Adaptive Tuning in a Dynamically Changing Resource Environment

Seyong Lee and Rudolf Eigenmann*
School of ECE, Purdue University
West Lafayette, IN, 47907
{lee222,eigenman}@purdue.edu

Abstract

We present preliminary results of a project to create a tuning system that adaptively optimizes programs to the underlying execution platform. We will show initial results from two related efforts. (i) Our tuning system can efficiently select the best combination of compiler options, when translating programs to a target system. (ii) By tuning irregular applications that operate on sparse matrices, our system is able to achieve substantial performance improvements on cluster platforms.

This project is part of a larger effort that aims at creating a global information sharing system, where resources, such as software applications, computer platforms, and information can be shared, discovered, and adapted to local needs.

1 Opportunities of Today's Computing Environment

1.1 Global Resource Sharing

Modern computing and information systems have progressed far beyond the single-workstation environment that contains all resources accessible to the user. Today, it is possible to find data, programs, and compute resources from across the world. Many of these resources can be accessed, used, and included into a new piece of work as if they resided locally. Owners interested in sharing such resources can announce their availability to the world with little effort.

The effort that some of these publish, discovery, and use actions take can be small, so that one may no longer be aware of the sophisticated, underlying mechanisms. For example, we have become accustomed to using Google – a thousand miles away – to search for information that may already lay on top of our desk. However, in other situations, using remote resources can be tedious, because the way they are published and accessed is not agreed upon, their performance and availability fluctuate, or security concerns impose constraints.

The work described in this paper ultimately aims at increasing this ease of use, thus exploiting the opportunities of today's computing environments to a higher degree. We envision that any resource – program, computer system, and data – can be published by its owner in a way that is as easy as writing a web page. Resources need not satisfy constraints to become eligible for publication; instead, the owner can advertise their features. Others can discover these resources with common tools and mechanisms. Similar to today's web search facilities, such discoveries search in open space, without the need to enter proprietary or constrained directories and brokers.

1.2 Adapting Discovered Resources to Local Needs

The flexibility of publication and the openness of search pose a substantial challenge. Discovered resources are unknown and thus untrusted. Discovered software modules may be equally untested and their advertised features may be faulty. Discovered computer systems may be accessed by many users, leaving their availability and use questionable. Discovered applications may run well on one platform, but inefficiently on others.

We address this challenge with a new Advanced Adaptive Tuning System, ATUNE, that works in concert with existing publication, discovery, and use mechanisms. This environment adaptively tunes available parameters in discovered resources, gradually achieving the best possible working point. Thus, when developing software, discovered modules combine in a way that achieves the programmer's goal most efficiently; applications adaptively find the best among the discovered execution platforms, and they recompile and tune dynamically to this platform; as the load on the network changes or new network resources become available, programs adaptively choose a new best working

*This work is supported in part by the National Science Foundation under grant CNS-0720471

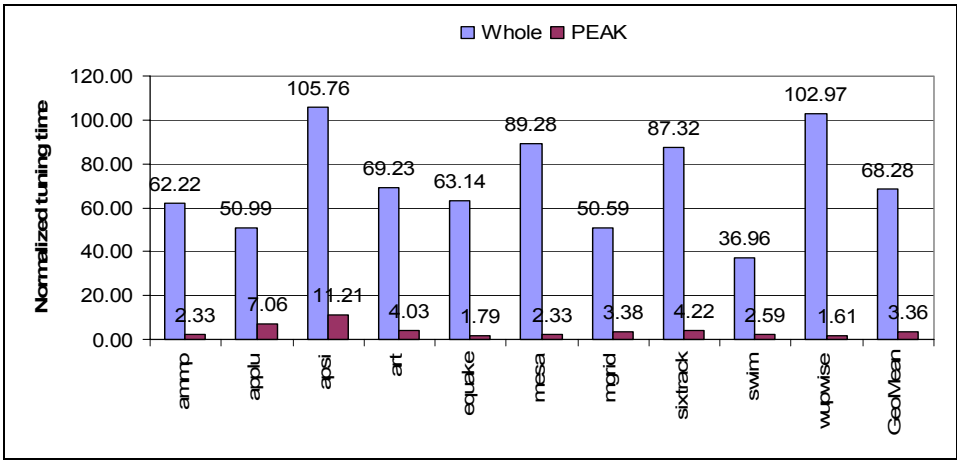

Figure 1. Reduction of Tuning Time Through Offline, Subroutine-Level Tuning. By measuring and tuning individual subroutines, rather than the whole program, our offline tuning engine exploits the fact that subroutines are invoked many times in the course of a program. Hence, a large number of tuning actions can be performed in one program execution, leading to a 20-fold decrease in tuning time, on average, compared to whole-program tuning of the SPEC FP2000 programs. (Normalized tuning time indicates the number of times the program needs to be run for tuning to complete.)

point.

Our work builds on prior results that have created dynamically adaptive program optimizations [8, 7, 6, 13] and an Internet sharing system that serves as both an integration framework and testbed for the new, combined techniques [10, 11].

2 ATUNE Overview

Our ATUNE system consists of a tuning engine that is integrated into a system for resource sharing. The iShare Internet sharing system provides the overall environment [10, 11]. It includes facilities to publish, discover, and use software and hardware resources.

The tuning engine builds on our prior work with dynamic and adaptive compilation [8, 13]. This work was the first to create a fully functional, integrated system for adaptively recompiling and tuning an application during its execution, using standard compilers [13]. It also showed that it can navigate a large optimization parameter search space more efficiently and effectively than others [12, 9].

The specific goal behind the described work is to enable *roaming applications*. In a simple case of publishing an application for use by others, the owner may specify a single platform that is capable of executing the application. By contrast, roaming applications are those that are able to run on a range of platforms. To this end, the execution system must be able to automatically install a newly discovered application on a newly discovered computer platform. For efficient operation, the installation is followed by a tuning process that aims at executing the applications in an optimal way.

We describe two contributions towards such a tuning process. They include (i) an *offline tuning* system that finds the best compilation options for a given application and platform and (ii) a dynamically adaptive system that tunes irregular parallel applications as they run on a compute cluster.

3 Offline Tuning of Compilation Options

In an initial project, we have implemented a tuning engine that optimizes programs in an *offline* manner, similar to profile-driven optimization. A key component of the tuning system is its algorithm that performs an empirical search on a large space of program transformation techniques. At each point in the search, the engine decides on the better combination of techniques by compiling the code, dynamically inserting it into the running application, and comparing execution times. Other important components of this mechanism are the timing comparison methods and the method for best selection of subroutines for efficient tuning. Details of these techniques are described in [8, 7, 6].

Figure 1 shows the tuning time for finding the best combination of the 38 GCC compiler options that are part of the highest optimization level. The two bars compare the tuning method that measures overall program timing with the one that tunes individual subroutines. There is a 20-fold reduction in optimization time, owing to the fact that tuning actions are performed at every subroutine invocation, rather

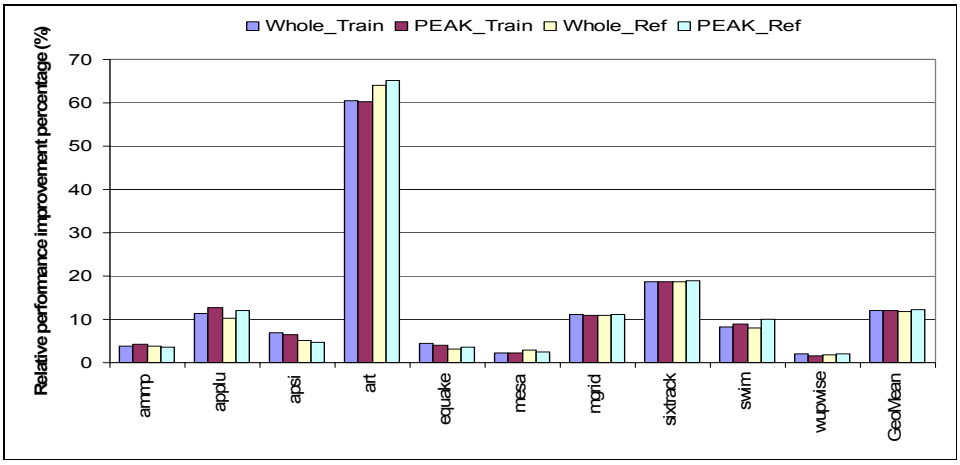

Figure 2. Performance Improvement Through Offline Tuning. The bars show tuning at the whole-program level and the subroutine level; "train" and "ref" indicate the program performance of these respective benchmark data sets. ("train" is always used during the tuning process). The data sets and whole vs. subroutine tuning result in similar performance. The geometric mean over the SPEC FP2000 benchmarks shows 12% improvement over the highest GCC optimization level (-O3).

than once per program run.

Figure 2 shows the performance improvements of our offline tuning system. On average, the program performance improves by 12% over the highest optimization level `-O3`. For individual programs, we have found up to 60% improvement. We have found that several of these improvements are due to subtle interactions of optimization techniques, and thus they are not commonly discovered by users experimenting with reasonable combinations of compiler options. The improvements are also not predictable by common performance models. In these cases, our empirical search mechanism appears to be superior to other optimization methods.

4 Dynamically Adaptive Tuning of Irregular Cluster Applications

This section presents an adaptive runtime tuning system for distributed parallel irregular applications. For irregular applications, such as sparse matrix-vector (SpMV) multiplication kernels, static performance optimizations are difficult because memory access patterns may be known only at runtime. Extensive studies have been conducted to improve irregular applications on single-processor or shared memory systems [14]. There has been a focus on architecture-oriented techniques, such as register blocking and cache blocking. While these techniques may be applied on distributed systems to tune individual nodes, they do not propose parallel distributed optimizations.

On distributed parallel applications, load balancing and communication cost reduction are two key issues. To address these issues, many graph-partitioning-based decomposition algorithms [2, 1] and decomposition heuristics [5, 15] have been proposed. They generally aim at distributing computations as evenly as possible or minimizing communication volume through preprocessing steps or static runtime allocations. However, these static methods do not capture dynamic runtime factors, such as compute speed differences among the assigned nodes and interconnection characteristics, which affect the performance of these parallel applications.

In this project, we have implemented a runtime tuning mechanism that can be applied dynamically to adapt distributed parallel irregular applications to the underlaying environments. No preprocessing steps are necessary. The proposed tuning system consists of two techniques: *iteration-to-process mapping* based on normalized iteration execution time and *runtime selection of communication algorithms*. The first technique achieves computational load balance by mapping iterations to processes dynamically, based on the measured execution time; the second technique finds the best possible communication method for message patterns generated by our adaptive mapping mechanism. In contrast to previous static runtime allocation approaches [5, 15], the proposed tuning system re-distributes the workload dynamically. In this way, our tuning system attains improved load balance.

A key component of the runtime tuning system is its algorithm that reduces incurred measuring and communication overheads. The tuning system approximates each iteration execution time using a normalization technique. To increase accuracy, the normalized execution time is recalculated whenever the iteration-to-process mapping is

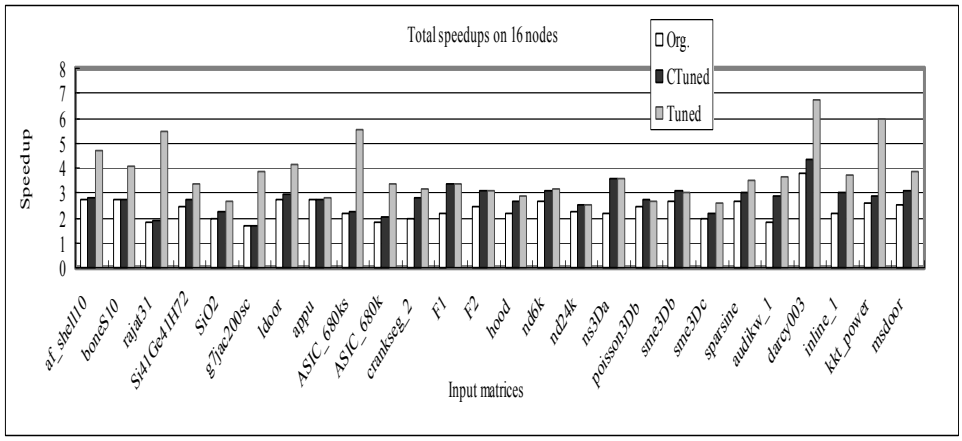

Figure 3. Performance Improvement Through Adaptive Runtime Tuning. The bars show the speedups of the base parallel version (*Org*), computation-tuning only version (*CTuned*), and tuned version (*Tuned*) on 16 nodes. Experiments on 26 real sparse matrices show that our adaptive mapping (*CTuned*) reduces execution times up to 37.8% (14% on average) and the overall tuning system(*Tuned*) reduces execution times up to 66.7% (33.3% on average) on 16 nodes.

Table 1. Execution time reduction by the proposed tuning system. In *A(B)* format, *A* represents the average of 26 matrices and *B* is the maximum value.

	4 nodes	8 nodes	16 nodes	32 nodes	overall
Main time reduction (CTuned) (%)	17.2 (40.4)	22.1 (55.2)	20.7 (51.7)	16.3 (45)	19.1 (55.2)
Main time reduction (Tuned) (%)	27.8 (44.1)	38.8 (62.1)	46.1 (75.9)	53.2 (79.3)	41.5 (79.3)
Total time reduction (CTuned) (%)	14.9 (34.2)	16.9 (44.7)	14 (37.8)	9.6 (32)	13.8 (44.7)
Total time reduction (Tuned)(%)	23.9 (39.6)	30.6 (55.6)	33.3 (66.7)	36 (68.8)	30.9 (68.8)

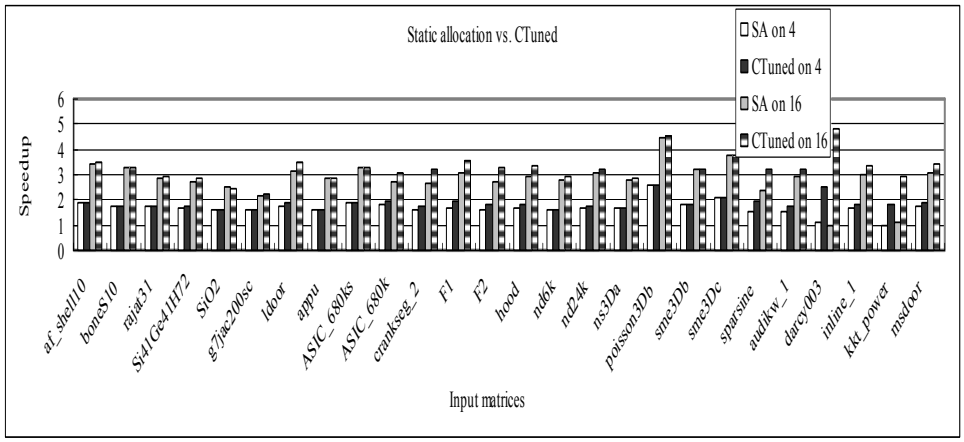

Figure 4. Performance Comparison of Static Allocation Method (*SA*) vs. Adaptive Iteration-to-process Mapping Method (*CTuned*) on 4 and 16 Nodes. This figure reveals that our adaptive mapping algorithm (*CTuned*) performs equally or better than the static allocation method (*SA*) in most cases.

changed, and the newly estimated value is used for the next re-mapping. For the runtime selection of communication methods, the runtime system inspects communication patterns generated by the mapping system and tries several communication algorithms to find the best method. More details of these techniques can be found in [4].

Figure 3 shows the performance improvements of our adaptive runtime tuning system. We applied our techniques to distributed parallel sparse matrix-vector (SpMV) multiplication kernels. Experiments were conducted on 26 real sparse matrices in the UF Sparse Matrix Collection [3], which cover a wide range of application areas, such as finite element methods, circuit simulation, and linear programming. These matrices have various non-zero data distributions from banded diagonals to random distributions, and their dimension sizes vary from 14K to 4.7M. The proposed tuning system reduces execution time up to 68.8% (30.9% on average) over base parallel SpMV kernels. Table 1 summarizes the execution time reductions measured for the 26 matrices. In the table, *main time* refers to the time to execute the main SpMV computation body, excluding the initial input data distribution phase.

Figure 4 compares our adaptive iteration-to-process mapping system with a static allocation method, which maps iterations to processes statically, such that non-zero elements in sparse matrices are distributed evenly. Figure 4 shows the speedups of both our mapping algorithm (CTuned) and the static allocation algorithm (SA) on 4 and 16 nodes. The figure demonstrates that our adaptive mapping system performs equally or better than the static allocation method, in most cases. In particular, for *darcy003* and *kkt_power*, the static method achieves little speedup on both 4 and 16 nodes, while our adaptive mapping performs significantly better.

5 Conclusion

We have presented results of two components of a larger adaptive tuning system that is able to optimize globally discovered resources to local needs. The first components is an offline tuning system that compiles programs automatically to a target platform, finding the best combination of optimization techniques. The second component is a tuning capability for optimizing irregular algorithms that operate on sparse matrices. These components tune programs on compute platforms, which makes them suitable for matching discovered software and hardware resources. The overall tuning system will include several additional components, such as composition functionality (enabling discovered software and hardware resources to combine in new ways), data integrators (allowing diverse data volumes to be discovered and form integrated databases), and history-based optimization methods (enabling resources to evolve with time).

References

[1] E. G. Boman and U. Catalyurek. Constrained fine-grain parallel sparse matrix distribution. *SIAM Workshop on Combinatorial Scientific Computing*, 2007.

[2] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed systems*, 10, July 1999.

[3] T. Davis. University of florida sparse matrix collection [online]. available: http://www.cise.ufl.edu/research/sparse /matrices/.

[4] S. Lee and R. Eigenmann. Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems. Technical Report ECE-HPCLab-08201, Purdue University, School of Electrical and Computer Engineering, High-Performance Computing Laboratory, 2008.

[5] S. G. Nastea and O. Frieder. Load-balancing in sparse matrix-vector multiplication. *Proceedings of 8th IEEE Symposium on Parallel and Distributed Processing*, page 218, 1996.

[6] Z. Pan and R. Eigenmann. Rating compiler optimizations for automatic performance tuning. In *SC2004: High Performance Computing, Networking and Storage Conference*, page (10 pages), Nov. 2004.

[7] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *The 4th Annual International Symposium on Code Generation and Optimization (CGO)*, page (12 pages), March 2006.

[8] Z. Pan and R. Eigenmann. Fast, automatic, procedure-level performance tuning. In *Proc. of Parallel architectures and Compilation Techniques*, pages 173–181, 2006.

[9] R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda, and H. A. G. Wijshoff. Statistical selection of compiler options. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*, pages 494–501, Volendam, The Netherlands, October 2004.

[10] X. Ren and R. Eigenmann. iShare - open internet sharing built on peer-to-peer and web. In *European Grid Conference*, pages 1117–1127, Feb. 2005.

[11] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Prediction of resource availability in fine-grained cycle sharing systems and empirical evaluation. *Journal of Grid Computing*, 5:173–195, 2007.

[12] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization*, pages 204–215, 2003.

[13] M. Voss and R. Eigenmann. High-Level Adaptive Program Optimization with ADAPT. In *Proc. of PPoPP'01: Principles and Practice of Parallel Programming*, Snow Bird, Utah, June 2001.

[14] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Proceedings of Supercomputing (SC)*, 2007.

[15] L. H. Ziantz, C. C. Ozturan, and B. K. Szymanski. Run-time optimization of sparse matrix-vector multiplication on SIMD machines. *Int. Conf. Parallel Architecture and Languages*, 817:313–322, July 1994.