Hewlett Packard Enterprise

INTRODUCING CHAPEL:
A PROGRAMMING LANGUAGE FOR PRODUCTIVE PARALLEL COMPUTING FROM LAPTOPS TO SUPERCOMPUTERS

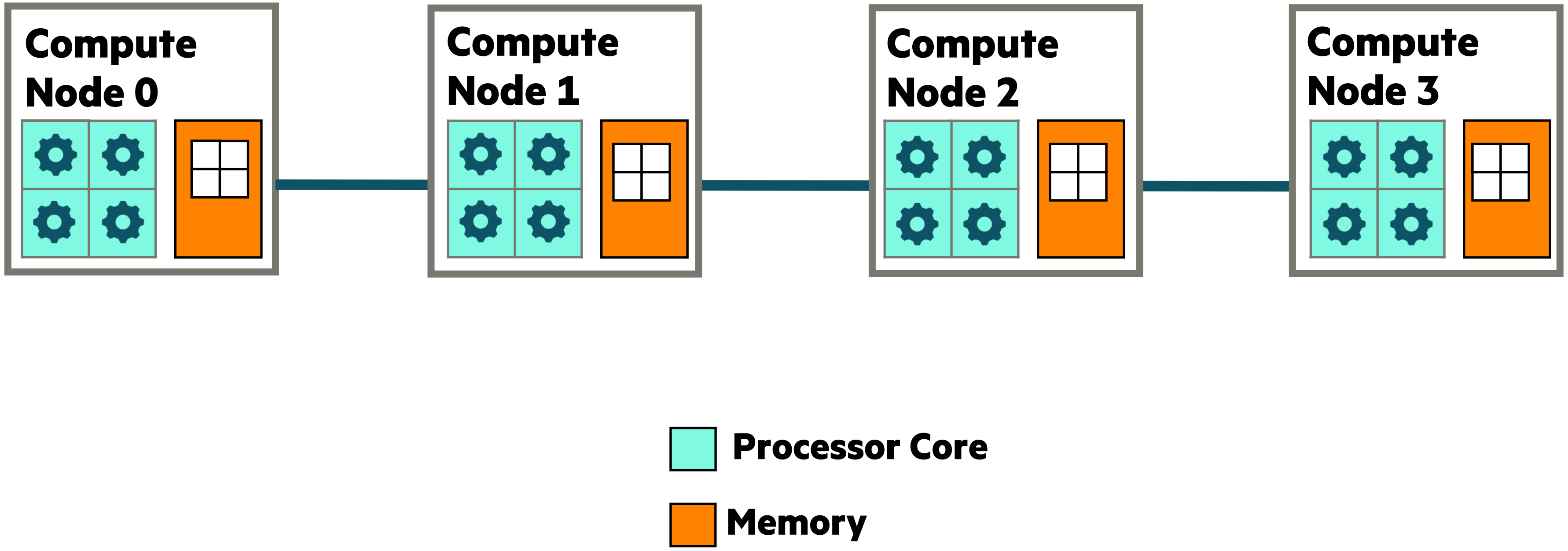
Brad Chamberlain, Distinguished Technologist

LinuxCon, May 11, 2023

PARALLEL COMPUTING IN A NUTSHELL

Parallel Computing: Using the processors and memories of multiple compute resources

- in order to run a program…
  - faster than we could otherwise
  - and/or using larger problem sizes



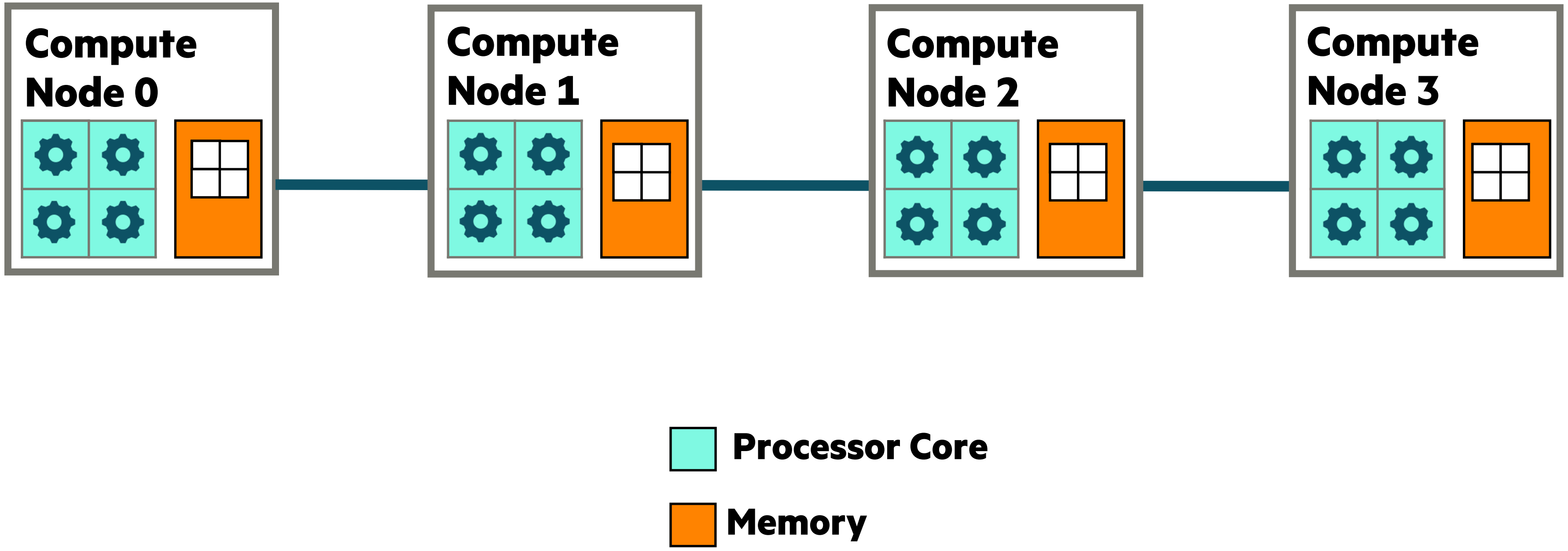
PARALLEL COMPUTING HAS BECOME UBIQUITOUS

Traditional parallel computing:

- supercomputers
- commodity clusters

Today:

- multicore processors
- GPUs
- cloud computing



OAK RIDGE NATIONAL LABORATORY'S FRONTIER SUPERCOMPUTER

- **74 HPE Cray EX cabinets**
- **9,408 AMD CPUs, 37,632 AMD GPUs**
- **700 petabytes of storage capacity, peak write speeds of 5 terabytes per second using Cray ClusterStor storage system**
- **HPE Slingshot networking cables providing 100 GB/s network bandwidth.**

TOP500

1

Built by HPE, ORNL's Frontier supercomputer is #1 on the TOP500.

1.1 exaflops of performance.

GREEN500

2

Built by HPE, ORNL's TDS and full system are ranked #2 & #6 on the Green500.

62.68 gigaflops/watt power efficiency for ORNL's TDS system, **52.23 gigaflops/watt** power efficiency for full system.

HPL-MxP

1

Built by HPE, ORNL's Frontier supercomputer is #1 on the HPL-MxP list.

7.9 exaflops on the HPL-MxP benchmark (formerly HPL-AI).

Source: May 30, 2022 Top500 release, HPL-MxP mixed-precision benchmark (formerly HPL-AI).

HPC BENCHMARKS USING CONVENTIONAL PROGRAMMING APPROACHES

STREAM TRIAD: C + MPI + OPENMP

```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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;
  }

#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_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```

HPCC RA: MPI KERNEL

```c
/* Perform updates to main table.  The scalar equivalent is:
 *
 *   for (i=0; i<NUPDATE; i++) {
 *     Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *     Table[Ran & (TABSIZE-1)] ^= Ran;
 *   }
 */

MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
 while (i < SendCnt) {
   /* receive messages */
   do {
     MPI_Test(&inreq, &have_done, &status);
     if (have_done) {
       if (status.MPI_TAG == UPDATE_TAG) {
         MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
         bufferBase = 0;
         for (j=0; j < recvUpdates; j ++) {
           inmsg = LocalRecvBuffer[bufferBase+j];
           LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                         tparams.GlobalStartMyProc;
           HPCC_Table[LocalOffset] ^= inmsg;
         }
       } else if (status.MPI_TAG == FINISHED_TAG) {
         NumberReceiving--;
       } else
         MPI_Abort( MPI_COMM_WORLD, -1 );
       MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                 MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
     }
   } while (have_done && NumberReceiving > 0);
   if (pendingUpdates < maxPendingUpdates) {
     Ran = (Ran << 1) ^ ((s64Int) Ran < ZERO64B ? POLY : ZERO64B);
     GlobalOffset = Ran & (tparams.TableSize-1);
     if ( GlobalOffset < tparams.Top)
       WhichPe = ( GlobalOffset / (tparams.MinLocalTableSize + 1) );
     else
       WhichPe = ( (GlobalOffset - tparams.Remainder) /
                   tparams.MinLocalTableSize );
     if (WhichPe == tparams.MyProc) {
       LocalOffset = (Ran & (tparams.TableSize - 1)) -
                     tparams.GlobalStartMyProc;
       HPCC_Table[LocalOffset] ^= Ran;
     }  else {
       HPCC_InsertUpdate(Ran, WhichPe, Buckets);
        pendingUpdates++;
     }
     i++;
   }
   else {
     MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
     if (have_done) {
       outreq = MPI_REQUEST_NULL;
       pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                            &peUpdates);
       MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
                 UPDATE_TAG, MPI_COMM_WORLD, &outreq);
       pendingUpdates -= peUpdates;
     }
   }
 }
 /* send remaining updates in buckets */
 while (pendingUpdates > 0) {
   /* receive messages */
   do {
     MPI_Test(&inreq, &have_done, &status);
     if (have_done) {
       if (status.MPI_TAG == UPDATE_TAG) {
         MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
         bufferBase = 0;
         for (j=0; j < recvUpdates; j ++) {
           inmsg = LocalRecvBuffer[bufferBase+j];
           LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                         tparams.GlobalStartMyProc;
           HPCC_Table[LocalOffset] ^= inmsg;
         }
       } else if (status.MPI_TAG == FINISHED_TAG) {
         /* we got a done message.  Thanks for playing... */
         NumberReceiving--;
       } else {
         MPI_Abort( MPI_COMM_WORLD, -1 );
       }
       MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
                 MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
     }
   } while (have_done && NumberReceiving > 0);

   MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
   if (have_done) {
     outreq = MPI_REQUEST_NULL;
     pe = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize,
                          &peUpdates);
     MPI_Isend(&LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe,
               UPDATE_TAG, MPI_COMM_WORLD, &outreq);
     pendingUpdates -= peUpdates;
   }
 }
 /* send our done messages */
 for (proc_count = 0 ; proc_count < tparams.NumProcs ; ++proc_count) {
   if (proc_count == tparams.MyProc) { tparams.finish_req[tparams.MyProc] =
                                       MPI_REQUEST_NULL; continue; }
   /* send garbage - who cares, no one will look at it */
   MPI_Isend(&Ran, 0, tparams.dtype64, proc_count, FINISHED_TAG,
             MPI_COMM_WORLD, tparams.finish_req + proc_count);
 }
 /* Finish everyone else up... */
 while (NumberReceiving > 0) {
   MPI_Wait(&inreq, &status);
   if (status.MPI_TAG == UPDATE_TAG) {
     MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
     bufferBase = 0;
     for (j=0; j < recvUpdates; j ++) {
       inmsg = LocalRecvBuffer[bufferBase+j];
       LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                     tparams.GlobalStartMyProc;
       HPCC_Table[LocalOffset] ^= inmsg;
     }
   } else if (status.MPI_TAG == FINISHED_TAG) {
     /* we got a done message.  Thanks for playing... */
     NumberReceiving--;
   } else {
     MPI_Abort( MPI_COMM_WORLD, -1 );
   }
   MPI_Irecv(&LocalRecvBuffer, localBufferSize, tparams.dtype64,
             MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
 }

 MPI_Waitall( tparams.NumProcs, tparams.finish_req, tparams.finish_statuses);
```

SCALABLE PARALLEL PROGRAMMING THAT'S AS NICE AS PYTHON?

Imagine having a programming language for parallel computing that was as...

...**programmable** as Python

...yet also as...

...**fast** as Fortran/C/C++

...**scalable** as MPI/SHMEM

...**GPU-ready** as CUDA/OpenMP/OpenCL/OpenACC/...

...**portable** as C

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

This is our motivation for Chapel

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

FIVE KEY CHARACTERISTICS OF CHAPEL

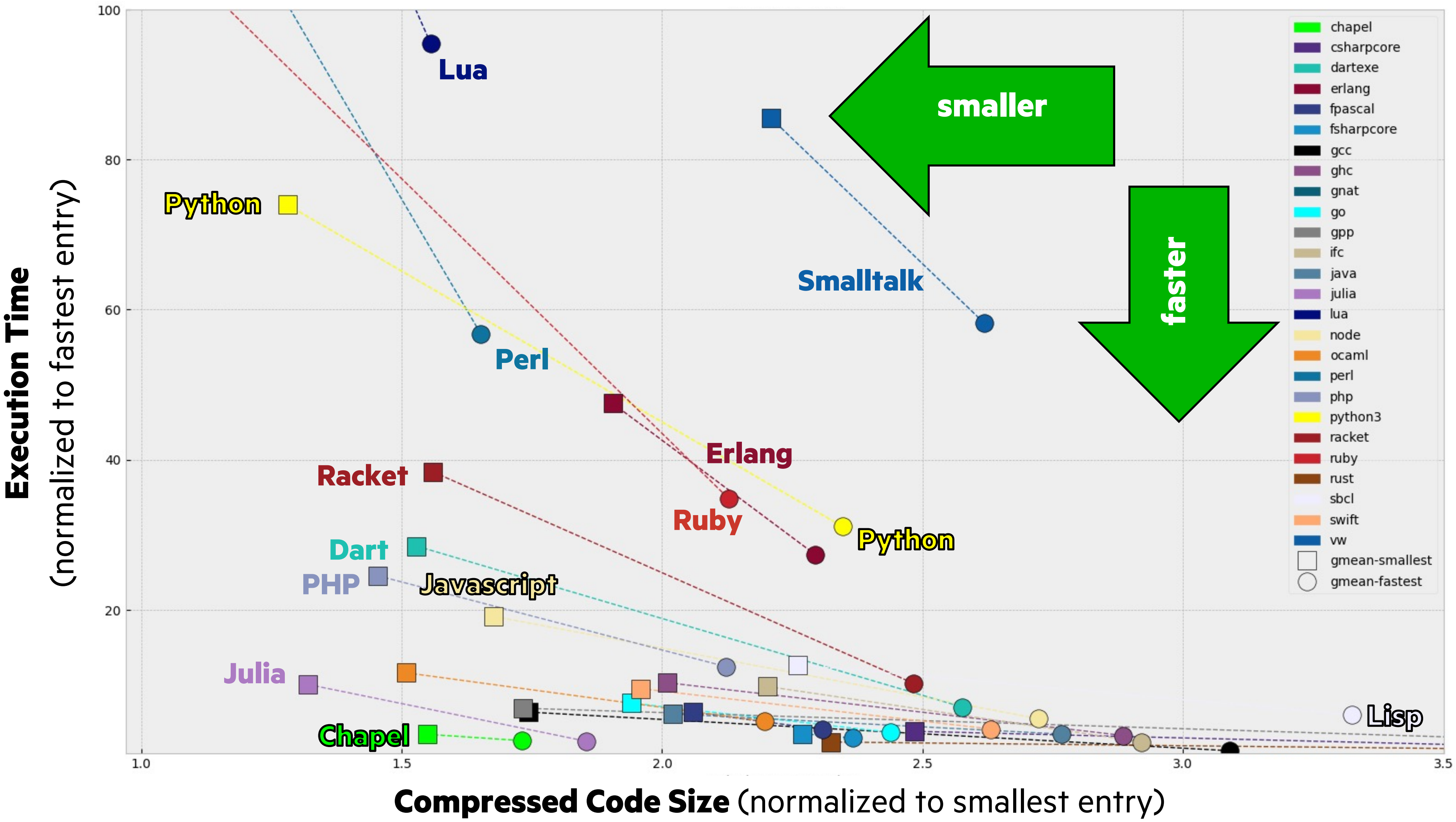
1. **compiled:** to generate the best performance possible
2. **statically typed:** to avoid simple errors after hours of execution
3. **interoperable:** with C, Fortran, Python, ...
4. **portable:** runs on laptops, clusters, the cloud, supercomputers
5. **open-source:** to reduce barriers to adoption and leverage community contributions

OUTLINE

- What is Chapel, and Why?
- Chapel Benchmarks and Apps
- Intro to Chapel, by Example
- Applications of Chapel
- Wrap-up

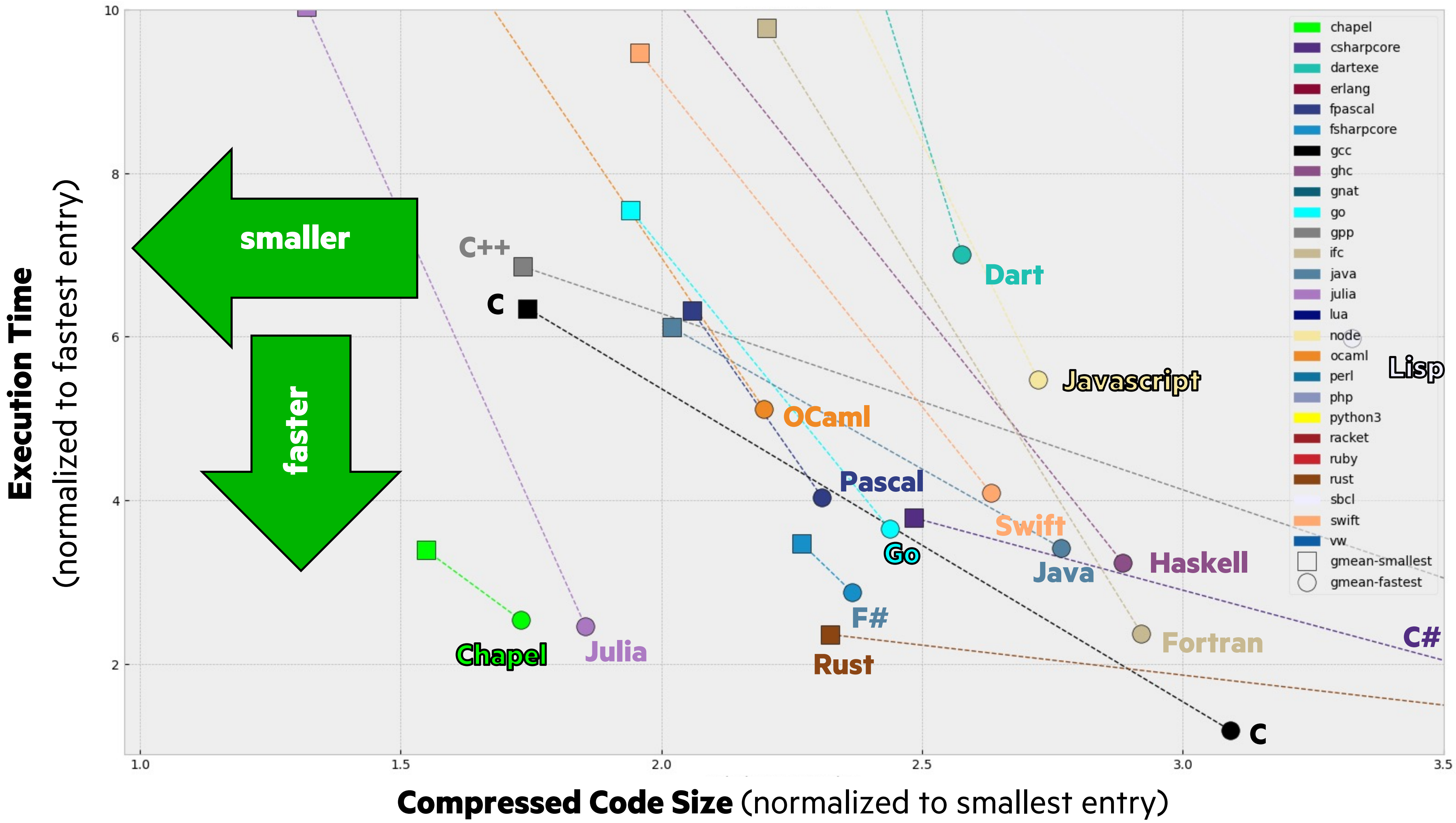
CHAPEL BENCHMARKS AND APPS

FOR DESKTOP BENCHMARKS, CHAPEL IS COMPACT AND FAST



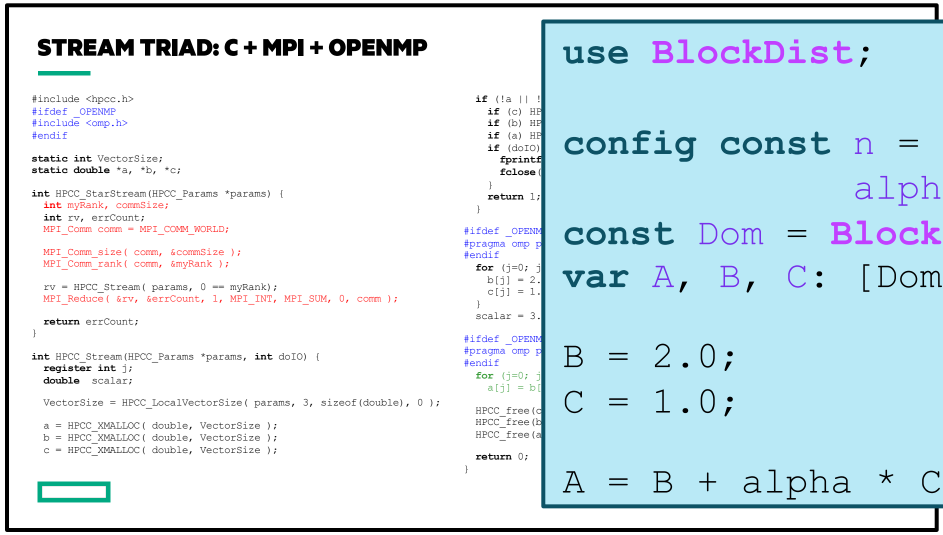
[plot generated by summarizing data from https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html as of Feb 8, 2023]

FOR DESKTOP BENCHMARKS, CHAPEL IS COMPACT AND FAST (ZOOMED)



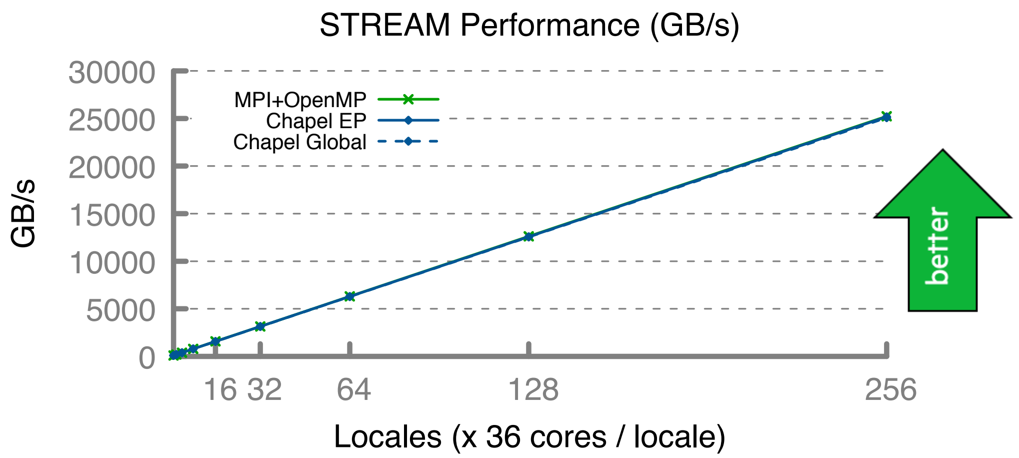
[plot generated by summarizing data from https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html as of Feb 8, 2023]

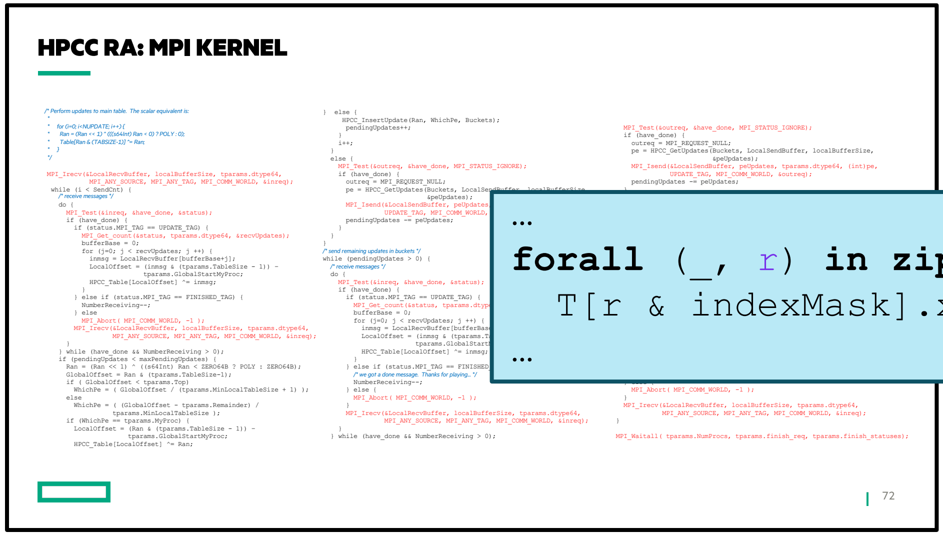
HPC BENCHMARKS: CONVENTIONAL APPROACHES VS. CHAPEL



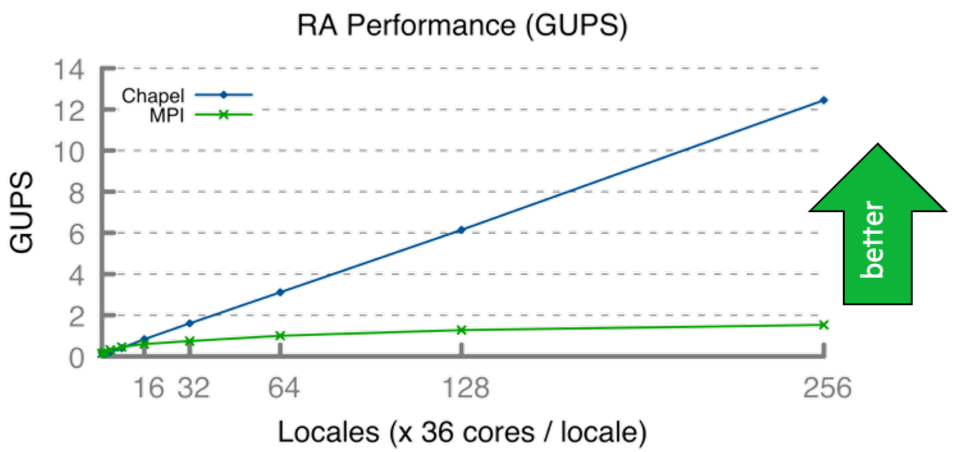

```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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 );
```

```
use BlockDist;

config const n = 1_000_000,
             alpha = 0.01;
const Dom = Block.createDomain({1..n});
var A, B, C: [Dom] real;

B = 2.0;
C = 1.0;

A = B + alpha * C;
```

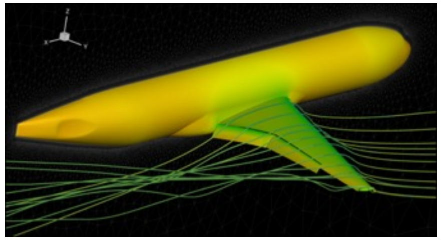




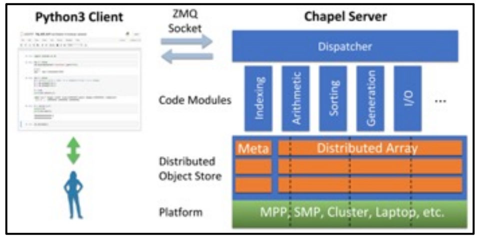

```
…
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
…
```



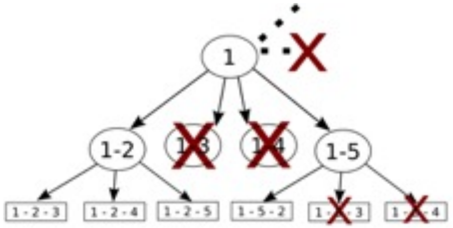
APPLICATIONS OF CHAPEL



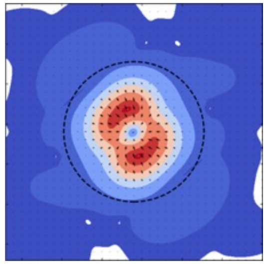
CHAMPS: 3D Unstructured CFD
Laurendeau, Bourgault-Côté, Parenteau, Plante, et al.
École Polytechnique Montréal



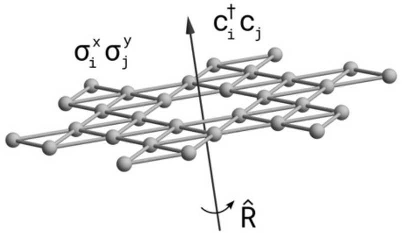
Arkouda: Interactive Data Science at Massive Scale
Mike Merrill, Bill Reus, et al.
U.S. DoD



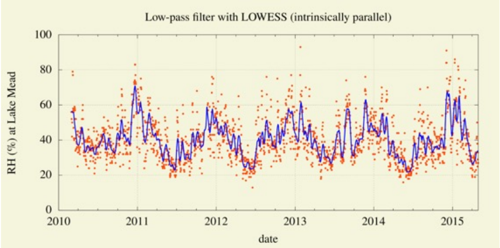
ChOp: Chapel-based Optimization
T. Carneiro, G. Helbecque, N. Melab, et al.
INRIA, IMEC, et al.



ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, et al.
Yale University et al.

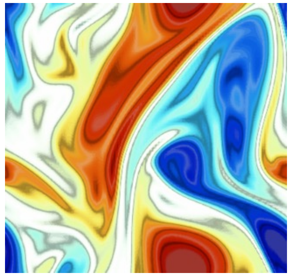


Lattice-Symmetries: a Quantum Many-Body Toolbox
Tom Westerhout
Radboud University

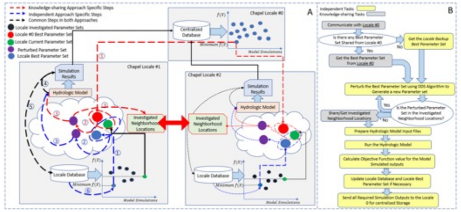


Desk dot chpl: Utilities for Environmental Eng.
Nelson Luis Dias
The Federal University of Paraná, Brazil

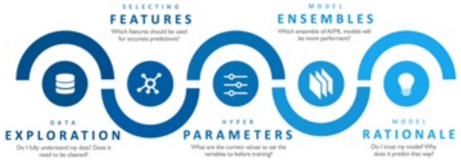
RapidQ: Mapping Coral Biodiversity
Rebecca Green, Helen Fox, Scott Bachman, et al.
The Coral Reef Alliance



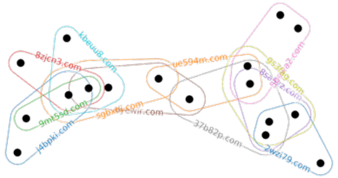
ChapQG: Layered Quasigeostrophic CFD
Ian Grooms and Scott Bachman
University of Colorado, Boulder et al.



Chapel-based Hydrological Model Calibration
Marjan Asgari et al.
University of Guelph



CrayAI HyperParameter Optimization (HPO)
Ben Albrecht et al.
Cray Inc. / HPE



CHGL: Chapel Hypergraph Library
Louis Jenkins, Cliff Joslyn, Jesun Firoz, et al.
PNNL

?

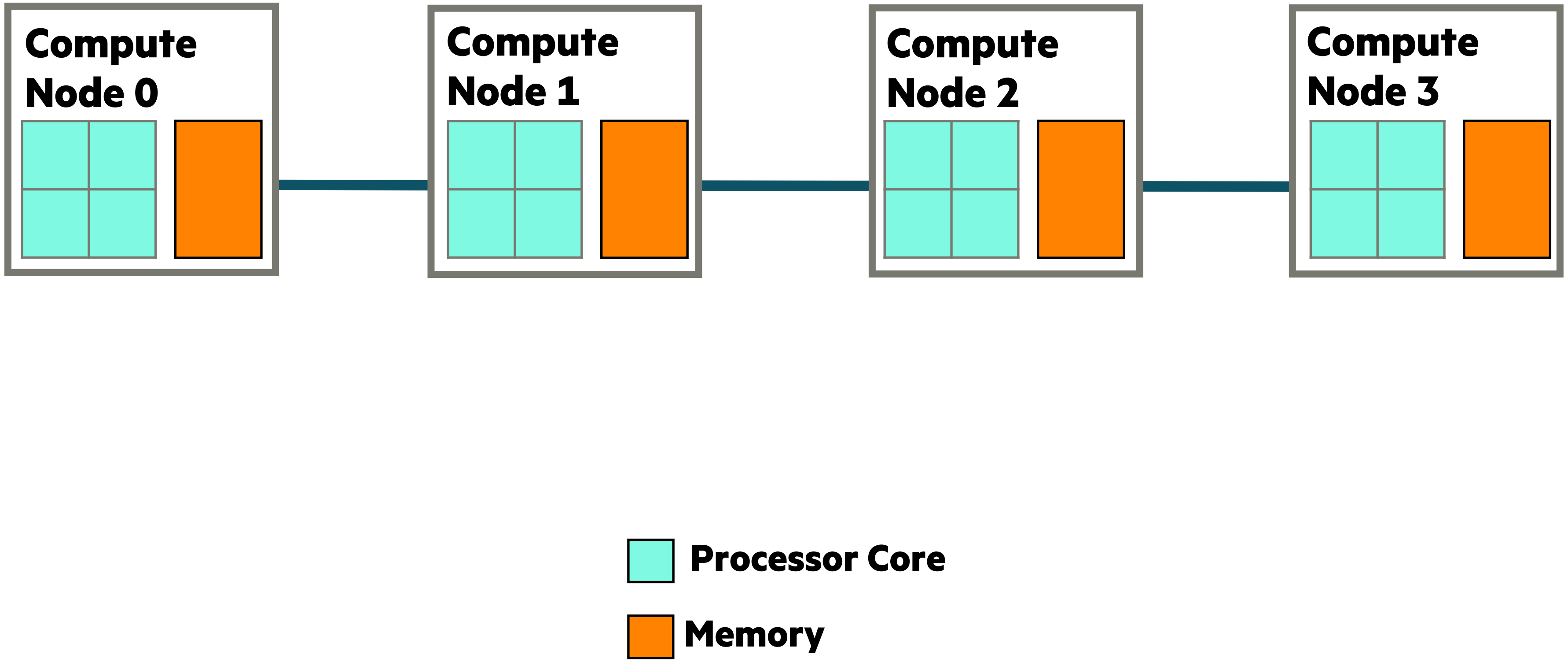
Your Application Here?

(images provided by their respective teams and used with permission)

INTRODUCTION TO CHAPEL, BY EXAMPLE

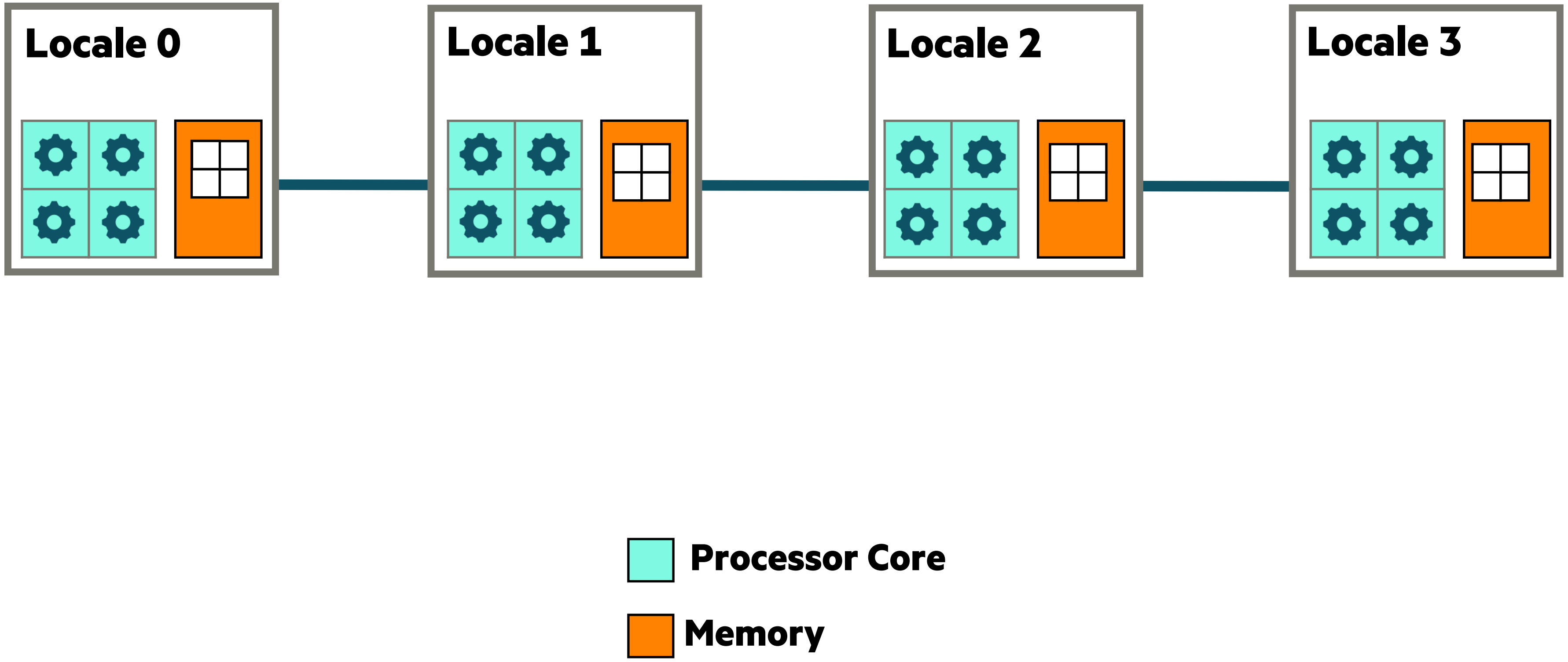
LOCALES IN CHAPEL

- In Chapel, a *locale* refers to a compute resource with…
  - processors, so it can run tasks
  - memory, so it can store variables
- For now, think of each compute node as being a locale



KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** What tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?



BASIC FEATURES FOR LOCALITY

basics-on.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

on Locales[1] {
  var B: [1..2, 1..2] real;

  B = 2 * A;
}
```

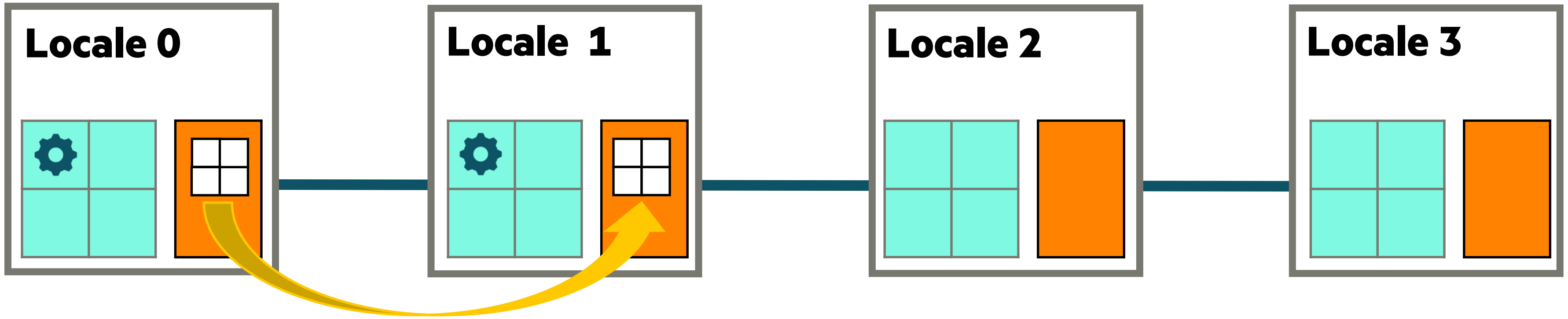
All Chapel programs begin running as a single task on locale 0

Variables are stored using the memory local to the current task

on-clauses move tasks to other locales

remote variables can be accessed directly

This is a serial, but distributed computation



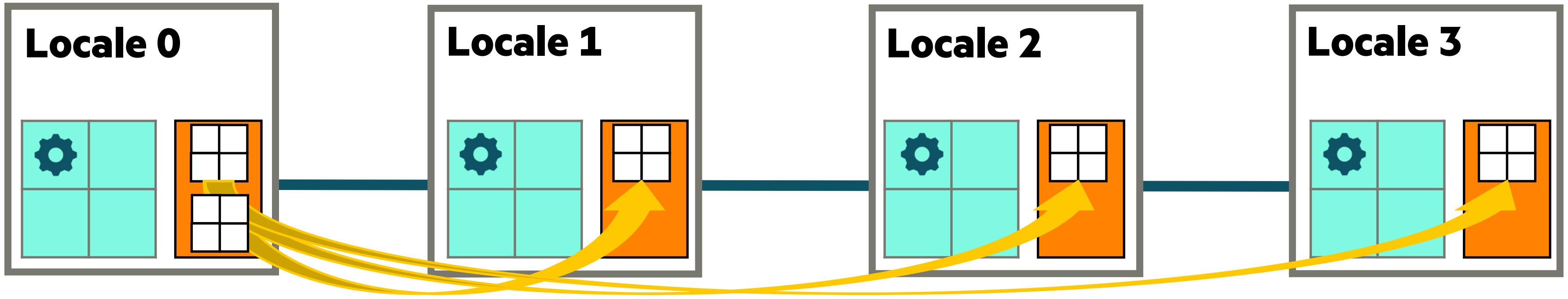
BASIC FEATURES FOR LOCALITY

basics-for.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

for loc in Locales {
  on loc {
    var B = A;
  }
}
```

This loop will serially iterate over the program's locales

This is also a serial, but distributed computation



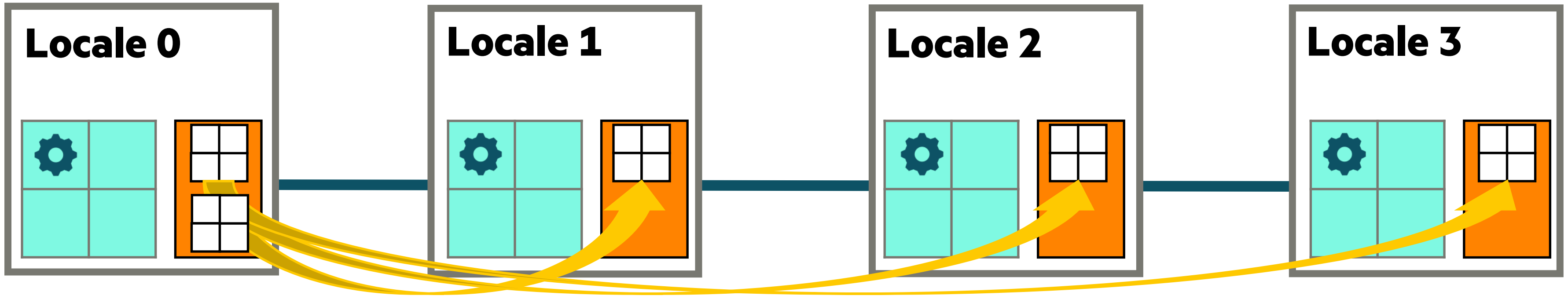
MIXING LOCALITY WITH TASK PARALLELISM

basics-coforall.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

coforall loc in Locales {
  on loc {
    var B = A;
  }
}
```

The coforall loop creates a parallel task per iteration

This results in a parallel distributed computation



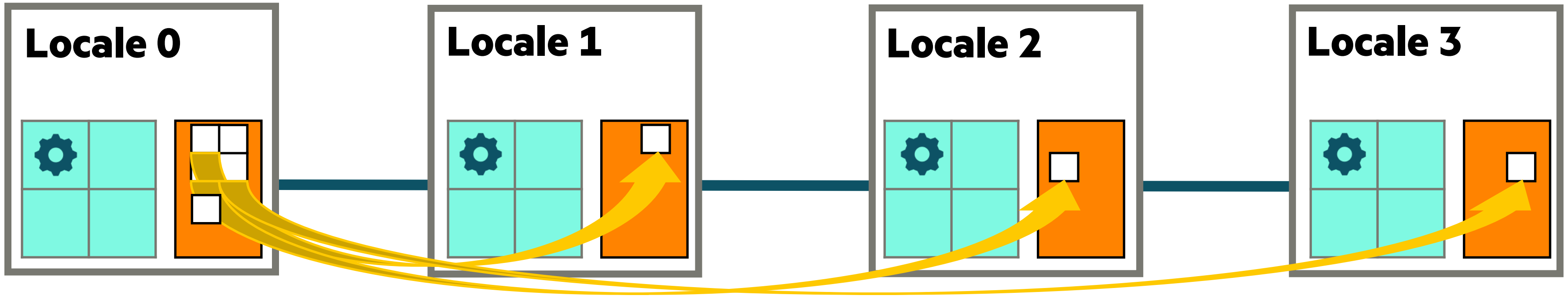
ARRAY-BASED PARALLELISM AND LOCALITY

basics-distarr.chpl

```
writeln("Hello from locale ", here.id);

var A: [1..2, 1..2] real;

use BlockDist;

var D = Block.createDomain({1..2, 1..2});
var B: [D] real;
B = A;
```

Chapel also supports distributed domains (index sets) and arrays

They also result in parallel distributed computation

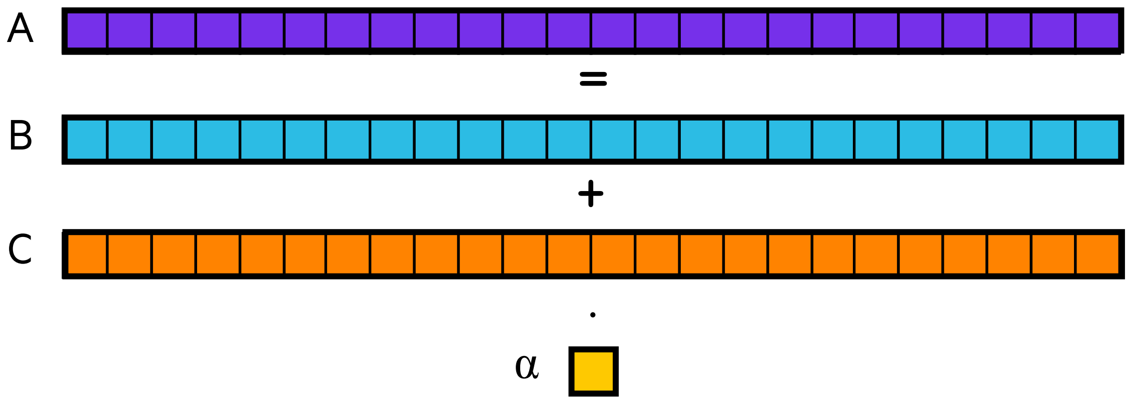


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: *n*-element vectors *A*, *B*, *C*

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

In pictures:

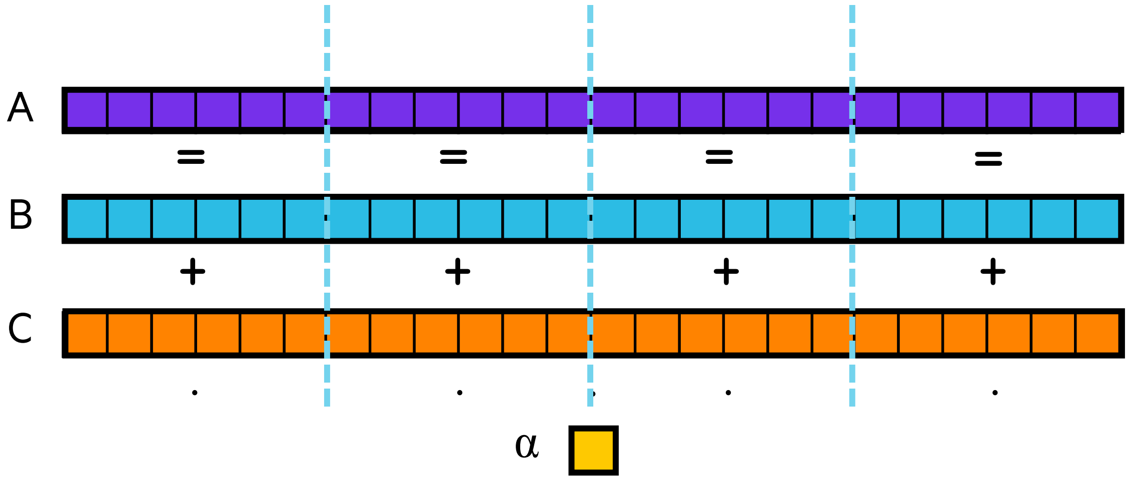


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: n-element vectors A, B, C

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

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

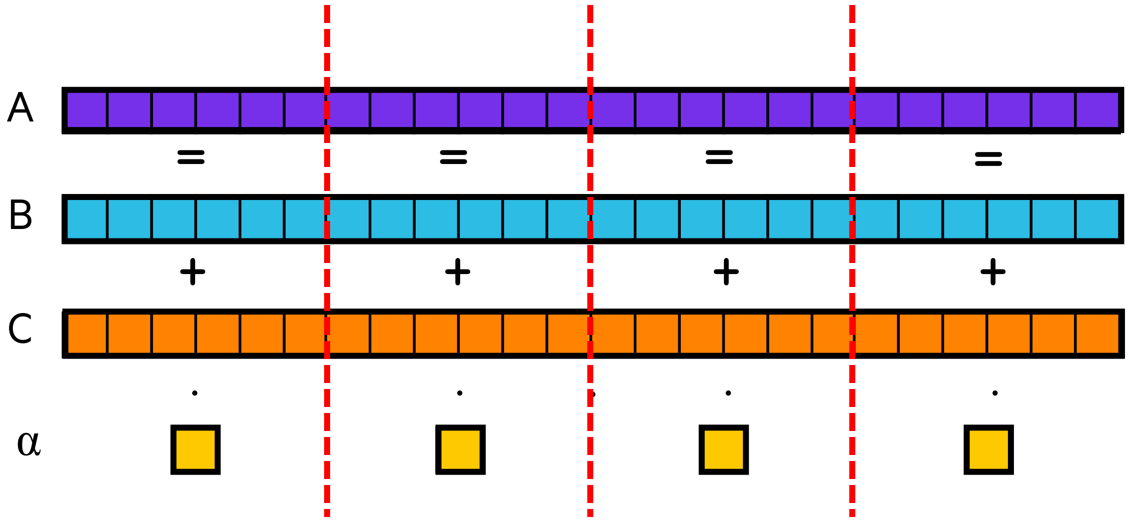


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: *n*-element vectors *A*, *B*, *C*

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

In pictures, in parallel (distributed memory)**:**

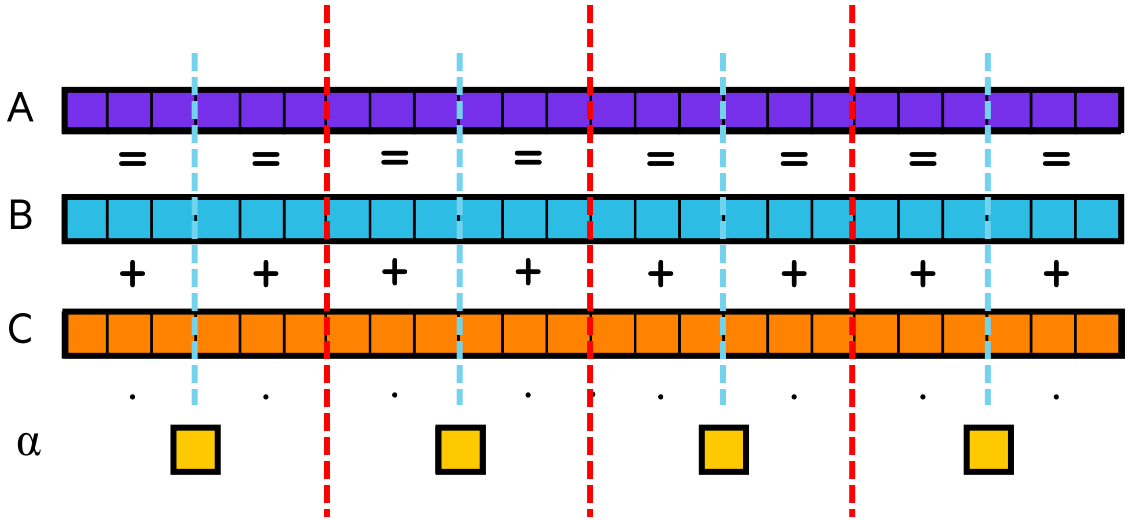


STREAM TRIAD: A TRIVIAL CASE OF PARALLELISM + LOCALITY

Given: n-element vectors A, B, C

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

In pictures, in parallel (distributed memory multicore)**:**



STREAM TRIAD: SHARED MEMORY

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;
```

'config' declarations support command-line overrides

```
$ chpl stream-ep.chpl
$ ./stream-ep
$ ./stream-ep --n=10 --alpha=3.0
```

compile the program

run with the default values

override those values

n α

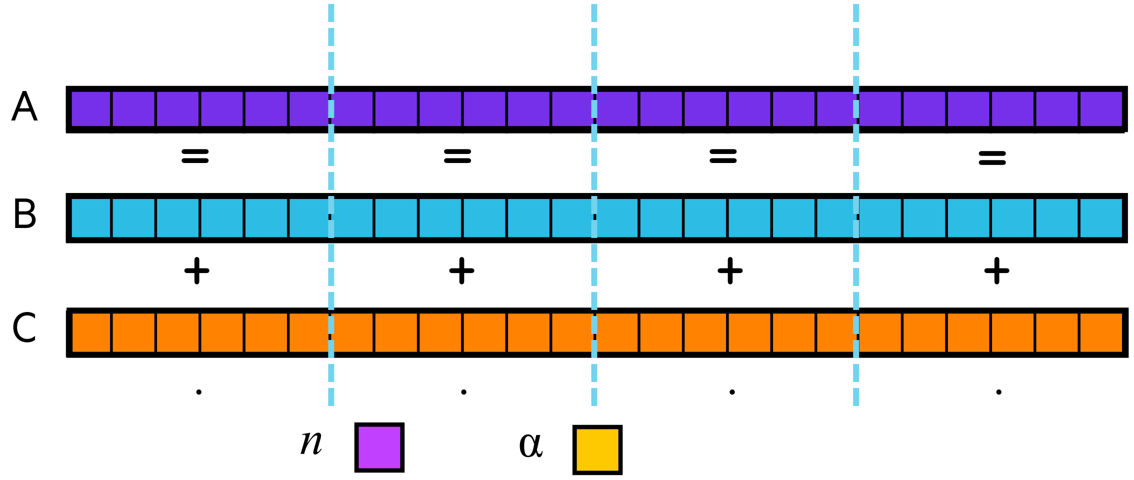
STREAM TRIAD: SHARED MEMORY

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;



    var A, B, C: [1..n] real;
    A = B + alpha * C;
```

declare three arrays of size 'n'

whole-array operations result in parallel computation



So far, this is simply a multi-core program

Nothing refers to remote locales, explicitly or implicitly

STREAM TRIAD: DISTRIBUTED MEMORY (EP VERSION)

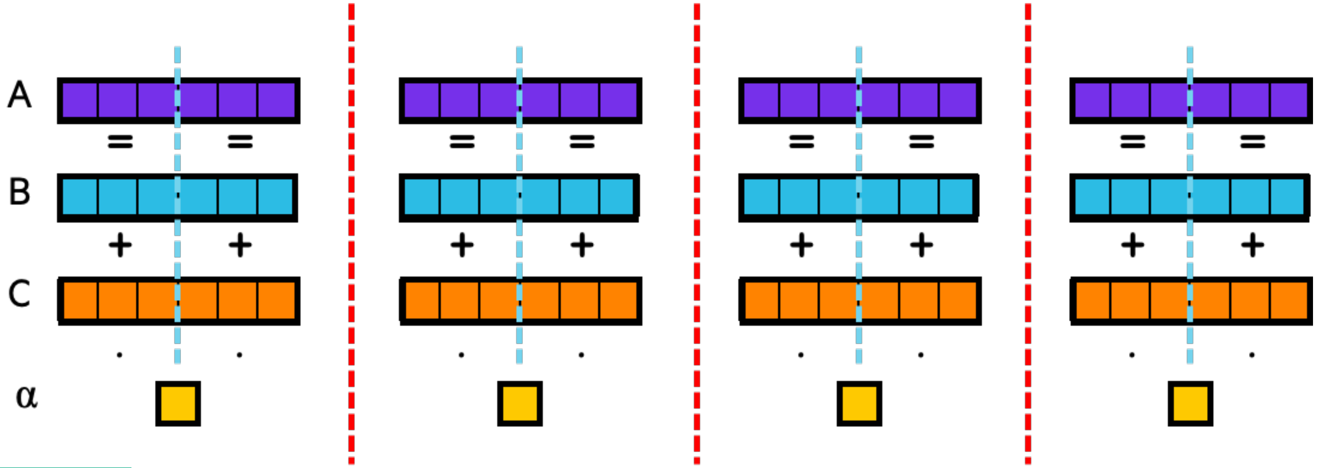
stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

coforall loc in Locales {
  on loc {
    var A, B, C: [1..n] real;
    A = B + alpha * C;
  }
}
```

create a task per locale…

…running ‘on’ its locale

then run multi-core Stream on local arrays, as before



STREAM TRIAD: DISTRIBUTED MEMORY (GLOBAL VERSION)

stream-glbl.chpl

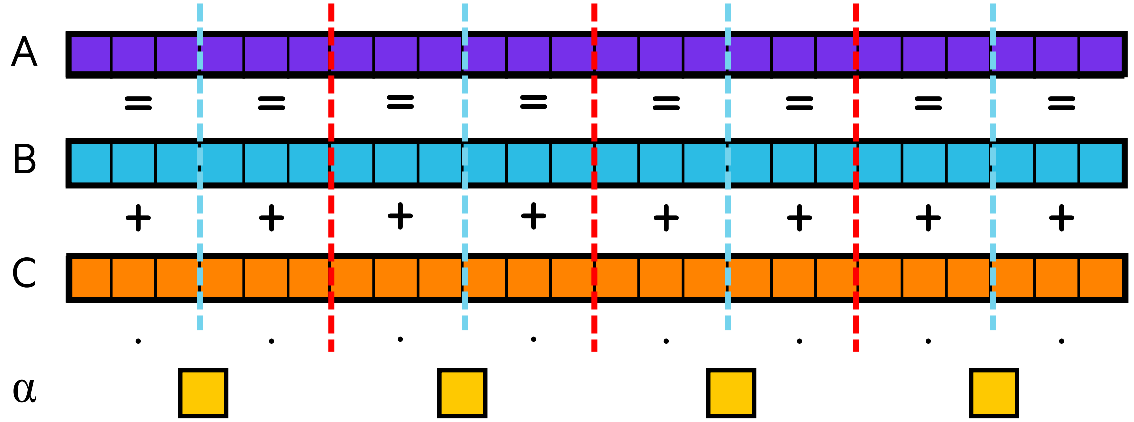
```
config const n = 1_000_000,
             alpha = 0.01;

use BlockDist;

const Dom = Block.createDomain({1..n});
var A, B, C: [Dom] real;

A = B + alpha * C;
```

'use' the standard block-distribution module

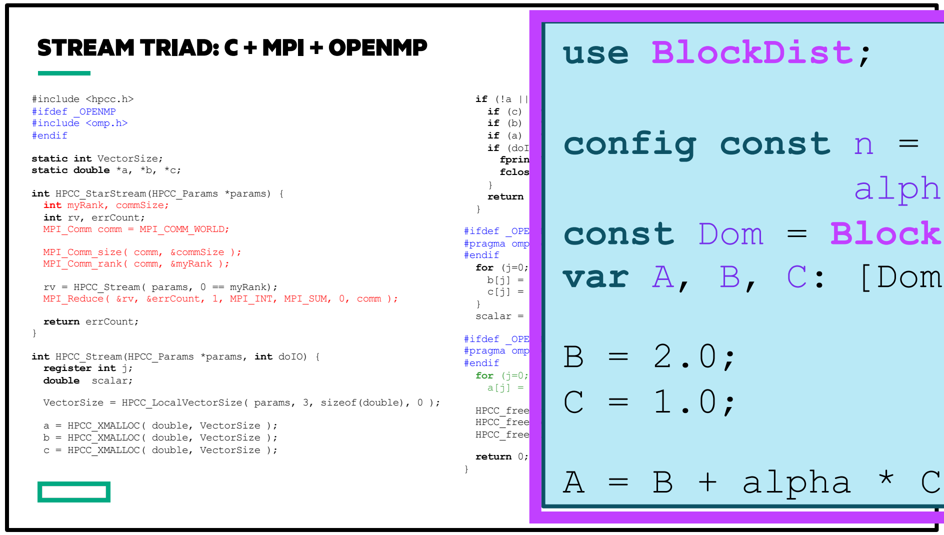
create a distributed domain (index set)...

...and distributed arrays

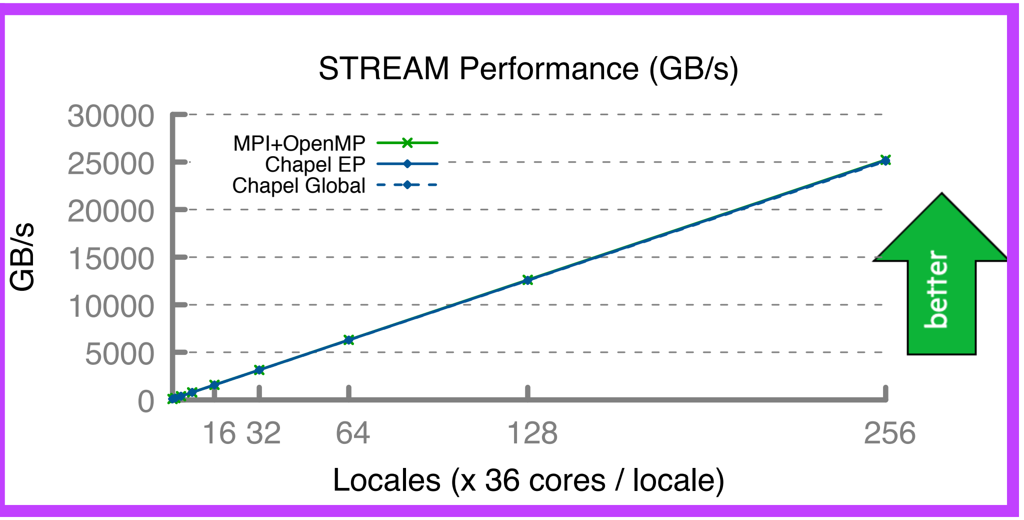
these whole-array operations will use all cores on all locales

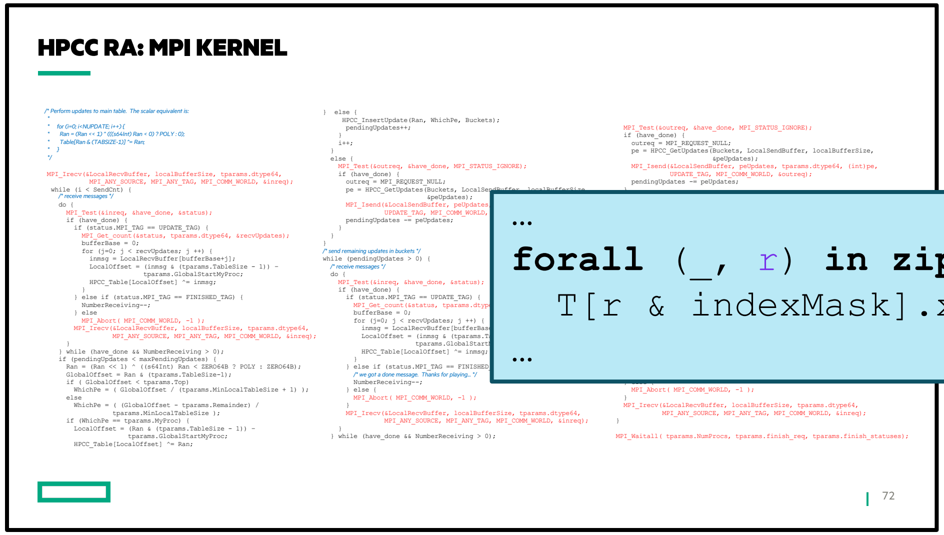


HPC BENCHMARKS: CONVENTIONAL APPROACHES VS. CHAPEL

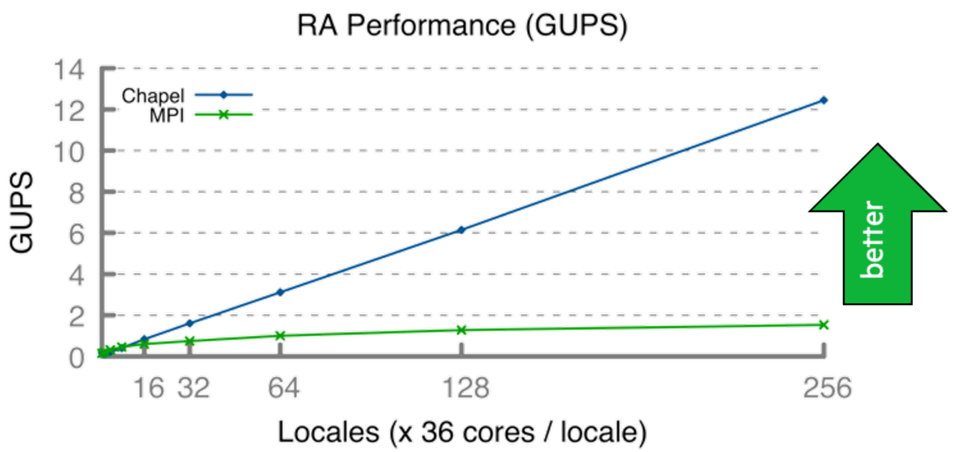



```
use BlockDist;

config const n = 1_000_000,
             alpha = 0.01;
const Dom = Block.createDomain({1..n});
var A, B, C: [Dom] real;

B = 2.0;
C = 1.0;

A = B + alpha * C;
```

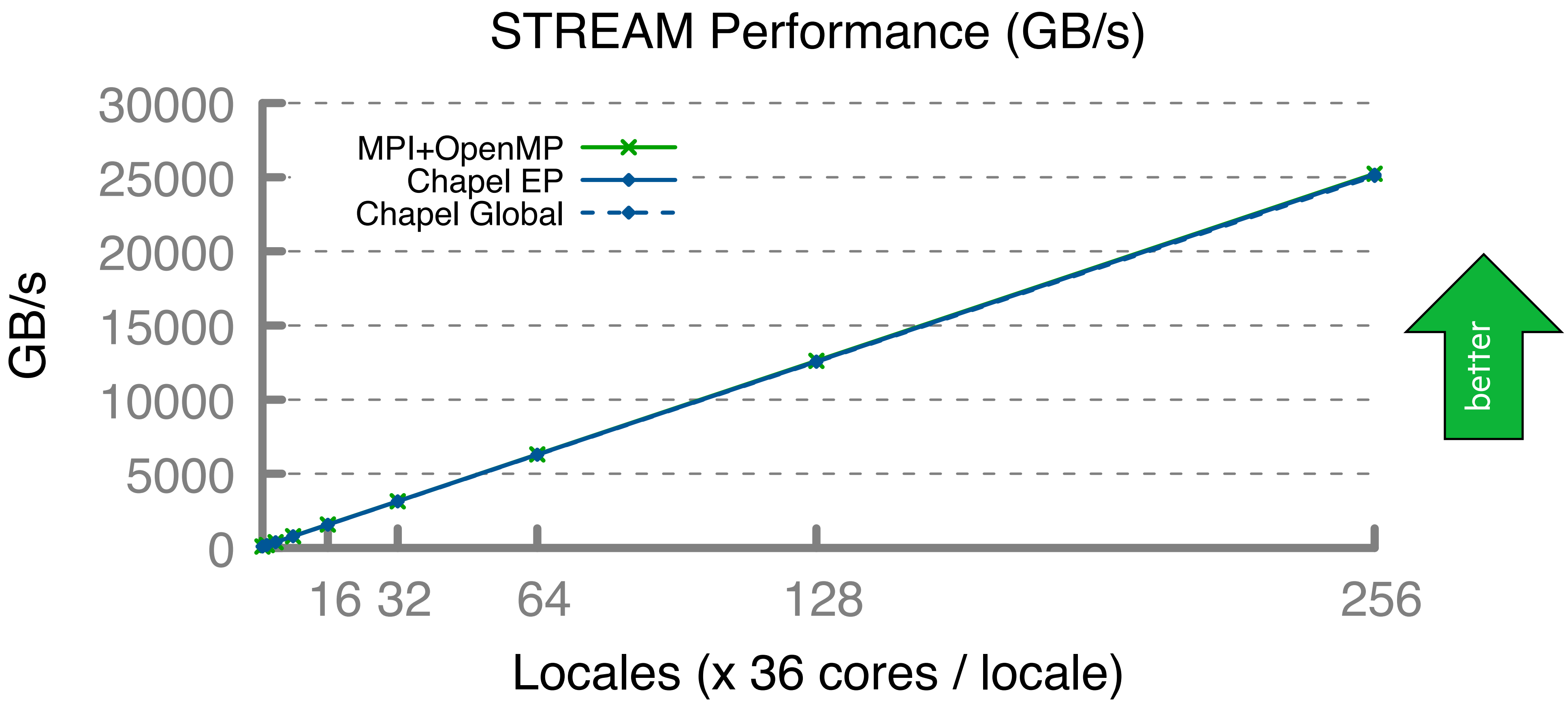





```
…
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
…
```

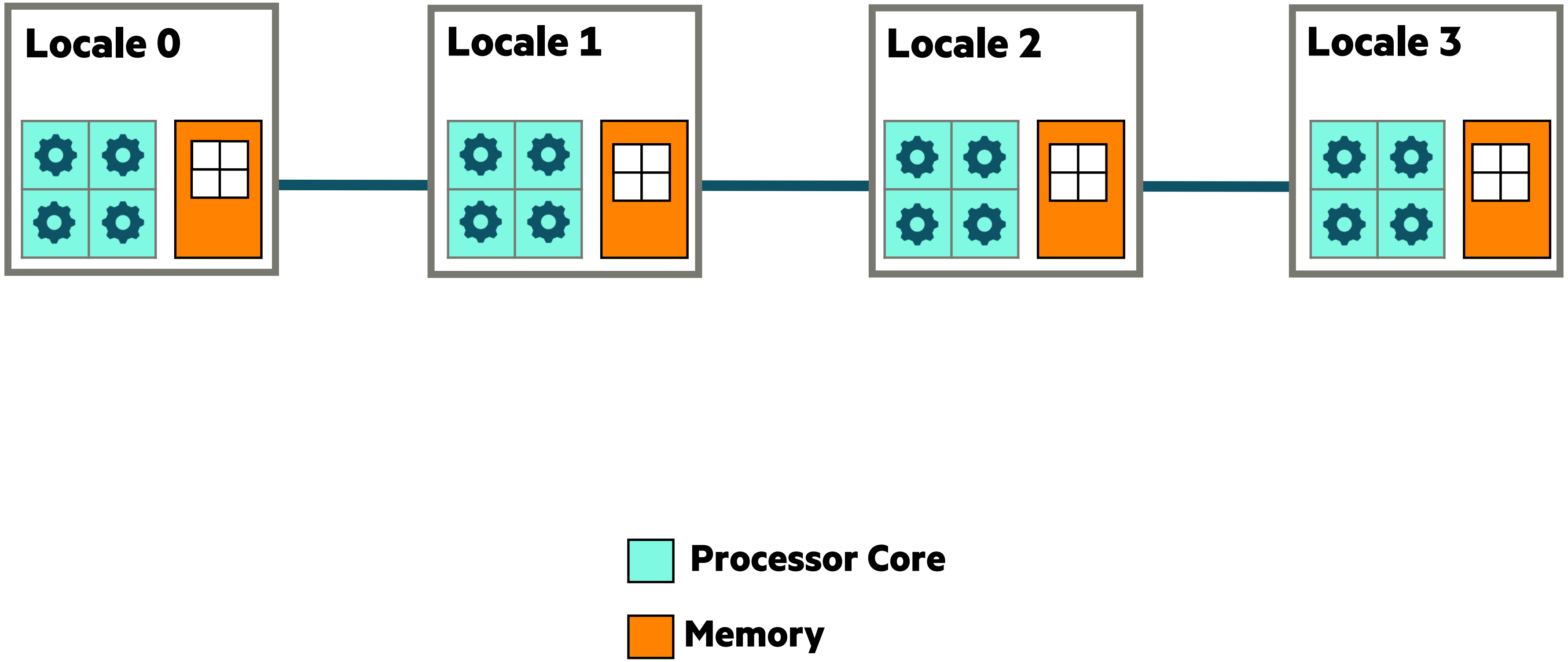


STREAM TRIAD: MPI + OPENMP VS. CHAPEL



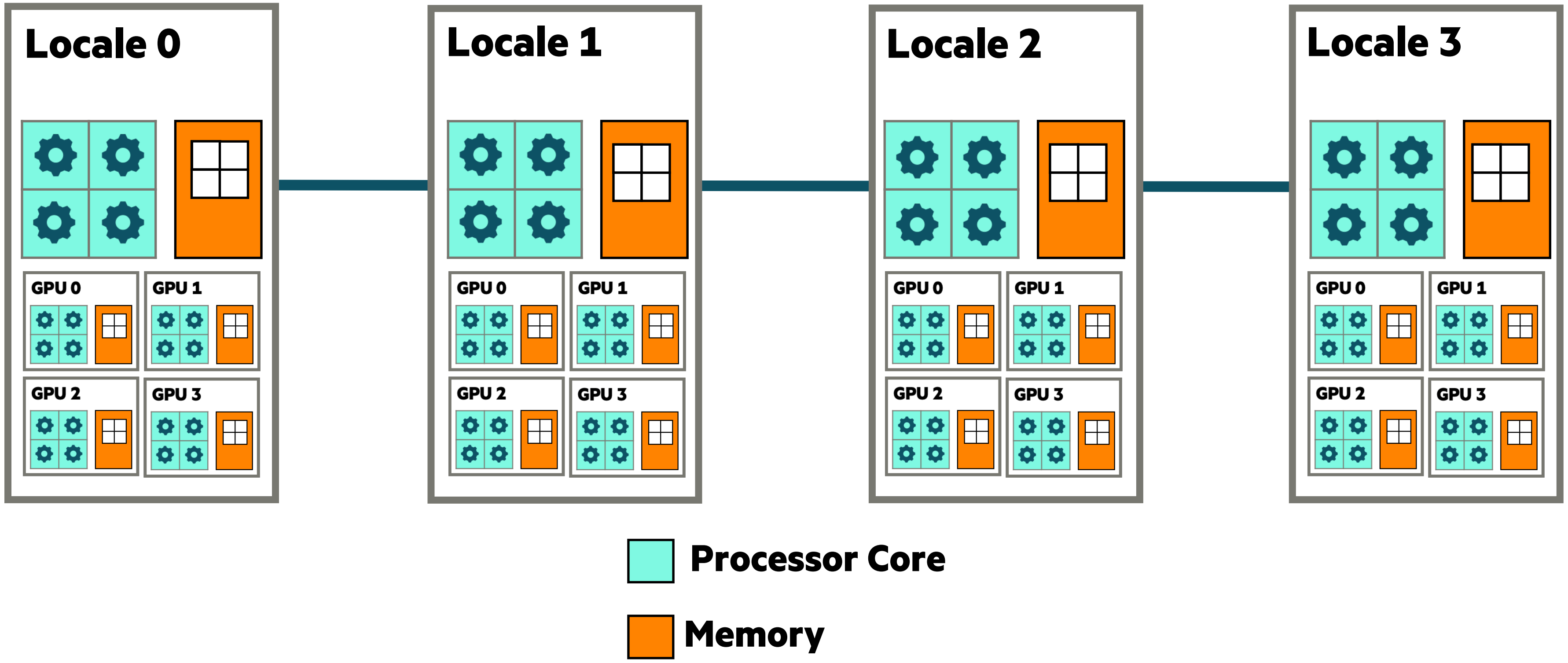
KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** What tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?
   - complicating matters, compute nodes now often have GPUs with their own processors and memory



KEY CONCERNS FOR SCALABLE PARALLEL COMPUTING

1. **parallelism:** What tasks should run simultaneously?
2. **locality:** Where should tasks run? Where should data be allocated?
   - complicating matters, compute nodes now often have GPUs with their own processors and memory
   - we represent these as *sub-locales* in Chapel



OAK RIDGE NATIONAL LABORATORY'S FRONTIER SUPERCOMPUTER

- **74 HPE Cray EX cabinets**
- **9,408 AMD CPUs, 37,632 AMD GPUs**
- **700 petabytes of storage capacity, peak write speeds of 5 terabytes per second using Cray ClusterStor storage system**
- **HPE Slingshot networking cables providing 100 GB/s network bandwidth.**

TOP500

Built by HPE, ORNL's Frontier supercomputer is #1 on the TOP500.

1.1 exaflops of performance.

GREEN500

Built by HPE, ORNL's TDS and full system are ranked #2 & #6 on the Green500.

62.68 gigaflops/watt power efficiency for ORNL's TDS system, **52.23 gigaflops/watt** power efficiency for full system.

HPL-MxP

Built by HPE, ORNL's Frontier supercomputer is #1 on the HPL-MxP list.

7.9 exaflops on the HPL-MxP benchmark (formerly HPL-AI).

Source: May 30, 2022 Top500 release, HPL-MxP mixed-precision benchmark (formerly HPL-AI).

STREAM TRIAD: DISTRIBUTED MEMORY, CPUS ONLY

stream-glbl.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

use BlockDist;

const Dom = Block.createDomain({1..n});
var A, B, C: [Dom] real;

A = B + alpha * C;
```

These programs are both CPU-only

Nothing refers to GPUs, explicitly or implicitly

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

coforall loc in Locales {
  on loc {
    var A, B, C: [1..n] real;
    A = B + alpha * C;
  }
}
```

STREAM TRIAD: DISTRIBUTED MEMORY, GPUS ONLY

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

coforall loc in Locales {
  on loc {

      coforall gpu in here.gpus do on gpu {
        var A, B, C: [1..n] real;
        A = B + alpha * C;
      }






  }
}
```

Use a similar 'coforall' + 'on' idiom to run a Triad concurrently on each of this locale's GPUs

This is a GPU-only program

Nothing other than coordination code runs on the CPUs

STREAM TRIAD: DISTRIBUTED MEMORY, GPUS AND CPUS

stream-ep.chpl

```
config const n = 1_000_000,
             alpha = 0.01;

coforall loc in Locales {
  on loc {
    cobegin {
      coforall gpu in here.gpus do on gpu {
        var A, B, C: [1..n] real;
        A = B + alpha * C;
      }
      {
        var A, B, C: [1..n] real;
        A = B + alpha * C;
      }
    }
  }
}
```

'cobegin { … }' creates a task per child statement

one task runs our multi-GPU triad

the other runs the multi-CPU triad

This program uses all CPUs and GPUs across all of our compute nodes

STREAM TRIAD: DISTRIBUTED MEMORY, GPUS AND CPUS (REFACTOR)

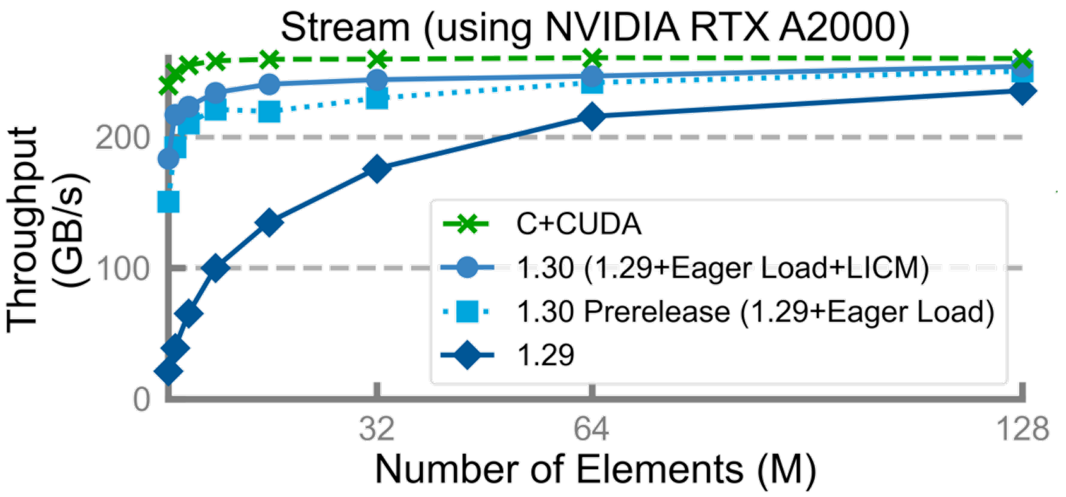
stream-ep.chpl

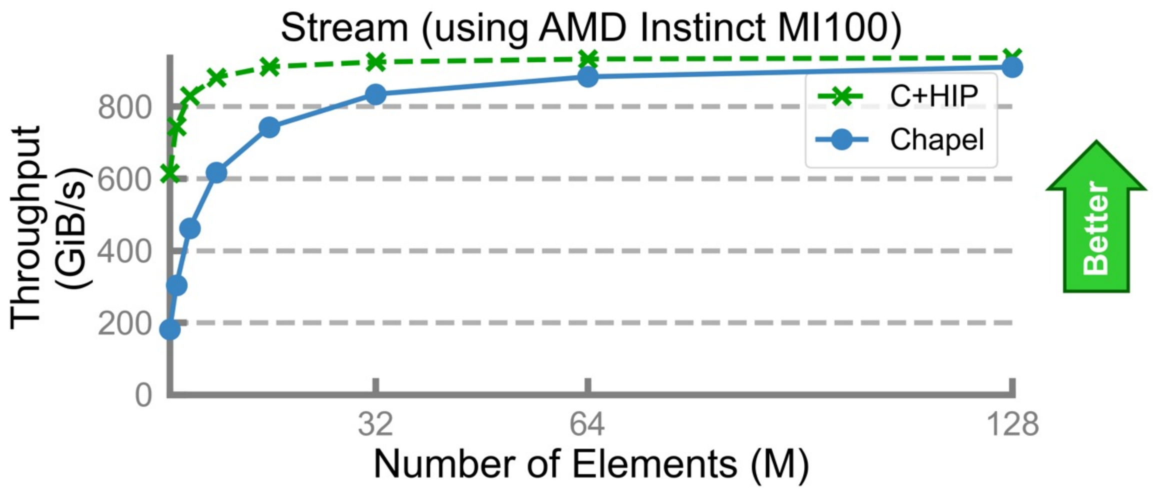
```
config const n = 1_000_000,
             alpha = 0.01;

coforall loc in Locales {
  on loc {
    cobegin {
      coforall gpu in here.gpus do on gpu {
        runTriad();
      }
      runTriad();
    }
  }
}
proc runTriad() {
   var A, B, C: [1..n] real;
   A = B + alpha * C;
}
```

we can also refactor the repeated code into a procedure for re-use

the compiler creates CPU and GPU versions of this procedure

STREAM TRIAD: GPU PERFORMANCE VS. REFERENCE VERSIONS

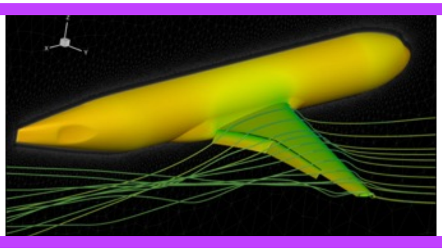




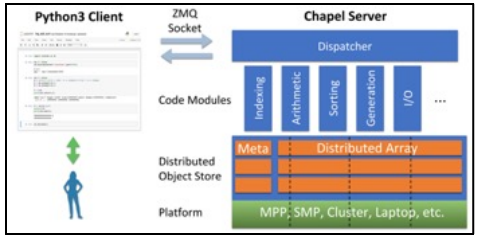
Performance vs. reference versions has become increasingly competitive over the past 4 months

APPLICATIONS OF CHAPEL

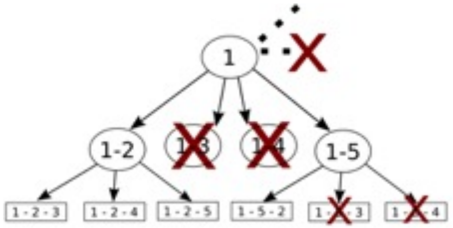
APPLICATIONS OF CHAPEL



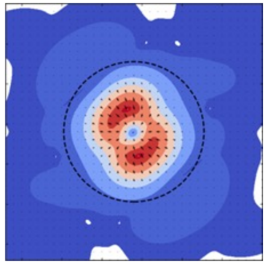
CHAMPS: 3D Unstructured CFD
Laurendeau, Bourgault-Côté, Parenteau, Plante, et al.
École Polytechnique Montréal



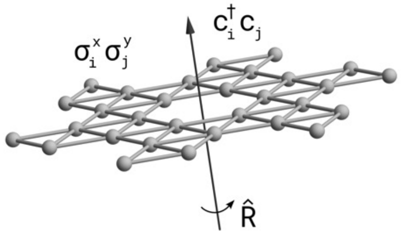
Arkouda: Interactive Data Science at Massive Scale
Mike Merrill, Bill Reus, et al.
U.S. DoD



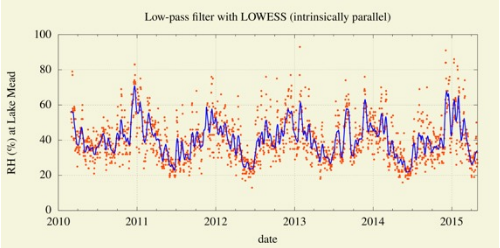
ChOp: Chapel-based Optimization
T. Carneiro, G. Helbecque, N. Melab, et al.
INRIA, IMEC, et al.



ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, et al.
Yale University et al.

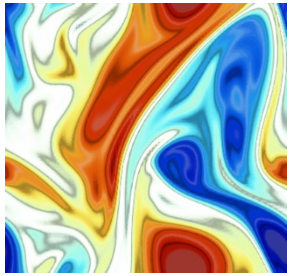


Lattice-Symmetries: a Quantum Many-Body Toolbox
Tom Westerhout
Radboud University

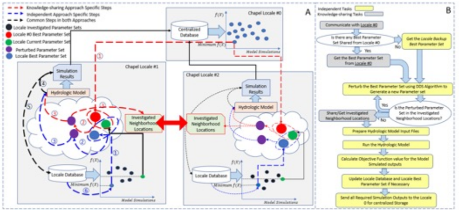


Desk dot chpl: Utilities for Environmental Eng.
Nelson Luis Dias
The Federal University of Paraná, Brazil

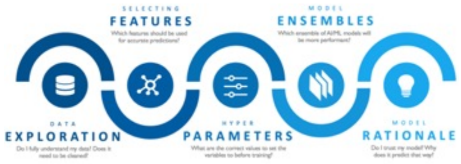
RapidQ: Mapping Coral Biodiversity
Rebecca Green, Helen Fox, Scott Bachman, et al.
The Coral Reef Alliance



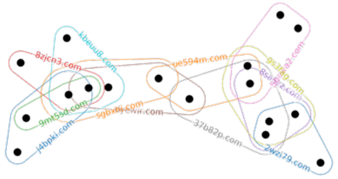
ChapQG: Layered Quasigeostrophic CFD
Ian Grooms and Scott Bachman
University of Colorado, Boulder et al.



Chapel-based Hydrological Model Calibration
Marjan Asgari et al.
University of Guelph



CrayAI HyperParameter Optimization (HPO)
Ben Albrecht et al.
Cray Inc. / HPE



CHGL: Chapel Hypergraph Library
Louis Jenkins, Cliff Joslyn, Jesun Firoz, et al.
PNNL

?

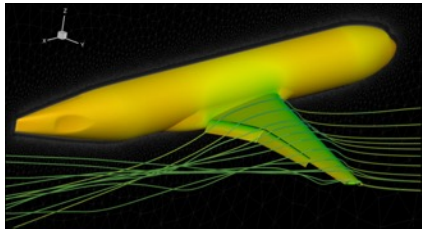
Your Application Here?

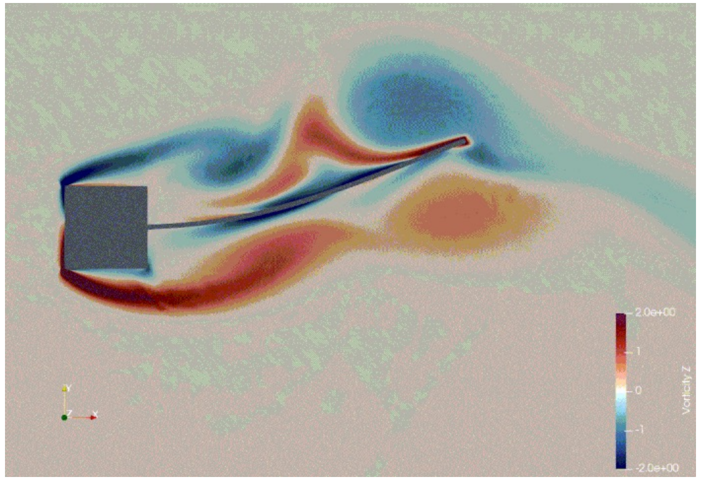
(images provided by their respective teams and used with permission)

CHAMPS SUMMARY

What is it?

- 3D unstructured CFD framework for airplane simulation
- ~85k lines of Chapel written from scratch in ~3 years

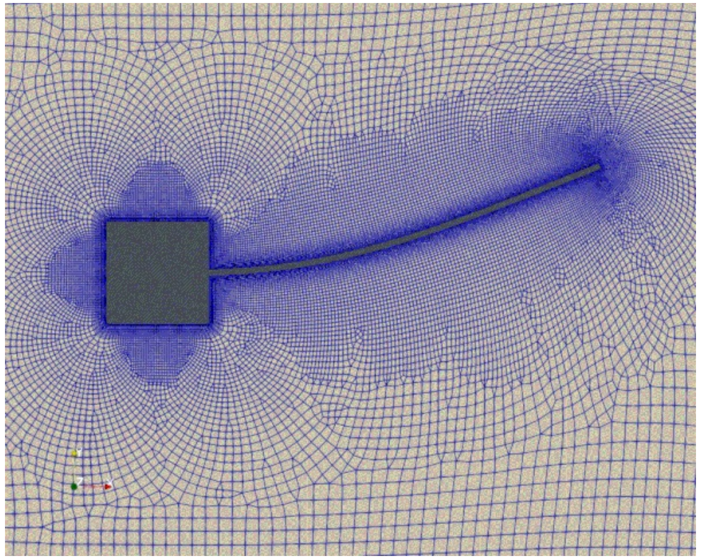




Who wrote it?

- Professor Éric Laurendeau’s students + postdocs at Polytechnique Montreal

POLYTECHNIQUE MONTRÉAL



Why Chapel?

- performance and scalability competitive with MPI + C++
- students found it far more productive to use
- enabled them to compete with more established CFD centers

(images provided by the CHAMPS team and used with permission)

CHAMPS: EXCERPT FROM ÉRIC'S CHIUW 2021 KEYNOTE (TRANSCRIPT)

HPC Lessons From 30 Years of Practice in CFD Towards Aircraft Design and Analysis (June 4, 2021)

"To show you what Chapel did in our lab… [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… ***So, I've got industrial-type code in 48k lines.****"*

"[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."*

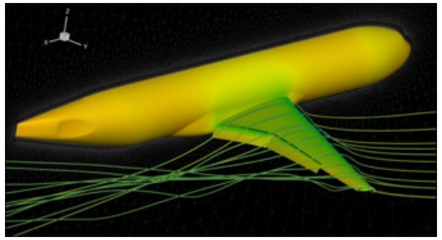
POLYTECHNIQUE MONTRÉAL

"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."*

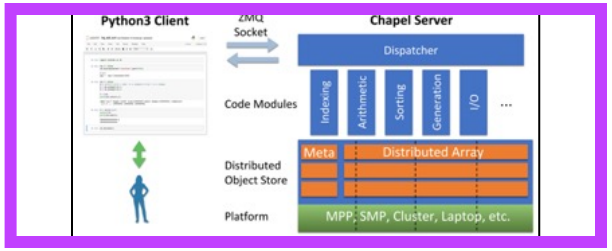
- Talk available online: https://youtu.be/wD-a_KyB8aI?t=1904 (hyperlink jumps to the section quoted here)

(images provided by the CHAMPS team and used with permission)

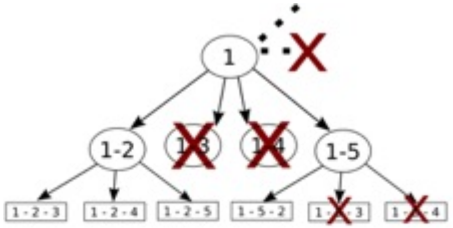
APPLICATIONS OF CHAPEL



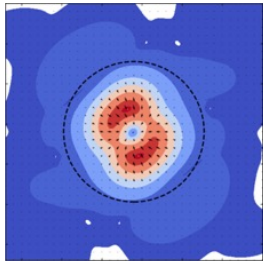
CHAMPS: 3D Unstructured CFD
Laurendeau, Bourgault-Côté, Parenteau, Plante, et al.
École Polytechnique Montréal



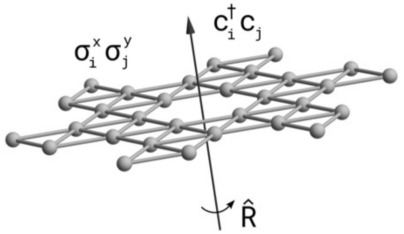
Arkouda: Interactive Data Science at Massive Scale
Mike Merrill, Bill Reus, et al.
U.S. DoD



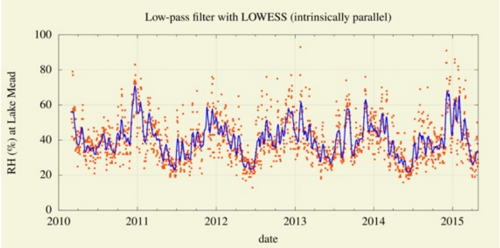
ChOp: Chapel-based Optimization
T. Carneiro, G. Helbecque, N. Melab, et al.
INRIA, IMEC, et al.



ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, et al.
Yale University et al.

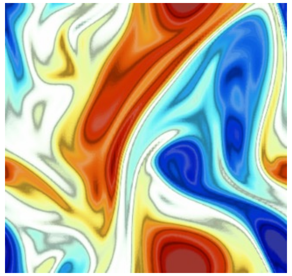


Lattice-Symmetries: a Quantum Many-Body Toolbox
Tom Westerhout
Radboud University

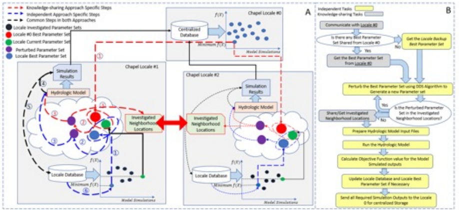


Desk dot chpl: Utilities for Environmental Eng.
Nelson Luis Dias
The Federal University of Paraná, Brazil

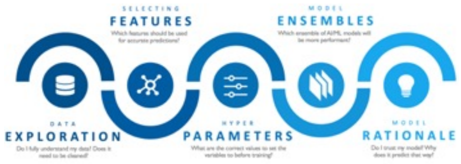
RapidQ: Mapping Coral Biodiversity
Rebecca Green, Helen Fox, Scott Bachman, et al.
The Coral Reef Alliance



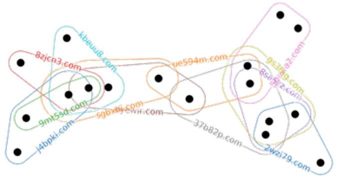
ChapQG: Layered Quasigeostrophic CFD
Ian Grooms and Scott Bachman
University of Colorado, Boulder et al.



Chapel-based Hydrological Model Calibration
Marjan Asgari et al.
University of Guelph



CrayAI HyperParameter Optimization (HPO)
Ben Albrecht et al.
Cray Inc. / HPE



CHGL: Chapel Hypergraph Library
Louis Jenkins, Cliff Joslyn, Jesun Firoz, et al.
PNNL

?

Your Application Here?

(images provided by their respective teams and used with permission)

DATA SCIENCE IN PYTHON AT SCALE?

Motivation: Imagine you've got…

…HPC-scale data science problems to solve

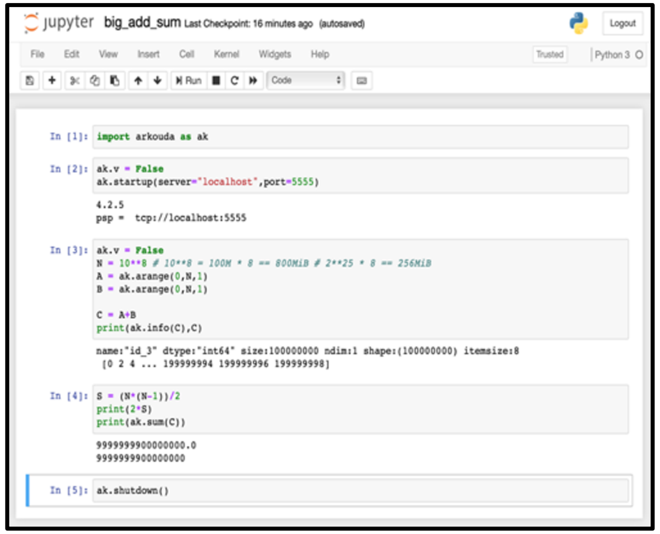
…a bunch of Python programmers

…access to HPC systems

How will you leverage your Python programmers to get your work done?

ARKOUDA: A PYTHON FRAMEWORK FOR INTERACTIVE HPC

Arkouda Client
(written in Python)



Arkouda Server
(written in Chapel)

User writes Python code in Jupyter, making familiar NumPy/Pandas calls

ARKOUDA SUMMARY

What is it?

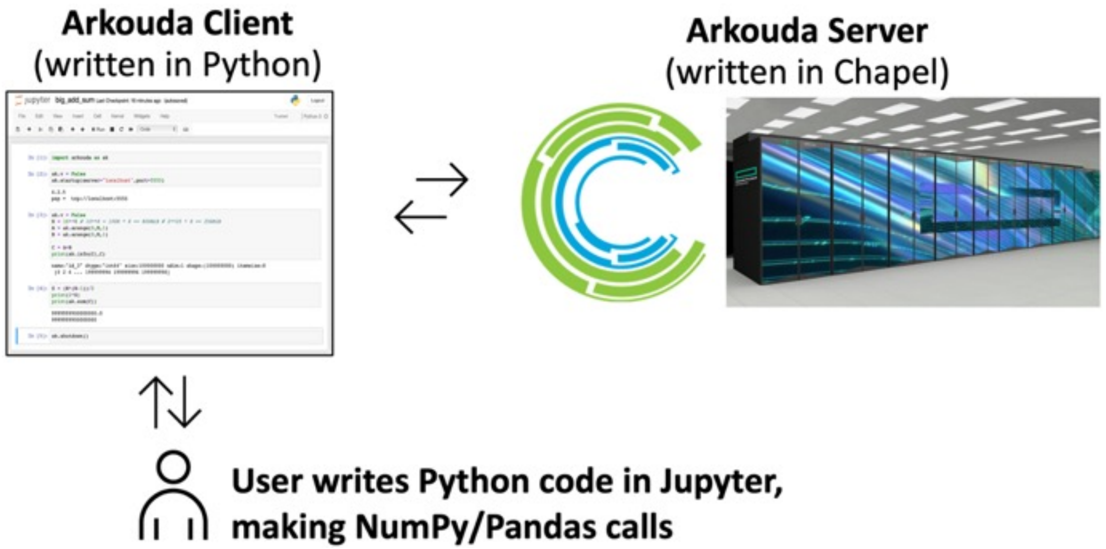
- A Python client-server framework supporting interactive supercomputing
  - Computes massive-scale results (TB-scale arrays) within the human thought loop (seconds to a few minutes)
  - Initial focus has been on a key subset of NumPy and Pandas for Data Science
- ~30k lines of Chapel + ~25k lines of Python, written since 2019
- Open-source: https://github.com/Bears-R-Us/arkouda

Who wrote it?

- Mike Merrill, Bill Reus, *et al.*, US DoD

Why Chapel?

- close to Pythonic
  - enabled writing Arkouda rapidly
  - doesn't repel Python users who look under the hood
- achieved necessary performance and scalability
- ability to develop on laptop, deploy on supercomputer



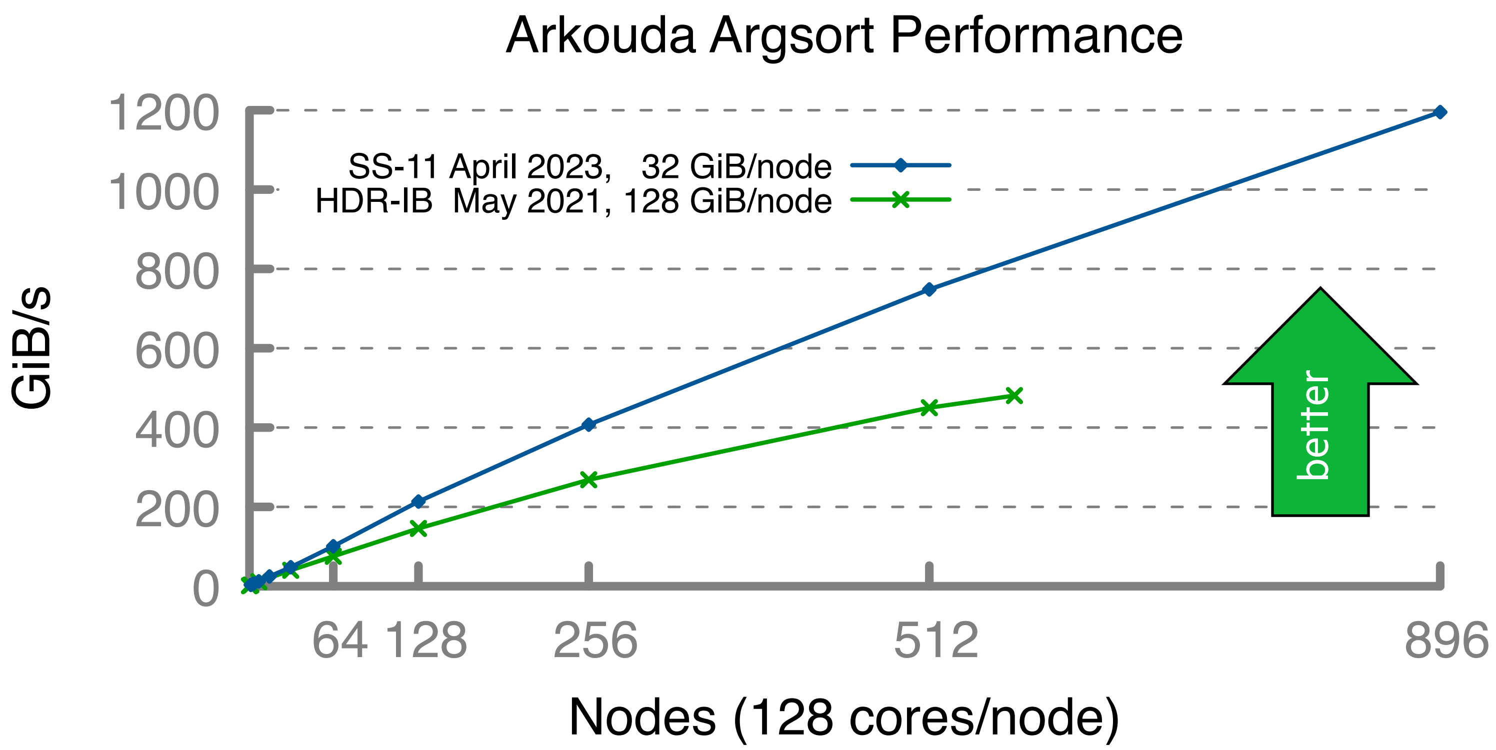
SCALABILITY OF ARKOUDA'S ARGSORT ROUTINE

HPE Cray EX (spring 2023)

- 114,688 cores of AMD Rome
- Slingshot-11 network (200 Gb/s)
- 28 TiB of 8-byte values
- 1200 GiB/s
  - 24 seconds elapsed time

HPE Apollo (summer 2021)

- 73,728 cores of AMD Rome
- HDR Infiniband network (100 Gb/s)
- 72 TiB of 8-byte values
- 480 GiB/s
  - 2.5 minutes elapsed time



A notable performance achievement in ~100 lines of Chapel

WRAP-UP

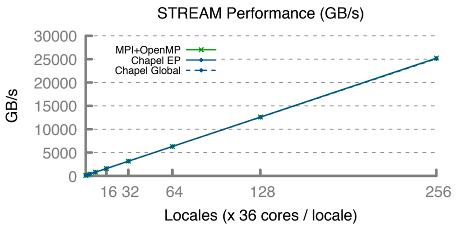
THE CHAPEL TEAM AT HPE

SUMMARY

Chapel is unique among programming languages

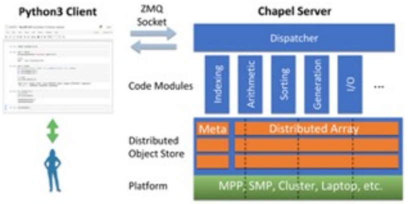
- built-in features for scalable parallel computing make it HPC-ready
- supports clean, concise code relative to conventional approaches
- ports and scales from laptops to supercomputers
- targets GPUs in a vendor-neutral manner

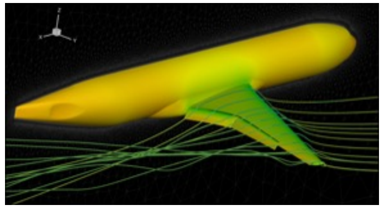
```
use BlockDist;

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;
```



Chapel is being used for productive parallel computing at scale

- users are reaping its benefits in practical, cutting-edge applications
- applicable to domains as diverse as physical simulations and data science

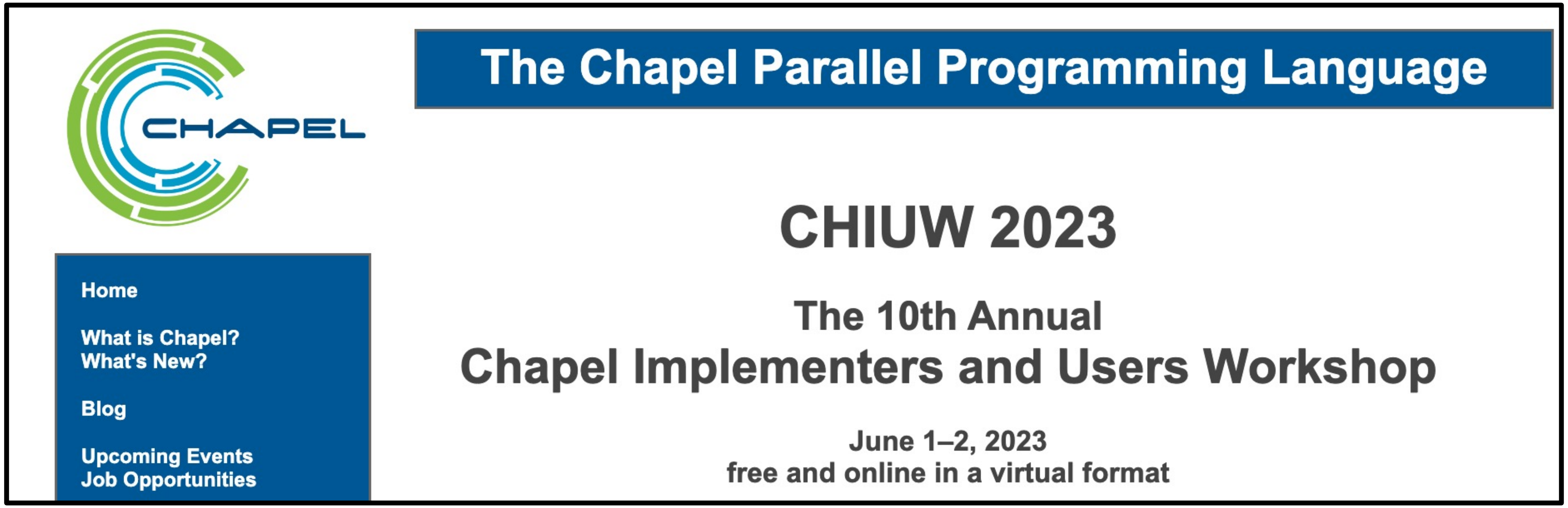




If you or your users are interested in taking Chapel for a spin, let us know!

- we're happy to work with users and user groups to help ease the learning curve

COMING UP: CHIUW 2023



- **What?** The Chapel community's annual workshop
- **When?** June 1–2
  - one day of interactive programming
  - one day of presentations
- **Where?** Online
- **Cost?** Free

Details at: https://chapel-lang.org/CHIUW2023.html

CHAPEL RESOURCES

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

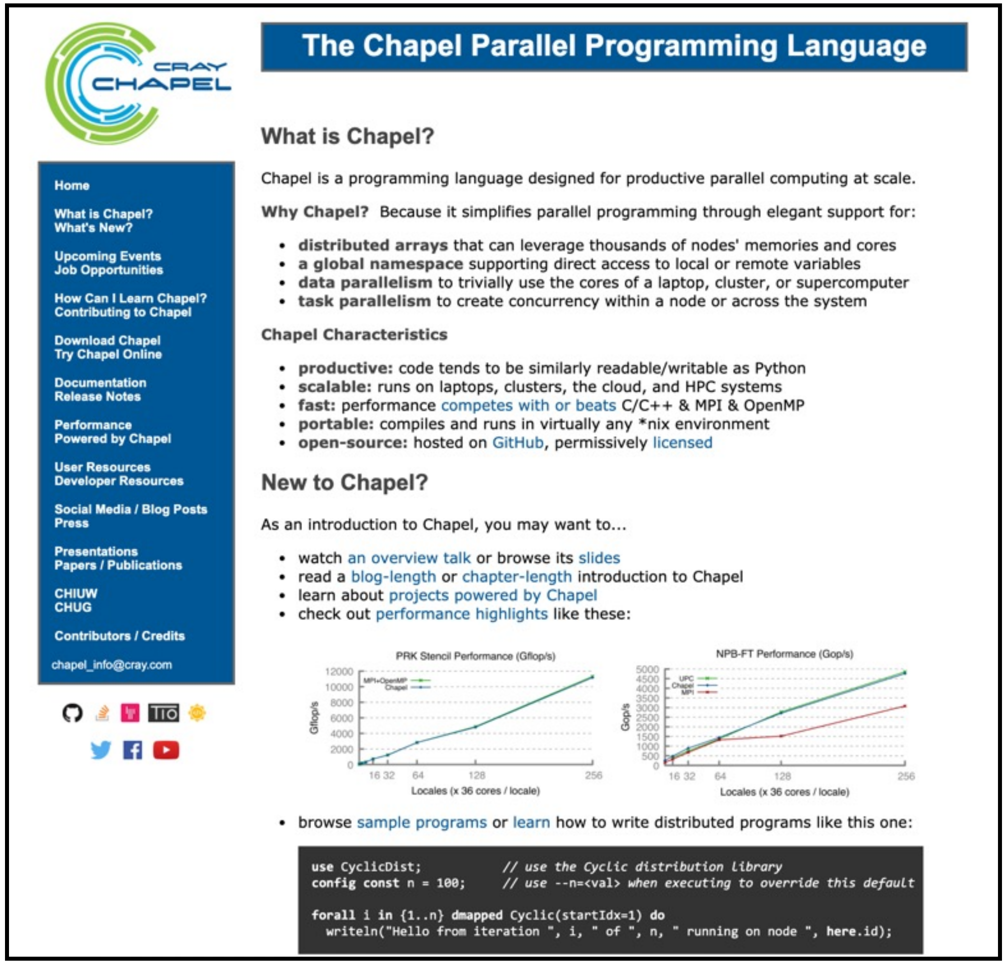
- (points to all other resources)

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

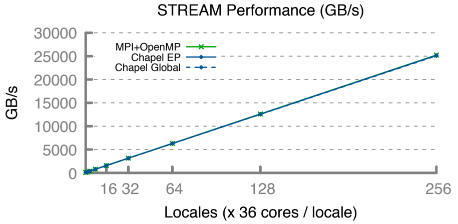


SUMMARY

Chapel is unique among programming languages

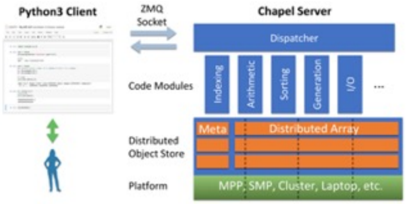
- built-in features for scalable parallel computing make it HPC-ready
- supports clean, concise code relative to conventional approaches
- ports and scales from laptops to supercomputers
- targets GPUs in a vendor-neutral manner

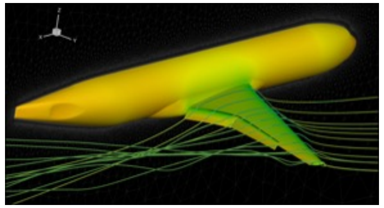
```
use BlockDist;

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;
```



Chapel is being used for productive parallel computing at scale

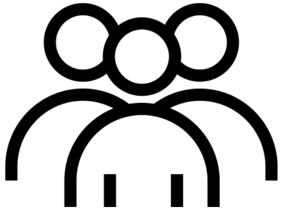
- users are reaping its benefits in practical, cutting-edge applications
- applicable to domains as diverse as physical simulations and data science





If you or your users are interested in taking Chapel for a spin, let us know!

- we're happy to work with users and user groups to help ease the learning curve



THANK YOU

https://chapel-lang.org
@ChapelLanguage