Hewlett Packard Enterprise

CHAPEL: AN EXAMPLE OF LANGUAGE-LEVEL PGAS SUPPORT

Lydia Duncan

July 22, 2021

CHAPEL: LANGUAGE-LEVEL PGAS

Introduction

- This talk will focus on the programming language Chapel, covering a number of its PGAS features
  - Attendees that are unfamiliar with the language will hopefully be able to follow along
    - There's also many full tutorials online at https://chapel-lang.org/learning.html
- It will use real world examples from two major recent Chapel programs, CHAMPS and Arkouda
- These programs are both irregular and data-intensive but come from different problem spaces

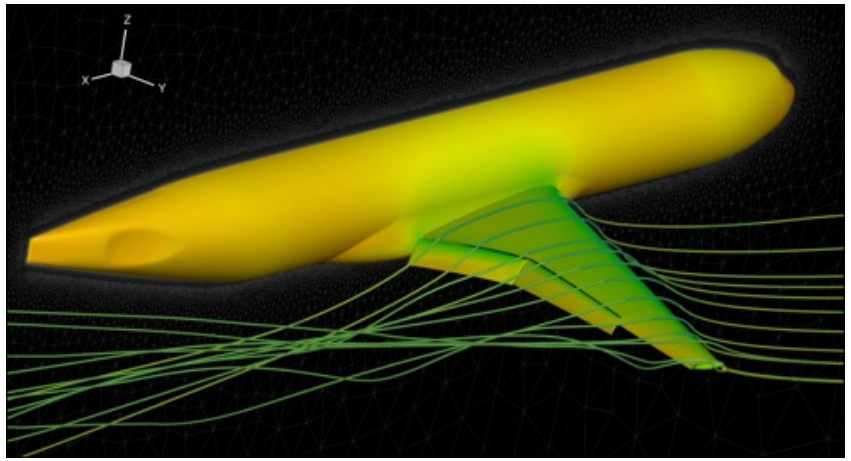
CHAMPS IN ONE SLIDE

What is it?

- Computational Fluid Dynamics framework for airplane simulation written from scratch
- Modular design, permitting various computational modules to be integrated (or not)
- ~48k lines written in ~2 years

Who did it?

- Professor Eric Laurendeau's team at Polytechnique Montreal
- not open-source (yet), but available by request to researchers



Why Chapel?

- performance and scalability competitive with MPI + C++
- provided a simpler coding experience for computational scientists
  - has enabled senior students to move faster
  - has permitted junior students to contribute more readily
- net result: achieves competitive results w.r.t. established, world-class frameworks from Stanford, MIT, etc.

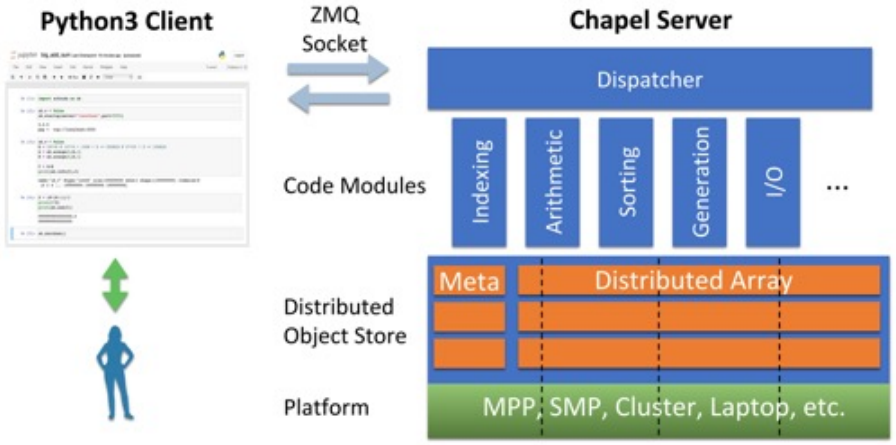
ARKOUDA IN ONE SLIDE

What is it?

- A Python library supporting a key subset of NumPy- and Pandas-like features for Data Science
- Implemented using a client-server model with Chapel as the server to support scalability
- Designed to compute results within the human thought loop (seconds to minutes on TB-scale arrays)
- 16k lines of Chapel, largely written in 1 year, further improved over 1-1/2 years

Who did it?

- Mike Merrill, Bill Reus, et al., US DOD
- Open-source: https://github.com/Bears-R-Us/arkouda



Why Chapel?

- high-level language with C-comparable performance
- great distributed array support
- ports from laptop to supercomputer
- close to Pythonic—doesn't repel Python users who look under the hood

OUTLINE

- Introduction (done!)
- PGAS/Chapel
- Optimizations
- Evaluation
- Conclusion

BRIEFLY: WHY PGAS?

- A Partitioned Global Address Space…

  …reduces management burden of ensuring multiple copies of a program are running in tandem

  …enables easier transfer of data across nodes

  …can make communication code easier to maintain and reduce program clutter

  …can enable optimizations like batching to reduce communication overhead

  …is easier to debug

CHAPEL: LANGUAGE-LEVEL PGAS

- Chapel was designed to use PGAS
  - A single program can have as many nodes as desired (called “locales”), specified at execution via ‘-nl’ flag
    - “nl” is short for “number of locales”

  - Throughout a single program, variables can be allocated on one of the specified locales
    - This is done using an ‘on’ statement, which takes a locale or a variable and migrates to the corresponding locale:

```
var a: int = 5;
writeln(a.locale.id);     // The program starts on locale 0, so this will print ‘0’
on Locales[3] {
  var b: int = 3;
  writeln(b.locale.id); // The ‘on’ statement moved us to locale 3, so this will print ‘3’
}
```

CHAPEL: LANGUAGE-LEVEL PGAS

- Variables on remote locales can be accessed remotely, but communication will occur

```
var a: int = 5;
on Locales[3] {
  var b: int = 3;
  a += 1;     // This update of 'a' will incur communication because it is on a different locale
  on Locales[0] {
    a += 1; // This update of 'a' will not incur communication now
    b += 1; // But this update of 'b' will
  }
}
a += 1;       // And this update of 'a' also will not incur communication, because we've returned to locale 0
```

CHAPEL: LANGUAGE-LEVEL PGAS

- Here's a concrete example using an 'on' statement from the CHAMPS 'main' function
  - Each iteration will execute in parallel on one of the locales
    - 'coforall' is a special flavor of 'for' loop that spawns a task for each iteration, continuing the program after all tasks finish
  - 'Flow' and 'turb' inputs are copied locally to minimize communication through the rest of the loop body
  - 'localZoneIndices' and 'localZones' refer to the portions of a distributed array that live on this current locale
    - I'll talk about arrays more soon

```
coforall loc in Locales do
on loc {
  // Storing on the Locale the flow inputs
  var localFlowInputs = flowInputs;
  var localTurbInputs = turbInputs;

  const localZonesIndices = globalHandle.zones_.localSubdomain();
  ref localZones = globalHandle.zones_.localSlice(localZonesIndices);
… }
```

CHAPEL: LANGUAGE-LEVEL PGAS

- Locales and on statements are an example of PGAS at the language-level
  - Their syntax is directly supported by the compiler
  - Every variable knows what locale it is defined on
    - And the locale of a segment of code can be easily determined using 'here'

```
writeln(here.id); // Prints the locale id of the current locale
```

  - Much of what is accomplished by these constructs is possible through a library
    - But requires more effort by the programmer and has more potential for mistakes
      - Must remember to start tracking the locales and clean them up afterwards
      - Types that want to track their definition point must explicitly add fields or calls to accomplish it
      - Resulting bugs can be subtle
  - Optimizations can be made that take locality into account
    - Communication can get batched at 'on' statement boundary, if compiler thinks that will be more efficient


ARRAYS AND DOMAINS

CHAPEL: ARRAYS AND DOMAINS

- The examples so far have been relatively simple, limited to a few variables
- This symposium is focused on data intensive applications, so let's transition to larger data structures

- Chapel's array support is a bit different from other languages
  - The index set for an array is a separable concept, known as a domain
    - Domains can be used to drive iteration over an array…

```
var D = {0..3};
var arr: [D] int;
for i in D {
  arr[i] = i;
}
```

… and can even be shared by multiple arrays

```
var D = {0..3};
var arr1, arr2: [D] int;
```

CHAPEL: ARRAYS AND DOMAINS

- Domains have all sorts of benefits
  - Can reduce potential for off-by-one errors by using domain of an array to iterate over its contents
  - Can adjust domain's declared size without needing additional adjustment to places it is iterated over
  - Can have different starting indices, alignments, or stride patterns
    - Indices covered can be sparse or dense
    - Indices could even be types other than integers, like strings (this is called an associative domain)
    - These features allow the domain to more naturally represent the information that is being stored
  - Can perform operations on the domain to get subsections of it, or transform it in various ways
    - E.g. 'interior' method will return a new domain with specified number of indices for each dimension

CHAPEL: ARRAYS AND DOMAINS

- Separating indices into its own type is another example of why language support makes a big difference
  - Library types could be written to perform a similar function
    - But implementation would be clunkier
  - Language-level support makes its handling natural
  - Indices can be reused more easily, both for traversal and for use in another array
  - Code can be more self-documenting

- The concept of domains is powerful in a single locale setting
  - It's even more powerful in a multi-locale setting…

CHAPEL: ARRAYS AND DOMAINS

- By default, a domain specifies memory on a single locale - the locale where the array is declared
  - But it can be adjusted to distribute index sets across multiple locales with some known patterns
    - This is done using a “domain map”, specified using the ‘dmapped’ clause when declaring the domain
    - For instance, the following will make an array laid out in a cyclic pattern across locales

```
use CyclicDist;
// The Cyclic domain map distributes data in order across all locales, so no index will be on the same locale as the previous.
// E.g. with numlocales=5, locale number for each index will be [0, 1, 2, 3, 4, 0, 1, 2, 3, 4 ...]
var D = {0..<50} dmapped Cyclic(startIdx=0);
var arr: [D] int;
for i in D {
  writeln(arr[i].locale.id);
}
```

CHAPEL: ARRAYS AND DOMAINS

- Other supported domain maps include:
  - Block – distributes the data evenly in continuous chunks across locales
  - Hashed – for distributing associative arrays across locales
  - Replicated – stores a copy of every element on each locale
  - Stencil – variant of Block distribution that creates read-only caches of elements adjacent to each locale's block

- Can be used to help minimize communication for a variety of different access patterns

- Many of these strategies depend on number of locales present
  - Changing number of locales used at execution time changes what indices are located on which locales
  - This can be used to further balance the program's data accesses

CHAPEL: ARRAYS AND DOMAINS

- Swapping to using a different domain map is easy, enabling experimentation during development
  - All it requires is changing the 'dmapped' clause when declaring the domain
  - Arkouda has a function* that returns a different domain depending on a configuration parameter:

```
proc makeDistDom(size: int) {
  select MyDmap {
    when Dmap.defaultRectangular { return {0..#size}; }
    when Dmap.blockDist {… return {0..#size} dmapped Block(boundingBox={0..#size}); …}
    when Dmap.cyclicDist { return {0..#size} dmapped Cyclic(startIdx=0); }
    …
  }
}
```

*from the source file SymArrayDmap.chpl, which is used in many places on its own and through calls to makeDistArray from the same source file

CHAPEL: ARRAYS AND DOMAINS

- Distributed arrays are beneficial for data intensive programs
  - When data set is large, sometimes doesn't fit on single node
  - When number of operations being performed is large, dividing work amongst locales increases speed
    - With caveat that communication should be minimized, as one would expect
  - Keeping distributed contents in same data structure can allow more general code to be written
    - E.g. compare code that loops over a single array (lines 195-197 of Arkouda's 'concatenateMsg' fn in ConcatenateMsg.chpl):

```
forall (i, s) in zip(newSegs.domain, newSegs) with (var agg = newDstAggregator(int)) {
  agg.copy(esa[i+segStart], s);
}
```

... versus code that must loop over an array per node (theoretical version of the above):

```
forall (i, s) in zip(newSegs0.domain, newSegs0) with (var agg = newDstAggregator(int))
{ agg.copy(esa0[i], s); }
forall (i, s) in zip(newSegs1.domain, newSegs1) with (var agg = newDstAggregator(int))
{ … }
…
```

CHAPEL: ARRAYS AND DOMAINS

- First snippet can adjust for various numbers of locales more easily
  - While second must add a traversal per locale
- If changes to the contents of the body are needed, first one requires an update in single place
  - While second must update each loop the same way

– Code snippet 1 (from previous slide):

```
forall (i, s) in zip(newSegs.domain, newSegs) with (var agg = newDstAggregator(int)) {
  agg.copy(esa[i+segStart], s);
}
```

– Code snippet 2:

```
forall (i, s) in zip(newSegs0.domain, newSegs0) with (var agg = newDstAggregator(int))
{ agg.copy(esa0[i], s); }
forall (i, s) in zip(newSegs1.domain, newSegs1) with (var agg = newDstAggregator(int))
{ … }
```

…

CHAPEL: ARRAYS AND DOMAINS

- In summary, arrays and domains in Chapel have tools for distributing their contents in a PGAS fashion
  - Different strategies for distributing data can easily be swapped in, enabling experimentation
  - And distributed arrays will adjust their distribution depending on number of locales available

- Domains can be used to traverse arrays defined on them in a robust manner
  - They define tools that enable local work to minimize communication and increase parallelism
  - They can be used by multiple arrays to minimize potential for incompatible array sizes


OPTIMIZATIONS

CHAPEL: OPTIMIZATIONS

- Making PGAS features part of a language rather than a library enables better compiler analysis
  - Keywords like ‘on’ give knowledge of locality and when locality shifts occur
    - Compiler can use this to optimize communication
  - Arrays and domains are known to handle how their data is distributed
    - Compiler can take advantage of this knowledge to properly cache information

- Chapel provides a lot of optimizations, which can be seen with ‘--help’ flag at compilation
  - The team is actively looking for new improvements that can be made

CHAPEL: OPTIMIZATIONS

Unordered Copy Optimization

- Remote copies that aren't used right away don't have to complete before next line(s) of code
  - Communication is expensive, hiding that latency is helpful when possible

- Compiler can optimize when all of the following are true:
  - Inside a forall loop (because an iteration isn't dependent on previous iterations)
  - Involved variables will outlive loop scope
  - Variables aren't used for synchronization
  - Result of copy is not used in the same iteration

- Users can explicitly request an unordered copy, too
  - These copies are implicitly fenced at task/forall termination, and can be explicitly fenced as well

```
unorderedCopy(dst, src); // Unordered copy of 'src' into 'dst'
unorderedCopyFence();    // Explicit fence
```

CHAPEL: OPTIMIZATIONS

Unordered Copy Optimization

- This optimization is beneficial for irregular access patterns
  - Especially when updates can be grouped into many simple loops
  - In tandem with well-chosen domain map, a lot of communication can be avoided or hidden

- Here's an example of an explicit 'unorderedCopy' call in Arkouda
  - From lines 615-619 of the 'peel' function in SegmentedArray.chpl

```
forall (srcStart, dstStart, len) in zip(oa, leftOffsets, leftLengths) {
  for i in 0..#(len-1) {
    unorderedCopy(leftVals[dstStart+i], va[srcStart+i]);
  }
}
```

- If you're curious about this optimization, more information can be found in Chapel 1.19 and 1.20 release notes performance decks, available here: https://chapel-lang.org/release-notes-archives.html

CHAPEL: OPTIMIZATIONS

Automatic Local Access Optimization

- Certain distributed array accesses incur a check to ensure access is local
  - Useful when access is not local, but overhead when it is

```
var D = newBlockDom({1..N});
var A: [D] int;
forall i in D do
  A[i] = calculate(i);
```

- But compiler can detect/adjust code to avoid check at runtime if certain conditions are true
  - Including when multiple arrays with same domain are involved
  - Otherwise, code will fall back on explicit check as a last resort

- Users can also take explicit actions to avoid the check, if necessary
  - These strategies are clunky, so it's nice when the compiler can make them unnecessary
- More information on this optimization can be found in the Chapel 1.23 release notes performance deck
  - And in Engin Kayraklioglu's CHIUW 2021 talk

CHAPEL: OPTIMIZATIONS

Automatic Copy Aggregation Optimization

- Enabled by '--auto-aggregation' compiler flag
- Can mitigate overhead of fine-grained communication by inserting aggregators
  - These aggregators batch up copies instead of performing them right away
  - Allows multiple values to be obtained from single remote access
    - Instead of paying cost of multiple remote accesses
  - Also helps with multiple accesses to same data (assuming it hasn't updated in the meanwhile)
    - Operating in a PGAS program can enable analysis to ensure only done when safe
    - In this case, that means knowing the code is operating in 'forall' loop, which ensures iterations are independent

- The motivation is similar to the principle behind Unordered Copy Optimization
  - Works when remote values aren't immediately needed

CHAPEL: OPTIMIZATIONS

Automatic Copy Aggregation Optimization

- One of the Arkouda code examples from earlier has an explicit aggregator in it:

```
forall (i, s) in zip(newSegs.domain, newSegs) with (var agg = newDstAggregator(int)) {
  agg.copy(esa[i+segStart], s);
}
```

- This example and other instances like it in Arkouda were a motivator behind adding this optimization
- And 2/3rds can be removed from the code as a result

- More information on this optimization can be found in Chapel 1.24 release notes performance deck
  - And in Engin Kayraklioglu's CHIUW 2021 talk


PERFORMANCE

CHAPEL: PERFORMANCE

- Traditionally, productive languages have tended to lag in performance
  - Oftentimes, productivity gains are so drastic that using them is considered worthwhile anyway
  - Languages like Python have been known to implement libraries in faster languages to sidestep this issue
    - Though this makes maintaining those libraries more difficult

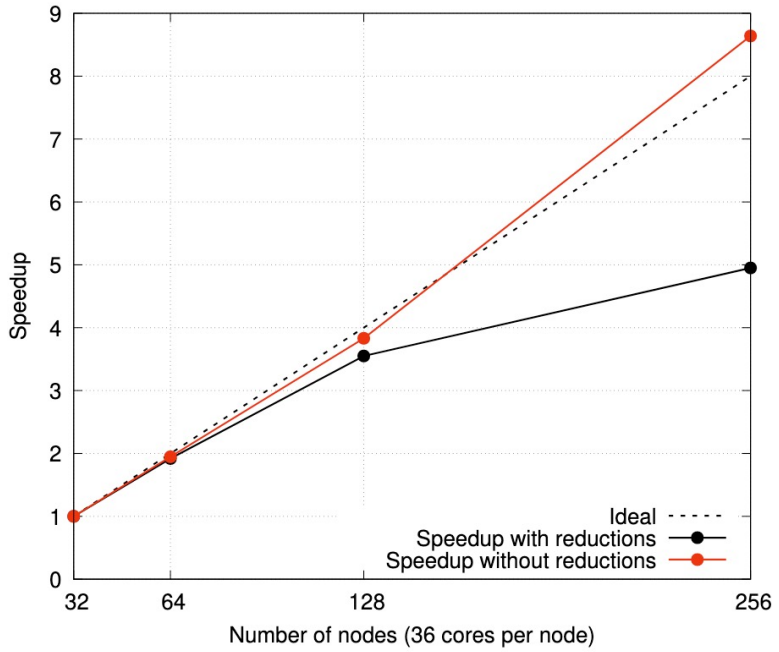
- Early in Chapel's life, we also were behind in that regard
- However, this has not been true for a while – Chapel now performs similarly to languages like C/Fortran
  - If you heard about Chapel during the old performance days and dismissed it, now is the time to look again!
- Chapel is also continually improving its performance

- Optimizations from previous section contribute to Chapel's great performance

CHAPEL: PERFORMANCE

CHAMPS Performance

- The following graphs were used to evaluate performance of CHAMPS as a CFD software
  - This graph covers strong scaling using a 3D Cartesian grid (with and without reductions)



(a) Strong scaling

Mesh is discretized with 800 elements in each direction (i,j,k)

CPU - Dual Intel Xeon E5-2695 v4 2.1 GHz (Broadwell), 36 cores per node

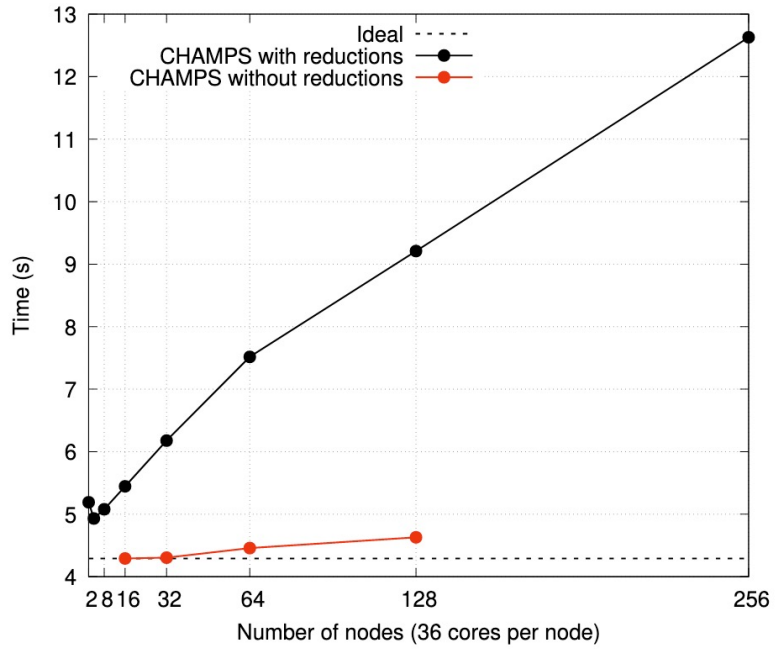
Memory - 128 GB DDR4-2400

– Development of Parallel CFD Applications with the Chapel Programming Language. Matthieu Parenteau, Simon Bourgault-Côté, Frédéric Plante, Engin Kayraklioglu, Éric Laurendeau. *AIAA Scitech 2021 Forum*. January 13, 2021.

CHAPEL: PERFORMANCE

CHAMPS Performance

- The following graphs were used to evaluate performance of CHAMPS as a CFD software
  - This graph covers weak scaling using a 3D Cartesian grid (with and without reductions)



(b) Weak scaling

Problem size is scaled with number of compute nodes to maintain roughly 1M cells per core

CPU - Dual Intel Xeon E5-2695 v4 2.1 GHz (Broadwell), 36 cores per node

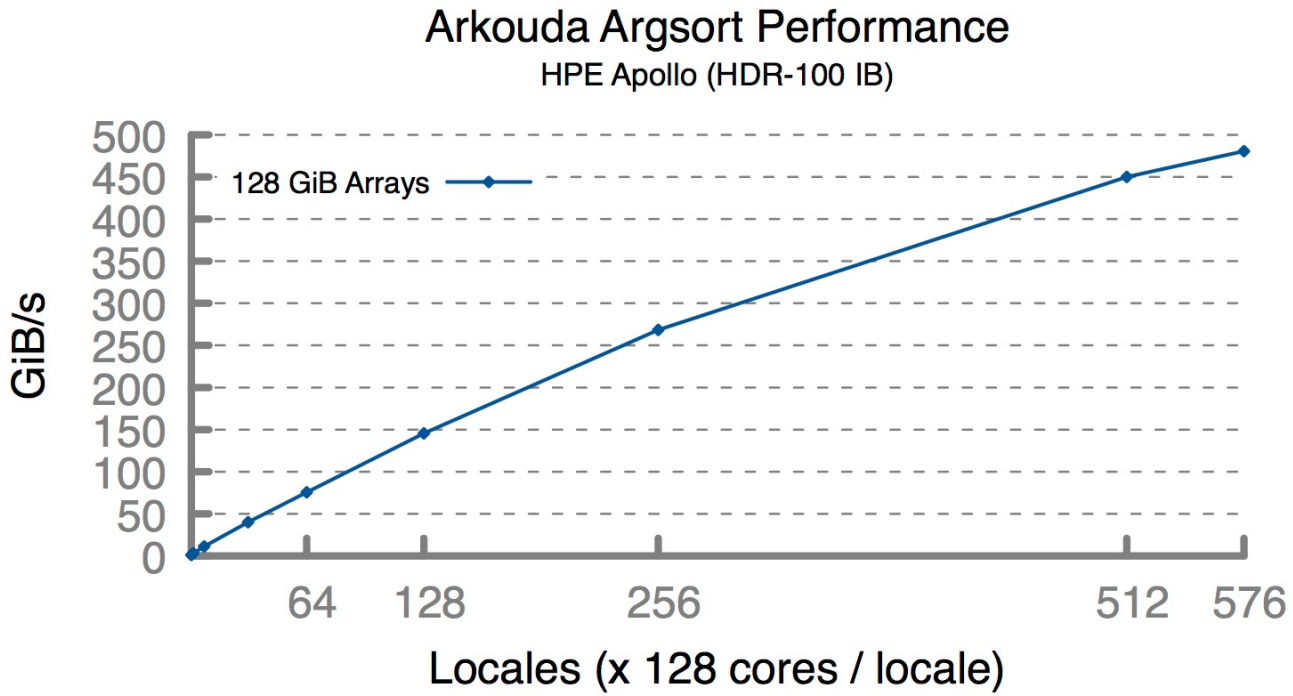
Memory - 128 GB DDR4-2400

– Development of Parallel CFD Applications with the Chapel Programming Language. Matthieu Parenteau, Simon Bourgault-Côté, Frédéric Plante, Engin Kayraklioglu, Éric Laurendeau. *AIAA Scitech 2021 Forum*. January 13, 2021.

CHAPEL: PERFORMANCE

Arkouda Performance

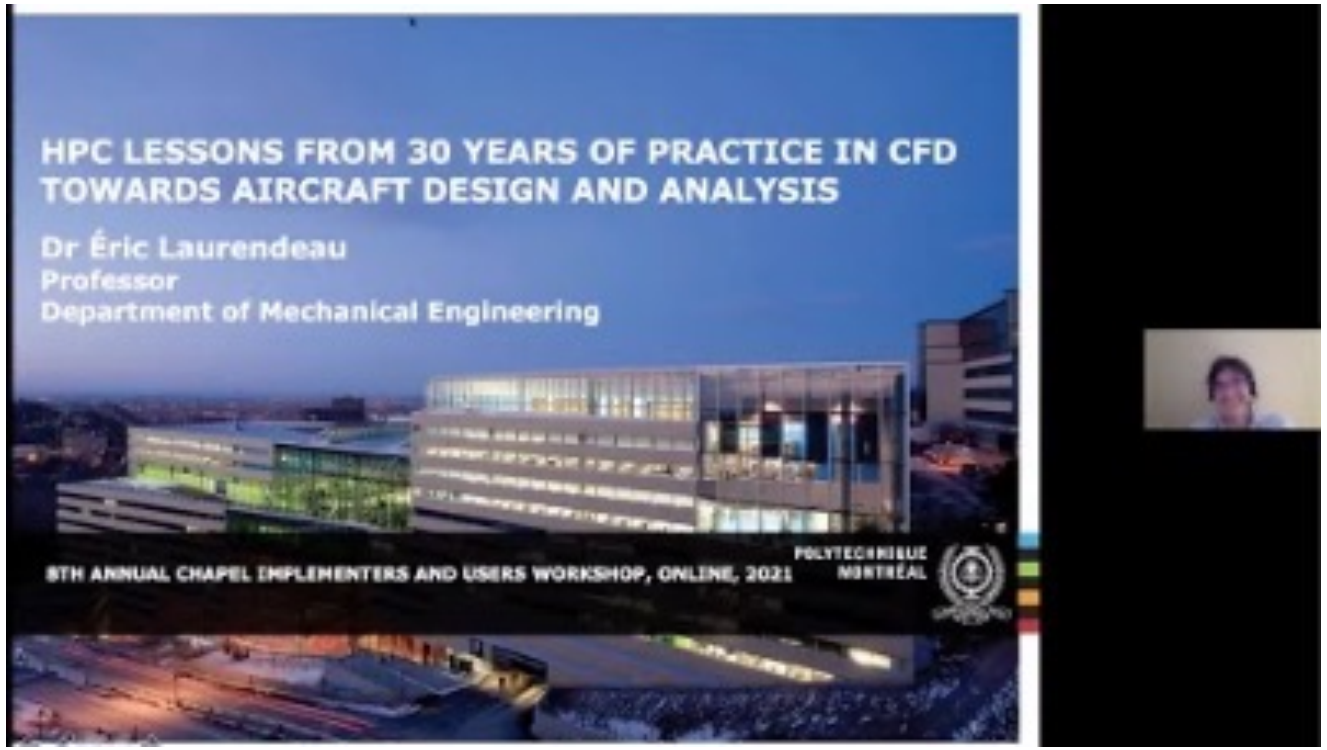
- Recent run performed on large in-house Apollo system
  - 72 TiB of 8-byte values
  - 480 GiB/s (2.5 minutes elapsed time)
  - used 73,728 cores of AMD Rome
  - ~100 lines of Chapel code (a snippet of Arkouda)




PRODUCTIVITY

CHAPEL: PRODUCTIVITY

- Productivity is difficult to measure precisely
  - Results are often anecdotal and cross language comparisons can be difficult to make fairly
  - That said, we have received a lot of positive feedback from a productivity perspective:



- Éric Laurendeau's talk: "CHIUW 2021 Keynote - HPC Lessons From 30 Years of Practice in CFD Towards Aircraft Design and Analysis" *Youtube*, uploaded by Chapel Parallel Programming Language, 11 June 2021 https://www.youtube.com/watch?v=wD-a_KyB8aI

CHAPEL: PRODUCTIVITY

- Arkouda developers express similar sentiments
  - This link is for Bill Reus's talk at NJIT's institute for Data Science in March: https://youtu.be/hzLbJF-fvjQ?t=1056

CHAPEL: PRODUCTIVITY

- Both CHAMPS and Arkouda are large projects that started in the last few years
  - CHAMPS had ~120 files and ~48k lines of code as of June 2021
  - Arkouda is currently at 52 files and ~15k lines of Chapel code in the ‘src’ directory as of July 2021

- Both projects proudly state that Chapel enabled them to develop their code so quickly
  - And are continuing to expand and develop using it

- Performance and productivity aren’t mutually exclusive

CONCLUSION

- PGAS programming models are beneficial when working on data intensive applications
  - They enable easier transfer of data between nodes
  - They reduce management burden by keeping all nodes contained within a single program

- Chapel is an excellent example of benefits of building PGAS concepts into a language
  - Having supporting syntax adds clarity and power
  - A backing compiler can perform optimizations the user would otherwise have to perform manually
    - Gives programs a higher baseline performance prior to explicit optimization efforts
    - Could reduce time spent hand-optimizing
  - Chapel is being used effectively and is starting to gain traction

- For your next data intensive application, why not give PGAS (and Chapel) a shot?

ACKNOWLEDGEMENTS

- Special thanks to Brad Chamberlain, Michelle Strout, Engin Kayraklioglu and the rest of the Chapel team for their help with making this talk and their work on what it covers
- And thanks to the CHAMPS team from Polytechnique Montreal for allowing me to use snippets of their code in this talk


THANK YOU –
QUESTIONS?
https://chapel-lang.org
@ChapelLanguage
lydia.duncan@hpe.com