The Exascale Programming Challenge and Chapel's Response

Bradford L. Chamberlain, Cray Inc.

As the scientific user community looks ahead to the exascale era of HPC architectures, it faces several challenges, many of which pertain to our collective uncertainty around what these architectures will be like. For the first time in decades, the HPC programming community is dealing with a significant shift in the abstract machine models that we think about when targeting HPC hardware. This shift can be seen currently in the use of GPUs and on the horizon in the form of Intel Phi or AMD Fusion, yet without a great deal of clarity as to what the machine model will be like in the steady state, or whether we'll need a spectrum of evolving abstract machine models from one vendor or machine model to the next.

In the face of this uncertainty, there is a certain amount of agreement about some of the general architectural trends that users will be facing, including:

- increased sensitivity to locality within the compute node;
- the likelihood of distinct memory or processor types within the system;
- reduced memory bandwidth per flop, motivating the development of algorithms with increased computational intensity.

All of these trends pose problems for our current parallel programming models, primarily due to the degree to which they embed architectural assumptions within their designs:

- MPI's process-oriented model of parallelism is almost certainly too heavyweight for such architectures without the use of some other technology to manage the parallelism within the compute node. Moreover, "MPI-everywhere" can become increasingly untenable for many computations as the number of cores per node grows, due to the surface-to-volume ratios resulting from over-decomposition.
- The static SPMD execution models of UPC and CAF are similarly heavyweight and static given that the hardware is becoming increasingly fine-grain and offload-oriented.
- Meanwhile our traditional shared memory models have traditionally been locality-oblivious, requiring changes and new features to adapt to the exascale era (e.g., OpenMP 4.0) and/or new dialects of shared memory programming that reflect the target architecture more directly (CUDA, OpenACC, OpenCL).

While it's certain that some hybrid use of conventional programming models is likely to endure going forward (commonly referred to as "MPI+X"), such models are unfortunate since, by definition, they use completely distinct concepts and abstractions to express the overarching concerns in HPC programming: parallelism and locality.

In addition to bringing challenges, the transition to exascale is also a potential opportunity for positive change. For example, it can be a chance for the HPC community to consider new languages that do a better job of insulating computations from underlying hardware details than past approaches have. Rather than sprinting to the simplest low-level solution, perhaps it's time to apply the effort, will, and investment to develop a longer-term solution.

Chapel strives to be just such a language: One of our main goals is to permit computational scientists to express the parallelism and locality in their parallel algorithms using high-level concepts that abstract away the details of mapping to the hardware. At the same time, we've designed Chapel so that a parallel-savvy programmers can tune the iterators, data structures, and architectural mappings to a given parallel system without unduly intruding upon the expression of the scientific computation itself.

In the past few years, we have demonstrated these principles within the context of DOE proxy applications ported to Chapel. For example, a Chapel implementation of the LULESH proxy application developed by LLNL supports the ability for a user to make data structure decisions such as "structured vs. unstructured," "local vs. distributed," and "sparse vs. dense" at compile-time via the declarations of the computation's key *domains* (index sets). From there, the physics of the computation is written in a manner that is independent of such factors, demonstrating that the parallel algorithms can successfully be isolated from details of mapping the data structures to the architecture.

In more recent work, we have been expanding Chapel's locality-oriented abstractions so that advanced end-users can define, and map computation to, to architectural substructures within a compute node, such as NUMA domains, GPUs, and/or distinct memory/processor types. As with previous work on iterator and array abstractions, our goal is to permit advanced users to control such details via layered abstractions that insulate the application code from their decisions.

In the interest of full disclosure, in its current state, Chapel is not a sufficiently stable solution for most mission-critical applications. Further optimizations need to be implemented to make performance competitive with conventional languages and programming models; emerging features like the *hierarchical locales* of the previous paragraph need to be brought to maturity; and other features need to be hardened or improved. We have recently embarked on a five-year push to improve Chapel from its current prototypical state to that of a more product-grade implementation.

That said, in its current form, Chapel is sufficiently stable to be ready for experimentation and feedback. We encourage interested users to write proxy computations or key idioms from their applications in Chapel and to report on their successes, failures, and frustrations to the Chapel team. Such feedback has already improved the language immeasurably and helped us develop a more useful language for early adopters.

For those who desire new parallel languages, it's unlikely that a productive, exascale-ready solution is going to spring fully-formed from anyone's forehead. To that end, users will have to decide whether to give up on ever using anything other than MPI+X; or to join the masses who spend their lives whining about the lack of good parallel languages and the (admittedly) long odds of ever getting one adopted; or they can champion and work with a language that they find promising, doing what they can to improve it and help it improve.

A growing number of users are indicating that they think Chapel is that language for them—that the next (first?) truly successful parallel HPC language will either be Chapel, or something Chapel-like that hasn't been developed yet. If, in investigating Chapel, you find yourself of a similar mindset, we hope for the opportunity to work with you in the future.

For further information:

- a blog article providing an overview of Chapel: http://blog.cray.com/?p=5889
- an presentation introducing Chapel: http://chapel.cray.com/presentations/ChapelForATPESC-forweb.pdf
- an book chapter introducing Chapel: http://chapel.cray.com/papers/BriefOverviewChapel.pdf
- a presentation on the proxy app work mentioned above:
  http://chapel.cray.com/presentations/ChapelForSIAMPP14-MiniMD-presented.pdf
- papers on user-defined parallel iterators and arrays in Chapel:
  User-Defined Parallel Zippered iterators in Chapel [slides]
  Authoring User-Defined Domain Maps in Chapel [slides]
- the Chapel website: http://chapel.cray.com