High Performance Languages

Original Charter

After a long era of only one new dominant programming language per decade (FORTRAN in the 60s, C in the 70s, C++ in the 80s, and Java in the 90s), there is a resurgence of new languages. Languages such as Go, Swift, Hack, Rust, Julia, Scala and Dart are only a few years old, but are already used by thousands of programmers.

Another promising trend is domain specific languages (DSLs). How will these trends affect high performance computing? Can high performance language(s) solve many of the productivity and performance portability issues burdening high performance computing? How can we make this a reality?

Risk Factors in Adopting a New Language

(compared to adopting a library)

- Completely new syntax, semantics, need to be learned
  - Embedded DSLs are not that different from libraries (extended libraries)
  - Marketing issues - DSL has the dreaded ‘L’. Rename to smart libraries (Milind is right!)
- New (i.e. untested/buggy) compiler
- Build and link issues (need to be interoperable with the rest of the code)
- Debugger (is there debugging support?)
- Interoperability between languages
  - Is there a ABI or native interface
  - Can data be exchanged (managed vs. direct)
  - Resource sharing (will multiple components coexist?)
    - In the single threaded world, libraries time-multiplexed, i.e. easily interoperable. But in complex parallel environments, this advantage is gone
- Support
  - How close is the collaboration with the technology developers
- Risk in case of an early demise?
  - A black box, if it breaks, can make the entire system useless
  - Source-to-source can partially mitigate (i.e. will work, perhaps slowly, without the technology)
  - Accessibility -- if the app. person is comfortable that they can fix a problem (libraries have an advantage over languages)

Rewrite vs. Incremental Adoption

Benefit of complete rewrite may be greater, e.g. LLNL effort to transition from vector to MPP; out of ten app teams, four chose a complete rewrite, and others chose incremental adoption to preserve existing approaches. All the ‘rewrite’ applications demonstrated much greater performance improvements than the ‘incremental’ applications. (ref. *Industrial Strength Parallel Computing*).

Today we face similar uncertainty over major technology changes.

Evaluating Proposed Technologies

- Can the software technology provider demonstrate that they have an application partner for whom the risk/reward proposition is attractive? Having closer interaction between language developers and application users is a must.
- Are there quality standards that language developers can work to meet to ensure adoption by labs and other users?
- Support for infrastructure to do the exploration and development of language / compiler / runtime?
- How to develop confidence in quality of the model and implementation?
- How can features of new languages be incorporated into existing languages/standards?

How to lower (real/perceived) barrier to entry?

Hackathons: 2-3 day efforts to try out new approach on a real-world problem

National center for evaluation: longer-running projects to evaluate approaches on large, real-world applications

Embedding graduate students with application teams

Funding for ‘micro-interactions’: ~3 month mini-project for technology stakeholder (language researchers) to collaborate with science stakeholder (application developers).
Outcomes: learning about application/technology requirements; integration of code into application; performance evaluation; proposal for future collaboration (or reasons why not!)