the case, using the distribution of the task execution times from figure 9, the distribution of the original loop iterations can be seen. Since we know that the Mandelbrot program has small tasks are the beginning and end of the program and large tasks in the middle, we can see that according to this visualization, the large tasks were at first distributed to the threads in the middle of the plot, while the high and low thread numbers started executing the small tasks.

4.3 Task time distribution

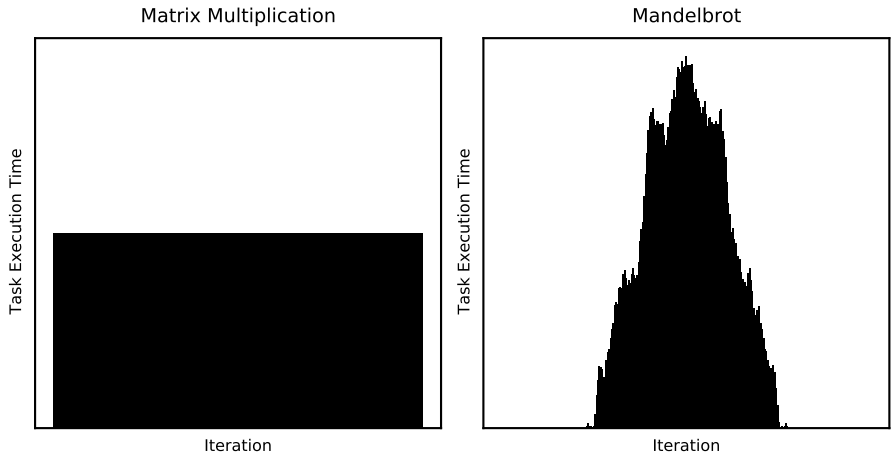

Figure 10: Task execution time distribution

4.4 Linear Taskloop with Guided scheduling

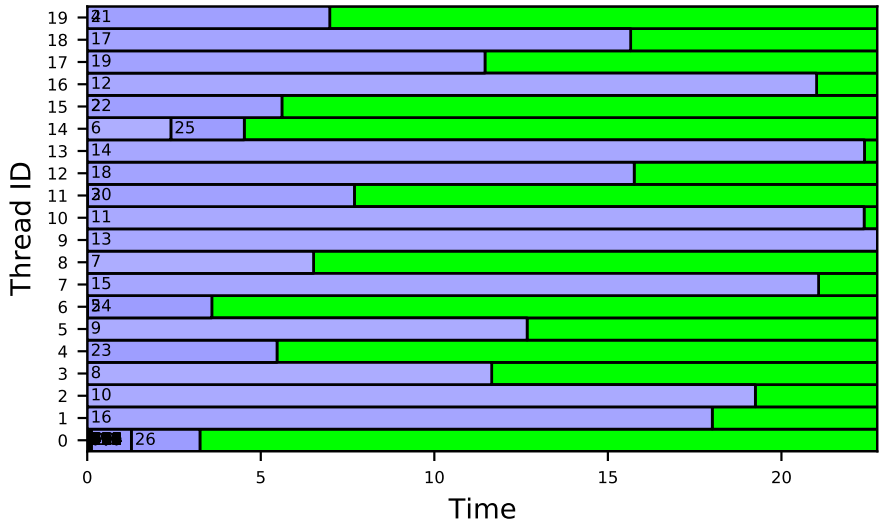

Figure 11: Linear taskloop with Guided scheduler

4.5 Standard LLVM and Vector task stealing

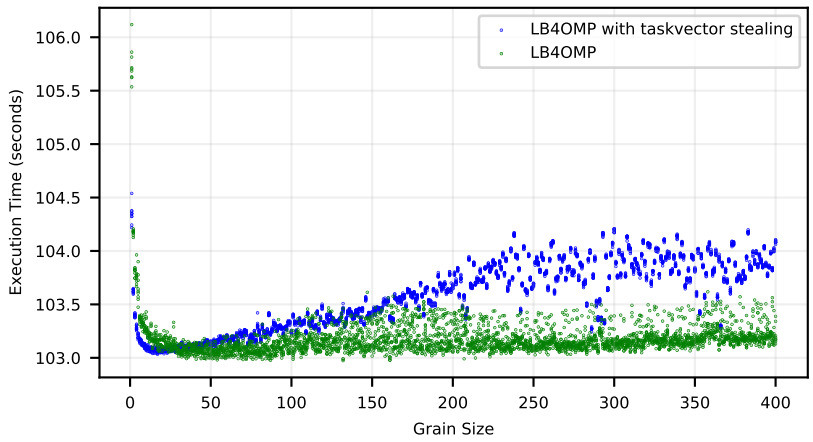

Figure 12: Program execution using standard library (left) and task vectors (right)

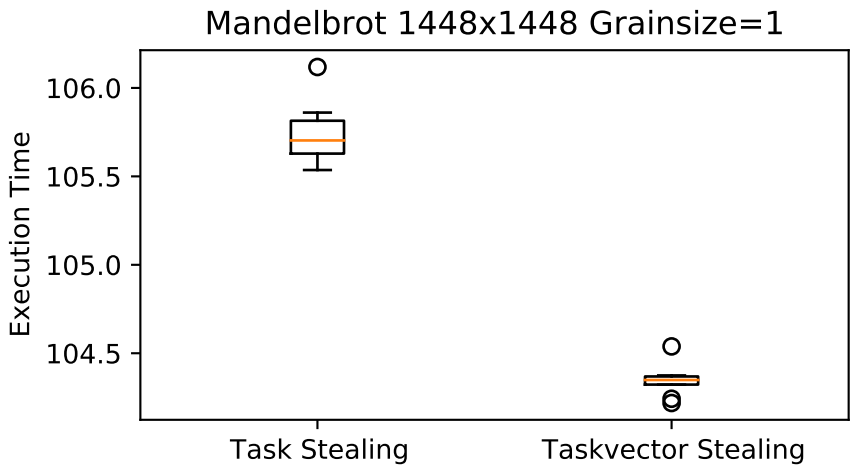

Figure 13: Execution time distribution for grain size 1

5 Discussion

5.1 Overhead, affinity, and load imbalance

It has long been known that picking a grain size is a battle between having too many tasks which will result in high scheduling overhead, and picking too large tasks which results in load imbalance at the end of the execution. The proposed solution in this paper is to take a grain size of 1, and then schedule the tasks in "chunks" where the size can be adapted during the execution of the program. As seen in figure 12, this concept was in part successful since the task scheduling overhead was reduced when there was a large amount of tasks. The problem that remains is that there is an increase in execution time when there are a small number of tasks.

In the graph, the break-even point can be seen at around a grain size of 30. In order to generalize this to other programs, the total execution time would have to be normalized. This would indicate that if the average task switching time remains constant when using the same system and the same number of threads, the break-even point between scheduling overhead and load imbalance is at approximately 680 tasks/second, or in other words 1.5 microsecond tasks on this system.

It is important to note that finding this break-even point required an exhaustive search of all reasonable grain sizes. Especially for programs with a very long runtime, it becomes infeasible to do this exhaustive search of grain sizes. In this case, the program can be run with a grain size of 1 and the taskvector stealing technique enabled (Figure 13).

5.2 Increasing task affinity without downsides

Aside from the more efficient task scheduling when dealing with a large number of tasks, another advantage of using a vector of tasks when stealing from another thread is the increased task affinity. If a series of tasks were in sequential order, which likely means that they access data in sequential order, they would still access data in sequential order after the tasks have been stolen by another thread. This could greatly decrease the number of cache misses. This was not measured in this experiment but would be an avenue for further research.

5.3 Reduced variance in program execution time

A noteworthy artifact of the program execution times in Figure 12 is that the variance of the execution times decreased when using smaller grain sizes. A possible explanation for this would be that load imbalance has much higher variation than task scheduling overhead does. Load imbalance is often referred to as a factor of the grain size in this paper, however it is very non-deterministic. Figure 14 below shows a simulation of the best-case scenario where all threads complete their last task at the exact same time, while Figure 15 shows the worst-case scenario where a thread starts the last task just before all other threads complete tasks. During a program's execution there is the possibility that all the threads could complete their last task at the exact same time, and there is also the possibility that it has a load imbalance of up to the time it takes for one task to execute. In the experiment with task vectors, the number of tasks in a vector was 1 at the end of the execution, which makes the execution times at least as precise as the execution times from the standard library.

The worst-case scenario in Figure 14 does assume that all tasks have the same execution time which is not the case in this experiment, however the concept can still explain why the variance in runtime increases together with the grain size. In this case the worst case scenario would be the biggest task being the last task that is execution, and the probabilistic distribution would be much harder to model.

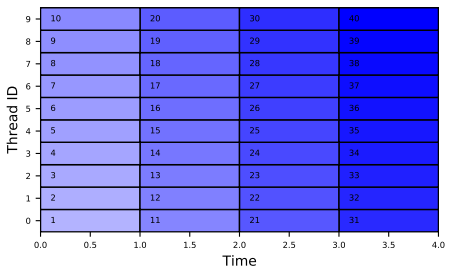

Figure 14: Best-Cast Scenario

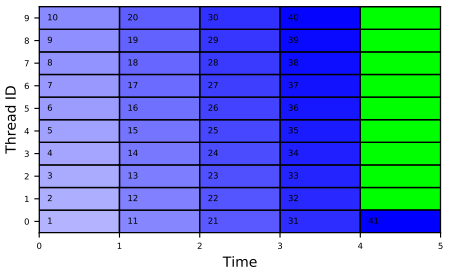

Figure 15: Worst-Cast Scenario

This could prove to be useful, for example if this program were in itself a task in a larger program using MPI, the increased consistency in the program's execution could reduce the load imbalance of the larger program.

5.4 Overhead and load imbalance bathtub curve

Now that we know the overhead incurred by task switching scales linearly with the number of tasks, and that the overhead caused by load imbalance has an upper-bound which is the runtime of the longest task and a lower bound of zero,

we can theretically model the execution time of a program from the grain size as follows.

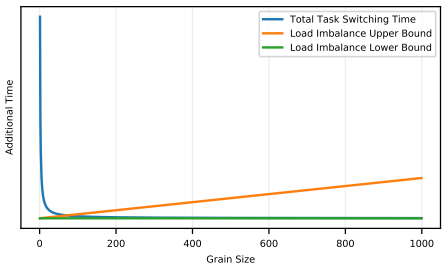

Figure 16: Factors contributing to execution time

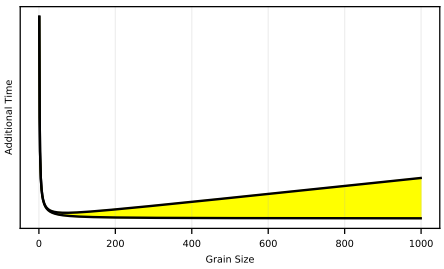

Figure 17: Possible theoretical execution times

6 Conclusion

These experiments have shown that there is a potential advantage to implementing a task stealing function that steals a vector of tasks at once when dealing with programs that generate a large number of tasks (more than 680 tasks/second). In theory, implementing this would also increase the task affinity in some scenarios, however this was not tested in this experiment.

There are other methods to fine tune the scheduling of tasks in OpenMP discussed in this paper, but none of the other methods showed a promising improvement that can be generally applied to all programs that might run on a given OpenMP runtime library. While it is worth knowing these methods, they can be implemented at the program level and would not be viable to implement inside the OpenMP runtime library.

This project also confirms the findings from [4] which is that using OpenMP tasks as they are implemented now will not necessarily improve a program's performance by any significant amount, but it will result in consistent performance when compared to an unknown scheduler.

References

[1] Tim Jammer, Christian Iwainsky, and Christian Bischof. A comparison of the scalability of openmp implementations. In *European Conference on Parallel Processing*, pages 83–97. Springer, 2020.

[2] Jannis Klinkenberg, Philipp Samfass, Christian Terboven, Alejandro Duran, Michael Klemm, Xavier Teruel, Sergi Mateo, Stephen L Olivier, and Matthias S Müller. Assessing task-to-data affinity in the llvm openmp runtime. In *International Workshop on OpenMP*, pages 236–251. Springer, 2018.

[3] RWTH Aachen University Michael Klemm, Intel Corp. Christian Terboven. Advanced openmp tasking. OpenMPCon, 2015.

[4] P. Nussbaum. Exploring opportunities for self-scheduling in parallel multitasking applications. B.S. Thesis, University of Basel, June 2020.

[5] Ahmad Qawasmeh, Abid M Malik, and Barbara M Chapman. Adaptive openmp task scheduling using runtime apis and machine learning. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 889–895. IEEE, 2015.

[6] Christian Terboven, Dirk Schmidl, Tim Cramer, and Dieter Mey. Assessing openmp tasking implementations on numa architectures. pages 182–195, 06 2012.