



**Hewlett Packard
Enterprise**

WHAT'S NEW WITH CHAPEL? APPLICATIONS, AGGREGATORS, AND ACCELERATORS

Brad Chamberlain

Northwest C++ Users' Group

January 19, 2022

CHAPEL MOTIVATION

Imagine having a programming language for HPC* that was as...

...**programmable** as Python

...yet also as...

...**fast** as Fortran

...**scalable** as MPI or SHMEM

...**portable** as C

...**flexible** as C++

...**type-safe** as Fortran, C, C++, ...

...**fun** as [your favorite programming language]

* = High Performance Computing

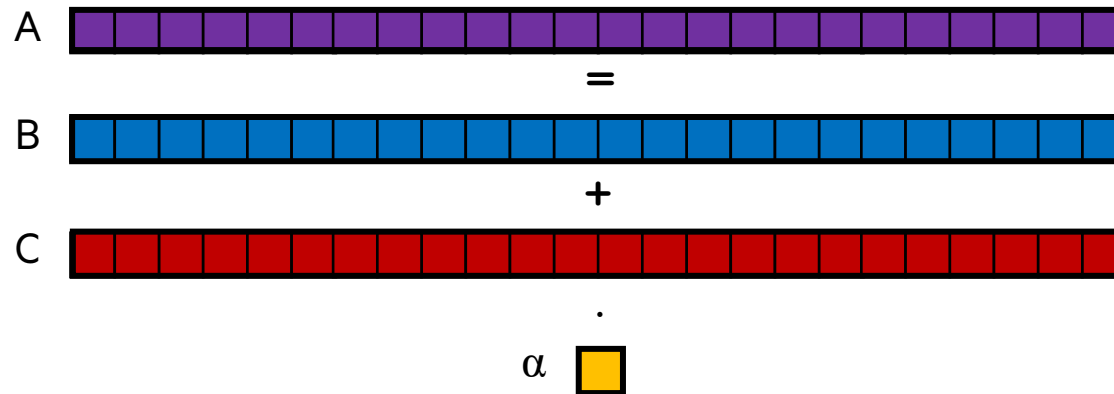


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: m -element vectors A, B, C

Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures:

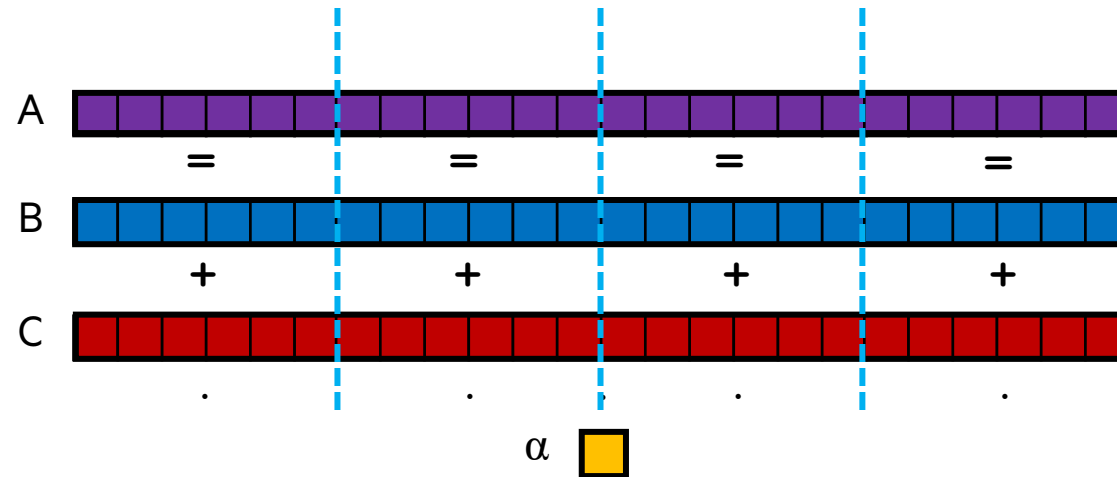


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: m -element vectors A, B, C

Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (shared memory / multicore):

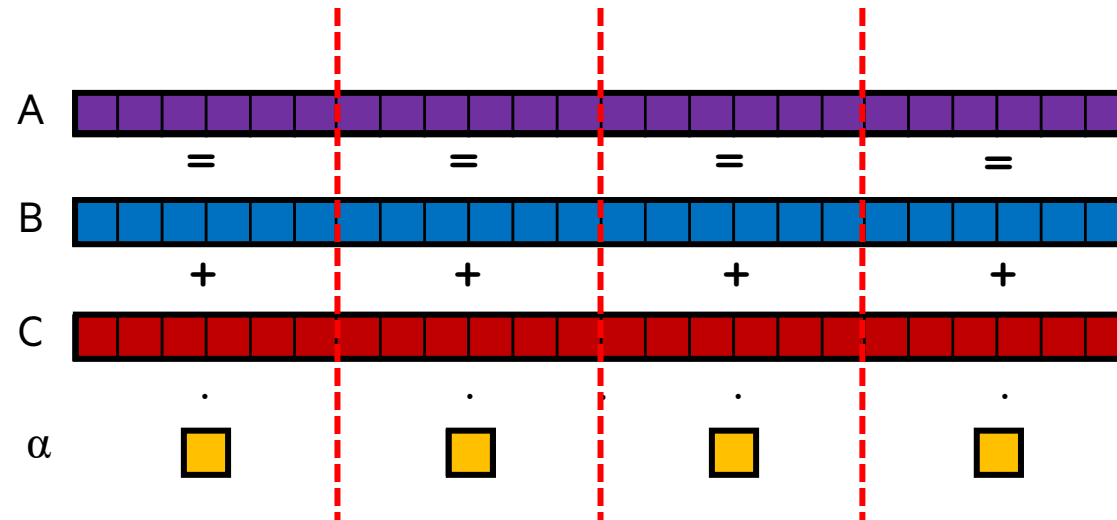


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: m -element vectors A, B, C

Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory):

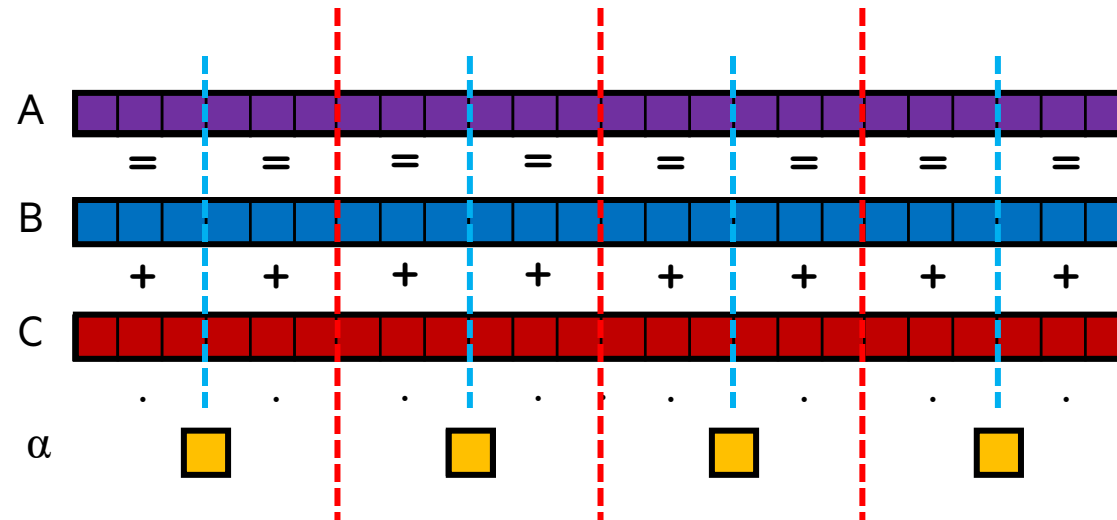


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: m -element vectors A, B, C

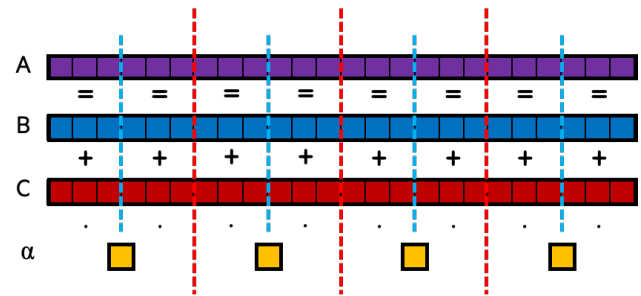
Compute: $\forall i \in 1..m, A_i = B_i + \alpha \cdot C_i$

In pictures, in parallel (distributed memory multicore):



STREAM TRIAD IN CONVENTIONAL HPC PROGRAMMING MODELS

Many Disparate Notations for Expressing Parallelism + Locality



```
#include <hpcc.h> MPI

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

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    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 );

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to
                allocate memory (%d).\n",
                    VectorSize );
            fclose( outFile );
        }
        return 1;
    }

    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 1.0;
    }
    scalar = 3.0;

    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];

    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);

    return 0; }

```

Note: This is a very trivial parallel computation—imagine the additional differences for something more complex!

Challenge: Can we do better?



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



WHAT DOES “PRODUCTIVITY” MEAN TO YOU?

Recent Graduates:

“Something similar to what I used in school: Python, Matlab, Java, ...”

Seasoned HPC Programmers:

“That sugary stuff which I can’t use because I need full control to ensure good performance”

Computational Scientists:

“Something that lets me focus on my science without having to wrestle with architecture-specific details”

Chapel Team:

“Something that lets computational scientists express what they want, without taking away the control that HPC programmers need, implemented in a language that’s attractive to recent graduates.”



SPEAKING OF THE CHAPEL TEAM...

Chapel is truly a team effort—we're currently at 19 full-time employees (+ a director), and we are hiring

Chapel Development Team at HPE



see: <https://chapel-lang.org/contributors.html>
and <https://chapel-lang.org/jobs.html>



OUTLINE



I. Motivation for Chapel

II. Chapel Applications

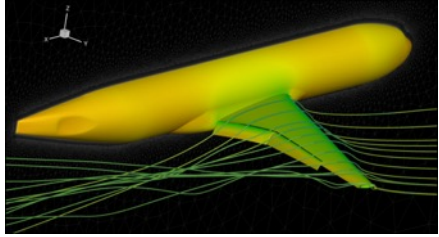
III. Some Chapel Features

IV. Aggregation in Chapel

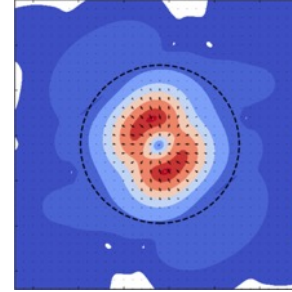
V. Chapel and Accelerators

VI. Wrap-up

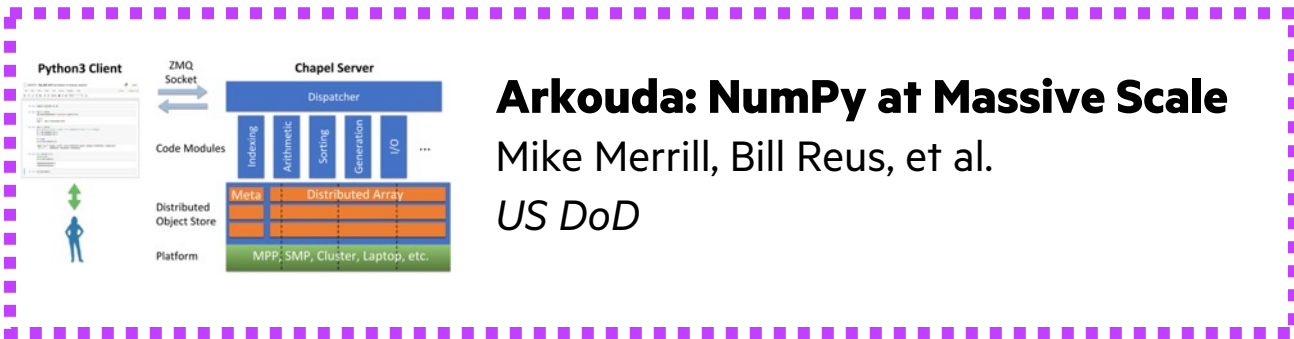
CURRENT FLAGSHIP CHAPEL APPLICATIONS



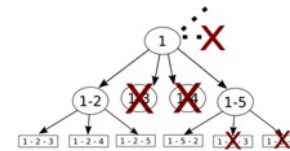
CHAMPS: 3D Unstructured CFD
 Éric Laurendeau, Simon Bourgault-Côté,
 Matthieu Parenteau, et al.
École Polytechnique Montréal



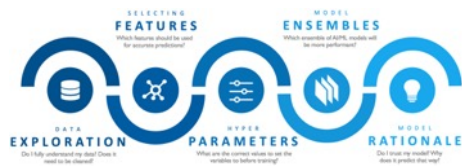
ChpUltra: Simulating Ultralight Dark Matter
 Nikhil Padmanabhan, J. Luna Zagorac, et al.
Yale University / University of Auckland



Arkouda: NumPy at Massive Scale
 Mike Merrill, Bill Reus, et al.
US DoD



ChOp: Chapel-based Optimization
 Tiago Carneiro, Nouredine Melab, et al.
INRIA Lille, France



CrayAI: Distributed Machine Learning
Hewlett Packard Enterprise



Your application here?



ARKOUDA IN ONE SLIDE

What is it?

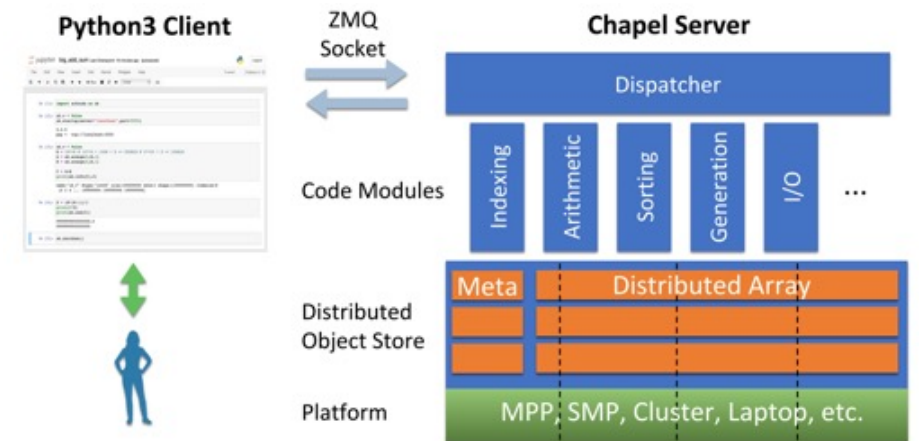
- A Python library supporting a key subset of NumPy and Pandas for Data Science
 - Computes massive-scale results within the human thought loop (seconds to minutes on multi-TB-scale arrays)
 - Uses a Python-client/Chapel-server model to get scalability and performance
- ~16k lines of Chapel, largely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

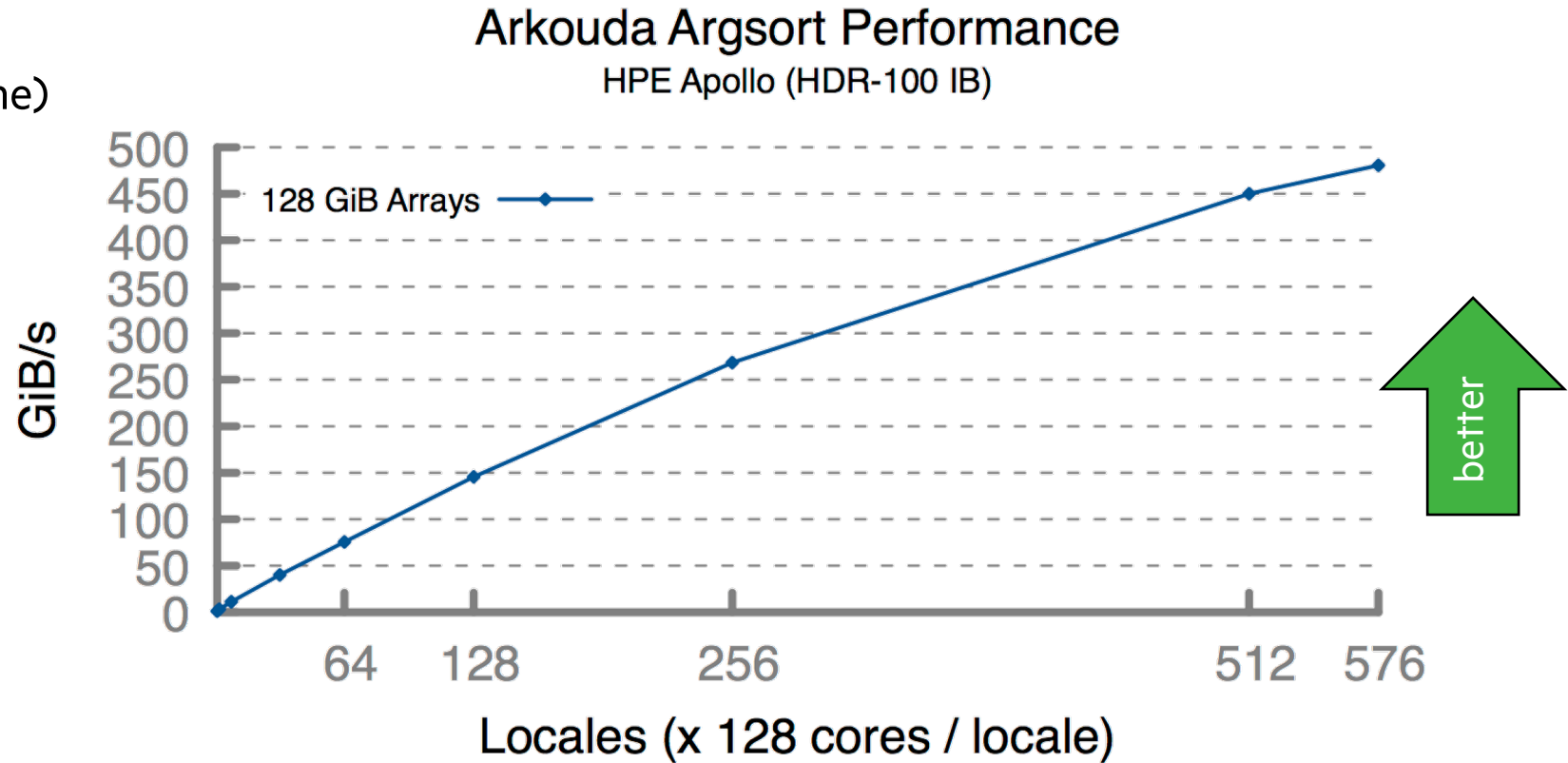
Why Chapel?

- 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

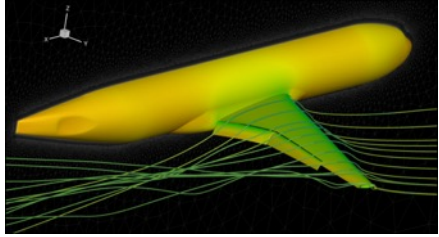
- Recent run performed on a large Apollo system
 - 72 TiB of 8-byte values
 - 480 GiB/s (2.5 minutes elapsed time)
 - used 73,728 cores of AMD Rome
 - ~100 lines of Chapel code



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

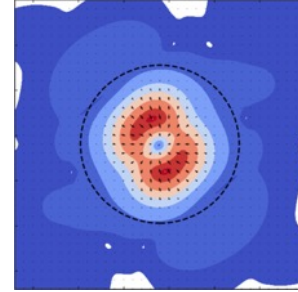


CURRENT FLAGSHIP CHAPEL APPLICATIONS



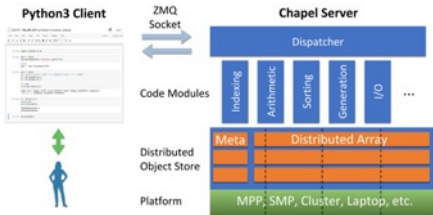
CHAMPS: 3D Unstructured CFD

Éric Laurendeau, Simon Bourgault-Côté,
Matthieu Parenteau, et al.
École Polytechnique Montréal



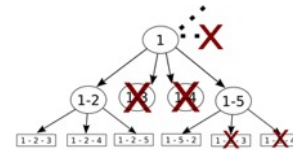
ChpUltra: Simulating Ultralight Dark Matter

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



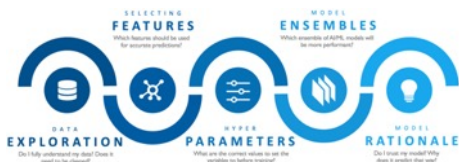
Arkouda: NumPy at Massive Scale

Mike Merrill, Bill Reus, et al.
US DoD



ChOp: Chapel-based Optimization

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



CrayAI: Distributed Machine Learning

Hewlett Packard Enterprise



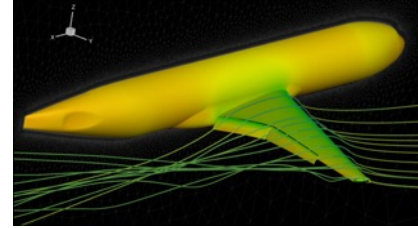
Your application here?



CHAMPS SUMMARY

What is it?

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

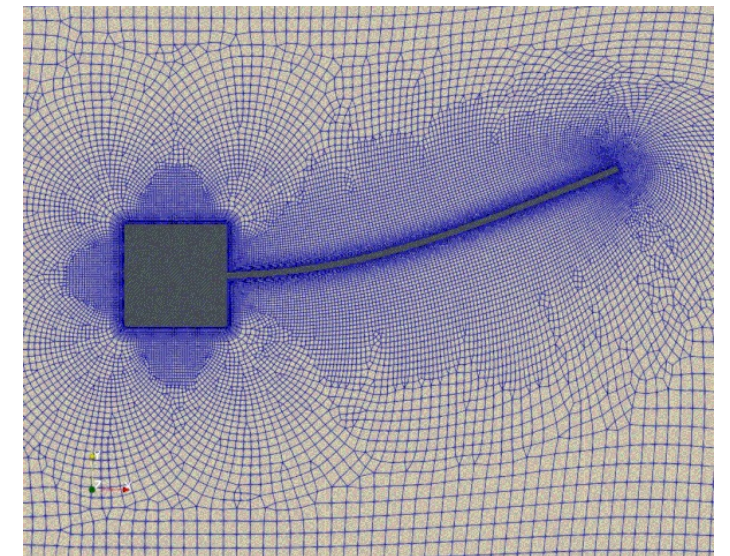
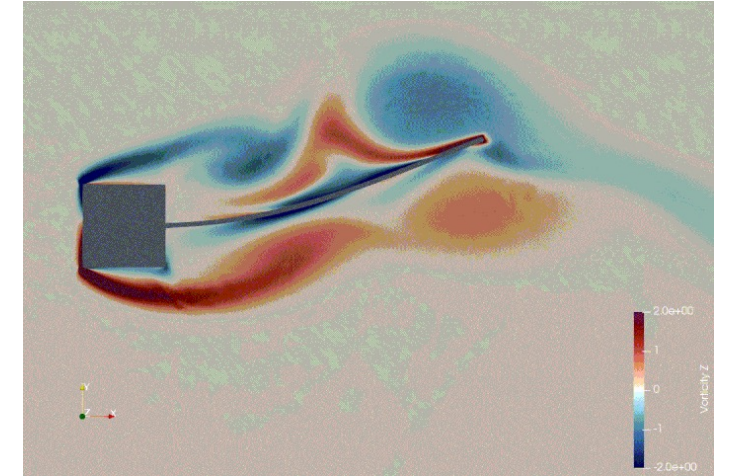


Who wrote it?

- 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: 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.”*

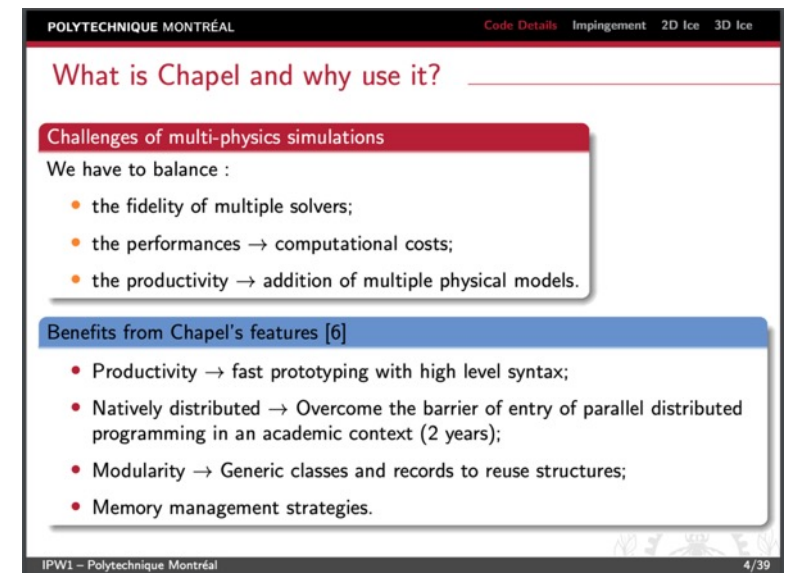
- Talk available online: https://youtu.be/wD-a_KyB8aI?t=1904 (hyperlink jumps to the section quoted here)



**POLYTECHNIQUE
MONTRÉAL**

CHAMPS 2021 HIGHLIGHTS

- Presented at CASI/IASC Aero 21 Conference
 - Participated in 1st AIAA Ice Prediction Workshop
 - Participating in 4th 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, ...

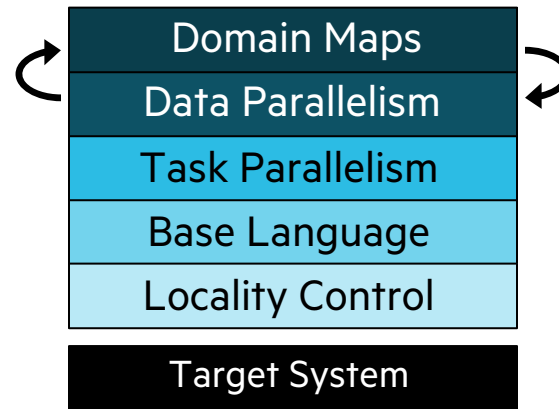




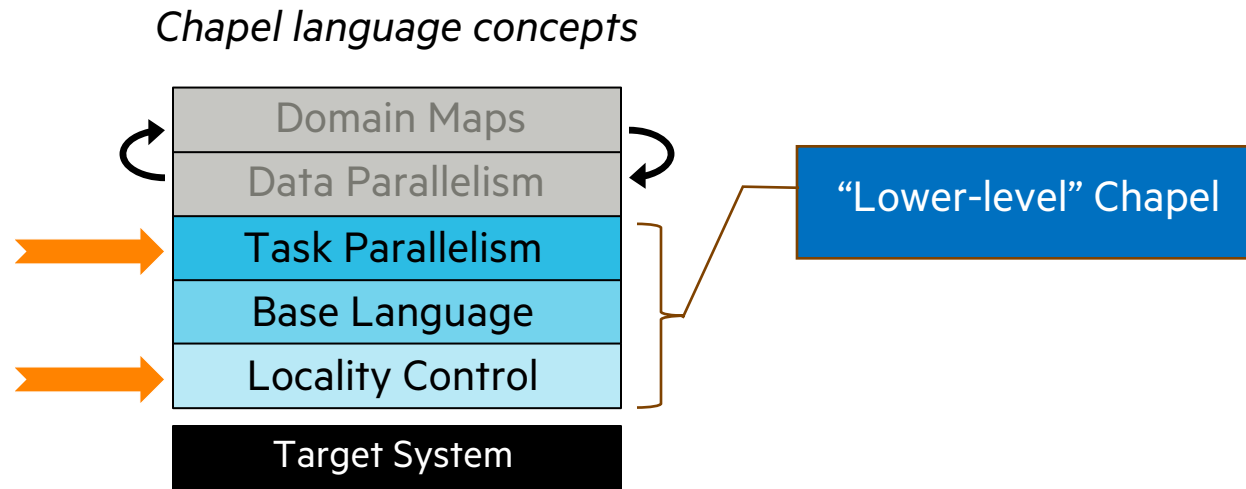
**SOME CHAPEL FEATURES
(THOSE NECESSARY FOR THE NEXT
FEW SECTIONS)**

CHAPEL FEATURE AREAS

Chapel language concepts



TASK PARALLELISM AND LOCALITY CONTROL

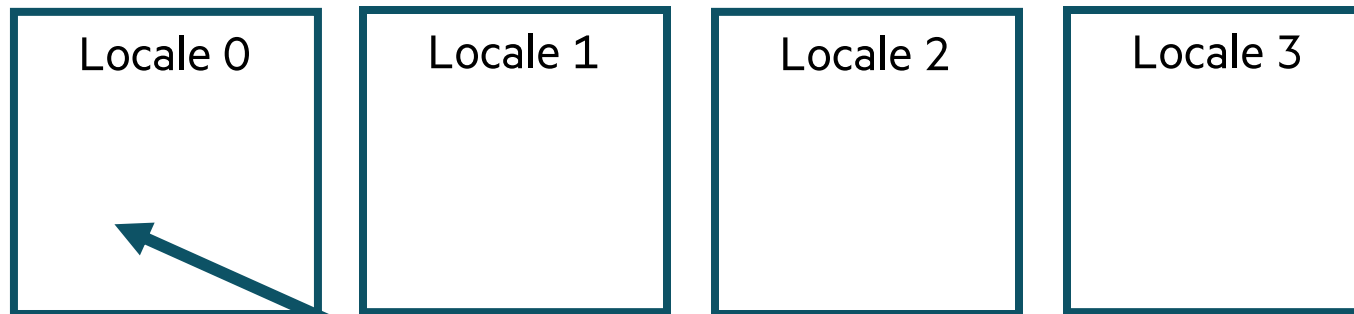


THE LOCALE: CHAPEL'S KEY FEATURE FOR LOCALITY

- *locale*: a unit of the target architecture that can run tasks and store variables
 - Think “compute node” on a typical HPC system

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

Locales array :



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



TASK-PARALLEL “HELLO WORLD”

helloTaskPar.chpl

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

‘here’ refers to the locale on which this code is currently running

how many parallel tasks can my locale run at once?

what’s my locale’s name?



TASK-PARALLEL “HELLO WORLD”

helloTaskPar.chpl

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

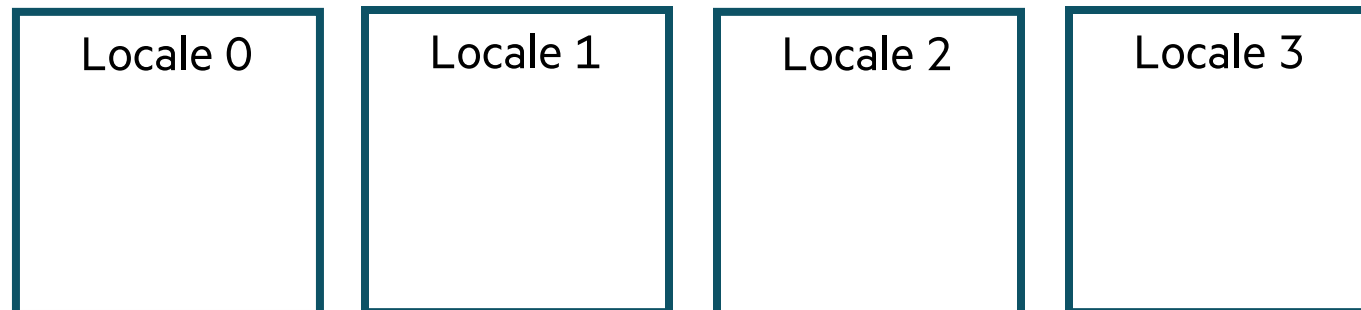
TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)

helloTaskPar.chpl

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

the array of locales we're running on
(introduced a few slides back)

Locales array:



TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)

helloTaskPar.chpl

```
coforall loc in Llocales {  
  on loc {  
    const numTasks = here.maxTaskPar;  
    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  
...
```

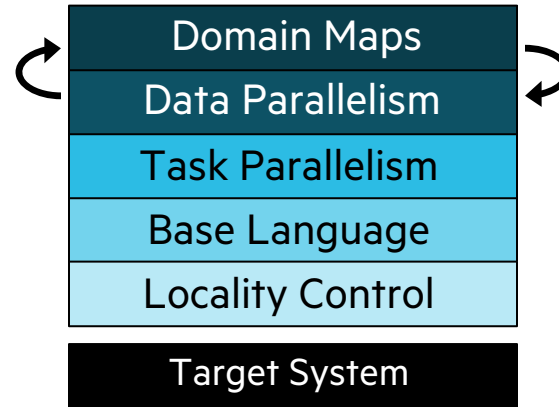

TASK-PARALLEL “HELLO WORLD” (DISTRIBUTED VERSION)

helloTaskPar.chpl

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

CHAPEL FEATURE AREAS

Chapel language concepts



SPECTRUM OF CHAPEL FOR-LOOP STYLES

for loop: each iteration is executed serially by the current task ('for i in 1..n' or 'for i in myIter()')

- predictable execution order, similar to conventional languages

```
for i in 1..n do ... // a serial loop over a range of integers
for j in myIter(...) do ... // a serial loop over a user-defined iterator
for (i, j) in zip(1..n, myIter(...)) do ... // zip over multiple iterables simultaneously, serially
```

forall loop: all iterations are executed in parallel in no specific order

- implemented using one or more tasks, locally or distributed, as determined by the iterand expression

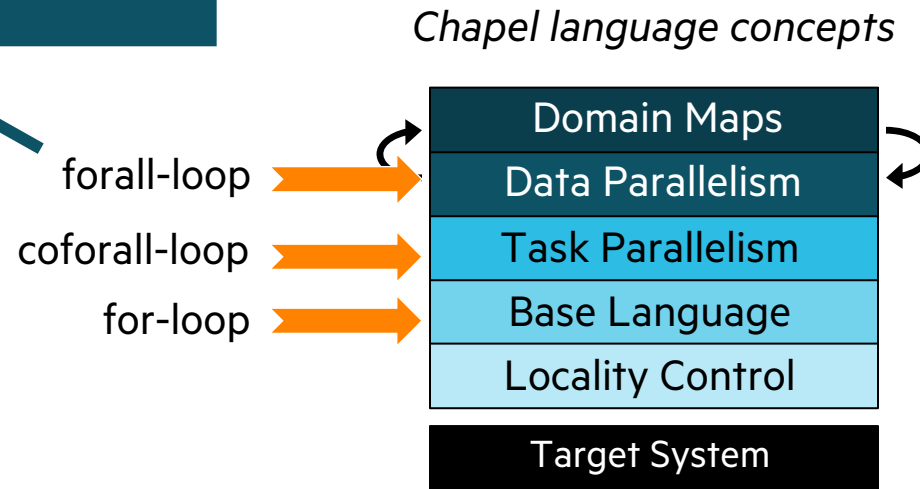
```
forall i in 1..n do ... // a parallel loop over a range of integers
forall j in myIter(...) do ... // invoke a parallel user-defined iterator
forall (i, j) in zip(1..n, myIter(...)) do ... // zip over multiple iterables simultaneously, in parallel
forall i in 1..n with (var t = ...) do ... // give each component task a task-local variable 't'
```

coforall loop: each iteration is executed concurrently by a distinct task

- explicit parallelism; supports synchronization between iterations (tasks)

CHAPEL FEATURE AREAS: WHERE LOOPS FIT IN

in Chapel, all forall-loops are implemented using lower-level features like coforall- and for-loops, on-clauses, etc.





CHAPEL AGGREGATORS

BALE INDEX GATHER KERNEL IN CHAPEL: NAÏVE VERSION

```
// Naive index gather: Dst = Src[Inds];  
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: Dst = Src[Inds];  
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>.maxTaskPar 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: Dst = Src[Inds];  
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>.maxTaskPar 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?

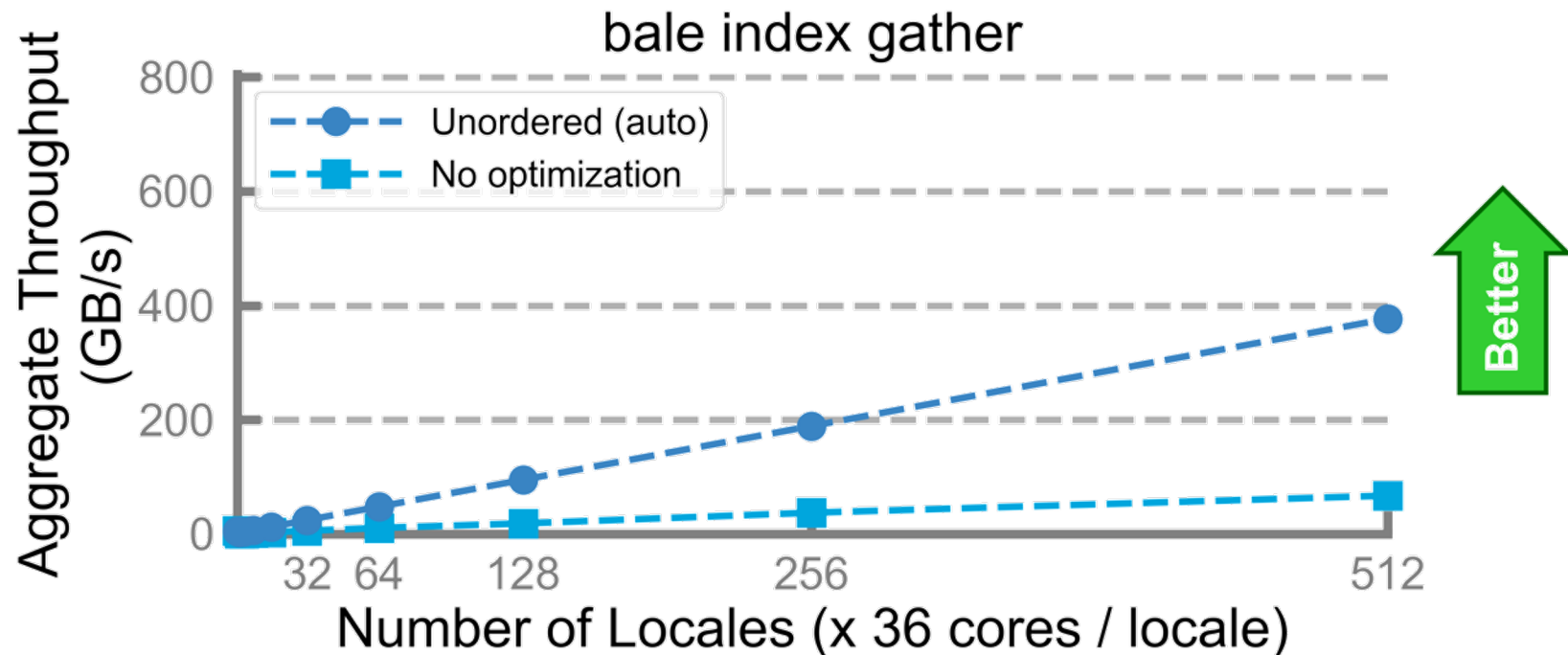
So, our compiler rewrites the inner loop to perform the ops asynchronously

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

- Implemented by Michael Ferguson and Elliot Ronaghan, 2019

BALE INDEX GATHER KERNEL IN CHAPEL: NAÏVE VERSION

```
// Naive index gather: Dst = Src[Inds];  
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]);
```

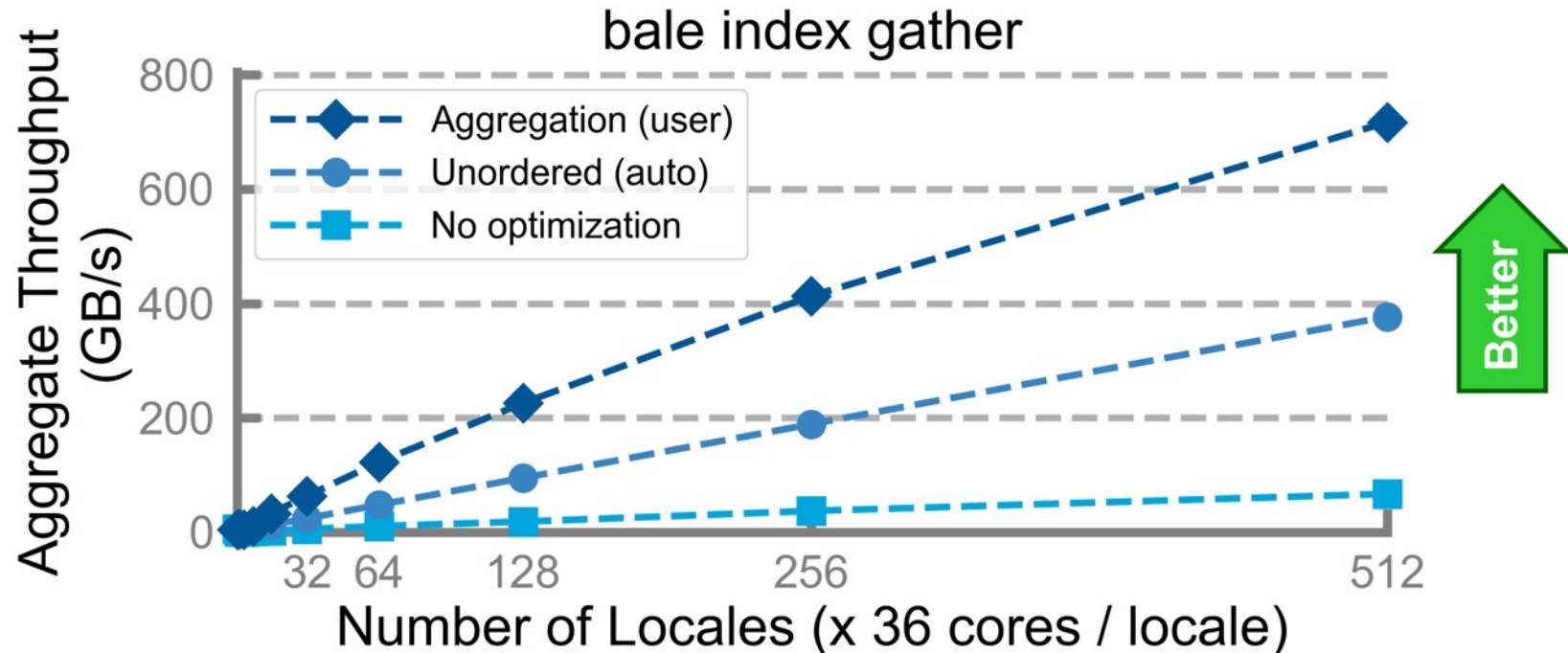
→ Give each task a "source aggregator", *agg*, which aggregates remote 'gets' locally, then performs them

→ To use it, we simply replace the assignment with 'agg.copy'

→ 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]);
```



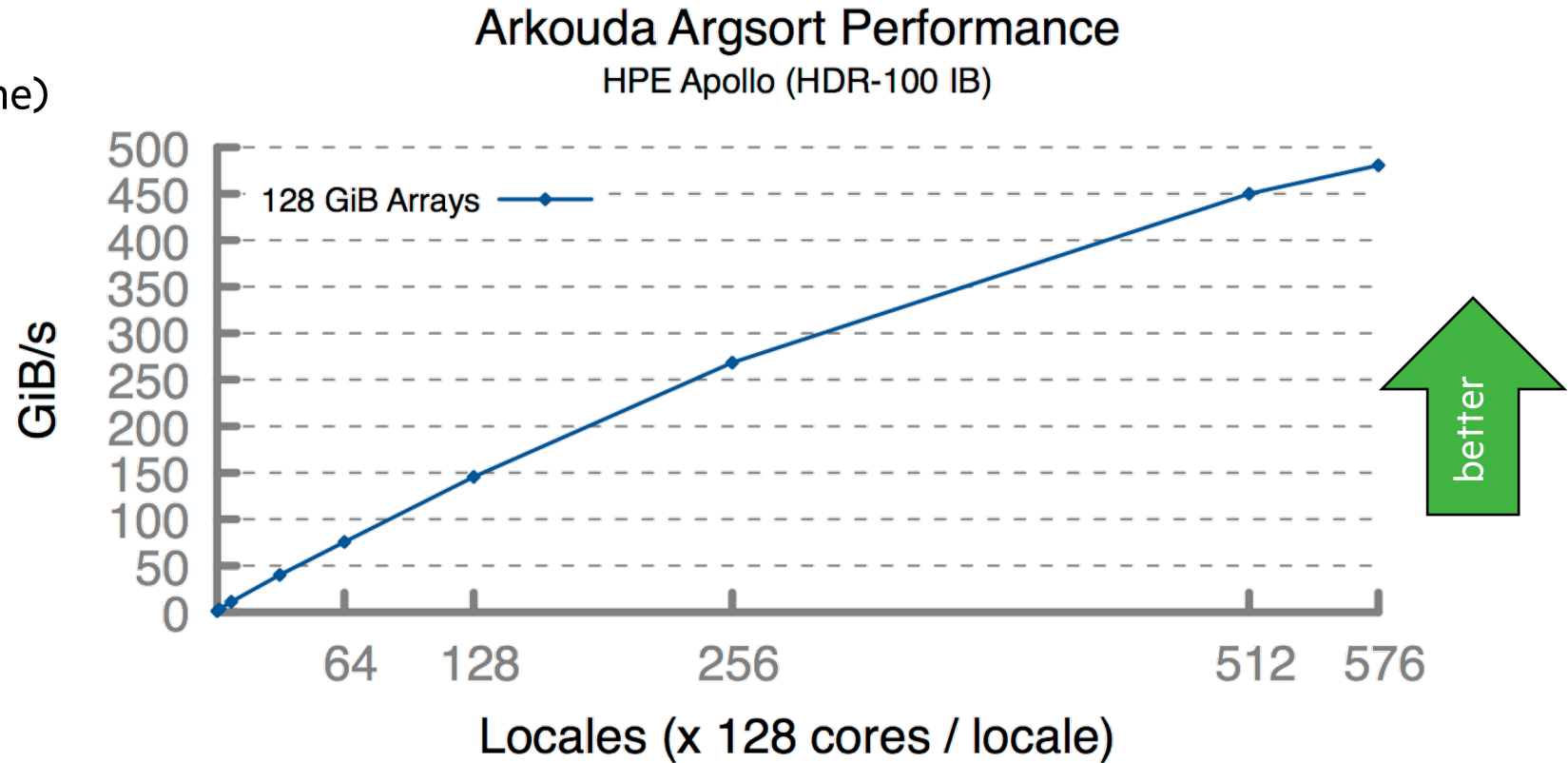
IMPLEMENTING CHAPEL'S AGGREGATORS

- 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: Dst = Src[Inds];  
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]);
```

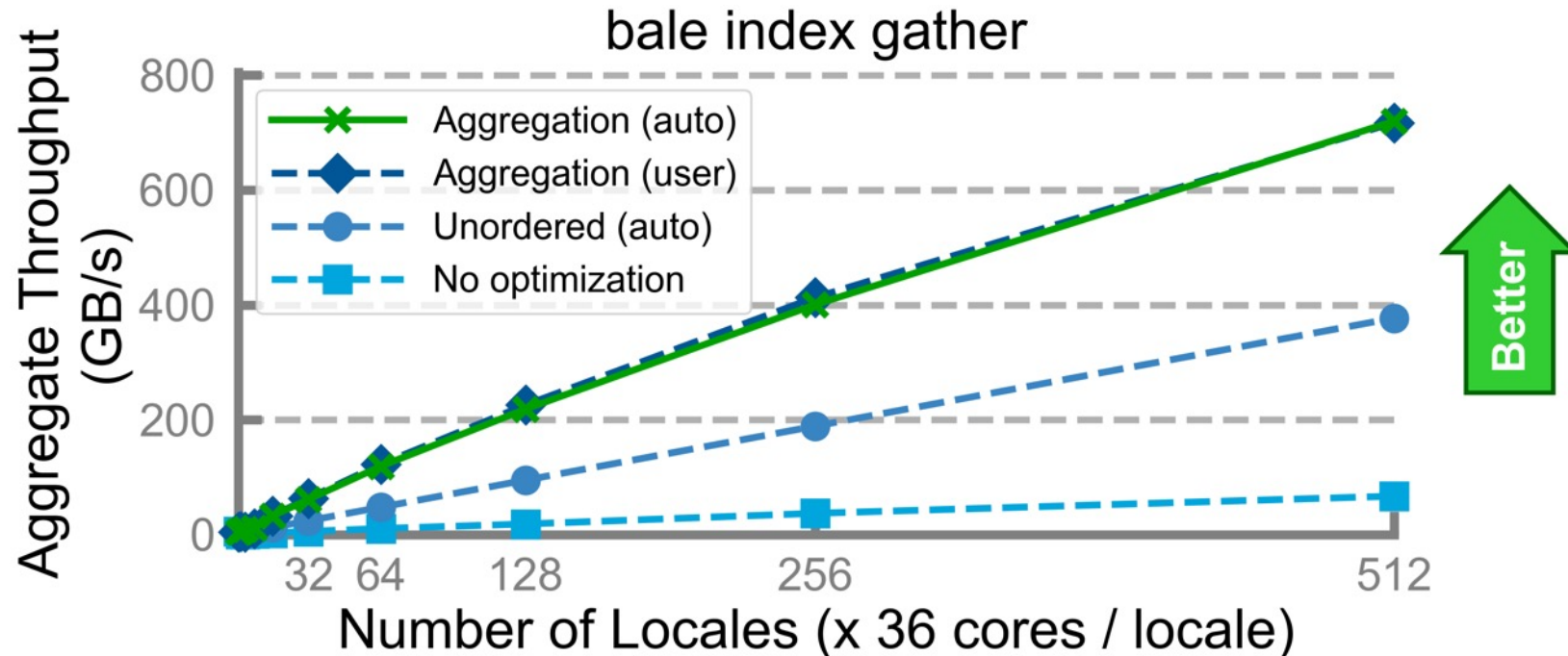
A: In many cases, yes

- developed by Engin Kayraklioglu, 2021
- combines previous ‘unordered’ analysis with a new locality analysis of RHS/LHS expressions
- for details, see Engin’s LCPC 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: Dst = Src[Inds];  
forall (d, i) in zip(Dst, Inds) do  
  d = Src[i];
```



BALE INDEX GATHER: CHAPEL VS. EXSTACK VS. CONVEYORS

Exstack version

```

i=0;
while( exstack_proceed(ex, (i==l_num_req)) ) {
  i0 = i;
  while(i < l_num_req) {
    l_indx = pckindx[i] >> 16;
    pe = pckindx[i] & 0xffff;
    if(!exstack_push(ex, &l_indx, pe))
      break;
    i++;
  }

  exstack_exchange(ex);

  while(exstack_pop(ex, &idx, &fromth)) {
    idx = ltable[idx];
    exstack_push(ex, &idx, fromth);
  }
  lgp_barrier();
  exstack_exchange(ex);

  for(j=i0; j<i; j++) {
    fromth = pckindx[j] & 0xffff;
    exstack_pop_thread(ex, &idx, (uint64_t)fromth);
    tgt[j] = idx;
  }
  lgp_barrier();
}

```

Conveyors version

```

i = 0;
while (more = convey_advance(requests, (i == l_num_req)),
       more | convey_advance(replies, !more)) {

  for (; i < l_num_req; i++) {
    pkg.idx = i;
    pkg.val = pckindx[i] >> 16;
    pe = pckindx[i] & 0xffff;
    if (! convey_push(requests, &pkg, pe))
      break;
  }

  while (convey_pull(requests, ptr, &from) == convey_OK) {
    pkg.idx = ptr->idx;
    pkg.val = ltable[ptr->val];
    if (! convey_push(replies, &pkg, from)) {
      convey_unpull(requests);
      break;
    }
  }

  while (convey_pull(replies, ptr, NULL) == convey_OK)
    tgt[ptr->idx] = ptr->val;
}

```

Elegant Chapel version (compiler-optimized w/ '--auto-aggregation')

```

forall (d, i) in zip(Dst, Inds) do
  d = Src[i];

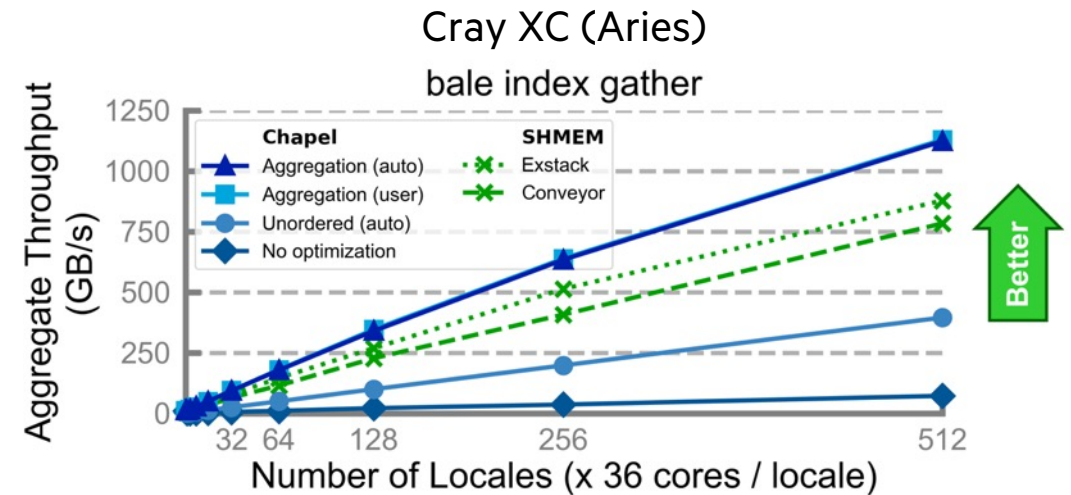
```

Manually Tuned Chapel version (using aggregator abstraction)

```

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

```





CHAPEL ON GPUS

THE CASE FOR CHAPEL ON GPUS

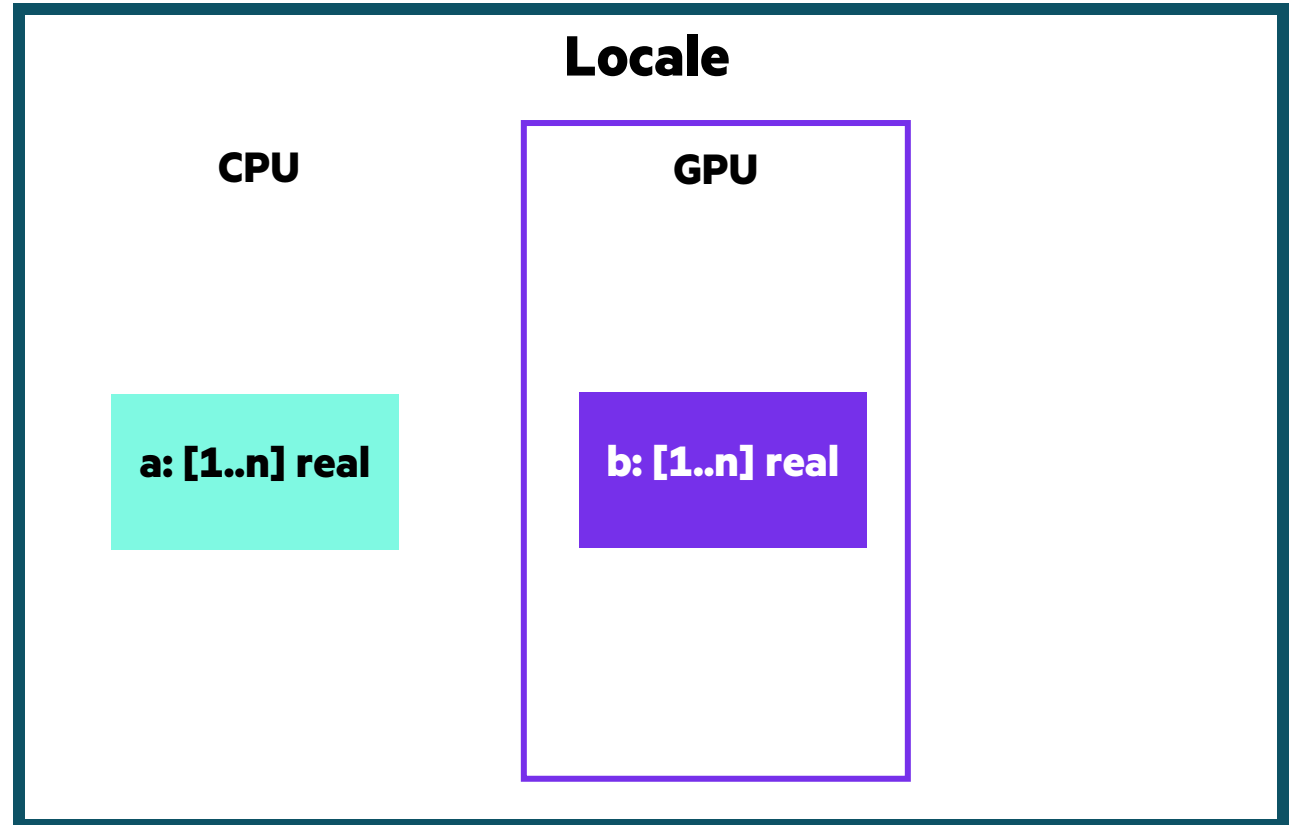
- “any parallel algorithm on any parallel architecture”
 - yet, Chapel has not supported compilation to GPUs—an obvious important case for many HPC systems
- Related efforts:
 - Albert Sidelnik et al. (UIUC), **Performance portability with the Chapel language**, *IPDPS 2012*
 - Brad Chamberlain, **Chapel 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
 - data-parallel loops and operations; on-clauses for saying where to store/execute things
 - 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 toy GPU computation, notionally:

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



GPUS: SIX MONTHS AGO

The toy 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);
```

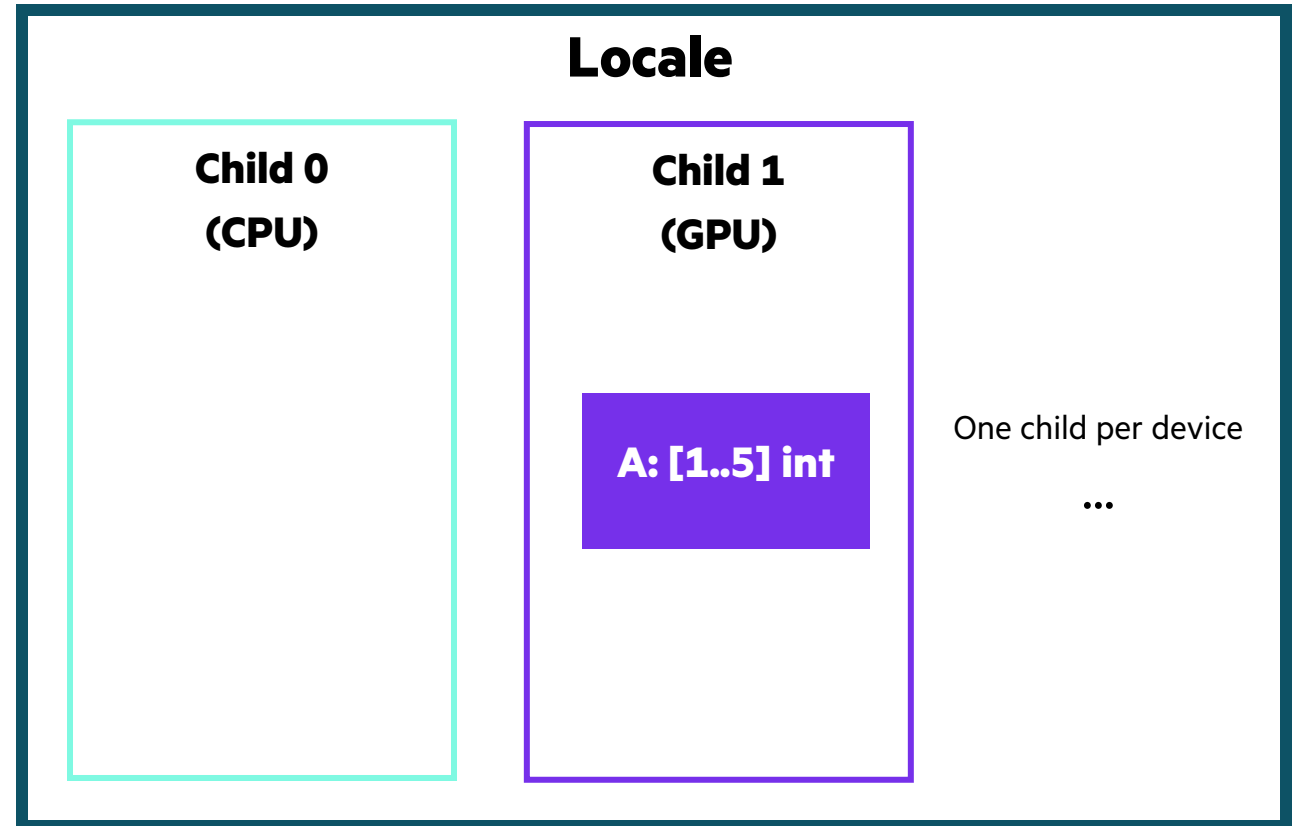
```
extern {
  #define FATBIN_FILE "chpl__gpu.fatbin"
  double createFunction() {
    fatbinBuffer = <read FATBIN_FILE into buffer>
    cuModuleLoadData(&cudaModule, fatbinBuffer);
    cuModuleGetFunction(&function, cudaModule,
                       "add_nums");}
}
```

Read
fat binary
and create a
CUDA
function

GPUS: TODAY

The toy GPU computation, in Chapel 1.25:

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

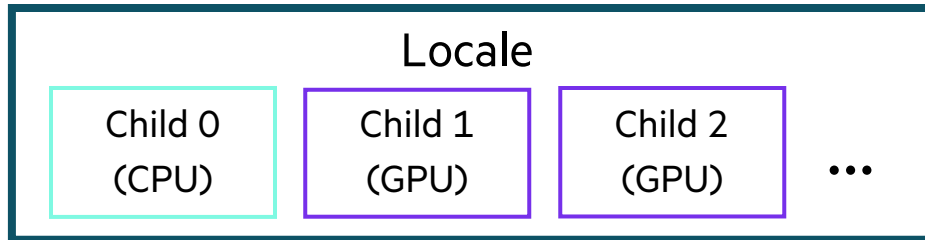


- developed by Engin Kayraklioglu, Andy Stone, and David Iten

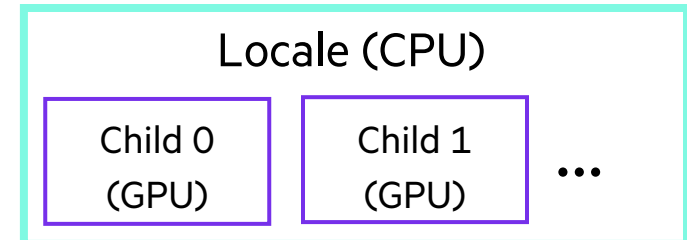


ALTERNATIVE GPU LOCALE MODELS

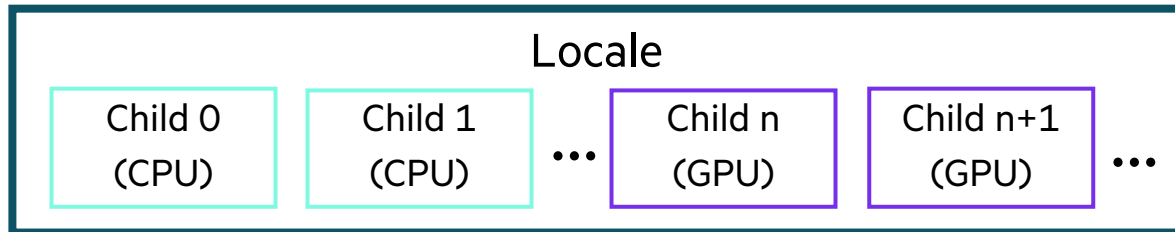
What we have now (sub-locale 0 = CPU)



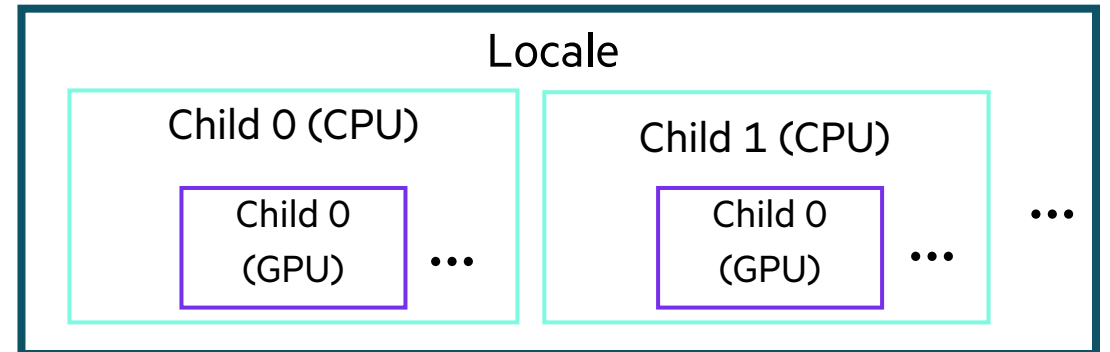
Locale for CPU; sub-locales for GPUs



NUMA-aware (flat)



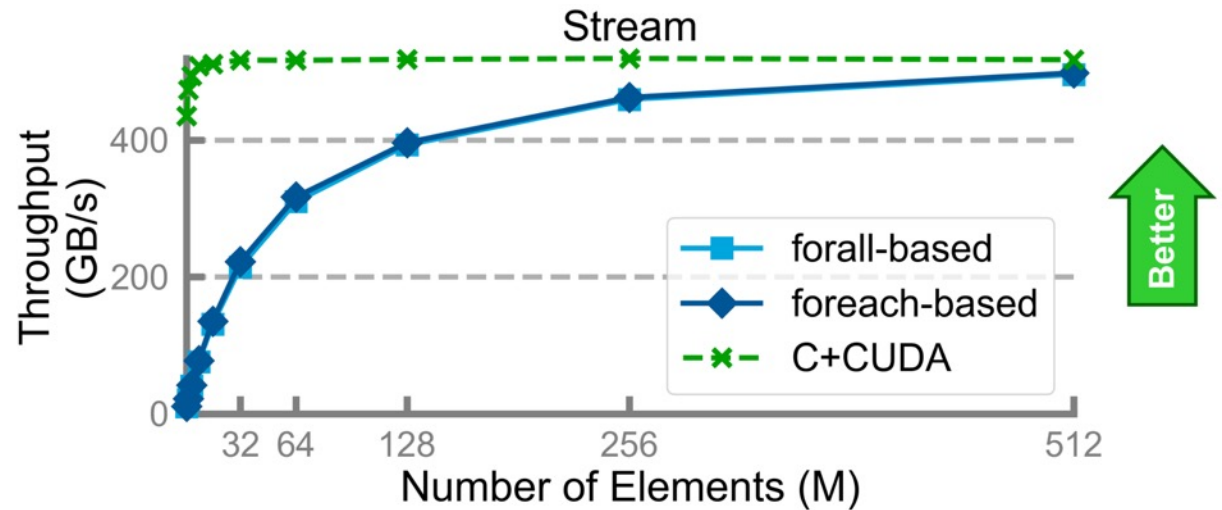
NUMA-aware (hierarchical)



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;  
  
  B = 1.0;  
  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)



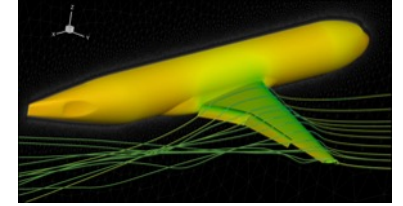
WRAP-UP



SUMMARY

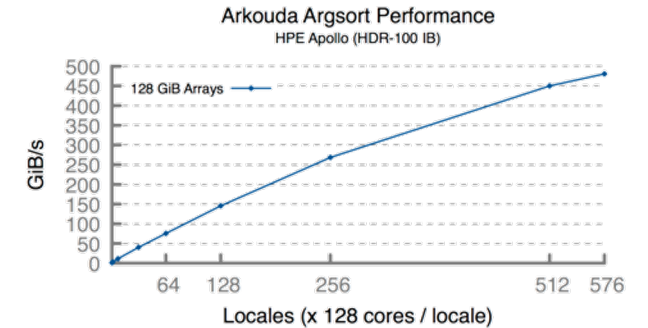
Chapel is being used for productive parallel programming at scale

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

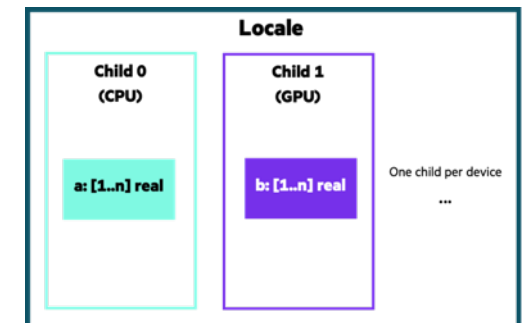


For gather/scatter/sort patterns, copy aggregation is key

- Chapel supports aggregation both through high-level abstractions and automatic optimizations



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



AGAIN, WE ARE HIRING

Chapel Development Team at HPE



see: <https://chapel-lang.org/contributors.html>
and <https://chapel-lang.org/jobs.html>



CHAPEL RESOURCES

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


- (points to all other resources)

Social Media:

- Twitter: [@ChapelLanguage](https://twitter.com/ChapelLanguage)
- Facebook: [@ChapelLanguage](https://www.facebook.com/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 Chapel Parallel Programming Language

What is Chapel?

Chapel is a programming language designed for productive parallel computing at scale.

Why Chapel?

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

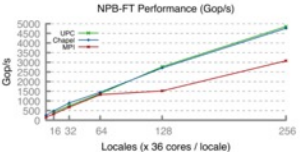
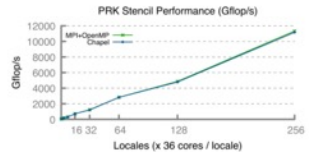
Chapel Characteristics

- **productive**: code tends to be similarly readable/writable as Python
- **scalable**: runs on laptops, clusters, the cloud, and HPC systems
- **fast**: performance *competes with or beats* C/C++ & MPI & OpenMP
- **portable**: compiles and runs in virtually any *nix environment
- **open-source**: hosted on GitHub, permissively licensed

New to Chapel?

As an introduction to Chapel, you may want to...

- watch an [overview talk](#) or browse its [slides](#)
- read a [blog-length](#) or [chapter-length](#) introduction to Chapel
- learn about [projects powered by Chapel](#)
- check out [performance highlights](#) like these:



The PRK Stencil Performance graph shows Chapel (green line) significantly outperforming MPI+OpenMP (red line) as the number of locales increases from 16 to 256. The NPB-FT Performance graph shows Chapel (green line) also outperforming MPI (red line) in a similar trend.

- browse [sample programs](#) or learn how to write distributed programs like this one:

```
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](#) (CHI UW 2021), June 2021
- [Chapel Release Notes](#) — current version 1.25, October 2021

Arkouda:

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

CHAMPS:

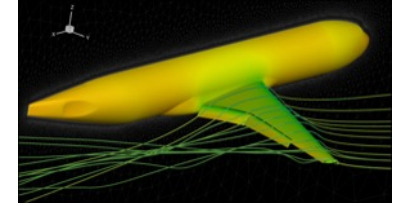
- Eric Laurendeau's CHI UW 2021 keynote talk: <https://chapel-lang.org/CHI UW 2021.html#keynote>
 - two of his students also gave presentations at CHI UW 2021, also available from the URL above
- Another paper/presentation by his students at <https://chapel-lang.org/papers.html> (search “Laurendeau”)



SUMMARY

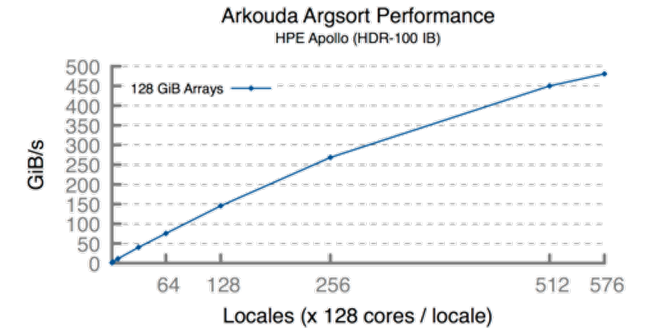
Chapel is being used for productive parallel programming at scale

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

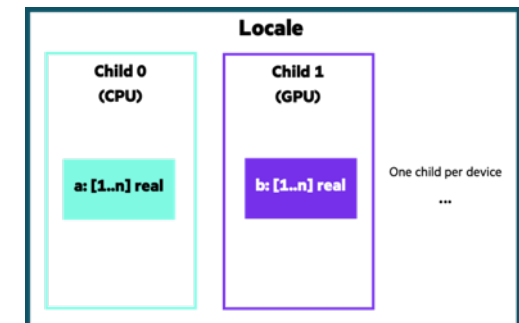


For gather/scatter/sort patterns, copy aggregation is key

- Chapel supports aggregation both through high-level abstractions and automatic optimizations



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





THANK YOU

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

