Finding Chapel's Peak: Introducing Auto-Tuning to the Chapel Parallel Programming Language

Ray S. Chen
Department of Computer Science
University of Maryland,
College Park, MD 20742
Email: rchen@cs.umd.edu

Abstract

The benefit of automated application tuning has been the focus of numerous research projects, yet applying this technology remains a completely manual and labor-intensive process. This paper explores first steps towards reducing the adoption cost of auto-tuning in the context of Chapel; a parallel programming language whose development is being led by Cray Inc. Novel information can be inferred from Chapel's syntax to locate likely auto-tuning parameter candidates and present them to a user for verification. The set of accepted parameters may then be used directly by Tuna, an Active Harmony command-line tuner whose development was motivated by this research project. To demonstrate the immediate utility of this system, two Chapel programs are shown to be auto-tunable with little or no internal knowledge of the program source. Finally, the groundwork for future automated auto-tuning in Chapel is laid through the development of a performance unit-test for thread/cache optimization.

1) Introduction

Ironically, the use of current auto-tuning software and frameworks is an intensely manual process. This is especially of true of recent advancements by Tiwari [7]. In his dissertation, he demonstrates auto-tuning within a single function of an application by effectively merging just-in-time compilation with the traditional feedback-directed optimization loop. A system that tunes at such a fine-grained level can provide excellent properties, such as performance improvements within a single execution. However, a high cost is paid during the initial incorporation of the auto-tuning framework. Few applications are written to handle the external modification of their internal variables during execution. Without skilled technicians and domain experts to analyze and prepare the target, it can at best be treated as a block box to which parameters are fed.

To a certain degree, this complexity is unavoidable due to the wide scope of auto-tuning. However, there are two conceptual tasks common to the adoption phase of every auto-tuning project. The first involves identifying tunable parameters and bounding their possible values. The second involves locating the regions of the source code that are affected by the chosen parameters. Command-line parameter tuning implicitly selects the entire binary, whereas adaptive code-generation requires the selection of specific functions or loops. The key observation is that these properties are difficult to infer after the application has been written. Presumably, the original application developers were familiar

enough with the problem domain to identify the tunable variables and regions of their own code. The task of automating auto-tuning would be greatly simplified had they been philanthropic enough to annotate the source with this information.

Of course, it is unreasonable to expect all applications to be developed with auto-tuning in mind; we must gather this information through alternate means. Fortunately, we need not abandon the original author as a resource for auto-tuning information. If the application was written in a modern programming language such as Chapel [1], novel information can be inferred regarding internal program variables and their intended use. We explore the possibility of using this information to reduce the adoption cost of auto-tuning technologies.

2) Chapel

Chapel is a parallel programming language born from Cray, Inc.'s entry in the High Productivity Computing System program. The goal of Chapel is to improve the productivity of high-performance computing by improving the programmability of multi-core systems and large-scale parallel computers. High-level abstractions for data and task parallelism free users from the tedium typical of low-level parallel programming languages while providing additional correctness and portability benefits. Performance is also a fundamental goal; Chapel seeks to meet or exceed the performance of programming models such as MPI.

We focus our overview of Chapel to the two features we plan to leverage for auto-tuning: its constructs for parallelism, and its notion of configuration variables.

2.1) Task Abstraction

Chapel uses the task abstraction to represent units of parallel work. Threads are considered a system-level concept by which Chapel executes its tasks. The distinction of tasks from threads is necessary since the management of (POSIX) threads require interaction with the system kernel, and hence incur higher overhead than what Chapel intends for its parallelism. Admittedly, this distinction is muddied in the default case, where tasks are assigned to threads in a first-in-first-out manner. Once paired, a task must run to completion before its thread will be released back to the pool of available threads. This tasking implementation was chosen for portability and allows Chapel to execute correctly on virtually any platform that supports threads.

To fully utilize the task abstraction, Chapel should be paired with a user-level threading library [2] such as Qthreads [3] from Sandia National Laboratories. This allows task generation to be guided by parallelism intrinsic to the problem-domain, whereas thread generation can be guided by hardware resources such as physical cores or logical processor threads. An open question posed by this distinction is the optimal thread/task ratio. The problem is likely system-dependent, and we explore answers to that problem later in this paper.

2.2) Configuration Variables

Chapel also provides the notion of configuration variables. Semantically, these are identical to normal variables, except that code is automatically generated for them to be overridden at load-time via

command-line parameters. This seemingly simple feature has deep implications for the auto-tuning community. Configuration variables are implicitly designed to handle being changed from one execution to the next, which in itself increases the likelihood of these variables as auto-tuning parameters. Unfortunately, these variables may also alter the program's correctness, such as problem-size modification. These constitute false-positives, and prevent us from using this class of variable carte blanche. However, they certainly represent a reasonable place to begin the search for candidate parameters.

More importantly, configuration variables provide incentives for application developers to delay coding decisions that would otherwise require undue effort to resolve optimally. This is especially true of performance related coding decisions. Consider buffer size as an example. Most high-performance computing (HPC) developers understand that buffer sizes have an effect on the performance of their application. However, the determination of an optimal value is based on several factors that may not be available at development time, such as influx versus outflow. Worse still, these values may be affected by CPU cache or memory page sizes, which vary from system to system. With Chapel, the programmer is free to add the "config" keyword, choose a reasonable value, and move on.

Configuration variables open a path for developers to implicitly express that better values may exist. It is our belief that this knowledge can be leveraged by auto-tuning technology as a first step towards fully-automatic auto-tuning. To that end, we have modified the Active Harmony framework to more easily work with Chapel configuration variables.

3) Active Harmony

The Active Harmony framework [4] has been used to auto-tune a wide range of applications at varying levels of granularity. The framework provides several methods for users to input a set of parameters and associated value ranges. A search-based strategy is then used to navigate the parameter space that would otherwise be computationally intractable to exhaustively explore. Specific parameter configurations are directly tested by the target application and a performance result is returned to the framework. These performance result values drive a derivative-free simplex search method such as Parallel Rank Ordering [5] or Nelder-Mead [6] to produce new candidate parameter configurations, which closes the traditional feedback loop common to most auto-tuning systems such as Orio [8] or POET [9].

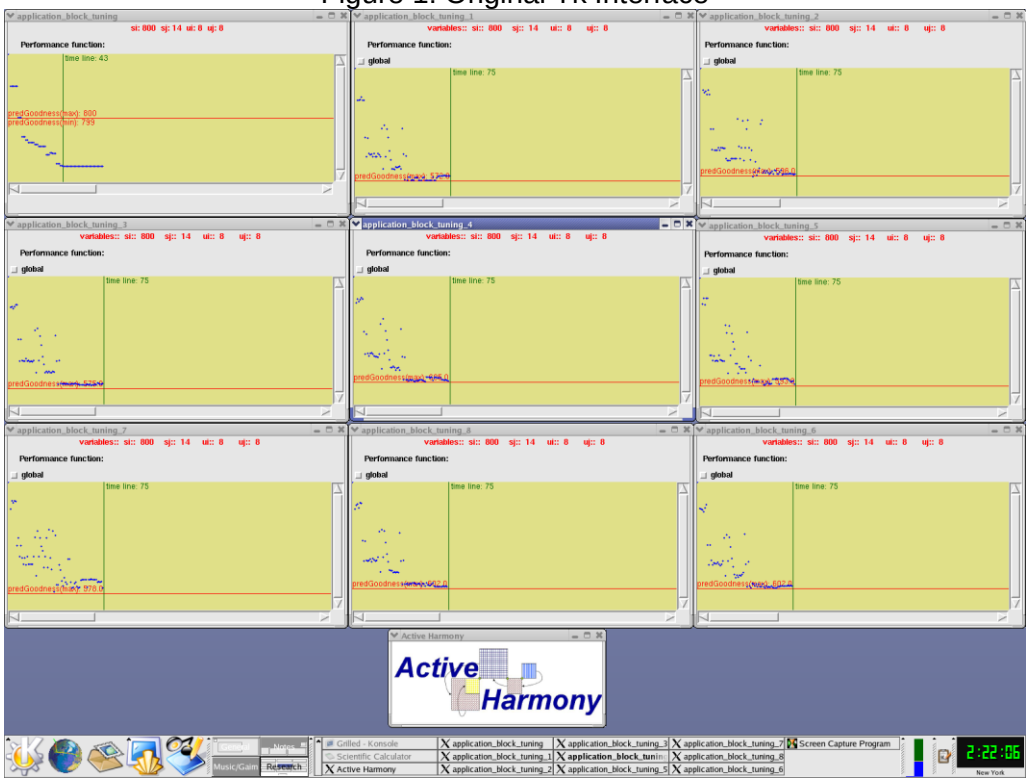
Several additions and modifications were required to adapt Active Harmony for use with Chapel's configuration variables. Firstly, a library dependency has been removed from Active Harmony to match the portability of the Chapel compiler package. Secondly, two value-added applications have been written and included with Active Harmony to improve the utility of the framework.

3.1) Web-based Interface

The original graphical interface for Active Harmony was written using the Tk toolkit. This provided the benefit of a quick development cycle that integrates naturally with the search implementation written in Tcl. However, this introduced the dependency on a sufficiently recent version of Tk on each of our

target platforms. Additionally, network latency becomes a problem if the server is launched on a remote computer. Due to Tk's reliance on X11 for its line protocol, the interface becomes sluggish when displayed over a local area network, and worse for the Internet at large. Finally, the Tk interface produces $n + 2$ windows, where n is the number of target application nodes. This structure will encounter scalability issues in the near future where auto-tuning is planned for super-computers with hundreds of thousands of compute nodes or more.

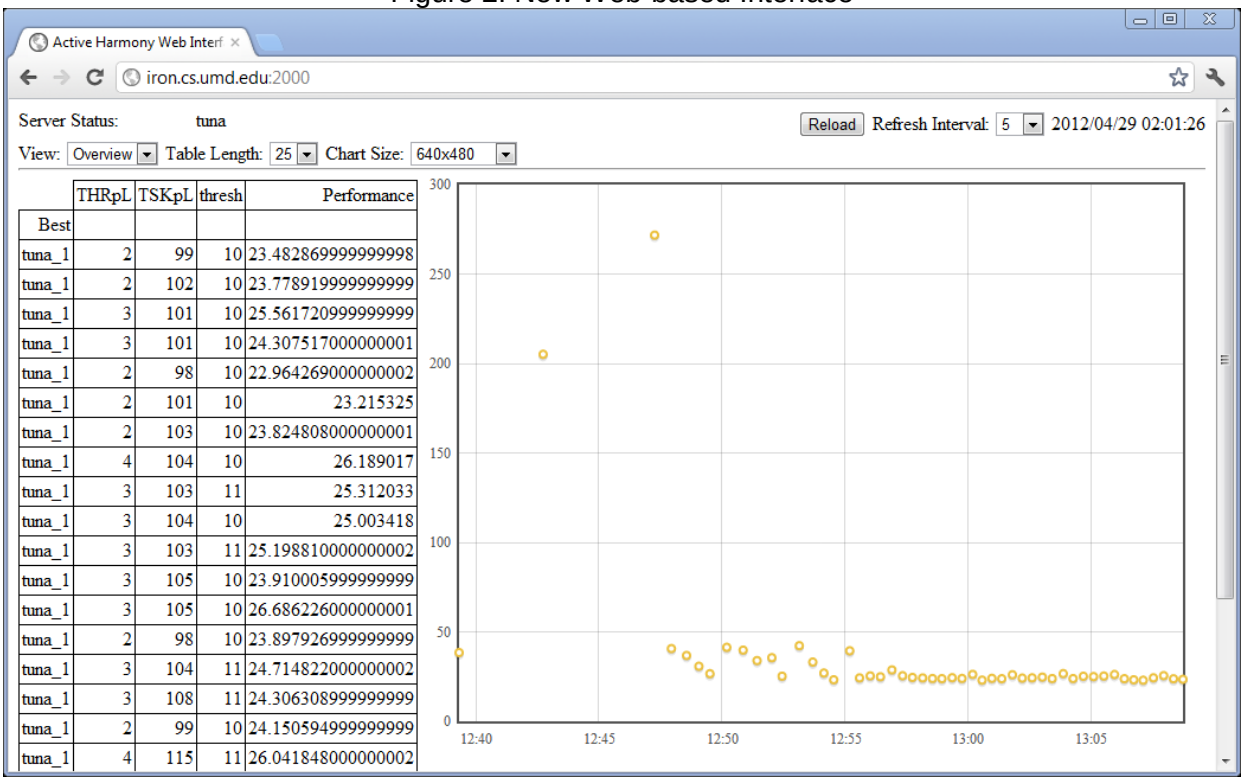
Figure 1: Original Tk Interface



Thankfully, the ubiquity of web-based javascript applications presents an opportunity to bypass all of these issues at once. Web browsers effectively provide a graphical display engine local to the user, freeing Active Harmony to transfer minimal search progress data and a small javascript support file. As an added benefit, the new interface is only delivered upon request, which minimizes server-side processing and allows multiple clients to connect and view a single optimization run. To remedy scalability issues, the new design focuses on visualizing the global parameter search space, rather than the progress of at each local compute node. Finally, all representations of the search are available through a single display window and summarized by a general performance timeline.

To provide this benefit without adding additional libraries, we developed and integrated a simple HTTP/1.1 server into the communication multiplexer of the Active Harmony server. Our web server listens for specific commands via the incoming URL to provide either the javascript application, or raw search progress update data.

Figure 2: New Web-based Interface



	THRpL	TSKpL	thresh	Performance
Best				
tuna_1	2	99	10	23.482869999999998
tuna_1	2	102	10	23.778919999999999
tuna_1	3	101	10	25.561720999999999
tuna_1	3	101	10	24.307517000000001
tuna_1	2	98	10	22.964269000000002
tuna_1	2	101	10	23.215325
tuna_1	2	103	10	23.824808000000001
tuna_1	4	104	10	26.189017
tuna_1	3	103	11	25.312033
tuna_1	3	104	10	25.003418
tuna_1	3	103	11	25.198810000000002
tuna_1	3	105	10	23.910005999999999
tuna_1	3	105	10	26.686226000000001
tuna_1	2	98	10	23.897926999999999
tuna_1	3	104	11	24.714822000000002
tuna_1	3	108	11	24.306308999999999
tuna_1	2	99	10	24.150594999999999
tuna_1	4	115	11	26.041848000000002

3.2) Tuna: The Command-line Tuner

In attempting to tune Chapel's configuration variables, the need for a general command-line tuning utility becomes clear. Different applications written in Chapel are likely to produce distinct sets of configuration variables for tuning. Active Harmony provides a client API that allows developers to build a tuner for their specific Chapel application, but these tuners would consist of overwhelmingly redundant code. Tuna was written to reduce future effort and facilitate the general use of auto-tuning technology.

Tuna maintains generality by accepting a tunable variable specification within its own command line or configuration file. The specification consists of a type, name, and value range; information required by the Active Harmony server to generate and bound the search space. Tuna then launches the target application in a loop, receiving candidate configurations from the server and reporting the tested performance value back to the server at the end of each execution. A printf-style format string is available should the target application require its input parameters in a specific format.

Tuna provides four built-in methods for measuring the performance of a target application. Wall time, user time, or system time used by the target application can automatically be use as a metric. The final method of performance measurement was designed with extension in mind. It involves monitoring the target application's output and reading a performance value from its final line. This allows virtually any measure of performance, so long as it can be collected by an external wrapper program and printed.

Figure 3: Tuna Usage Example

```
> ./tuna -i=tile,1,10,1 -i=unroll,2,12,2 –n25 matrix_mult -t % -u %
```

The usage example in figure 3 defines two integer parameters for the Active Harmony server. One variable, named "tile," is permitted to be between 1 and 10, inclusive. The other variable, named "unroll," is permitted to be even numbers between 2 and 12, inclusive. In this example, Tuna will launch the matrix_mult binary with server-chosen values of "tile" and "unroll" as arguments 2 and 4, respectively. The target application will be launched 25 times, or until the search converges, whichever occurs first. Wall time is the measure of performance, since it is otherwise unspecified.

Tuna was used extensively for the performance tests reported in this paper. However, the utility of Tuna extends beyond the realm of Chapel and configuration variables. Any application that provides command-line parameters related to performance is immediately available for tuning. As an example, the GCC compiler suite provides hundreds of command-line switches to control various details of its compilation process. Finding optimal values for these switches would normally be a daunting task but, with Tuna, the user need only specify which switches are relevant for their optimization task and a method to measure the resulting performance.

4) Tuning Chapel Applications

Chapel applications are at a particular advantage with regard to configuration variable auto-tuning. As alluded to earlier, Chapel makes a distinction between application tasks and system threads. However, the method for determining an optimal thread/task ratio is unclear, even from a theoretical standpoint. Perhaps anticipating this dilemma, the Chapel developers provide three built-in configuration variables to control the number of tasks and threads that a Chapel application will implicitly initiate. This automatically makes every Chapel application an excellent candidate for auto-tuning, even if the application developer never declares a single additional configuration variable.

It seems reasonable to believe that the optimal ratio is dependant on application factors such as the amount of synchronous code, as well as system-level factors such as thread scheduling. To test our hypothesis, we use Tuna to perform the same tuning tasks on four different machines. Each machine represents a different CPU type, core count, or operating system.

Table 1: Test Platform Architecture Specifications

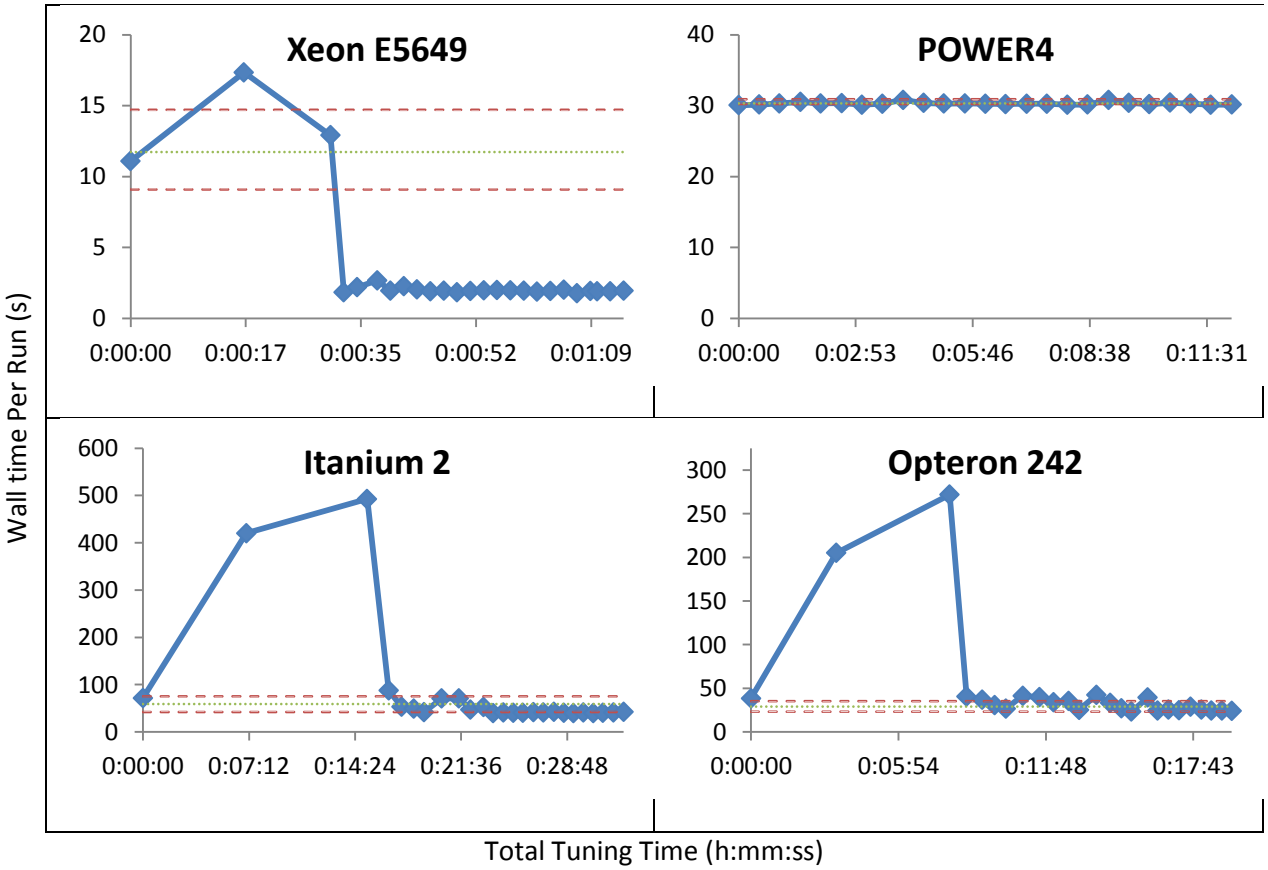
CPU Type	Xeon E5649	POWER4	Itanium 2	Opteron 242
Core Speed	2.53 GHz	1.1 GHz	900 MHz	1.6 GHz
Cores/Threads	6/12	1/1	2/2	2/2
L1 Data Cache	32 KB	32 KB	16 KB	64 KB
L2 Cache	0.25 MB	5.60 MB	0.25 MB	1.00 MB
L3 Cache	12 MB	128 MB	24 MB	N/A
System RAM	24 GB	6 GB	512 MB	4 GB
OS	Linux	Linux	Linux	Linux
Word Size	64-bit	32-bit	64-bit	32-bit

4.1) Tuning Quicksort

We begin our tuning experiments with a quicksort example provided in the source distribution of Chapel. Without viewing the source code, one can immediately see this application provides several configuration variables by using the "--help" switch. Chapel automatically adds a help routine to any program it compiles which includes a listing of valid configuration variables. Even at this point, the "thresh" variable looks interesting. A brief investigation of the code reveals this particular implementation of quicksort allows the user to choose a maximum recursion depth before a serial bubble sort is used as the base case.

Our test involves sorting a 64-megabyte array of double-precision floating-point random values. We use Tuna to create a search space based on three variables: the number of threads to spawn (between 1 and 16, inclusive), the number of tasks to automatically initiate (between 1 and 256, inclusive), and the application-specific threshold depth (between 1 and 16, inclusive). Ten runs of quicksort using default values are used as a control, and are along with the tuning data. Default run minimum and maximum values are represented via dashed lines, and the average is represented via a dotted line between them.

Figure 4: Tuning Quicksort Across Multiple Platforms



In the best case, Active Harmony was able to find configurations that out-perform the default by over 300% compared to the average of our 10-run default-value performance. This is not a miracle of auto-

tuning; the original application only generates one level of parallelism by default. On the one hand, it should be no surprise that an embarrasingly parallel program runs faster in the presence of more parallelism. On the other hand, a better default threshold value may not exist. Recall that the default threading model of Chapel spawns a new thread for each task that is requested, and intuition suggests spawning more threads than logical cores will result in performance-degrading overhead[1]. If the number of system cores is unavailable, it becomes impossible to provide an optimal default threshold value. It should be noted that Chapel does provide a mechanism for retrieving the number of system cores, and this implementation could easily be re-written to take advantage of this information.

The POWER4 result also deserves investigation. As the only machine with a single core, it should be heavily affected by additional threads being spawned. Instead, its execution times are completely unwavering. Our only explanation is that Chapel detected the system to be non-parallel and runs serially, no matter how many threads or tasks are requested.

Overall, Active Harmony is able to find acceptable configurations within four search steps. The search could be further improved had the search space been tailored to the target architecture using external knowledge such as core count; the tested search space includes the sequential non-parallel case. As a methodology, we wished to keep the search as wide as possible to capture the generality of Active Harmony's simplex search method. This is an excellent result for Chapel and automated auto-tuning. With little more than a glance at the source code, this example proved itself to be tunable with positive results.

Table 2: Comparison of Best Quicksort Configurations

Platform	Xeon E5649	POWER4	Itanium 2	Opteron 242
Best Default	Wall time: 9.10s	Wall time: 30.23s	Wall time: 42.28s	Wall time: 23.28s
Best Active Harmony	Threads: 16 Tasks: 241 Threshold: 16 Wall time: 1.78s	Threads: 1 Tasks: 1 Threshold: 1 Wall time: 30.07s	Threads: 3 Tasks: 67 Threshold: 12 Wall time: 39.36s	Threads: 2 Tasks: 93 Threshold: 10 Wall time: 23.23s

4.2) Tuning HPC Challenge Benchmarks

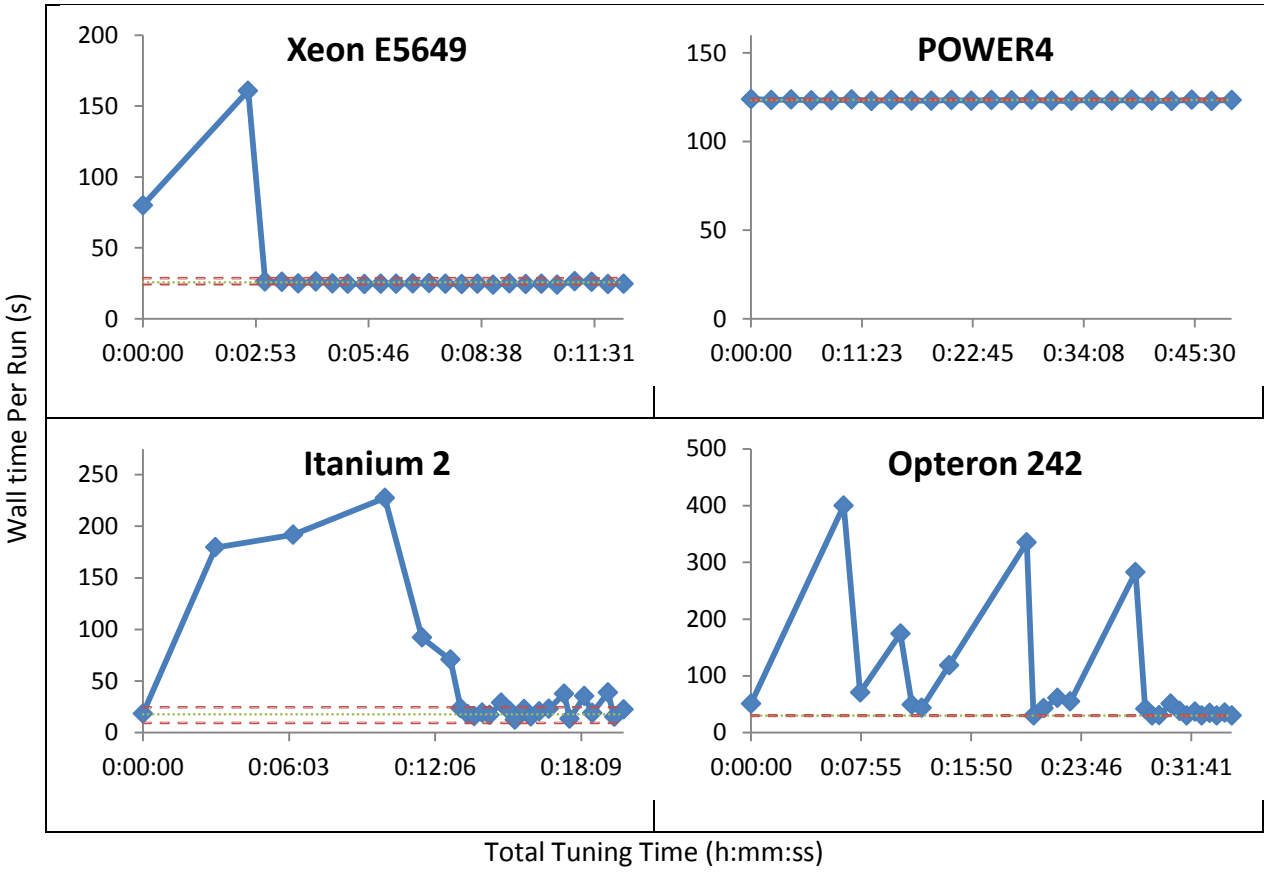
Moving beyond toy programs, the source distribution of Chapel version 1.5 also includes implementations of selected real-world parallel benchmarks written in Chapel. Since these programs represent highly tuned benchmarks, they offer no configuration variables that affect performance without affecting the problem being solved. Nevertheless, Chapel's built-in configuration variables provide enough leverage for tuning. We use an implementation of the STREAM benchmark from HPC Challenge, which is a simple synthetic benchmark that measures sustainable memory bandwidth and the corresponding computation rate for a simple vector kernel.

Again, Tuna was used to create a search space based on three variables: the number of threads to spawn (between 1 and 16, inclusive), the number of tasks to automatically initiate (between 1 and 256, inclusive), and the minimum task granularity (between 256 and 32768, by increments of 256). The last

[1] Our experiments corroborate this intuition, in general.

variable describes how tasks should be initiated for data parallel loops, meaning each task will be responsible for at least this number of loop iterations. The granularity may ultimately be interesting for cache studies, which we investigate in the section 4.3. We compare against ten runs of STREAM using its default values. Default run minimum and maximum values are represented via dashed lines, and the average is represented via a dotted line between them.

Figure 5: Tuning STREAM Across Multiple Platforms



Active Harmony had greater difficulty finding optimal configurations for this real-world application compared to our toy quicksort application. This is unsurprising; benchmark code is designed to run as optimally as possible. Considering that STREAM is effectively a worse-case scenario, Active Harmony still performs exceedingly well as optimal values are always found within 10 search steps.

Table 3: Comparison of Best STREAM Configurations

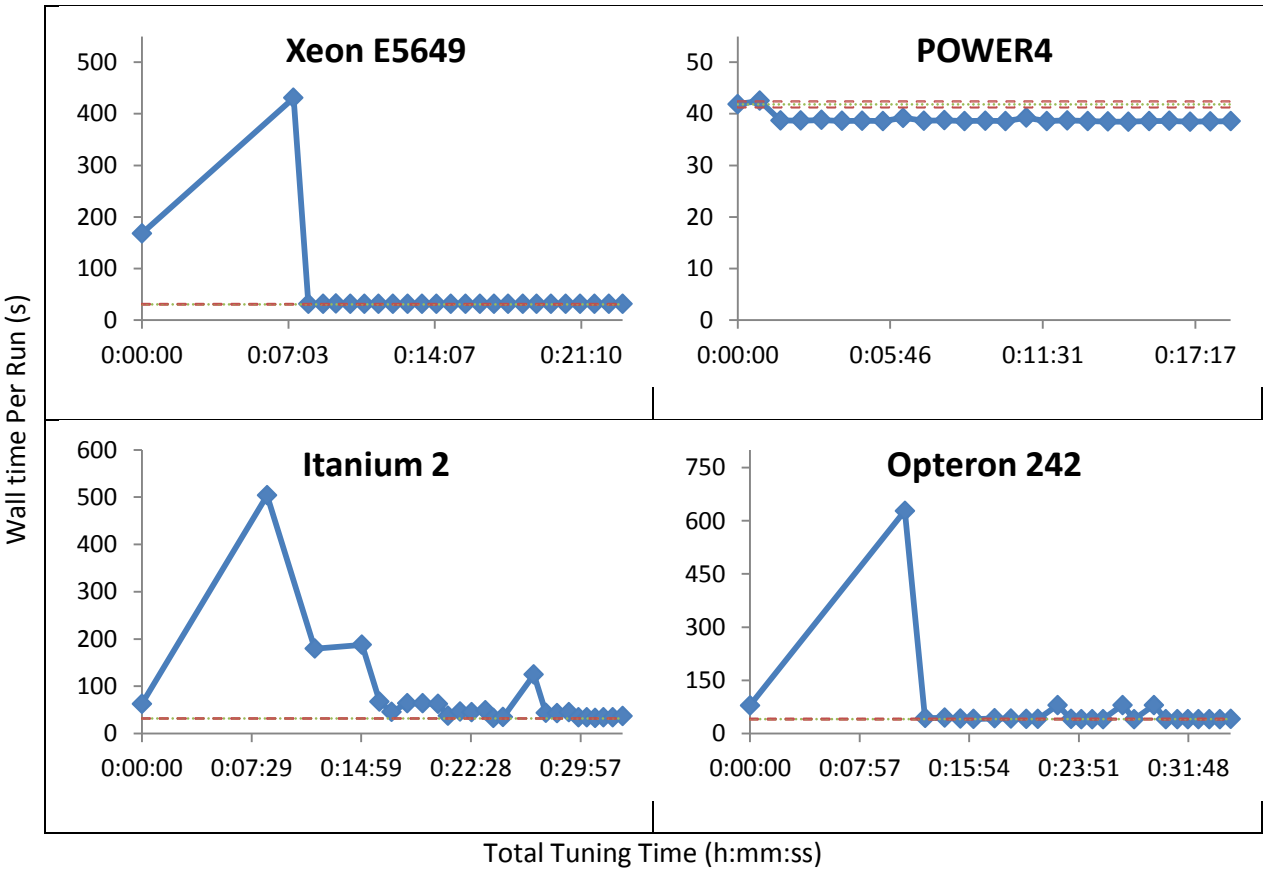
Platform	Xeon E5649	POWER4	Itanium 2	Opteron 242
Best Default	Wall time: 24.40s	Wall time: 123.25s	Wall time: 9.50s	Wall time: 29.38s
Best Active Harmony	Threads: 12 Tasks: 256 Granularity: 30464 Wall time: 1.78s	Threads: 16 Tasks: 167 Granularity: 27392 Wall time: 122.86s	Threads: 1 Tasks: 2 Granularity: 256 Wall time: 12.45s	Threads: 2 Tasks: 125 Granularity: 14848 Wall time: 29.67s

4.3) Towards Performance Unit Tests

To shed some light on the question of optimal thread/task ratio detailed in section 2.1, we developed a simple parallel cache memory stress test in Chapel. The test consists of a data-parallel loop that iterates over an array of word-size integers, reading the value at $i - 1$, and writing an incremented version of that value into position i. This high-locality memory access should be sensitive to cache size and access patterns, as determined by task count and loop granularity.

Like the HPC Challenge benchmark, Tuna was used to create a search space based on threads, tasks, and task granularity. For comparison, we hand-calculate a theoretically optimal configuration based on CPU L1 cache sizes and system core count. Since the optimal task count is unclear, that decision is left to Chapel's default. As with previous figures, these hand-calculated optimal values are represented via dashed lines, with the average as the dotted line between them.

Figure 6: Optimal Parameter Search Across Multiple Platforms



Again, the POWER4 results give us pause. If our prior supposition was true, none of our tunable parameters should have any effect on this platform. Yet, our search was able to produce configurations that improved upon theoretically optimal results. Deeper analysis will be required to answer this question adequately, and is beyond the scope of this paper.

Table 4: Comparison of Best Cache Configurations				
Platform	Xeon E5649	POWER4	Itanium 2	Opteron 242
Best Manual	Threads: 12 Granularity: 4096 Wall time: 30.20s	Threads: 1 Granularity: 8192 Wall time: 41.26s	Threads: 2 Granularity: 2048 Wall time: 31.95s	Threads: 2 Granularity: 16384 Wall time: 39.57s
Best Active Harmony	Threads: 16 Tasks: 256 Granularity: 27648 Wall time: 31.16s	Threads: 16 Tasks: 256 Granularity: 25600 Wall time: 38.44s	Threads: 3 Tasks: 32 Granularity: 256 Wall time: 33.08s	Threads: 2 Tasks: 256 Granularity: 16384 Wall time: 40.14s

Surprisingly, Active Harmony was able to find configurations that exceed the performance of our hand-crafted optimal in several cases. As expected, the optimal values determined for each variable are highly system-dependent. There are no universal values for optimal performance. We present this as evidence that there is plenty of room for auto-tuning in the realm of performance optimization.

We envision the use of these short trials to help guide performance decisions on the target system at run time. By developing a suite of performance "unit-tests," developers would be free to save optimization decisions until on-site evidence is available.

5) Augmenting Chapel

Configuration variables have proven useful in our tuning experiments, but more information is required for the system to become fully automatic. Namely, bounding ranges for each configuration variable were manually selected before being sent to Tuna. Again, the original developer is a likely resource for this information but, in this case, they have no means to express it. The Chapel language as written currently does not allow for ranges to be associated with configuration variables.

We modified the Chapel language grammar as a proof of concept to enable the association of value ranges with configuration variables. No new keywords were required; we made use of the existing "in" keyword and Range variable type. The range may be specified directly after the variable name, as shown in the following example. In our implementation, these ranges are correctly parsed and stored by the Chapel compiler.

Table 5: Augmented Chapel Examples	
Original Chapel	`config var num1 = 5;` `config var num2 = 5 :int(64);`

Augmented Chapel	`config var num3 in 1..100 = 5;`
	`config var num4 in 1..100 by 5 = 5; // Invalid value`
	`config var num5 in 1..100 by 5 align 50 = 5;`
	`config var num6 in 1..100 :int(64) = 5;`
	`config var num7 in 1..100 by 5 : int(64) = 5; // Invalid value`
	`config var num8 in 1..100 by 5 align 50 :int(64) = 5;`

Additional code is automatically generated and run at program launch time to verify each configuration variable. For instance, num4 and num7 of the example in table 3 provide invalid default values. These

lines will compile successfully, but will cause a fault at run-time if not overridden by the user to a valid value such as 1, 6, or 11.

We believe the addition of a bounding range to configuration variables would be useful outside the realm of auto-tuning because it allows for additional correctness guarantees on user input. The association of a value range is considered optional in the grammar, so existing code maintains its correctness. Additionally, applications will incur minimal run-time overhead, as these checks are defined to occur once at launch time only. We intend to submit our changes to the developers of Chapel for consideration, along with our rationale. The inclusion of our patch would greatly facilitate the automation of auto-tuning on Chapel applications.

6) Future Work

This work stops short of fully-automatic auto-tuning within Chapel in two respects. Firstly, Active Harmony has no means to automatically use configuration variable information; Tuna is still required to bridge that gap. However, if the changes we propose to Chapel are accepted, all the information Active Harmony requires for parameter space bounding will reside within the compiled application. Optimization could then be enabled with an additional run-time command line switch. Secondly, user intervention is required to determine which configuration variables are suitable for tuning. This is an open problem, and will require more research for a solution.

Looking inward, this work only investigates auto-tuning at the command-line parameter level. Tiwari successfully showed that auto-tuning can be applied to individual loops of parallel applications [7]. Since Chapel clearly delineates which portions of its code execute in parallel, the possibility of automatically applying their loop level optimization to compiled Chapel applications should be investigated.

Looking outward, users of the Chapel language represent a relatively small audience. However, increasing the applicability of fully-automatic auto-tuning requires adapting it to existing languages such as C or Fortran that do not implement configuration variables. Determining which variables may be arbitrarily modified is an open problem, and it is unclear how far static analysis can be utilized towards this end. As an alternative, the configuration of libraries can also widen the audience-base of fully-automatic auto-tuning users. For example, the upcoming MPI 3.0 standard makes provisions for control variables, which are conceptually equivalent to Chapel's configuration variables. Adapting Active Harmony to work with MPI's control variables will immediately introduce fully-automatic auto-tuning to a new audience and possibly lead to new research opportunities.

7) Conclusion and Summary

This work represents first steps towards a fully-automatic auto-tuning system. The Chapel programming language provides an excellent base for these first steps due to its notion of configuration variables. Encoded in the definition of such variables is the implication of mutability, specifically at program launch

time. We tested this implication on three different Chapel applications, and were successfully able to tune these programs without modification, and with little or no familiarity with their source code.

Active Harmony was modified in two ways to support this work. Firstly, a new user interface was written that reduces the number of dependent libraries, uses a far more efficient data protocol, allows for multiple clients, and is provided on demand. Secondly, the Tuna command-line tuner was introduced which allows anyone to reap the benefits of command-line auto-tuning without the need of additional coding.

Improvements to the Chapel programming language were also implemented and will be submitted to the developers at Cray Inc. for consideration. If approved, further gains in fully-automatic auto-tuning may be made. In the mean time, we introduce the possibility of performance unit tests to determine system-local optimal values.

Bibliography

[1] David Callahan, Bradford L. Chamberlain, Hans P. Zima. The Cascade High Productivity Language. *In 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 52-60. IEEE Computer Society, April 2004

[2] Kyle B. Wheeler, Richard C. Murphy, Dylan Stark, Bradford L. Chamberlain. The Chapel Tasking Layer Over Qthreads, *CUG 2011*, June 2011

[3] Wheeler, K.B.; Murphy, R.C.; Thain, D., "Qthreads: An API for programming with millions of lightweight threads," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, vol., no., pp.1-8, 14-18 April 2008

[4] C. Ţăpuş, I. Chung, and J. K. Hollingsworth. Active harmony: towards automated performance tuning. In *Proceedings of Supercomputing '02*, pages 1-11, November 2002

[5] Ananta Tiwari, Vahid Tabatabaee, and Jeffrey K. Hollingsworth. 2009. Tuning parallel applications in parallel. Parallel Comput. 35, 8-9 (August 2009), 475-492

[6] J. Nelder and R. Mead. A Simplex Method for Function Minimization. Computer Journal, 7:308-313, 1965

[7] Tiwari, A. N. (2011). *Tuning parallel applications in parallel.* University of Maryland, College Park). *ProQuest Dissertations and Theses*

[8] Hartono and S. Ponnuswamy. Annotation-Based Empirical Performance Tuning Using Orio. In Proceedings of the 23rd International Parallel and Distributed Processing Symposium, May 2009

[9] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized Optimizations for Empirical Tuning. *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, pages 1-8, March 2007