Auto-tuning Parallel Programs at Compiler- and Application-Levels

Ananta Tiwari

July 14, 2009

Abstract

Auto-tuning has recently received its fair share of attention from the High Performance Computing community. Most auto-tuning approaches are specialized to work either on specific domains - dense/sparse linear algebra, stencil computations etc.; or only at certain stages of program execution - compile-time, launch-time or run-time. Real scientific applications, however, demand a cohesive environment that can efficiently provide auto-tuning solutions at all stages of application development and deployment. Towards that end, in this paper, we describe a unified end-to-end approach to auto-tuning scientific applications. A unique feature of our search-based auto-tuning system is a powerful parallel search algorithm, which leverages parallelism to effectively navigate the search space defined by compiler-level and application-level tunable parameters. Our system is general-purpose and the results presented in this paper demonstrate its applicability in tuning compiler-generated and application-specific input parameter spaces.

1 Introduction

The complexity and diversity of today's parallel architectures overly burdens application programmers in porting and tuning their code. At the very high end, processor utilization is notoriously low, and the high cost of wasting these precious resources motivates application programmers to devote significant time and energy to tuning their codes. This tuning process must be largely repeated to move from one architecture to another, as too often, a code that performs well on one architecture faces bottlenecks on another. As we are entering the era of petascale systems, the challenges facing application programmers in obtaining acceptable performance on their codes will only grow.

To assist the application programmer in managing this complexity, much research in the last few decades has been devoted to developing *auto-tuning* software that employs empirical techniques to evaluate a set of alternative mappings of computation kernels to an architecture and select the mapping that obtains the best performance. Auto-tuning software can be grouped into three categories: (1) compiler-based auto-tuners that automatically generate and search a set of alternative implementations of a computation [3, 18, 7]; (2) application-level auto-tuners that automate empirical search across a set of parameter values proposed by the application programmer [4, 13]; and, (3) run-time auto-tuners that automate on-the-fly adaptation of application-level and code-transformation level parameters to react to the changing conditions of the system that executes the application [1, 17]. What is common across all these different categories of auto-tuners is the need to *search* a range of possible implementations to identify one that performs comparably to the best-performing solution. The resulting search space of alternative implementations can be prohibitively large. Therefore, a key challenge that faces auto-tuners, especially as we expand the scope of their capabilities, involves scalable search among alternative implementations.

While it is important to keep advancing the start-of-art in auto-tuning software from the above three categories, we argue that full applications demand and benefit from a cohesive environment that can seamlessly combine these different kinds of auto-tuning techniques and that employs a scalable search to manage the cost of the search process. Towards that end, we introduce *parallel Active Harmony*, a search-based infrastructure that provides auto-tuning solution for all stages of application development and deployment. We provide a high-level overview of the auto-tuning framework in Section 2. We then present experimental results that show Active Harmony's promise in tuning compiler-level as well as application-level parameters.

2 Active Harmony

The design of the system was originally proposed by Hollingsworth et al [9] and since then multiple researchers [6, 5, 15] have advanced the state-of-art along several dimensions. The current design uses the client-server model. The client is the “harmonized” application, which measures some performance metric associated with the key computational elements of the application. The measurements are consumed by the server, which bases its adaptation decisions on performance data. “Harmonization” of an application involves making fairly small changes to the application code to export tuning options to the server.

At the heart of the Harmony framework is its parameter tuning algorithm. In the past, this algorithm has ranged from simple a greedy algorithm [9] to the Nelder-Mead Simplex algorithm [6]. Despite its popularity, the Nelder Mead algorithm has several weaknesses [12]. Consequently, to handle constrained high-dimensional parameter spaces, a more sophisticated algorithm is needed. To address this need, in our earlier work [16], we proposed a parallel search algorithm that can effectively deal with high-dimensional search spaces with an unknown objective function. The algorithm that we proposed, the Parallel Rank Ordering (PRO) Algorithm, belongs to a class of direct search algorithms known as the Generating Set Search (GSS) methods. GSS methods can effectively handle high-dimensional constrained spaces and have the necessary conditions for convergence. These algorithms can leverage parallelism to speedup convergence [10].

For a function of N variables, PRO maintains a set of kN points forming the vertices of a simplex in an N-dimensional space. Each simplex transformation step of the algorithm generates up to $kN-1$ new vertices by reflecting, expanding, or shrinking the simplex around the best vertex. After each transformation step, the objective function value associated with each of the newly generated points are calculated in parallel. The search stops when the simplex converges to a point in the search space or after a pre-defined number of search steps. One of the unique features that distinguishes the tuning framework presented in this paper from its earlier incarnations is a powerful parallel search algorithm. The algorithm leverages parallel architectures to search across a set of optimization parameter values. Multiple, sometimes unrelated, points in the search space are evaluated at each timestep. With this approach, we both explore multiple parameter interactions at each iteration and also have different nodes of the parallel system evaluate different configurations to converge to a solution faster. Furthermore, the tuning system also empowers application programmers to develop *self-tuning* applications that includes compile-time and run-time code transformations.

In the next two sections, we present the results of using our system to perform compile-time and application level input parameter tuning. Each section first motivates the need the to perform tuning at respective stages of application development and deployment.

3 Compiler-Based Tuning for Matrix Multiplication Kernel

Complex architecture features and deep memory hierarchies that characterize the state-of-art HPC platforms require applying nontrivial optimization strategies on loop nests to achieve high performance. Compounding these challenges is the fact that different loop optimizations usually have different goals, and when combined they might have unexpected (and sometimes undesirable) effects on each other. Even optimizations with similar goals but targeting different resources, such as unroll-and-jam plus scalar replacement targeting data reuse in registers, and loop tiling plus data copy for reuse in caches, must be carefully combined. The unroll factors must be tuned so that reuse in registers is exploited without causing register spilling or instruction cache misses. On the other hand, tiling plus data copying for reuse in caches changes the iteration order and data layout, and may affect reuse in registers. Given these complex interactions between the optimization strategies and the lack of precise analytical models, we resort to empirical tuning technique. Such technique involves performing a systematic search over a collection of automatically generated code-variants. We use Matrix Multiplication kernel to demonstrate how Active Harmony can effectively nagivate the search-space defined by loop optimization parameters.

The use of MM kernel for the experiments presented in this section was motivated by two goals. First, the optimization of the MM kernel has been extensively studied in the past and as such, we can easily compare the effectiveness of our approach to that of well-tuned MM libraries (e.g. Atlas). And second, the MM kernel exhibits the complex parameter interactions that were discussed earlier. Therefore, the results obtained for MM can be extrapolated to generic loop-nests beyond the realm of linear algebra. CHiLL [2], a polyhedra-based framework, is used to generate code-variants. CHiLL provides a high-level script interface that allows compilers and application programmers to use a common interface to describe parameterized code transformations to be applied to a computation. The optimization strategy for MM reflected in the CHiLL script in Table 1 exploits the reuse of `C(I,J)` in registers, and the reuse of `A(I,K)` and `B(K,J)` in caches. Data copying is applied to avoid conflict misses. The values for the five unbound parameters `TI`, `TJ`, `TK`, `UI` and `UJ` are determined by Active Harmony, which uses the PRO algorithm to navigate this five-dimensional

naive	*CHiLL – Recipe*	*Constraints*

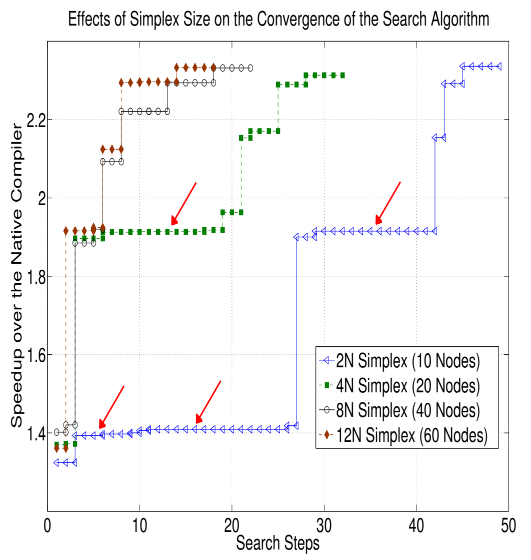
`DO K = 1, N`
`  DO J = 1, N`
`    DO I = 1, N`
`      C[I,J] = C[I,J]+A[I,K]*B[K,J]`
	`permute([3,1,2])`
	`tile(0,2,TJ)`
	`tile(0,2,TI)`
	`tile(0,5,TK)`
	`datacopy(0,3,2,1)`
	`datacopy(0,4,3)`
	`unroll(0,4,UI)`
	`unroll(0,5,UJ)`
		$TK \times TI \leq \frac{1}{2}\left(\frac{size_{L2}}{2}\right)$
		$TK \times TJ \leq \frac{1}{2}\left(\frac{size_{L1}}{2}\right)$
		$UI \times UJ \leq size_R$
		
		TI, TJ, TK
		$\in [0, 2, 4, \ldots, 512]$
		$UI, UJ \in [1, 2, \ldots, 16]$

Table 1: Matrix Multiplication Optimization

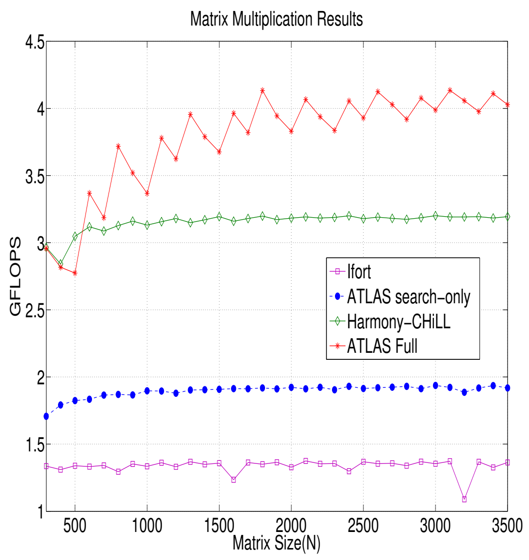
search space.

The experiments were performed on a 64-node Linux cluster (henceforth referred to as umd-cluster). Each node is equipped with dual Intel Xeon 2.66 GHz (SSE2) processors. L1- and L2-cache sizes are 128 KB and 4096 KB respectively. To study the effect of simplex size, we considered four alternative simplex sizes - `2N`, `4N`, `8N` and `12N`, where `N(=5)` is the number of unbound parameters. Figure 1(a) shows the performance of the best point in the simplex across search steps. Search conducted with `12N` and `8N` simplices clearly use fewer search steps than the search conducted with smaller simplices. The existence of long stretches of consecutive search steps with minimal or no performance improvement (marked by arrows in Figure 1(a)) in `2N` and `4N` cases show that more search steps are required to get out of local minimas for smaller simplices. At the same time, by effectively harnessing the underlying parallelism, `8N` and `12N` simplices evaluate more unique parameter configurations (see Table 2) and get out of local minimas at a faster rate. Results summarized in Table 2 also show that as the simplex size increases, the number of search steps decreases, thereby confirming the effectiveness of the increased parallelism.

The next question regarding the effectiveness of Active Harmony relates to the quality of the search result. Figure 1(b) shows the performance of the code variant produced by a `12N` simplex across a range of problem sizes along with the performance of native `ifort` compiler, Atlas' search-only and full version. In addition to a near exhaustive sampling of the search space, Atlas uses carefully hand-tuned BLAS routines contributed by expert programmers. Therefore, to make a meaningful comparison, we provide the performance of the search-only version of Atlas - code generated by the Atlas Code Generator via pure empirical search. Our code version performs, on average, 2.36 times faster than the native compiler. The performance is 1.66 times faster than the search-only version of Atlas. Our code variant also performs within 20% of Atlas' full version (with processor-specific hand coded assembly).



(a) Effects of Different Degree of Parallelism on PRO



(b) Results for MM Kernel

Figure 1: Performance of the Optimization Algorithm

Table 2: MM Results - Alternate Simplex Sizes

	$2N$	$4N$	$8N$	$12N$
Number of Function Evals.	276	571	750	961
Number of Search Steps	49	32	22	18
Speedup over Native	2.30	2.33	2.32	2.33

4 Application-Level Tuning on High-Performance Linpack Benchmark

Scientific applications and libraries use tunable input parameters that users can select at run-time to optimize the application's performance. These input parameters are meant to control several important aspects of the application performance such as data decomposition and alignment, numerical algorithm selection, communication protocol selection, etc. Choosing appropriate values for these parameters is essential in getting maximum application throughput. However, the task of making a good selection of the input parameters is non-trivial because this requires a concrete understanding of the interactions between the input parameters and the underlying algorithmic behaviors that they are meant to control. Moreover, the input parameters also interact with the elements of the target architecture. Under these conditions, it is practically impossible to manually tune the parameters and optimize application performance; therefore automated parameter tuning is the only plausible solution.

Parameter	**Domain**
$P{\times}Q$	Depends on the number of processors used (usually square grids are better)
N	Up to 80 % of the available memory, step size: 256
NB	32-256: step size: 2
pfact	left, right, crout
nbmin	2-10
ndiv	2-10
rfact	left, right, crout

Table 3: Tunable Input Parameters for HPL

To demonstrate the applicability and the effectiveness of our system in tuning application-level input parameters, we use the High-Performance Linpack (HPL) benchmark. Apart from demonstrating the strength of our auto-tuning framework on input-parameter search, the experimental results also establish the benefits of using our parallel search over Nelder-Mead simplex algorithm. Nelder-Mead simplex algorithm was used as the underlying search algorithm in the previous versions of Active Harmony.

HPL is a popular message-passing implementation of the Linpack benchmark. HPL solves an order N dense system of linear equations of the form $Ax{=}b$ using LU factorization. The matrix is divided into $NB \times NB$ blocks, which are then dealt onto a $P \times Q$ processor grid a using block-cyclic data distribution. HPL is built on top of the Basic Linear Algebra Subroutine (BLAS) package. We used high-performance Goto BLAS [8] in our installation. The performance of the system is measured in GFlops/second. This measurement is provided as a part of the program output. The goal is to select a good matrix size N and blocking factor NB to maximize this metric. In addition, HPL exposes 15 other input parameters that can be set at run-time to tailor the execution of the code on different platforms. However, with very coarse-grained instructions on how to set these parameters, users are left with no choice but to hand-tune the parameters. Thus, finding a good input configuration is tedious and can take substantial time. We use Active Harmony's offline tuning mechanism to automate the search for input parameters. Once the parameter space definition is exported to the Harmony server, no intermediate feedback is required to guide the search process. With no default input configuration available, this experiment provides a strict comparison between PRO and Nelder Mead algorithms.

We took note of previous research that studied the application behavior of HPL [14, 11]. These results suggested that not all HPL parameters have noticeable impact on performance. Of course, parameters that do not affect performance in one system might have significant impact on another.

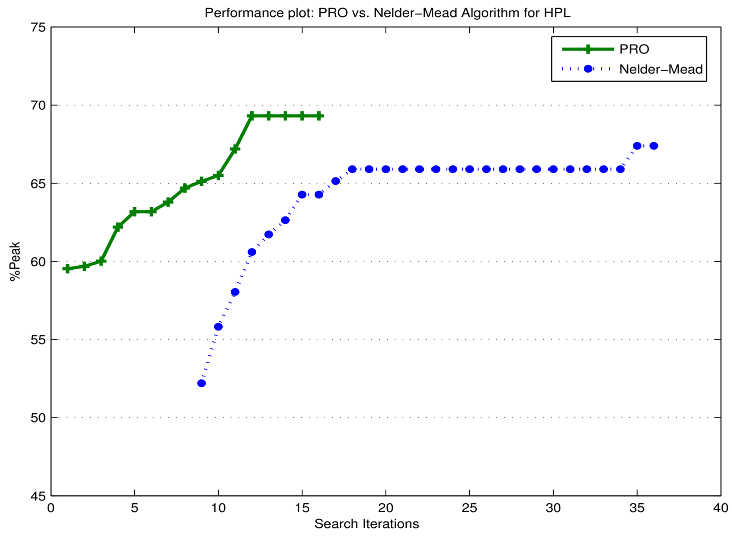


Figure 2: Comparison of best performing points of PRO and Nelder-Mead Algorithm

We conducted a parameter study to determine what parameters had significant impact on performance on our cluster. We vary only one parameter at a time to try and measure its importance. After trying all the applicable options for a given parameter (when the other parameters are fixed), if the HPL performance remains roughly within the noise (3% differential) for all options, we remove the parameter from our search space. After the study, we short-listed the parameters that have a noticeable impact on HPL performance on our system. This parameter list along with their domain values is provided in Table 4. *pfact* (and *rfact*) specifies panel (recursive) factorization method. *nbmin* specifies the number of sub-panels and *ndiv* specifies the number of columns in the recursive base case. The matrix size N should not exceed the amount of memory available across the nodes in the cluster. HPL guidelines suggest using 80% of available memory for the matrix to get the maximum performance leaving 20% for the Operating System and other background activities.

The experiment was conducted for 8 Nodes (16 CPUs). The performance metric (objective function) is R_{max}, which is the maximum measured HPL performance in GFlops/second. To calculate the efficiency of the system, we divide R_{max} by the theoretical peak performance, R_{peak}, for the system. R_{peak} is calculated by multiplying the total number of processors, the processor clock frequency and the theoretical number of 64-bit floating-point operations per clock. Figure 2 shows the iteration history of both PRO and Nelder Mead algorithms. PRO input configurations reached 69.3% of R_{peak} after 11 iterations and the algorithm evaluated 96 unique candidate configurations in the process. Meanwhile Nelder-Mead configurations could not get more than 65.9% of R_{peak} for the first 33 iterations. The performance slightly improved to 67.4% after 33 iterations. In conclusion, PRO finds a better input configuration 3 times faster than Nelder-Mead simplex algorithm.

5 Conclusion and Future Work

In this paper, we described a general-purpose tuning framework for scientific applications. The search-based tuning system is empowered with a parallel parameter tuning algorithm, which can take advantage of the available parallelism inherent in today's High Performance Computing systems. We showed that our tuning algorithm can effectively deal with high-dimensional search spaces. The fact that the search algorithm converges to solutions in only a few tens of search-steps while simultaneously tuning multiple parameters demonstrates its capability of taking into account the latent interactions between tunable parameters.

Going forward, our work will focus on making run-time tuning of parallel programs a practical goal. The goal is to design an infrastructure that will provide tuning options not only for symbolic parameters but also for parameters that require dynamic code-generation and compilation. Conceptually, this can be viewed as a merger of the traditional and just-in-time compilation.

References

[1] C. Cascaval, E. Duesterwald, P. Sweeney, and R. W. Wisniewski. Multiple page size modeling and optimization. *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 339–349, 17-21 Sept. 2005.

[2] C. Chen. *Model-Guided Empirical Optimization for Memory Hierarchy.* PhD thesis, University of Southern California, 2007.

[3] C. Chen, J. Chame, and M. W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2005.

[4] I.-H. Chung and J. K. Hollingsworth. Using Information from Prior Runs to Improve Automated Tuning Systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 30, Washington, DC, USA, 2004. IEEE Computer Society.

[5] I.-H. Chung and J. K. Hollingsworth. A Case Study Using Automatic Performance Tuning for Large-Scale Scientific Programs. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on High Performance Distributed Computing*, pages 45–56, 2006.

[6] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active harmony: towards automated performance tuning. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[7] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[8] Goto-Webpage. `http://www.tacc.utexas.edu/resources/software`. [last accessed: October 5, 2007].

[9] J. Hollingsworth and P. Keleher. Prediction and adaptation in Active Harmony. pages 180–188, Jul 1998.

[10] T. G. Kolda, R. M. Lewis, and V. Torczon. Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods. *SIAM Review*, 45(3):385–482, 2004.

[11] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 249–258, New York, NY, USA, 2007. ACM.

[12] K. I. M. McKinnon. Convergence of the Nelder–Mead Simplex Method to a Nonstationary Point. *SIAM J. on Optimization*, 9(1):148–158, 1998.

[13] Y. Nelson, B. Bansal, M. Hall, A. Nakano, and K. Lerman. Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[14] K. Singh, E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Predicting parallel application performance via machine learning approaches. *Concurrency And Computation: Practice and Experience*, 19(17):2219–2235, 2007.

[15] V. Tabatabaee and J. K. Hollingsworth. Automatic software interference detection in parallel applications. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[16] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel Parameter Tuning for Applications with Performance Variability. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 57, Washington, DC, USA, 2005. IEEE Computer Society.

[17] M. Voss and R. Eigenmann. ADAPT: Automated De-coupled Adaptive Program Transformation. *Parallel Processing, 2000. Proceedings. 2000 International Conference on*, pages 163–170, 2000.

[18] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate High-Performance BLAS? *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation*, 93(2):358–386, Feb. 2005.