Adaptive and Architecture-Independent Task Granularity for Recursive Applications

Antoni Navarro[1] Sergi Mateo[1] Josep Maria Perez[1] Vicenç Beltran[1] Eduard Ayguadé[1,2]

[1] Barcelona Supercomputing Center [2] Universitat Politècnica de Catalunya

{antoni.navarro,sergi.mateo,josep.m.perez,vbeltran, eduard.ayguade}@bsc.es

Abstract. In the last few decades, modern applications have become larger and more complex. Among the users of these applications, the need to simplify the process of identifying units of work increased as well. With the approach of tasking models, this want has been satisfied. These models make scheduling units of work much more user-friendly. However, with the arrival of tasking models, came granularity management. Discovering an application's optimal granularity is a frequent and sometimes challenging task for a wide range of recursive algorithms. Often, finding the optimal granularity will cause a substantial increase in performance.

With that in mind, the quest for optimality is no easy task. Many aspects have to be considered that are directly related to lack or excess of parallelism in applications. There is no general solution as the optimal granularity depends on both algorithm and system characteristics. One commonly used method to find an optimal granularity consists in experimentally tuning an application with different granularities until an optimal is found. This paper proposes several heuristics which, combined with the appropriate monitoring techniques, allow a runtime system to automatically tune the granularity of recursive applications. The solution is independent of the architecture, execution environment or application being tested. A reference implementation in OmpSs — a task-parallel programming model — shows the programmability, ease of use and competitive performance of the proposed solution. Results show that the proposed solution is able to achieve, for any scenario, at least 75% of the performance of optimally tuned applications.

Keywords: OmpSs, Cost, Autotuning, Threshold, Granularity, Cutoff

1 Introduction

The optimal unit of work in a parallel code depends on many factors. To name a few; input data, resources allocated or the current load of a machine are some of the most important ones to take into account. Statically setting a certain granularity for an application in a specific environment, machine and/or input may cause a dramatic decrease of performance when that same application runs with other parameters or on different machines. Once the granularity is set, often, it will be immutable for the entire execution. Static granularities then, are too rigid and cause applications to suffer a decrease in performance when executed with different configurations.

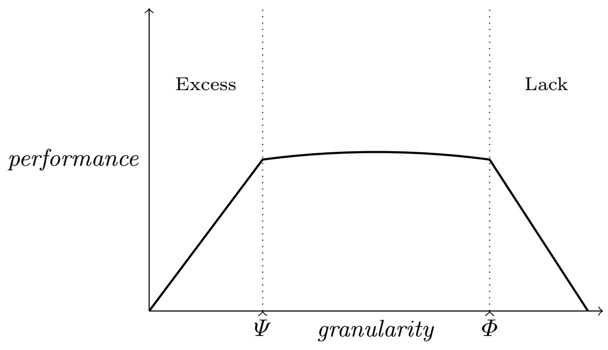

Fig. 1: Effects of different granularities in performance

Figure 1 generically illustrates the consequences of choosing incorrect granularities. When a certain limit Ψ of granularity is not met, fine-grained tasks are generated and that leads to an excess of parallelism. This is worsened by stressing runtime libraries with the management of an excessive amount of tasks. On the other hand, once a certain threshold Φ on granularities is surpassed, tasks are too coarse, which leads to a lack of parallelism and load imbalance. These granularity thresholds are highly dependent on a large number of factors. Some are linked to the hardware architecture, and some others are related to the dynamic behavior of the application and the system on which it runs.

Offline tuning is an extensively used technique when searching for good granularities. It is based on executing applications with different input parameters and granularity cutoffs until an optimal performance is reached. Negative impacts on performance caused by choosing inadequate granularities can be softened by limiting the number of tasks from a runtime's point of view. This is done in order not to stress the runtime with the creation and handling of fine-grained tasks that would not payoff in computation. The latter approach behaves as expected when tasks are fine grained and thus the creation of too many of these would expose too much parallelism. However, for coarse-grained tasks this technique prevents a level of parallelism which is needed.

Visible flaws from the aforementioned techniques and the difficulty of writing architecture-independent code raise interest for a way to auto-tune applications and to detect optimal granularities, taking into account factors such as input sizes or resources. These techniques could improve if precise information about the computation being performed by the tasks is known. Simultaneously, this interest creates a demand for task-based programming models [1] that are able to manage themselves through monitoring. By monitoring, these would have access to certain metrics, which could be used to take scheduling decisions.

This paper presents the OmpSs Autofinal Module. Its purpose is to enable the automatic detection of the optimal granularity of recursive applications, regardless of the architecture, input size or execution environment being used. It consists of a monitoring infrastructure, several heuristics and language support through clauses. The monitoring infrastructure provides precise execution metrics that combined with the heuristics and usage of language constructs, provide information to the runtime to decide when it is worth to instantiate tasks. In addition, this paper shows what kind of language support is recommended to provide useful information to the monitoring infrastructure. Results show that the usage of this technique achieves performance that is competitive in comparison to manually tuned applications on several architectures and diverse applications.

The rest of the paper is structured as follows. Section 2 overviews the related work and introduces the most frequent techniques on granularity management. Section 3 introduces the idea of the *cost* clause from a programming model's point of view. Section 4 thoroughly explains an implementation of Autofinal on OmpSs. Section 5 presents a case study and evaluates its performance and lastly, Section 6 concludes this work.

2 Related Work

In most task-based recursive codes, the overhead of task creation and management causes the overall performance of the code to decrease at a certain recursion depth. Because of this, runtime libraries provide techniques to ensure limits on these depths. Managing and limiting granularities have previously been considered in task-based programming models. For instance, OpenMP [2] provides several clauses to tune the task granularity. Below are some of the clauses that are most related to this paper.

– *final*: For recursive problems it may be beneficial for performance to stop task creation at certain depths to expose enough parallelism and reduce overhead. That is, the task will not instantiate children. Instead the code is executed on a single unit of work. Its syntax is

$$final\ (expression)$$

where *expression* evaluates into a boolean value that determines whether the task is *final*.

– *if*: Its syntax is the following:

$$if\ (expression)$$

Before instantiating the task, the *expression* is evaluated. If its value is false, the task is not instantiated and instead its code runs as part of the current task.

– *mergeable*: In certain scenarios, if this clause is present, the task's parent will share its data environment with the task.

Autotuning is often described as a general technique to automatically adapt the execution of a program to a given parallel computer. This adaptation is done in order to optimize one or more runtime performance metrics such as execution time [3]. Other works, such as the one conducted by *Ray S. Chen* [4] or the one conducted by *Chung & Hollingsworth* [5], researched ways to autotune applications. Their works rely on running applications several times by differing parameters until an optimal configuration or granularity is found. This paper focuses on dynamically autotuning applications without a previous training phase. The tuning determines the granularity to apply to the *final* clause.

Other works such as the one conducted by *Alejandro Duran* [6] discussed a similar way of autotuning granularity, however without the usage of the *final* clause nor the *cost* clause. It made use of bare execution times in order to determine when a granularity is fine enough, regardless of the type of the task. This approach might work as expected with some specific architectures, applications and input sizes, however introducing variability in any of these three factors might trigger wrong cutoff decisions, resulting in decreases of performance. This paper however, focuses on precisely determining the computational weight of the tasks and make accurate predictions of execution times, regardless of the architecture, input size or application.

3 Language Support

To make the most of the OmpSs Autofinal module, application developers have to provide additional hints to the runtime. First, the developer must express the intention to use Autofinal. This is necessary in order for task monitoring to happen. Therefore, one could extend the *final* clause to accept keywords like the following:

$$final\ (auto)$$

where *auto* is the keyword that expresses the desire to automatically establish *final* cutoffs.

Furthermore, the accuracy of the data obtained by monitoring can be improved by normalizing it. This is done through information about the computational weight of tasks. For this purpose, the *cost* clause is introduced with the following syntax:

$$cost\ (expression)$$

where $expression$ corresponds to a single or a combination of algebraic functions that evaluate into a positive real number.

4 Implementation

The OmpSs Autofinal implementation consists of the following parts: a monitoring infrastructure, several heuristics and a collection of constructs used by the runtime and provided by the programming model. Each of these parts is explained in the next sections.

The following work was implemented in the OmpSs [7] programming model. OmpSs — OpenMP Superscalar — is a task-based, data-flow aware parallel programming model developed at the Barcelona Supercomputing Center. Its proximity to OpenMP makes it relevant to runtime developers working on similar implementations.

4.1 Task Monitoring

A monitoring infrastructure was added to give the runtime the ability to decide when a task's granularity is fine enough. The infrastructure is capable of providing a histogram of task execution times. Once a task has been executed, its metrics are aggregated per task type. However, this information is not used as-is, the metrics are first normalized. This is explained in the next subsection. Some of the metrics provided are execution time, runtime execution time, number of leaf tasks and number of tasks that have at least created one task.

4.2 Final Clause

This sections explains the implementation of the *final* clause in OmpSs. The compiler first makes use of closures and duplicates every function of the user application which creates tasks. Once those functions are executed and if the *final* clause evaluates to true, the task enters in *final* mode. This mode prevents the task from generating any other task.

4.3 Cost Clause

Predictions on execution times could be done by simply using the metrics obtained by the monitoring infrastructure. This approach is exactly the one taken if no other information is available. Nonetheless, the information provided by the *cost* clause can be used to normalize the predicted execution time of a task. The normalization and the contents of the *cost* clause are important because the execution time of a task does not depend only on the type of the task. The contents of the *cost* clause allow comparing the expected execution time with other tasks of the same type. When a task does not rely on any algorithmic function, and thus does not contain the *cost* clause, the normalized *cost* will simply be the average execution time and thus the clause will not have any negative impact on the decision. However if the task's computation follows an algorithmic function, the normalized average will provide much more veracious information towards the runtime's decision. For the OmpSs implementation of Autofinal, this normalization consists on dividing the expected execution time by the contents of the *cost* clause. Figure 2 shows the normalization under the assumption that *cost* is available for the task.

This clause however is not bound to be the computational weight of a task, as it could evaluate other kinds of expressions. As an example, an I/O intensive task could use the *cost* clause by giving hints about the I/O operations it is going to perform. Other tasks such as memory bound tasks, could make use of the *cost* clause by specifying the number of load/stores it is going to perform.

```
Function normalize_cost(task):
    expected_time = get_prediction(task->get_type());
    normalized_cost = expected_time / task->get_cost();
    return normalized_cost;
end
```

Fig. 2: Normalization of *cost* for a task.

The usefulness of the clause then, relies on acting as a hint given to the runtime library. This hint is mostly used as information about the relative computational weight of a task — In other words, the algorithmic cost. However, it may be used for other purposes such as the ones discussed previously and in Section 6. Figure 3 shows the usage of this clause applied to a few well known algorithms.

In order to achieve an adaptive strategy, OmpSs' implementation of Autofinal has an average normalized *cost*. This *cost*, also referred in this document as *unitary cost*, is seen by the runtime as the expected execution time for a unit of *cost* extracted from the clause. That is, taking into account that the *cost* evaluates into a positive real number, as explained in Section 3. The average

```
#pragma oss task cost(N*N)
void InsertionSort(int * src, int * dst, size_t N);

#pragma oss task cost(N*N*N)
void MatMul(int N, double * A, double * B, double * C);

#pragma oss task cost(N*log(N))
void MergeSort(int * src, int * dst, size_t N);
```

Fig. 3: A few algorithms showcased using the *cost* clause.

unitary cost is obtainable by having a window of measurements of unitary cost. By using this strategy, normalized costs are adapted throughout the execution of the application and its changes of behavior. In other words, the average is obtained using the `'N'` latest measurements.

OpenMP's *final* clause, as shown previously in Section 2, allows users to manually set a threshold on task granularities. To use it, the developer must have knowledge of application behaviors and the computation being done by the affected tasks. All of this forces the developer to study said applications and to execute them with different thresholds and figure out their behavior on the given hardware and software setup.

Hence why it is interesting for the runtime to monitor the behavior of tasks and automatically activate the *final* clause when desired. It is from this idea that Autofinal was created. Autofinal uses task monitoring in order to estimate the execution time of future tasks and determine whether they should be *final* or not. Figure 4 shows a pseudo code with the heuristics that have been chosen to be taken into account in the decisions of automatic *final* appliance. These are thoroughly explained immediately after.

```
Function is_automatically_final(task):
    arity = children_tasks / parent_tasks;
    if no_cost_available(task) then
        current_tasks = pow(arity, recursive_depth);
        maximum_tasks = total_cpus × TASKS_PER_CPU;
        return current_tasks > maximum_tasks;
    end
    else
        expected_time = cost × unitary_cost;
        return expected_time < THRESHOLD;
    end
end
```

Fig. 4: Pseudo code of heuristics used to determine if a task should be *final*.

The decision has two well-differentiated heuristics. When executing a certain task, if the runtime does not have timing information of its type it can be due to two reasons. The first occurs when a task is the first of its type to be executed. The second occurs when all previous tasks from the same type have not finalized their execution and therefore have not contributed to task monitoring yet. For better understanding of the second scenario, one could think of recursive algorithms like mergesort or fibonacci in which non-leaf tasks contain a taskwait, which waits for its children to finish. In these, there is no complete timing information available until one of the recursive branches of the algorithm reaches a leaf task.

When either of the previous scenarios is met and therefore no timing information is available, it is still useful to limit the number of instantiated and not finalized tasks. Otherwise, tasks with very fine granularity end up being created. The best way to do this is limiting the maximum number of tasks of a certain type at a certain moment and, in order to do this, some metrics are needed. In OmpSs, these metrics are the recursive depth of tasks from the same type — `recursive_depth` — and the average number of tasks created by a certain type or, as referred in the pseudocode, the `arity` of a task. The `arity` is computed taking into account the number of tasks from a certain type which have at least a children task (`parent_tasks`) and the number of leaf tasks (`children_tasks`). If at a task's creation point the average (`current_tasks`) surpasses a limit, the task is created as a *final* task. This limit is calculated using a configurable number of tasks per CPU and the total number of CPUs.

In the event that the runtime has timing information for a certain type of task, it can estimate the execution time for future tasks of that type. This estimation can be more precise if the developer provides computational information about the task through the *cost* clause. If the execution time estimation does not meet a certain configurable threshold, the task is generated as a *final* task.

5 Results

To test Autofinal's effectiveness, four very different recursive benchmarks were used.

- **The Fibonacci sequence**: Fibonacci was chosen because it is a benchmark with very fine granularity. The computation of tasks at the end of recursivity is as simple as returning an integer. The sequence of the first 35 Fibonacci numbers was chosen as the size for this benchmark.
- **Mergesort**: The mergesort algorithm was chosen to test a wide range of granularities. The computation of a task can be as simple as a comparison between two numbers or as coarse as sorting and merging two big chunks of an array. An array of 10^8 `double`s was used as this benchmark's input.
- **NQueens problem**: In the NQueens benchmark, granularities grow exponentially. Testing this attribute challenges effectiveness and accuracy of predictions. The board size used for this benchmark is 15 rows by 15 columns.

- **Strassen Matrix Multiply**: The Strassen matrix multiplication algorithm was chosen to evaluate a data intensive compute-bound benchmark. The size of the matrixes was 2^{13}.

5.1 Autofinal heuristics

- **Cost:** For this heuristic, a warmup iteration was executed for every benchmark. That is, a whole execution of the benchmark was performed to fill histograms with timing information. By doing this, in the second iteration, even the first tasks to be executed will be compared against a prediction. This heuristic covers the scenario of having previous timing information of tasks.
- **Hybrid:** This heuristic cuts recursive depth early to avoid fine-grained tasks. Once the histograms are filled with timing information, this heuristic abandons the first technique and continues by using the cost heuristic. Hence why this heuristic is named hybrid. This heuristic covers the scenario of not having previous timing information of tasks.

5.2 System configuration

Benchmark results were obtained on four different architectures in order to test variability of performance in different architectures and configurations. Results were always obtained in a single node using all the available CPUs in the node. Next is a list of all the machines used and the number of cores used in each.

- **MinoTauro:** Contains a cluster with 39 R421-E4 Servers, each with 2 Intel Xeon E5-2630v3 (Haswell) 8-core processors, each @ 2.4 GHz. 16 cores were used.
- **ThunderX:** Contains 4 Nodes each equipped with 2 Cavium ThunderX sockets, each of them with 48 ARMv8-A cores, each @ 1.8 GHz. 96 cores were used.
- **KNL:** Each KNL machine contains an Intel Xeon Phi socket @ 1.40 GHz, with 68 cores in each socket and 4 threads per core. 68 cores were used with 1 thread per core.
- **Power8:** Contains 2 Machines with 2 sockets Power8 10C @ 3.49 GHz, 8 threads each core. 20 cores were used.

5.3 Performance results

The obtained measurements demonstrate how a statically chosen granularity for one architecture does not perform well on others and how Autofinal improves this situation. For each benchmark, the best granularity was manually found on four different architectures. After that, the benchmarks were ran in every machine with the best granularities of all four machines and with autofinal with two different heuristics.

Figure 5 shows the results. It contains four plots, from top to bottom: Fibonacci, Mergesort, NQueens & Strassen. The horizontal axis corresponds to the execution of a host, and each host has six measurements. The first 4 correspond to the best granularity of the benchmark in each host, and the last two correspond to autofinal executions with two different heuristics. These are explained in Section 5.1.

The first plot then, contains the comparison of the Fibonacci benchmark. In the horizontal axis are the hosts where the benchmark is ran. That means that the very first six bars in the plot correspond, from left to right to:
1. Executing Fibonacci on MinoTauro with MinoTauro's optimal granularity. Hence why the speedup is 1.
2. Executing Fibonacci on MinoTauro with ThunderX's optimal granularity.
3. Executing Fibonacci on MinoTauro with KNL's optimal granularity.
4. Executing Fibonacci on MinoTauro with Power8's optimal granularity.
5. Executing Fibonacci on MinoTauro with Autofinal's cost heuristic.
6. Executing Fibonacci on MinoTauro with Autofinal's hybrid heuristic.

Sometimes, statically choosing a granularity for a certain architecture causes a dramatic decrease on performance on every other. This is highly visible when executing Fibonacci in the ThunderX system with KNL's optimal granularity. In this scenario, it barely achieves 20% of the performance of the optimal granularity. Autofinal however, achieves at least 90% of ThunderX's optimal performance. Another scenario, but not the last, is when executing Strassen in the KNL system with Power8's optimal granularity. The performance obtained is around 68% of the performance of the optimal granularity, while Autofinal achieves more than 90%.

The cost heuristic is used with a previous warmup iteration of the application and therefore can make timing predictions from the start of the execution. These plots also exposed that the cost heuristic behaves better than the hybrid heuristic because the cost heuristic is fed on a warmed up environment. It relies on previous timing information. The loss of performance of the hybrid heuristic due to not having previous timing information is at most 20%, while the penalty for using a granularity from another architecture can be as high as 90%.

Autofinal then, achieves competitive performance regardless of the architecture and benchmark where it is tested. Results show that it achieves at least 75% of the optimal performance in every scenario and that using granularities from other architectures can lead to only obtaining around 20% of the optimal performance.

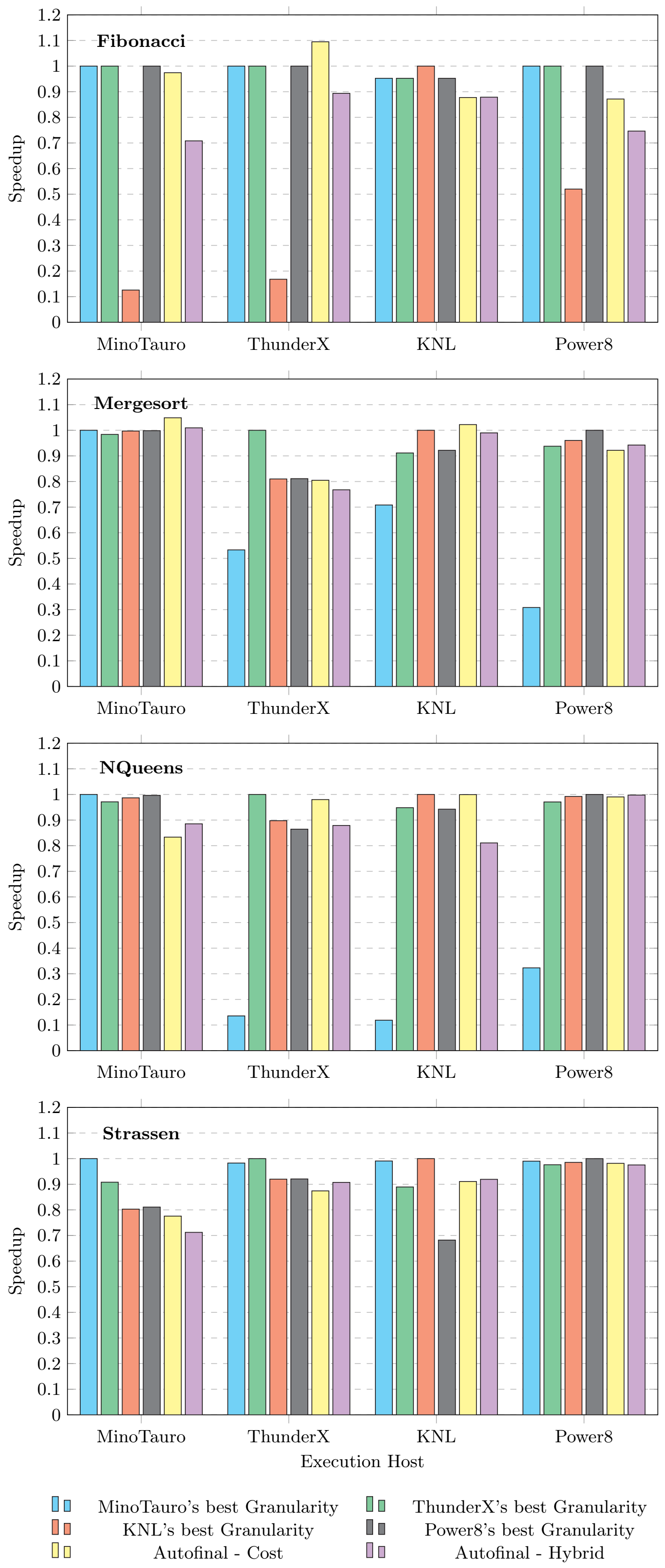

Fig. 5: Performance comparison of Heuristics vs. Optimal Granularities.

Results also show that in some cases, Autofinal is able to find an even better granularity than the apparently optimal one. This is possible because it does not just set a fixed granularity. Instead, it decides at execution time whether a task should be *final*. Hence, in some cases it chooses a mix of granularities. This is visible when executing Mergesort on the KNL system or the MinoTauro system, with any Autofinal heuristic. The results indicate that the optimal granularity is a mixture of tasks with *final* granularity 10^7 and 10^6.

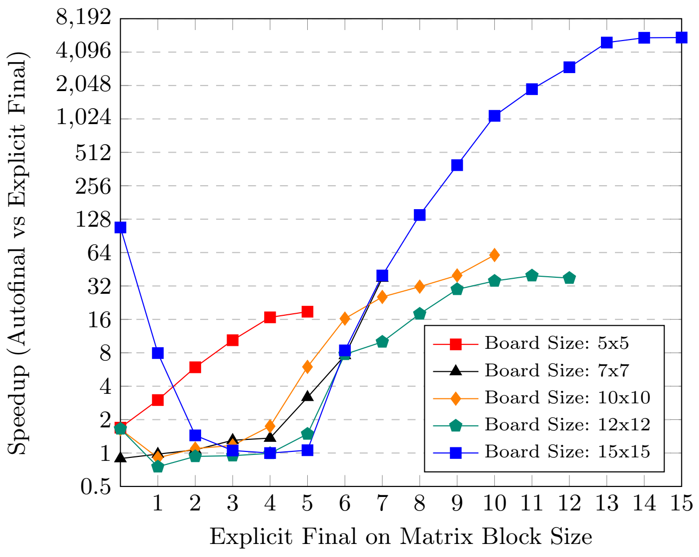

Fig. 6: Autofinal vs Manual Tuning for the NQueens Benchmark with different input sizes

Figure 6 shows the performance of Autofinal against manually tuning the granularity of tasks for the NQueens Benchmark with the *final* clause. This comparison is done by executing NQueens with different input sizes in order to test Autofinal's adaptiveness to an application's settings. As shown, the optimal granularity often comes linked to the input size of the application and Autofinal is able to adapt to these settings. For completeness, this section includes Figures 7 & 8. These show the speedup obtained by executing the previous benchmarks with Autofinal versus executing them with a wide range of manual cut-off depths. Each series corresponds to an architecture and the speedup is computed comparing Autofinal's cost heuristic's performance against each cut-off step's performance. These plots show which are the appropriate granularities for each benchmark in each architecture as well, and how Autofinal performs against these. The best granularity for each benchmark then was chosen to plot the performances seen in Figure 5.

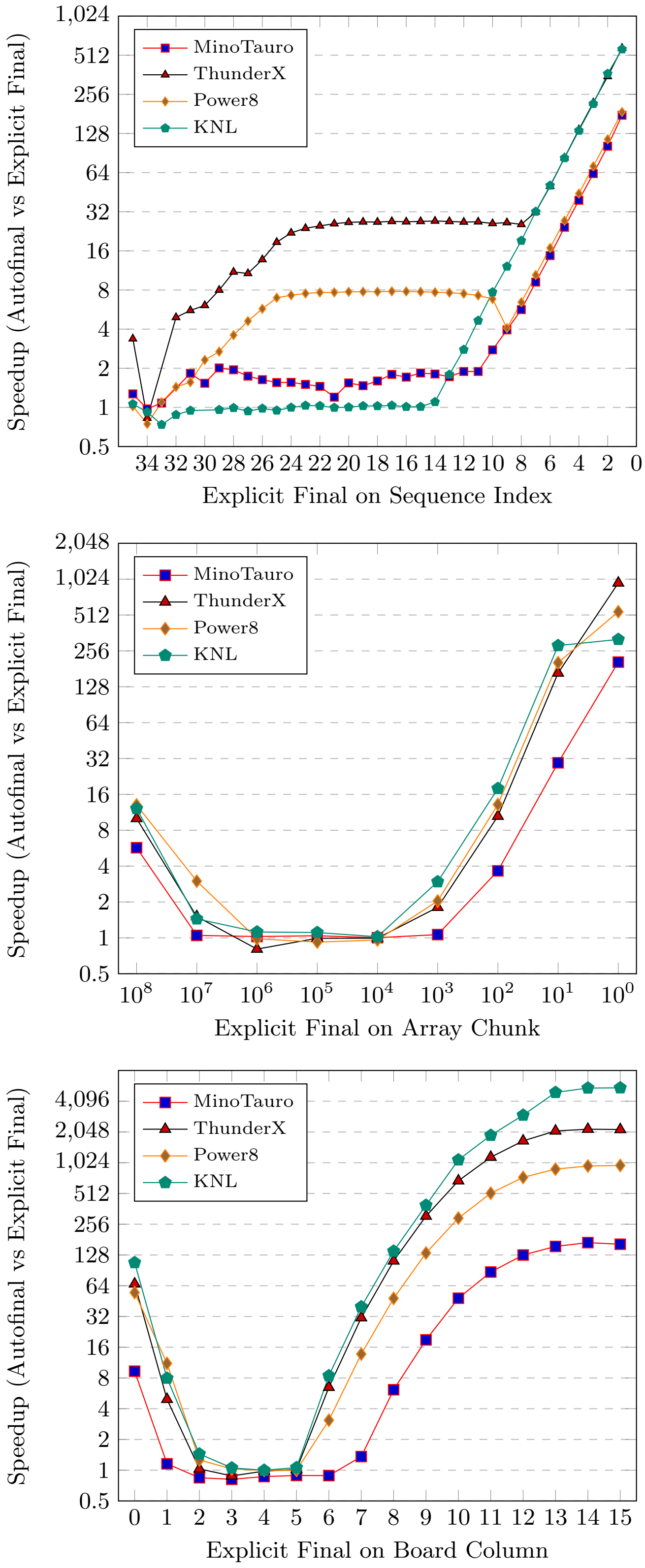

Fig. 7: Autofinal vs. Manual Tuning for Fibonacci (Top), Mergesort (Middle) & NQueens (Bottom)

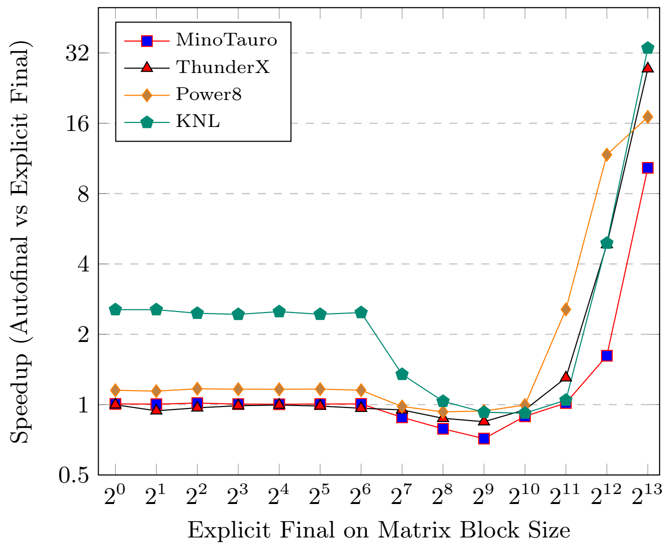

Fig. 8: Autofinal vs Manual Tuning for the Strassen Benchmark

For the Fibonacci plot, each cutoff corresponds to the index number of the sequence at which recursive depth is cut with the *final* clause. The Mergesort plot's cutoffs correspond to the array chunk size at which recursive depth is cut. For the NQueens benchmark, the cutoff corresponds to the board's column index. Lastly, for the Strassen benchmark, the cutoff chosen is the matrix block size at which recursive depth is cut.

6 Conclusion & Future Work

This paper presented how automatically detecting optimal granularity cutoffs can be integrated into a task parallel programming model. Furthermore, it showed which runtime features, as well as language support, are needed to allow using Autofinal. Having specific information about the computation of tasks allows making precise predictions as well as offering a general solution to automatically find well performing granularities for applications. The evaluations show that making the runtime aware of the computational weight of tasks and monitoring them allows to predict with precision task execution times and hence to find granularities that adapt to the architecture and the runtime environment.

The Autofinal technique raised interest in exploring its behavior with processors that allow dynamic frequencies. With that in mind, it would also be interesting to compare static granularities against the usage of Autofinal in the aforementioned processors.

The introduction of the *cost* clause in the language also provides specific metrics to create new capabilities on the runtime library, like *cost* based scheduling policies. This raised interest in using the *cost* clause with the taskloop construct. The idea relies on having extra information about the computational weight of iterations from a taskloop in order to better balance and schedule the workload.

7 Acknowledgments

This work has been supported by the Spanish Ministry of Science and Innovation (contract TIN2015-65316), the grant SEV-2015-0493 of Severo Ochoa Program awarded by the Spanish Government, and by Generalitat de Catalunya (contract 2014-SGR-1051)

References

1. Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. *The Design of OpenMP Tasks.* In *IEEE Transactions On Parallel Distributed Systems, Vol.20, No.3*, pages 404–418, March 2009.
2. OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.5, November 2015.
3. Vaidyeswaran Rajaraman and Chebiyyam Siva Ram Murthy. In *Parallel Computers: Architecture and Programming*, pages 378–380, August 2004.
4. Ray S. Chen. *Finding Chapel's Peak: Introducing Auto-Tuning to the Chapel Parallel Programming Language.* November 2012.
5. I-Hsin Chung and Jeffrey K. Hollingsworth. *Using Information from Prior Runs to Improve Automated Tuning Systems.* November 2004.
6. Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. *An adaptive cut-off for task parallelism.* In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, November 2008.
7. Barcelona Supercomputing Center. OmpSs Specification, March, 30th 2017.