Evaluation of Runtime Cut-off Approaches for Parallel Programs

Alcides Fonseca[1] and Bruno Cabral[1]

University of Coimbra, Portugal
{amaf,bcabral}@dei.uc.pt

Abstract. Parallel programs have the potential of executing several times faster than sequential programs. However, in order to achieve its potential, several aspects of the execution have to be parameterized, such as the number of threads, task granularity, stack sizes, etc. In this work we focus on studying the granularity of regular and irregular parallel programs on symmetrical multicore machines. The granularity can be controlled by a loop division factor, or by a cut-off mechanism that stops the parallelization of work and executes the remaining serially. The cut-off mechanism can impact the execution time of the program by several orders of magnitude.
Existing studies have analyzed only two cut-off approaches at a time, each with its own set of benchmarks and machines. In this work we present a comparison of a manual threshold approach to 5 state-of-the-art algorithms (MaxTasks, MaxLevel, Adaptive Tasks Cutoff, Load-Based and Surplus Queued Task Count) and 3 new derivative approaches (Max Queue Size, StackSize and MaxTasks With StackSize). The evaluation was performed using 13 parallel programs, including divide-and-conquer and loop programs, on two different machines with 24 and 32 hardware threads, respectively.
We concluded that the best approach is machine-dependent and program-dependent. Despite not being the ideal conclusion, it is significant to show that existing studies did not prove that one algorithm was better than the others overall. Existing approaches are also not ideal for all cases.

Keywords: Runtime, Cut-off Mechanism, Granularity, Multicore

1 Introduction

Nowadays, making parallel programs faster is a manual process that relies on a lengthy trial-and-error process in order to achieve the best parameters. This process also requires a domain expertise on optimizing parallel programs. Factors like thread or task creation, memory allocation and cache usage are fundamental into obtaining the best performance out of a multicore machine.

Although parallel programs can be more complex, we will focus on two of the most common parallelism patterns: for-loops and recursive programs. Parallelization of for-loops has been the basis of several wide-adopted frameworks,

such as OpenMP[1]. Recursive programs have been the focus of parallelization in other frameworks such as Cilk[2] and ForkJoin[3].

In both patterns, the decision when to create new parallel tasks, or to execute some parts sequentially is a central problem for improving the performance of the program. If the tasks are too coarse, there is still potential parallelism to extract since some of the hardware threads will not be in use. If the parallelism is too fine-grained, too many tasks will be created, imposing an overhead in task scheduling and management that will increase the duration of the execution. Achieving a good balance for all kinds of programs on different machines is thus crucial to achieve a good performance.

In this paper, we will study the most relevant state-of-the-art approaches to control the granularity of tasks at runtime. Alongside these, two new approaches will be analyzed and studied. The goal of the study is to understand how these algorithms perform on parallel programs with different natures and on different machines, in order to understand which one should be used and when.

The remaining of the paper is organized as follows: Section 2 introduces the topic of granularity control; Section 3 details several approaches for controlling the Cut-Off threshold for parallelization; Section 4 evaluates some of those Cut-Off mechanisms; and finally, Section 5 presents the final conclusions of this study.

2 Granularity Control

Parallelizing compilers try to match parallel tasks with the layout of the underlying hardware. The static scheduling divides a loop in N chunks, one for each processor[4]. However, not all programs have this regular and static parallelism. However some programs have a more dynamic behavior, and the number of tasks changes across time. For these programs, runtime-based approaches are needed.

One common approach is to use a work-stealing scheduler, with Lazy Task Creation[5](LTC) as a granularity control mechanism. Potential parallel tasks might be executed inlined, or added to the work queue as a new task, according to different cut-off techniques. These cut-off techniques will be described in Section 3.

This cut-off mechanism to decide whether to parallelize has a great impact on the performance of programs. An OpenMP evaluation[6] has compared two approaches (*maxlevel* and *maxtasks*) and there were differences of up to 3x of speedup. Two other studies, one also within OpenMP[7] and other comparing OpenMP to other approaches [8], have shown differences between the two cut-off mechanisms on unbalanced task graphs. This paper makes a broader evaluation, looking at how this and other decision algorithms perform on a heterogenous benchmark suite.

3 Cut-off Mechanisms

In this section we will introduce and describe several mechanisms for controlling the granularity of tasks at runtime. A cut-off mechanism is an algorithm that

decides whether a task will spawn new tasks for parallel work, or it will execute tasks sequentially.

Programmer-defined cut-off limit - The simplest approach is to have a condition which stops the parallelism, customized by the programmer for a specific program. However, this programmer-defined cut-off requires the developer to have a knowledge of the domain and parallelization method, as well as ability to test the programs several times on the target hardware. With automatic parallelization, this is not possible.

Load Based - This simple heuristic is based on whether all cores are being used or not. A new task is only created if there is at least one idle core[9].

Maximum task recursion level (max-level) - Divide-and-conquer algorithms create tasks in a tree-shaped structure. In order to avoid the creation of too many tasks, the cut-off limit may be defined by the depth of the recursion[6], which can be calculated by the number of ancestors of the running task.

Maximum number of tasks (max-tasks) - In this approach, tasks are created until the total number of active tasks in all worker queues reaches a certain threshold[6]. After that point, all new computations are inlined instead of spawning another thread. When the number of active tasks lowers, new tasks can be created until the threshold is reached again.

The threshold in this approach is typically defined as the number of processor threads on the machine, adapting to different machines, but being oblivious to other factors such as memory and processor speed.

In order to decrease the overhead of computing the size of queues, the size of other queues is estimated from the size of the current queue after applying a factor of (number of idle threads / active threads), because idle threads are known to have 0 tasks in their queue. This estimation assumes a regular distribution among threads, which may not always happen.

Adaptive Tasks Cut-Off (ATC) - Adaptive Tasks Cut-Off[9] changes the policy of the cut-off mechanisms according to the recursion. Tasks are only created if two conditions are met. The first is that there are fewer tasks than the number of threads on a given recursion level. This condition forces the threads to expand in depth, creating work for all threads and being within a certain bound limit. The second condition is that the depth-level is less than a certain threshold. Thus, ATC is the combination of *max-level* and *max-tasks*.

ATC adds a profiler that saves information regarding how much time a subtree takes to execute, and predicts further subtrees (if the prediction is larger than 1ms, the task will be created). This is, however, based on the assumption that all tasks inside a level have a similar behavior, which does not happen in unbalanced parallelism.

Surplus Queued Task Count (surplus) - This approach is included in Java's Fork Join framework[3] and it relies on the size of work-stealing queues. Before creating a new task, the number of queued tasks in the current thread that exceeds the number of tasks in other queues is compared to a threshold limit (usually 3 in existing ForkJoin benchmarks).

Oracle - In Oracle scheduling[10], each recursive call is annotated with a user-defined asymptotic complexity and task depth. This extra-information allows the runtime to predict wether or not it is beneficial to create a new task. The runtime measures the cost of executing each task, saving that information as a numeric constant. The asymptotic complexity function is applied to this constant to predict the relative cost of further calls. In order to adjust to the program execution, this information is retrieved using a moving average of several runs.

Maximum Queue Size (maxtasksinqueue) - We introduce this new approach, which limits the number of tasks in the local queue to a certain threshold. This approach differs from *maxtasks* in only looking at the local queue, instead of all the queues, reducing the time by not accessing information from other threads. If the threshold is one or two tasks higher than the threshold of *maxtasks*, queues will have excedentary tasks that can be stolen by other threads.

Stack Size - In recursive divide-and-conquer programs, the recursivity limit of the platform imposes heavy limitations on the parallelization of programs. Recursive calls are necessary to inline the execution of tasks inside the same worker thread. As such, the performance of programs decreases when the stack grows beyond a certain size.

Having this in mind, we introduced a new approach, which counts the number of stack frames produced at a given moment, and only allows the creation of tasks if that count is lower than a predefined threshold.

Together with this approach, we introduced a hybrid version between MaxTasks and StackSize, *maxtasks-ss*, that first avoids task creation if the number of stack frames is higher than the threshold. If not, the creation of tasks is regulated by the MaxTasks mechanism.

4 Cut-off Mechanism Evaluation

In this section, we begin by introducing the methodology, the experimental setup and the benchmark used. Then, we analyze and compare the different approaches. Finally, we evaluate how they perform with different numbers of workers.

4.1 Cut-off Mechanism Selection

This study intends to evaluate as many cut-off mechanisms as possible. From the ones previously presented, only Oracle has not been evaluated. The Oracle approach required special annotations with domain-specific knowledge of the program, giving it an unfair advantage and not being suitable for automatic parallelization techniques.

4.2 Experimental Environment

The cut-off mechanisms being evaluated were implemented in the Æminium runtime[11]. The Æminium Runtime is the component of the Æminium language

framework responsible for the execution of DAGs on top of the Java Virtual Machine. The Æminium Runtime has a work-stealing scheduler, based on the Java ForkJoin framework[3], with modifications to support atomic tasks and datagroups.

The two machines used were running Ubuntu 12.10 server 64-Bit and OpenJDK 64-Bit Runtime Environment (IcedTea 2.3.10) JVM. The OpenJDK 8 runtime was not used because of a bug that would make memory allocation slower, thus making the execution of programs longer and less accurate. The machines were used for the high number of cores, which enable a high level of parallelism. The different CPU speeds also allows to understand the impact of the CPU speed in task scheduling.

Name	Processor	CPU Cores	Threads	RAM
ingrid	Intel(R) Xeon(R) CPU X5660 @ 2.80GHz	12 cores	24 threads	24GB
astrid	Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz	16 cores	32 threads	32GB

Table 1. Details of the hardware used in the experiments. Note that despite the different RAM limits, programs rarely exceed 20GB of RAM.

Given the large difference between execution times of some programs with some algorithms, each run had a 500 second timeout, at which point the execution was canceled. This was necessary because although all programs run within that value with the manual approach, several programs executed for more than 48 hours before they were initially cancelled.

The execution time of the benchmark for each approach is the sum of the medians of each of the programs in the benchmark. Therefore, the weight of the program is proportional to how long it takes compared to other programs. In the case of time-outs, the considered time is 500s, which imposes a maximum difference between approaches.

4.3 Benchmark Suite

The Æminium Benchmark Suite was used to evaluate the cut-off mechanisms across different parallel programs. The suite aims to include different types of parallelism: divide-and-conquer, both regular and irregular, for cycles and hybrid programs. Table 2 describes the programs used. The benchmark is available at `https://github.com/AEminium/AeminiumBenchmarks`

4.4 Comparison of Cut-off Approaches

Figure 1 compare the results obtained for each algorithm with the best parameters for that machine. For instance, *maxlevel6* refers to the max-level algorithm with 6 maximum levels of recursion. The parameter evaluation for each cut-off was omitted for brevity sake.

Program	Source	Type	Balancing	Input size	Manual Cutoff
Breadth-first Search (BFS)	PBBS[12]	Recursive	Regular	d=23,w=2	depth >= 21
Black-Scholes	PARSEC[13]	For-loop	Regular	10000^2	LBS
Do-All		For-loop	Regular	100 million	LBS
FFT	ForkJoin[3]	Recursive	Regular	8388608	l <= 1024
Fibonacci	ForkJoin[3]	Recursive	Irregular	n=39	n <= 26
Genetic Knapsack		For-loop	Regular	g=100,p=100, p_r=20%,p_m=20%	LBS
Integrate	ForkJoin[3]	Recursive	Irregular	s=-2101,e=200, error=10^{-11}	error <= 10
KDTree	PBBS[12]	Recursive	Regular	n=10000000	depth >= 100
Matrix Multiplication (MM)	ForkJoin[3]	For-loop	Regular	p=10000, q=r=1000	LBS
MergeSort	ForkJoin[3]	Recursive	Regular	n= 100000000	n <= 100
N-Body	PBBS[12]	For-loop	Irregular	n=50000, it=3	LBS
N-Queens	ForkJoin[3]	Recursive	Regular	n=8..15	n <= 8
Pi		For-loop	Regular	n=100.000.000	LBS

Table 2. Description of the programs used in the benchmark

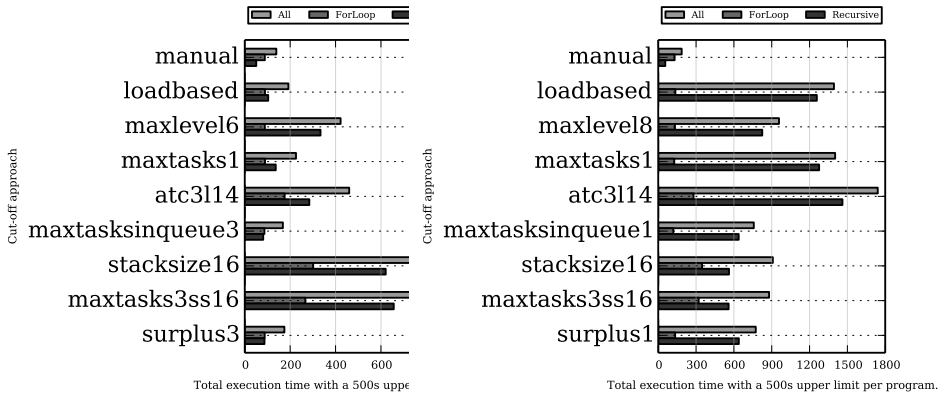

Fig. 1. Total time of execution of the benchmark suite using different cut-off approaches on *ingrid* (left) *astrid* (right) machines.

As expected, none of the approaches is better in overall than the manual solution, in spite of having better results in some programs.

The fastest solutions, *loadbased*, *maxlevel*, *maxtask*, *maxtasksinqueue* and *surplus*, have a very similar performance in programs mainly consisting of parallel for-loops. Their structure is well-defined and these different approaches do not show a large improvement. On the other hand, different approaches have distinct performances on recursive programs, which vary from one machine to the other. MaxTasks is slightly slower than LoadBased, Surplus or MaxTasksInQueue and

MaxLevel is the least performant of the group. MaxLevel consistently generates less tasks than the other approaches, which indicates that cutting off parallelism at that level of recursion does not generate the best number of tasks. On astrid, the impact of the approaches is much higher given that recursive programs are much slower than on ingrid. MaxTasksInQueue and Surplus approaches are more efficient, generating less tasks than other approaches in programs such as MergeSort and Integrate.

The difference of stacksize-based approaches between machines is also interesting. On ingrid, their performance is worse by a large gap in both recursive and loop programs. On the other hand, stacksized-based approaches are at the same level as the best approaches on astrid. While loop programs run slower than other approaches, recursive programs perform better using these approaches. The Do All program, made of very simple for-loops, can be used to understand this behavior. The performance of stacksize-based approaches is much worse than the manual approach while the number of tasks remain similar. Therefore, the difference lies on the distribution of tasks by queues. This is confirmed by the number of steals, which is more than 10 times higher in stacksize-based solutions. Thus, we conclude that the stack-size limit does not create the tasks in the best distribution among threads, requiring more communication between workers. On the NQueens program, another example of for-loops, both *maxtasks-ss* and *stacksize* did not finish the execution within the 500 seconds limit. Thus, limiting the creation of tasks if the number of stack frames is exceeded does not improve the performance of a regular for-loop programs.

Despite this decrease of performance on loop programs, stacksize-based approaches have the best automatic performance on a slower machine, astrid. For instance, in the case of Fibonacci, the only approaches to finish execution within the 500s limit are *manual*, *maxtasks-ss* and *stacksize*. This shows that this approach is very interesting, specially in recursive programs on machines similar to astrid.

5 Conclusions and Future Work

In this paper we have presented the results of several approaches for dynamically adjusting the granularity of parallel programs during run time. Those results have shown that the performance of each approach is highly dependent on the hardware, in CPU speed and number of workers, even with algorithms designed to adapt do any number of cores.

This conclusion may not appear useful as there is no direct answer to which cut-off technique to use, but it reveals that most studies performed to evaluate a cut-off mechanism are insufficient and require a more thorough evaluation across different machines and programs. We have made our benchmark available for other to use it and compare different approaches.

As a guideline, MaxTasks and Surplus have shown to have a good overall performance, with StackSize being a good alternative for CPU-intensive irregular programs.

For future work, we intend to analyze the structure of the source code to infer the type of parallelism and use machine-learning techniques to predict the best cut-off mechanism.

Acknowledgments

This work was partially supported by the Portuguese Research Agency FCT, through CISUC (R&D Unit 326/97), the `CMU|Portugal` program (R&D Project Aeminium CMU-PT/SE/0038/2008). The first author was also supported by the Portuguese National Foundation for Science and Technology (FCT) through a Doctoral Grant (SFRH/BD/84448/2012).

References

1. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE **5**(1) (1998) 46–55
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Volume 30. ACM (1995)
3. Lea, D.: A java fork/join framework. In: Proceedings of the ACM 2000 conference on Java Grande, ACM (2000) 36–43
4. Haghighat, M.R., Polychronopoulos, C.D.: Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In: Languages and Compilers for Parallel Computing. Springer Berlin Heidelberg (1993) 567–585
5. Mohr, E., Kranz, D., Halstead, R.: Lazy task creation: a technique for increasing the granularity of parallel programs. IEEE Transactions on Parallel and Distributed Systems **2**(3) (July 1991) 264–280
6. Duran, A., Corbal, J., Ayguad, E.: Evaluation of OpenMP Task Scheduling Strategies. (2008) 100–110
7. Olivier, S.L., Prins, J.F.: Evaluating openmp 3.0 run time systems on unbalanced task graphs. In: Evolving OpenMP in an Age of Extreme Parallelism. Springer (2009) 63–78
8. Olivier, S.L., Prins, J.F.: Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs. International Journal of Parallel Programming **38**(5-6) (2010) 341–360
9. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press (2008) 36
10. Acar, U., Charguéraud, A., Rainey, M.: Oracle scheduling: controlling granularity in implicitly parallel languages. ACM SIGPLAN Notices (2011)
11. Stork, S., Marques, P., Aldrich, J.: Concurrency by default: using permissions to express dataflow in stateful programs. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, ACM (2009) 933–940
12. Shun, J., Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Kyrola, A., Simhadri, H.V., Tangwongsan, K.: Brief announcement: the problem based benchmark suite. In: Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures, ACM (2012) 68–70
13. Bienia, C.: Benchmarking Modern Multiprocessors. PhD thesis, Princeton University (January 2011)