Ongoing Efforts

Chapel Team, Cray Inc.

Chapel version 1.13

April 7, 2016

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.

Context for these Slides

- **Our release notes largely report on work appearing in 1.13**
- **However, several other efforts are worth describing as well**
  - Important features that are being designed
  - Work that was not complete in time for 1.13
- **This deck reports on some of those efforts**

Outline

- Construction/Initialization
- Error Handling
- Debian Packaging of Chapel
- numa Locale Model
- Chapel Package Manager
- Twitter Workflow
- Parallel Research Kernels
- Other Notable Ongoing Efforts

Construction/Initialization

Construction/Initialization: Background

- **Chapel's OOP features have been naïve in terms of:**
  - constructors and destructors
  - initialization vs. assignment
  - user-defined default values, parallel initialization, …

- **Need to get this right for:**
  - Correct resource management
    - Some internal types handled unusual memory cases with workarounds
    - Ideal implementation would accommodate these types and more
  - Reasonable handling of const and ref fields

Construction/Initialization: This Effort

- **Refinement of constructor/initializer story for Chapel**
  - What does the method look like?
    - How it works, interaction w/ inheritance, syntax
    - When is it invoked?

- **Goal: principled, broad coverage of likely scenarios**
  - Design influenced by Swift, D
  - As part of principled approach, will refer to as "initializers" only
    - Method name will reflect this:

```
proc Foo () {…}   // old constructor, used name of type
proc init () {…}  // new initializer, uses "init" as name for every type
```

Construction/Initialization: Two Init Phases

- **Initializer body divided into two phases**

  Phase 1:
  - Whole object not yet ready for use – still initializing individual fields
  - Fields must be initialized in order
  - Can leave off fields to use field initializer value, or default for type
    - Such fields are initialized in declaration order, interspersed w/ explicit initialization
  - Explicit and implicit initialization of a field can depend on earlier fields
  - Can define and use local helper variables
  - Can't call methods on 'this' instance (not fully valid until Phase 2)
  - 'field = value' means initialization in this phase

  Phase 2:
  - Object can be treated as a whole
  - Can call methods on 'this'
  - Every field in valid initial state (some may still be modified)
  - Modifications of fields are considered assignment at this point
  - 'field = value' means assignment

Construction/Initialization: Two Init Phases

- **Additional notes on Phase 1/Phase 2:**
  - Cannot assign to const fields in Phase 2
    - Still under discussion
    - Plan to start strict, loosen restriction later if justified
  - If not explicitly noted, initializer body assumed to be in Phase 2
    - Results in more backwards compatibility
    - Could optimize Phase 1-compliant initializers to be treated as Phase 1

Construction/Initialization: Syntax

- **Syntax of initializer**
  - Considered many proposals
  - No clear winner, so we chose one:

```
proc init () {
  … // Phase 1
  super.init(); // Call to parent initializer separates Phase 1 and 2
  … // Phase 2
}
```

Pros:
- Simple syntax
- Can share local variables across phases

Cons:
- Potential to lose dividing line in more complex initializer body
- Not obvious that code behaves differently on either side of init() call
  - e.g., assignment-as-initialization in phase 1 vs. plain-old assignment in phase 2

Construction/Initialization: Alternate Syntax

- **Alternate syntax**

```
proc init () {
  … // Phase 1
  // Can call super.init() as last statement, if desired
} finalize {
  … // Phase 2
}
```

- Finalize block can be dropped

Pros:
  - Clear division between phases

Cons:
  - Sharing a variable between phases is more difficult
  - More naturally supports Phase 1 as the default

- Syntax equally appealing, could switch at a later time
  - Implementation could easily accommodate either syntax w/ same rules

Construction/Initialization: Initializing Parents

- **Calling other initializers**
  - Calls to parent initializer are formatted as:

    ```
    super.init(…);
    ```

    - As an example, flow from child to parent initializer resembles:

```
proc Child.init (…) {
  writeln("Child Phase 1");
  // Can't access parent fields yet
  super.init(…);
  writeln("Child Phase 2");
}
```

```
proc Parent.init (…) {
  writeln("Parent Phase 1");
  super.init(…); // no-op, no parent
  writeln("Parent Phase 2");
  // Since child fields initialized,
  // whole object use is valid
}
```

which will print out

```
Child Phase 1
Parent Phase 1
// Parent of Parent output would go here, if it existed
Parent Phase 2
Child Phase 2
```

Construction/Initialization: Forward to Sibling

- **Calling other initializers**
  - Calls to sibling initializers look like:

    ```
    this.init(…);
    ```

    - Motivation: support Phase 1 code re-use given that methods can't be called

```
proc Child.init (…) {
  writeln("Orig Phase 1");
  // Can't initialize fields
  this.init(…);
  writeln("Orig Phase 2");
}
```

```
proc Child.init (…) {
  writeln("Sibling Phase 1");
  // Should initialize child fields
  super.init(…); // no-op, no parent
  writeln("Sibling Phase 2");
}
```

which will print out

```
Orig Phase 1
Sibling Phase 1
// Parent output would go here, if it existed
Sibling Phase 2
Orig Phase 2
```

Construction/Initialization: Calling Initializers

- **Calling other initializers**
  - If no super.init() or this.init(), makes implicit no-argument super.init() call
    - At start of initializer body (because body is assumed to be Phase 2)
  - super.init() calls are currently most applicable to classes
    - Record inheritance story is not yet fully defined
    - For records, or when no parent is present, super.init() call is no-op

Construction/Initialization: Compiler Initializers

- **Compiler-generated initializers**
  - Initializes all fields by default
    - Will use field declaration's initializing value, if present
    - Otherwise will use default value for type
  - Not generated if any user-defined initializer present
    - In step w/ current behavior
  - Would like a way to opt back in for creation of default
    - Defining semantics for this is nonessential, future work

Construction/Initialization: Noinit

- **Related Topic: Noinit**
  - Constructs instance, doesn't initialize (all of) it yet
  - Especially useful for arrays and other large data structures
    - Can skip default initialization when unnecessary and costly
    - For optimization purposes:

```
var A: [1..1000] int = noinit;    // "Don't initialize because I'm about to."

A[1] = 14;                        // Helps compilers avoid being conservative when
for i in 2..1000 {                //  unable to prove the default init is unnecessary.
  A[i] = i*A[i-1];
}
```

Construction/Initialization: Noinit

- **Noinit, continued**
  - Invalid to use instance before initialization is finalized

```
var A: [1..1000] int = noinit;
var badAccess = A[15]; // A[15] is garbage memory right now
```

  - Supported by any type unless type designer opts out
    - See slide on noinit and compiler-generated initializers for details
  - Previous implementation was all-or-nothing
    - All of instance initialized, or uninitialized
    - Some types have fields which must always be valid
    - E.g. arrays should always have a domain defined for space allocation
    - Led to desire for more fine-grained control on what noinit means for a type

Construction/Initialization: Noinit

- **How does noinit work in initializers?**
  - Can use on fields in Phase 1 of initializer

```
proc init (…) {
  field = noinit;
  …
}
```

    - Phase 2 should give value to field or will need to be very careful in methods
  - Invalid in Phase 2 (all fields already initialized)

Construction/Initialization: Noinit

- **Noinit will be implicit param argument to initializer**
  - Compiler will call initializer with extra argument “noinit = true”
    - If constructor doesn’t handle it, will cause error “noinit not defined on type”
  - Allows code sharing between init/noinit initialization, e.g.

```
proc init (param noinit=false) {
  if (noinit) {
    field = noinit;
  } else {
    field = …;
  }
  // Rest of Phase 1 code, followed by Phase 2
}
```

Construction/Initialization: Noinit

- **Compiler-generated initializers include noinit arguments**
  - Applies noinit to all fields when set to ‘true’
  - Presence of a user-defined initializer disables this support
    - since it disables the compiler-generated initializer altogether
  - Thus, user-defined initializers must explicitly support noinit arguments
    - (when the capability is desired)

Construction/Initialization: Copies

- **Related Topic: When are record copies added?**
- **Background:**
  - Compiler today very ad hoc, “as many as are necessary”
    - Often buggy
  - Desire to document intended behavior and make compiler adhere
- **Status: still under discussion, promising direction**
  - Describes when added (compiler implementation)
  - Describes user’s mental model (aimed at spec/user’s guide)
  - Provides details on arrays, specifically
    https://github.com/mppf/chapel/blob/copy-semantics-chip/doc/chips/10.rst

Construction/Initialization: Status

- Design document available
- **Current design is sufficient to begin implementation**
  - We are prepared to adjust in some areas as needed
    - Const field assignment
    - Alternate syntax
    - etc.

Construction/Initialization: Next Steps

- **Start on Implementation**
- **Continue discussing some open areas**
  - Copy initializers:
    - control behavior when compiler-created copies occur
    - considering an approach similar to D's postblit
    - syntax remains an area of discussion
    - also:
      - Will there be a super call?
      - Could a type define more than one copy initializer?
      - Can you set const fields in its body?
      - Should method calls be allowed in its body?
  - Move initializers:
    - support compiler optimization when copying dead expressions
    - haven't reached consensus on this topic yet

Error Handling

Error Handling: Background

- **Chapel currently lacks a general strategy for errors**
- **Standard library uses two primary approaches at present:**
  - calls `halt()`
  - uses optional output arguments (`out error: syserr`)
    - if argument is provided, user must handle; otherwise call halt()
- **Each of these approaches has serious drawbacks:**
  - halting the program is not appropriate in library code
  - current output argument approach…

    …only returns error codes, not additional state

    …doesn't permit users to easily add new error codes or state
- **A more general strategy is desired, supporting:**
  - the ability to write bulletproof code
    - ideally, in a way that supports propagation of errors, as with exceptions
  - the ability to get useful messages when errors are not handled

Error Handling: This Effort

- **Design a new approach for error handling**
- **We considered:**
  - using generalized error objects instead of error codes
  - returning (result, error) tuples
  - returning error objects via optional out arguments
  - exceptions along the lines of C++
  - an exception-like approach (inspired by Swift)
- **Exception-like approach preferred:**
  - Represents a middle ground
    - arguably acceptable to devotees of both exceptions and error codes
  - Easier to implement than stack unwinding
    - re-uses the existing return mechanisms
  - Fits well with existing task parallelism
- **Detailed proposal: CHIP 8**

Error Handling: Basic Model

- **Functions that can raise an error are declared with `throws`**

```
proc canThrowErrors() throws { … }
proc cannotThrowErrors() { … }
```

- **Calls that `throw` should be decorated with `try` or `try!`**
  - makes the control flow possibilities clear without inspecting the callee
    - `try` propagates the error to an enclosing `do/catch` block or out of a throwing function
    - `try!` halts if an error occurred

- **Programs can respond to errors with `do/catch` statements**

on failure:

invoke **`catch`** block

***halt*()** program

```
do {
    try canThrowErrors();
    try! canThrowErrors();
} catch {
    writeln("The first call failed.");
}
```

Error Handling: Tasks

- Can capture errors on task join

```
proc throwsError() throws { throw new Error(); }
proc doesNotThrowError() throws { }
do {
    try cobegin {
        { try throwsError();             }
        { try doesNotThrowError (); }
        { try throwsError ();            }
    }
} catch errors : Errors {
    for e in errors {
        writeln(e);
    }
}
```

2 tasks **`throw`** error

2 elements in `errors`

`writeln()` invoked twice

Error Handling: Iterators

- Errors can be raised in iterators too
- Such errors end serial iteration

```
iter glob(pattern: string): string {
    ...;
    if (err != 0 && err != GLOB_NOMATCH) then
        throw new Error("unhandled error in glob()");
}
do {
  try for x in glob() {
    writeln(x);
  }
} catch e: Error {
  writeln("Error in glob: ", e);
}
```

invoke **catch** block when iterator **throws**

Error Handling: Status, Next Steps

Status:

- Group consensus on general direction in **CHIP 8**
  https://github.com/chapel-lang/chapel/blob/master/doc/chips/8.rst
- Some questions remain:
  - can `try` or `try!` apply to a region of code?
  - when is `try` required?
    - strict rules → more checking: `try` required for all calls that can `throw`
    - relaxed rules → easier-to-read code: `try`, `throw` assumed by default
  - what compile-time flags or knobs should control behavior?
    - e.g. a flag / scope could control halting or ignoring errors in relaxed mode
  - how to handle runtime errors (e.g. out of memory)?

Next Steps:

- Resolve open design questions
- Start implementation

Debian Packaging of Chapel

Packaging: Background

- **Chapel is currently available via:**
  - Building from source

  - Homebrew package for OS X
  - Cray RPM for Cray systems

Packaging: This Effort

- **Debian package for Chapel**
  - Removed these third-party libraries/stubs from the package:
    - GASNet, fltk, libhdfs3, massivethreads, LLVM
  - Building with the following as a dependency:
    - GMP
  - Package will include these third-party libraries:
    - hwloc, jemalloc, qthreads, re2, utf8-decoder
    - desirable for good single-locale performance

- **Package characteristics:**
  - single-locale only
  - will support good performance due to jemalloc, qthreads, hwloc
  - will support regular expressions and GMP

Packaging: Status

- **Packaging setup scripts and debian files available at:**
  - https://github.com/chapel-lang/chapel-packaging
  - Scripts build package from release tarball and debian files

- **At the time of writing,**
  - 1.13 Debian package is drafted
  - Nearly ready for review, followed by merge into 'Debian/sid'
    - Review process can span weeks, depending on complexity of package

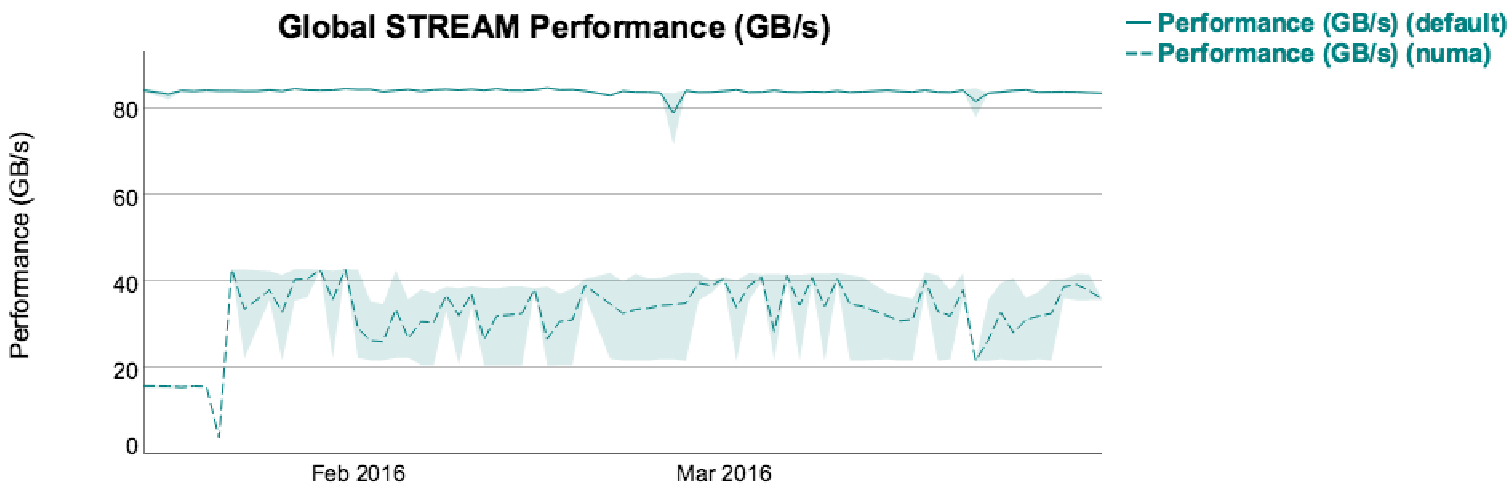
Packaging: Next Steps

- **Submit pull request for our package into 'Debian/sid'**
  - After merging, the following will happen automatically:
    - Chapel will deploy with Debian 9 (2017)
    - Ubuntu and other downstream distributions will pull Chapel package
- **Backport Chapel package for Debian 8**
  - So that Chapel is available on current release of Debian
- **Expand packaging to other large distributions**
  - (or find community developers who are interested in doing so)
  - e.g.,
    - Arch Linux
    - Fedora, RHEL, CentOS
    - Suse, OpenSUSE

numa Locale Model

numa Locale Model: Background

- **'numa' locale model doesn't produce desired performance**
  - Gets data placement right only inadvertently
    - when inter-loop task placement is similar with respect to data locality
    - when new variables occupy already-placed memory
  - When similarly "lucky", 'flat' beats 'numa' due to less overhead:



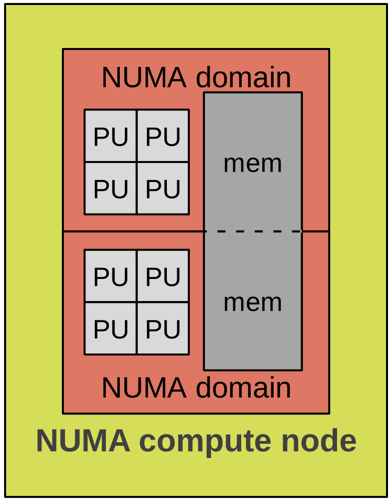
- **Want 'numa' to ourperform 'flat' on any NUMA-friendly app**
  - With a few caveats: e.g., big enough to amortize overheads

numa Locale Model: This Effort

- **Add numa-awareness to DefaultRectangular arrays**
  - Distribute array data predictably across NUMA domains
  - Matching how DefaultRectangular domain already distributes tasks

```
forall … do <something> ;

coforall … on … {      // across numa sublocales
  coforall … on … {  // across PUs within subloc
    for … do <appropriate subset of something>
  }
}
```



  - Arrays were *single-ddata*: 1 data block per node
  - Now may be *multi-ddata*: 1 data block per sub-locale

- **Goals:**
  - Principled NUMA-oriented performance, not dependent on "luck"
  - No loss of performance when multi-ddata isn't used

numa Locale Model: a Work-in-Progress

- **Enabled when > 1 sublocale/node, for “big enough” arrays**
  - 10**6 elements during development, but subject to tuning later

- **Implementation has been slower than desired**
  - Complicated coding environment
    - DefaultRectangularArr is the heart of the array implementation
  - Lots of optimization tweaks
    - RADopt (Remote Access Data – caches network-remote array metadata)
    - bulk transfer optimization
    - bulk I/O
  - Interactions with iterator implementation
  - Need to limit overheads, maintain performance
    - can’t hurt single-ddata array performance

numa Locale Model: Status and Next Steps

Status:

- Optimizations disabled: RADopt, bulk transfer, bulk I/O
- Was fully functional before taking a step back to focus on performance
- Performance still doesn't quite match that of flat

Next Steps:

- Bring performance up
- Enable optimizations currently turned off
- Start on related memory management work:
  - numa-awareness in runtime memory layer(s)
  - on Cray X* systems, integrate with use of hugepages

Chapel Package Manager

Package Manager: Background

- **Current approach:**
  - modules are stored in our repository
  - modules are released with Chapel
- **This approach will fail as the community grows**
  - Developers must sign a CLA
  - Code must be under a compatible license
  - The core team needs to review each module
  - Modules are gated for release alongside the compiler
- **And, a better model could help grow the community faster**
  - Simplify sharing of code
  - Reduce barriers to doing so

Package Manager: This Effort

- **CHIP 9 proposes a package manager called *mason***
  - *mason* is a command-line tool for building Chapel programs
  - *mason.toml* is a file storing module metadata for an application/library
    - specifies dependencies which can be downloaded during a build
  - proposes using *Nix* to manage C dependencies
  - proposes writing mason primarily in Chapel

Package Manager: Status and Next Steps

Impact:

- Would improve ability to use community-contributed Chapel code
- Would simplify sharing Chapel code within the community

Status:

- Initial proposal created

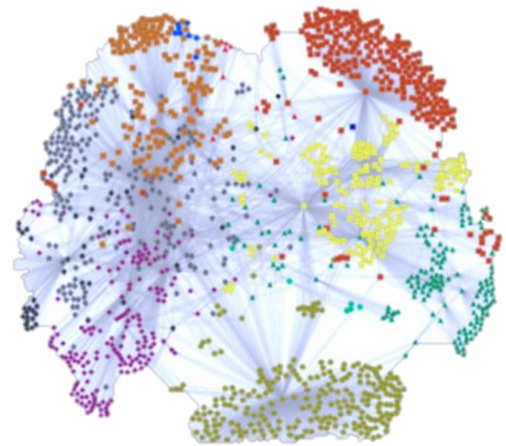
Next Steps:

- Develop proposal further
- Solicit feedback from the community
- Start implementation

Twitter Workflow

Twitter Workflow

- **Background:** **Studying Twitter Workflow Analysis as a benchmark in data analytics against Spark**
- **Since the 1.12 report:**
  - Committed benchmark to test suite
  - Completed and merged supporting library changes



- **Impact:**
  - Chapel now supports:
    - JSON support for List
    - Skipping unknown fields when reading JSON records/classes
    - Generating random permutations
- **Next Steps:**
  - Re-prioritize this effort
  - Apples-to-Apples benchmarks between Chapel and Spark
  - Study and understand the performance delta between them

Parallel Research Kernels

Parallel Research Kernels: Background

- **About Parallel Research Kernels (PRK):**
  - Kernel computations developed to investigate parallel performance
    - Motifs common in parallel applications
    - Not intended as benchmarks
  - 10 small easy-to-port kernels
  - 14 submitted implementations and counting...
    - Including serial C, OpenMP, MPI, SHMEM, UPC, etc.
  - Good opportunity to explore Chapel's strengths and weaknesses
    - Particularly in multi-locale performance and scalability
  - Will allow comparisons between Chapel and other parallel approaches

Parallel Research Kernels: This Effort

- **Chose 3 most popular kernels:**

Stencil:

- Apply stencil operation to 2D square input matrix writing to output matrix
- Input matrix is updated each iteration to add communication per iteration
- We have primarily focused on stencil performance to date

Synch:

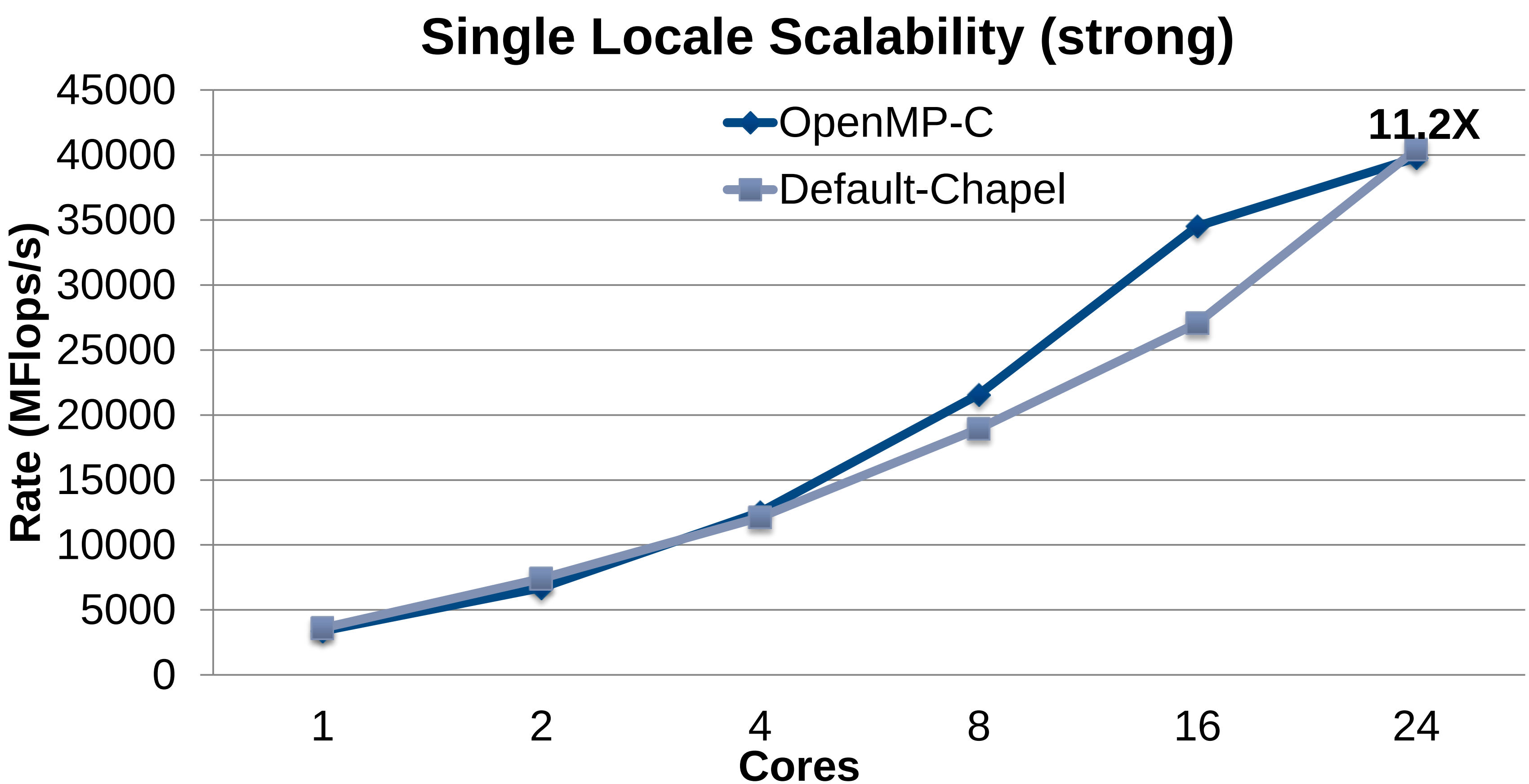
- Point-to-point synchronization (p2p_synch)
- Running stencil operation across matrix with different iterations running simultaneously across locales.

Transpose:

- Transpose square matrix A into matrix B
- Parallelize matrix across columns and perform transpose

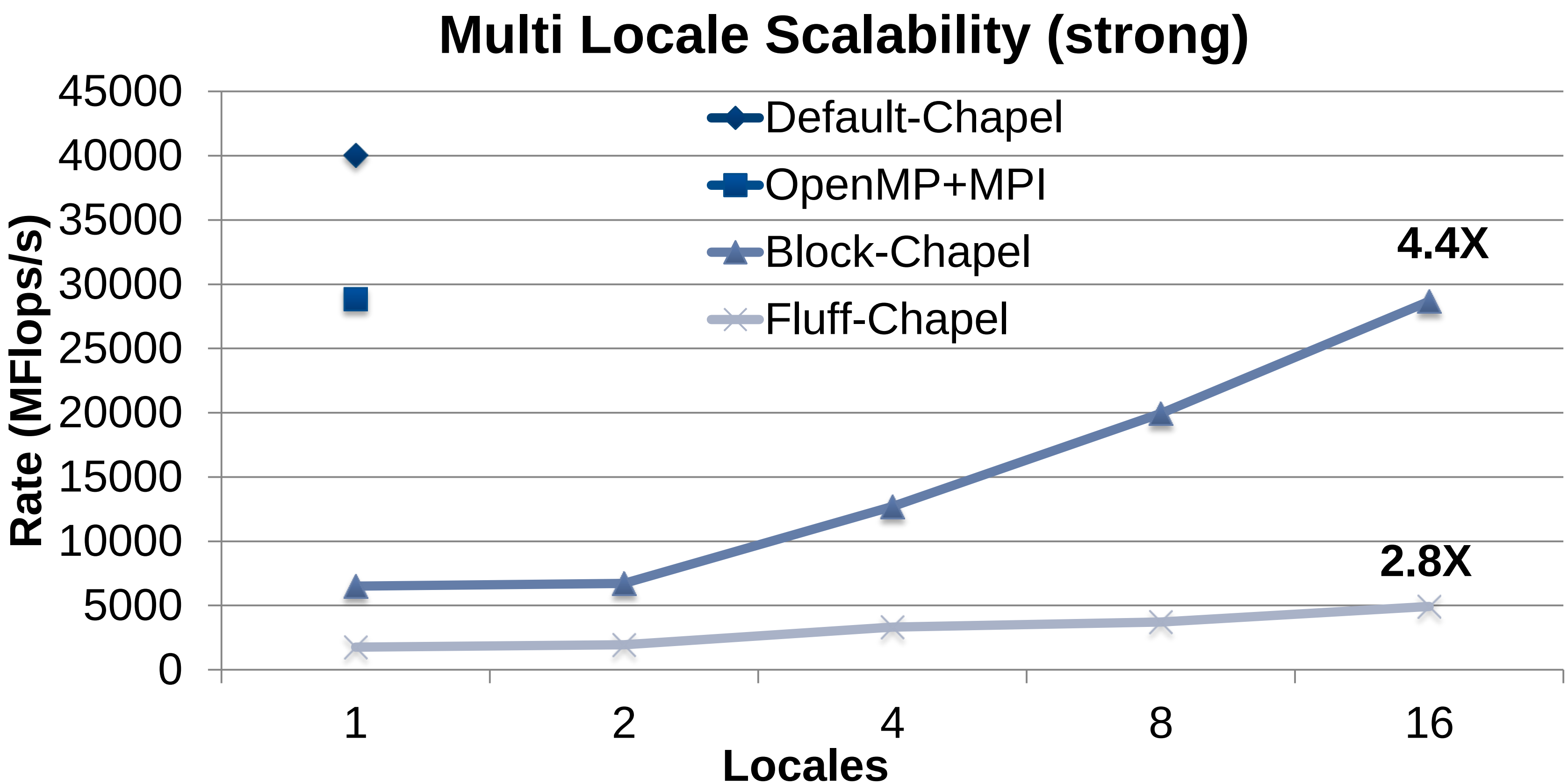
Parallel Research Kernels: Stencil - Single Locale

- Chapel scaling & performance are competitive with C / OpenMP
  - Cray XC, 1 locale, Chapel: qthreads-gnu
  - 3 iterations, 32000x32000 stencil matrix, untiled, star stencil operation



Parallel Research Kernels: Stencil - Multi Locale

- Chapel not currently competitive with C / OpenMP + MPI (not shown)
  - not surprising given minimal effort put into stencils so far (see MiniMD slides)
  - Cray XC, 24 cores / locale, Chapel: ugni-qthreads-gnu
  - 3 iterations, 32000x32000 stencil matrix, untiled, star stencil operation



Parallel Research Kernels: Stencil - Scalability

- **Implementations differ for single- vs. multi-locale**
  - Single-locale: uses Default Distribution
  - Multi-locale: uses Block Distribution or Stencil Distribution
- **For single locale (24 cores):**
  - Default-Chapel achieves 138.8% performance of OpenMP+MPI
  - Block-Chapel achieves 22.5% performance of OpenMP+MPI
  - Fluff-Chapel achieves 6.1% performance of OpenMP+MPI
- **Positive outlook:**
  - Block-Chapel is a naïve implementation of the Stencil algorithm
  - There's no reason Fluff-Chapel should underperform Block-Chapel
    - It essentially *is* Block with support for ghost cells ("fluff")
    - Suggests additional problems with Stencil Distribution
      - or improvements to *BlockDist* not reflected in *StencilDist*?
    - Scalability should dramatically improve as Stencil Distribution improves
      - due to opportunity for communication optimizations

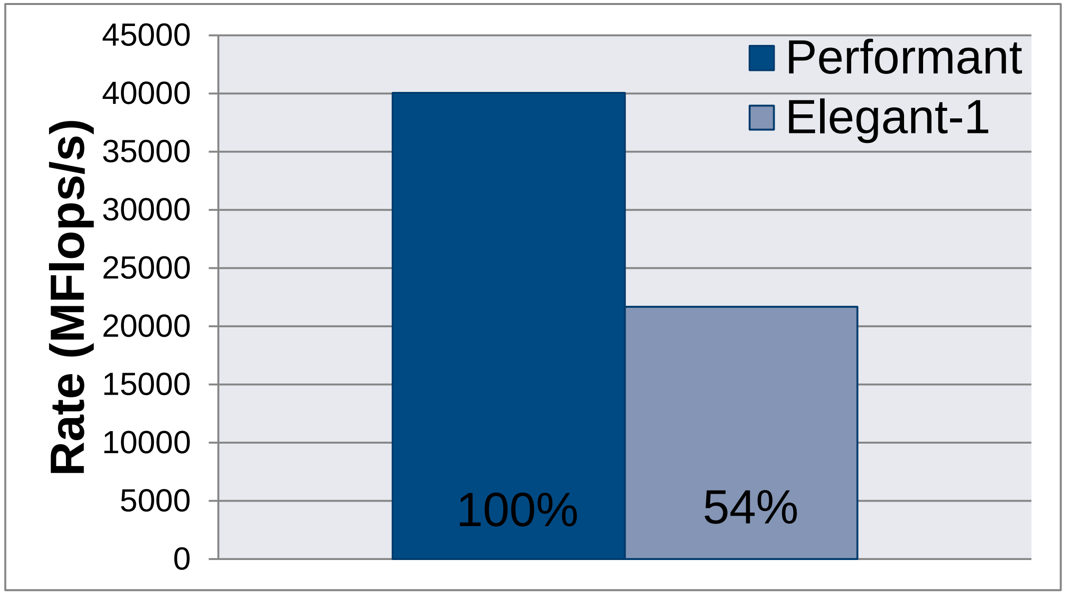
Parallel Research Kernels: Stencil - Performance

- **Exploration of more elegant solutions**
  - Many expressions in Stencil can be expressed with idiomatic Chapel
    - Some of these idiomatic expressions were found to perform poorly
    - As Chapel matures, these performance deltas should decrease
- **Some of the performant-to-elegant differences:**

```
// Incrementing input

// Performant: explicit
forall (i,j) in Dom {
  input[i, j] += 1.0;
}

// Elegant-1: promoted +=
input += 1.0;
```

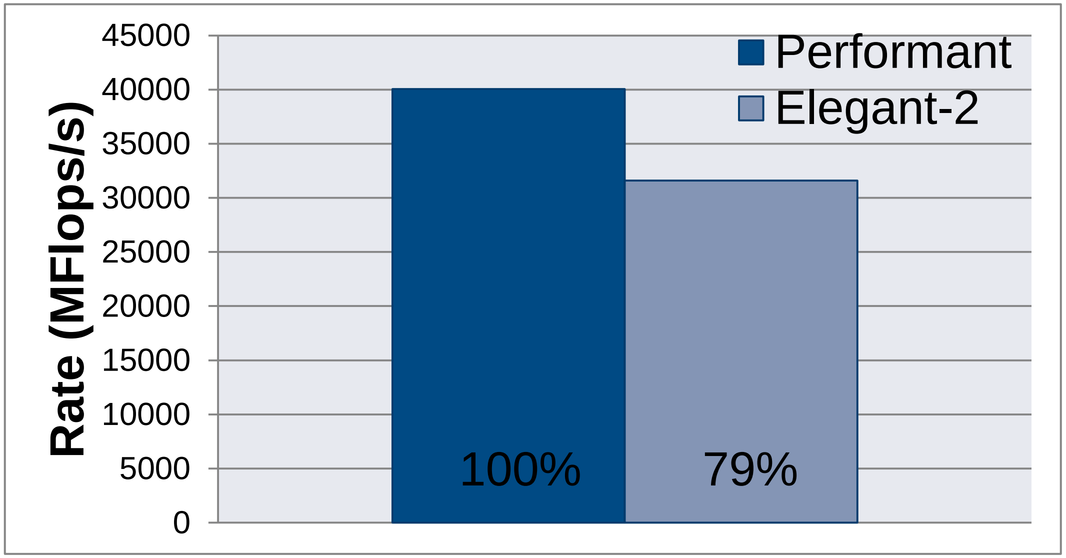


Parallel Research Kernels: Stencil - Performance

```
// Writing stencil operation results to output

// Performant: write to local temp variable, then write to output once
for ii in -R..R do
  for jj in -R..R do
    tmpout += weight[R1+ii][R1+jj] * input[i+ii, j+jj];
output[i, j] += tmpout;

// Elegant-2: write directly to output each inner iteration
for ii in -R..R do
  for jj in -R..R do
    output[i, j] += weight[R1+ii][R1+jj] * input[i+ii, j+jj];
```



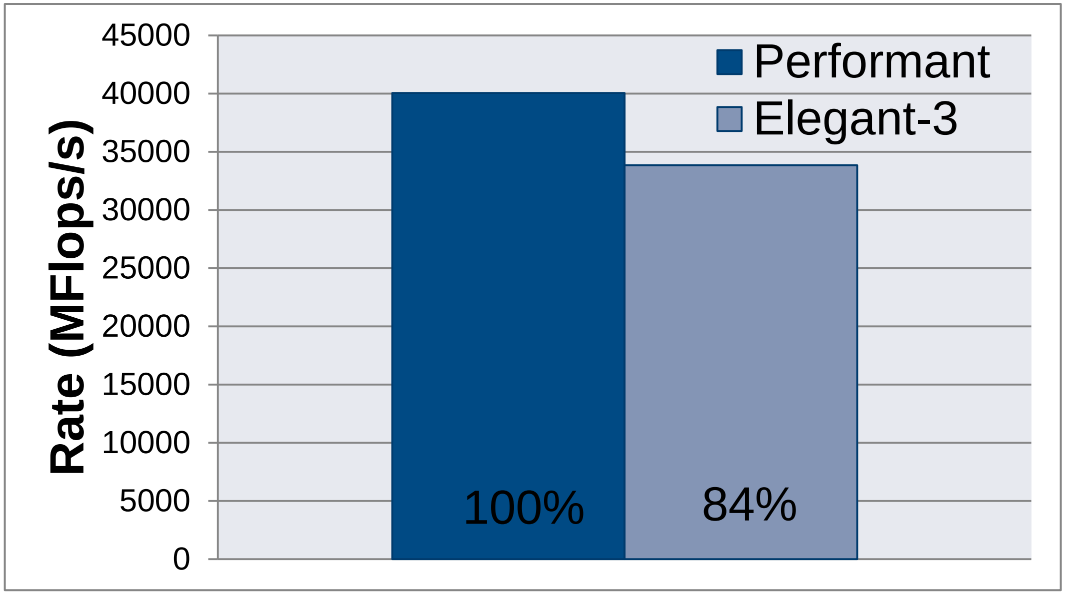
Parallel Research Kernels: Stencil - Performance

- **Weight matrix representation**
  - Tightly looped over for innermost computation
  - Tuple of tuples representation performs better currently
  - 2D Array representation is slower
    - Tuples map to static C arrays, Chapel arrays to heap-allocation + metadata

```
// Data structure of weight matrix

// Performant: tuple of tuples
var weight: Wsize*(Wsize*(dtype));

// Elegant-3: 2D array
var weight: [weightDom] dtype;
```



Parallel Research Kernels: Next Steps

- **Stencil**
  - Prioritize improving Stencil Distribution (Fluff-Chapel)
    - Strive to reduce gap relative to OpenMP+MPI
  - Reduce performance delta between performant/elegant expressions
- **Other PRKs**
  - Investigate Transpose and Synch at the same level of detail as Stencil
  - Investigate additional kernels as time and interest permit
- **Merge PRK implementation into ParRes/Kernels**
  - Intel team has expressed interest in studying Chapel performance in their framework

Other Notable Ongoing Efforts

Other Notable Ongoing Efforts

- Users Guide (covered in documentation deck)
- Support for KNL HBM (covered in portability deck)

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

CRAY CHAPEL

CRAY

THE SUPERCOMPUTER COMPANY