Auto-tuning Static Schedules for Task Data-flow Applications

Andreas Diavastos
Department of Computer Science
University of Cyprus
diavastos@cs.ucy.ac.cy

Pedro Trancoso
Department of Computer Science, University of Cyprus
CSE Department, Chalmers University of Technology
pedro@cs.ucy.ac.cy

ABSTRACT

Scheduling task-based parallel applications on many-core processors is becoming more challenging and has received lots of attention recently. The main challenge is to efficiently map the tasks to the underlying hardware topology using application characteristics such as the dependences between tasks, in order to satisfy the requirements. To achieve this, each application must be studied exhaustively as to define the usage of the data by the different tasks, that would provide the knowledge for mapping tasks that share the same data close to each other. In addition, different hardware topologies will require different mappings for the same application to produce the best performance.

In this work we use the synchronization graph of a task-based parallel application that is produced during compilation and try to automatically tune the scheduling policy on top of any underlying hardware using heuristic-based Genetic Algorithm techniques. This tool is integrated into an actual task-based parallel programming platform called *SWITCHES* and is evaluated using real applications from the *SWITCHES* benchmark suite. We compare our results with the execution time of predefined schedules within *SWITCHES* and observe that the tool can converge close to an optimal solution with no effort from the user and using fewer resources.

KEYWORDS

Task Parallelism, Data-flow, Auto-tuning, Genetic Algorithm

ACM Reference Format:
Andreas Diavastos and Pedro Trancoso. 2017. Auto-tuning Static Schedules for Task Data-flow Applications. In *ANDARE '17: 1st Workshop on AutotuniNgSystems, September 9, 2017, Portland, OR, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3152821.3152879

1 INTRODUCTION

In this new many-core era we observe a trend for increasing number of cores in a processor as the way to deliver computational power in an efficient and scalable way. Programming models and runtime systems are adapting to this new reality as to allow users to fully exploit the available resources. Nevertheless, programming and managing resources for these systems is becoming an increasingly difficult task.

ANDARE '17, September 9, 2017, Portland, OR, USA

ACM ISBN 978-1-4503-5363-2/17/09…$15.00
https://doi.org/10.1145/3152821.3152879

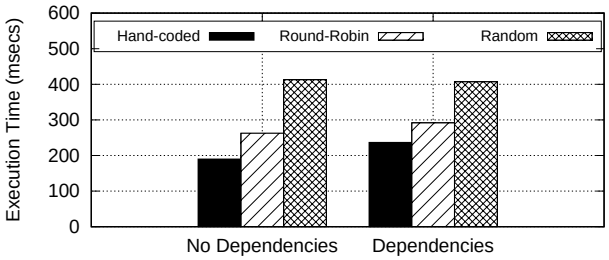

Figure 1: Execution time of *Round-Robin* and *Random* policies compared to hand-coding an optimum schedule for synthetic kernels implemented in *SWITCHES*. This test was performed on an Intel Xeon Phi using 240 threads.

Designing a parallel program has its degree of difficulty. Several programming models have been proposed to alleviate this problem. The current trend that seems to be more appropriate to exploit large degree of parallelism in an application is to specify the program as a set of tasks which may also be related with each other by data-flow dependences. Many runtime and programming systems today support the task-based model of execution with the most widely known system to be the latest release of OpenMP v4.5 [14]. Analyzing the original code and generating these tasks and dependences is already a difficult task. Nevertheless, as systems increase their scale, it is not only the number of processing elements that increase but also the heterogeneity of the system as a whole. Thus, the allocation of tasks to resources becomes a huge challenge. The challenge is not only to address runtime dynamic behavior of the tasks but also to determine a schedule of the tasks that results in the best execution time, as for example, a task that produces data that is consumed by another task should be co-located in the same resource or placed nearby as to avoid or reduce the data transfer overhead. Given the complexity of the underlying infrastructure and of the synchronization graph representing an application, this mapping is a difficult problem to solve.

Figure 1 shows the performance of *Round-Robin* and *Random* schedules for two synthetic task-based kernels (one without dependences and one with dependences between a number of tasks), compared to hand-coding an optimum schedule. Hand-coding an efficient schedule of any application requires a highly experienced programmer with significant knowledge of the application characteristics (tasks, dependences and data usage) and the underlying hardware. Results show that in such a scenario, execution time can be reduced by as much as $2x$. Nevertheless, the complexity and the variety of both applications and hardware systems increases with time and make the *Handcoded* scenario an unfeasible solution.

Therefore, for future complex applications and large-scale parallel systems, we propose to use a machine-learning approach as to determine a task-to-resource mapping, that will take into account both the application and the hardware characteristics. We propose a technique that uses a Genetic Algorithm (GA) where the population is composed of different task-to-resource assignment schedules and the generations evolution is guided by the evaluation metrics (e.g. execution time or energy consumption) as to converge to the best schedule, depending on the metric defined by the user. We integrate the GA with an actual parallel programming and runtime system, thus the iteration over the different steps of the optimization such as creation of new schedules, execution and evaluation, are executed in an autonomic way, on real applications. The final schedule is determined automatically by the process, but this is not without cost as it requires the execution of the application several times. In our work we choose carefully the initial population so that the GA converges faster towards the better schedule. We foresee this optimization step as being done either ahead of the execution of the real application, in an initialization/setup phase of the system and application tuning, or as part of a scenario where the same application is to be executed multiple times on the same system and each time it executes its metrics results are stored and used by the GA to improve the schedule for the future runs.

The main contribution of this work is the integration of a well known GA (NSGA-II [16]) in a real parallel programming and runtime task-based system (*SWITCHES* [5]) that offers an auto-tuning scheduling tool for parallel applications. This integration, allows the sharing of the application synchronization graph with the auto-tuning tool to use it to extract information about the tasks, their dependences and their data usage. The *SWITCHES* runtime system uses a static scheduler and avoids any interference in the execution of other runtime overheads as to isolate the auto-tuning process for better understanding of the results. The auto-tuning tool is designed and implemented to optimize schedules for execution time, power consumption and temperature. Because the NSGA-II algorithm used is a multi-objective GA, the tool also supports the optimization of schedules for any combination of the above metrics.

We tested the proposed approach on a real system using real applications and we observed that the auto-tuning tool was able to select the best schedule for most of the scenarios with improvements to the best default original schedules. It also manages to find solutions that produce the same performance while using less resources than the default original schedules.

This paper is organized as follows: in Section 2 we briefly describe the *SWITCHES* programming platform, in Section 3 we describe the GA implemented for this work and in Section 4 we describe how we integrated the two into a single tool. In Section 5 we present our evaluation of this integration, while in Section 6 we present other GA-based scheduling works from the literature. Finally, in Section 7 we outline our conclusions.

2 THE SWITCHES SYSTEM

SWITCHES [5] is a task-based system that uses concepts from the Data-flow [3, 4] execution model in order to produce high-levels of application parallelism. The *SWITCHES* runtime system is built to satisfy two major requirements: (1) the scalability of application performance, and (2) the reduction of runtime overheads. To achieve scalability, it implements a fully de-centralized runtime by evenly distributing the scheduling operations to all software threads. To reduce runtime overheads, tasks are assigned to threads statically at compile-time. In addition, each task holds its own scheduling structures that will be loaded in the scheduler during execution. Its' design requires minimum support for dynamic scheduling of tasks, consequently reducing runtime overheads.

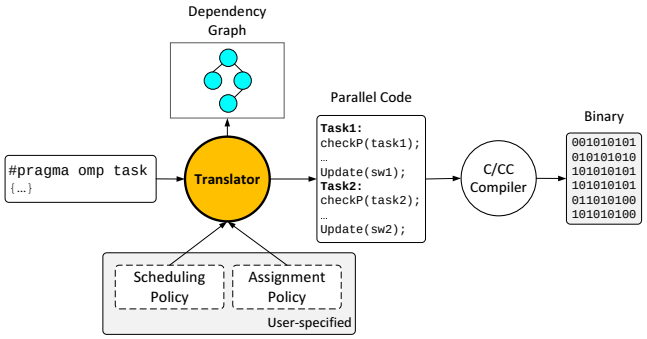

Figure 2: The process of translating an application using the *SWITCHES* Translator [5].

SWITCHES provides an API that is an extension to the latest OpenMP v4.5 [14]. Applications are written in C/C++ with tasks and dependences declared using the OpenMP compiler directives. The translation of a directive-based application to a *SWITCHES* parallel program is automatically done by a source-to-source tool. This tool, called the *Translator*, reads the input source code embedded with the directives, analyses the code and produces a Synchronization Graph (SG) with the tasks and dependences of the application. Finally, it produces the parallel source code to be compiled by any commodity compiler and executed on the target platform (see Figure 2). The user can provide as input to the *Translator*, the *scheduling policy*, that defines how tasks are divided to participating software threads, *the assignment policy*, that denotes how software threads are mapped to hardware resources, and *the number of threads*, that will execute the application.

We chose *SWITCHES* as our baseline programming platform because it provides static scheduling and assignment policies that can be defined at compile time through the *Translator*. This gives us the ability to create different scheduling policies (i.e. generating schedules using a GA) for an application before its execution and load them to the *Translator* during compilation. The *Translator* will then automatically produce parallel code with the provided schedule. *SWITCHES* is an open-source platform that can be downloaded from [1].

3 THE NSGA-II GENETIC ALGORITHM

A Genetic Algorithm is a heuristic procedure that tries to find the optimal solution to a presented problem. The main principle of a GA is that crossing two individuals can result an offspring that is better than both the parents. Also, a slight mutation of the produced offspring can generate better individuals. The crossover mating takes two individuals of a population as inputs and generates two new

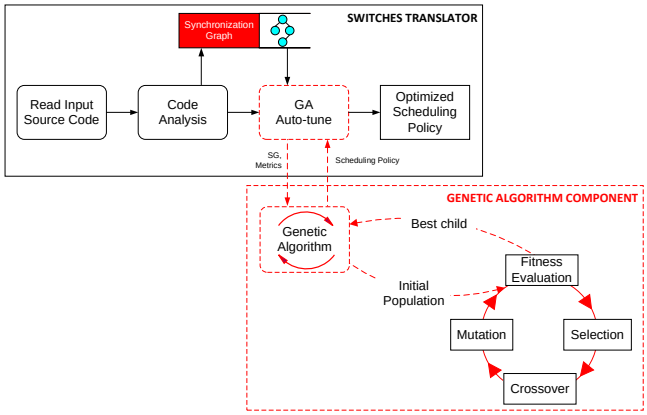

Figure 3: The design of the auto-tuning tool and how it is integrated in the *SWITCHES* Translator.

offsprings. This way some of the parent characteristics are maintained in individual offsprings in new generations. The mutation randomly transforms an offspring that was also randomly chosen from the set of all new offsprings produced by the crossover process. Finally, the best solutions are selected using a fitness function and transferred as inputs to the next generation and the next crossover mating. A fitness function is defined upon the problem that the GA is trying to solve and the best individual corresponds to the one having the best fitness value. For example, in most scenarios tested in this work the fitness function detects the smaller execution time therefore, the best individual corresponds to the one with the smallest execution time.

A GA is a loop that starts with an initial population and evolves through generations using a selection followed by a sequence of crossovers and a sequence of low-probability mutations. The loop can terminate either by a limit on the total number of iterations or the stability of the results defined by the fitness evaluation function.

To optimize the scheduling of the tested applications in this work, we used an already existing and well known multi-objective genetic algorithm, the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [2]. The NSGA-II was proposed in order to address the main disadvantages of the previous NSGA algorithm proposed in [16]. The original NSGA algorithm suffered from high computational complexity ($O(MN^3)$), lack of elitism and the need for specifying the shared parameter. The NSGA-II alleviates the above disadvantages and presents a solution with a fast non-dominated sorting approach with $O(MN^2)$ computational complexity.

The NSGA-II is using a selection operator that creates a mating pool by combining the parent and offspring populations and selecting the best (with respect to the fitness function) N solutions. Being a multi-objective algorithm the NSGA-II can also use more than one parameter to its fitness function. It can combine the values of 2 or more variables as to select the best individuals within a population. This can be particularly useful in scheduling problems where more than one parameters might be important in the execution of a task-based application (e.g. power-performance efficiency).

We omit the details of the NSGA-II implementation [2, 8] from this paper as we are using it unmodified for this work.

Table 1: Experimental workloads description and data set sizes.

Benchmark	Description	Complexity
MMULT	Matrix multiply [17]	1024 · 1024
RK4	Differential equation [17]	19200
Poisson2D	5-point 2D stencil computation [18]	16384 · 128
No-Dependences	Random generated tasks	1024 · 100
Dependences	Random generated tasks with dependences	1024 · 100

4 AUTO-TUNING SCHEDULING

To create the auto-tuning scheduling tool, we integrated the NSGA-II algorithm inside the *SWITCHES Translator*. Figure 3 shows the design of this integration and how the GA works together with the *Translator* to produce an optimized schedule for any input application. The directive-based source code of the application is given to the *Translator* as it would normally happen for a *SWITCHES* program. The *Translator* then analyzes the code and produces its SG. The SG is then passed to the Genetic Algorithm Component (GAC), that implements the NSGA-II. The GAC also needs some parameters that are required for the GA execution. These parameters are: (i) the number of generations to execute the algorithm, (ii) the size of the population for each generation, (iii) the objectives that the fitness function will use to evaluate each schedule and (iv) the mutation and crossover probabilities.

The number of generations and the size of each population can be explicitly defined by the user. The values are based on the cost tolerance accepted by the user for a specific application. The larger the size of a population or the more generations requested, the more time it will take for the auto-tuning tool to finish and produce a schedule that is optimal. The fitness objectives are used by the GAC to evaluate each produced schedule and rank it in the current population. The objectives currently supported by the tool are performance (execution time), power consumption and processor temperature. The tool ranks the produced schedules in a population based on the objective chosen by the user. As mentioned in Section 3, the NSGA-II is a multi-objective algorithm that allows using multiple objectives to decide the classification of the fitness evaluation. Therefore, the user can chose more than one objective to be considered for the evaluation fitness of the population. Finally, the mutation and crossover probabilities are usually decided empirically as they are affected by the problem that is to be solved.

When the GAC receives the SG and the GA parameters, it produces an initial population with random schedules. As an optimization we include the default *SWITCHES* [5] schedules in the initial population. The GAC then starts executing the application with each schedule in the population and stores the evaluation results when the execution is finished (*fitness evaluation*). At the end of each generation, each schedule is ranked based on the objective/s requested. If, for example, the objective is performance, the schedules are ranked in ascending order, starting with the schedule that produces the smallest execution time. At the *Selection* stage, the GAC implements a tournament selection process [11] that uses this ranking to decide which schedules in the population should be used for the *Crossover*, that are then used to create the population of the next generation. When the new generation is created, a *Mutation*

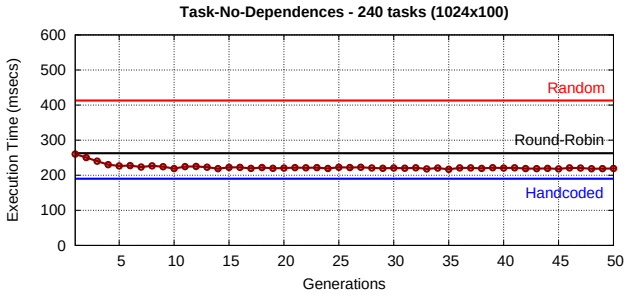

Figure 4: The execution times for the *No-Dependences* synthetic kernel. The dotted line shows the results obtained by the auto-tuning tool for 50 generations. The produced schedule converges within 15% of the optimal scenario, while it uses 30% less computational resources.

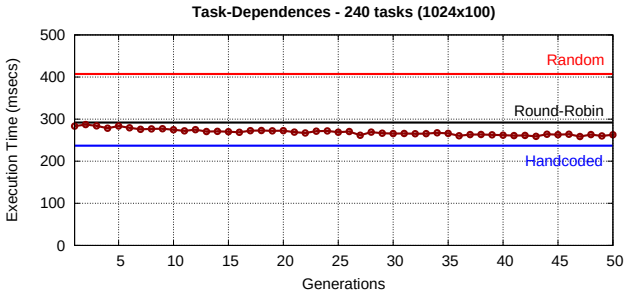

Figure 5: The execution times for the *Dependences* synthetic kernel. The dotted line shows the results obtained by the auto-tuning tool for 50 generations. The produced schedule converges within 10% of the optimal scenario, while it uses 30% less computational resources.

function is applied to the population that alters one or more values of a schedule. The mutation process is useful to maintain the genetic diversity from one generation to the next. Based on the mutation probability, it is likely that one or more schedules don't change from one generation to the next.

When the GAC execution reaches the limit of generations defined by the user, the best ranked policy is identified and passed back to the *Translator* to produce the parallel source code. This schedule is also stored in a text file, that can be loaded by the *Translator* at a later time.

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

This auto-tuning tool is evaluated on a set of 3 data- and task-parallel real applications from the *SWITCHES* evaluation suite. We also tested two computation kernels with randomly generated task graphs (*No-Dependences* and *Dependences*). The former allows fully parallel execution of the tasks without imposing any kind of dependences between them, while the latter represents a synthetic kernel with randomly generated dependences between tasks. Details for all the application and their input sizes are shown in Table 1. The complexity column represents the total number of computation iterations each application executes on its data.

The data-parallel applications used are : (1) MMULT, implements a matrix multiplication algorithm [17] and (2) RK4, solves a differential equation [17]. The task-parallel application used is Poisson2D, a 5-point 2D stencil computational kernel of the Poisson equation from the KASTORS Benchmark suite [18].

Our evaluation platform is an Intel Xeon Phi 7120P with 61 cores and 4 threads per core (totaling 244 hardware threads). Note that we only used 60 cores as to avoid any interference with the OS that always uses the last core of the system. This board has a total of 16GB of main memory and runs at 1.238GHz. To cross-compile applications for the Xeon Phi we use the Intel icc v.17.0.2 compiler (and the corresponding libiomp5 library) with the *-mmic* flag indicating the Many Integrated Architecture (MIC) target and the *-O3* optimization flag. The results are presented as execution time that is collected using the `gettimeofday` system call that provides a resolution of microsecond. The time is measured from the start of the first parallel function until the last, including all runtime costs (such as thread creation and scheduler initialization).

The parameters used for the GA algorithm are 10 generations for the real applications tested and 50 for the synthetic kernels. Each generation has a population of 64 schedules. The mutation and crossover probabilities are 0.0001 and 0.6 respectively. We decided on these values based on our study of the literature and small scale testing scenarios with various other values that show that beyond these values, no significant difference is observed. It is important to notice that different problems require different values, therefore for new applications it is important to test various options.

5.2 Synthetic Applications

We evaluated the synthetic kernels using two *SWITCHES* predefined policies, *Round-Robin* that assigns equal number of tasks to all threads in a round-robin way and *Random* that assigns the tasks to the threads in random way. We also analyzed the synthetic kernels to find the data usage of the tasks in both of them and the task dependences in the second kernel. We used this information to create a *Handcoded* scheduling policy that we deem as the optimal scenario. Results of these executions are shown in Figures 4 and 5. Running the synthetic kernels through the auto-tuning tool shows that it can converge close to the optimal scenario (within 15% for the *No-Dependences* and 10% for the *Dependences* scenarios) and in both cases achieves better performance than the original predefined *SWITCHES* policies. We observe that the kernel with the task dependences requires more generations of the algorithm and this happens due to the dependences between the tasks and complexity of its SG.

In addition, an important outcome of these scenarios is that the schedule produced by the auto-tuning tool that is near the optimal *Handcoded* schedule is achieved by using 30% less computing resources. Scenarios *Random*, *Round-Robin* and *Handcoded* are using all 240 hardware threads of the system, while the schedule of the auto-tuning tool is using 168 hardware threads for the *No-Dependences* and 166 hardware threads for the *Dependences*.

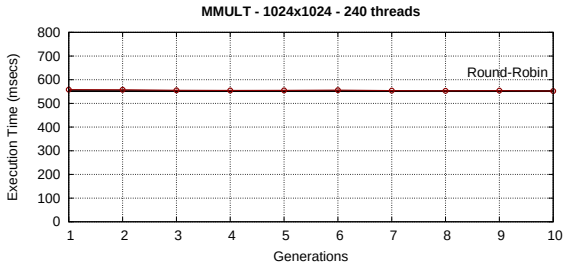

Figure 6: The performance achieved by the auto-tuning tool for MMULT is the same as what is already achieved by the *Round-Robin* policy of *SWITCHES*.

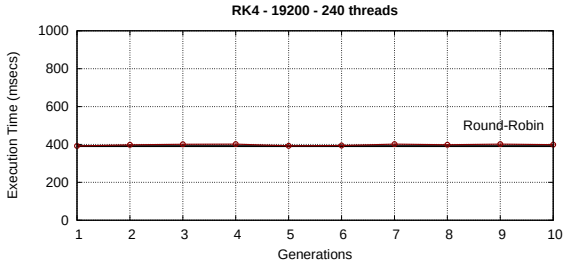

Figure 7: The performance achieved by the auto-tuning tool for RK4 is the same as what is already achieved by the *Round-Robin* policy of *SWITCHES*.

5.3 Data-Parallel Applications

In this scenario we used two data-parallel application from the *SWITCHES* benchmark suite, MMULT and RK4. The results are shown in Figures 6 and 7 respectively. The data set of both applications is equally divided to all tasks with most of the tasks using their own data. In the cases that tasks share data, they are shared in a consecutive way. That is, consecutive tasks share data from consecutive memory locations. Therefore, the best policy is to assign tasks in a consecutive way. This is exactly what the *Round-Robin* policy does and this is why we see no extra benefit achieved from the auto-tuning tool. Finally, in contrast to the synthetic kernels presented earlier, for these data-parallel applications, the auto-tuning tool chooses to use all hardware resources available to achieve the performance shown.

5.4 Task-Parallel Applications

In this scenario we used the Poisson2D, a task-parallel application from the *SWITCHES* benchmark suite. This application achieves its highest speedup with the default *SWITCHES* schedule at 32 hardware threads (on an Intel Xeon Phi system). It also shows significant performance loss for any number of threads greater than 32. In Figure 8 we run the auto-tuning tool for only 32 hardware threads and observe a small performance improvement compared to the default *SWITCHES* schedule. Studying the policy produced by the auto-tuning tool, we see that it chooses to use hardware threads that belong to the same core, while the *Round-Robin* policy in this case is using separate cores for each of the 32 hardware threads

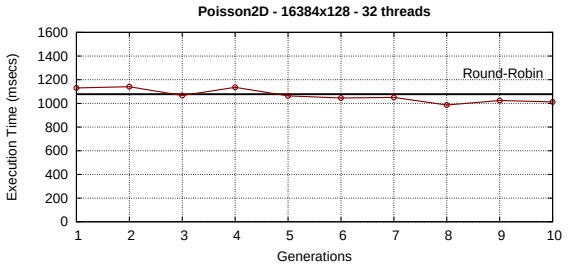

Figure 8: The auto-tuning tools slightly reduces the execution time by using hardware threads from the same cores, in contrast to the *Round-Robin* policy that uses different cores.

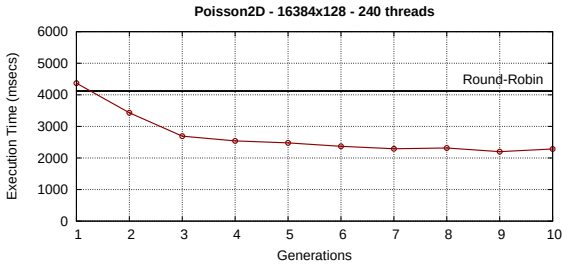

Figure 9: The execution time of Poisson2D when using all available resources is reduce by 2× when using the auto-tuning tool. The tool also achieves this by using only 70% of the total hardware resources.

used. The auto-tuning tool chooses to place depended tasks that share the same data on the same cores and thus minimizes the data transfer overhead.

As explained earlier, using all the hardware resources for Poisson2D with *SWITCHES* results in decreasing performance due to algorithmic limitations of the application and inefficient assignment of the tasks by the *SWITCHES* default policy. Figure 9 shows the results of the auto-tuning tool for a Poisson2D execution using 240 threads. The auto-tuning tool finds a schedule that significantly increases the performance and achieves a 2× improvement compared to the default *Round-Robin* policy. Similar to the case of the synthetic kernels presented earlier, this results is achieved by using 30% less hardware resources.

5.5 Seed Optimization

As explained in Section 4, in its initial population the auto-tuning tool includes the default schedules that *SWITCHES* can produce. We found that using the default schedules as a starting point for the auto-tuning tool reduces the size of the search space of the GAC and improves the convergence speed to an optimal solution. This is especially important in many-core systems where the search space of the auto-tuning tool (*i.e.* the number of resources) will be very large, as in such a case the number of combinations of resources that create a possible schedule will also be large. Figure 10 shows results of this optimization tested with MMULT.

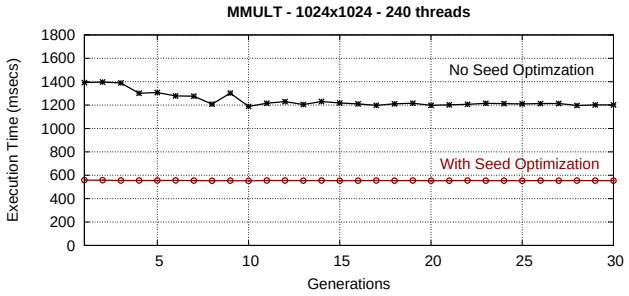

Figure 10: The auto-tuning tools performs better when we include the *SWITCHES* predefined assignment policies in its initial population.

5.6 Summary

Summarizing our results, we conclude that the auto-tuning tool:

- Improves the performance of task-based applications with data access patterns that are driven by complex dependences, as it chooses to place depended tasks that share the same data on the same cores as to minimize the data transfer overhead;
- Can achieve maximum performance using fewer resources than what is available and what is used by the default policies, therefore improving the efficiency of the system.

6 RELATED WORK

The task scheduling problem is a well studied field and appears numerous times in the literature. Focusing our background search in GAs or even the more general heuristic-based solutions for task scheduling problems we observed a large number of algorithms proposed to solve different scheduling problems [6, 7, 9, 10, 12, 13, 15, 19, 20]. All these algorithms are implemented and tested on multi-processor systems. Some are targeting heterogeneous systems with computation units with variable capabilities and other were proposed for homogeneous parallel systems. What all these have in common though is that they are all tested using randomly generated task graphs and are implemented and evaluated as simulations. In contrast, in our work we implement a GA within a parallel programming and runtime system that allows for using it with real applications and producing schedules for real hardware systems.

7 CONCLUSIONS

In this work we developed a tool for auto-tuning task-based parallel applications by taking into account the tasks to be executed and their dependences. We achieved this by integrating a well-known GA (NSGA-II) within the *SWITCHES* programming tool (the *Translator*). *SWITCHES* uses static schedules to execute the tasks in parallel, thus allowing us to feed the *Translator* with any policy we wish during compilation. This allowed us to implement a GA that produces schedules that are fed to the *Translator* and evaluated at the same time.

While the auto-tuning tool comes with the cost of executing the application several times, it will produce an optimal schedule without any user knowledge of either the application or the target system. Our results show that performance is improved, especially for application with complex dependences whose data accesses are driven by task dependences. We also observe that maximum performance can be achieved by using significantly less resources than what is available and what the default scheduling policies use. This improves the efficiency of the execution as higher performance with fewer resources results in a reduction of the power consumption.

REFERENCES

[1] Andreas Diavastos. 2017. SWITCHES Platform. https://github.com/diavastos/SWITCHES. (2017). [Online].
[2] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (Apr 2002), 182–197. https://doi.org/10.1109/4235.996017
[3] J. B. Dennis. 1974. First version of a data flow procedure language. In *Programming Symposium*. Springer, 362–376.
[4] J. B. Dennis and D. P. Misunas. 1975. A Preliminary Architecture for a Basic Data-flow Processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture (ISCA '75)*. ACM, New York, NY, USA, 126–132.
[5] Andreas Diavastos and Pedro Trancoso. 2017. SWITCHES: A Lightweight Runtime for Data-flow Execution of Tasks on Many-cores. *ACM Trans. Archit. Code Optim. 14, 3, Article 31 (August 2017), 23 pages* (2017). https://doi.org/10.1145/3127068
[6] H. M. Ghader, K. Fakhr, M. Javadi, and G. Bakhshzadeh. 2010. Static task graph scheduling using learner Genetic Algorithm. In *2010 International Conference of Soft Computing and Pattern Recognition*. 357–362. https://doi.org/10.1109/SOCPAR.2010.5686731
[7] S. Gupta, V. Kumar, and G. Agarwal. 2010. Task Scheduling in Multiprocessor System Using Genetic Algorithm. In *2010 Second International Conference on Machine Learning and Computing*. 267–271. https://doi.org/10.1109/ICMLC.2010.50
[8] Kanpur Genetic Algorithms Laboratory. 2011. Multi-objective NSGA-II code in C. http://www.iitk.ac.in/kangal/codes.shtml. (2011). [Online].
[9] Kamaljit Kaur, Amit Chhabra, and Gurvinder Singh. 2010. Heuristics based genetic algorithm for scheduling static tasks in homogeneous parallel system. *International Journal of Computer Science and Security (IJCSS)* 4, 2 (2010), 183–198.
[10] T. Lewis and H. El-Rewini. 1993. Parallax: a tool for parallel program scheduling. *IEEE Parallel Distributed Technology: Systems Applications* 1, 2 (May 1993), 62–72. https://doi.org/10.1109/88.218176
[11] Brad L Miller, David E Goldberg, et al. 1995. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems* 9, 3 (1995), 193–212.
[12] M. M. Najafabadi, M. Zali, S. Taheri, and F. Taghiyareh. 2007. Static Task Scheduling Using Genetic Algorithm and Reinforcement Learning. In *2007 IEEE Symposium on Computational Intelligence in Scheduling*. 226–230. https://doi.org/10.1109/SCIS.2007.367694
[13] R. A. Al Na'mneh and K. A. Darabkh. 2013. A new genetic-based algorithm for scheduling static tasks in homogeneous parallel systems. In *2013 International Conference on Robotics, Biomimetics, Intelligent Computational Systems*. 46–50. https://doi.org/10.1109/ROBIONETICS.2013.6743576
[14] OpenMP Architecture Review Board. 2015. OpenMP 4.5 API C/C++ Syntax Reference Guide. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. (2015). [Online].
[15] S. Pei, J. Wang, W. Cui, L. Jiang, T. Geng, J. L. Gaudiot, and S. Zuckerman. 2016. Codelet Scheduling by Genetic Algorithm. In *2016 IEEE Trustcom/BigDataSE/ISPA*. 1492–1499. https://doi.org/10.1109/TrustCom.2016.0233
[16] N. Srinivas and Kalyanmoy Deb. 1994. Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms. *Evolutionary Computation* 2, 3 (1994), 221–248. https://doi.org/10.1162/evco.1994.2.3.221
[17] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso. 2008. TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems. In *37th International Conference on Parallel Processing*. 25–34.
[18] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier. 2014. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *Proceedings of the 10th International Workshop on OpenMP (IWOMP 2014)*. 16–29.
[19] Yun Wen, Hua Xu, and Jiadong Yang. 2011. A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system. *Information Sciences* 181, 3 (2011), 567 – 581. https://doi.org/10.1016/j.ins.2010.10.001
[20] A. Y. Zomaya, C. Ward, and B. Macey. 1999. Genetic scheduling for parallel processor systems: comparative studies and performance issues. *IEEE Transactions on Parallel and Distributed Systems* 10, 8 (Aug 1999), 795–812. https://doi.org/10.1109/71.790598