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, etc. This work studies the task granularity of regular and irregular parallel programs on symmetrical multicore machines. Task granularity is how many parallel tasks are created to perform a certain computation. If the granularity is too coarse, there might not be enough parallelism to occupy all processors. But if granularity is too fine, a large percentage of the execution time may be spent context switching between tasks, and not performing useful work.

Task granularity can be controlled by limiting the creation of new tasks, executing the workload sequentially in the current task. This decision is performed by a cut-off algorithm, which defines a criterion to execute a task workload sequentially or asynchronously. The cut-off algorithm can have a performance impact of several orders of magnitude.

This work presents three new cut-off algorithms: MaxTasksInQueue, StackSize and MaxTasksSS. MaxTasksInQueue limits the size of the current thread queue, StackSize limits the number of stacks in recursive calls, and MaxTasksSS limits both the number of tasks and the number of stacks. These new algorithms can improve the performance of parallel programs.

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 approaches (MaxTasksInQueue, StackSize and MaxTasksSS). The evaluation was performed using 24 parallel programs, including divide-and-conquer and loop programs, on two different machines with 24 and 32 hardware threads, respectively.

Our analysis provided insight of how cut-off algorithms behave with different types of programs. We have also identified the best algorithms for combinations of balanced/unbalanced and loop/recursive programs.

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 parallel programming expertise and knowledge of the program domain. Factors like thread or task creation, memory allocation and cache usage are fundamental in 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 this work we consider task-based parallelization. Work is divided into several tasks that can be executed in parallel, on top of a work-stealing runtime. This kind of runtime executes one thread per CPU core, or hyperthread when supported, and stores a queue of pending tasks per thread. When a thread has no tasks to pop from the queue, it steals from the end of another thread queue. Tasks are used because they have less scheduling overhead than threads, as they do not require a system call, and they can be used to balance the load in irregular unbalanced programs.

In both for-loop and recursive parallelism patterns, choosing the best granularity is an important issue. In our evaluation, the same program could execute within seconds or within days, depending on the granularity selected. The granularity is defined as how many parallel tasks are created to perform a certain workload. Tasks are a representation of blocks of code that can execute independently on different threads.

If tasks are too coarse, there might not be enough tasks to occupy the hardware threads, resulting in unused hardware resources that could have been used to improve performance. If 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, three new approaches will also 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 cut-off algorithms; 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. 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 algorithms. These cut-off algorithms will be described in Section 3.

Cut-off algorithms have a great impact on the performance of programs. An OpenMP evaluation[6] has compared two approaches (*MaxLevel* and *MaxTasks*, explained in detail in Section 3) and it found differences of up to 3x of speedup, but could not provide any guidance of how to choose an ideal cut-off. Later, a second study focused on the granularity and found that *ATC*, also detailed in Section 3, was better than the worst approach, but not always better than the best approach [7].

Two other studies, one also within OpenMP[8] and another comparing OpenMP to other approaches [9], have shown differences between the two cut-off mechanisms on unbalanced task graphs. Given the random nature of the benchmark programs used, there was no information obtained over which of the two (*MaxLevel* and *MaxTasks*) approaches was better.

3 Cut-off Mechanisms

A cut-off mechanism is an algorithm that decides whether a task will spawn new tasks for parallel work, or execute sequentially. Using LTC, it is possible to introduce a condition that stops the parallel execution of the program.

Different criteria have been proposed for deciding between parallel and sequential execution:

- **LoadBased** - The task will execute sequentially when all threads have work to perform in their queues. If there is at least one empty queue, it will execute in parallel [7].
- **MaxLevel**, or Maximum task recursion 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. If the task has more than a parameterized threshold of ancestors, it will execute sequentially. This approach is more suitable for balanced programs, where all subtrees have the same depth.
- **MaxTasks**, or Maximum number of tasks - Using this approach, tasks are created until the total number of active tasks in all worker queues reaches

a parameterized threshold[6]. After that point, all new computations are inlined instead of spawning another tasks. When the number of active tasks is lower, new tasks can be created in parallel until the threshold is surpassed 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.
- **ATC**, or Adaptive Tasks Cut-off - This approach is a hybrid of MaxTasks and MaxLevel, changing the parallelization policy based on the recursion depth[7]. Tasks are only created if two conditions are met: there are fewer tasks than the number of threads on a parameterized recursion level; and the depth is less than a parameterized threshold. This approach forces the tasks to be created in breadth in the lower depths, and aggregated in the higher depths. The idea is to improve the speed of task distribution while preventing over-sheduling of smaller tasks in higher depths. ATC adds a profiler that saves information regarding how much time a sub-tree takes to execute, and predicts further subtrees (if the prediction is larger than 1ms, the task will be created). This is based on the assumption that all tasks inside a level have a similar behavior, which does not happen in unbalanced programs.
- **Surplus**, or Suplus Queues Task Count - 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 parameterized threshold limit (usually 3 in existing ForkJoin benchmarks). If the surplus tasks count is higher than the threshold, the task will be executed sequentially, meaning that the current queue already has enough work for other threads to steal. If the surplus tasks count is lower than the threshold, the task is created in parallel to create more stealable work.

In this paper, we introduce three new algorithms for performing the parallel-sequential decision for each task:

- **MaxTasksInQueue**, or Maximum Queue Size - If the current queue size is lower than a parameterized threshold, the task will be executed in parallel. If the current queue is already at its maximum capacity, tasks will be executed sequentially. This approach is similar to MaxTasks, which limits the overall number of tasks, but considers the local queue only in order to reduce the overhead in accessing information from other threads.
- **StackSize** - Many of the fine-grained irregular programs would crash from stack overflows using existing granularity algorithms. The crash would occur later in the program, much after the fully sequential version of the program

would have finished. MaxLevel would consider the recursion depth of the program, but not that of the work-stealing runtime. We propose a cut-off algorithm based on the number of stack frames of the program. If the number of stack frames is lower than a parameterized value, the task will be executed in parallel.

– **MaxTasksSS**, or Maximum Tasks with Stack Size - Being based on depth, the StackSize approach is not suitable for irregular programs. In order to improve its performance, StackSize was combined with MaxTasks, resulting in a new approach. This approach uses the StackSize criteria to prevent very high granularity and uses MaxTasks criteria to allow for tasks to be created at the lower depths.

4 Cut-off Mechanism Evaluation

In this section, we begin by introducing the experimental setup and the benchmark suite. Then, we analyze and compare the different approaches.

4.1 Experimental Environment

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

Table 1. Details of the hardware used in the experiments.

Two machines (Table 1) were used in order to generalize results to more than one machine, both running Ubuntu 14.04 and Java HotSpot(TM) 64-Bit Server VM with Java 1.8. Programs were implemented on top of the Æminium Runtime [10].

To collect values, a practical statically rigorous methodology [11] was applied. For each combination of program and cut-off, we obtained a mean and the 95% confidence interval for the execution time in steady state from several executions until the Coefficient of Variance was below 5% or up to 30 executions. Each program had a timeout of 1 hour. All programs were executed in the same conditions, changing only the cut-off algorithm. There was no other load on the machine besides the experiment and the operating system.

4.2 Benchmark Suite

In order to evaluate cut-off algorithms, we use a benchmark suite comprised of different fork-join programs that represent the different types of programs being written for task-based work-stealing runtimes. Table 2 shows the list of the 24 programs used, their sources and the input sizes used.

The included programs are examples of divide-and-conquer, pipelined parallelism, do-all loops, do-across loops, nested parallelism and partial parallelism in a sequential algorithm. There are balanced and unbalanced programs in the benchmark suite.

Except for do-all, all programs are real-world examples and some are used in other benchmark suites, because of their heterogeneity. Compared with other evaluations, this is the largest and most heterogeneous set of programs ever used for evaluating cut-off algorithms. The benchmark suite is freely available at `https://github.com/AEminium/AeminiumBenchmarks`.

Program	Source	Input size	Type	Balance
BFS	PBBS [12]	d=26,w=2	Recursive	Regular
Black-Scholes	PARSEC [13]	10000^2	Loop	Regular
Convex-Hull	PARSEC [13]	10000^2	Recursive	Regular
Do-All		100 million	Loop	Regular
FFT	Cilk[14]	8388608	Recursive	Regular
Fibonacci	ForkJoin [3]	n=47	Recursive	Irregular
Fibonacci	ForkJoin [3]	n=49	Recursive	Irregular
Fibonacci	ForkJoin [3]	n=51	Recursive	Irregular
Genetic Knapsack		g=100,p=100	Loop	Regular
Health	BOTS [15]	l=7	Loop	Regular
Heat	ForkJoin [3]	4096x4096, it=1024	Loop	Regular
Integrate	ForkJoin [3]	error=10^{-9}	Recursive	Irregular
KDTree	PBBS [12]	n=10000000	Recursive	Regular
LUD	ForkJoin [3]	4096x4096	Recursive	Regular
Matrix Mult	ForkJoin [3]	p=10000, q=r=1000	Loop	Regular
MergeSort	ForkJoin [3]	n= 100000000	Recursive	Regular
MolDyn	JGrande [16]	it=1 size=40	Loop	Regular
MolDyn	JGrande [16]	it=5 size=30	Loop	Regular
MonteCarlo	JGrande [16]	10000x60000	Recursive	Regular
N-Body	PBBS [12]	n=50000, it=3	Loop	Irregular
N-U Knapsack		items=30, corr=3	Recursive	Irregular
NeuralNet		it=500000	Recursive	Regular
N-Queens	Cilk [14]	n=8..15	Loop	Irregular
N-Queens	Cilk [14]	16	Loop	Irregular
Pi		n=100.000.000	Loop	Regular
Quicksort	ForkJoin [3]	n=10000000	Recursive	Regular
RayTracer	JGrande [16]	n=2000	Loop	Regular

Table 2. Description of the programs used in the benchmark suite

4.3 Comparison of Cut-off Approaches

In sync with findings from prior works [6, 7], this section corroborates that no cut-off approach performed better than the others for all programs. Here, the differences in performance from the algorithms are addressed. Since the time distribution of the algorithms is not normal, swarm plots will be used. For parametrized cut-off approaches, we have used the parameters that achieved the best global time in a preliminary evaluation. MaxLevel had a depth limit of 12; MaxTasks had a task limit of twice the number of threads; ATC was configured with the two limits; StackSize had a limit of 16 stacks, MaxTasksSS has the same limits as MaxTasks and StackSize; Surplus had a limit of 3 surplus task count.

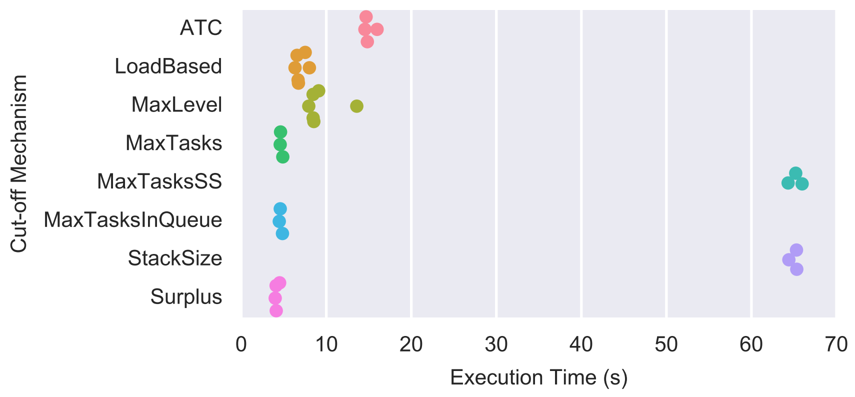

Fig. 1. Swarm plot of different cut-off approaches for the Do-all program on the *ingrid* machine.

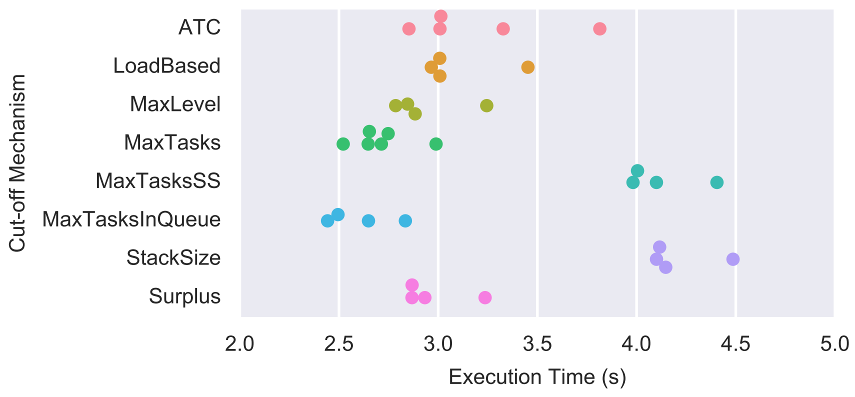

Fig. 2. Swarm plot of different cut-off approaches for the Matrix Multiplication program on the *astrid* machine.

Do-all is made of parallel loops with several iterations doing only one operation. Fig. 1 shows the performance of different cut-off mechanisms in the *ingrid* machine. MaxTasks, MaxTasksInQueue and Surplus were the most efficient strategies and they are all based on having enough work on each queue for others to steal. LoadBased has a similar approach, but does not have extra work in queues. In this case, allowing more threads to steal work results in a faster work distribution across the CPU cores. Recursion-depth approaches like MaxLevel and ATC are slower because, in this case, the depth considered was too deep and it created too many tasks. In this case a smaller depth, such as 6 would result in fewer tasks created, and less overhead, but in other programs it would result in worse performances. Stack-size approaches create too many tasks as well in this case. Fig. 2 shows the same behavior in the Matrix Multiplication program, which also has lightweight tasks in a 2 dimensional loop cycle.

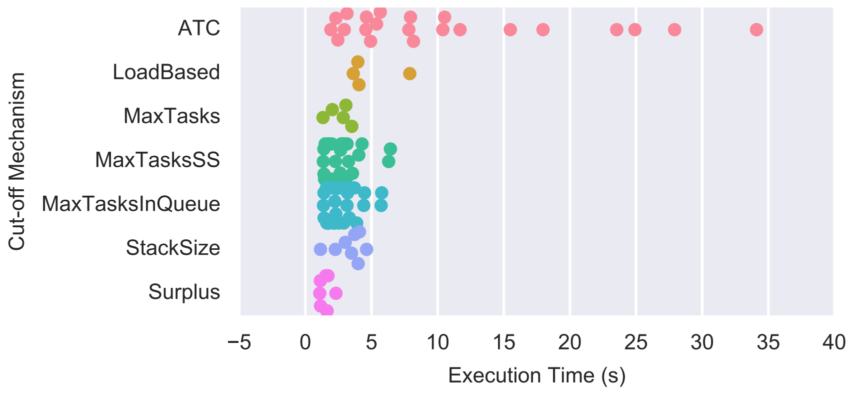

Fig. 3. Swarm plot of different cut-off approaches for the Fibonacci program on the *ingrid* machine.

Fig. 3 shows the Fibonacci program with different cut-offs. Fibonacci is a highly irregular program that generates a skewed parallelization tree, with an extremely lightweight computation. In this case all approaches handle the program reasonably well, but MaxLevel is not able to finish the program within the defined timeout, as its execution time is not shown. Fig. 4 shows integrate, another highly irregular program, in which cut-off programs show the same relative performance, with MaxLevel being much slower than its counterparts.

In the N-Queens program, in Fig. 5, Loadbased, MaxLevel and Surplus are the fastest algorithms. This program has a high branching factor and a high penalty for over-creating tasks, because it needs to allocate memory for each parallelization. MaxLevel avoids going too deep in the recursion tree, preventing the unwanted unnecessary memory allocation. LoadBased also prevents creating extra tasks and performs similarly to MaxLevel.

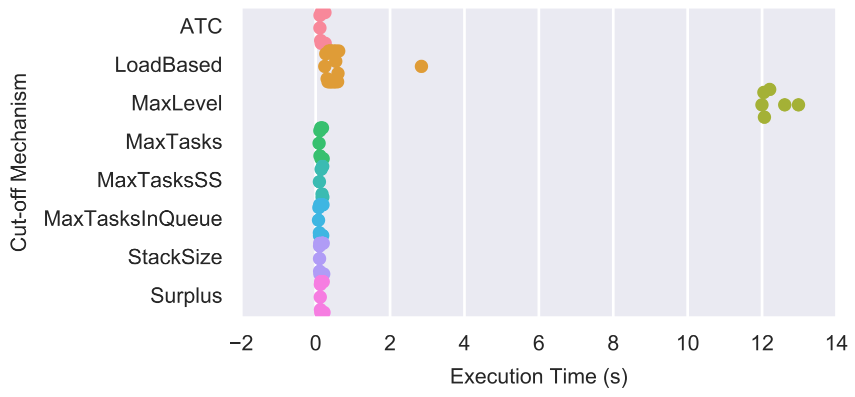

Fig. 4. Swarm plot of different cut-off approaches for the Integrate program on the *ingrid* machine.

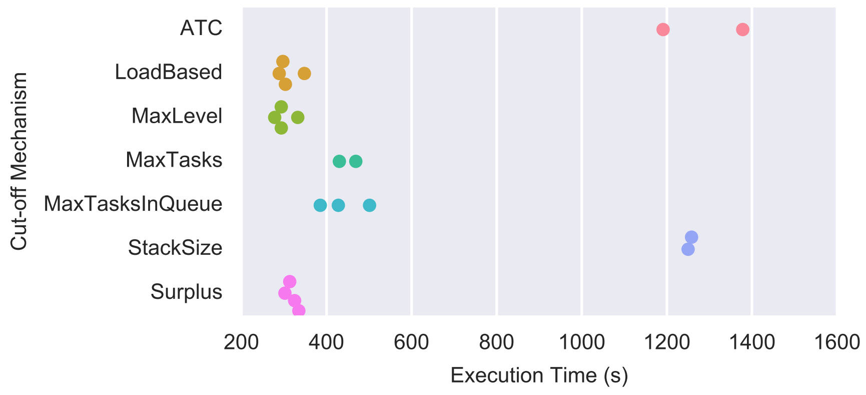

Fig. 5. Swarm plot of different cut-off approaches for the N-Queens program on the *ingrid* machine.

Fig. 6 shows the cut-off performance in the FFT program. Only ATC and MaxTasksSS have finished the program 3 times within the timeout. FFT is a program that allocates a large amount of memory in its divide-and-conquer process. The allocation of tasks on top of the baseline allocation of the sequential program penalize the creation of a large number of tasks. The two best approaches have two mechanisms to limit the creation of tasks, one limiting the queue size, and another preventing from going too deep in the recursion level. The difference between the two is that ATC limits using the program recursion and MaxTasksSS uses the internal recursion of the work-stealing runtime.

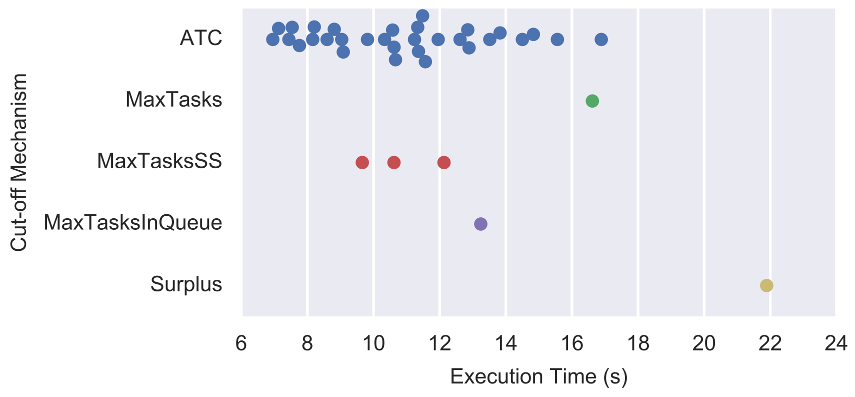

Fig. 6. Swarm plot of different cut-off approaches for the FFT program on the *ingrid* machine.

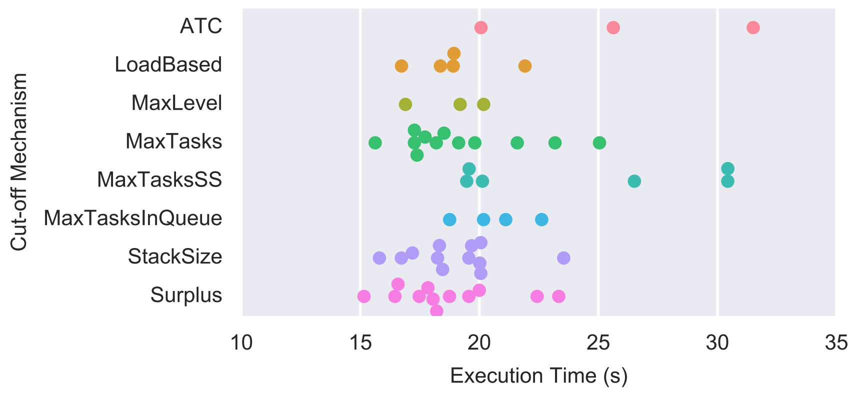

Fig. 7. Swarm plot of different cut-off approaches for the Raytracer program on the *ingrid* machine.

In Fig. 7, we can see the opposite behavior in which ATC and MaxTasksSS are the worst approaches. One reason for this is that these hybrid approaches use two mechanisms to improve their worst-case programs, but introduce overhead in cases where the individual algorithms are ideal.

Fig. 8 shows the same plot for the Neural Network program. In this program, creating tasks has a relatively large overhead compared to the program and only StackSize approaches have been able to complete the program within the time-out, in a relatively small time. This is one example that justifies the introduction of stack-size approaches, in which the workload of tasks is very light and there is expensive work in merging the result of each recursive call. This is the same

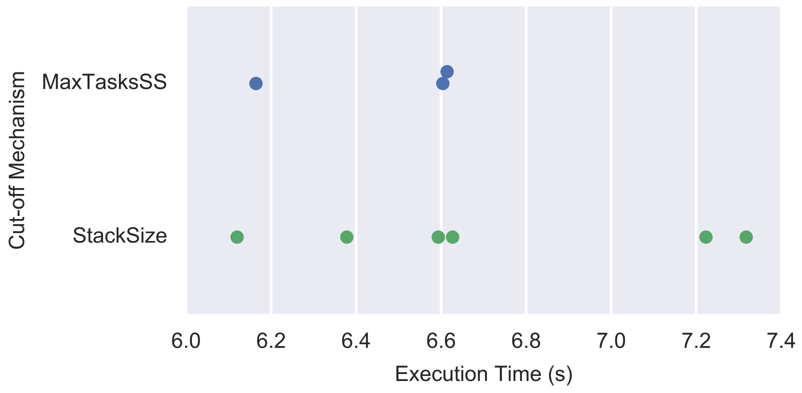

Fig. 8. Swarm plot of different cut-off approaches for the Neural Network training program on the *astrid* machine.

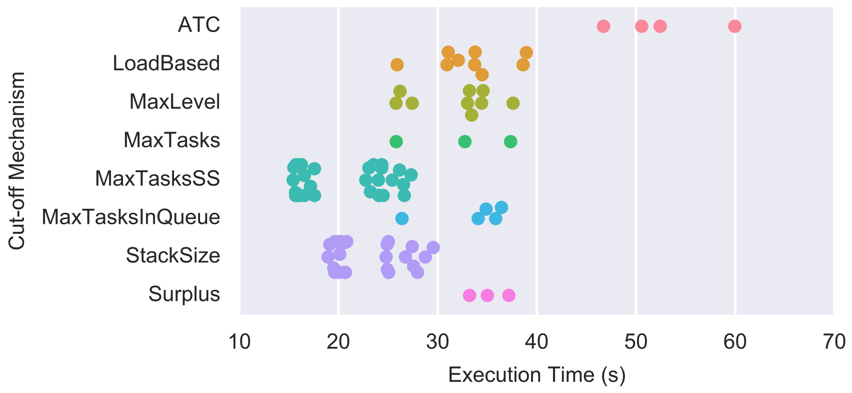

Fig. 9. Swarm plot of different cut-off approaches for the KDTree training program on the *astrid* machine.

behavior as that of the Fibonacci program with a very large input. KD-Tree is another program where Stack-based approaches are also advantageous, but not by a larger difference, which can be seen in Fig. 9.

Fig. 10 shows the execution time of all the programs that are a combination of a given type and balance. While previous results were specific to each program, these plots show aggregated information from several programs. Irregular for-loop programs can be efficiently optimized using either LoadBased or MaxLevel algorithms. In general, these two algorithms do not schedule as many tasks as others because they are created either when there is an empty queue, or only in the beginning of the program. In irregular loop programs, MaxTasks is the best algorithm as it allows the creation of enough tasks to spread work across all threads, even in later iterations. Recursive irregular programs work best under Surplus because, similarly to MaxTasks, it allows for extra work to be scheduled, which improves the distribution of irregular work across threads. Finally, recursive balanced programs perform well under StackSize, which prevents high levels of recursion, regardless of the nature of the algorithm.

5 Conclusions and Future Work

In this paper we have introduced three new algorithms for dynamically managing the granularity of parallel programs. Additionally, we have evaluated new and existing cut-off algorithms over a 24-program benchmark suite. The three proposed algorithms were able to outperform existing algorithms in at least one of the programs in the benchmark suite.

We have identified MaxTasks as a reasonable cut-off algorithm for a large set of programs. In irregular loop programs, Load-based and MaxLevel can be used to improve the performance of programs. Additionally, the proposed StackSize algorithm can be used in regular recursive programs or when the memory allocation required for parallelization is high.

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).

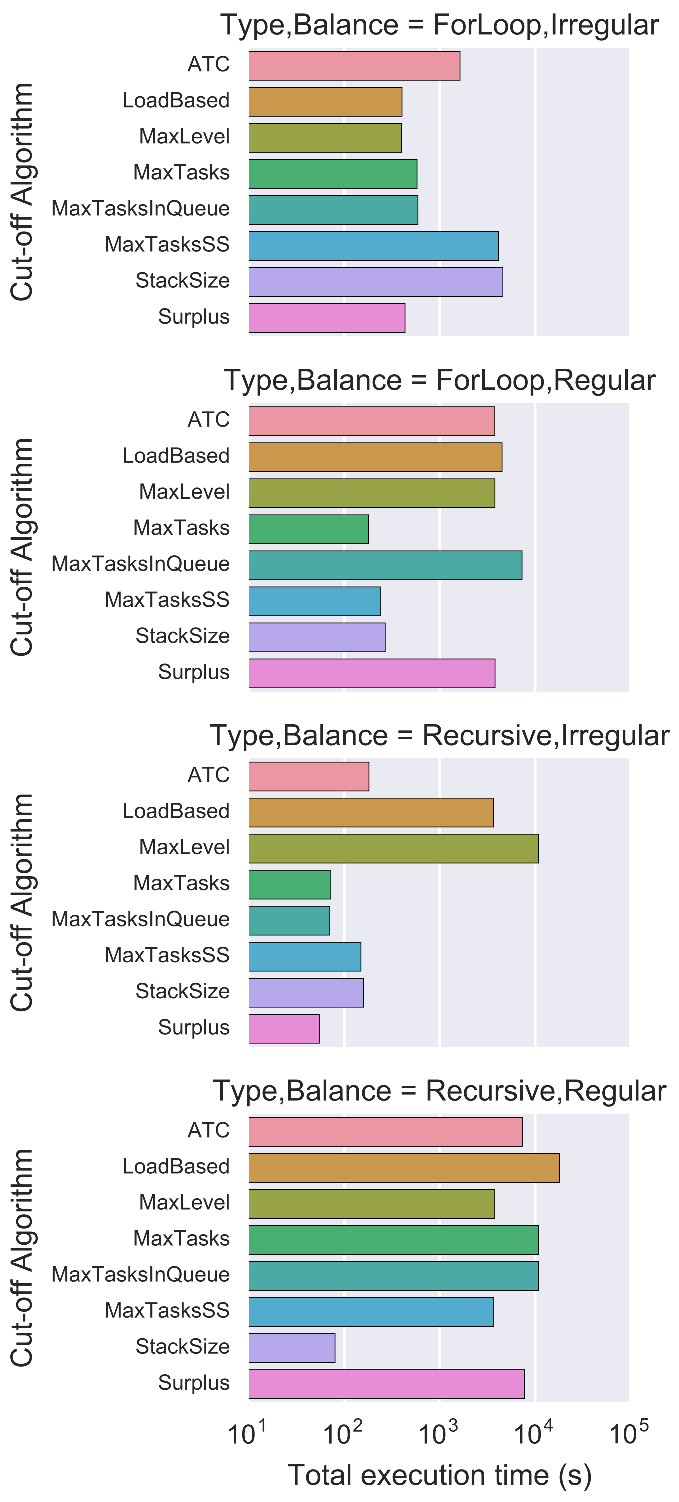

Fig. 10. Total execution time of programs that fit a certain Type-Balance pattern on the *astrid* machine.

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. 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
8. 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
9. 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
10. Stork, S., Naden, K., Sunshine, J., Mohr, M., Fonseca, A., Marques, P., Aldrich, J.: Æminium: A permission-based concurrent-by-default programming language approach. ACM Transactions on Programming Languages and Systems (TOPLAS) **36**(1) (2014) 2
11. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. ACM SIGPLAN Notices **42**(10) (2007) 57–76
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)
14. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. In: ACM Sigplan Notices. Volume 33., ACM (1998) 212–223
15. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguadé, E.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: 38th International Conference on Parallel Processing. (2009) 124–131
16. Smith, L.A., Bull, J.M., Obdrizalek, J.: A parallel java grande benchmark suite. In: Supercomputing, ACM/IEEE 2001 Conference, IEEE (2001) 6–6