Creating an Open-Source Language: From Research Prototype to Production

Brad Chamberlain
UW CSE 403
March 6, 2019
Who am I?

Education:
• Earned Ph.D. from University of Washington CSE in 2001
  • focused on the ZPL data-parallel array language
• Remain associated with UW CSE as an Affiliate Professor

Industry:
• Currently a Principal Engineer at Cray Inc.
• Technical lead / founding member of the Chapel project
• Also spent a year at a startup: QuickSilver Technology
Piz Daint: One of Today’s Most Powerful Supercomputers

https://www.cscs.ch/computers/piz-daint/
### Model Cray XC40/Cray XC50

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Hybrid Compute Nodes</td>
<td>5,704</td>
</tr>
<tr>
<td>Number of Multicore Compute Nodes</td>
<td>1,431</td>
</tr>
<tr>
<td>Peak Floating-point Performance per Hybrid Node</td>
<td>4.761 Teraflops Intel Xeon E5-2690 v3/Nvidia Tesla P100</td>
</tr>
<tr>
<td>Peak Floating-point Performance per Multicore Node</td>
<td>1.210 Teraflops Intel Xeon E5-2695 v4</td>
</tr>
<tr>
<td>Hybrid Peak Performance</td>
<td>27.154 Petaflops</td>
</tr>
<tr>
<td>Multicore Peak Performance</td>
<td>1.731 Petaflops</td>
</tr>
<tr>
<td>Hybrid Memory Capacity per Node</td>
<td>64 GB; 16 GB CoWoS HBM2</td>
</tr>
<tr>
<td>Multicore Memory Capacity per Node</td>
<td>64 GB, 128 GB</td>
</tr>
<tr>
<td>Total System Memory</td>
<td>437.9 TB; 83.1 TB</td>
</tr>
<tr>
<td>System Interconnect</td>
<td>Cray Aries routing and communications ASIC, and Dragonfly network topology</td>
</tr>
<tr>
<td>Sonexion 3000 Storage Capacity</td>
<td>8.8 PB</td>
</tr>
<tr>
<td>Sonexion 3000 Parallel File System Theoretical Peak Performance</td>
<td>112 GB/s</td>
</tr>
<tr>
<td>Sonexion 1600 Storage Capacity</td>
<td>2.5 PB</td>
</tr>
<tr>
<td>Sonexion 1600 Parallel File System Theoretical Peak Performance</td>
<td>138 GB/s</td>
</tr>
</tbody>
</table>

[https://www.cscs.ch/computers/piz-daint/](https://www.cscs.ch/computers/piz-daint/)
Outline

✓ Who’s Brad? Cray?

➢ What’s Chapel?
  • Software Engineering & Chapel
  • Parting Thoughts
  • Chapel Resources
What is Chapel?

**Chapel:** A productive parallel programming language

- portable & scalable
- open-source & collaborative

**Goals:**

- Support general parallel programming
  - “any parallel algorithm on any parallel hardware”
- Make parallel programming at scale far more productive
Chapel and Productivity

Chapel aims to be as...

...**programmable** as Python
...**fast** as Fortran
...**scalable** as MPI, SHMEM, or UPC
...**portable** as C
...**flexible** as C++
...**fun** as [your favorite programming language]
CLBG Cross-Language Summary (Dec 18, 2018)
CLBG Cross-Language Summary (Dec 18, 2018, zoomed)

Compressed Code Size (normalized to smallest entry)

Execution Time (normalized to fastest entry)

compressed

type smaller faster

Haskell
OCaml
Pascal
Typescript
JavaScript
Racket

C# Swift
Java Fortran

C++

Rust

© 2019 Cray Inc.
CLBG Cross-Language Summary (Dec 18, 2018, zoomed)

Compressed Code Size (normalized to smallest entry)

Execution Time (normalized to fastest entry)

smaller

faster
CLBG Cross-Language Summary (Dec 18, 2018)

- Faster
- Smaller

Compressed Code Size (normalized to smallest entry)

Execution Time (normalized to fastest entry)
HPCC RA: buffering vs. network atomics

RA Performance (GUPS)

- Chapel 1.19 pre (unordered)
- Chapel 1.19 pre
- Reference (bucketing)

Locales (x 36 cores / locale)

GUPS

© 2019 Cray Inc.
/* Perform updates to main table. The scalar equivalent is:

   for (j = 0; j < NUPDATE; j++) {
      Table[Ran & (TABSIZE - 1)] ^= Ran;
      if ((status>MPI_TAG == UPDATE_TAG) {
         MPI_Isend(&LocalSendBuffer, localBufferSize, tparams.dtype64, (int)j, 
            UPDATE_TAG, MPI_COMM_WORLD, &outreq);
         pendingUpdates += pendingUpdates;
      } else if ((status>MPI_TAG == FINISHED_TAG) {
         MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64, (int)j, 
            UPDATE_TAG, MPI_COMM_WORLD, &inreq);
         pendingUpdates -= pendingUpdates;
      } /* send our done messages */
      MPI_Test(&inreq, shave_done, status);
Perform updates to main table. The scalar equivalent is:

```bash
for (i=0; i<NUPDATE; i++) {
    Ran = (Ran << 1) ^ ((s64Int) Ran < 0) ? POLY : 0;
    Table[Ran & (TABSIZE-1)] ^= Ran;
}
```

Chapel Kernel

```chapel
forall (_, r) in zip(Updates, RAStream()) do
T[r & indexMask].xor(r);
```
The Chapel Team at Cray (May 2018)

~13 full-time employees + ~2 summer interns + ~2 GSoC students
Software Engineering & Chapel
Disclaimers

• Anything I say may not translate at all to any other job / project you may take on
• I’m reporting on my group’s practices and not necessarily those of Cray broadly
• These slides are not particularly pretty…
My year at QuickSilver (between UW and Cray)

- Worked for an Extreme Programming (XP) software group
  - Nowadays, more likely to be Agile software development, Scrum, Kanban, …
- My takeaways:
  - frequent planning (sprints)
  - customer involvement / focus
  - daily standups
  - test-first development
  - pair programming always
  - brags
  - sustainable pace
My year at QuickSilver (between UW and Cray)

• Worked for an Extreme Programming (XP) software group
  • Nowadays, more likely to be Agile software development, Scrum, Kanban
• My takeaways:
  • frequent planning (sprints)
  • customer involvement / focus
  • daily standups
  • test-first development
  • pair programming always
  • brags
  • sustainable pace
A Brief History of Chapel

2003-2006: Initial Concept / Splashing Around (2-4 devs)
  • blank slate development

2006-2012: Developing a Research Prototype (6-7 devs)
  • research focus

2013-2018: Transition from Research to Production-Grade (~12 devs)
  • increased focus on users, adoption, stability

2019-2021: Striving for Adoption (12-15(?) devs)
  • lock down production use cases and users
What did we set out to create?

• Language definition
• Compiler
• Runtime (access to system-level capabilities: memory, network, threads, …)
• Standard Libraries
• Tools (minimal)
Compiling Chapel

Chapel Source Code → chpl → Chapel Executable

Standard Modules (in Chapel)
Chapel Compiler Architecture

Chapel Source Code → Chapel-to-C Compiler → Generated C Code → Standard C Compiler & Linker → Chapel Executable

- Chapel Compiler
- Chapel-to-C Compiler
- Generated C Code
- Standard C Compiler & Linker
- Chapel Executable

- Standard Modules (in Chapel)
- Internal Modules (in Chapel)
- Runtime Support Library (in C)
  - Tasks/Threads
  - Communication
  - Memory
  - ...
Chapel Compilation Architecture

Chapel Source Code → Chapel-to-C Compiler → Generated C Code → Standard C Compiler & Linker → Chapel Executable

- Standard Modules (in Chapel)
- Internal Modules (in Chapel)

Runtime Support Library (in C)
- Tasks/Threads
- Communication
- Memory
- ...
Runtime Communication Layer: Communication

Chapel Runtime Support Library (in C)

Communication

- none (single locale)
- gasnet
- ugni
- ofi

GASNet (universal)
Cray uGNI (Cray networks)
libfabric (universal)
Do you want to impose a style on developers?

• Yes?
  • How strict?

```c
if (flag) {
    foo();
}
else {
    bar();
}
else {
    bar();
}
```

```c
if (flag) {
    foo();
}
else {
    bar();
}
else {
    bar();
}
```
Do you want to impose a style on developers?

- Yes?
  - How strict?

```c
if (isConst(expr)) { ... }
if (isConst(expr) == true) { ... }
if ((isConst(expr) == true) == true) { ... }  // ???
```
Do you want to impose a style on developers?

• Yes?
  • How strict?
• No?
  • “Let each developer’s style serve as their handwriting”
  • Can use code review to squash bad habits, develop common sense of taste
What about code reviews?

- Can you afford the time to do them?
- Can you afford not to?
- What is the intention / what do you expect to gain from it?
What level of code documentation will you require?

- Comment every file / routine / variable / code block?
- Write self-documenting code?
- Document in commit / merge comments?
Do you want to develop open- or closed-source?

- (even if ultimate goal is to open-source…)
- Potential advantages to open-source:
  - leverage open-source community, developers, and code (?)
  - get immediate and continual feedback
  - “seems like the right thing to do”
- Potential disadvantages:
  - means living with your warts showing
  - no such thing as a free lunch
Source Control Management?

• “Yes” is the only reasonable answer
• Historically, Chapel has used:
  • CVS
  • SVN
  • Git
  • Git familiarity is perhaps the most valuable SW dev skill after programming
    • as well as GitHub or GitLab (hosting sites that support git repositories)
Chapel’s use of GitHub

• use GitHub issues to track bugs, feature requests, stories, tasks, epics
Chapel’s use of GitHub

- use GitHub issues to track bugs, feature requests, tasks, stories
- submit proposed changes as pull requests (PRs)
  - must be reviewed by core developer (someone who takes turns doing triage)
  - use GitHub comments/reviews to give feedback
  - if reviewer / reviewee can’t agree, escalate to the group for more opinions
Chapel’s use of GitHub

• use GitHub issues to track bugs, feature requests, tasks, stories
• submit proposed changes as pull requests (PRs)
  • must be reviewed by core developer (someone who takes turns doing triage)
  • use GitHub comments/reviews to give feedback
  • if reviewer / reviewee can’t agree, escalate to the group for more opinions
• must also pass testing
  • at minimum, a complete run of linux testing
  • optionally, other configurations as considered valuable
• Travis testing run automatically on each PR
  • key smoke tests also run post-merge
Chapel’s use of GitHub

- use GitHub issues to track bugs, feature requests, tasks, stories
- submit proposed changes as pull requests (PRs)
- use ZenHub for tracking tasks (a Kanban-style board for tasks/stories)
Chapel’s use of GitHub

• use GitHub issues to track bugs, feature requests, tasks, stories
• submit proposed changes as pull requests (PRs)
• use ZenHub for tracking tasks (a Kanban-style board for tasks / stories)
• releases hosted on GitHub as well
Testing: Our key to sanity

• homegrown system
• crawls directory structure looking for things to test
  • simplest form:
    • hello.chpl  # source file
    • hello.good  # expected output of compilation + execution steps
  • extended form:
    • additional files to specify:
      • command-line options for compiler and executable
      • actions to take before compiling, running, diffing, …
    • etc.
What do we run? (Correctness testing)

- 9500+ tests
  - x back-end compilers (gnu, clang, llvm, icc, cce, pgi)
  - x platforms (Linux, Mac OS X, Crays, Cygwin, ...)
  - x processor types (x86, arm, knl, ...)
  - x machine models (flat, numa)
  - x tasking options (fifo, qthreads)
  - x options for communication (local, gasnet, ugni, libfabric, ...)
  - x build options (quickstart, preferred, valgrind, ...)
  - x compiler options (normal, --fast, --baseline, --verify, ...)
  - x ...
What do we run? (Correctness testing)

- 9500+ tests …or 300+ (release examples only) or ~6 (“hellos”)
  - x back-end compilers (gnu, clang, llvm, icc, cce, pgi)
  - x platforms (Linux, Mac OS X, Crays, Cygwin, …)
  - x processor types (x86, arm, knl, …)
  - x machine models (flat, numa)
  - x tasking options (fifo, qthreads)
  - x options for communication (local, gasnet, ugni, libfabric, …)
  - x build options (quickstart, preferred, valgrind, …)
  - x compiler options (normal, --fast, --baseline, --verify, …)
  - x …
Testing managed through Jenkins
Testing managed through Jenkins
Also Performance Tests
Current gaps in our testing

- no unit testing
  - all tests are end-to-end runs of the compiler
  - (happily, compilers are more amenable to this than some types of software)
- no testing of examples in code-based documentation
  - language specification is (mostly) tested
  - wishlist: something that would test code in my Powerpoint slides…
- no “fuzzing” / random testing
  - an important way to simulate novice users?
Parting Thoughts
Technical Choices

• Find / create ways to eat your own dogfood
• Create tools to help yourself
  • Particularly with repetitive tasks, recurring pain points
• Use existing techniques and technologies when available and appropriate
  • “Why waste an hour in the library when you can spend a month in the lab?”
Healthy Attitudes for the Tech World

• Be comfortable with uncertainty, imperfection, changes—they’re bound to occur
  • Don’t be afraid to rewrite code
  • Don’t be overly protective of code that you’ve written
• Don’t be a bean counter (at least about unimportant beans)
  • true problems have a tendency to make themselves known
• Find ways to make your process fun for yourself / your team
Who you are matters a ton

• Don’t be a jerk
  • If you are a jerk, fake it until you’re not
• It’s truly a small world
• Being capable is so much more important than seniority, expertise, …
One more

- Don’t expect that you’ll remember everything forever (you won’t)
  
  => Create notes, documentation, comments for yourself as much as anyone
Tips from my team

• Learn how to break problems into smaller subcomponents
  • easier to estimate level of effort required
  • easier to determine edge cases, avoid backtracking
Chapel Resources
Chapel Central

https://chapel-lang.org

- downloads
- presentations
- papers
- resources
- documentation

The Chapel Parallel Programming Language

What is Chapel?
Chapel is a modern programming language that is...
- parallel: contains first-class concepts for concurrent and parallel computation
- productive: designed with programmability and performance in mind
- portable: runs on laptops, clusters, the cloud, and HPC systems
- scalable: supports locality-oriented features for distributed memory systems
- open-source: hosted on GitHub, permissively licensed

New to Chapel?
As an introduction to Chapel, you may want to...
- read a blog article or book chapter
- watch an overview talk or browse its slides
- download the release
- browse sample programs
- view other resources to learn how to trivially write distributed programs like this:

```c
use CyclicDist;
// use the Cyclic distribution library
config const n = 100;
// use --n=call when executing to override this default
forall i in [1..n] mapped Cyclic(startId=1) do
  writeln("Hello from iteration ", i, ", of ", n, ", running on node ", here.id);
```

What's Hot?
- Chapel 1.17 is now available—download a copy or browse its release notes
- The advance program for CHIUW 2018 is now available—hope to see you there!
- Chapel is proud to be a Rails Girls Summer of Code 2018 organization
- Watch talks from ACCU 2017, CHIUW 2017, and ATPESC 2016 on YouTube
- Browse slides from SIAM PP18, NWCPP, SeaLang, SC17, and other recent talks
- Also see: What's New?
Chapel Social Media (no account required)

http://twitter.com/ChapelLanguage

http://facebook.com/ChapelLanguage

https://www.youtube.com/channel/UCHmm27bYjhknK5mU7ZzPGsQ/
Chapel Community

https://stackoverflow.com/questions/tagged/chapel

https://github.com/chapel-lang/chapel/issues

https://gitter.im/chapel-lang/chapel

read-only mailing list: chapel-announce@lists.sourceforge.net (~15 mails / year)
Suggested Reading: Chapel history and overview

Chapel chapter from *Programming Models for Parallel Computing*
- a detailed overview of Chapel’s history, motivating themes, features
- published by MIT Press, November 2015
- edited by Pavan Balaji (Argonne)
- chapter is also available online
Chapel Comes of Age: Making Scalable Programming Productive

Chapel is a programming language designed to support productivity, general-purpose parallel computing at scale. Chapel's approach can be thought of as striving to create a language whose code is as attractive to read and write as Python, yet which supports the performance of Fortran and the scalability of MPI. Chapel also aims to compete with C in terms of portability, and with C++ in terms of flexibility and reusability. Chapel is designed to be general-purpose in the sense that when you have a parallel algorithm in mind and a parallel system on which you wish to run it, Chapel should be able to handle that scenario.

Chapel's design and implementation are led by Cray Inc. with feedback and code contributed by users and the open-source community. Though developed by Cray, Chapel's design and implementation are parallel, permitting its parallelism to be exploited by libraries and applications that use Chapel as their parallelizing layer or as a basis for creating new libraries and applications.

Chapel is available on various computer systems and platforms, and it is supported by Cray's parallel runtime systems and libraries. Chapel's open-source nature allows it to be used in a variety of applications, from scientific computing to data analytics.

The development of the Chapel language was undertaken by Cray Inc. as part of its participation in the DARPA High Productivity Computing Systems (HPCS) program. HPCS wrapped up in late 2013, at which time Chapel was a competing prototype, having successfully demonstrated several of the main scientific computing applications that are targeted at HPCS. Chief among these was supporting large and small parallel simulations in a single program, with no noticeable loss in performance compared to its C++ counterpart. This was accomplished by supporting the creation of high-level data parallel abstractions like parallel loops and arrays in terms of lower-level Chapel features such as classes, instances, and tasks.

Under HPCS, Chapel also successfully supported the integration of parallelism using distinct language features from those used to control locality and affinity—that is, Chapel programmers specifically which computations should be run on a particular node. This permits Chapel programs to support multithreaded, multi-node, and heterogeneous computing within a single unified language.

Chapel's implementation under HPCS demonstrated that the language could be implemented portably while still being optimized for HPC-specific features such as the RDMA support available in Cray® Gemini™ and Aries™ networks. This allows Chapel to take advantage of native hardware support for remote-pair, gather, and atomic memory operations.

Despite these successes, at the close of HPCS, Chapel was not at all ready to support production codes in the field. This was not surprising given the language's aggressive design and implementation, but it did mean that the potential users were sufficiently positive to start a CUG effort to improve Chapel and move it towards being a production-ready language. Consequently, we refer to this effort as the “two-year push.”

This paper’s contribution is to describe the results of this two-year effort, providing readers with an understanding of Chapel’s progress and achievements since the end of the HPCS program. In doing so, we directly compare the status of Chapel version 1.17, released last month, with Chapel version 1.5, which was released five years ago in April 2013.
SAFE HARBOR STATEMENT

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts.

These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.
THANK YOU

QUESTIONS?

bradc@cray.com
@ChapelLanguage
chapel-lang.org

cray.com
@cray_inc
linkedin.com/company/cray-inc-