Towards Co-evolution of Auto-tuning and Parallel Languages

Ray S. Chen
Department of Computer Science
University of Maryland
College Park, MD USA
rchen@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland
College Park, MD USA
hollings@cs.umd.edu

1. INTRODUCTION

There exists a gap between the massive parallelism available from today's HPC systems and our ability to efficiently develop applications that utilize such parallelism. Technologies such as POSIX threads, OpenMP, and MPI communication primitives are notoriously difficult to use and worse to debug [2]. Several emerging programming languages aim to narrow this gap by providing high-level abstractions for data and/or control parallelism, effectively shifting the responsibility of tedious low-level tasks (e.g. data decomposition) from the programmer to the compiler. However, this model requires the compiler to make performance altering decisions based on information unavailable at compile time, such as workload specific information or number of hardware threads available on a given node. If such decisions could be deferred until run time, it becomes possible to introduce auto-tuning to quickly find optimal, or near optimal values on the target architecture itself.

Meanwhile, adapting auto-tuning to an existing program typically involves the labor-intensive process of identifying what can be auto-tuned. Some parallel languages provide constructs that aid with this process. With minor language modification, identification can become fully automatic.

Our work aims to improve both auto-tuning and parallel languages by co-evolving them to work together.

2. LANGUAGE SUPPORT

Key constructs already exist within Chapel [1], an emerging parallel programming language developed by Cray, Inc., that can be considered a first steps towards co-evolution. Specifically, Chapel supports the notion of "configuration variables," which are simply variables whose value may be overridden at load time. In the context of co-evolution, configuration variables are especially useful for controlling performance related aspects of an application. Consider TCP socket buffer size; calculating the optimal value at compile time is difficult, if not impossible. In Chapel, the programmer is free to declare it as a configuration variable, choose a reasonable default, and move on.

Additionally, application developers are not the only source of configuration variables. The Chapel compiler itself adds two configuration variables to every program that affect data decomposition, `dataParTasksPerLocale` and `dataParMinGranularity`. These variables determine how data processed in parallel is divided among executing threads. This makes every Chapel program a candidate for auto-tuning, and we use these variables for our experiments.

Supercomputing '13 Denver, Colorado USA

3. AUTO-TUNING SUPPORT

With language support in place, the auto-tuner must evolve to take advantage of tunable variables within a target application. To this end, we implemented Tuna as a component of the Active Harmony auto-tuning framework. Tuna executes a target application multiple times in succession, testing specific command-line parameters with each iteration, and sending an associated performance value to the Active Harmony framework. Internally, Active Harmony uses simplex-based gradient-free numerical optimization methods to search for optimal configurations while only empirically testing a small fraction of the search space. This functionality is designed to work in concert with Chapel's configuration variables which may be set from the command-line.

Furthermore, the auto-tuner cannot be a rigid system. The domain of viable auto-tuning targets is limited only by the flexibility of the auto-tuning framework. And so, we developed a plug-in interface to allow for modular execution within our framework. This allows Active Harmony to quickly support any novel tuning opportunities.

4. VIABILITY EXPERIMENTS

To demonstrate the viability of our model, we experiment with Chapel's implementation of the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) proxy application [3], which solves a simple Sedov blast problem. We ran our experiments on a single node, 8 core (16 thread) Intel Sandy Bridge Xeon E5-2670 with 32 GB of RAM. The GNU compiler (version 4.6.3) was used to build Chapel (version 1.7.0) with Qthreads enabled. A wide range of values were considered for `dataParTasksPerLocale` (8 through 128 by steps of 4) and `dataParMinGranularity` (128 through 4096 by steps of 128) on problem sizes of 32^3 and 48^3.

Wall time is the metric for our experiment, and so we wish to minimize the performance value. We begin by performing an exhaustive search of the parameter space to produce a baseline. The resulting surface plots show two important

properties. Firstly, performance varies significantly based solely on the choice of these two variables. A drop of over 75% is seen between the best and worst running time within a single surface plot. Secondly, the optimal point depends upon the program's input, $(88, 2176)$ for problem size 32^3 and $(104, 1792)$ for problem size 48^3. Additionally, using the optimal point from problem size 32^3 for an execution of problem size 48^3 or vice versa would result in a performance loss of 13% or 47%, respectively.

Finally, using Tuna, we are able to achieve 24% and 14% improvement over the default parameters for problem sizes of 32^3 and 48^3, respectively. Both searches converge after only 12 search steps. To show that this performance improvement scales, we increase the problem size by an order of magnitude (128^3) and still achieve a 13% performance improvement over the default.

5. GRAND VISION

These experiments demonstrate the utility of auto-tuner and parallel language co-evolution. We believe wider support for auto-tuning constructs within parallel languages will enable HPC programmers to productively focus on issues of logic rather than performance. Moreover, tighter integration with auto-tuning can enable performance gains within a single execution [4], bringing parallel languages that much closer to closing the performance gap.

6. REFERENCES

[1] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *in Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '04)*, pages 52–60, 2004.

[2] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz. Parallel programmer productivity: A case study of novice. In *Parallel Programmers, International Conference for High Performance Computing, Networking and Storage*, 2005.

[3] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, December 2012.

[4] A. Tiwari and J. K. Hollingsworth. Online adaptive code generation and tuning. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 879–892, Washington, DC, USA, 2011. IEEE Computer Society.