

# WHAT'S NEW WITH CHAPEL: APPLICATIONS, AGGREGATORS, AND ACCELERATORS

Brad Chamberlain  
Charm++ 2021 workshop  
October 18, 2021

# WHAT IS CHAPEL?

---

**Chapel:** A modern parallel programming language

- portable & scalable
- open-source & collaborative



## Goals:

- Support general parallel programming
- Make parallel programming at scale far more productive



# CHAPEL, RELATIVE TO OTHER LANGUAGES

---

**Chapel strives 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]



# CHAPEL BENCHMARKS TEND TO BE CONCISE, CLEAR, AND COMPETITIVE

## STREAM TRIAD: C + MPI + OPENMP

```
#include <hpcc.h>
#include _OPENMP
#include 
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &commSize);
    MPI_Comm_rank(comm, &myRank);

    rv = HPCC_Stream(params, 0 == myRank);
    MPI_Reduce(&rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm);
    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;

    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );
    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    scalar = 3.0;

    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 1.0;
    }
    scalar = 3.0;

    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++)
        a[j] = b[j]*scalar*c[j];
}

HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );
HPCC_free(a);
HPCC_free(b);
HPCC_free(c);
return 0;
}
```

```
use BlockDist;

config const m = 1000,
      alpha = 3.0;
const Dom = {1..m} dmapped ...;
var A, B, C: [Dom] real;

B = 2.0;
C = 1.0;

A = B + alpha * C;
```

## HPCC RA: MPI KERNEL

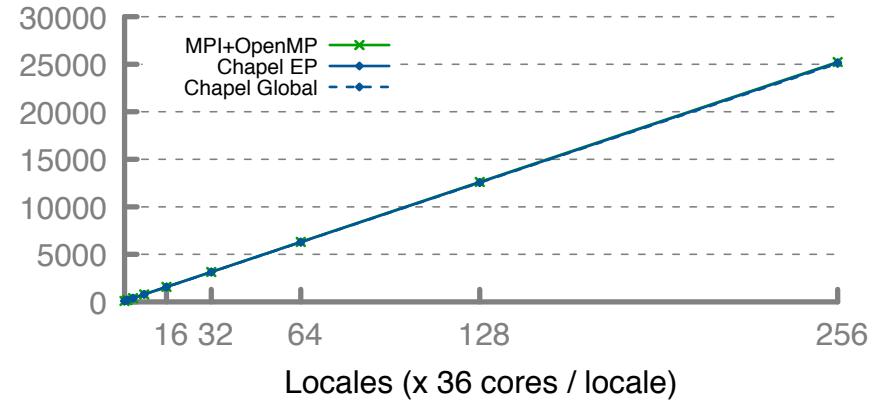
```
/* Perform updates to main table. The scalar equivalent is:
   for (i=0;i<tableSize;i++)
       for (j=0;j<tableSize;j++)
           if (table[i][j] > 0)
               table[i][j] += 1;
   */

MPI_Irecv(localRecvBuffer, localBufferSize, tparams.dtyped4,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, iinseq);
while (i < SendCount) {
    /* receive message */
    MPI_Recv(&status, 1, MPI_STATUS_IGNORE);
    if (status.MPI_TAG == UPDATE_TAG) {
        if (status.MPI_SOURCE == i) {
            if (status.MPI_TAG == UPDATE_TAG) {
                for (j = 0; j < localBufferSize; j++) {
                    imsg = LocalRecvbuffer[j];
                    locOffset = (imsg < tableSize ? imsg : 0) -
                               tableStartIndex;
                    HPMC_TableLocOffset = ~imsg;
                    if (imsg < tableSize) {
                        HPMC_InsertUpdate(locOffset, WhichPv, Buckets);
                        pendingUpdates++;
                    }
                }
            }
            if (status.MPI_TAG == UPDATE_TAG) {
                if (status.MPI_SOURCE == i) {
                    if (status.MPI_TAG == UPDATE_TAG) {
                        for (j = 0; j < localBufferSize; j++) {
                            imsg = LocalRecvbuffer[j];
                            locOffset = (imsg < tableSize ? imsg : 0) -
                                           tableStartIndex;
                            HPMC_TableLocOffset = ~imsg;
                            if (imsg < tableSize) {
                                HPMC_InsertUpdate(locOffset, WhichPv, Buckets);
                                pendingUpdates++;
                            }
                        }
                    }
                }
            }
        }
    }
    if (status.MPI_TAG == FINISHED_TAG) {
        if (status.MPI_SOURCE == i) {
            MPI_Waitall(iinseq, &status, MPI_STATUSES_IGNORE);
            MPI_Irecv(localRecvBuffer, localBufferSize, tparams.dtyped4,
                      MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, iinseq);
        }
    }
    while (haveDone < NumberReceiving) {
        if (pendingUpdates < maxPendingUpdates) {
            if (i < SendCount) {
                if (i < tableSize) {
                    if (table[i][i] > 0)
                        table[i][i] += 1;
                }
                MPI_Isend(&table[i][i], 1, MPI_Datatype,
                          MPI_COMM_WORLD, i, MPI_TAG);
            }
            else {
                MPI_Isend(&table[i][i], 1, MPI_Datatype,
                          MPI_COMM_WORLD, i, MPI_TAG);
            }
        }
        else {
            MPI_Waitall(iinseq, &status, MPI_STATUSES_IGNORE);
            MPI_Irecv(localRecvBuffer, localBufferSize, tparams.dtyped4,
                      MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, iinseq);
        }
    }
    i++;
}
i = 0;
while (i < tableSize) {
    if (table[i][i] > 0)
        table[i][i] += 1;
    i++;
}
```

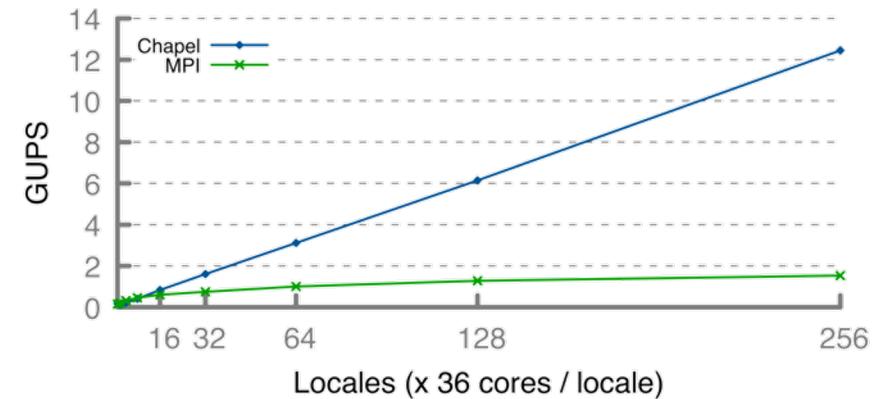
```
forall (_, r) in zip(Updates, RASTream()) do
    T[r & indexMask].xor(r);
```

GB/s

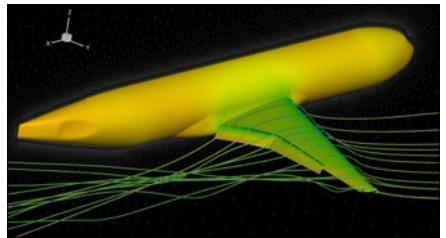
STREAM Performance (GB/s)



RA Performance (GUPS)

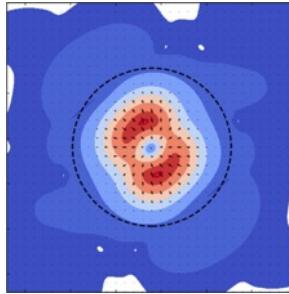


# NOTABLE CURRENT APPLICATIONS OF CHAPEL



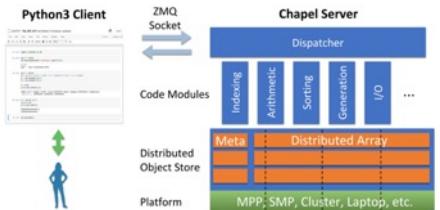
## CHAMPS: 3D Unstructured CFD

Éric Laurendeau, Simon Bourgault-Côté,  
Matthieu Parenteau, et al.  
École Polytechnique Montréal  
~48k lines of Chapel



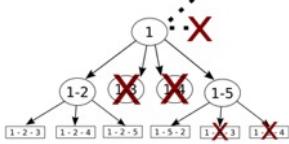
## ChplUltra: Simulating Ultralight Dark Matter

Nikhil Padmanabhan, J. Luna Zagorac,  
Richard Easter, et al.  
Yale University / University of Auckland



## Arkouda: NumPy at Massive Scale

Mike Merrill, Bill Reus, et al.  
US DoD  
~16k lines of Chapel



## ChOp: Chapel-based Optimization

Tiago Carneiro, Nouredine Melab, et al.  
INRIA Lille, France



## CrayAI: Distributed Machine Learning

Hewlett Packard Enterprise



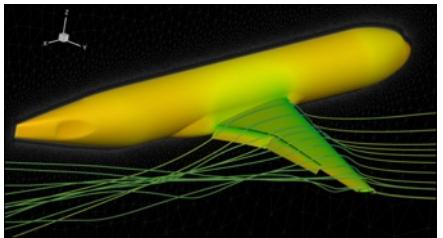
## Your Project Here?

# OUTLINE

- I. Context for Chapel
- II. Chapel Applications
- III. Basic Chapel Features
- IV. Aggregation in Chapel
- V. Chapel and Accelerators
- VI. Wrap-up

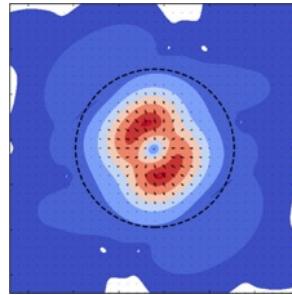


# NOTABLE CURRENT APPLICATIONS OF CHAPEL



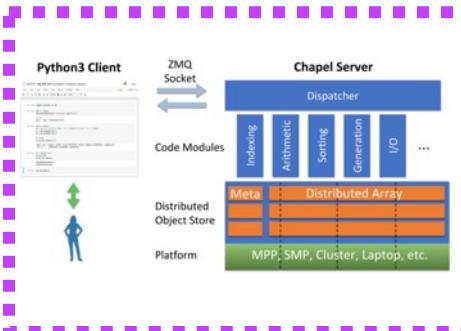
## CHAMPS: 3D Unstructured CFD

Éric Laurendeau, Simon Bourgault-Côté,  
Matthieu Parenteau, et al.  
École Polytechnique Montréal  
~48k lines of Chapel



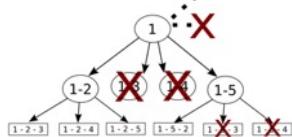
## ChplUltra: Simulating Ultralight Dark Matter

Nikhil Padmanabhan, J. Luna Zagorac,  
Richard Easter, et al.  
Yale University / University of Auckland



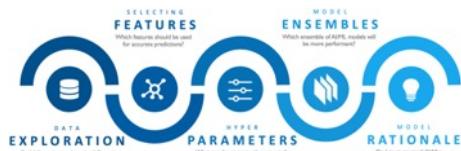
## Arkouda: NumPy at Massive Scale

Mike Merrill, Bill Reus, et al.  
US DoD  
~16k lines of Chapel



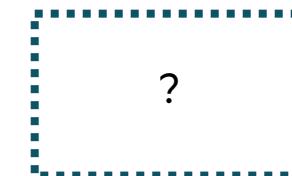
## ChOp: Chapel-based Optimization

Tiago Carneiro, Nouredine Melab, et al.  
INRIA Lille, France



## CrayAI: Distributed Machine Learning

Hewlett Packard Enterprise



## Your Project Here?

# ARKOUDA IN ONE SLIDE

## What is it?

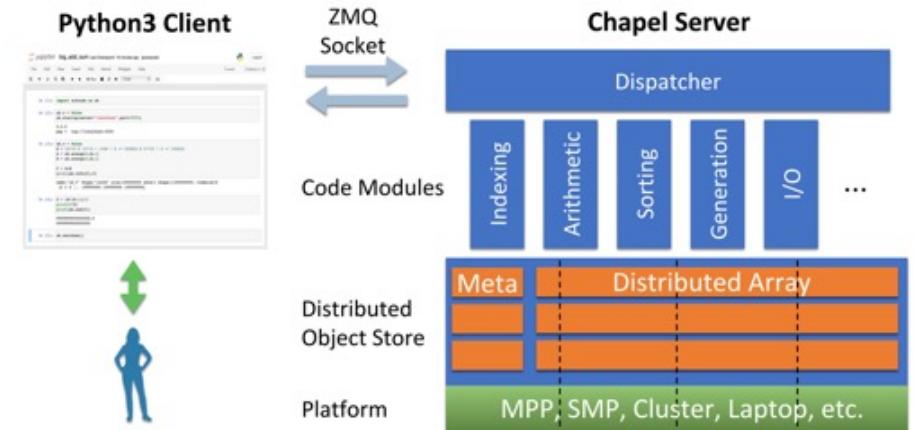
- A Python library supporting a key subset of NumPy and Pandas for Data Science
  - Computes massive-scale results within the human thought loop (seconds to minutes on multi-TB-scale arrays)
  - Uses a Python-client/Chapel-server model to get scalability and performance
- ~16k lines of Chapel, largely written in 2019, continually improved since then

## Who wrote it?

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

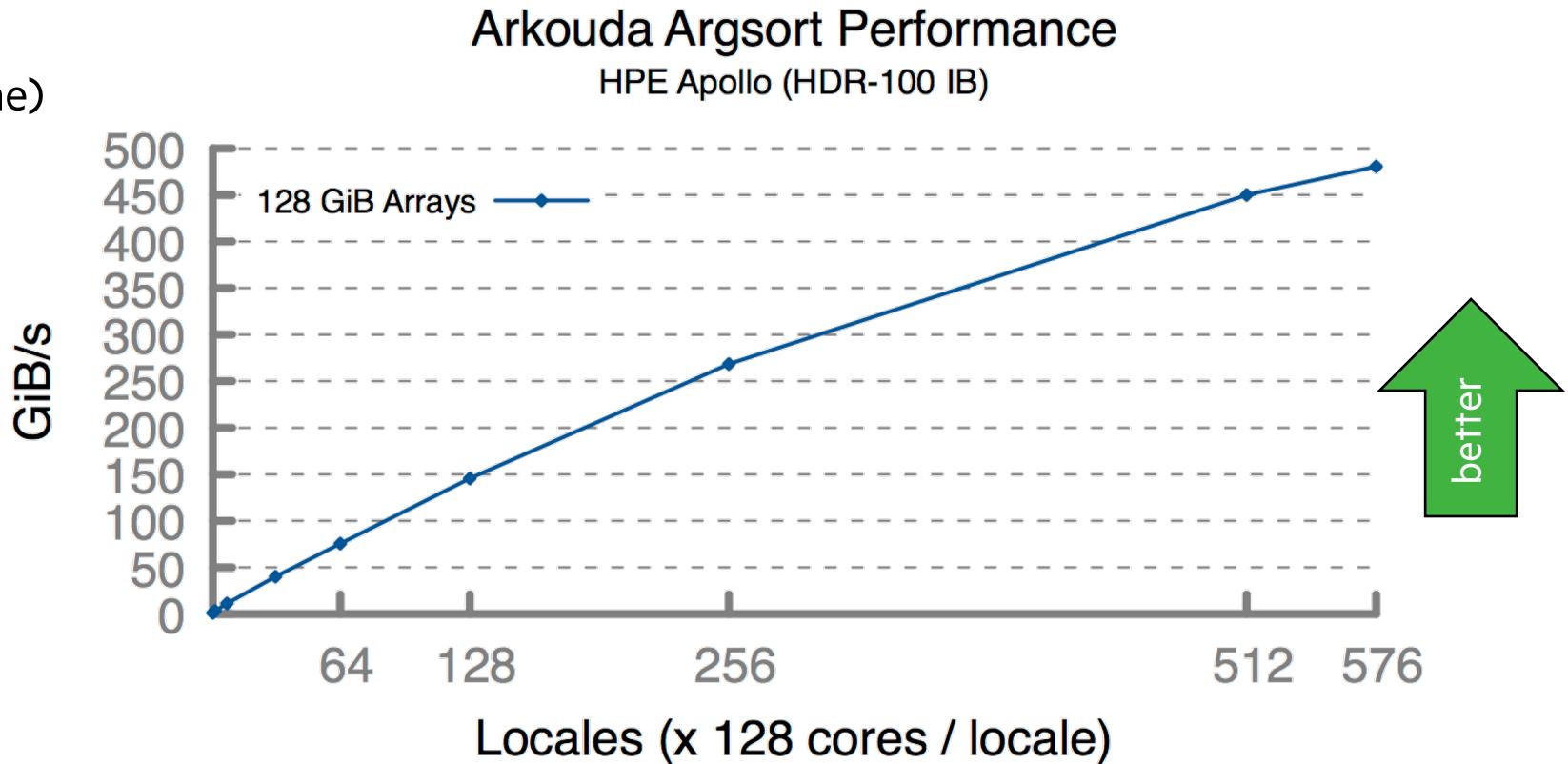
## Why Chapel?

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



# ARKOUDA ARGSORT: HERO RUN

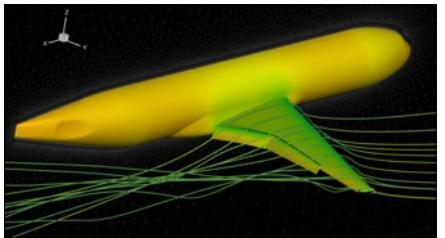
- Recent run performed on a large Apollo system
  - 72 TiB of 8-byte values
  - 480 GiB/s (2.5 minutes elapsed time)
  - used 73,728 cores of AMD Rome
  - ~100 lines of Chapel code



*Close to world-record performance—Quite likely a record for performance::lines of code*

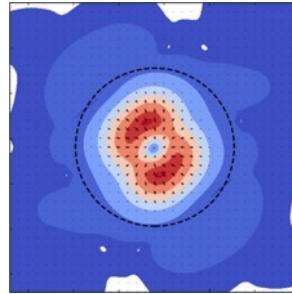


# NOTABLE CURRENT APPLICATIONS OF CHAPEL



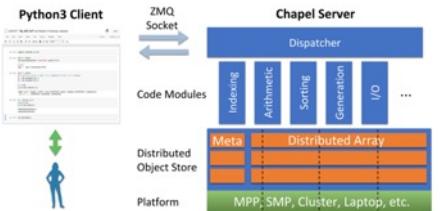
## CHAMPS: 3D Unstructured CFD

Éric Laurendeau, Simon Bourgault-Côté,  
Matthieu Parenteau, et al.  
École Polytechnique Montréal  
~48k lines of Chapel



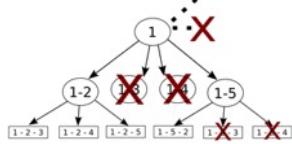
## ChplUltra: Simulating Ultralight Dark Matter

Nikhil Padmanabhan, J. Luna Zagorac,  
Richard Easter, et al.  
Yale University / University of Auckland



## Arkouda: NumPy at Massive Scale

Mike Merrill, Bill Reus, et al.  
US DoD  
~16k lines of Chapel



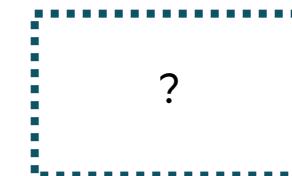
## ChOp: Chapel-based Optimization

Tiago Carneiro, Nouredine Melab, et al.  
INRIA Lille, France



## CrayAI: Distributed Machine Learning

Hewlett Packard Enterprise

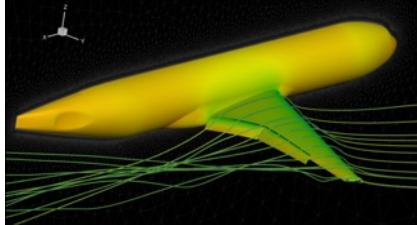


## Your Project Here?

# CHAMPS SUMMARY

## What is it?

- 3D unstructured CFD framework for airplane simulation
- ~48k lines of Chapel written from scratch in ~2 years



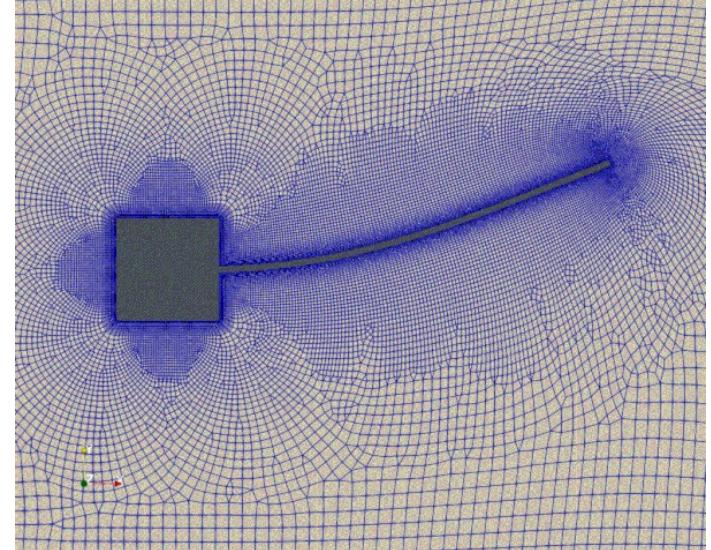
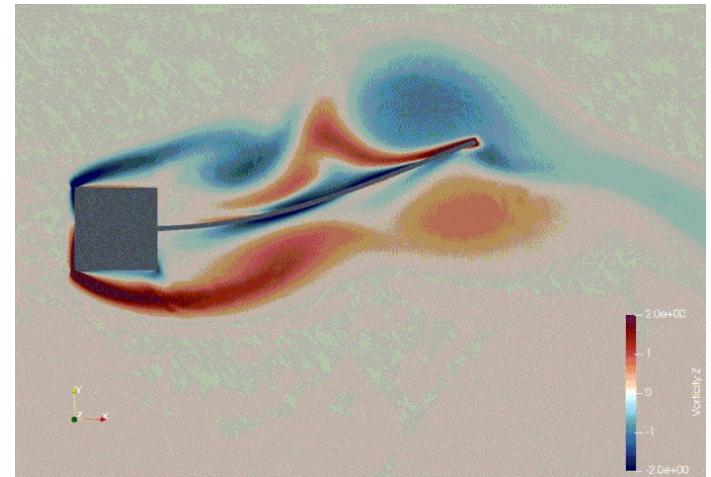
## Who wrote it?

- Professor Éric Laurendeau's team at Polytechnique Montreal



## Why Chapel?

- performance and scalability competitive with MPI + C++
- students found it far more productive to use



# CHAMPS 2021 HIGHLIGHTS

- Presented at CASI/IASC Aero 21 Conference
  - Participated in 1<sup>st</sup> AIAA Ice Prediction Workshop
  - Participating in 4<sup>th</sup> AIAA CFD High-lift Prediction Workshop
  - Student presentation to CFD Society of Canada (CFDSC)
- 
- **Achieving large-scale, high-quality results comparable to other major players in industry, government, academia:**
    - e.g., Boeing, Lockheed Martin, NASA, JAXA, Georgia Tech, ...



The screenshot shows a presentation slide with a red header bar containing the text "What is Chapel and why use it?". Below the header, there is a section titled "Challenges of multi-physics simulations" which lists three points: "the fidelity of multiple solvers;", "the performances → computational costs;", and "the productivity → addition of multiple physical models.". Another section titled "Benefits from Chapel's features [6]" lists five points: "Productivity → fast prototyping with high level syntax;", "Natively distributed → Overcome the barrier of entry of parallel distributed programming in an academic context (2 years);", "Modularity → Generic classes and records to reuse structures;", and "Memory management strategies.". The footer of the slide includes the text "IPW1 – Polytechnique Montréal" and "4/39".

# CHAMPS: EXCERPT FROM ERIC'S CHIUW 2021 KEYNOTE

## HPC Lessons From 30 Years of Practice in CFD Towards Aircraft Design and Analysis

*"To show you what Chapel did in our lab... [NSCODE, our previous framework] ended up 120k lines. And my students said, 'We can't handle it anymore. It's too complex, we lost track of everything.' And today, they went **from 120k lines to 48k lines, so 3x less.***

*But the code is not 2D, it's 3D. And it's not structured, it's unstructured, which is way more complex. And it's multi-physics: aeroelastic, aero-icing. **So, I've got industrial-type code in 48k lines.***

*So, for me, this is like the proof of the benefit of Chapel, **plus the smiles I have on my students everyday in the lab because they love Chapel as well.** So that's the key, that's the takeaway.*

*[Chapel] promotes the programming efficiency ... **We ask students at the master's degree to do stuff that would take 2 years and they do it in 3 months.** So, if you want to take a summer internship and you say, 'program a new turbulence model,' well they manage. And before, it was impossible to do."*

- Talk available online: [https://youtu.be/wD-a\\_KyB8al?t=1904](https://youtu.be/wD-a_KyB8al?t=1904) (hyperlink jumps to the section quoted here)



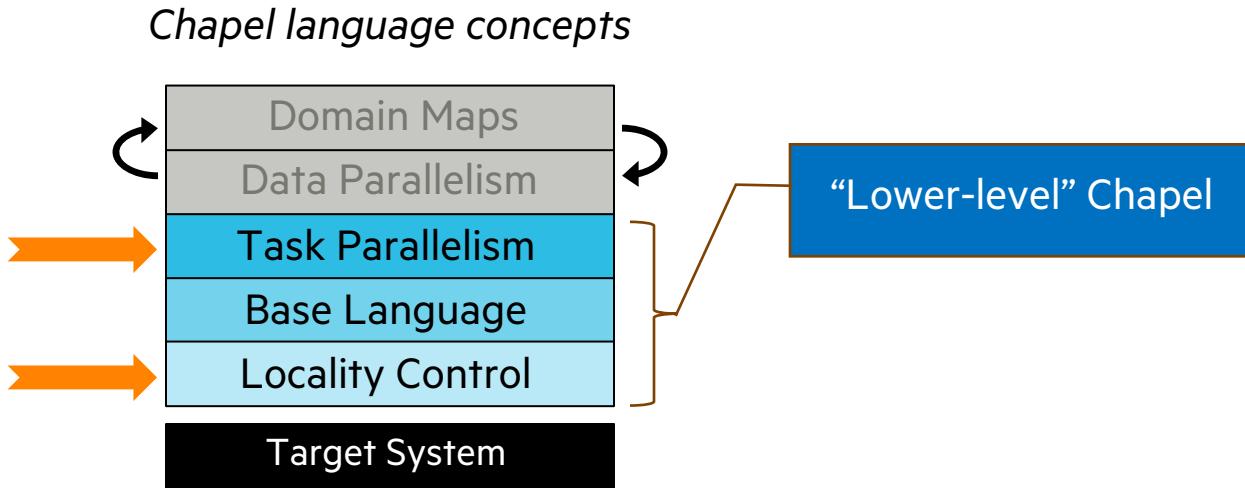
POLYTECHNIQUE  
MONTRÉAL

# **BASIC CHAPEL FEATURES**



# CHAPEL’S “LOWER-LEVEL” FEATURES

---



# CHAPEL TERMINOLOGY: LOCALES

- **Locales:** a unit of the target architecture that can run tasks and store variables
  - Think “compute node” on a parallel system
  - User specifies number of locales on executable’s command-line

```
prompt> ./myChapelProgram --numLocales=4      # or '-nl 4'
```

**Locales** array :



User's code starts running as a single task on locale 0

# TASK-PARALLEL “HELLO WORLD”

helloTaskPar.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
    writef("Hello from task %n of %n on %s\n",
           tid, numTasks, here.name);
```

# TASK-PARALLEL “HELLO WORLD”

helloTaskPar.chpl

```
const numTasks = here.numPUs();  
coforall tid in 1..numTasks do  
    writef("Hello from task %n or %n on %s\n",  
          tid, numTasks, here.name);
```

‘here’ refers to the locale on which we’re currently running

how many processing units (think “cores”) does my locale have?

what’s my locale’s name?

# TASK-PARALLEL “HELLO WORLD”

helloTaskPar.chpl

```
const numTasks = here.numPUs();  
coforall tid in 1..numTasks do ——————>  
    writef("Hello from task %n of %n on %s\n",  
          tid, numTasks, here.name);
```

a 'coforall' loop executes each iteration as an independent task

```
prompt> chpl helloTaskPar.chpl  
prompt> ./helloTaskPar  
Hello from task 1 of 4 on n1032  
Hello from task 4 of 4 on n1032  
Hello from task 3 of 4 on n1032  
Hello from task 2 of 4 on n1032
```

# TASK-PARALLEL “HELLO WORLD”

helloTaskPar.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
    writef("Hello from task %n of %n on %s\n",
           tid, numTasks, here.name);
```

```
prompt> chpl helloTaskPar.chpl
prompt> ./helloTaskPar
Hello from task 1 of 4 on n1032
Hello from task 4 of 4 on n1032
Hello from task 3 of 4 on n1032
Hello from task 2 of 4 on n1032
```

**So far, this is a shared-memory program**

Nothing refers to remote locales,  
explicitly or implicitly

# TASK-PARALLEL “HELLO WORLD”

helloTaskPar.chpl

```
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
    writef("Hello from task %n of %n on %s\n",
           tid, numTasks, here.name);
```

# TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)

helloTaskPar.chpl

```
coforall loc in Locales {
    on loc {
        const numTasks = here.numPUs();
        coforall tid in 1..numTasks do
            writef("Hello from task %n of %n on %s\n",
                   tid, numTasks, here.name);
    }
}
```

# TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)

```
helloTaskPar.chpl
```

```
coforall loc in Locales {  
    on loc {  
        const numTasks = here.numPUs();  
        coforall tid in 1..numTasks do  
            writef("Hello from task %n of %n on %s\n",  
                  tid, numTasks, here.name);  
    }  
}
```

create a task per locale  
on which the program is running

have each task run ‘on’ its locale

then print a message per core,  
as before

```
prompt> chpl helloTaskPar.chpl  
prompt> ./helloTaskPar -numLocales=4  
Hello from task 1 of 4 on n1032  
Hello from task 4 of 4 on n1032  
Hello from task 1 of 4 on n1034  
Hello from task 2 of 4 on n1032  
Hello from task 1 of 4 on n1033  
Hello from task 3 of 4 on n1034  
Hello from task 1 of 4 on n1035  
...
```



**CHAPEL AGGREGATORS**

# BALE INDEX GATHER KERNEL IN CHAPEL: NAÏVE VERSION

---

```
// Naive index gather
forall (d, i) in zip(Dst, Inds) do
    d = Src[i];
```

‘Src’ is a distributed array with  
*numEntries* elements

‘Dst’ and ‘Inds’ are distributed arrays with  
*numUpdates* elements

# BALE INDEX GATHER KERNEL IN CHAPEL: NAÏVE VERSION

```
// Naive index gather
forall (d, i) in zip(Dst, Inds) do
    d = Src[i];
```

Gets lowered roughly to...

```
coforall loc in Dst.targetLocales do
    on loc do
        coforall tid in 0..<here.numPUs() do
            for idx in myInds(loc, tid, ...) do
                D[idx] = Src[Inds[idx]];
```

A concurrent loop over the compute nodes

A nested concurrent loop over each node's cores

A serial loop to compute each task's chunk of gathers

# BALE INDEX GATHER KERNEL IN CHAPEL: NAÏVE VERSION

```
// Naive index gather
forall (d, i) in zip(Dst, Inds) do
    d = Src[i];
```

Gets lowered roughly to...

```
coforall loc in Dst.targetLocales do
    on loc do
        coforall tid in 0..<here.numPUs() do
            for idx in myInds(loc, tid, ...) do
                D[idx] = Src[Inds[idx]];
```

But, for a parallel loop with no data dependencies,  
why perform these high-latency operations serially?

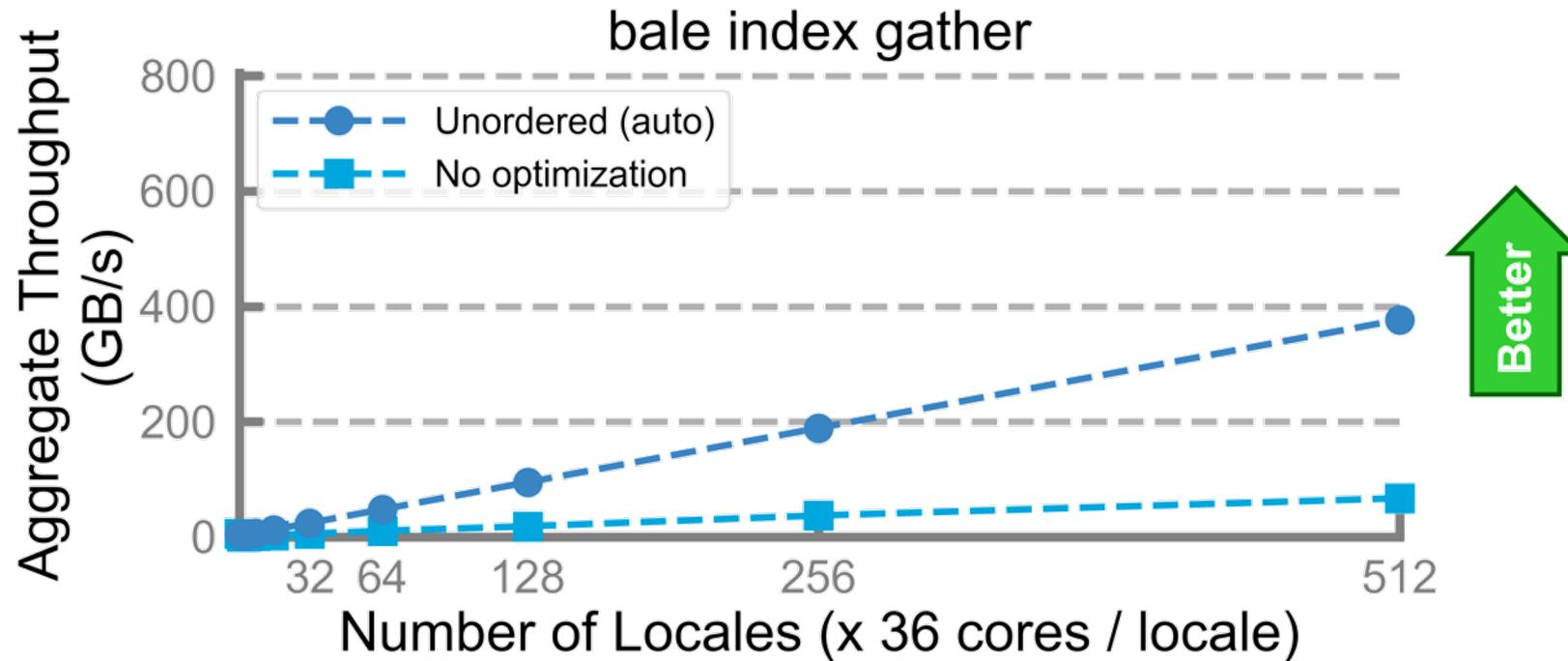
```
for idx in myInds(loc, tid, ...) do
    unorderedCopy(D[idx], Src[Inds[idx]]);
    unorderedCopyTaskFence();
```

So, our compiler rewrites the inner loop  
to perform the ops asynchronously

- Implemented by Michael Ferguson and Elliot Ronaghan, 2019

# BALE INDEX GATHER KERNEL IN CHAPEL: NAÏVE VERSION

```
// Naive index gather
forall (d, i) in zip(Dst, Inds) do
    d = Src[i];
```



# BALE INDEX GATHER KERNEL IN CHAPEL: AGGREGATOR VERSION

```
use CopyAggregation;
```

‘use’ the module providing the aggregators

```
// Aggregated index gather
```

```
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do  
    agg.copy(d, Src[i]);
```

‘with (var ...)’ creates a variable per task  
that’s executing the ‘forall’ loop

To use it, we simply replace  
the assignment with ‘agg.copy’

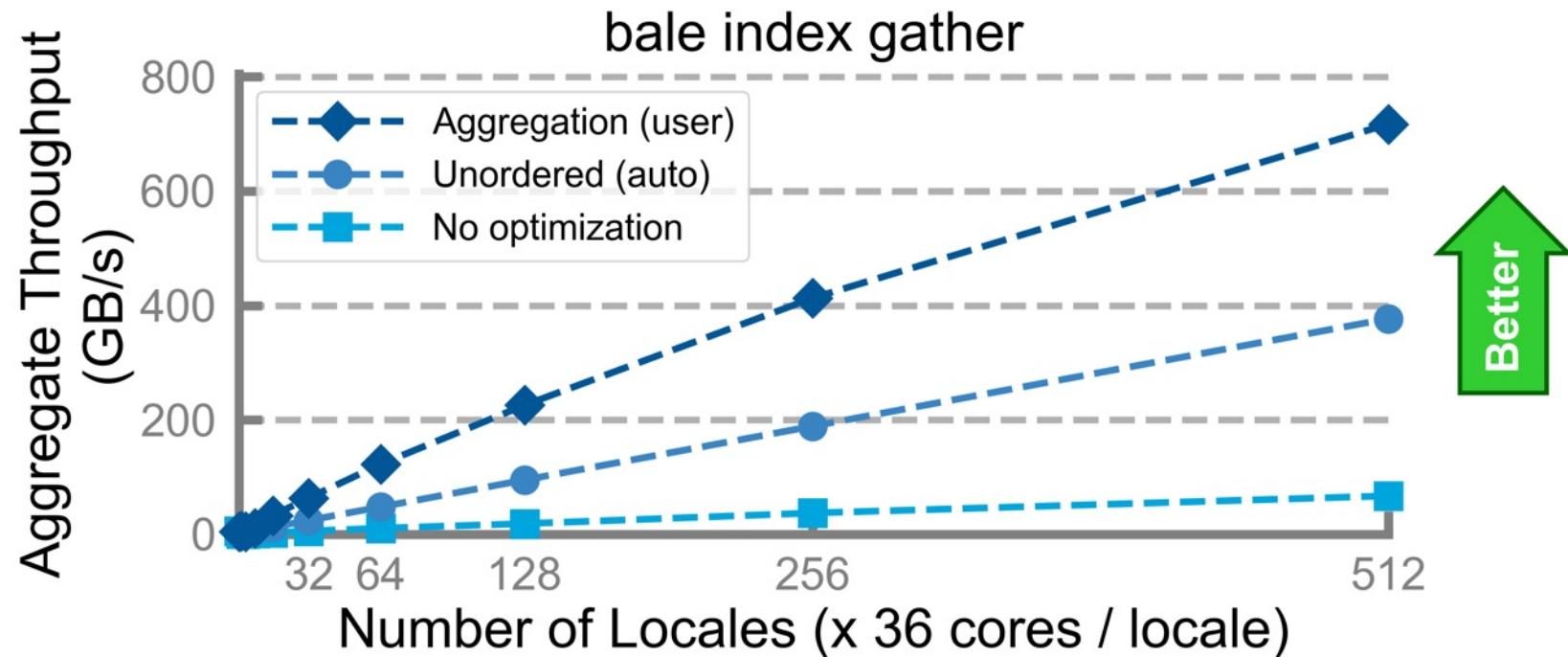
Here, we’re giving each task a “source aggregator”, *agg*,  
which aggregates remote ‘gets’ locally, then performs them

As the aggregator’s buffers fill up, it communicates the operations  
to the remote locale, automatically and asynchronously

# BALE INDEX GATHER KERNEL IN CHAPEL: AGGREGATOR VERSION

```
use CopyAggregation;

// Aggregated index gather
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
    agg.copy(d, Src[i]);
```



# IMPLEMENTING CHAPEL'S AGGREGATORS

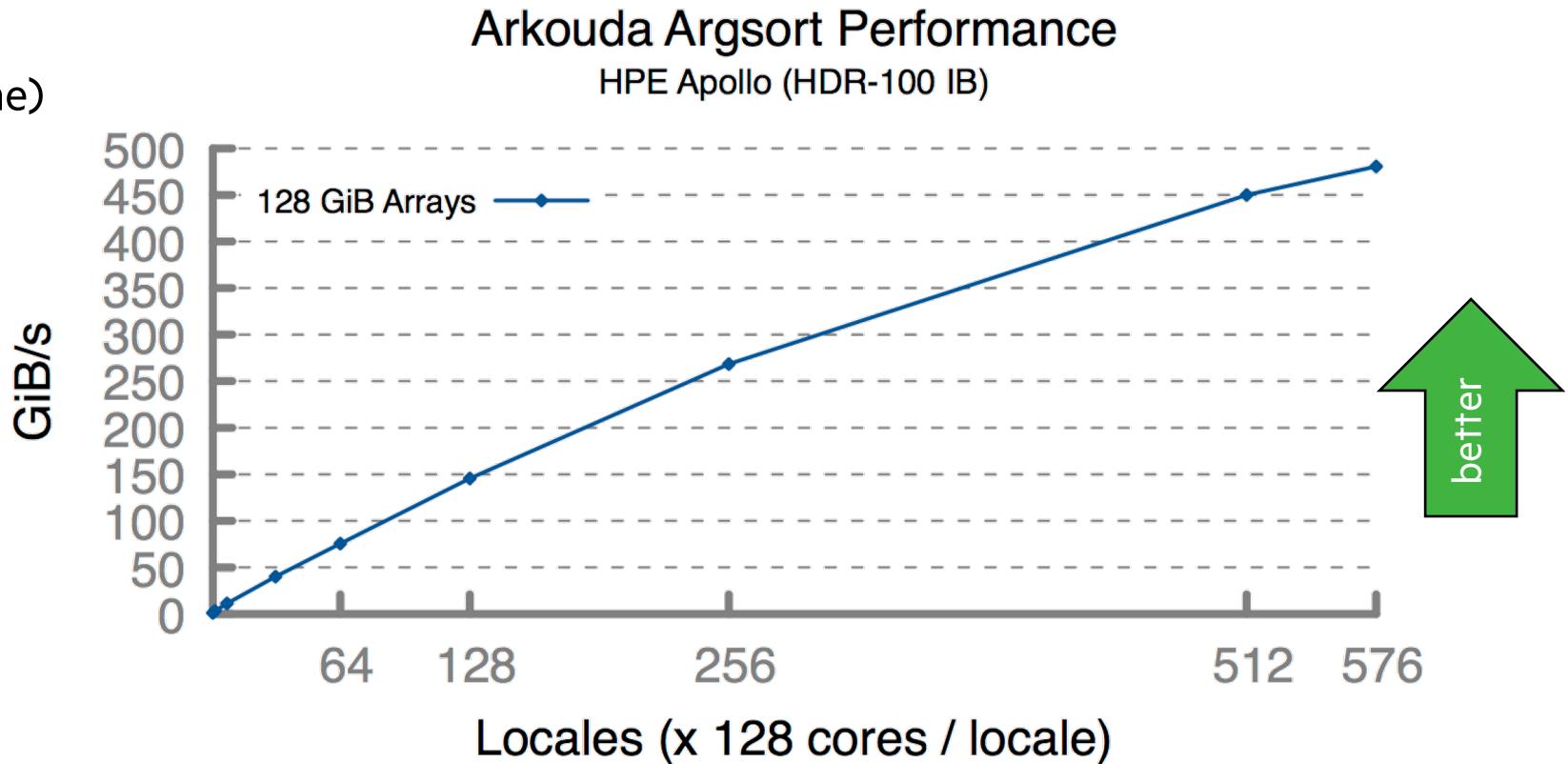
---

- Chapel's aggregators are implemented as Chapel source code
  - no language or compiler changes were required
  - ~100 lines of reasonably straightforward code to implement 'SrcAggregator' used here
    - (~420 lines for the entire 'CopyAggregation' module)
- Developed by Elliot Ronaghan, 2020–present



# ARKOUDA ARGSORT: HERO RUN

- Recent hero run performed on a large 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



Aggregators have been key to getting results like these



# CAN WE AUTOMATE AGGREGATION?

**Q:** Is there an opportunity for the compiler to introduce aggregators automatically?

```
// Naive index gather
forall (d, i) in zip(Dst, Inds) do
    d = Src[i];
```

user writes straightforward code  
compiler optimizes as:

```
use CopyAggregation;

// Aggregated index gather
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
    agg.copy(d, Src[i]);
```

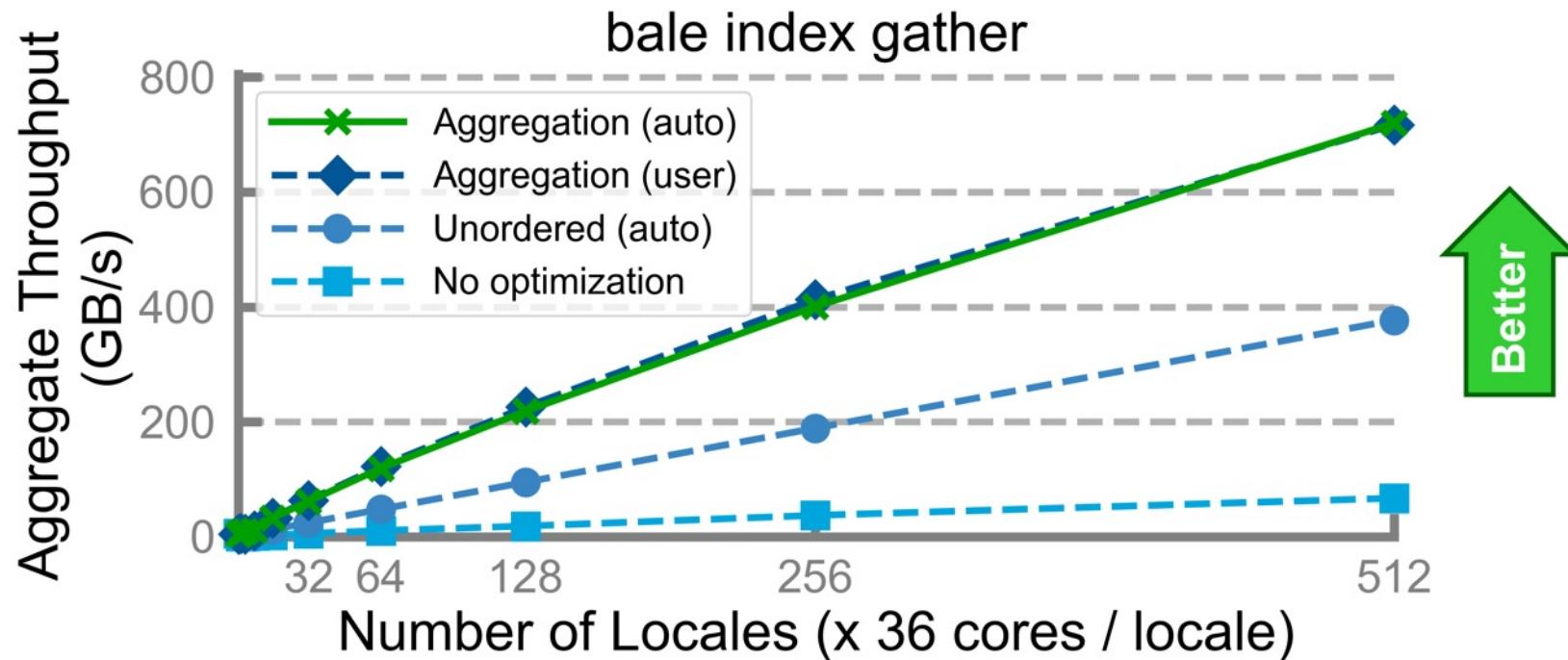
**A:** In many cases, yes

- developed by Engin Kayraklıoglu, 2021
- combines previous ‘unordered’ analysis with a new locality analysis of RHS/LHS expressions
- for details, see Engin’s LCP 2021 paper: <https://lcpc2021.github.io/>

# AUTO-AGGREGATION: IMPACT

- As a result, the naïve version can now compete with the user-written aggregators

```
// Naive index gather
forall (d, i) in zip(Dst, Inds) do
    d = Src[i];
```



**CHAPEL ON GPUS**



# THE CASE FOR CHAPEL ON GPUS

---

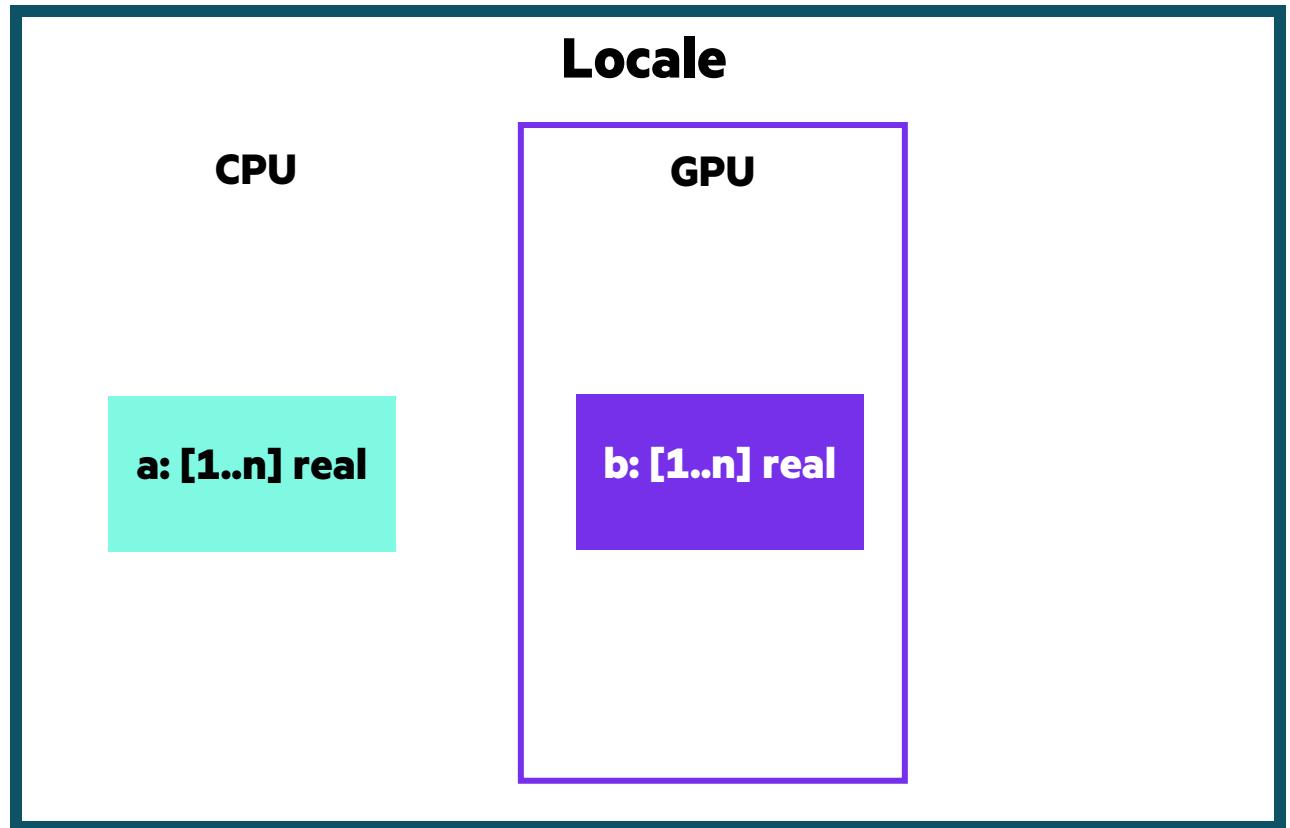
- “any parallel algorithm on any parallel architecture”
  - yet, Chapel has not supported compilation to GPUs—an obvious important case for many HPC systems
- Related efforts:
  - Albert Sidelnik et al. (UIUC), **Performance portability with the Chapel language**, IPDPS 2012
  - Brad Chamberlain, **Chapel Support for Heterogeneous Architectures via Hierarchical Locales**, PGAS-X 2012
  - Mike Chu et al. (AMD), **various works**, CHI UW 2015–2018
  - Akihiro Hayasi et al. (Georgia Tech), **various works**, CHI UW 2019–present
- Users *have* used Chapel with GPUs through interoperating with kernels written in CUDA, OpenCL, ...
  - e.g., the CHAMPS and ChOp applications do this
- Yet, Chapel’s features for parallelism and locality are a good match for GPUs
  - code generation has been the major sticking point
  - we’re currently leveraging our LLVM-based back-end to address this



# HIERARCHICAL LOCALES: A NOTIONAL CPU+GPU LOCALE MODEL

- A simple ‘gpu’ locale model might have a sub-locale for the GPU

```
var a: [1..n] real;  
  
on here.GPU {  
    var b: [1..n] real;  
    ...  
}
```



# GPUS: NOTIONAL GOAL

---

## A Sample GPU Computation, notionally:

```
on here.GPU {  
    var A = [1, 2, 3, 4, 5];  
    forall a in A do  
        a += 5;  
}
```

# GPUS: SIX MONTHS AGO

## A Sample GPU Computation, as of Chapel 1.24:

```
pragma "codegen for GPU"
export proc add_nums(A: c_ptr(real(64))) {
    A[0] = A[0]+5;
}

var funcPtr = createFunction();
var A = [1, 2, 3, 4, 5];
__primitive("gpu kernel launch", funcPtr,
           <grid and block size>,...,
           c_ptrTo(A), ...);

writeln(A);
```

```
extern {
#define FATBIN_FILE "chpl__gpu.fatbin"
double createFunction() {
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                       "add_nums"); }

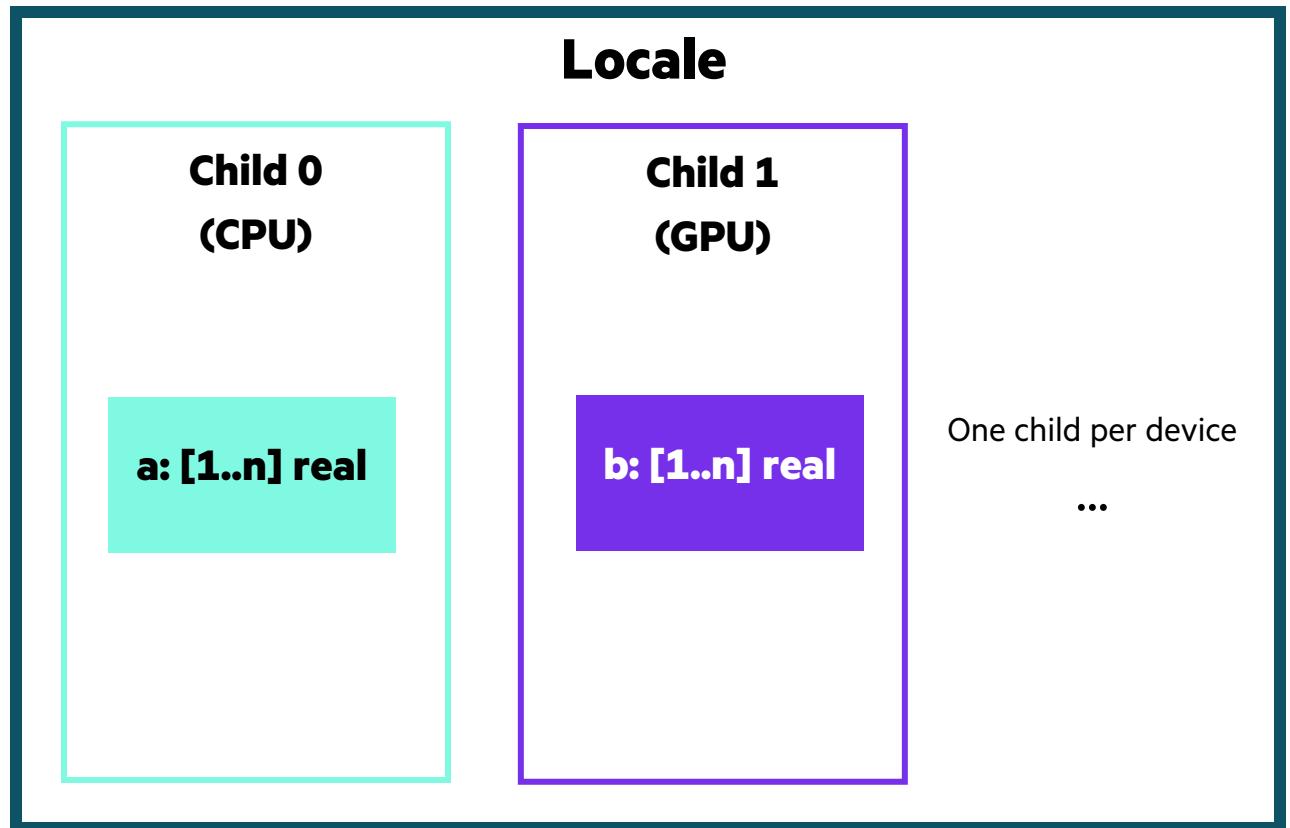
}
```

Read  
fat binary  
and create a  
CUDA  
function

# GPUS: TODAY

## A Sample GPU Computation, in Chapel 1.25:

```
on here.getChild(1) {  
    var A = [1, 2, 3, 4, 5];  
    forall a in A do  
        a += 5;  
}
```

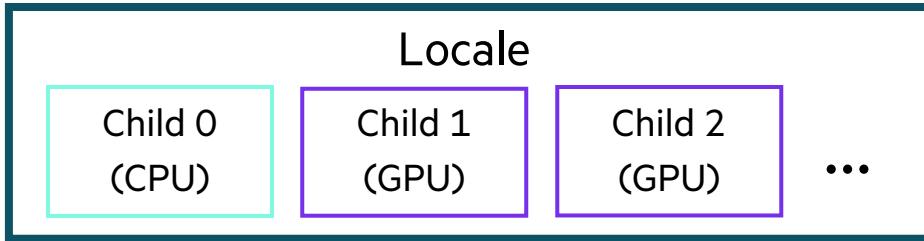


- developed by Engin Kayraklıoglu, Andy Stone, and David Iten

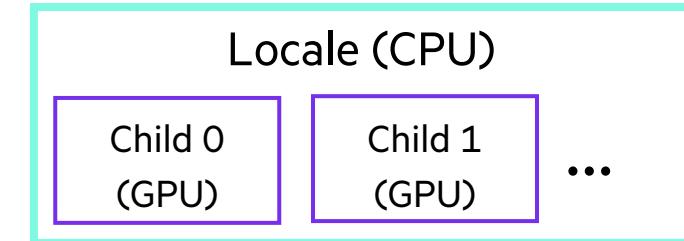


# ALTERNATIVE GPU LOCALE MODELS

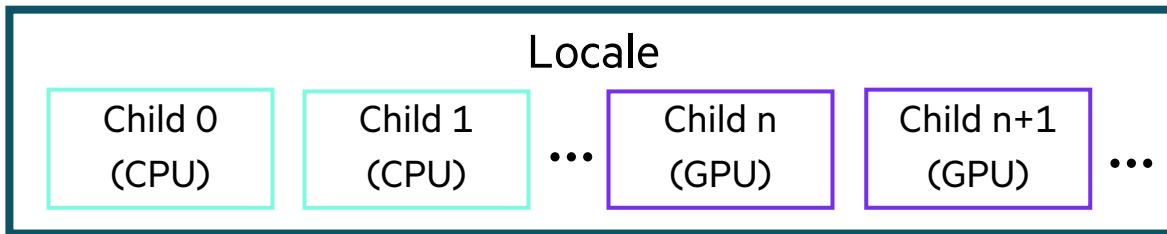
## What we have now (sub-locale 0 = CPU)



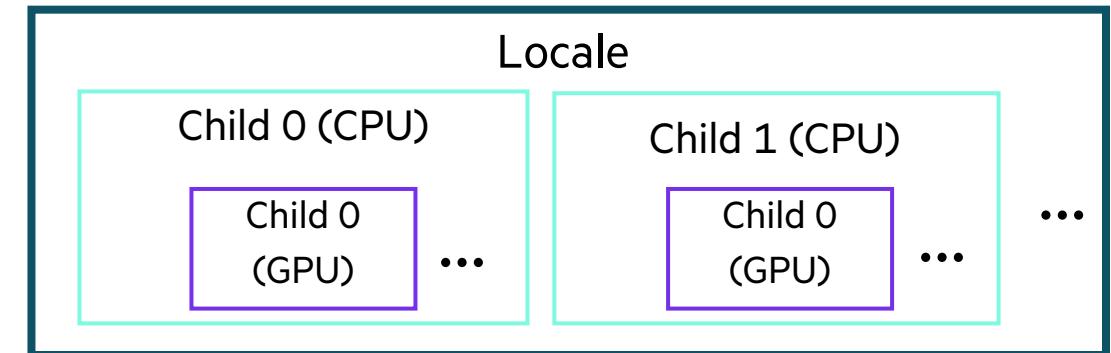
## Locale for CPU; sub-locales for GPUs



## NUMA-aware (flat)



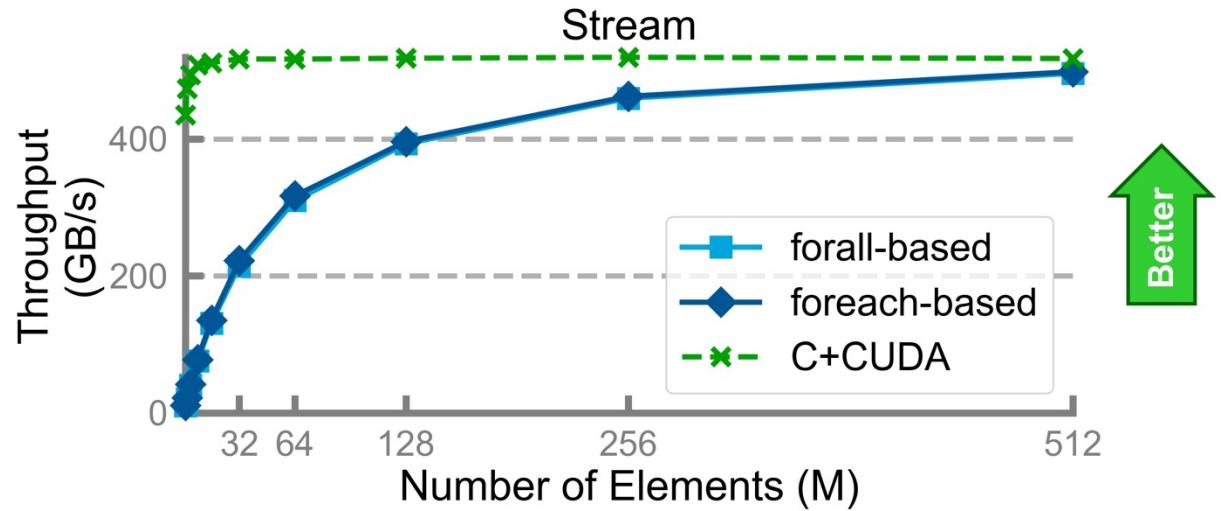
## NUMA-aware (hierarchical)



# GPUs: INITIAL PERFORMANCE STUDY

**HPCC Stream:** very few changes needed to our typical Stream code to target GPUs

```
on here.getChild(1) {  
    var A, B, C: [1..n] real;  
    const alpha = 2.0;  
  
    forall b in B do b = 1.0;  
    forall c in C do c = 2.0;  
  
    forall a, b, c in zip(A, B, C) do  
        a = b + alpha * c;  
}
```



## GPUS: NEXT STEPS

---

- Plenty of housecleaning, refactoring, streamlining, etc.
- Language design issues
- Further performance analysis and optimization
- Support richer and more flexible styles of programming
- Support a richer model of memory and inter-device data transfers (today: unified memory only)
- Support a wider variety of vendors (today: Nvidia only)



# **WRAP-UP**



# CHAPEL RESOURCES

**Chapel homepage:** <https://chapel-lang.org>

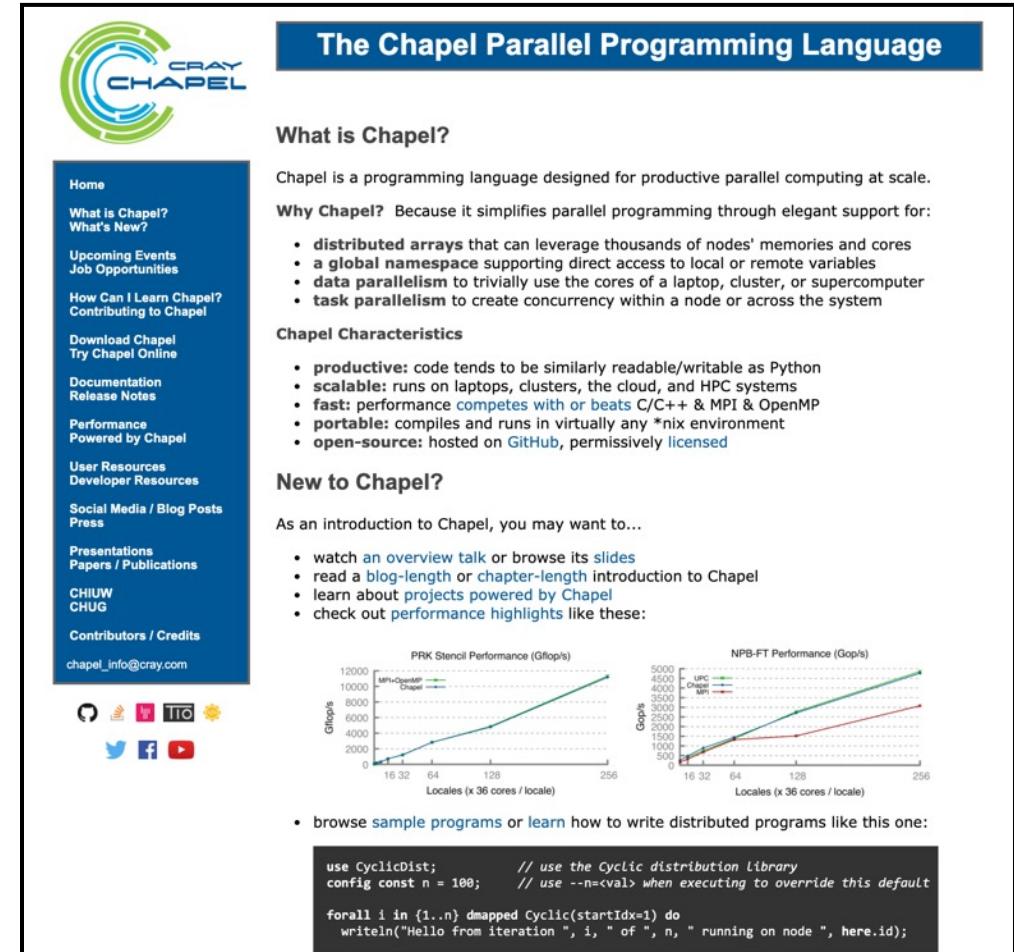
- (points to all other resources)

## Social Media:

- Twitter: [@ChapelLanguage](#)
- Facebook: [@ChapelLanguage](#)
- YouTube: <http://www.youtube.com/c/ChapelParallelProgrammingLanguage>

## Community Discussion / Support:

- Discourse: <https://chapel.discourse.group/>
- Gitter: <https://gitter.im/chapel-lang/chapel>
- Stack Overflow: <https://stackoverflow.com/questions/tagged/chapel>
- GitHub Issues: <https://github.com/chapel-lang/chapel/issues>



The screenshot shows the official Chapel website homepage. At the top right, a blue header bar reads "The Chapel Parallel Programming Language". To the left of the header is the Cray Chapel logo, which consists of a stylized green and blue circular graphic followed by the word "CRAY" above "CHAPEL". The main content area has a dark blue sidebar on the left containing links to various sections: Home, What is Chapel? (with sub-links What's New?, Upcoming Events, Job Opportunities), How Can I Learn Chapel? (Contributing to Chapel), Download Chapel (Try Chapel Online), Documentation (Release Notes), Performance (Powered by Chapel), User Resources (Developer Resources), Social Media / Blog Posts (Press), Presentations (Papers / Publications), CHI'16, CHUG, and Contributors / Credits. Below the sidebar is an email address: chapel\_info@cray.com. At the bottom of the sidebar are social media icons for GitHub, YouTube, LinkedIn, and others. The main body of the page is titled "What is Chapel?" and contains a brief introduction: "Chapel is a programming language designed for productive parallel computing at scale." It also includes a section titled "Why Chapel?" explaining its purpose: "Because it simplifies parallel programming through elegant support for: distributed arrays that can leverage thousands of nodes' memories and cores; a global namespace supporting direct access to local or remote variables; data parallelism to trivially use the cores of a laptop, cluster, or supercomputer; task parallelism to create concurrency within a node or across the system". There are two graphs comparing Chapel performance against MPI and OpenMP: "PRK Stencil Performance (Gflop/s)" and "NPB-FT Performance (Gop/s)". Both graphs show performance increasing with the number of locales (36 cores per locale) from 16 to 256. The PRK graph compares MPI, OpenMP, and Chapel, while the NPB-FT graph compares UPC, MPI, and Chapel. A section titled "New to Chapel?" provides links to an overview talk, slides, blog-length introductions, projects powered by Chapel, and performance highlights. Finally, a code snippet shows how to use Chapel's Cyclic distribution library:

```
use CyclicDist;           // use the Cyclic distribution Library
config const n = 100;      // use --n=<val> when executing to override this default
forall i in {1..n} dmapped Cyclic(startIdx=1) do
    writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```

# SUGGESTED READING / VIEWING

---

## Chapel Overviews / History (in chronological order):

- [\*Chapel\*](#) chapter from [\*Programming Models for Parallel Computing\*](#), MIT Press, edited by Pavan Balaji, November 2015
- [\*Chapel Comes of Age: Making Scalable Programming Productive\*](#), Chamberlain et al., CUG 2018, May 2018
- Proceedings of the [\*8th Annual Chapel Implementers and Users Workshop\*](#) (CHIUW 2021), June 2021
- [\*Chapel Release Notes\*](#) — current version 1.25, October 2021

## Arkouda:

- Bill Reus's CHIUW 2020 keynote talk: <https://chapel-lang.org/CHIUW2020.html#keynote>
- Arkouda GitHub repo and pointers to other resources: <https://github.com/Bears-R-Us/arkouda>

## CHAMPS:

- Eric Laurendeau's CHIUW 2021 keynote talk: <https://chapel-lang.org/CHIUW2021.html#keynote>
  - two of his students also gave presentations at CHIUW 2021, also available from the URL above
- Another paper/presentation by his students at <https://chapel-lang.org/papers.html> (search “Laurendeau”)



# CHAPEL IS HIRING

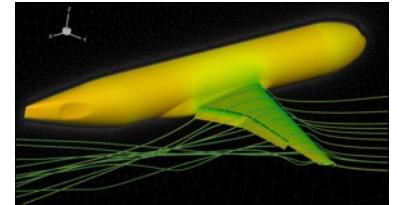
- Chapel team at HPE is currently 18.5 full-time employees
  - planning to add 1–2 more during 2021–2022
  - see: <https://chapel-lang.org/jobs.html>
- During summers, we also host interns and mentor Google Summer of Code students



# SUMMARY

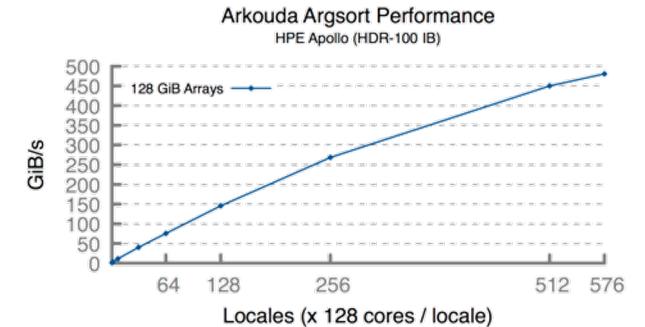
## Chapel is being used for productive parallel programming at scale

- recent users have reaped its benefits in 16k–48k-line applications



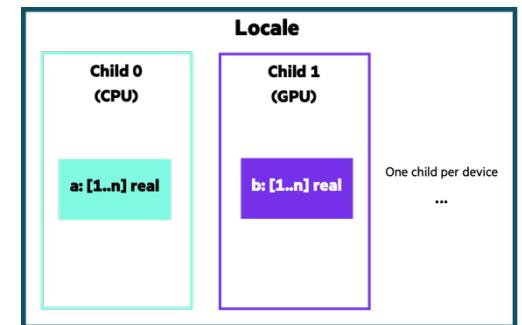
## For gather/scatter/sort patterns, copy aggregation is key

- particularly important for key operations in Arkouda



## Though Chapel support for GPUs is still in its early days, it's improving by leaps and bounds

- should enable users like the CHAMPS team to leverage GPUs more productively



# THANK YOU

---

<https://chapel-lang.org>  
@ChapelLanguage

