International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

Special Issue Article

The International Journal of High Performance Computing Applications
27(4) 394–402

DOI: 10.1177/1094342013493198
hpc.sagepub.com
SAGE

Towards fully automatic auto-tuning: Leveraging language features of Chapel

Ray S Chen and Jeffrey K Hollingsworth

Abstract

Application auto-tuning has produced excellent results in a wide range of computing domains. Yet adapting an application to use this technology remains a predominately manual and labor intensive process. This paper explores first steps towards reducing adoption cost by focusing on two tasks: parameter identification and range selection. We show how these traditionally manual tasks can be automated in the context of Chapel, a parallel programming language developed by Cray Inc. Potential auto-tuning parameters may be inferred from existing Chapel applications by leveraging features unique to this language. After verification, these parameters may then be passed to an auto-tuner for an automatic search of the induced parameter space. To further automate adoption, we also present Tuna: an auto-tuning shell designed to tune applications by manipulating their command-line arguments. Finally, we demonstrate the immediate utility of this system by tuning two Chapel applications with little or no internal knowledge of the program source.

Keywords

auto-tuning, fully automatic, performance, Active Harmony, Tuna, Chapel

1. Introduction

Automatic performance tuning, or auto-tuning, has been the focus of numerous research projects over the past decade. Multiple auto-tuning systems (Ţăpuş et al., 2002; Yi et al., 2007; Hartono et al., 2009) have been developed to automatically search parameter spaces that are computationally intractable to explore exhaustively. Fundamentally, all auto-tuning systems are based on a feedback loop. The tuner generates a configuration that will be directly tested by the target application. Performance values collected during the test are then reported back to the tuner, allowing it to generate another configuration based on this new information.

In all research to date, the term "automatic" refers only to the methods that drive a configuration search. No consideration is given towards automatically identifying possible tunable parameters or incorporating the auto-tuner with the target application. Both of these tasks require human guidance from domain experts, auto-tuning specialists, or both. We believe this represents a significant gap in the field, and ultimately limits the use of auto-tuning technology.

In this paper, we explore first steps towards a fully automatic auto-tuning system that eliminates the need for manual intervention. By leveraging language features at the source code level, one may infer information regarding potential tunable parameters and greatly reduce the burden of producing a tuning parameter space.

2. Related work

A wide breadth of computing fields have successfully benefited from auto-tuning technology. In the realm of e-commerce services, Chung and Hollingsworth (2004) apply auto-tuning to the TPC-W benchmark to achieve throughput improvements of up to 70%. In high-performance computing, Shin et al. (2010) use auto-tuning and specialization to improve run-time of a spectral-element code by up to 38%. More recently, Rahman et al. (2011) target power consumption as a performance metric to optimize the efficiency of scientific codes.

The depth to which auto-tuning can be applied has also been investigated. A first-order application of auto-tuning involves modifying an application's input parameters and running the application to completion. Tiwari and Hollingsworth (2011) provide a finer granularity by demonstrating auto-tuning inside an application at the loop level by effectively merging just-in-time compilation with the traditional feedback-directed optimization loop. Using this method, performance gains can be realized within a single

Department of Computer Science, University of Maryland, USA

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

execution of the target application, eliminating the need for training runs.

As stated earlier, existing auto-tuning frameworks such as Active Harmony (Ţăpuş et al., 2002), POET (Yi et al., 2007), and Orio (Hartono et al., 2009) rely on the user to define a parameter space for auto-tuning. Their contribution involves automating the search for optimal configurations, not automating the use of the frameworks themselves.

Bui et al. (2009) developed a component-based environment for automated performance experiments which comes the closest to achieving our vision. They broaden the definition of "automatic" to refer to the connections between the phases of a given experiment, such as initialization, data collection, and performance analysis. While greatly simplified, the configuration of their environment is still a manual process, as it was not their intention to produce an end-to-end solution.

3. Approach

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 sections of the target application that are affected by the chosen parameters. To simplify our system, we eliminate the second task by focusing on tuning applications as a whole. Different command-line parameters are tested, and performance values are measured from a complete execution of the target application.

With this simplification, our problem is reduced to finding tunable parameters. Effectively, we wish to automatically determine which internal program variables affect performance without affecting correctness. However, this determination is difficult for third parties after the program has been written. The key observation is that the original developers had the knowledge we seek. Presumably, they were familiar enough with the problem domain to make this determination. Our task would be greatly simplified had they been philanthropic enough to annotate the source with such information.

Since it is unreasonable to expect all applications to be developed with auto-tuning in mind, this information must be gathered through alternate means. Fortunately, we need not abandon the original author as a resource. If the application was written in a modern programming language such as Chapel (Callahan et al., 2004), we can view certain keywords as a natural way to annotate the intended use of internal program variables.

4. Chapel

Chapel is a parallel programming language from Cray, Inc., developed as part of 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 tedious 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 leverage: its constructs for parallelism and its notion of configuration variables.

4.1. 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 that Chapel intends for its parallelism. 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 such as Qthreads (Wheeler et al., 2008) from Sandia National Laboratories. This allows task generation to be guided by parallelism intrinsic to the problem domain (Wheeler et al., 2011), 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 this question in Section 6.3.

4.2. Configuration variables

Chapel also provides the notion of configuration variables. Semantically, these are identical to normal variables, except that additional code is generated to override their values at load time via the command line. This seemingly simple feature has deep implications for auto-tuning. Configuration variables are designed to handle being changed from one execution to the next, which increases the suitability of these variables as auto-tuning parameters. For instance, scientific applications may find it beneficial to specify the amount of floating-point precision as a configuration variable (Buttari et al., 2008). Unfortunately, these variables may also alter the program's correctness. These constitute false positives and prevent us from using this class of variable carte blanche, and thus still require human verification. However, they certainly represent a reasonable place to begin the search for candidate parameters. Additional possibilities for bypassing manual verification are discussed in Section 8.

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 TCP socket buffer sizes as an example. The determination of an optimal value is based on several factors that are not available at development time, such as interface bandwidth, free system memory, and current network conditions (Semke et al., 1998). 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.

5. Active Harmony

The Active Harmony framework (Ţăpuş et al., 2002) has been used to auto-tune a wide range of applications at varying levels of granularity. At the heart of this framework are gradient-free simplex search methods such as Parallel Rank Ordering (Tiwari et al., 2009) and Nelder–Mead (Nelder and Mead, 1965).

Several additions and modifications were required to adapt Active Harmony for use with Chapel's configuration variables. The most prominent of which was the development of Tuna.

5.1. Tuna: The command-line tuner

Active Harmony provides a client API to flexibly support the incorporation of auto-tuning within a target application at different granularities. However, such flexibility requires modifications to a target's source code, ultimately increasing the burden of adoption. Tuna was written to alleviate this burden.

Tuna is a general tool to facilitate command-line parameter tuning. After a user specifies a parameter space and a target application, Tuna has everything it needs to establish an auto-tuning loop. Successive loop iterations use Active Harmony's client API to retrieve testing values, execute the target application using these values as arguments, and report a performance value back to the framework after each execution. Note that Tuna users require no prior knowledge of the client API; those details are managed internally.

Tuna provides three built-in methods for measuring the performance of a target application. These include wall-time, user-time, and system-time used by the target application. To support a wider range of possible performance metrics, a fourth method monitors output from the target application and parses a floating-point value from its final line as the performance value. This allows virtually any measure of performance, so long as it can be collected by an external wrapper program and printed.

The usage example in Figure 1 defines a parameter space with two integer variables via the -i flag. The first variable, named "*tile*," is permitted to be between 1 and 10, inclusive. The second variable, named "*unroll*," is permitted to be even numbers between 2 and 12, inclusive. The tuning loop is limited to at most 25 iterations due to the optional -n flag. The remaining parameters specify that the target application (matrix_mult) should be launched with Harmony-chosen values for "*tile*" and "*unroll*" as the second and fourth arguments, respectively. Wall-time of each execution will be measured and reported, as it is the default performance metric.

```
> ./tuna -n=25 \
         -i=tile,1,10,1 -i=unroll,2,12,2 \
         matrix_mult -t %tile -u %unroll
```

Figure 1. Tuna usage example.

Tuna is 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 arguments to control various details of its compilation process (Free Software Foundation, 2013). Finding optimal values for these arguments is a natural task for Tuna. A user need only specify which arguments are relevant for their optimization task and a method to measure the resulting performance.

6. Tuning Chapel applications

Chapel applications are at a particular advantage with regard to configuration variable auto-tuning. As alluded to in Section 4.1, 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 dependent on application factors such as the amount of synchronous code, as well as system factors such as thread scheduling. To test our hypothesis, we use Tuna to perform the same tuning tasks on four different machines. As detailed in Table 1, each machine represents a different CPU type, core count, or operating system.

6.1. Quicksort

We begin our tuning experiments with a quicksort example provided in the source distribution of Chapel. As shown in Figure 2, the configuration variables accepted by this

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

```
> ./quicksort --help | tail -5
quicksort config vars:
                          n: int(64)
                     thresh: int(64)
                    verbose: int(64)
                     timing: bool
```

Figure 2. Listing of quicksort's config variables.

application can be retrieved via the "–help" switch. This help routine is automatically added by the Chapel compiler.

Since the "thresh" configuration variable's use is not immediately clear, we begin a focused investigation of the source code that quickly reveals its purpose. This particular implementation of quicksort uses the "thresh" configuration variable to control the maximum recursive depth before a serial bubble sort is used as the base case.

Note that a full investigation of this application's source code was not necessary. We were able to greatly limit such manual analysis by simply viewing the configuration variables of the program.

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 their results are displayed along with the tuning data. Graphs of the resulting tuning sessions are shown in Figure 3. Minimum and maximum performance values for the default configuration are represented by dashed lines, and the average is represented by the dotted line between them.

Active Harmony was able to find configurations that improve the performance of quicksort by more than 400% over the best performing control test on the Xeon machine. 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 algorithm conducive to parallelism performs better in the presence of more parallelism. On the other hand, a better default threshold value may not exist. Chapel certainly provides a mechanism for retrieving the number of local system cores, and the example implementation could easily be rewritten to take advantage of this information. However, this may not be a desirable default value if the system is heavily burdened with other processes.

The Power architecture result also deserves investigation. As the only machine with a single core, it should be heavily affected by the spawning of additional threads. Instead, its execution times are completely unwavering. We have verified that the number of spawned threads is indeed correlated to the number of threads requested, so the operating system must be responsible for the invariable performance behavior.

Overall, Active Harmony is able to find acceptable configurations on each of our test systems within four search steps. The search could be further improved had the search space been tailored to each target architecture. For instance, the search space we defined for this experiment includes the nonparallel case where "thread" is set to one. As a methodology, we wished to keep the search as wide as possible to capture the generality of Active Harmony's simplex-based 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. This is comes very close to our goal of treating target applications as black boxes. Details of the best configurations discovered are provided in Table 2.

6.2. HPC Challenge: STREAM

Moving beyond toy programs, the source distribution of Chapel version 1.6 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 program correctness. 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 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 Section 6.3. We compare our tuning run against 10 runs of STREAM using its default values and plot their minimum, maximum, and average values on the graph using dotted and dashed lines. Graphs of the resulting tuning sessions are shown in Figure 4.

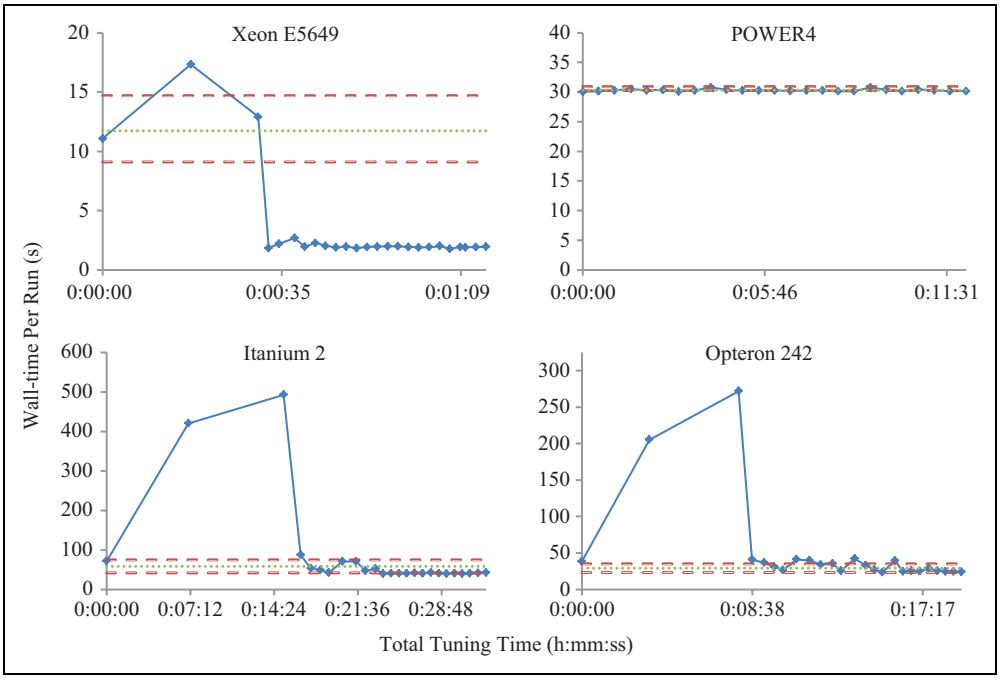

Figure 3. Tuning quicksort across multiple platforms. Solid blue line represents performance results discovered over time. Dashed and dotted lines represent the min, max, and average of 10 control runs.

Table 2. Comparison of best quicksort configurations.

Platform	Xeon E5649	POWER4	Itanium 2	Opteron 242
Best default	Time: 9.1s	Time: 30.2s	Time: 42.3s	Time: 23.3s
Best Active Harmony	Thread: 16	Thread: 1	Thread: 3	Thread: 2
	Task: 241	Task: 1	Task: 67	Task: 93
	Thresh: 16	Thresh: 1	Thresh: 12	Thresh: 10
	Time: 1.8s	Time: 30.1s	Time: 39.4s	Time: 23.2s

Active Harmony had greater difficulty finding configurations that improve upon the performance of this benchmark. This is unsurprising since STREAM is designed to test performance and care was presumably taken to achieve maximal performance using default parameter values. Details of the search results are provided in Table 3. Considering that benchmark codes are effectively a worse-case scenario, Active Harmony still performs exceedingly well as optimal values are always found within 10 search steps. Note that, while poor configurations may be tested after optimal values are found, knowledge of the best configuration is retained. This is certainly the case for our tests on the Opteron architecture.

6.3. Towards performance unit tests

To shed some light on the question of optimal thread/task ratio detailed in Section 4.1, we developed a simple parallel cache memory stress test in Chapel designed to be sensitive to cache size and access patterns. The test allocates an array of word-size integers and instructs multiple tasks to process the array by reading the value at $i - 1$, and writing an incremented version of that value into position i. Internal Chapel logic will determine array stride and chunk size as a function of task count and loop granularity, respectively. 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 value is left at Chapel's default. Graphs of the resulting tuning sessions are shown in Figure 5. As with the previous figures, these hand calculated optimal values are represented via dashed lines, with the average as the dotted line between them.

Table 4 shows that we were able to match hand-calculated optimal results within search steps on all tested platforms. Of particular interest is the Power architecture, where we achieve a performance improvement. In all prior experiments on this platform, performance results have remained steady and unaffected by configuration variable modification. Deeper analysis will be required to fully understand the how parallelism effects the cache efficiency of Chapel applications. However, that study is beyond the scope of this paper.

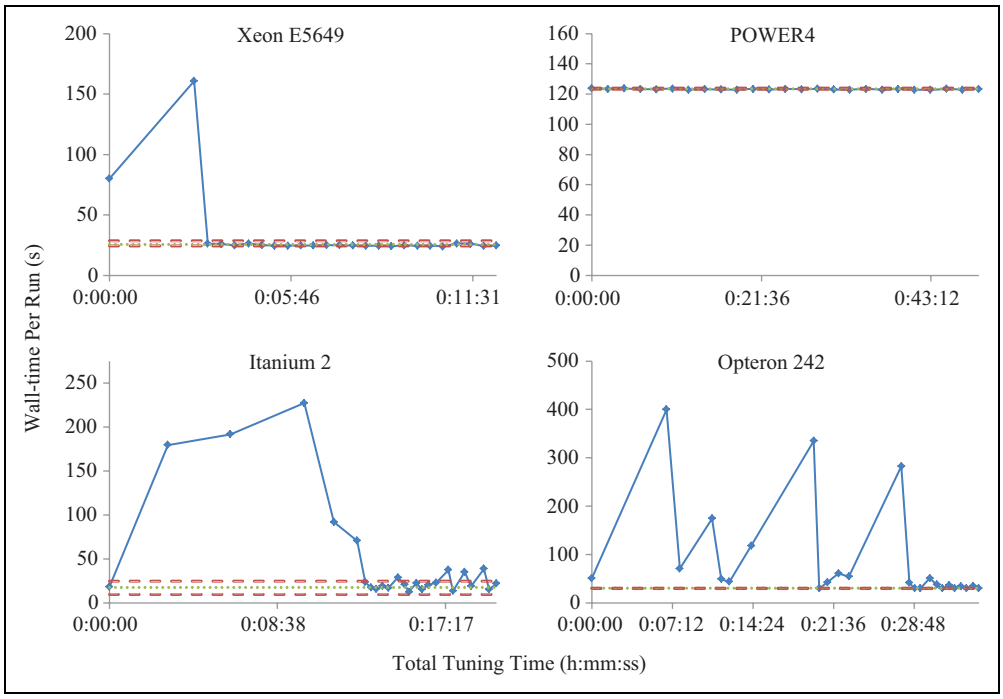

Figure 4. Tuning STREAM across multiple platforms. Solid blue line represents performance results discovered over time. Dashed and dotted lines represent the min, max, and average of 10 control runs.

Table 3. Comparison of best STREAM configurations.

Platform	Xeon E5649	POWER4	Itanium 2	Opteron 242
Best default	Time: 24.4s	Time: 123.3s	Time: 9.5s	Time: 29.4s
Best Active Harmony	Thread: 12	Thread: 16	Thread: 1	Thread: 2
	Task: 256	Task: 167	Task: 2	Task: 125
	Gran: 30.75k	Gran: 27.75k	Gran: 0.25k	Gran: 14.5k
	Time: 24.0s	Time: 122.9s	Time: 12.5s	Time: 29.7s

7. 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 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 or data types were required; we used the existing "in" keyword and Range variable type. The range may be specified directly after the variable name, as shown in Table 5. In our implementation, these ranges are correctly parsed and stored by the Chapel compiler.

Additional code is automatically generated and run at program launch time to verify each configuration variable. For instance, num4 and num7 in Table 5 provide invalid default values. This is because ranges begin their stride with the first number in the definition unless explicitly aligned via the "align" keyword. These lines will compile successfully, but 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.

As mentioned in Section 4.2, false positives must be dealt with in a meaningful way. A simple solution would be to add a new keyword such as "tunable" that is functionally equivalent to configuration variables, but syntactically imply that these variables may be modified without effecting correctness.

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.

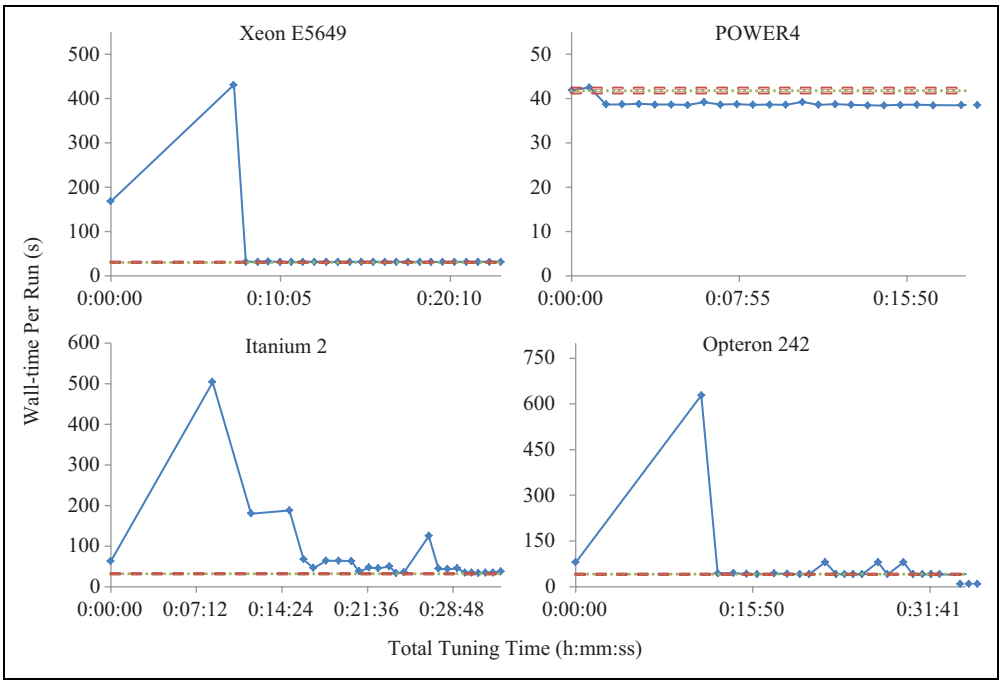

Figure 5. Optimal parameter search across multiple platforms. Solid blue line represents performance results discovered over time. Dashed and dotted lines represent the min, max, and average of 10 control runs.

Table 4. Comparison of best cache configurations.

Platform	Xeon E5649	POWER4	Itanium 2	Opteron 242
Best manual	Thread: 12 Gran: 4k Time: 30.2s	Thread: 1 Gran: 8k Time: 41.3s	Thread: 2 Gran: 2k Time: 32.0s	Thread: 2 Gran: 16k Time: 39.6s
Best Active Harmony	Thread: 16 Task: 256 Gran: 27k Time: 31.2s	Thread: 16 Task: 256 Gran: 25k Time: 38.4s	Thread: 3 Task: 32 Gran: 0.25k Time: 33.1s	Thread: 2 Task: 256 Gran: 16k Time: 40.1s

Table 5. Examples of proposed Chapel syntax for range association.

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

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

With these proposed language changes in place, the manual process of parameter verification can finally be eliminated. Tuna can be extended to extract range information exclusively from "tunable" variables and perform a tuning experiment without any manual guidance.

8. Future directions

Within Chapel, we can widen the scope of this work by lifting our simplifying assumption from Section3 and increasing the granularity beyond entire programs. Specifically, the "forall" keyword is another implicit annotation by the original author to mark loops as parallelizable. This parallelism can be used to quickly search for optimal loop transformations similar to work done by Tiwari and Hollingsworth (2011). This sort of tuning could also be applied automatically after a simple static analysis of the source code reveals the location of these loops.

Outside of Chapel, adapting similar techniques to existing languages such as C or Fortran will be difficult, since

they 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 shared libraries can also increase the utility of fully automatic auto-tuning users. For example, the MPI 3.0 standard makes provisions for control variables MPI Forum (2012), which are conceptually equivalent to Chapel's configuration variables.

9. 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 two different Chapel applications, and were successfully able to tune these programs without modification, and with little or no familiarity with their source code.

The Active Harmony framework was used and extended to support this work, introducing Tuna as a general tool for command-line auto-tuning.

Funding

Partial support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under award numbers ER25763 and ER26054. Support was also provided by the UMD Partnership with the Laboratory of Telecommunications Sciences, Contract Number H9823013D00560002.

References

Bui V, Norris B and McInnes LC (2009) An automated component-based performance experiment environment. In: *Proceedings of the 2009 workshop on component-based high performance computing (CBHPC 2009)*. Also available as Preprint ANL/MCS-P1666-0809.

Buttari A, Dongarra J, Kurzak J, Luszczek P and Tomov S (2008) Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Transactions of Mathematical Software* 34(4): 17:1–17:22. doi:10.1145/1377596.1377597.

Callahan D, Chamberlain BL and Zima HP (2004) The cascade high productivity language. In: *Ninth international workshop on high-level parallel programming models and supportive environments (HIPS '04)*, pp. 52–60.

Chung IH and Hollingsworth JK (2004) Automated cluster-based web service performance tuning. In: *Proceedings of the 13th IEEE international symposium on high performance distributed computing (HPDC '04)*. Washington, DC: IEEE Computer Society, pp. 36–44. doi:10.1109/HPDC.2004.4.

Free Software Foundation (2013) Optimize Options – Using the Gnu Compiler Collection (GCC). Available at: http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

Hartono A, Norris B and Sadayappan P (2009) Annotation-based empirical performance tuning using Orio. In: *IEEE international symposium on parallel distributed processing (IPDPS 2009)*, pp. 1–11. doi:10.1109/IPDPS.2009.5161004.

MPI Forum (2012) MPI: A message-passing interface standard (version 3.0). Section 14.3.6, pp. 567–572. Available at: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

Nelder JA and Mead R (1965) A simplex method for function minimization. *The Computer Journal* 7(4): 308–313. doi:10.1093/comjnl/7.4.308.

Rahman SF, Guo J and Yi Q (2011) Automated empirical tuning of scientific codes for performance and power consumption. In: *Proceedings of the 6th international conference on high performance and embedded architectures and compilers (HiPEAC '11)*. New York: ACM Press, pp. 107–116. doi:10.1145/1944862.1944880.

Semke J, Mahdavi J and Mathis M (1998) Automatic tcp buffer tuning. In: *Proceedings of the ACM SIGCOMM '98 conference on applications, technologies, architectures, and protocols for computer communication (SIGCOMM '98)*. New York: ACM Press, pp. 315–323. doi:10.1145/285237.285292.

Shin J, Hall MW, Chame J, Chen C, Fischer PF and Hovland PD (2010) Speeding up nek5000 with autotuning and specialization. In: *Proceedings of the 24th ACM international conference on supercomputing (ICS '10)*. New York: ACM Press, pp. 253–262. doi:10.1145/1810085.1810120 .

Ţăpuş C, Chung IH and Hollingsworth JK (2002) Active Harmony: towards automated performance tuning. In: *Proceedings of the 2002 ACM/IEEE conference on supercomputing (Supercomputing '02)*. Los Alamitos, CA: IEEE Computer Society Press, pp. 1–11.

Tiwari A and Hollingsworth JK (2011) Online adaptive code generation and tuning. In: *Proceedings of the 2011 IEEE International parallel & distributed processing symposium (IPDPS '11)*. Washington, DC: IEEE Computer Society, pp. 879–892. doi:10.1109/IPDPS.2011.86.

Tiwari A, Tabatabaee V and Hollingsworth JK (2009) Tuning parallel applications in parallel. *Parallel Computing* 35(8–9): 475–492. doi:10.1016/j.parco.2009.07.001.

Wheeler K, Murphy R and Thain D (2008) Qthreads: An API for programming with millions of lightweight threads. In: *IEEE International symposium on parallel and distributed processing (IPDPS 2008)*, pp. 1–8. doi:10.1109/IPDPS.2008.4536359.

Wheeler KB, Murphy RC, Stark D and Chamberlain BL (2011) The Chapel tasking layer over Qthreads. In: *Proceedings of CUG 2011*.

Yi Q, Seymour K, You H, Vuduc R and Quinlan D (2007) POET: Parameterized optimizations for empirical tuning. In: *IEEE International symposium on parallel and distributed processing (IPDPS 2007)*, pp. 1–8. doi:10.1109/IPDPS.2007.370637.

Author biographies

Ray S Chen is a graduate student at the University of Maryland's Department of Computer Science. His research interests include dynamic binary instrumentation and automated performance tuning. He is advised by Dr. Jeffrey K. Hollingsworth.

Jeffrey K Hollingsworth is a Professor in the Computer Science Department at the University of Maryland, College Park, and affiliated with the Department of Electrical Engineering and the University of Maryland Institute for Advanced Computer Studies. His research interests include instrumentation and measurement tools, resource aware computing, high performance distributed computing, and programmer productivity. Dr. Hollingsworth's current projects include the Dyninst runtime binary editing tool, and Harmony - a system for building adaptable, resource-aware programs. He received his PhD and MS degrees in computer science from the University of Wisconsin. He earned his B.S. in Electrical Engineering from the University of California at Berkeley. Dr. Hollingsworth is a senior member of IEEE and a member of ACM.