A Comparative Study about Task Parallelism in OpenMP and Cilk

Guilherme Rezende Alles, Lucas Mello Schnorr
Institute of Informatics – Federal University of Rio Grande do Sul
Caixa Postal 15.064 – CEP 91.501-970 – Porto Alegre – RS – Brazil

***Abstract*—In this paper, we explore task parallelism as an approach to parallel programming. More specifically, two parallel frameworks - OpenMP and Cilk - are compared to one another with respect to their support regarding task parallelism. In order to do that, we will use an algorithm that provides not only task independency but also load imbalancing issues, to assess how well different parallel frameworks handle the resources available to them. The benchmark of choice for this study is the Mandelbrot set calculation, which is a classic problem for irregular workloads. By the end of this study, we seek to have a better understanding of the implementation of each runtime and also conclude which one offers the best performance of the two.**

I. Introduction

With the massive popularity of parallel processors and the stagnation of single core performance due to energy and heat dissipation constraints, the focus on parallel algorithms and parallel execution flow now demands programmers to worry about the parallel execution of algorithms, in order to increase performance and/or efficiency in modern chips. Consequently, the number of frameworks available that enable parallel software development has grown a lot, varying in many aspects such as complexity of development, structure, support for parallel programming paradigms, efficiency and performance. Thus, simpler parallel programming models are of major interest among software developers.

In this study, we compare two different parallel frameworks that support task parallelism: Intel Cilkplus and OpenMP with tasks. The emphasis of the comparison is on the performance of both frameworks, and we will be using the mandelbrot set calculation as the prime benchmarking algorithm. This algorithm was chosen based on it being a classic irregular workload problem. By implementing a problem of this nature, we are able to identify load balancing issues that may affect performance.

Considering that OpenMP and Cilk constructs for task parallelism present different semantics, we expect to produce different results when comparing them in the same conditions. At first glance, OpenMP presents itself as a lower overhead API (compared to Cilk) and, for that reason, should perform better than its counterpart.

The objective of this comparison is primarily to better understand the use cases in which Cilk and OpenMP differ from one another, and conclude which one offers a better tradeoff between performance and ease of use. Another goal is to understand the implementation decisions that were involved in developing each framework and what effects they have in overall performance.

II. Background and Experimental Context

A. Background

OpenMP[1] is a language extension that aims for providing parallel tools for programmers, in the form of compiler directives and a runtime library to command the creation and management of threads. Because OpenMP is fairly extense, it provides a large collection of directives and constructs that enable the programmer to express and explore both task and data parallelism. OpenMP supports a large variety of parallel programming paradigms, such as the SMPD pattern, loop parallelism, the divide-and-conquer paradigm and others.

Cilk[2] is a runtime library that extends the capabilities of programming languages (usually C or C++) to incorporate parallel programming constructs. Contrasting with OpenMP, Cilk's main objective is to provide parallel constructs in a simpler, more intuitive way for the programmer. On these grounds, Cilk is mainly focused on task parallelism and it does so by providing just a few keywords that allow the programmer to express parallelism.

For the sake of comparing the performance of both frameworks, it is important to understand how each one handles the given thread pool and what are the runtime implications of the thread management. While the OpenMP environment allows the programmer to explicitly control a fair amount of settings and runtime behaviours (such as the threads scheduler, shared variables and overall memory model), Cilk keeps this sort of control to its own runtime. Also, expressing parallel sections in Cilk code does not mean that the code will be certainly executed in parallel: the runtime will decide in execution time which parts of the code will be executed sequentially and which parts will not. This is done through a work-stealing routine that tries to avoid parallel execution of very small tasks, that would be executed much faster if done sequentially.

OpenMP, on the other hand, offers much less runtime support, forcing the programmer to explicitly write code that will create threads and a task pool to be executed in parallel. This approach theoretically allows for a lower overhead with the cost of a less intelligent runtime, which will execute everything that is determined by the programmer in parallel.

B. Experimental Context and Workload Details

The experimental design considers the mandelbrot set calculation as the benchmarking algorithm. This algorithm was chosen because we can leverage the irregular workload characteristic to assess how well the schedulers of the two frameworks can distribute work among worker threads, while also maintaining a somewhat closer to the reality scenario (as the majority of algorithms are not completely regular and with static tasks complexities). The Mandelbrot set algorithm was, thus, implemented in the C language, and the runtime settings for both Cilk and OpenMP are left in their default state. The number of threads created by the program is, thus, the amount of hardware threads available in the testing system. We used an R script to create a CSV file with a full factorial design, in which each experiment was replicated 20 times. The factors that are present in the design are the maximum number of iterations per point (which affects the irregularity of the workload) and the granularity of the problem (explained below).

With respect to the granularity of the problem, our Mandelbrot set calculation accepts a variable grain size as a parameter. This means that we can define an experiment with different grain sizes to also assess how well OpenMP and Cilk handle low, medium and high number of tasks in the task pool. This specific parameter is aimed at measuring the overhead involved in creating a task when compared to computing the Mandelbrot point in each framework.

The task creation is handled in the following way: a worker thread is assigned as responsible for creating the task pool. This thread then iterates through the discrete Mandelbrot space, creating tasks of equal grain size (using the constructs #pragma omp task in OpenMP and cilk_spawn in Cilk) and adding them to the pool. Idle threads are, then, able to execute those tasks. A synchronization construct (#pragma omp taskwait in OpenMP and cilk_sync in Cilk) ensures the threads finish the work properly.

The testing environment is a NUMA machine with two Intel Xeon E5-2650 processors, with a total of 32 hardware threads and 32 GB of RAM. No NUMA specific features were explored in this study.

Both implementations (using OpenMP Tasks and using Cilk) produce an image of dimensions 2000x2000, and the programs were kept as equal as possible. The estimated memory consumption for the mandelbrot calculation for this input is of 12 MB. The program was compiled using GCC version 6.1.0 and the optimization flag -O2.

III. Related Work and Motivation

While comparative studies between parallel frameworks have already been done in the past, the general focus of published papers is to present an overview about the performance on different use cases. For instance, [3] presents an analysis of different types of algorithms with different parallel programming paradigms such as divide and conquer and loop parallelization. The parallel frameworks studied in this paper are the Intel TBB, Cilk and OpenMP. Although the comparison between the three frameworks yielded solid results, the scope of this analysis does not allow for it to explore the runtime overhead of each approach, and explain which factors are able to impact performance the most.

[4], on the other hand, presents a comparison of task parallel frameworks using a highly unbalanced benchmark: the Unbalanced Tree Search. While this benchmark is more specific and the benchmark used is suitable regarding load imbalancing, the study lacks an explanation for the results that were observed. In the two papers, the general conclusion is that OpenMP is, in most cases, outperformed by Cilk.

[5] also presents an overall comparison between different task parallel frameworks, using the Mandelbrot set calculation as the benchmarking scheme. This paper provides an overview of the design and implementation of many frameworks (including Cilk and OpenMP), and cites many task parallelism issues such as synchronization and memory models, which were also observed in the current study.

IV. Obtained Results

By executing the Mandelbrot algorithm with a grain of size 1 (i.e. when for every point a task is created), we obtained the result shown in Figure 1.

This graphic presents a behaviour that was not expected at all: when using OpenMP, the execution time of the program was more than 10 times higher than its Cilk counterpart, and the standard error in the results is also very high (between 5 and 7 seconds, on average). Additionally, there seems to be no relation whatsoever between the number of maximum iterations per point. In fact, the execution time is lower when the number of iterations increases, which may indicate that the work is not being equally distributed among threads, and that the time of each individual thread is not being spent with calculation. These results clearly state that something in the OpenMP runtime is preventing the threads to work properly. In order to investigate this matter more closely, we used the Score-P profiling and analysis tools to trace the execution of the OpenMP program, which yielded the plot presented in Figure 2.

By analysing this trace, we can see that the threads spend most of the time blocked by an implicit barrier imposed by OpenMP, which prevents them from calculating the Mandelbrot set properly. Another observation is that the workload is not well balanced among all available threads (as we had assumed before, and this is caused probably by the threads being blocked most of the time.

In order to distribute work more equally between threads and reduce the irregularity of the workload, we benchmarked the program using a grain size equivalent to one line of the Mandelbrot fractal. The results are shown in Figure 3.

As we can see, by increasing the grain size, the behaviour of the program is somewhat normalized. This result can indicate two things at the same time: OpenMP is not able to efficiently handle tasks that are too small and possibly irregular, and also that the excessive amount of tasks can massively degrade the performance of an OpenMP application. This behaviour was

Fig. 1. Average time comparison considering a task for each mandelbrot point

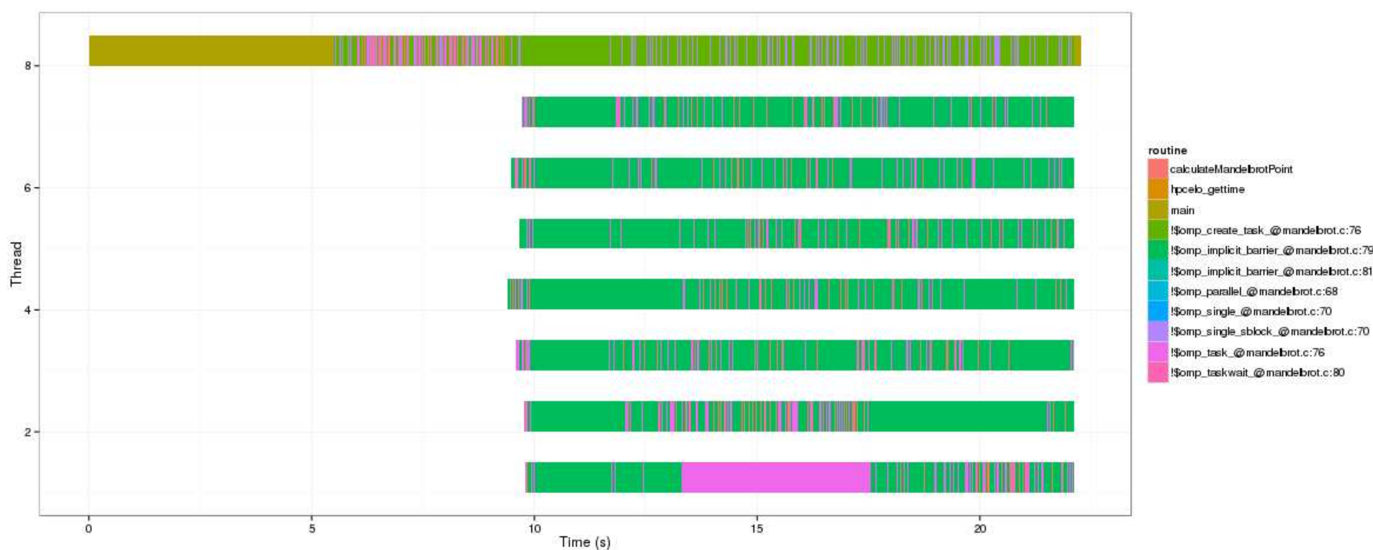

Fig. 2. Trace of the execution of the application using 8 threads

not observed in Cilk, though, probably because of the way that the runtime handles task creation and manages the task pool (i.e. by deciding in execution time whether the tasks will be executed in parallel or not). When the number of tasks created by the program is not too big, both OpenMP and Cilk offer similar performance.

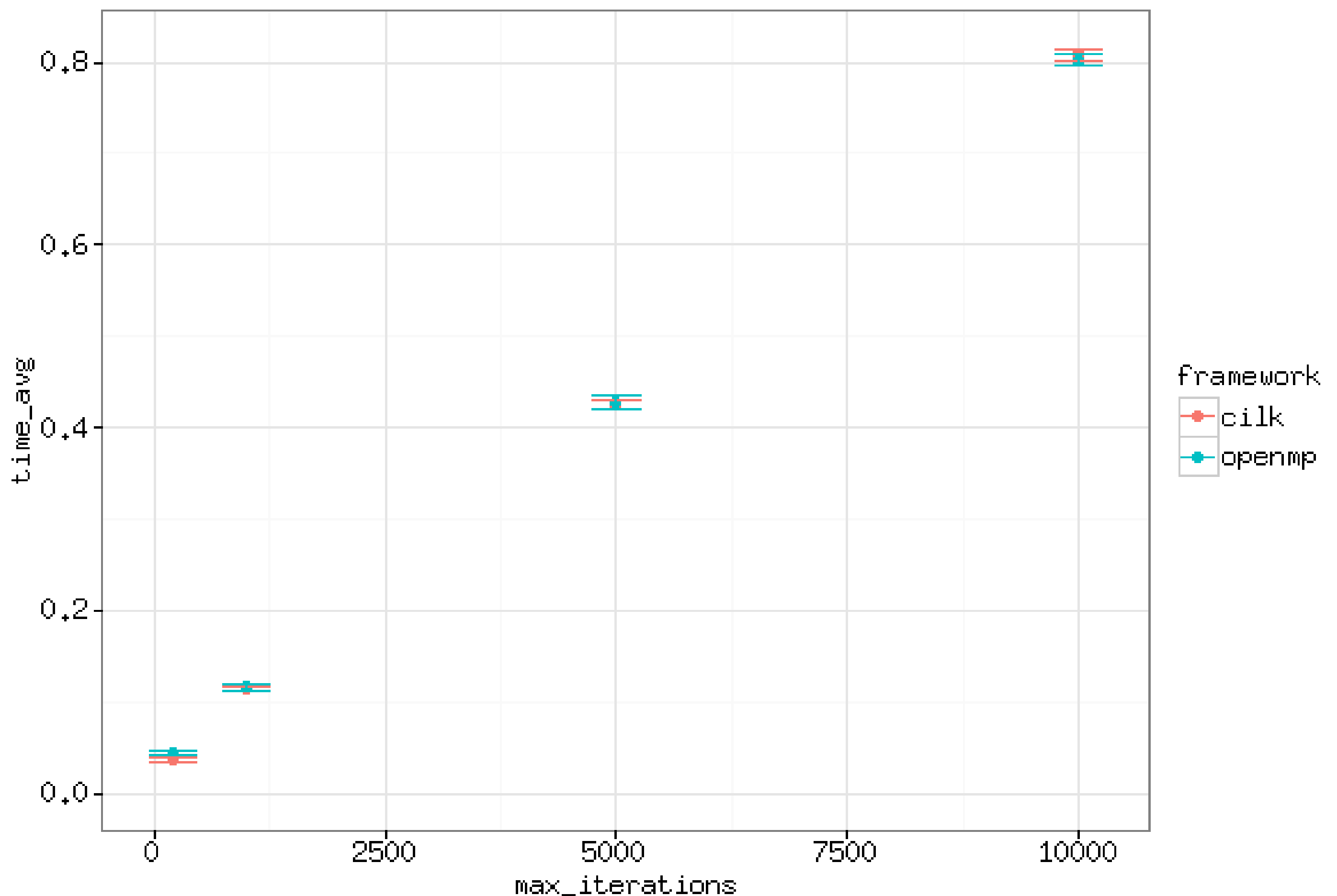

Fig. 3. Average time comparison considering a task for each line of the mandelbrot fractal

V. Conclusion and Future Work

By analysing the performance offered by OpenMP and Cilk when task parallelism is concerned, we were able to identify that the OpenMP runtime is very sensitive when handling the creation of tasks, that is, the programmer needs to be very careful with the task constructs that they insert on the code, because they might be introducing barriers that can ultimately be slowing down the execution of the program. When considering a controlled amount of tasks (which, in this study, was simulated by a coarser grain size), both the performances of OpenMP and Cilk are comparable to one another.

This comparison, however, has proved itself incomplete, especially with regards to the OpenMP barriers that are implicitly inserted in the code, which does not allow for a fair comparison between the two frameworks. As a future study, we seek to understand what is causing these barriers to occur and how to avoid them, in order to increase the overall quality of the experimental design.

Acknowledgements

The results reported in this study were generated in virtue of the agreement between Hewlett Packard Enterprise (HPE) and the Federal University of Rio Grande do Sul (UFRGS), financed by resources in return for the exemption or reduction of the IPI tax, granted by Brazilian Law nº 8248, 1991, and its subsequent updates.

References

[1] "Openmp specification," in *http://openmp.org/wp/*.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Journal of Parallel and Distributed Computing*, 1995, pp. 207–216.

[3] A. Leist and A. Gilman, "A comparative analysis of parallel programming models for c++," in *The Ninth International Multi-Conference on Computing in the Global Information Technology*. IARIA, 2014, pp. 121,127.

[4] S. L. Olivier and J. F. Prins, "Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs," in *Int J Parallel Prog*. Springer, 2010, pp. 341–360.

[5] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin, "Task parallelism and data distribution: An overview of explicit parallel programming languages," in *25th International Workshop on Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2012, pp. 174–189.