Chapel:

Productive Parallel Programming

Parallel computing has resulted in numerous significant advances in science and technology over the past several decades. However, in spite of these successes, the fact remains that only a small fraction of the world's programmers are capable of effectively using the parallel programming models employed within HPC and mainstream computing. Chapel is an emerging parallel language being developed at Cray Inc. with the goal of addressing this issue and making large-scale parallel programming far more productive and generally accessible.

Chapel originated from the DARPA High Productivity Computing Systems (HPCS) program, which challenged vendors like Cray to improve the productivity of high-end computing systems. Engineers at Cray noted that the HPC community was hungry for alternative parallel programming languages and developed Chapel as part of our response. The reaction from HPC users so far has been very encouraging—most would be excited to have the opportunity to use Chapel once it becomes production-grade.

Chapel Overview

Though it would be impossible to give a thorough introduction to Chapel in the space of this article, the following characterizations of the language should serve to give an idea of what we are pursuing:

General Parallelism: Chapel has the goal of supporting any parallel algorithm you can conceive of on any parallel hardware you want to target. In particular, you should never hit a point where you think "Well, that was fun while it lasted, but now that I want to do *x*, I'd better go back to MPI."

Separation of Parallelism and Locality: Chapel supports distinct concepts for describing parallelism ("These things should run concurrently") versus locality ("This should be placed here; that should be placed over there"). This is in sharp contrast to conventional approaches that either conflate the two concepts or ignore locality altogether.

Multiresolution Design: Chapel is designed to support programming at higher or lower levels, as required by the programmer. Moreover, higher-level features—like data distributions or parallel loop schedules—may be specified by advanced programmers within the language.

Productivity Features: In addition to all of its features designed for supercomputers, Chapel also includes a number of sequential language features designed for productive programming. Examples include type inference, iterator functions, object-oriented programming, and a rich set of array types. The result combines productivity features as in Python™, Matlab®, or Java™ with optimization opportunities as in Fortran or C.

Chapel's implementation is also worth characterizing:

Open Source: Since its outset, Chapel has been developed in an open-source manner, with collaboration from academics, computing labs, and industry. Chapel is released under a BSD license in order to minimize barriers to its use.

Portable: While Cray machines are an obvious target for Chapel, the language was designed to be very portable. Today, Chapel runs on virtually any architecture supporting a C compiler, UNIX-like environment, POSIX threads, and MPI or UDP.

Optimized for Crays: Though designed for portability, the Chapel implementation has also been optimized to take advantage of Cray-specific features.

Chapel: Today and Tomorrow

While the HPCS project that spawned Chapel concluded successfully at the end of 2012, the Chapel project remains active, growing, and ongoing. The Chapel prototype and demonstrations developed under HPCS were considered compelling enough to users that Cray plans to continue the project over the next several years. Current priorities include:

Performance Optimizations: Under HPCS, the implementation effort focused primarily on correctness over performance. Improving performance is typically considered the number one priority for growing the Chapel community.

Support for Accelerators: Modern compute nodes are increasingly likely to contain accelerators like GPUs or Intel® MIC chips. We are currently working on extending our locality abstractions to better handle such architectures.

Interoperability: Beefing up Chapel's current interoperability features is a priority, to permit users to reuse existing libraries or gradually transition applications to Chapel.

Feature Improvements: Having completed HPCS, we now have the opportunity to go back and refine features that have not received sufficient attention to date. In many cases, these improvements have been motivated by feedback from early users.

Outreach and Evangelism: While improving Chapel, we are seeking ways to grow Chapel's user base, particularly outside of the traditional HPC sphere.

Research Efforts: In addition to hardening the implementation, a number of interesting research directions remain for Chapel. These include resilience mechanisms, applicability to "big data" computations, energy-aware computing, and support for domain-specific languages.

For More Information

For more information about Chapel, the best introduction is *A Brief Overview of Chapel*, which provides a concise summary of Chapel's history, motivating themes, and features. This paper, along with other tutorials, presentations, and papers, can be found on the project website at **http://chapel.cray.com**. To download Chapel or join various mailing lists, visit our SourceForge page at **http://sourceforge.net/projects/chapel**.

***Chapel: Productive Parallel Programming* was written by Brad Chamberlain, Principal Engineer at Cray. It was originally published on the Cray blog at http://blog.cray.com.**