An Introduction to Autotuning Parallel Applications

Pedro Bruel
phrb@ime.usp.br

DCC - IME
Universidade de São Paulo
São Paulo, June 28, 2015

Contents

1 Introduction

A number of factors make programming **H**igh-**P**erformance **C**omputing applications a very complex task. To achieve high performance it is necessary to write programs that take into consideration the specifics of a parallel architecture, as well as the basic concepts of parallel programming. Parallel computing platforms are increasingly available and heterogeneous. Processors are now expected to be multi-core, and even personal computers have coprocessors and accelerators such as GPUs. Consequently, it is easy to over-optimize a parallel program to a specific parallel computing platform.

This leads to highly efficient hand-optimized programs, capable of leveraging the potential of a specific parallel computing platform. Although, despite the efforts made during implementation, this highly-optimized program will not able to achieve the same performance in a different parallel architecture. In other words, the performance achieved by extensive hand-optimization is not *portable*. The lack of performance portability of HPC applications, as well as the immense efforts needed to optimize them, justify the research for automated methods of optimization for parallel programs.

Extensive research in the last decade produced powerful tools for automatically tuning programs, or *autotuners*, in a variety of problem domains. Despite being able to achieve good results on porting the performance of parallel programs,these tools are not widely utilized by the scientific and programming communities, perhaps because of their novelty.

Autotuners can be model-based or empirical. Model-based autotuners predict program performance following a model of the target architecture. The model is built previous to the tuning and execution, during an *installation* phase. Empirical autotuners discover the best optimization of a program by executing different optimized versions and measuring their performance. No model is built or used, instead empirical autotuner commonly use search techniques to explore the space defined by the possible optimizations of a program, in a given architecture. Despite being arguably slower than model-base autotuners, given a model is already built, empirical autotuners can produce optimized versions of a program during tuning time. The empirical tuning strategy allows programmers to focus their efforts in designing programs that explicitly expose their implementation and optimization choices, leaving the task of discovering the best choice to the autotuner.

The search techniques used by an autotuner can target the optimization of various program metrics. Simple metrics such as runtime can be obtained by simply running the candidate optimization, but more complex metrics, such as I/O patterns or memory accesses are not so straight-forward to measure. *Profilers* are tools that offer such specific measurements and analysis of programs during runtime. For instance, if a programmer was interested in optimizing the CPU load, or the duration and depth of a function call stack during the execution of her program, she could use a profiler to produce such measurements to an empirical autotuner.

This Introduction to Parallel Tuning aims to familiarize the reader with the state-of-the-art applications of autotuning techniques to the optimization of high-performance parallel programs. The following section discuss the autotuning technique under the Algorithm Selection framework. Section 2.2 discusses the INSIEME compiler project for parallel applications. Section 2.3 describes the OpenTuner framework, an autotuning framework that can be used to implement autotuners for parallel applications. Section 3 list some profiling tools that can be used to measure metrics of parallel programs. Finally, section 4 summarizes the discussion.

2 Autotuning

The idea behind empirical autotuning techniques consists of using the performance-impacting features of architectures, problems and algorithms in a domain to define sets of possible algorithm implementations, configurations and optimizations. These sets describe a *search space* that can then be explored by searching, optimization and Machine Learning techniques.

In contrast, model-based autotuning uses perfomance-impacting features to build a model, and then uses this model to autotune an application. The model can be obtained by formally describing an algorithm and analysing its behaviour or, when building a model for a computer architecture, empirically exploring the search space. Models are then used to implement highly optimized libraries for a given domain, that are then used to implement programs in that domain. Different compiled versions of the library functions must be provided for all target architectures.

Instead of manually exploring the search space of algorithm optimizations, configurations and implementations, programmers should be motivated to expose these features to an autotuner. This approach to designing and implementing computer programs is called *Programming by Optimization*, and was first described by Hoos [10].

Autotuning techniques have been used since as early as 1997, when the PHiPAC system [3] used code generators and search scripts to automatically generate high performance code for matrix multiplication. Since then the autotuning problem has been tackled in multiple domains, with a great variety of strategies. Whaley *et al.* [18] introduce the ATLAS project, that produced optimized dense matrix multiply routines. The OSKI [17] library provides automatically tuned kernels for sparse matrices. The FFTW [8] library provides tuned C subroutines for computing the Discrete Fourier Transform.

More recently, there have been efforts in the direction of breaking the domain boundaries, through the implementation of tools that generalize autotuning. PetaBricks [1] is a language, compiler and autotuner for domain-independent applications, that introduce new abstractions such as the *either...or* keywords, which let programmers define multiple algorithms for a same problem. The user can define *input features* [6] for a given algorithm, which allow Petabricks to group and specifically optimize programs for classes of training inputs.

The OpenTuner framework [2] implements ensembles of search techniques that are used to search a user-defined space of program configurations. OpenTuner is further discussed in section 2.3. The framework has already been used to implement a domain specific language for data-flow programming [4] and a framework for the optimization of recursive parallel algorithms [7]. The ParamILS framework [11] implements state-of-the-art search methods for algorithm configuration and parameter tuning.

In an effort to provide a common representation of multiple parallel programming models, the INSIEME compiler project [12] implements abstractions for OpenMP, MPI and OpenCL. The INSIEME compiler is able to generate optimized parallel code for heterogeneous multi-core architectures. Section 2.2 discusses the INSIEME compiler project.

The remaining of this section presents the *Algorithm Selection Problem* and describes autotuning as an instance of this problem.

2.1 The Algorithm Selection Problem

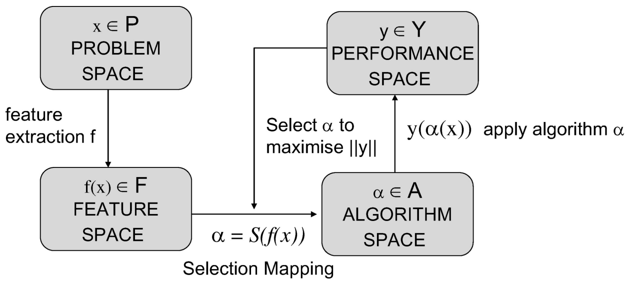

Figure 1: Schematic diagram of the Algorithm Selection Problem as described by Rice [15]. Reproduction of a diagram from Smith-Miles [16].

The description of the Algorithm Selection Problem was first published by Rice in 1976 [15]. The problem is described as follows. Given a set A of *algorithms*, a set P of *problems* and a set F of *problem features*, the Algorithm Selection Problem consists of finding a *mapping* of algorithms to problems that minimizes the time to solve all problems in the set, taking problem features in consideration. The *performance space* Y is composed of the measurements of each algorithm $\alpha \in A$ in each problem $x \in P$.

Note that the *algorithms* that compose a set are not limited to configurations or combinations of algorithms. Each $\alpha \in A$ can represent different abstractions, such as programs, heuristics, or configurations. The set of *problems* usually contains instances of a problem, and the set of *problem features* contains representations of the performance-impacting features. Ding et al. [6] further discusses the feature extraction processes. Figure 1 shows a schematic diagram for the Algorithm Selection Problem, as described by Rice.

Kotthoff [14] presents a survey of the algorithm selection field and its applications in combinatorial search problems. Smith-Miles [16] surveys the applications of the algorithm selection problem to the machine learning and meta-learning fields.

The Algorithm Selection Problem is hard. Its NP-completeness has been proved when calculating static distributions of algorithms in parallel machines [5]. Its Undecidability in the general case was also shown [9]. Therefore, methods such as *Stochastic Local Search* can be used to approach the algorithm selection problem with good results, as described by Hutter et al. [11].

2.2 INSIEME Compiler Project

The INSIEME Compiler Project[1] of the University of Innsbruck is a C/C++ source-to-source compiler that aims to automatically optimize parallel programs for heterogeneous parallel computing platforms, regardless of the parallel programming tools used in the implementation of the programs.

INSIEME [12] uses the INSPIRE intermediate representation [13] to commonly represent parallel programs in various parallel programming abstractions, such as OpenMP, MPI and OpenCL. INSIEME also offers a runtime system for online tuning and performance monitoring.

An overview of the architectures of the INSIEME compiler and runtime systems is presented in Figures 2 and 3. Those diagrams can be found at the INSIEME Compiler Project website[2].

[1] insieme-compiler.org
[2] insieme-compiler.org/architecture.html

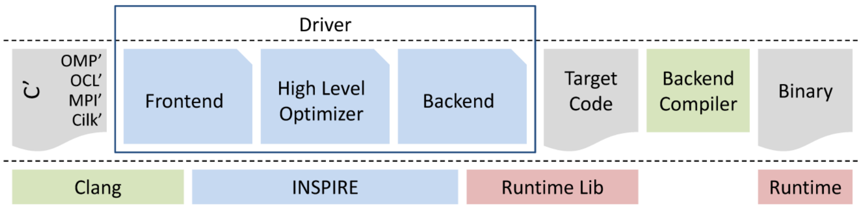

Figure 2: Overview of the INSIEME compiler.

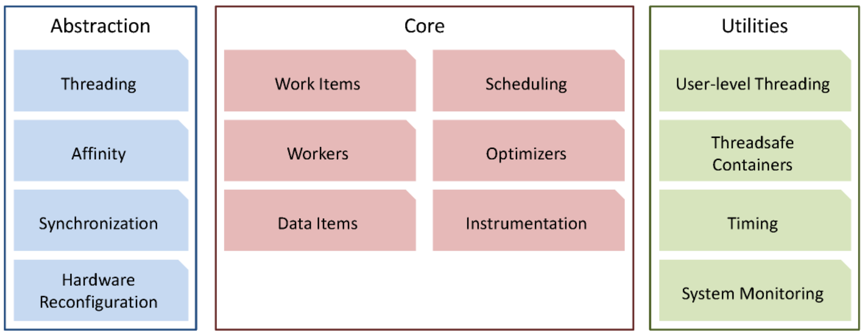

Figure 3: Overview of the INSIEME runtime.

The runtime uses multiple threads to schedule and distribute work. The optimization and scheduling choices are made based on previous executions of the program and on knowledge of the underlying parallel computing platform.

2.3 OpenTuner Framework

The OpenTuner framework [2] provides domain-agnostic tools for defining search spaces and implementing autotuners.

The search spaces are defined by instantiating the different *Parameter* types provided by the framework. Each parameter type, such as *FloatParameter*, *IntegerParameter* or *BooleanParameter* implements its own manipulation functions, that allow the search techniques to navigate the values allowed to each parameter, thus exploring the search space defined by the user. A user of the framework can implement her own parameter types.

OpenTuner implements ensembles of optimization and Machine Learning techniques that perform well in different problem domains, and are used to search user-defined search spaces. The results found during the search process are shared between techniques through a common results database. OpenTuner uses *meta-techniques* for coordinating the distribution of resources between techniques in an ensemble.

An OpenTuner application can implement its own search techniques and meta-techniques, adding them to existing ensembles or creating completely new ones. By separating search space definition from search method implementation, OpenTuner allows fast implementation of autotuners for different problem domains.

The search techniques implemented in OpenTuner share the results found through a common database. This allows techniques to benefit from each other's executions and reach better results faster.

Figure 4 illustrates the main components of the framework, the interactions between them. Note that both the Search and Measurement components read from and write to the Results Database. This communication of intermediate results effectively steers the autotuning process.

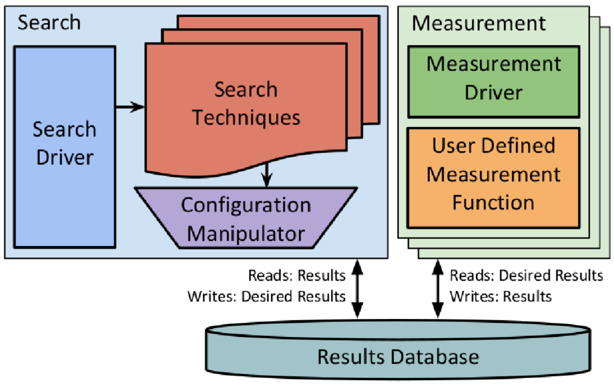

Figure 4: Main components of the OpenTuner framework and their interactions. Reproduction of a diagram found in Ansel et al. [2].

3 Profiling Tools

System	Features	License
perf tools	Sampling profiler supporting hardware events on several architectures.	GPL
LTTng	Collects data on processes blocking, context switches, and execution time. Helps identify performance problems over multiple processes or threads.	GPL
gprof	Several tools with combined sampling and call-graph profiling. A set of visualization tools, VCG tools, uses the Call Graph Drawing Interface (CGDI) to interface with gprof.	BSD
Allinea MAP	I/O, communication, floating point operation usage, memory access costs, MPI and OpenMP support.	Proprietary
Streamline	Graphical performance visualization of hardware and software for ARM CPUs, Mali GPUs, OpenCL, power consumption metrics.	Proprietary
VTune	Serial and threaded performance analysis. Hotspot, call tree and threading analysis on both Intel and AMD x86 processors. Hardware event sampling that uses the on chip performance monitoring unit requires an Intel processor.	Proprietary
CodeXL	GPU and CPU profilers, GPU debugger and static kernel analyzer.	Proprietary
NVIDIA Visual Profiler	Performance profiling for optimizing CUDA C/C++ applications.	Proprietary

Figure 5: Some profilers for parallel applications and their features.

Figure 5 show some of the available profiling tools for parallel applications[3] and their main features. Note that there are not many open source profiling tools.

[3]Filtered from wikipedia.org/wiki/List_of_performance_analysis_tools.

The INSIEME Compiler Project references three important benchmarks for parallel applications which can be used as performance profilers. The INNCABS[4] cross-platform and cross-library offer benchmarks for C++ parallel constructs. uCLbench[5] measures various performance metrics of OpenCL CPU and GPU applications, through a set of micro-benchmarks. MPICacheBench[6] measures the effects of CPU caches in point-to-point and collective MPI operations.

4 Conclusion

This work presented a limited overview of some of the most recent research in parallel applications autotuning, and listed some parallel profiling tools and benchmarks. This work is intended to continuously evolve, and future work will extend the description of the INSIEME and OpenTuner systems to contain samples of implementations and performance analysis. Experiments with profiling tools and benchmarks, targeting different optimization objectives, will also be included. Finally, a new section describing autotuning applications and domain-specific autotuning systems will be included.

[4]github.com/PeterTh/inncabs
[5]github.com/PeterTh/uCLbench
[6]github.com/motonacciu/mpi-cache-bench

References

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. 2013.

[3] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 253–260, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2840-1. doi: 10.1145/2591635.2667174.

[4] Jeffrey Bosboom, Sumanaruban Rajadurai, Weng-Fai Wong, and Saman Amarasinghe. Streamjit: a commensal compiler for high-performance stream programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 177–195. ACM, 2014.

[5] Marin Bougeret, P-F Dutot, Alfredo Goldman, Yanik Ngoko, and Denis Trystram. Combining multiple heuristics on discrete resources. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[6] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. 2014.

[7] David Eliahu, Omer Spillinger, Armando Fox, and James Demmel. Frpa: A framework for recursive parallel algorithms. Master's thesis, EECS Department, University of California, Berkeley, May 2015.

[8] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[9] Haipeng Guo. *Algorithm selection for sorting and probabilistic inference: a machine learning-based approach*. PhD thesis, Citeseer, 2003.

[10] Holger H Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.

[11] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

[12] Herbert Jordan, Peter Thoman, Juan J Durillo, Sara Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12. IEEE, 2012.

[13] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. Inspire: The insieme parallel intermediate representation. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 7–17. IEEE, 2013.

[14] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *arXiv preprint arXiv:1210.7959*, 2012.

[15] John R Rice. The algorithm selection problem. 1976.

[16] Kate A Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41(1):6, 2008.

[17] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[18] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X.