Online Adaptive Code Generation and Tuning

Ananta Tiwari
Department of Computer Science,
University of Maryland,
College Park, MD, 20742
Email: tiwari@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science,
University of Maryland,
College Park, MD, 20742
Email: hollings@cs.umd.edu

***Abstract*—In this paper, we present a runtime compilation and tuning framework for parallel programs. We extend our prior work on our auto-tuner, Active Harmony, for tunable parameters that require code generation (for example, different unroll factors). For such parameters, our auto-tuner generates and compiles new code on-the-fly. Effectively, we merge traditional feedback directed optimization and just-in-time compilation. We show that our system can leverage available parallelism in today's HPC platforms by evaluating different code-variants on different nodes simultaneously. We evaluate our system on two parallel applications and show that our system can improve runtime execution by up to 46% compared to the original version of the program.**

***Keywords*-Auto-tuning; Parallel Search; Active Harmony**

I. Introduction

Today's complex and diverse architectural features require applying nontrivial optimization strategies on scientific codes to achieve high performance. As a result, programmers usually have to spend countless hours in rewriting and tuning their codes. Furthermore, a code that performs well on one platform often faces bottlenecks on another; therefore, the tuning process must be largely repeated to port the code to a new computing architecture.

Our research focuses on automating this tedious and error-prone process of tuning and porting parallel applications. In our earlier work [23], we showed that for well-defined benchmark kernels (such as matrix multiplication), compiler-based offline auto-tuning can deliver significant improvements over the optimizations offered by native compilers. However, for full applications, it is not as effective. Based on the input dataset, a given parallel application can have vastly different execution profiles. Input datasets can specify physical domain, solver type(s), solver parameters, discretization order, and so on. Taking an offline tuning approach to tune for all possible computational bottlenecks is not a tractable goal. Instead, on-demand tuning during production execution is a desirable approach. This on-demand approach also benefits from the availability of real-time performance data, which can be linked back to specific code sections and architecture-specific features.

Development of an online auto-tuner, however, presents its own set of challenges. Managing the cost of the search process and the cost of generating and compiling code-variants on-the-fly are two daunting challenges that runtime auto-tuners face. Addressing these challenges and making online tuning practical is the topic of this paper. Our goal is to enable application developers to write applications once and have the auto-tuner adjust the application execution automatically when run on new systems. *Our work can be viewed as the merger of traditional compiler techniques and just-in-time compilation.*

We take a search-based collaborative approach to auto-tuning. Our system, Active Harmony, allows application programmers, compilers, library writers and performance modelers to describe and export a set of performance related tunable parameters. These parameters define a tuning search-space. More often than not, this search-space is high-dimensional and exponential in size and thus, cannot be explored manually. Our system monitors the program performance and makes adaptation decisions. The decisions are made by a central controller using a parallel search algorithm. The parallel search algorithm leverages parallel architectures to search across a set of optimization parameter values. Different nodes of a parallel system evaluate different configurations at each timestep.

In Active Harmony vocabulary, *harmonization* refers to the process of adding "hooks" in an application (using the Active Harmony API) to make it tunable. The process involves making fairly small changes to the application code to export tuning options to the server. The phrase *harmonized application* is used to refer to an application that uses Active Harmony to adapt its execution.

To summarize, in this paper, we make the following contributions:

1) An online auto-tuner that can recompile programs during a single execution.
2) Support for tuning multiple code-sections simultaneously.
3) Refinements of our parallel search algorithm to include a penalization technique.
4) A system design to support runtime code generation and code replacement for large scale parallel applications.
5) An empirical analysis that shows that even when training runs are not available, our system can improve performance of an application within a single execution.

II. Challenges and Requirements

In this section, we review some key challenges that runtime auto-tuners face and features of Active Harmony that address them.

A. Minimal overhead

The costs of using an online tuning system must be minimal. Otherwise, such costs can overshadow any benefit realized in application performance. If the performance of harmonized code is better (or at least not worse) than that of untuned version of the code, the minimal overhead objective is achieved.

B. Avoid "bad" regions in the search-space

Our earlier experience [8], [23] has shown that tuning search-spaces often exhibit a clear demarcation between "good" and "bad" regions. We observed that there are frequently many good points near the optimal point and that there is also often another region where the application performance is relatively worse. Such a topography argues for online auto-tuning systems that rapidly get applications out of bad regions of performance and into the good ones. Achieving optimal performance is a secondary goal once good performance is reached.

C. Need to coordinate tuning for multiple code-sections

Auto-tuning for full applications typically involves tuning for multiple code-sections from several libraries simultaneously. Therefore, an important question is how to coordinate this process. Left uncoordinated (for example, if each library had an embedded auto-tuner), it would be impossible to tell which change (and on what code-section) was actually improving the overall performance of the program. In fact, it is likely that one change might improve performance and the second hurt performance and the net performance benefit is little or none. There are a variety of approaches possible ranging from simple arbitration to ensure that only one code-section is tuned at a given instance to a fully unified system that allows coordinated simultaneous search of parameters originating from different code-sections.

In the Active Harmony project, we take the approach of a coordinated system to allow simultaneous tuning of multiple code-sections. We accomplish this by having each code-section expose its tunable parameters via a simple, but expressive constraint language (discussed below). A central search engine then uses these constraints to manage the evaluation of possible auto-tuning steps.

D. Constraint language

An important aspect of searching parameters is to allow the precise specification of valid parameters. Such specification could be as simple as expressing the minimum, maximum and initial values for a parameter. Sometimes not all parameters values within a range should be searched, so an option to specify a step function is useful. Likewise for parameters with a large range (i.e. a buffer that could be from 1K to 100Megabytes), it is useful to specify that parameter should be searched based on the log of the value.

Another critical factor is that not all parameters are independent. Frequently, there is a relationship between parameters (e.g. when considering tiling a two dimensional array, it is often useful to have the tiles be rectangles with a definite aspect ratio). To meet the needs of having a fully expressive way for developers to define parameters, we have developed a simple and extensible language (Constraint Specification Language, CSL) that standardizes parameter space representation for search-based auto-tuners. CSL allows tool developers to share information and search strategies with each other. Meanwhile, application developers can use CSL to export their tuning needs to auto-tuning tools.

TABLE I
A SIMPLE CSL SPECIFICATION

```
search space simple {
     # parameter definitions
     parameter x int {
       range [1:8:1];
     }
     parameter y int {
       range [1:8:1];
     }
     parameter z int {
       range [1:8:1];
     }
     # constraints
     constraint cone {
       x+z>=z;
     }
     constraint ctwo {
       y>z;
     }
     #putting everything together
     specification {
       cone && ctwo;
     }
}
```

CSL provides constructs to define tunable parameters and to express relationships between those parameters. Dependencies between different parameters can be easily specified using mathematical expressions. CSL supports a fairly comprehensive list of mathematical, logical and relational operators. Hints to the underlying search algorithm in the form of initial points to start the search, default values for parameters, simple constraints on parameters such as MPI message-sizes, number of OpenMP threads, etc. can be easily expressed using CSL. Information from performance models can be specified in the form of constraints to guide the search. Furthermore, parameters can also be grouped into different categories to allow application of similar tuning strategies. This is particularly helpful when there are multiple code-sections that benefit from the same optimization. We provide a simple parameter specification example in Table I. In this example, the search space consists of three parameters — `x`, `y` and `z`. The relationships between these parameters are expressed using two constraint definitions — `cone` and `ctwo`. Full CSL grammar is provided in the Appendix.

III. PARAMETER TUNING ALGORITHM

A key to any auto-tuning system is how it goes about selecting the specific combinations of parameters to try. While a simple parameter space might be exhaustively searched, most applications contain too many combinations to try them all. Instead, an auto tuning system must rely on efficient search algorithms to evaluate only a sub-set of the possible

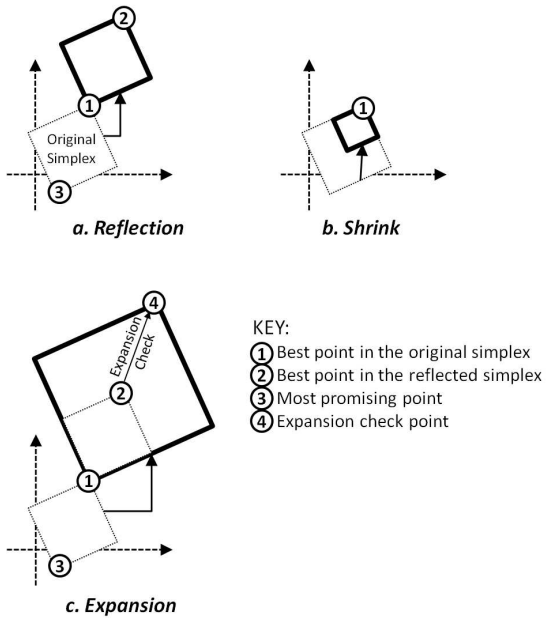

Fig. 1. Original 4 point simplex in a 2-dimensional space, along with the simplex transformation steps.

configurations while trying to find an optimal one (or least as nearly optimal as practical).

The algorithm that we use is based on the Parallel Rank Order (PRO) algorithm proposed by Tabatabaee et al [22]. For a function of N variables, PRO maintains a set of K (where K is at least $N+1$ and is usually set to the number of cores the harmonized application is run on) points forming the vertices of a simplex in an N-dimensional space. Each simplex transformation step of the algorithm generates up to $(K-1)$ new vertices by reflecting, expanding, or shrinking the simplex around the best vertex. After each transformation step, the objective function value f associated with each of the newly generated points are calculated in parallel. The reflection step is considered *successful* if at least one of the $(K-1)$ new points has a better f than the best point in the simplex. If the reflection step is not successful, the simplex is shrunk around the best point. A successful reflection step is followed by expansion check step. If the expansion check step is successful, the expanded simplex is accepted. Otherwise, the reflected simplex is accepted and the search moves on to the next iteration. The search stops if the simplex converges to a point in the search space (or after a pre-defined number of search steps). A graphical illustration for reflection, expansion and shrink steps are shown in Figure 1 for a 2-dimensional search space and a 4-point simplex.

Note that before computing all expansion points, we check the outcome of the expansion for only the most promising case first. The most promising point (shown in Figure 1-(c)) is the point in the original simplex whose reflection around the best point returns a better function value. This seems to be counter-intuitive at first glance, since we are not taking full advantage of the parallelism. However, in our experiments, we realized there are some expansion points with very poor performance that can slow down the algorithm (for example, setting a tile size to 0). Therefore, to avoid these time consuming instances and to ensure good transient behavior of the search algorithm, we calculate the expansion point performance for the most promising case first and only if it is successful, perform a full expansion of the simplex.

One of the unique features that distinguishes PRO from other algorithms used in auto-tuning systems is that 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. For online tuning of SPMD-based parallel applications, different nodes participating in the tuning process can evaluate different versions of given loop-nests. We depict this graphically in Figure 2-(a) (more description of the figure appears in the next section)[1].

Parameter tuning is a constrained optimization problem. Therefore after each simplex transformation step, we have to make sure that the points returned by the search algorithm for evaluation are admissible, i.e. they satisfy the constraints. We consider two types of parameter constraints: internal discontinuity constraints and boundary constraints. Internal discontinuity constraints arise because some tuning parameters can only have discrete admissible values (e.g. integer parameters). For these constraint violations, the computed parameters are rounded to an admissible discrete value.

To handle boundary constraints, we use a penalization method. We add a penalty factor to the performance metric associated with the points that violate the constraints. The idea is to discourage the simplex from moving towards illegal regions of the search space. This approach has been used previously in the context of constrained optimization using genetic algorithms [11]. In all of the experimental results presented in this paper we use this method because it is simple and light-weight. This feature of Active Harmony is new and has not been considered in our previous work.

IV. System Design

In this section, we describe our online auto-tuning approach for parameters that require new code and present our system design. Examples of parameters that require new code include loop unrolling factors, loop fusion and split parameters and data-copy parameters. With the addition of this capability, Active Harmony supports the development of self-tuning applications that include runtime code-transformation and replacement.

To make runtime auto-tuning practical, the key issue that needs to be addressed is the efficient *runtime* management of the process of generating, compiling, and maintaining a set of alternative implementations and searching among them. A given loop-nest generally requires more than one flavor of transformation strategy. As the number of transformations

[1]There are, of course, some parameters that are global (such as data decomposition). These are searched sequentially.

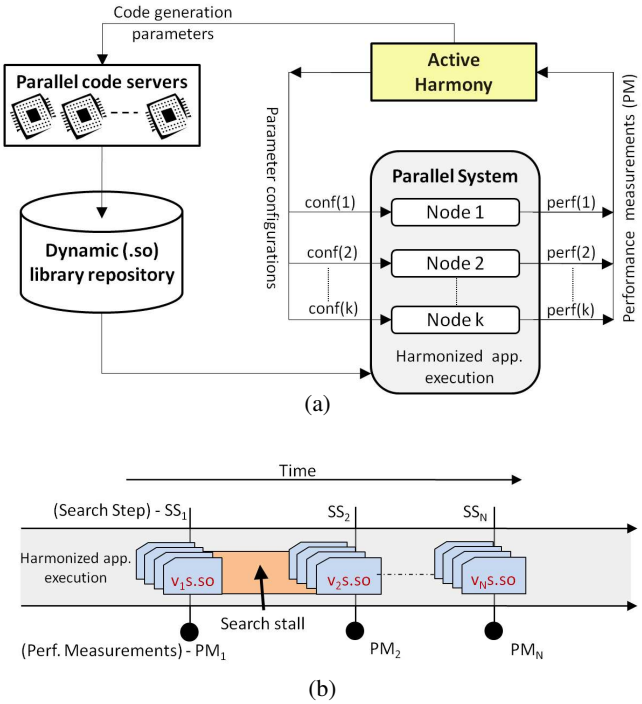

Fig. 2. Fig.2-(a) shows the overall online tuning workflow. Fig.2-(b) shows application level view of the auto-tuning workflow

increases, the number of alternative code-variants grows exponentially. A brute-force approach of generating all possible combinations is, thus, not practical. Instead, our approach generates code variants on-demand. This selective approach requires an efficient code generator that can produce correct code during runtime.

Developing an efficient code transformation framework that ensures correctness is a large area of research in and of itself. Recently, a variety of compiler-based code transformation frameworks [7], [10], [4], [17] have been developed to give programmers and compiler experts greater control over what optimization strategies to use for complex codes. These frameworks are designed to facilitate manual exploration of the large optimization space of possible compiler transformations and their parameter values. Each of these frameworks has its strengths and weaknesses and as such, it is desirable to design a system that can easily select among the available tools based on our code transformation requirements.

Active Harmony relies on standalone code-generation utility (or code-servers) for on-demand code generation. Here we describe the two most important features of this utility. First, the design of code-servers allows the users to easily select and switch between available code transformation tools. We separate the search-based navigation of the code-transformation parameter space and the code-generation process, which allows us to easily switch between different underlying code-generation tools (e.g. if we are tuning CUDA code, we can switch to a code-transformation framework that supports GPUs via CUDA or OpenCL). Second, our code-generation utility can take advantage of idle (possibly remote) machines for distributed code-generation and compilation. Users provide a set of available machines at the start of the tuning session. These machines do the actual code-generation work. Once all code-variants are generated, the compiled code-variants are transported to the scratch filesystem of the parallel machine, where the application being tuned is executing. After the code-generation is complete, our code-generation utility notifies the Active Harmony server about the status.

Figure 2-(a) shows a schematic diagram of the workflow within our online tuning system. Figure 2-(b) shows the application-level view of the tuning process. At each search step, the Active Harmony server issues a request to the code-servers to generate code variants with a given set of parameters for loop transformations. The code-variants that are generated are compiled into a shared library (denoted as `v_N.so` in the figure 2-(b)). Once the code-generation is complete, the application receives a code-ready message from the Active Harmony server. The nodes allocated to the parallel application then load the new code using the `dlopen-dlsym` mechanism. The new code is executed and the measured performance values (denoted as `PM_N` in the figure 2-(b)) are consumed by the Active Harmony server to make simplex transformation decisions. The timing of actual loading of new code is determined by hooks (inserted using the Active Harmony API) in the application. For example, in most programs, we load new code only on timestep boundaries.

Preparing an application for auto-tuning starts with outlining the compute-intensive code-sections to separate functions. We then insert appropriate calls to the outlined functions using function pointers. These function pointers are updated when new codes become available. Currently, the code-sections are outlined manually. In the future, we intend to automate this process using the ROSE compiler framework [12]. Each node running the application keeps track of the best code-variant it has seen thus far in the tuning process. If the code-server fails to deliver new versions on time, the nodes continue their execution with the best version that they have discovered up to that point in the tuning process. The period where no new code is available is referred as `search_stall` (see figure 2-(b)). The non-blocking relationship between application execution and dynamic code-generation is important in minimizing the online tuning overhead. The application does not have to wait until the new code becomes available. Furthermore, this asynchronous relationship enables our auto-tuner to exercise control over what code-generation utility to use, how many parallel code-servers to run and how many code-variants to generate in any given search iteration. The policy decisions about what code-variants to generate and evaluate at each iteration is made completely by the centralized tuning server.

V. Experiments

In this section, we present an experimental evaluation of our framework. First, we conduct a study using as a test application, a Poisson's equation solver program, to determine the least number of parallel code-servers needed to ensure

that the `search_stall` phase does not dominate the tuning workflow. Second, we use two parallel applications to demonstrate the effectiveness of our system on three different computing platforms. We compare the performance of harmonized applications with that of original applications compiled with the vendor-suggested highest level of optimization flags turned on. Moreover, once the harmonized application is done with its execution, we take the best code-variants returned by the Active Harmony server and run the application using those code-variants. We call these runs `post-harmony` runs.

Active Harmony utilizes the first search iteration to generate uniformly distributed random configurations from the search space. These configurations are evaluated in parallel. We call this iteration exploratory iteration[2]. The best among these configurations then serves as the starting point for the initial simplex construction. For all our experiments, unroll factors and tile sizes are constrained by the storage capacity of their associated memory hierarchy levels.

To control for performance variability[3], we use the multiple sampling method. Each configuration is evaluated twice (i.e. the performance of two consecutive timesteps is recorded) and the minimum of the two samples is sent to the Active Harmony server. We showed in our previous work [22] that even in the presence of 5% variability due to background noise, taking the minimum of two samples is enough to ensure the convergence of the search algorithm.

A. Platforms

The experiments were performed on three platforms. The first platform is a 64-node Linux cluster (henceforth referred to as umd-cluster). Each node is equipped with dual Intel Xeon 2.66 GHz (SSE2) cores (with Hyper-Threading enabled). Nodes are connected via a Myrinet interconnect. The second platform (at NERSC [6]) is named Carver, which is an IBM iDataPlex system with 400 compute nodes. Each node contains two quad-core Intel Nehalem 2.67 GHz cores (3,200 cores total in the machine). Nodes are connected via a 4X QDR Infiniband interconnect. These architectures are different from each other not only in terms of the core architectures — Carver's cores are several generations newer — but also in terms of the interconnect used to connect the nodes. Finally, the third platform, which is named Hopper[4], is a Cray XT5 machine at NERSC. Each node consists of two 2.4 GHz AMD Opteron Shanghai quad-core cores (5,312 cores total in the machine). Nodes are connected via a Seastar2 interconnect.

[2]Note that this randomized method can sometimes select points in the search space that have poor performance. We pay this penalty for one iteration at the beginning of the search to gather some knowledge about the search space. The cost of this step is usually amortized within the first few PRO steps.

[3]Besides the tunable parameters, there are many other factors affecting a program's performance. Therefore, even for a fixed set of tunable parameters, the application performance varies in time. Network contention, operating system jitter, and memory architecture complexity are common sources of performance variability.

[4]Hopper has since been upgraded to a Cray XE6 with 153,408 cores.

B. Code-generation utility

For all the experimental results presented in this paper, we use CHiLL [7], a polyhedra-based compiler framework, to generate code-variants. As we discussed in section IV, we can use any available code-generation utility within our system. We chose CHiLL because it provides a convenient 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 (see tables III and IV). In CHiLL nomenclature, these scripts are called "recipes". Besides making it easy to interface with the code-generation utility, these code transformation recipes offer an additional advantage. Unlike traditional compiler optimizations which must be coded into the compiler, these recipes can be evolved and reused over time. A recipe library, created by compiler experts and developers based on their experience working with real codes, can then be consulted by auto-tuners to tune arbitrary loop-nests.

For the experiments on the umd-cluster, the code-generation and compilation is delegated to idle local machines. For the experiments on Carver, the code-generation is out-sourced to an eight-core 64-bit x86 machine at UMD (i.e. code is generated just in time and shipped across the continental United States). This was done because the Carver scheduler does not permit synchronized (co-scheduled) jobs yet, which meant that we could not launch a code-generation job simultaneously with the application job. For the experiments on Hopper, the code-generation and compilation takes place on the login nodes.

C. Calculating the "net" speedup

Our runtime tuning strategy uses extra cores to generate and compile new code. Ideally, a fair comparison would be between the execution time of the harmonized application to that of the original application run on $N_h + C$ cores, where N_h is the number of cores the harmonized application is run on and C is the number of cores used for code-generation. However, this is not always possible due to application's data distribution semantics (for example, the application may require the number of cores to be a power of 2). Instead, to account for these extra cores, we calculate a new metric — *net speedup*. We define charge factor (equation 1) as the ratio of the number of cores used to run the application and the total number of cores used for both code-generation and harmonized application execution.

$$c.f. = \frac{N_h}{N_h + C} \tag{1}$$

We then multiply the speedup of harmonized applications over the original application by this charge factor to derive the net speedup.

D. Code-server sensitivity

With the experimental results presented in this section, we attempt to answer the following question — how many parallel code-servers are needed to ensure that the auto-tuner does not have to wait for too long before the new code is

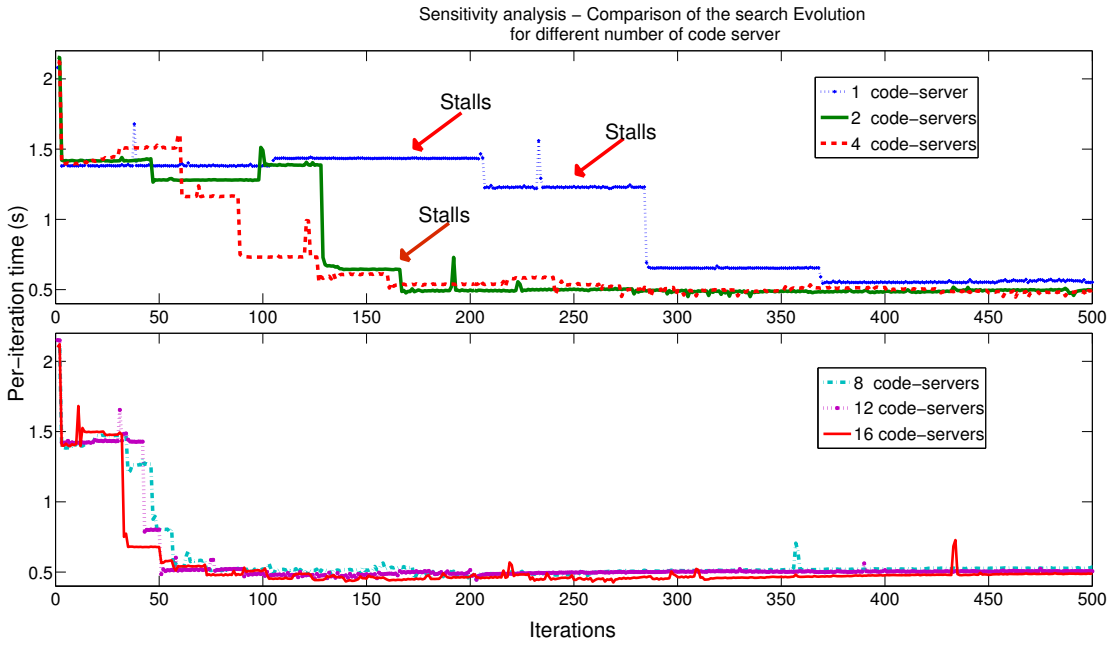

Fig. 3. Sensitivity results demonstrating how the change in the number of code-servers affects the search evolution

ready? A related question is — How often is the system in the `search_stall` phase? These questions are important because if the `search_stall` phase is long, the application can possibly continue with mediocre parameter configurations for extended periods of time.

The experiments were conducted on the umd-cluster using the PES application (described in section V-E1). We controlled the input problem size (1024^3) and the number of cores running the application (128). All 128 cores participate in the tuning process, which means at each search step, code-servers have to generate and compile up to 128 code-variants. This is a typical number of code-variants required per search iteration in all the experiments reported in this paper. We vary the number of code-servers running in parallel and record the average number of time-steps that the application had to continue with the old code. We call this metric "stalled" iterations.

Figure 3 shows how per-iteration performance measurements change over time for auto-tuning conducted with alternate number of code-servers. Long stretches of consecutive application timesteps with no performance improvement (marked by arrows) in experiments conducted with 1, 2 and 4 code-servers indicate that the application continued its execution with poor configurations for several timesteps. The same is not true for tuning conducted with 8, 12 and 16 code-servers.

Table II summarizes the results for this experiment. We direct the readers to columns 3 and 5. As the number of code-servers is increased from 1 to 4, the average number of stalled iterations goes down significantly. This is to be expected. What is surprising is that the addition of extra code-servers from 8 to 16 does not significantly change the application speedup or the number of stalled iterations. The reason for this is that as the search algorithm evolves and starts converging to a point in the search space, the load on code-servers goes down (i.e., more points in the simplex become identical). The data in column 4 of table II shows the number of unique code-variants evaluated in different experiments. We can see that as the average number of search iterations goes from 6 for runs using 1 code-server to 17 for runs using 2 code-servers, the average number of unique code-variants goes up by only 208.

Our end goal with this experiment was to set a minimum number of code-servers required to ensure a short `search_stall` phase for the rest of the experiments. Having said that, we acknowledge that there are other factors that can play important roles in setting this minimum. In one of our preliminary experiments, we discovered that the size of the search space can also dictate this minimum. When we ran the experiments with small tuning space, the exploratory iteration (initial random search of points) was able to find good parameter configurations. In this case, the number of stalled iterations did not matter because applications were already executing with good configurations. Moreover, the number of minimum code-servers can (and most probably will) change if we switch between different code-generation tools. Currently, we are looking into more robust ways to account for these issues and to derive the value for the minimum required parallel code-servers. For the rest of the auto-tuning experiments we describe in this paper, we used 8 code-servers.

E. Subject applications, tuning strategies and results

1) Poisson's Equation Solver (PES): Poisson's equation is a partial differential equation that is used to characterize many processes in electrostatics, engineering, fluid dynamics, and theoretical physics. To solve for Poisson's equation on a three-

TABLE II
SENSITIVITY EXPERIMENT RESULTS

# of code servers	Avg. # of search iters	Avg. # of stalled iters	Avg. # of code evaluated	Avg. speedup
1	6*	46	502	0.75
2	17*	13	710	0.97
4	27	7.18	928	1.04
8	23	4.48	818	1.23
12	22	4.06	833	1.21
16	26	3.59	931	1.24

* - search algorithm did not converge

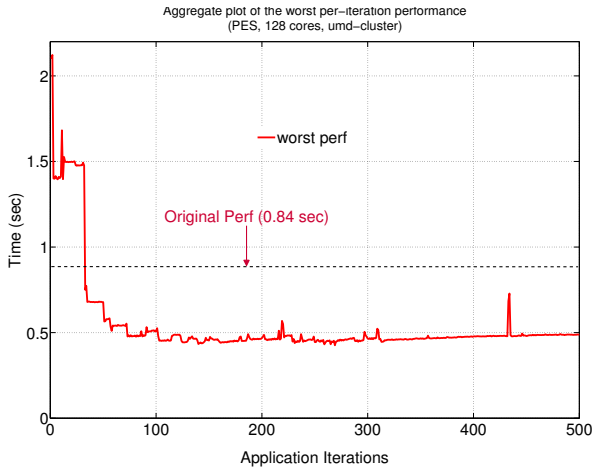

Fig. 4. A plot for aggregate worst timing at each iteration.

dimensional grid, we use a modified version of the parallel implementation provided in the KeLP-1.4 distribution [1]. The application is written in C++ and Fortran. The implementation uses the redblack successive over relaxation method to solve the equation. The core of the computational time is spent on the relaxation function, which uses a 7-point stencil operation, and the error calculation function, which calculates the sum of squares of the residual over the 3D grid. These two code-sections are tuned simultaneously using the Active Harmony framework. The code-sections are outlined in table III.

The original implementation of the relaxation operation uses the first half of one iteration to update "red" array points and the second half of the iteration to update the "black" points. In the adapted version of the application, we use the fused version of the relaxation operation. The fused version orders the loop iteration so that black points in each column are updated immediately after the red points in the next column and vice versa [20]. This fused version (see Table III) serves as the baseline for comparing the net speedup of harmonized PES.

Our tuning strategy for this application combines symbolic parameter tuning[5] and tuning with dynamic code-generation. For the relaxation function, we use symbolic tuning. We tile the two outermost loops and use Active Harmony to determine the dimension of the tiles. The error function is optimized

[5]Symbolic tuning refers to tuning for parameters that are symbolic, i.e. no new code is necessary to move between parameter values.

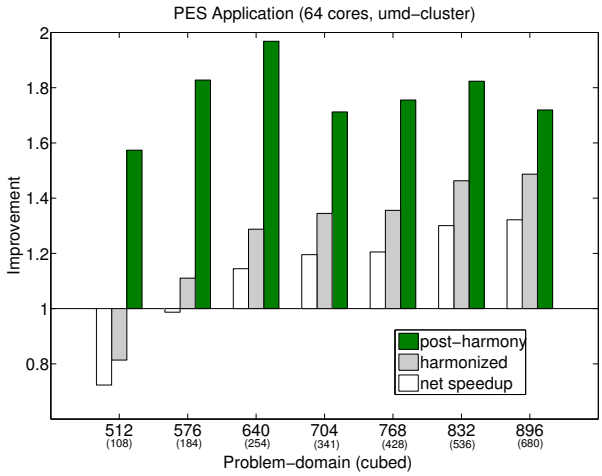

Fig. 5. Performance improvement of harmonized PES, net speedup, and post-harmony run of the solver (64 core run on umd-cluster)

using the dynamic code-generation method. For this function, we tile all three loops and the innermost loop is unrolled. The search space is, thus, six-dimensional (two tunable parameters for the relaxation function and four for the error function). All cores allocated to the application participate in the tuning process. Thus, a 128-core run of this application evaluates up to 128 tiling configurations simultaneously for the relaxation function and up to 128 loop-variants simultaneously for the error function in a single search step. The optimization strategy (expressed in terms of the CHiLL recipe) along with the constraints for unbound tunable parameters is provided in table III. For a 512^3 problem size run on 64-cores, the search space has approximately 3.11×10^7 possible configurations.

We performed two sets of auto-tuning experiments — one using 64 cores and one using 128 cores. Both sets of experiments were done on the umd-cluster. For each core count, we select multiple input domain sizes. Figure 4 shows how Active Harmony steers per-iteration performance of the harmonized PES. This experiment uses a grid size of 1024^3. The figure plots the timing of the worst performing configuration for each application iteration. The running time of an SPMD-based application is bounded, at each timestep, by the slowest configuration. Per-iteration time for the original application is indicated by the horizontal line in the figure. The figure shows that Active Harmony suggested configurations outperform the original application's per-iteration timing within the first few tens of iterations.

Figures 5 and 6 plot the net and harmonized speedups achieved within one full execution of the harmonized PES. The original application execution times (in seconds) are shown in parentheses below the label for x-axis. The application was run on 64 and 128 cores on the umd-cluster for varying input data sizes. As expected, as the size of the problem domain increases, the performance of the harmonized application increases as well. This is intuitive because with the increase in the problem size, the Active Harmony server gets more time to explore the search space before the application

TABLE III
PES AUTO-TUNING

Kernel	*Original Code*	*Transformation Strategy/Recipe*	*Search Constraints and space description*

relaxation	`do kk=wl2-1,wh2`
	`do k=kk+1,kk,-1`
	`if ((k.le.wh2).and.`
	`(k.ge.wl2)) then`
	`do j=wl1,wh1`
	`do i=wl0+mod(kk+j+1,2),wh0,2`
	`u(i,j,k) = c * (u(i-1,j,k)`
	`+ u(i+1,j,k) + u(i,j-1,k)`
	`+ u(i,j+1,k) + u(i,j,k-1)`
	`+ u(i,j,k+1)- c2*b(i,j,k))`
		`Manual tiling:`
		`i and j loops`
		`(TI1, TJ1)`
			$TI1 \times TJ1 \leq \frac{1}{2}\left(\frac{cache\_size}{2}\right)$
			$TI1 \geq TJ1$
			$TI1 \in [0, 4, \ldots, prob\_size]$
			$TJ1 \in [0, 4, \ldots, prob\_size]$

error	`do k = 1, N`
	`do j = 1, N`
	`do i = 1, N`
	`du = c*(u(i-1,j,k) + u(i+1,j,k)`
	`+ u(i,j-1,k) + u(i,j+1,k)`
	`+ u(i,j,k-1) + u(i,j,k+1)`
	`- c2*u(i,j,k))`
	`r = b(i,j,k) - du`
	`err = err + r*r`
		`original()*`
		`tile(0, 3, TI2)**`
		`tile(0, 3, TJ2)`
		`tile(0, 3, TK2)`
		`unroll(0, 6, UI2)***`
			$TI2 \in [0, 4, \ldots, prob\_size]$
			$TJ2 \in [0, 4, \ldots, prob\_size]$
			$TK2 \in [0, 4, \ldots, prob\_size]$
			$TI2 \geq TJ2$
			$TJ2 \geq TK2$
			$UI2 \in [1, 2, \ldots, register\_size]$
			
			$Search\ space\ dimension: 6$
			$Parameters: [TI1, TJ1, TI2, TJ2, TK2, UI2]$
			
			$Sample\ search\ space\ size: 3.11 \times 10^7$
			$possible\ configurations\ for$
			$64 - core\ run\ with\ 512^3 domain - size$

* - keep original loop order. ** - tile(statement #, loop level to tile, tiling factor) *** - unroll(statement #, loop level to unroll, unroll factor)

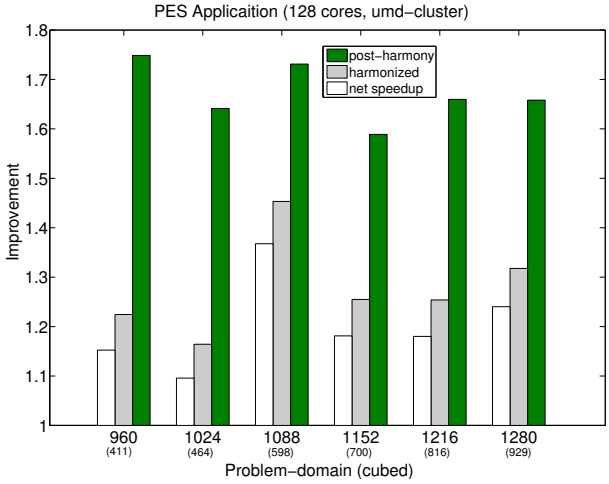

Fig. 6. Performance improvement of harmonized PES, net speedup, and post-harmony run of the solver (128 core run on umd-cluster)

completes its execution. For the 512^3 problem size (see figure 5), the program runtime is too short (108 seconds) and the harmonized application runs 28% slower than the original untuned version. The 28% slowdown does incorporate the charge factor for 8 extra cores used for code-generation. The harmonized application is unable to overcome the penalty of using some poor configurations early in the short run of the program (132 seconds elapsed time).

On average, for both core counts and different problem domains, harmonized PES, in terms of the net speedup, performs 1.16 times faster than the original application. Thus, even after allowing for the code-servers, a single execution with no prior runs is, on average, 16% faster than the original application. The best net speedup for the harmonized application is 1.37. Post-harmony runs, which use Active Harmony suggested parameter configurations and code-variants, on average, perform 1.72 times faster than the original application. This indicates the performance gain if the program was run a second time on the same machine with similar inputs.

2) Parallel Multiblock Lattice Boltzmann (PMLB): The Lattice Boltzmann Method (LBM) is a widely used method in solving fluid dynamic systems. In contrast to the conventional methods in fluid dynamics, which are based on the discretization of macroscopic differential equations, the LBM has the ability to deal efficiently with complex geometrics and topologies [25]. For our experiments, we use the parallel multiblock implementation (extended to 3D problems) of the LBM developed by Yu et al [26]. The test case lattice model for our experiments is `D3Q19` (19 velocities in 3D) with the collision and streaming operations. The application is written in C.

The PMLB code is divided into six main operations: initialization, collision, communication, streaming, physical and finalization. Collision, communication, streaming and physical operations are executed within a loop. Initialization and finalization operations are performed once. We focus our attention on the streaming operation, which accounts for more than 75% of the execution time. The streaming operation moves particles in motion to new locations along with their respective 19 velocities. This operation requires a significant number of memory copy operations.

The streaming operation consists of five separate triply-

TABLE IV
PMLB AUTO-TUNING

Kernel	*Original Code*	*Transformation Strategy/Recipe*	*Search Constraints and space description*

streaming 1	`for (i=1; i<=imax;i++)`
	`  for(j=1; j<=jmax; j++)`
	`  for(k=1; k<=kmax; k++)`
	`  {`
	`  c1 = i*(ny_local);`
	`  c2 = c1+(ny_local);`
	`  c3 = (c1+j)*(nz_local);`
	`  c4 = c3+(nz_local);`
	`  c5 = (c2+j)*(nz_local);`
	`  c6 = c5+(nz_local);`
	`  c7 = (c3+k)*en;`
	
	`  fi[ 6+c7]=fi[6+(k+1+c3)*en];`
	`  fi[ 4+c7]=fi[4+(k+c4)*en];`
	`  fi[18+c5]=fi[18+(k+1+c4)*en];`
	`  fi[ 2+c7]=fi[ 2+(k+c5)*en];`
	`  fi[14+c7]=fi[14+(k+1+ c5)*en];`
	`  fi[10+c7]=fi[10+(k+c6)*en];`
	`  }`
		`original()*`
		`tile(0, 1, TI)**`
		`tile(0, 3, TJ)`
		`unroll(0, 5, UK)***`
		`known(imax>1)****`
		`known(jmax>1)`
		`known(kmax>1)`
			$TI \in [0, 4, \ldots, prob\_size]$
			$TJ \in [0, 4, \ldots, prob\_size]$
			$UK \in [1, 2, 3, 4]$
			$TI \geq TJ$
			
			$Search\ space\ dimension: 6$
			$2\ sets\ of\ [TI, TJ, UK]:$
			$one\ for\ fused\ kernels\ and$
			$one\ for\ non-fused\ kernels$
			
			$Sample\ search\ space\ size: 8.92 \times 10^6$
			$possible\ configurations\ for$
			$128-core, 512^3\ problem\ size$

* - keep original loop order. ** - tile(statement #, loop level to tile, tiling factor) *** - unroll(statement #, loop level to unroll, unroll factor)
**** - known predicate

nested kernels, which are tuned simultaneously. Our optimization strategy utilizes loop-fusion, loop-tiling and loop-unrolling. The tuning is done in two phases. The first few iterations of the LBM method are used to identify the best fusion configuration for the five triply nested loops within the streaming operation. For this stage, we use the exhaustive search. Once we identify the best performing fusion configuration, the tuning moves to the second stage, which involves tiling the outermost two loops and unrolling the innermost loop[6]. The second stage uses the parallel rank ordering algorithm to determine two sets of tiling and unrolling factors — one for the fused loop-nests and another for the remaining loop-nests. The search parameter space for all PMLB experiments is, thus, six-dimensional. The optimization strategy (expressed in terms of the CHiLL recipe) along with the constraints for unbound tunable parameters is provided in table IV. Out of the five deeply nested kernels in the streaming operation, we show only one kernel in the table. Other kernels are similar in structure. For a 512^3 problem size run on 128-cores, the search space has approximately 8.92×10^6 possible configurations.

PMLB tuning experiments were done on the umd-cluster, Carver and Hopper. Figures 7 and 8 plot speedup results for harmonized and post-harmony PMLB runs using 64 and 128 cores on umd-cluster. We use multiple input datasets for different core counts. Again, we see that the increase in the size of the problem domain leads to a better performance for the harmonized PMLB. On average, harmonized PMLB performs 1.14 times faster than the original application, while post-harmony runs perform 1.38 times faster than the original application.

For the experiments on Carver, we use two different core counts — 256 and 512. We were limited in terms of the

[6]Simple code modifications were required to remove scalar dependencies between different levels of loop-nests to ensure legality of code transformations.

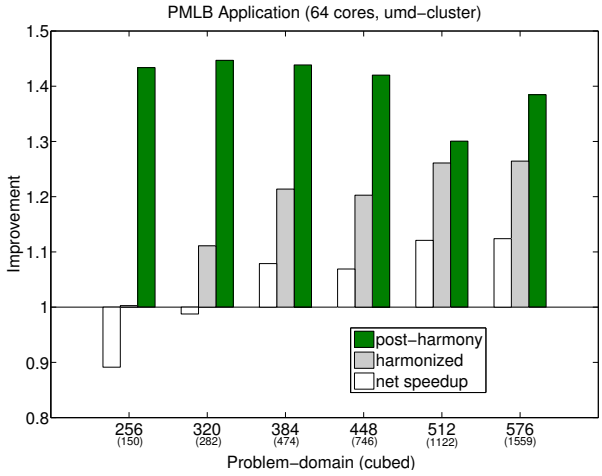

Fig. 7. Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (64 core run on umd-cluster)

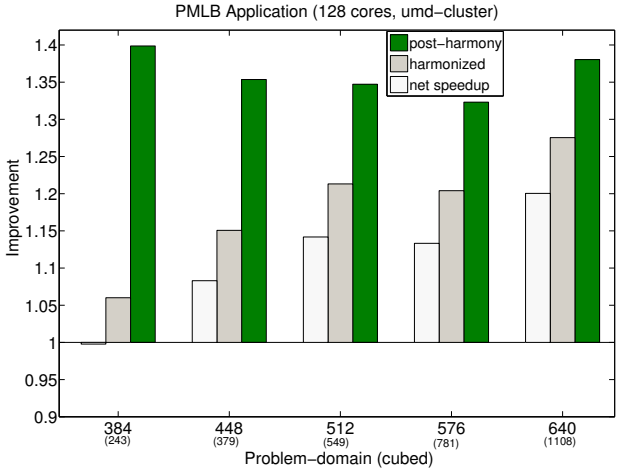

Fig. 8. Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (128 core run on umd-cluster)

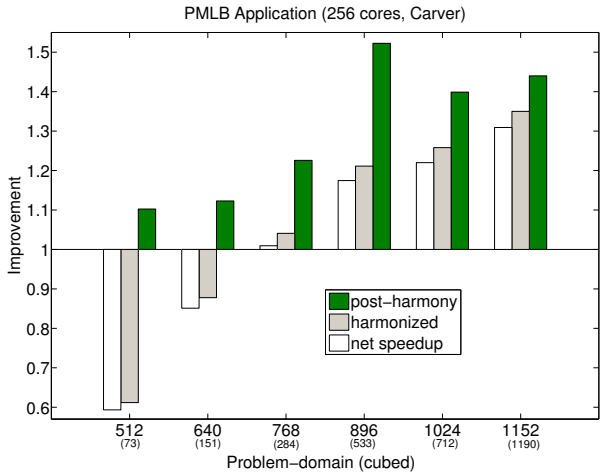

Fig. 9. Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (256 core run on Carver)

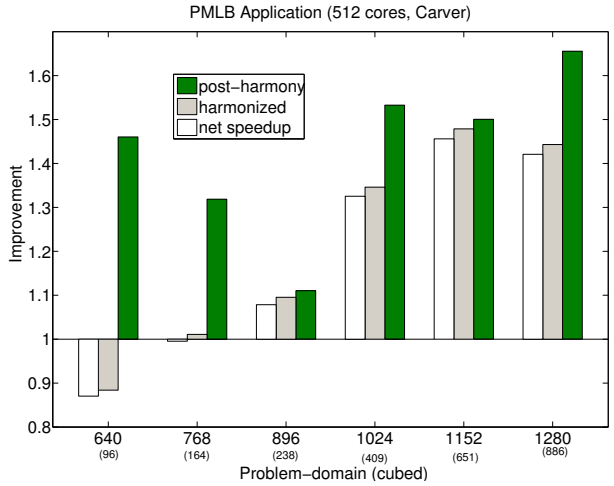

Fig. 10. Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (512 core run on Carver)

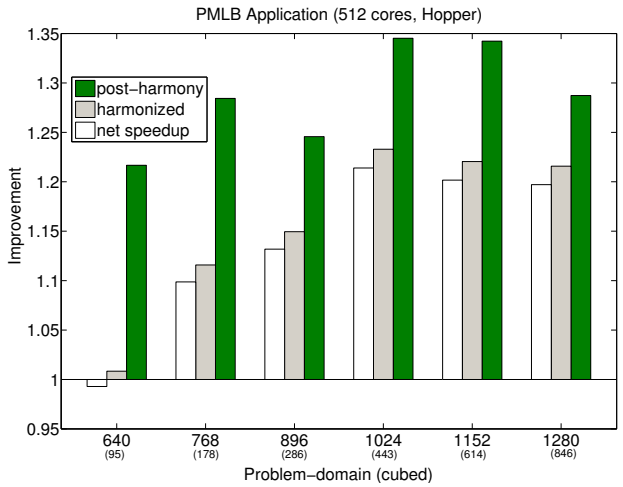

Fig. 11. Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (512 core run on Hopper)

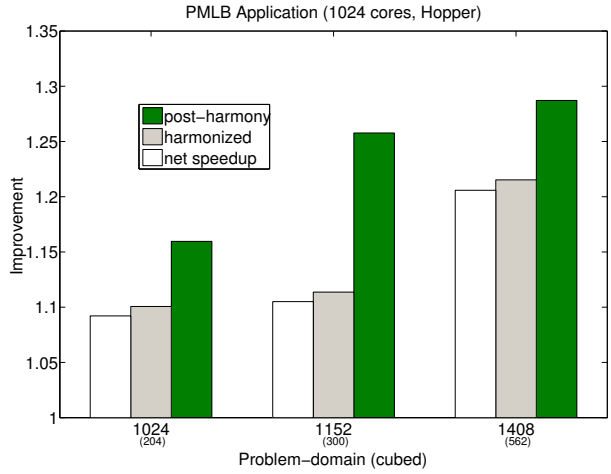

Fig. 12. Performance improvement of harmonized PMLB, net speedup, and post-harmony run of PMLB (512 core run on Hopper)

number of core counts because 512 is the maximum core count a user can reserve on Carver. Figures 9 and 10 plot speedup results for harmonized and post-harmony runs using 256 and 512 cores of Carver. In terms of the net speedup, on average, harmonized PMLB performs 1.11 times faster than the original application. The best net speedup for a harmonized run is 1.46, i.e. even after factoring in the extra cores for code-generation, a single execution of harmonized PMLB is up to 46% faster than the original application. Post-harmony runs perform, on average, 1.37 times faster than the original application.

Experiments on Hopper were done using 512 and 1024 cores. Figures 11 and 12 plot speedup results for harmonized and post-harmony runs using 512 and 1024 cores of Hopper. In terms of the net speedup, on average, harmonized PMLB performs 1.14 times faster than the original application. The best net speedup for a harmonized run is 1.21. Post-harmony runs perform, on average, 1.28 times faster than the original application.

F. Cross-platform comparison

In the preceding section, we showed how we applied Active Harmony to auto-tune the execution of two parallel applications on three different platforms. A logical question is how do the parameters that Active Harmony selected for different platforms relate to each other. To answer this question, we conducted a controlled study using 64-cores on all three systems. We selected three problem sizes for the PMLB application. The selection of the problem sizes was based on whether Active Harmony's search converges to a solution or not within a single execution of the harmonized PMLB. This was done to ensure that Active Harmony gets a fair chance to select good configurations on all the systems. We then use Active Harmony suggested parameter configurations for a given problem size on one system to conduct post-harmony runs for the same problem size on other systems.

The results are summarized in table V. Post-harmony runs conducted on the umd-cluster using the configurations sug-

TABLE V
RESULTS FOR CROSS-PLATFORM EXPERIMENTS

Problem-size	speedups for post-harmony on umd-cluster			speedups for post-harmony on Carver			speedups for post-harmony on Hopper		
	w/ umd confs	w/ Carver confs	w/ Hopper confs	w/ Carver confs	w/ umd confs	w/ Hopper confs	w/ Hopper confs	w/ Carver confs	w/ umd confs
448^3	1.42	1.13	1.00	1.51	1.38	1.34	1.28	1.30	1.27
512^3	1.30	1.26	0.95	1.34	1.31	1.33	1.34	1.31	1.28
576^3	1.38	1.16	1.02	1.42	1.39	1.27	1.31	1.35	1.30

gested for Carver do perform better than the original version of the application. However, the speedup difference between post-harmony runs conducted on the umd-cluster with umd-cluster configurations and Hopper configurations is rather significant. Upon closer look at the parameter values, we observed that for umd-cluster and Carver, Active Harmony only fuses the first and the third kernels in the streaming operation. While on Hopper, the first, the third and the fifth kernels are fused together. Furthermore, on Hopper, the second and the fourth kernels are fused as well. We suspect that the poor performance can be attributed to the properties and size of the instruction cache on umd-cluster. It is also possible that excessive application of loop-fusion causes more register spills on umd-cluster than for the other two platforms, thereby degrading the performance of the PMLB [18]. This result argues tuning not only for specific architecture but also for specific processor implementation. In the future, we plan to look at the question of how the configurations found for different input datasets for the same harmonized application relate to each other.

It is also interesting to see that the post-harmony runs conducted on Carver using Hopper configurations provide similar speedups when compared to the post-harmony runs conducted on Carver using Carver configurations. The same is true for post-harmony runs conducted on Hopper. We attribute this to the processor architecture similarity of the two systems.

In the work presented in this paper, we focused exclusively on optimizing computation at per-core level. Our future work will look into communication auto-tuning. We believe that the difference in the interconnect technology between the Carver and Hopper systems will show the benefits of tuning for specific interconnect technology.

VI. RELATED WORK

Several techniques have been proposed to dynamically adapt a program to a given input and the runtime environment. CPO (Continuous Program Optimization) [5] uses monitoring agents that collect information from all layers of an application execution stack. This information is used to model the application behavior and predict the relative benefit of using large pages for different application data structures. Autopilot [19] is an online tuning framework for parallel applications. Based on application request patterns and observed system performance, Autopilot's real-time adaptive control mechanism automatically chooses and configures resource management algorithms.

AppLes [3] and Odyssey [16] both are application-centric tools and emphasize application level resource awareness. Applications adapt by (re)allocating the resources based upon a customized scheduling to maximize their performance. Rather than leaving the adaptation decisions to applications, our approach uses a centralized server to control such decisions.

Dynamic code-generation and runtime loading of different versions of code-sections is a technique that has been used both in the context of dynamic software updating [15] and auto-tuning [24]. ADAPT [24] is a compiler-supported infrastructure for high-level adaptive program optimization. It allows developers to leverage existing compilers and optimization tools by describing a runtime heuristic for applying techniques in a domain specific language, ADAPT Language. ADAPT supports remote dynamic compilation, parameterization and run-time sampling, allowing developers the flexibility in heuristic development. Our work is distinct from ADAPT in two ways. First, we target SPMD-based parallel applications and use the parallelism to our advantage by evaluating multiple parameter configurations within one search iteration. Second, our system is designed to tune multiple code-sections simultaneously.

MATE [14] performs dynamic tuning in three basic and continuous phases: monitoring, performance analysis and application modification. This environment dynamically and automatically instruments a running application to gather information about the application's behavior. The analysis phase receives events, searches for bottlenecks, detects their causes and gives solutions on how to overcome them. Finally, the application is dynamically tuned by applying a solution described by simple performance models. Our work is distinct from MATE in that we use effective and light-weight algorithms to search for optimal parameters.

VII. FUTURE WORK

There are several natural extensions of our parallel code generation and tuning framework.

A. Exploiting spatial locality of PRO

In the future, we plan to exploit PRO's spatial locality[7] to generate code-variants speculatively in anticipation of the search algorithm needing them. This speculative generation of code-variants uses knowledge of what the search algorithm will likely do in the next search step. With the code-server

[7]In the realm of Parallel Rank Ordering algorithm and related direct search methods, spatial locality refers to the likelihood of evaluating a given point in the search space is higher if a nearby point was just evaluated.

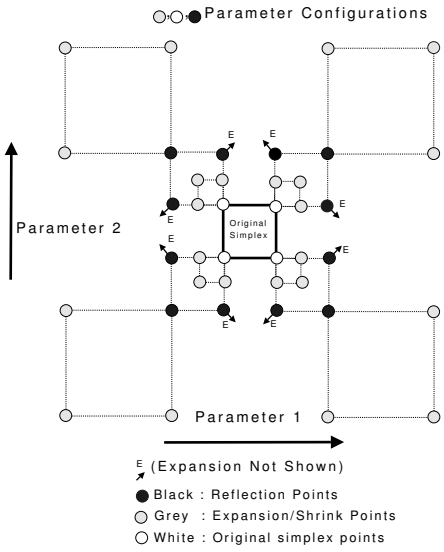

Fig. 13. Exploiting PRO's Spatial Locality

continuously generating the anticipated code-variants, we can shorten the `search_stall` phase (see figure 2-(b)).

PRO's spatial locality derives from the fact that a reflection step in PRO is followed by either a shrink step or an expansion step. Therefore, irrespective of what the best point in the simplex will turn out to be, we can easily enumerate an exhaustive list of possible points for the next search step. The dedicated code-servers can utilize this list to continuously generate code in the background. We provide a graphical illustration of this strategy in Figure 13. The figure shows a two-dimensional search space. PRO uses a 4-point simplex. The first search step evaluates the original simplex points (white points). The black points constitute all possible reflection points that could potentially be evaluated in the second search step. Finally, the grey points constitute all possible expansion and shrink points for the third search step. Some expansion points are not shown to reduce clutter on the figure.

In the worst case, this scheme will require $O(N^2)$, where N is the size of the simplex, number of code-variants to be generated and maintained dynamically at each search step.

B. Retiring unreachable code-variants

For large-scale production runs, the number of code-variants generated grows fairly rapidly. Therefore, it is useful to have a mechanism to periodically retire (discard as not be used again) code-variants — an idea analogous to garbage-collection. The decision to retire a given code-variant will be based on the distance between the current simplex points and the point the variant is associated with. Put simply, if the code-variant is not reachable[8] within n number of search steps, it is retired. The value of n can be determined empirically.

[8]Reachability is determined by testing whether multiple simplex transformation steps can produce the given point.

C. Online Tuning for AMR Codes

Active Harmony's online tuning capability can help Adaptive Mesh Refinement (AMR) codes. Rather than relying on one global grid resolution, AMR codes have the ability to change the underlying granularity of the mesh or grid locally during a single production run of the application [13]. Areas in the domain that need finer grid resolution (e.g. area near the heat source in heat diffusion problem) can benefit from AMR technique because this allows shifting of the computational resources to the parts of the domain that need these resources the most. This dynamic change in the mesh structure also changes the execution characteristics of the application. Thus, an offline auto-tuner cannot adequately address the auto-tuning needs for AMR codes. Instead, Active Harmony's online adaptive code-generation and tuning is better suited to tune AMR codes. Our runtime auto-tuner can help AMR codes react to the changes in workloads and suggest different code-variants based on the grid resolution.

D. Power Auto-tuning

As we are entering the era of exascale systems, the key problem that the HPC community is trying to address is the "power wall" problem. The problem arises from the fact that as compute nodes (consisting of multi/many-cores) become increasingly powerful, they also become increasingly power-hungry. The problem is further exacerbated by the fact that these cores do need to be cooled down as well. Going forward, we see the power-aware computing research in the HPC community focus in two main areas. The first area of research will consist of projects [2], [9] that are involved in developing simple and low-cost hardware and software solutions to construct the power consumption profile of scientific applications. The second area of research will be led by auto-tuners that utilize the information provided by the first to automatically generate code-variants that reduce the power foot-print of different code-sections.

Active Harmony is well positioned to make contributions to the second area of research. An obvious starting point is to redefine the objective function from application-level performance (e.g. execution time, cache hits) to a system-level metric that captures power consumption (e.g. FLOPS/watt). We can then use the compiler-based auto-tuning design presented in this paper to find code-variants that reduce power consumption.

We expect application developers and library writers to implement and evaluate alternative implementations of key algorithms (e.g. data distribution, collective communication) to make their code ready for exascale systems. Each of these alternatives can have drastically different power consumption and performance profiles. For example, an implementation which aims at reducing the power consumption by limiting inter-node communication can increase computational load at core-level. Active Harmony can help developers quantify this tradeoff and make choices that balance application performance and power consumption. Active Harmony can also help

limit the load on some overused resource (e.g. interconnect) by switching to algorithms that reduce the load on the resource.

VIII. Conclusion

In this paper, we presented our runtime compilation and tuning infrastructure designed to improve the performance of parallel applications within a single execution. Since the system does not rely on any specific code-generation system, new code transformations can be easily incorporated within our system. We showed that for two programs, auto-tuning improves performance without training runs.

Even if the intent is to auto-tune an application for a specific machine and leave it fixed, runtime code-generation is useful. By generating and trying multiple configurations in a single run, we greatly reduce the time required to auto-tune a program.

Our system enables application developers to write applications once and adjust the application execution automatically when run on new systems. We demonstrated the value of our system by applying it on real application codes. The performance improvement of up to 46% for a 512-core parallel application execution can be achieved within a single execution of the application.

IX. Acknowledgements

The work supported in part by DOE grants DE-FC02-06ER25763 and ER25925 and NSF grant EIA-0080206. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] http://cseweb.ucsd.edu/groups/hpcl/scg/KeLP1.4/. [last accessed: Feb, 2010].

[2] D. Bedard, Min Yeol Lim, R. Fowler, and A. Porterfield. Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *Proceedings of the IEEE SoutheastCon '10*, pages 479 –484, 2010.

[3] Francine Berman and Richard Wolski. Scheduling From the Perspective of the Application. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 100, Washington, DC, USA, 1996. IEEE Computer Society.

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[5] C. Cascaval, E. Duesterwald, P.F. 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.

[6] National Energy Research Scientific Computing Center. www.nersc.gov.

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

[8] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey 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.

[9] Xizhou Feng, Rong Ge, and K.W. Cameron. Power and energy profiling of scientific applications on distributed systems. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, page 34, 2005.

[10] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[11] J.A. Joines and C.R. Houck. On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with GA's. pages 579–584 vol.2, June 1994.

[12] Chunhua Liao, Daniel J. Quinlan, Richard Vuduc, and Thomas Panas. Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization. In *LCPC '09: International Workshop on Languages and Compilers for Parallel Computing*, Newark, Delaware, 2009.

[13] D.F. Martin and K.L. Cartwright. Solving poisson's equation using adaptive mesh refinement. Technical Report UCB/ERL M96/66, EECS Department, University of California, Berkeley, 1996.

[14] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. MATE: monitoring, analysis and tuning environment for parallel distributed applications. *Concurr. Comput. : Pract. Exper.*, 19(11):1517–1531, 2007.

[15] Iulian Neamtiu. *Practical Dynamic Software Updating*. PhD thesis, University of Maryland, College Park, August 2008.

[16] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. *SIGOPS Oper. Syst. Rev.*, 31(5):276–287, 1997.

[17] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time . In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

[18] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 249–258, New York, NY, USA, 2006. ACM.

[19] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, page 172, Washington, DC, USA, 1998. IEEE Computer Society.

[20] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of Supercomputing '00*, November 2000.

[21] Sebastien Ros. "State of the Art Expression Evaluation". http://www.codeproject.com/KB/recipes/sota_expression_evaluator.aspx, November 2007.

[22] Vahid Tabatabaee, Ananta Tiwari, and Jeffrey 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.

[23] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *23rd IEEE International Parallel & Distributed Processing Symposium*, Rome, Italy, May 2009.

[24] Michael J. Voss and Rudolf Eigenmann. High-level adaptive program optimization with ADAPT. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001.

[25] Xingfu Wu, Valerie Taylor, Charles Lively, and Sameh Sharkawi. Performance analysis and optimization of parallel scientific applications on cmp cluster systems. In *ICPPW '08: Proceedings of the 2008 International Conference on Parallel Processing - Workshops*, pages 188–195, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Dazhi Yu and Sharath S. Girimaji. Multi-block lattice boltzmann method: Extension to 3d and validation in turbulence. *Physica A: Statistical Mechanics and its Applications*, 362(1):118 – 124, 2006.

Appendix

We provide the Extended Backus Naur Form (EBNF) grammar for the Constraint Specification Language below. The syntax for expressions is adapted from an expression evaluator example written in ANTLR [21]. This expression syntax closely resembles the C (and other high-level language) EBNF syntax. We use the following convention to present the CSL grammar. Symbols and strings that appear as a part of the parameter specification are "double-quoted" (e.g. "search",

"+"). Unquoted curly brackets (i.e. { and }) are used for better readability of the grammar. Symbols ?, ⋆ and + have the following meaning:
?: Symbol (or a group of symbols in curly brackets) can appear zero or one time.
⋆: Symbol (or a group of symbols in curly brackets) can appear zero or multiple times.
+: Symbol (or a group of symbols in curly brackets) has to appear at least once and can appear multiple times.

```
<param space> ::= "search" "space" <param space name>
                  "{"
                     <space body>
                  "}"

<space body> ::=       {<constant decl>}*
                       {<code region decl>}*
                       {<region set decl>}*
                       {<param decl>}+
                       {<constraint decl>}*
                       {<constraint spec>}*
                       {<grouping info>}*
                       {<ordering info>}*

<param space name> ::= <ident>

Constant Declaration Part:
<constant decl> ::= "constants" "{" {<constants>}+ "}"

<constants>::=<param type> <const name>"="<dval>";"

<const name> ::= <ident>

Code Region and Region Set Declaration Part:
<code region decl> ::= "code_region" <region name> ";"

<region name> ::= <ident>

<region set decl> ::= "region_set" <region set name>
                      "["
                      <region name> <region name list>
                      "]"";"

<region set name> ::= <ident>

<region name list> ::= <region name> {"," <region name>}*

Parameter Declaration Part:
<param decl> ::= "param" <param name> <param type>
                 "{"
                  <drestrc>{<def spec>}?{<reg spec>}?
                 "}"

<param name> ::= <ident>

<param type> ::= "int"
                 | "float"
                 | "string"
                 | "bool"
                 | "mixed"

Domain Specification:
<drestrc>::="range" "["<dval>":"<dval>(":"<dval>)?"]"";"
          |"prange" "["<dval>":"<dval>":" <dval>"]"";"
          |"array"  "[" arr ("," "[" arr "]")* "]" ";"

<array> ::= <dval> ("," <dval>)*

<dval> ::= <int>
        | <float>
        | <str>
        | <bool>

<def spec> ::= "default" <int> ";"
             | "default" <float> ";"
             | "default" <str> ";"
             | "default"  <bool> ";"

<reg spec> ::= "region" <region set name> ";"

Constraint Declaration Part:
<constraint decl> ::= "constraint" <constraint name>
                                "{" <expr> ";" "}"

<constraint name> ::= <ident>

<expr> ::= <logicalexpr>

<logicalexpr> ::= <boolandexpr>{"||"<boolandexpr>}*

<boolandexpr> ::= <eqexpr>{"&&"<eqexpr>}*

<eqexpr> ::= <relexpr>{("="|"!=")<relexpr>}*

<relexpr> ::= <addexpr>{("<"|"<="|">"|">=")<addexpr>}*

<addexpr> ::= <multexpr> {("+"|"-")<multexpr>}*

<multexpr> ::= <powexpr> {("*"|"/"|"%")<powexpr>}*

<powexpr> ::= <unaryexpr> {"^" <unaryexpr>}*

<unaryexpr> ::= <primexpr>
              | "!" <primexpr>
              |  "-" <primexpr>

<primexpr> ::= "(" <logicalexpr> ")"
             |  <dval>
             |  <param ref>

<param ref> ::= <ident> | <ident> "." <ident>
                        |  <ident> "." "value"

Bind everything together with specification:
<constraint spec> ::= "specification"
                       "{"
                        <specexpr>";"
                       "}"

<specexpr> ::= <primspecexpr>(("&&"|"||")<primspecexpr>)*

<primspecexpr> ::= "(" <specexpr> ")"
                 | <constraint name>

Parameter Grouping:
<grouping info> ::= "groups"
                    "{" {<set decl>}+ "}"

<set decl> ::= "set" "["
                     <param ref> {"," <param ref>}*
                    "]" ";"

Parameter Ordering:
<ordering info> ::= "ordering"
                    "{" <param list> "}" ";"

<param list> ::= <param name>
               | <param name> {"," <param name>}+
```