Load Balancing Framework Comparison

Author
Loek A. van der Gugten
University Of Twente

ABSTRACT

The main goal of this paper is to show how the load balancing frameworks Lace, OpenMP, Intel Thread Building Blocks (TBB) and CilkPlus perform compared to each other. The research focuses on the task-based nature of these frameworks, looking at the overhead of task creation but also at the overall performance of the frameworks. Tests were run using three benchmarks: Unbalanced Tree Search, Fibonacci and Strassen matrix multiplication. The benchmarks were compiled with the Intel Compiler as well as the GNU Compiler to study what consequences this has for the performance. The experiments revealed that out of the four frameworks Lace performs best, CilkPlus and TBB intermediate while OpenMP had the worst performance by far. CilkPlus was a little bit better than TBB. Promising options for future work are discussed.

Keywords

task-based load balancing, load balancing frameworks, openmp, intel tbb, lace, cilk, work-stealing, parallel programming, benchmark

1. INTRODUCTION

Complex programs often require a long time to compute results. Many strategies have been developed to speed up the execution time of these programs. One such strategy is parallel programming, making use of multiple processors or processor cores. Some programs can be modeled as task-based computations, where tasks rely on independent subtasks. Independent (sets of) tasks are executed on separate processing units simultaneously. This means that the tasks do not have to wait for other tasks to be executed. Depending on the number of tasks and their size, as well as the number of available processing units, this could yield a great reduction in execution time. This process of distributing a program over processing units in the form of tasks is called task-based load balancing. Many strategies for task-based load balancing exist [1]. With multicore processors becoming more common [2] efficient load balancing strategies are more important than ever.

When executing a program in parallel, multiple workers (i.e. threads) work on separate subtasks of the initial task simultaneously. Once a worker has finished a subtask, it moves on to its next one. However when it has no tasks left, it will simply become idle. Having the worker that has finished all its tasks remain idle is a waste of that workers potential. To increase efficiency, a form of dynamic load balancing is required. One such method is called work-stealing. With work-stealing the idle worker attempts to steal a task from another worker. This way the optimal distribution of tasks can be achieved even if at the start they have been distributed differently. A theoretical analysis of the Work-stealing strategy has shown that it is an efficient strategy for task-based parallel execution [3].

For the programming languages C and C++, multiple frameworks exist that allow for a programmer to define tasks for parallel execution. The framework makes sure that the code is compiled in such a way that it distributes the tasks among the processing units during execution. Each framework has its own strategy for load balancing, meaning that there can be differences in performance between the frameworks. One such framework is CilkPlus, which has shown good performance relative to other frameworks before [4]. However CilkPlus has been deprecated by Intel and GNU in 2018 [5], making it extra interesting see if it performs better than Intel's alternatives OpenMP and TBB, which are not deprecated. This paper aims to compare four existing frameworks: Lace, OpenMP, Intel Thread Building Blocks and CilkPlus, to get an indication of how much each framework reduces execution time. In particular, the following questions are investigated:

- What is the difference in performance between the different load balancing frameworks?
- What is the difference in performance when compiling with different compilers?
- What is the difference in performance when compiling with different optimization flags?

The tests are all written in C and C++. They are compiled using the Intel C++ Compiler (icc and icpc) and the GNU Compiler Collection (GCC). For each framework, a test is compiled a total of six times; once for each optimization flag (O1, O2 and O3) for both compilers. The executables are then executed on a processor with four cores and eight threads on a system running Ubuntu 18.04 LTS.

This paper will first look into some related work in Section 2. Then the Approach is described in Section 3, explaining the chosen benchmarks, compilers, frameworks and testing setup. Section 4 contains the results of the tests, followed by a discussion in Section 5. Finally, the conclusion can be found in Section 6 plus options for future work.

33rd Twente Student Conference on IT July 3rd, 2020, Enschede, The Netherlands.

2. RELATED WORK

In 2009 a task suite was developed containing benchmarks for task parallelism in OpenMP [6]. The task suite, called Barcelona OpenMP Task Suite (BOTS), contains a variety of benchmarks such as Fibonacci and Merge Sort intended to test the effectiveness of OpenMP task parallelism implementations. Over the years changes have been made to OpenMP, so the results from back then can not be applied to the current version of OpenMP. The open source benchmarks from the task suite however form a good resource when designing such benchmarks yourself.

Comparisons between multiple existing load balancing frameworks have already been made in the past. In 2010 research has been conducted into a comparison between Cilk++, OpenMP 3.0 and Wool [7]. This research has shown that four OpenMP 3.0 compilers perform significantly worse than Cilk++ and Wool for the tested fine-grained parallel tasks. Additionally, it showed that Wool outperforms Cilk++ for all tested fine-grained tasks.

In the same year research has been carried out to the performance of OpenMP 3.0, Cilk, Cilk++ and Intel Thread Building Blocks specifically on Unbalanced Task Graphs (UTS) [4]. The results showed that Cilk, Cilk++ and Intel Thread Building Blocks outperformed all OpenMP 3.0 implementations.

Sandia National Laboratories analyzed six load balancing frameworks in 2012: Intel CilkPlus, Intel Thread Building Blocks, Intel and GCC implementations of OpenMP, Qthreads and High Performance ParalleX (HPX) [8]. They created four benchmarks including Fibonacci and Unbalanced Tree Search that create fine-grained tasks, because they focused on the scheduling strategies of the frameworks. From their results they concluded that no single framework is optimal in all cases, since they found that frameworks that perform well on one benchmark usually do not perform well on another benchmark.

Another comparison was made by Chun-Kun Wang in 2017 [9]. He investigated the scheduling strategies of OpenMP 3.0, Cilk and High Performance ParalleX (HPX-5) and tested them on six benchmarks. Wang found that Cilk and OpenMP have higher overhead with task creation compared to HPX-5. Additionally he stumbled upon the problem that Cilk has a fixed spawn depth limit for tasks, causing Cilk to abort on the benchmarks that spawn large amounts of tasks.

Researchers from Oakland University compared multiple threading parallel programming models for high performance computing, such as Intel CilkPlus, Intel Thread Building Blocks, OpenMP and PThreads [10]. They investigated the features of all selected models, followed by a performance comparison of OpenMP, CilkPlus and C++11 using their own benchmarks. They show results for up to 32 cores, but there is no clear winner.

There has also been research into the effect of different C compiler optimization options on the performance of parallel programs [11]. Researchers at the Federal University of Pelotas, Brazil, compared the performance of five frameworks, including OpenMP, CilkPlus and TBB. They compiled a benchmark from NASA using the Intel C Compiler and the GNU C Compiler, for the three optimization flags O1, O2 and O3. Their results showed that flag O2 from the Intel C Compiler had the biggest impact on performance. They also discovered that the impact of the optimization flags can differ per architecture.

A comparison between OpenMP 4.0 implementations in C/C++ compilers has also been performed. Compilers AOCC, Clang, G++, Intel C++, PGC++ and Zapcc were included in research from Colfax in 2017 [12]. This research only covered the implementation of OpenMP 4.0 of the aforementioned six compilers. It showed that the Intel C++ compiler generally performed best in the conducted tests.

3. APPROACH

This section describes the approach for the load balancing framework comparison. It explains the benchmarking algorithms, compilers, frameworks and testing environment used in this research.

3.1 Benchmarks

As mentioned in Section 2, benchmarks for the selected frameworks have been developed before. The Barcelona OpenMP Task Suite (BOTS) [6] is a good example containing twelve different benchmark algorithms developed for OpenMP under the GNU General Public License Version 2. The BOTS paper shows some characteristics of each benchmark such as the potential number of tasks they generate and the average amount of arithmetic operations per task, that are useful in determining which benchmarks to use.

Each benchmark selected for this research targets a specific aspect of load balancing. The first aspect is the overhead from creating and running tasks. This can be evaluated by defining a benchmark that generates many tasks, and only tasks with a small workload, i.e. many fine-grained tasks. With fine-grained tasks the workers have to spend only little time working on the tasks, so a larger portion of the execution time is determined by the overhead. By having many of these tasks, the execution time increases such that the amount of overhead stands out even more. Whether a benchmark has many fine-grained tasks can be checked in the results from the BOTS [6]. It shows for each benchmark the amount of arithmetic operations per task, indicating the workload of each task. It also shows the number of potential tasks per benchmark.

The second aspect is dealing with indeterministic tasks. This tests the effectiveness of the dynamic load balancing strategy of the frameworks. Consider a benchmark with many fine-grained tasks. If all tasks have similar workload and all tasks spawn the same amount of subtasks, the effectiveness of the dynamic load balancing strategy does not affect the execution time that much. After the first few tasks have been distributed equally among all threads, very little tasks will have to switch threads, since the original tasks have similar workloads and a similar amount of subtasks. However if the tasks have a different, unpredictable amount of subtasks and workloads, dynamic load balancing is important, since at the beginning of execution the amount of time a task will take can not yet be determined (i.e. is indeterministic). During execution some threads will end up with much less work than other threads, meaning they need to take over some tasks from other threads to prevent becoming idle. To test the dynamic load balancing implementation of each framework a benchmark with indeterministic tasks is required, but also a benchmark with equally distributed tasks, so that we can check if a difference in performance for the indeterministic benchmark is also present for the deterministic benchmark. If the difference is present for both benchmarks, then it is probably not caused by a difference in the dynamic load balancing implementation.

The third aspect is tasks with a large workload, also known

as coarse-grained tasks. In practice, many parallel programs have coarse-grained tasks rather than fine-grained tasks. This means that usually the overhead has a small impact on the execution time. To ensure that this common use case of the frameworks is not left out, a benchmark with coarse-grained tasks is required. Additionally, to check whether a difference in performance with fine-grained tasks can be attributed to larger overhead, a benchmark with coarse-grained tasks is required as a sort of negative control. If the framework shows similar bad performance with the coarse-grained task, the cause of bad performance with fine-grained tasks can not be explained by large overhead only, since that should have a much smaller impact on the performance for coarse-grained tasks.

The chosen benchmarks aim to address the aspects described above. They are adaptations from the public benchmarks from the Lace repository [13]. For each benchmark one version has been made for each framework. The code of all benchmarks can be found in [14]. Below a short description of the benchmarks can be found, with a motivation for their use:

Fibonacci: computes the n-th Fibonacci number. Implemented using recursion. Each recursive call spawns a new task. There is an option to specify a cutoff number smaller than n. When a recursive call is made with n smaller than the cutoff, the recursive call will not be parallelized anymore, thus increasing the workload of tasks.

The used implementation is not the most efficient way to compute a Fibonacci number but it is a useful benchmark to test the overhead of task creation for a framework, since each task has a very small workload (only 2.5 arithmetic operations per task) but it spawns a lot of them (40 G potential tasks for Fibonacci 50) [6]. Additionally, each task spawns a consistent amount of subtasks compared to the other tasks, so this benchmark can be used as the check for the indeterministic benchmark. Because Fibonacci is not a complex algorithm, it was also a convenient way to become familiarized with each framework.

Unbalanced Tree Search (UTS): explores an Unbalanced Tree to count the number of nodes. The algorithm was originally created by The Unbalanced Tree Search Project Team for the evaluation of the performance of dynamic load balancing frameworks [15]. It uses shape, depth, size and imbalance parameters to generate trees. To ensure that the same parameters always yield the same tree, a constant seed is used to initialize the pseudo-random number generator. The algorithm explores each parent node depth first, creating a new task for each child node as soon as it is found. With work-stealing enabled a breadth first search perspective is added. Although tasks are not as fine-grained as they are for Fibonacci, they still have a small workload. The difference is that for Fibonacci every task will spawn a consistent amount of new tasks, but for UTS the amount of new tasks is indeterministic (the tree is unbalanced, meaning that each parent node has a random number of child nodes). This causes the threads to have different workloads meaning that some threads will have finished their tasks before the other threads. These threads will have to steal tasks from the other threads, so the *dynamic* load balancing strategy is more important. Because of its tasks with this indeterministic nature and the fact that it has a paper thoroughly analyzing its effectiveness and implementation [15], it has been selected as the second benchmark.

Strassen Matrix Multiplication: multiplies two large square matrices. It partitions the matrices into four block matrices with equal size (requiring the matrices to have equal square dimensions dividable by sixteen). For each block matrix a new multiplication task is spawned. Each task performs seven matrix multiplications on the partitioned block matrices, thus spawning seven subtasks that will then repeat this process on the smaller matrices.

A cutoff is used to prevent the algorithm from parallelizing for too small-sized matrices, such that the created tasks have a larger workload. This larger workload per task is a more realistic use case of parallel programming. It also means that the benchmark focuses less on the overhead of task creation. Additionally, as mentioned before each task spawns seven subtasks until the cutoff is reached. Therefore the generated tasks have a very consistent workload, such that dynamic load balancing does not play that much of a role in the performance as it does for UTS.

3.2 Compilers

To compile the code, two compilers were used. The first one is the Intel C Compiler (ICC) and Intel C++ Compiler (ICPC) version 19.1.1.217 from Intel® Parallel Studio XE 2020 Update 1 for Linux, installed with a student license. This compiler has been selected because it implements CilkPlus and OpenMP, and it includes the TBB library by default. Additionally TBB and CilkPlus are implemented by Intel, so it makes sense to include the Intel Compiler to test them. On top of that research into the performance of C/C++ compilers has shown that the Intel Compiler performs best of all compilers when compiling OpenMP version 4.x code [12].

The second compiler is the GNU Compiler Collection (GCC) version 7.5.0. Note that the Intel Compiler version is compatible with this GCC version. GCC is a widely used compiler for C and C++ code, and also performs well compared to other C/C++ compilers [12].

Both compilers provide optimization flags to enable optimization features for the compiled code, such as the flags -O1, -O2 and -O3 [16]. To determine if, and if so what difference it would make to choose a different optimization option for parallelized code, all benchmarks have been compiled three times per compiler, once for each optimization flag -O1, -O2 and -O3.

3.3 Frameworks

As mentioned in the introduction, the four Load Balancing Frameworks that will be compared are Lace, CilkPlus, Intel Thread Building Blocks and OpenMP. Each framework is listed below with a brief description and a simple example of a use case of the framework:

```
21  TASK_1(int, fiblace, int, n)
22  {
23    if (n < 2) return n;
24    int i, j;
25    SPAWN(fiblace, n-1);
26    j = CALL(fiblace, n-2);
27    i = SYNC(fiblace);
28    return i+j;
29  }
```

Figure 1: Code Snippet of Lace's version of Fibonacci.

Lace: Lace is a load balancing framework library designed at the University of Twente. It uses a non-blocking work-stealing deque based on the split task queue [17]. Lace

```
__int64_t fibcilk(__int64_t n) {
  if (n < 2) return n;
  int x, y;
  x = cilk_spawn fibcilk(n - 1);
  y = fibcilk(n - 2);
  cilk_sync;
  return x + y;
}
```

Figure 2: Code Snippet of Cilk's version of Fibonacci.

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
        n(n_), sum(sum_)
    {}
    task* execute() {      // Overrides virtual function task::execute
        if( n<2 ) {
            *sum = n;
        } else {
            long x, y;
            FibTask& a = *new( allocate_child() ) FibTask(n-1,&x);
            FibTask& b = *new( allocate_child() ) FibTask(n-2,&y);
            // Set ref_count to 'two children plus one for the wait".
            set_ref_count(3);
            // Start b running.
            spawn( b );
            // Start a running and wait for all children (a and b).
            spawn_and_wait_for_all(a);
            // Do the sum
            *sum = x+y;
        }
        return NULL;
    }
};
```

Figure 3: Code Snippet of TBB's version of Fibonacci.

consists of a header file and a source file that have to be compiled and linked with the program that is parallelized using Lace. In Lace, to be able to spawn a function as a task, the function has to be defined as a task with the keyword TASK_N(function name, parameters). The keyword SPAWN(function name, parameters) is used to spawn a defined task and SYNC(function name) to wait for a single task to finish. See Figure 1 for the Lace implementation of Fibonacci.

CilkPlus: CilkPlus is a task-based load balancing framework for C and C++, originating from a 1994 project called Cilk. In 2006 Cilk was commercialized with the creation of Cilk Arts, forming Cilk++. In 2009 Intel took over Cilk Arts, creating CilkPlus. Since 2018, CilkPlus has been deprecated by Intel and GCC, but it still remained embedded in their compilers. CilkPlus is not a library, it has to be implemented by the compiler to be able to use it. An open-source version of Cilk is still being maintained by MIT, the original developers of Cilk [18]. CilkPlus implements work-stealing deques [3]. It also has a work-first policy, meaning that when a worker spawns a new task, it finishes the new task first before continuing with the old task [19]. In CilkPlus any function can be spawned as a task with the keyword "cilk_spawn" in front of the function name. "cilk_sync;" is used to wait for all tasks spawned in the current scope. See Figure 2 for the Cilk implementation of Fibonacci.

Intel TBB: TBB is a library for task-based load balancing in C++ developed by Intel. It also uses work-stealing. The

```
__int64_t fibomp(__int64_t n) {
  if (n < 2) return n;
  int x, y;
  #pragma omp task untied shared(x) firstprivate(n)
  x = fibomp(n - 1);
  y = fibomp(n - 2);
  #pragma omp taskwait
  return x + y;
}
```

Figure 4: Code Snippet of OMP's version of Fibonacci.

TBB scheduler was inspired by the early Cilk scheduler [20]. TBB is already built in with the Intel C++ compiler (icpc) and can be used during compilation with the "-tbb" flag. Since TBB is a C++ library it should work for any C++ compiler, but due to time constraints only the icpc built in version is used. In TBB, a task is defined as a class. To spawn a task an object of its class has to be created and then called in one of the spawn functions, for example "spawn_and_wait_for_all(task_object)". See Figure 3 for the Cilk implementation of Fibonacci.

OpenMP: OpenMP is a parallel programming framework defined by many big computer hardware and software vendors such as Intel and IBM. The implementation of OpenMP differs per compiler, but for GCC and ICC it supports many strategies to parallelize code, including task-based load balancing with work-stealing. [21]. In this research OpenMP version 4.5 is used. Like CilkPlus, OpenMP is embedded in the compiler, so it has to be implemented by the compiler to be able to use it. To spawn tasks with work-stealing enabled in OpenMP, the pragma "#pragma omp task untied" can be used. "#pragma omp taskwait" is used to wait for all tasks in the current scope to finish. See Figure 4 for the OpenMP implementation of Fibonacci.

3.4 Testing Environment

To run the tests a Lenovo Thinkpad `p51 20HH000TUS` with an Intel i7 processor with four cores, eight virtual-cores/threads is used [22]. It runs Ubuntu 18.04.4 LTS. Athough this is not a multiprocessor system, it represents the target architecture of many parallel programs and it still has four cores. For reliability the Intel Turbo Boost technology [23] is disabled on all cores. To ensure that the performance is not limited by power-save settings, the CPU governor is set to performance mode before running tests. Running time is measured by calling the C function "gettimeofday()" from sys/time.h right before and after the execution of the relevant code and then looking at the difference.

To run the UTS benchmark with Intel TBB, the system stack size has to be increased with the command "ulimit -s <value>". The value used was 1,000,000. To run the UTS benchmark with OpenMP the OMP stack size has to be increased as well by putting the keyword "OMP_STACKSIZE=<value>" in front of the execution command. For T3L the value used was 1,000,000.

3.5 Running the Benchmarks

Every benchmark has been run on one, two, four and eight threads, such that the speedup of using more threads can be shown. Furthermore, for every amount of threads, the benchmark has been run five times to remove possible outliers.

The goal of the runs was to get execution times of over one second. Having smaller execution times could mean that small differences in performance are not clearly visible.

Table 1: Average execution time in seconds of all framework & benchmark & compiler combinations with eight threads, for optimization flag O3.

	Fib 45 (No cutoff)		Fib 45 (Cutoff=30)		Strassen 4096		UTS T3L	
Framework	ICC 03	GCC 03	ICC 03	GCC 03	ICC 03	GCC 03	ICC 03	GCC 03
Lace	3.049	1.726	0.885	1.024	4.632	4.550	5.354	5.463
OMP	178.838	1031.591	0.935	1.487	4.586	4.711	18.329	22.794
CilkPlus	13.172	23.240	0.906	1.471	4.302	4.427	segfault	segfault
TBB	39.268	n.d.	0.840	n.d.	4.368	n.d.	7.031	n.d.

Table 2: Fibonacci 45 (No Cutoff) for Lace with all optimization flag-compiler combinations (average time in seconds).

Framework	ICC O3	ICC O2	ICC O1	GCC O3	GCC O2	GCC O1
Lace	3.049	11.740	14.524	1.726	2.444	2.733

However due to time constraints and the fact that every benchmark had to be run five times for every amount of threads for every framework (80 times total per benchmark), the runs could not last too long, so an execution time of 10 minutes was the maximum. The parameters of the benchmarks have been selected with this in mind.

For Fibonacci, 45 was the selected input parameter. It was the highest Fibonacci number that could be calculated by all frameworks in a reasonable amount of time for any amount of workers. For OpenMP with eight workers, Fibonacci 46 already took longer than 10 minutes to finish without cutoff, and with less than eight workers this was expected to take even longer. A cutoff of 30 was chosen because it seemed to lower the execution times the most in the experimental runs. It does not matter that much which number around 30 it is, because it is just intended to change the tasks from fine-grained to course-grained, which would also happen with a cutoff of for example 29 or 31.

UTS was run with the parameters of tree T3L. This results in a tree of size `111345631`, depth `17844` and `89076904` leaves. The input variables were as follows:

- Type: Binomial.
- Root Branching Factor: `2000`.
- Probability of non-leaf node: `20.0014%`.
- Number of Children for non-leaf node: Five.
- Root seed: Seven.

This tree was also used for this benchmark in the development of Lace [17], where it had `158,566` task steals, meaning that the dynamic load balancing strategy is put to the test. Because of the amount of steals this tree causes with Lace it is expected that the runs of other frameworks will require the threads to steal a lot of tasks as well, meaning it is a suitable input tree to test the dynamic load balancing strategy of the frameworks. T3L also executes in anywhere between 5 to 40 seconds for all frameworks, meaning that the time constraints are met as well.

The input parameter for Strassen was 4096. This denotes the size of the two nxn matrices that are to be multiplied. The input parameter was required to be a power of 2 and a multiple of 16 for the algorithm to work, so the closest alternatives were 2048 and 8192. 2048 turned out to be too simple generating too small execution times, and 8192 too complex, so with the time constraints in mind 4096 was used.

4. RESULTS

This section describes the results of the research. Section 5 attempts to explain and evaluate the observations described here. All results can be found as supplementary data in [14].

The results of the runs with eight threads are summarized in Table 1. It contains the average execution time of all frameworks and compilers for all benchmarks. Every benchmark was run five times to filter out outliers. The standard deviation turned out to be very low for all test runs, with the highest being OpenMP's Fibonacci 45 (No Cutoff) with only 0.8%. Therefore the average of the five runs is considered to be accurate enough for a good indication of the execution time. The table displays the results of the tests for the optimization flag O3, since all tests performed best with that flag.

Table 1 shows a clear difference in performance between all frameworks for the Fibonacci and the UTS T3L benchmark. As can be seen in the table, Lace performs best for Fibonacci without cutoff and UTS T3L with both compilers. The difference between the frameworks for Fibonacci with cutoff is very small and for Strassen only OpenMP has a big difference.

What is interesting to see is that OpenMP has bad performance for Fibonacci without cutoff and for UTS. Especially for Fibonacci without cutoff there is an extreme difference in execution time with the other compilers (1.7 seconds for Lace compared to 1869 seconds for OpenMP).

CilkPlus has a hard coded deque limit of 1024 entries, which is exceeded when running UTS T3L [9]. Therefore these tests resulted in a segmentation fault, indicated in Table 1 with "segfault". As explained in Section 3 there are no entries for GCC with TBB, which is why those entries in Table 1 contain n.d. (no data).

As it turned out from the results, there were only two cases in which there was a noticeable difference between different optimization flags. The first was the Intel Compiler for Fibonacci with cutoff, where the switch from O1 to O2 increased the execution time with a little over 50%. The second case was Lace's test run for Fibonacci without cutoff. Here all three optimization flags showed a different timing, for both compilers. Table 2 shows the execution times for this second case.

Figure 5 displays the timings of Strassen for the runs with one through eight threads. It shows that the small differences in timings between the frameworks persist when less threads are used. Additionally, the speedup from four threads to eight threads seems to be close to zero. The same patterns were present in the data of Fibonacci with cutoff, also a benchmark with coarse-grained tasks.

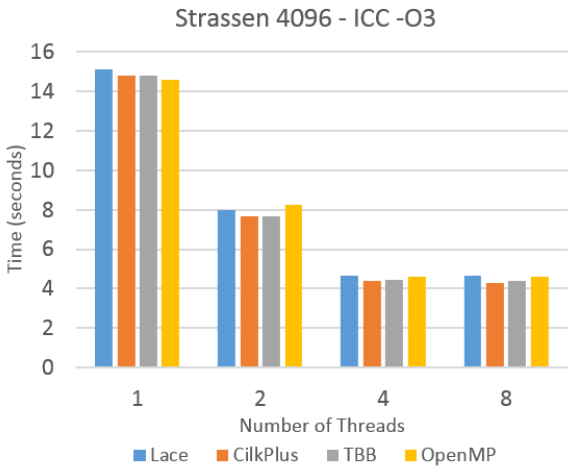

Figure 5: Average execution time in seconds of Strassen for different number of threads.

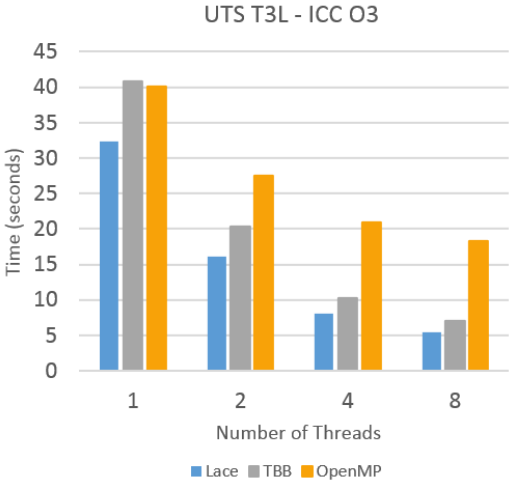

Figure 6: Average execution time in seconds of UTS for different number of threads.

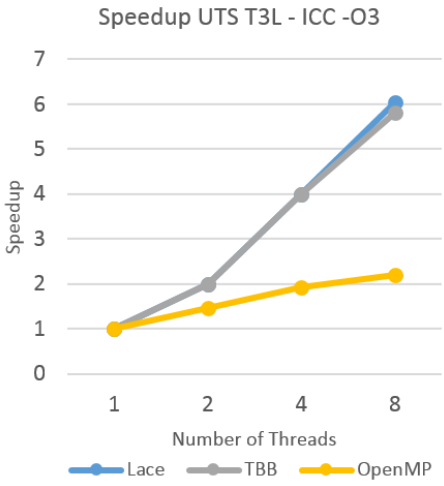

Figure 7: UTS Speedup of each framework based on average execution time.

The timings of UTS for all numbers of threads can be found in Figure 6. It shows how the difference between Lace and

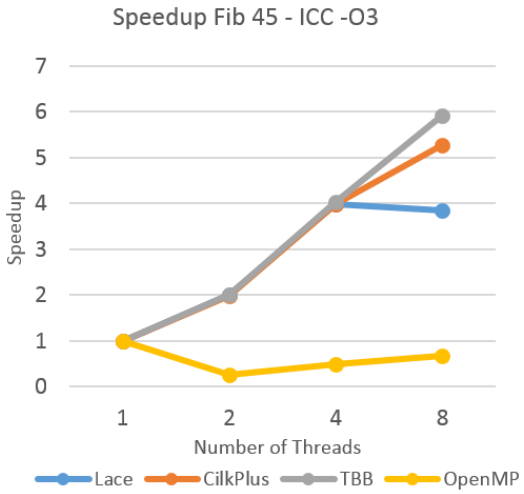

Figure 8: Fibonacci without cutoff Speedup of each framework based on average execution time.

TBB roughly remain the same with a different number of threads, but OpenMP gets closer to the two frameworks when less threads are used. This difference is more visible in the speedup in Figure 7, where Lace and TBB show similar speedup to OpenMP. Speedup is determined as speedup from using a single thread. It is calculated by dividing the timing of the run with a single thread by the timing of the run with n threads, to get the speedup for n threads.

Perhaps the most surprising results followed from running Fibonacci without cutoff with less threads. OpenMP turned out to have a lower execution time when running on a single thread, with 157 seconds versus 179 seconds with eight threads. This behaviour, displayed in Figure 8, shows that the timings have actually deteriorated when more than one thread is used. There is however still a speedup from two to four threads and from four to eight threads. Additionally, the CilkPlus and TBB graph show a speedup when switching from four threads to eight threads, but Lace does not speed up there. This is interesting because CilkPlus and TBB did not show this speedup from four to eight threads with the Strassen and Fibonacci with cutoff runs, for which their graphs were similar to the graph of Lace.

5. DISCUSSION

The main goal of this paper was to show how the load balancing frameworks Lace, OpenMP, Intel Thread Building Blocks and CilkPlus perform compared to each other.

The first research question was: What is the difference in performance between the different load balancing frameworks? The results show that Lace has lower execution times than the other frameworks for two benchmarks (Table 1). The biggest difference is in Fibonacci without cutoff, but there is also a big difference in the UTS benchmark. As explained in Section 3.1 Fibonacci spawns many fine-grained tasks, testing the overhead of task creation. Since there is no big difference between the frameworks when running Fibonacci with cutoff and Strassen where the tasks are more coarse-grained, the good performance in Fibonacci without cutoff likely means that Lace adds less overhead with task creation compared to the other frameworks.

As also explained in 3.1 UTS tests the dynamic load balanc-

ing strategy of the frameworks. The low execution time for Lace in UTS can therefore indicate that Lace has a more efficient dynamic load balancing strategy (work-stealing) than the other frameworks. However it can not be said with certainty that this is the only cause for the difference in performance since UTS also spawns fine-grained tasks, which means that overhead could also play a role.

In contrast to Lace, OpenMP does not perform well for Fibonacci and UTS. Especially for Fibonacci, OpenMP had very long execution times. This most likely means that OpenMP has a lot of overhead on task creation. Since the difference in performance is much less significant for UTS than it is for Fibonacci, it is not yet certain that OpenMP has a bad dynamic load balancing strategy, since the difference could also be caused by the large overhead.

TBB performed average relative to the other frameworks. Although for Fibonacci and UTS its execution times were higher than Lace, they were still closer to those of Lace than those of OpenMP. CilkPlus was in between TBB and Lace for Fibonacci and UTS, and it was similar to the other frameworks for Fibonacci with cutoff. CilkPlus is not suitable for programs that create a large amount of tasks (because of its hardcoded spawn depth limit) as was noticed from the "segfault" entries for UTS in Table 1. Why there was no segmentation fault for Fibonacci even though Fibonacci handles more tasks could be explained by Cilk's work-first policy [19]. This policy causes threads in Cilk to execute a newly spawned subtask first before continuing with the task that spawned that subtask. For Fibonacci this means that Cilk has a depth-first approach, meaning that with eight threads and the Fibonacci implementation in Figure 2, a maximum of `43 + 41 + 40 + 39 + 39 + 38 + 38 + 38 = 316` tasks can be generated before the first thread has reached the depth (i.e. `n<2`) of its task. Since at that point all currently running tasks must have reached their maximum depth, and because of the work-first policy each task that finishes goes back to the task that had spawned it, new tasks are spawned at a slower pace than currently existing tasks finish. Therefore the spawn depth limit is never reached.

Based on the results Intel's choice to deprecate CilkPlus is questionable. Although TBB performed similar to CilkPlus for the two benchmarks with coarse-grained tasks, TBB did not perform *better* than CilkPlus, and TBB even performed worse on the fine-grained and indeterministic benchmarks. On top of that TBB cannot be used in C, while Intel's alternative for use in C, OpenMP, clearly performs worse than CilkPlus.

The speedup of each framework is also interesting. The speedup graphs following from running the tests with different numbers of threads show that for the coarse-grained tasks all frameworks have similar speedup. Additionally, the speedup seems to correspond to the amount of threads that are used; two threads leads to a speedup of close to two and four threads to close to four. This makes sense because having two threads instead of one means that each thread only has to do half the work, given that the load balancing is performed optimally. Strangely enough for the same benchmarks the speedup does not increase when going from four to eight threads (and for some it even lowers by a small amount). This can most likely be explained by the fact that the processor that was used to run the benchmarks had eight virtual-cores or threads, but only four physical cores, meaning that when eight threads were used, four of those eight could not actually run in parallel. Although this does seem like a plausible theory, the speedup graphs from UTS and Fibonacci without cutoff contradict it, since they still show a big increase in speedup from four threads to eight threads. It is unclear if this can be attributed to more virtual cores handling the overhead of task creation better, or maybe to more virtual cores handling dynamic load balancing better, or even something else.

Another interesting observation from the speedup graph of Fibonacci without cutoff is that OpenMP has a speedup of lower than 1.0 when using more threads, meaning it was slower for OpenMP to use more threads. This could mean that OpenMP has an inefficient scheduler, since Fibonacci spawns many tasks that all have to be scheduled for the active threads, however further investigation would be required to confirm this.

The answer to research question two, about the performance of the Intel Compiler versus the GNU Compiler, did not become very clear from the results. Lace performed better with the GNU compiler for Fibonacci without cutoff, while on the other hand OpenMP and CilkPlus performed much better with the Intel Compiler for the same benchmark. The execution time for Fibonacci with cutoff was shorter for all frameworks with the Intel Compiler, although for Lace the difference was not that large. For Strassen there was hardly any difference between the two compilers. UTS T3L showed OpenMP once again performing better with the Intel Compiler, while for Lace there was no clear difference in performance. These characteristics were consistent for the different numbers of threads.

To get a better indication of which compiler is best for which framework, it would be useful to run sequential tests, i.e. benchmarks without parallelization that, apart from the task-based load balancing, have the same behavior as the parallel benchmarks. The Intel and GNU Compiler probably already have differences in performance for the sequential benchmarks, which therefore to some extent causes differences in performance for the parallel benchmarks. Based on the results of the sequential benchmarks a relative speedup can be determined such that the initial difference for the sequential versions does not influence the results. Since Fibonacci shows better performance with GNU for Lace, but with Intel for CilkPlus and TBB, there is an interesting difference for future work. If it turns out that a framework actually performs better with one of the compilers, it could be interesting to look at what the compilers do different that causes that compiler to perform better.

Research question three was: What is the difference in performance when compiling with different optimization flags? As described in Section 4 most Benchmark-Compiler combinations did not result in a noticeable difference between the compiler flags O1, O2 and O3. It also has a similar problem to research question two. Although all timings usually decrease when a better optimization flag is used, this does not necessarily mean that the optimization flag affects the performance of the mechanics of a load balancing framework. If the same difference is present when running sequential tests, then the mechanics of the frameworks are not affected by the optimization flag, but rather the sequential code. Therefore this also requires future work.

A few other studies mentioned in 2 have already explored differences in performance between task-based load balancing frameworks and compilers implementing the frameworks. The comparison from Podobas et al. [7] has shown that Cilk++ performed better than OpenMP 3.0 on fine-grained tasks. This is in line with the results in this paper

for Fibonacci without cutoff in Table 1.

The comparison from Olivier et al. [4] showed that Cilk, Cilk++ and Intel Thread Building Blocks all outperformed the OpenMP 3.0 implementations. This is again in line with the execution times seen in the results for Fibonacci and UTS.

Sandia National Laboratories compared CilkPlus, TBB and OpenMP [8]. They concluded that although there were differences in performance, none of the frameworks could be considered the best, since every framework that did well on one benchmark, scored worse than the others on a different benchmark. My results are not in line with this observation, since the frameworks that performed well on Fibonacci and UTS did not perform worse than the others on Strassen. This could however change when my benchmarks are run on a system with more cores.

The research focusing on optimization flags [11] discussed that the switch from O1 to O2 had the biggest impact on performance. Although for most benchmark-compiler combinations in my research there was no noticeable difference between the optimization flags, the case where there was a difference (Fibonacci with cutoff for Intel Compiler) shared the observation that the biggest impact was from O1 to O2.

The compiler comparison performed by Colfax [12] showed that the OpenMP version 4.x code performs better when compiled by the Intel Compiler than when compiled by the GNU compiler. My results showed that for OpenMP version 4.5 the code compiled by the Intel Compiler also performed better than the code compiled by the GNU compiler.

6. CONCLUSION

This paper shows the performance of the load balancing frameworks Lace, OpenMP, Intel TBB and Cilk compared to each other. Tests are run using three benchmarks: Unbalanced Tree Search, Fibonacci and Strassen matrix multiplication. The benchmarks were compiled with the Intel Compiler as well as the GNU Compiler to study what consequences this has for the load balancing framework performance. The experiments revealed that on the overhead of task creation and the efficiency of the dynamic load balancing implementation Lace has the lowest execution times, CilkPlus and TBB score intermediate and OpenMP has the worst performance. CilkPlus was a little bit better than TBB. Speedup-wise Lace, CilkPlus and TBB performed similarly for UTS and Fibonacci, with the exception for the eight-threads runs. The difference there can most likely be explained by the fact that there were only four physical cores and the remaining four were virtual cores. OpenMP showed smaller speedups than the rest for Fibonacci and UTS. The results for Fibonacci suggest that Lace has the least overhead on task creation, followed by CilkPlus, then TBB and then OpenMP. The results from UTS suggest that Lace has the best dynamic load balancing strategy, followed by TBB and then OpenMP. For the benchmarks with coarse-grained tasks all frameworks had close to equal execution times and speedups.

No consistent difference was found between the Intel Compiler and GNU Compiler. For some compiler-benchmark combinations the Intel Compiler showed better results, whereas for other combinations the GNU compiler performed better. Future work is required before reliable conclusions can be drawn regarding with which compiler a framework performs better.

As expected, using a higher optimization flag yields better performance (albeit only slightly better) for nearly all test runs. It can not be excluded, however, that this is purely caused by optimizations in the sequential components of the benchmarks. To determine whether the mechanics of the frameworks are also affected by the optimizations of the optimization flags, future work is required.

6.1 Future Work

Some relevant concepts have not been explored yet due to the limited time for this project. One such thing is running the benchmarks on multiprocessor systems with more available cores. This way it can be tested whether the observed increase in speedup in this research will be continued when more than 8 threads are used.

It would also be interesting to test the frameworks with more benchmarks to cover more problem domains and scenarios. Examples of such benchmarks are are Alignment and SparseLU from the Barcelona OpenMP Task Suite [6] which had even more coarse-grained tasks than Strassen.

Monitoring performance with tools such as Linux's "perf" could provide new insights into lower-level events that occur during execution. Furthermore a study into the generated executables could show how the frameworks are interpreted by the compilers, which helps in finding the cause of differences in performance and may lead to new strategies to improve performance.

References

[1] Marc Willebeek-LeMair and Anthony P. Reeves. "Strategies for dynamic load balancing on highly parallel computers". In: *IEEE Transactions on Parallel and Distributed Systems* 4.9 (1993), pp. 979–993. DOI: 10.1109/71.243526.

[2] Bryan Schauer. "Multicore Processors - A Necessity". In: *ProQuest Discovery Guides* (Jan. 2008).

[3] Robert D. Blumofe and Charles E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *J. ACM* 46.5 (1999), pp. 720–748. DOI: 10.1145/324133.324234.

[4] Stephen Olivier and Jan F. Prins. "Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs". In: *International Journal of Parallel Programming* 38.5-6 (Oct. 2010), pp. 341–360. DOI: 10.1007/s10766-010-0140-7.

[5] B. Hansang. *Intel Cilk Plus is being deprecated*. Sept. 2017. URL: https://community.intel.com/t5/Software-Archive/Intel-Cilk-Plus-is-being-deprecated/td-p/1127776.

[6] Alejandro Duran et al. "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP". In: *ICPP*. IEEE Computer Society, 2009, pp. 124–131. DOI: 10.1109/ICPP.2009.64.

[7] Arthur Podobas, Mats Brorsson, and Karl-Filip Faxen. "A Comparison of some recent Task-based Parallel Programming Models". In: *3rd Workshop on Programmability Issues for Multi-Core Computers* (Jan. 2010), pp. 47–60.

[8] Kyle B. Wheeler, Dylan Stark, and Richard C. Murphy. *A Comparative Critical Analysis of Modern Task-Parallel Runtimes*. Tech. rep. Sandia National Laboratories, Dec. 2012.

[9] Chun-Kun Wang. "Selection of Parallel Runtime Systems for Tasking Models". In: *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*. Dec. 2017, pp. 1091–1096. DOI: 10.1109/CSCI.2017.190.

[10] Solmaz Salehian, Jiawen Liu, and Yonghong Yan. "Comparison of Threading Programming Models". In: *IPDPS Workshops*. IEEE Computer Society, May 2017, pp. 766–774. DOI: 10.1109/IPDPSW.2017.141.

[11] Roger da Silva Machado et al. "Comparing Performance of C Compilers Optimizations on Different Multicore Architectures". In: *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. Oct. 2017, pp. 25–30. DOI: 10.1109/SBAC-PADW.2017.13.

[12] *A Performance-Based Comparison of C/C++ Compilers*. Tech. rep. Colfax Research, 2017. URL: https://colfaxresearch.com/compiler-comparison/.

[13] Tom van Dijk. *Lace - implementation of work-stealing in C*. Nov. 2016. URL: https://github.com/trolando/lace.

[14] Loek van der Gugten. *load balancing framework comparison*. June 2020. URL: https://github.com/LoekAtMe/load-balancing-framework-comparison.

[15] Stephen Olivier et al. "UTS: An Unbalanced Tree Search Benchmark". In: *LCPC*. Vol. 4382. Lecture Notes in Computer Science. Springer, Nov. 2006, pp. 235–250. DOI: 10.1007/978-3-540-72521-3_18.

[16] João Cardoso, José G. Coutinho, and Pedro Diniz. In: *Embedded Computing for High Performance*. Morgan Kaufmann, June 2017. Chap. 6.3.

[17] Tom van Dijk and Jaco C. van de Pol. "Lace: Non-blocking Split Deque for Work-Stealing". In: LNCS 8806 (Aug. 2014), pp. 206–217. DOI: 10.1007/978-3-319-14313-2_18.

[18] Charles Leierson E., Tao Schardl B., and I-Ting Lee A. *Cilk Hub*. 2018. URL: http://cilk.mit.edu/.

[19] Yi Guo et al. "Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism". In: *IPDPS*. IEEE Computer Society, May 2009, pp. 1–12. DOI: 10.1109/IPDPS.2009.5161079.

[20] Alexey Kukanov and Michael J. Voss. "The Foundations for Scalable Multi-Core Software in Intel® Threading Building Blocks". In: *Intel Technology Journal* 11.4 (Nov. 2007). DOI: 10.1535/itj.1104.05.

[21] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. "Evaluation of OpenMP Task Scheduling Strategies". In: *IWOMP*. Vol. 5004. Lecture Notes in Computer Science. Springer, 2008, pp. 100–110. DOI: 10.1007/978-3-540-79561-2_9.

[22] *ThinkPad P51*. Tech. rep. Lenovo, 2020. URL: https://psref.lenovo.com/Detail/ThinkPad/ThinkPad_P51?M=20HH000TUS.

[23] *Intel® Turbo Boost Technology 2.0*. Tech. rep. Intel Corporation, 2020. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.