Towards the Co-evolution of Auto-tuning and Parallel Languages

Ray Chen, Jeff Hollingsworth (advisor)
Department of Computer Science
University of Maryland

Introduction

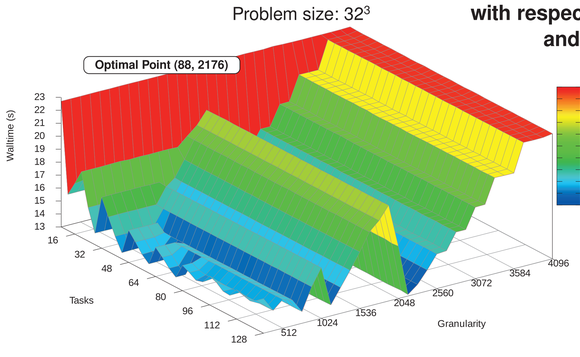
There exists a gap between the massive parallelism available from today's HPC systems and our ability to efficiently develop applications that utilize such parallelism. Emerging parallel programming languages such as Chapel aim to narrow this gap through high-level abstractions of data and control parallelism. However, optimizing such programs requires the compiler to make decisions based on information unavailable at compile time. Consider thread and chunk size, as seen in the figures below.

By evolving the Active Harmony auto-tuning framework (detailed on the left) along with the Chapel parallel programming language (detailed on the right), such decisions could be deferred until run time when it's possible to test for optimal values.

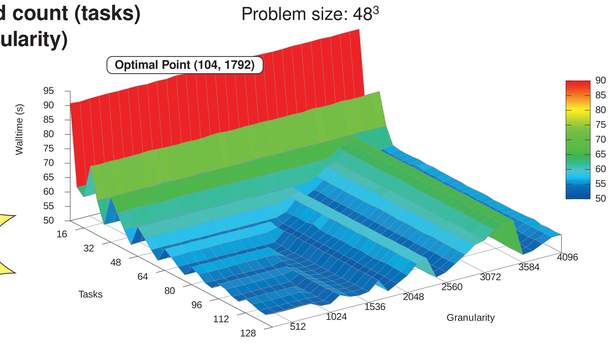
What is the optimal thread count and data chunk size for a given application?

Performance of single-node LULESH [1] written in Chapel with respect to light-weight thread count (tasks) and data chunk size (granularity)

Problem size: 32^3



Problem size: 48^3



Optimal task/granularity depends on the problem size!

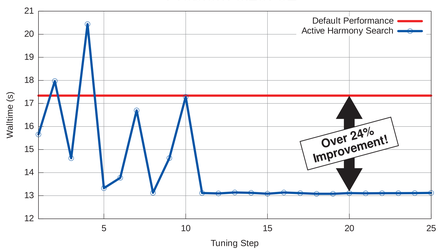
See what happens if one optimal point is used on the other problem.

All experiments run using an 8 core (16 thread) Intel Sandy Bridge Xeon E5-2670 with 32 GB of RAM

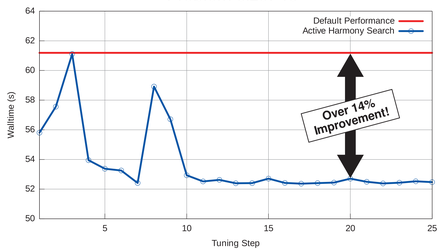
Can auto-tuning be used to mitigate these performance issues?

Using Active Harmony to search the task/granularity parameter space

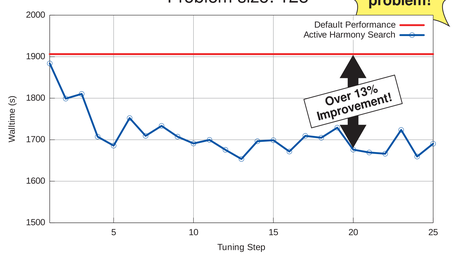
Problem size: 32^3



Problem size: 48^3



Problem size: 128^3



Similar gains from a much larger problem!

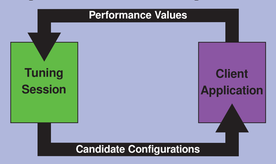
Conclusion

These experiments demonstrate the utility of auto-tuner and parallel language co-evolution. Looking ahead, the next step in the evolution involves auto-tuner detection of tunable variables within a program, if defined.

We believe wider support for auto-tuning constructs within parallel languages will enable HPC programmers to productively focus on issues of logic rather than performance. Moreover, tighter integration with auto-tuning can enable performance gains within a single execution [2], bringing parallel languages that much closer to closing the performance gap.

Current Status of Active Harmony

Like all empirical auto-tuning frameworks, Active Harmony uses a feedback loop where candidate configurations are sent to a client application, and performance values are reported back.



The tuning session considers performance values as they are reported from the client application, and uses this information to generate future candidate configurations. This pattern continues until the tuning session considers the search converged.

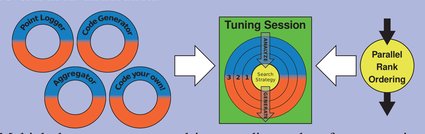
Modularizing Active Harmony

To ease the adoption of our auto-tuning framework, we created a set of abstractions for concepts common to every tuning session. One such abstraction is the search strategy, responsible for generating new configurations based on prior performance data. Active Harmony provides a library of four to choose from.

A new session may choose any one of these to drive the search. Have an idea for a better search strategy? Active Harmony also provides an API so you can write your own.

Extending Active Harmony

To extend the flexibility of our framework, the feedback loop can be altered by loading processing layers. Each layer handles two events: one before the search strategy analyzes a performance report (blue side), and one after a configuration has been generated (orange side). Like strategies, Active Harmony provides several in its distribution.



Multiple layers are processed in ascending order after generation, and in reverse order before analysis. These abstractions allow virtually any auto-tuner to be built from parts.

Current Status of Chapel

Chapel provides a special type of variable called "Config Variables," which are declared with the **`config`** keyword. The value of these variables can be overridden at launch time via command line parameters.

```
> cat prog.chpl
config var num = 10;
writeln("num is ", num);
```

Yes, this is a complete Chapel program.

```
> chpl prog.chpl -o prog
> ./prog
num is 10
> ./prog --num=15
num is 15
```

Several config variables are built into every program Chapel compiles. We use two built-in config variables for our experiment. **`dataParTasksPerLocale`** controls the number of tasks (lightweight threads) to use over data-parallel loops, and **`dataParMinGranularity`** controls how much data each task will process at a time (data chunk size).

Augmenting Config Variables

From an auto-tuning perspective, the ability to override the value of variables is not sufficient. A valid value range is also necessary to prevent empirical trials with invalid parameters. To this end, we modified the Chapel parser to accept an optional range within config variable definitions.

```
config var regular_var = 5;
config var bounded_var in 1..100 = 5;
config var strided_var in 1..100 by 2 = 5;
config var aligned_var in 1..100 by 2 align 2 = 6;
```

Examples of augmented config variables. Proposed syntax is optional and highlighted in yellow.

Associating a range with config variables has utility outside the realm of auto-tuning as well. Since config variables effectively represent user input, the additional range is a simple way to provide bounds-checking automatically.

Adding Chapel Keywords

Finally, there are many uses of config variables that are invalid for auto-tuning. For example, it's useful for the LULESH benchmark to define the problem size as a config variable. Auto-tuning variables must not affect the target's input-output behavior.

We propose **`tunable`** as a natural keyword that distinguishes between these use cases, allowing an auto-tuner to freely adjust the proper variables.

```
config  var regular_var = 5;
tunable var tunable_var in 1..100 = 5;
```

An example of the tunable keyword. Proposed syntax is highlighted in yellow.

We intend to submit our language changes to the developers of Chapel at Cray, Inc. for consideration. Look for these additions in a release of Chapel in the near future!

[1] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, December 2012.
[2] A. Tiwari and J. K. Hollingsworth. Online adaptive code generation and tuning. In Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, pages 879–892, Washington, DC, USA, 2011. IEEE Computer Society.

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND