ASYNCHRONOUS TASK-BASED AGGREGATED COMMUNICATION IN CHAPEL

Elliot Ronaghan

AMTE 2022
August 23, 2022

WHAT IS CHAPEL?

Chapel: A modern parallel programming language

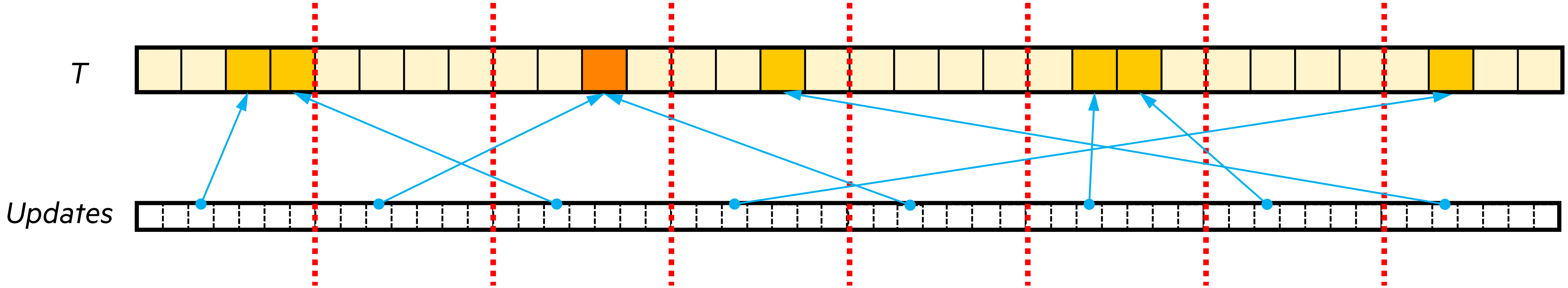
- portable & scalable
- open-source & collaborative

Goals:

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

HPCC RANDOM ACCESS (RA)

Data Structure: distributed table



Computation: in parallel, update random table elements with random values

Declarations: distributed table and index space of updates in Chapel:

```
var T: [newBlockDom(0..<tableSize)] atomic int;
const Updates = newBlockDom(0..<numUpdates);
```

HPCC RA: MPI KERNEL

```
/* 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);
```

HPCC RA: CHAPEL VS. MPI KERNEL COMMENT

```
/* 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;
 *   }
 */
```

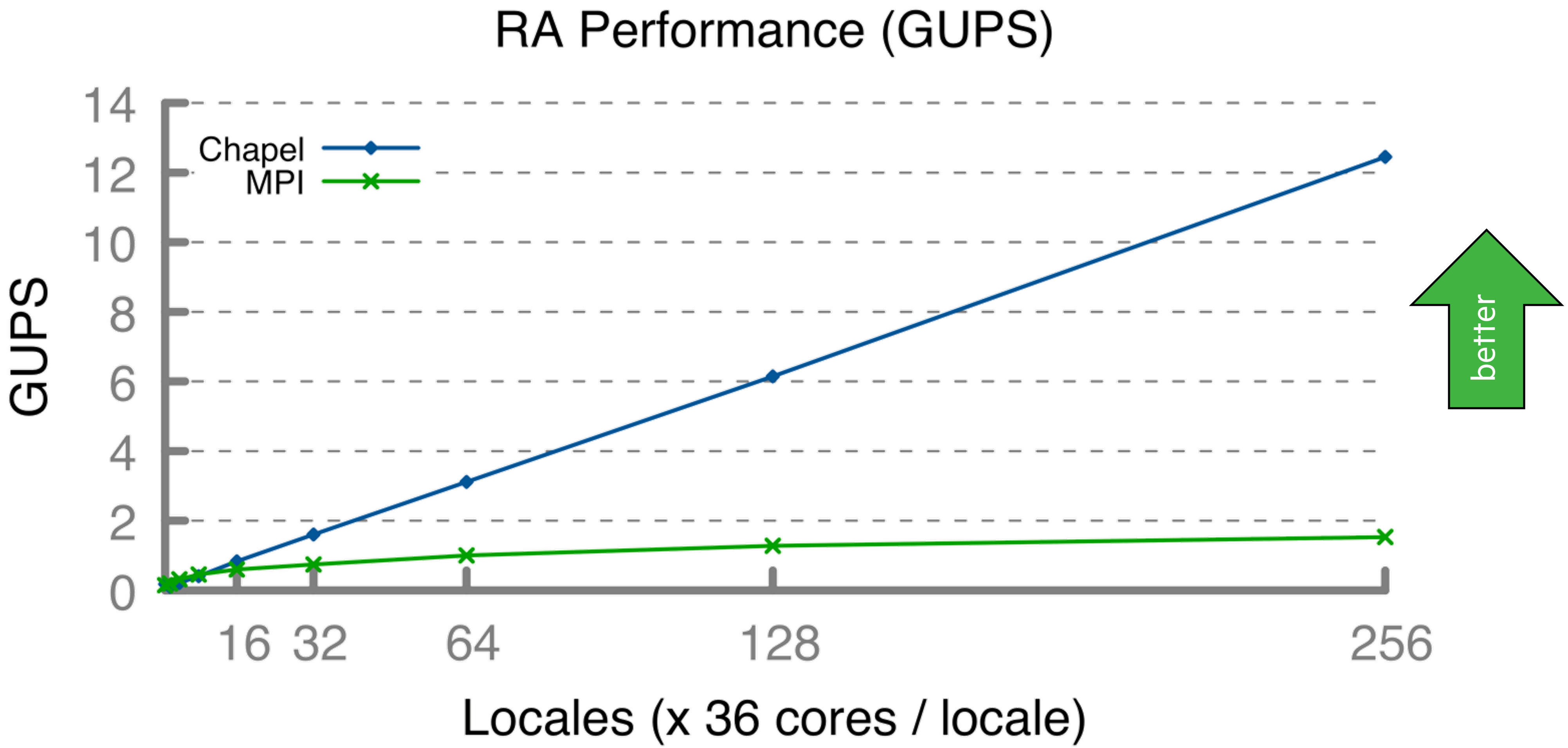
Chapel Kernel

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

MPI Comment

```
/* 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;
 *     }
 */
```

HPCC RA: CHAPEL VS. C+MPI



HPCC RA IN CHAPEL: NAÏVE IMPLEMENTATION

```
/* 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;
 *   }
 */
```

Chapel Kernel

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

Gets lowered roughly to…

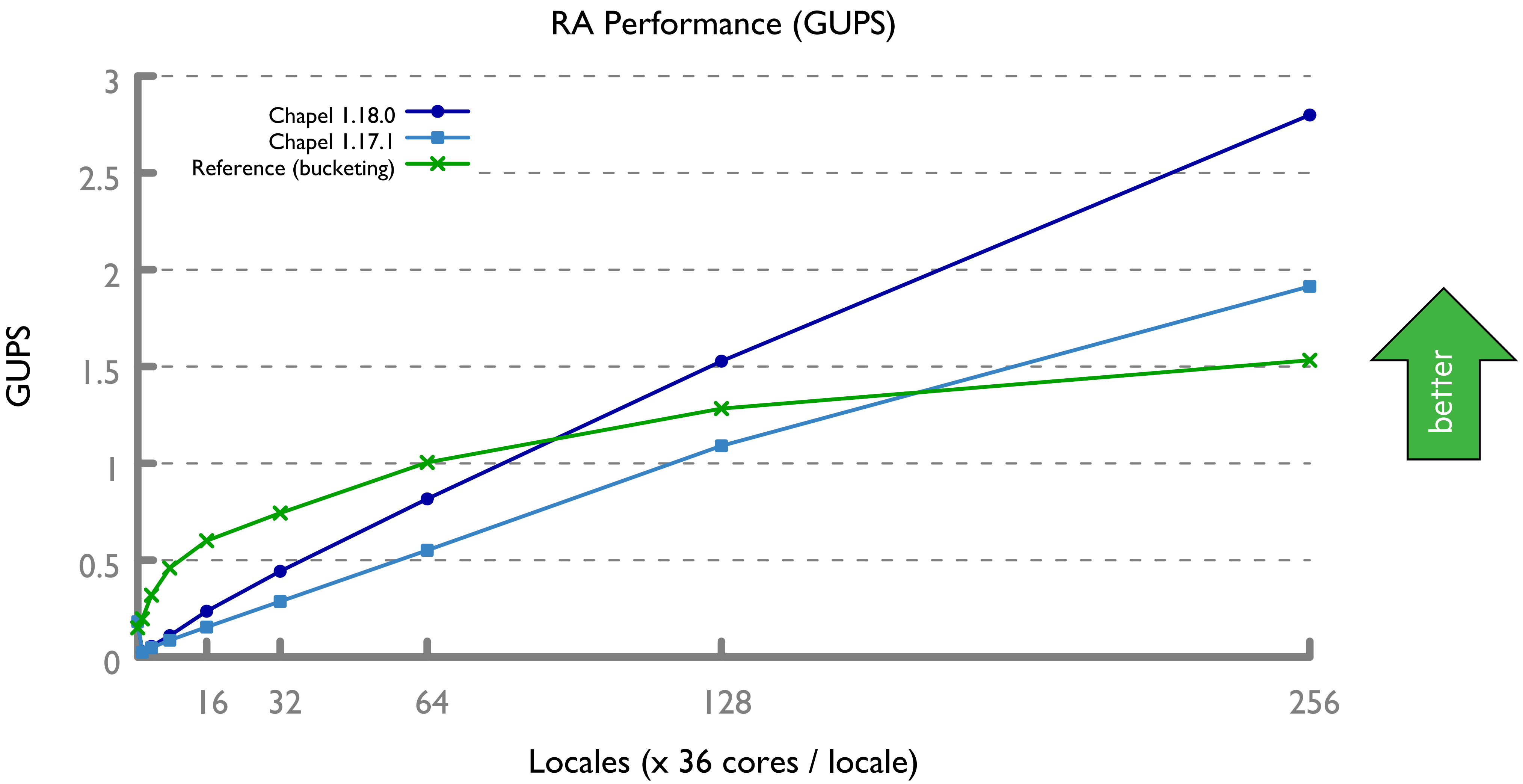
```
coforall loc in Updates.targetLocales do
  on loc do
    coforall tid in 1..here.numPUs() do
      for idx in myInds(loc, tid, …) do
        T[idx & indexMask].xor(idx);
}
```

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 updates

HPCC RA: NAÏVE CHAPEL VS. C+MPI (SEPTEMBER 2018)



UNORDERED OPERATION OPTIMIZATION

```
/* 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;
 *   }
 */
```

Chapel Kernel

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

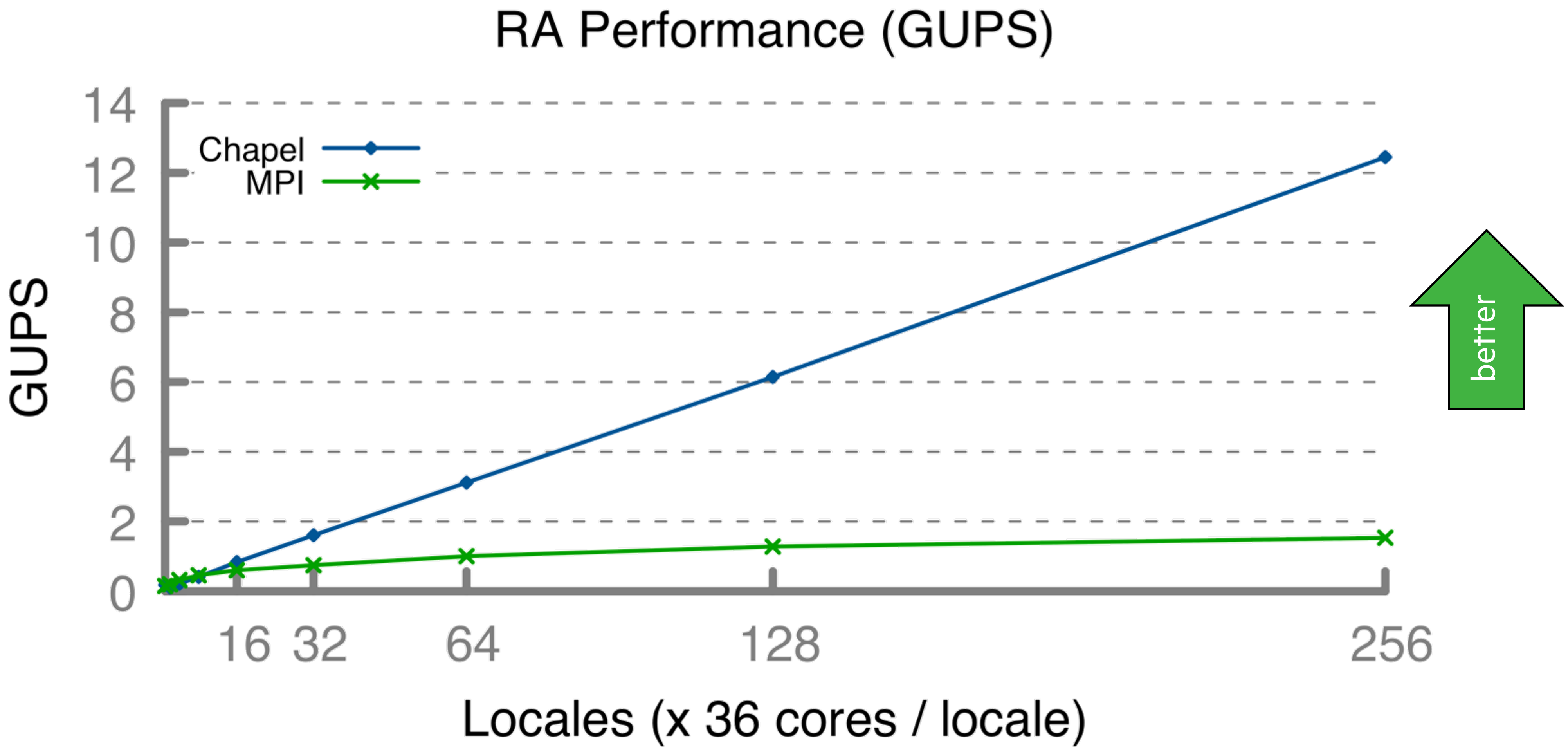
```
coforall loc in Updates.targetLocales do
  on loc do
    coforall tid in 1..here.numPUs() do
      for idx in myInds(loc, tid, …) do
        T[idx & indexMask].xor(idx);
}
```

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
  T[idx & indexMask].unorderedXor(idx);
unorderedFence();
```

HPCC RA: CHAPEL VS. C+MPI (TODAY)



HPCC RA: CHAPEL VS. C+MPI

```
/* 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, loca
          MPI_ANY_SOURCE, MPI_A
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;
          inmsg = LocalRecvBuffer[b
          LocalOffset = (inmsg & (t
                        tparams.Glo
          HPCC_Table[LocalOffset] ^
        }
      } else if (status.MPI_TAG == FINISHED_TAG) {
        NumberReceiving--;
      } else
        MPI_Abort( MPI_COMM_WORLD,
      MPI_Irecv(&LocalRecvBuffer, l
                MPI_ANY_SOURCE, MPI
    }
  } 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++;

            UPDATE_TAG, MPI_COMM_WORLD, &outreq);
    pendingUpdates -= peUpdates;
  }
}

if (have_done) {
  if (status.MPI_TAG == UPDATE_TAG) {

      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;
                                    ndBuffer, localBufferSize,
                              , tparams.dtype64, (int)pe,
                                &outreq);

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 */
                                 64, proc_count, FINISHED_TAG,
                                 s.finish_req + proc_count);

  if (status.MPI_TAG == UPDATE_TAG) {
    MPI_Get_count(&status, tparams.dtype64, &recvUpdates);
                                 +) {
                                 ferBase+j];
                                 ams.TableSize - 1)) -
                                 lStartMyProc;
                                 nmsg;
    }
  } 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);
```

Chapel Kernel

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

Now, think about what it would take for a compiler to optimize the C+MPI code…

…or for a user to target the Cray XC's network atomics manually (and portably?)

OUTLINE

I. Chapel by Example: HPCC RA

II. Chapel Motivation & Context

III. Tasking and Locality Features

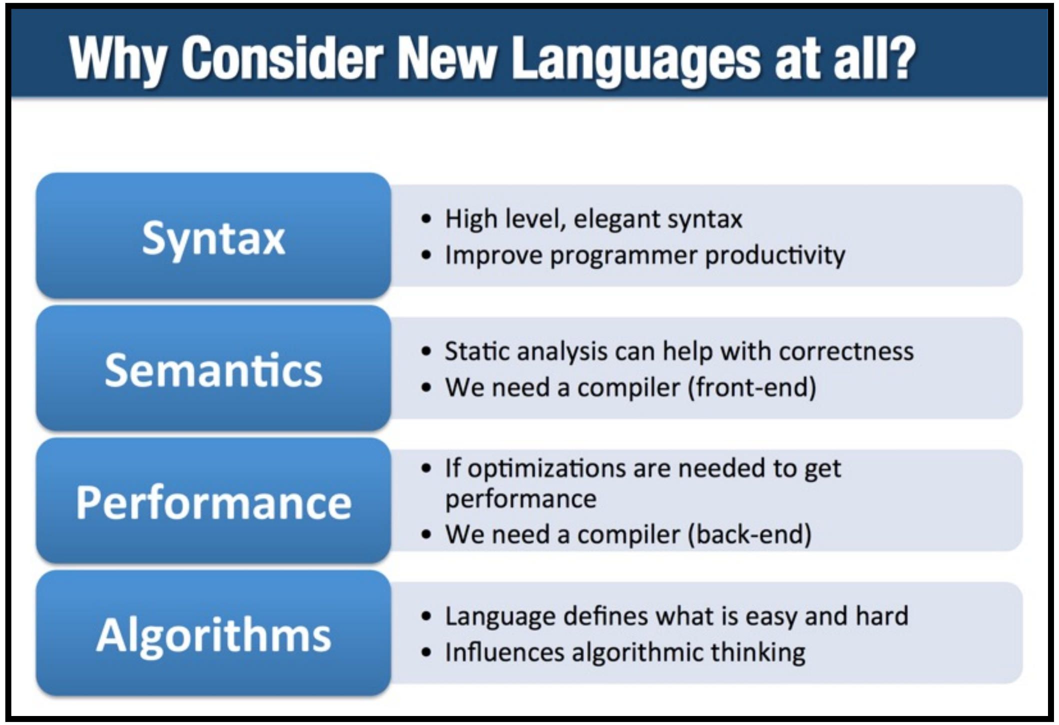
IV. Chapel Aggregators

V. Arkouda and Aggregation

VI. Wrap-up

WHY CREATE A NEW LANGUAGE?

- **Because parallel programmers deserve better**
  - the state of the art for HPC is a mish-mash of libraries, pragmas, and extensions
  - parallelism and locality are concerns that deserve first-class language features



[Image Source:
Kathy Yelick's (UC Berkeley, LBNL)
CHIUW 2018 keynote:
Why Languages Matter More Than Ever,
used with permission]

- **And because existing languages don't fit our desires…**

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++

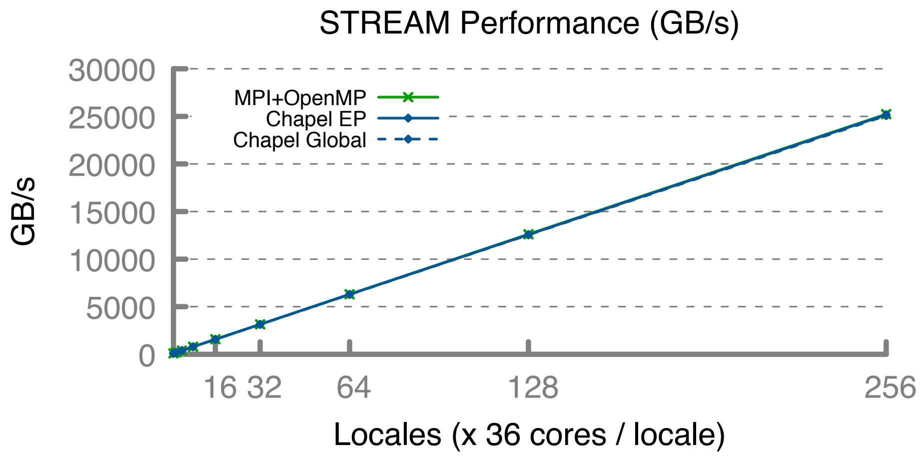
…**fun** as [your favorite programming language]

CHAPEL BENCHMARKS TEND TO BE CONCISE, CLEAR, AND COMPETITIVE

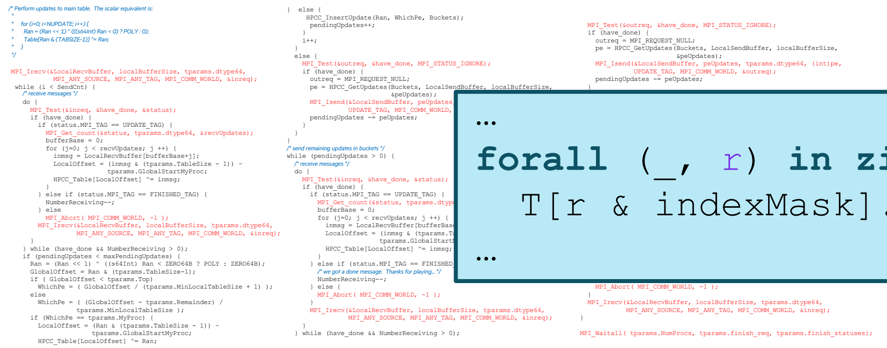
STREAM TRIAD: C + MPI + OPENMP

```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
      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;
}
```

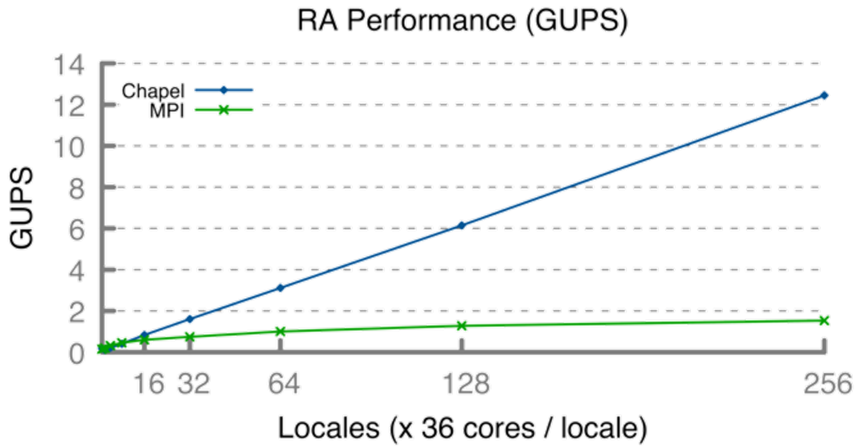
```chapel
use BlockDist;

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;
```



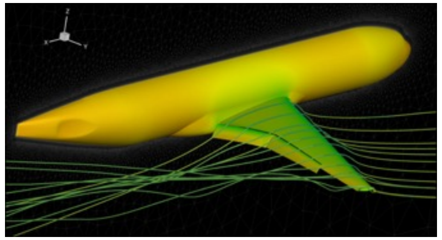
HPCC RA: MPI KERNEL



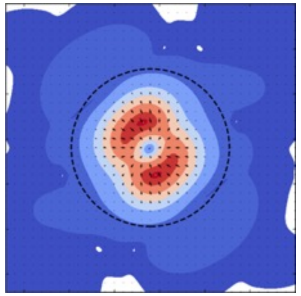
```chapel
…
forall (_, r) in zip(Updates, RAStream()) do
  T[r & indexMask].xor(r);
…
```



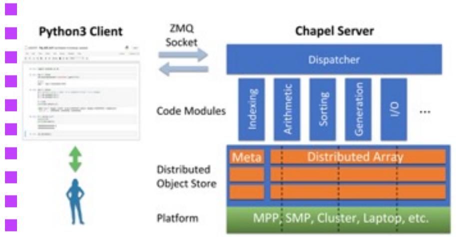
NOTABLE CURRENT APPLICATIONS OF CHAPEL



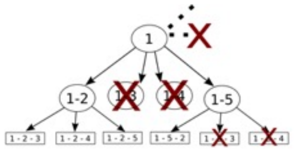
CHAMPS: 3D Unstructured CFD
Éric Laurendeau, Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal
~120k lines of Chapel



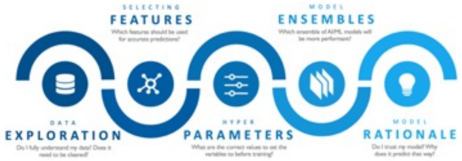
ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan, J. Luna Zagorac, Richard Easther, *et al.*
Yale University / University of Auckland



Arkouda: NumPy at Massive Scale
Mike Merrill, Bill Reus, et al.
US DOD
~22k 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?

CHAPEL'S MULTIRESOLUTION PHILOSOPHY

1. Users should be able to program at high levels of abstraction and get good performance

```
Dst = Src[Inds];      // whole-array index gather
```

2. Yet, when more control / better performance is needed, they can drop to lower levels…

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

…and even lower levels, as necessary…

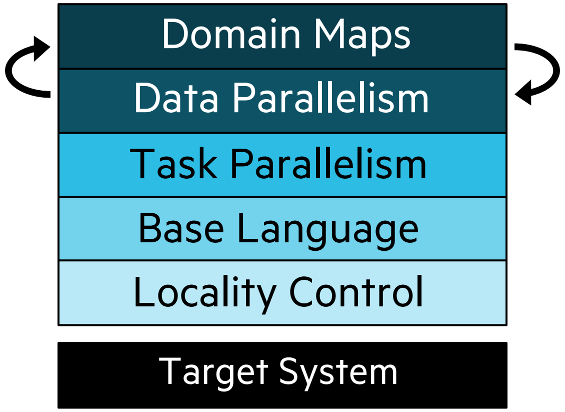
```
coforall loc in Dst.targetLocales do     // explicit SPMD-style index gather
  on loc do
    forall i in Dst.localSubdomain do
      Dst.localAccess[i] = Src[Inds.localAccess[i]];
```

…where "calling out to C / CUDA / MPI / etc." is effectively the lowest level

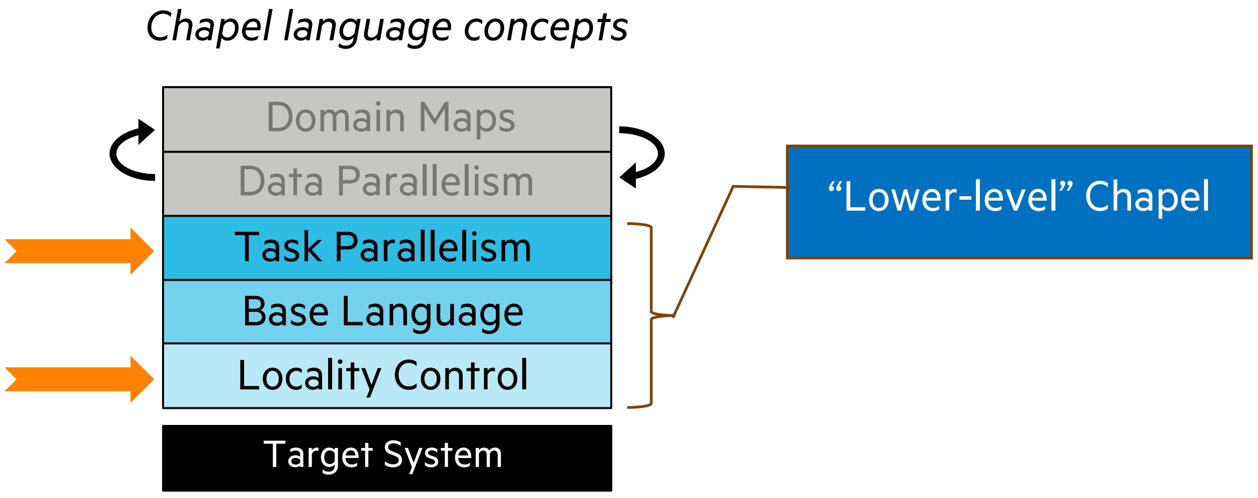
3. Chapel builds its higher-level abstractions in terms of the lower-level ones to guarantee compatibility

CHAPEL'S MULTIRESOLUTION FEATURE STACK

Chapel language concepts



CHAPEL'S "LOWER-LEVEL" FEATURES



TASKING AND LOCALITY FEATURES

CHAPEL, ASYNCHRONY, AND THIS TALK

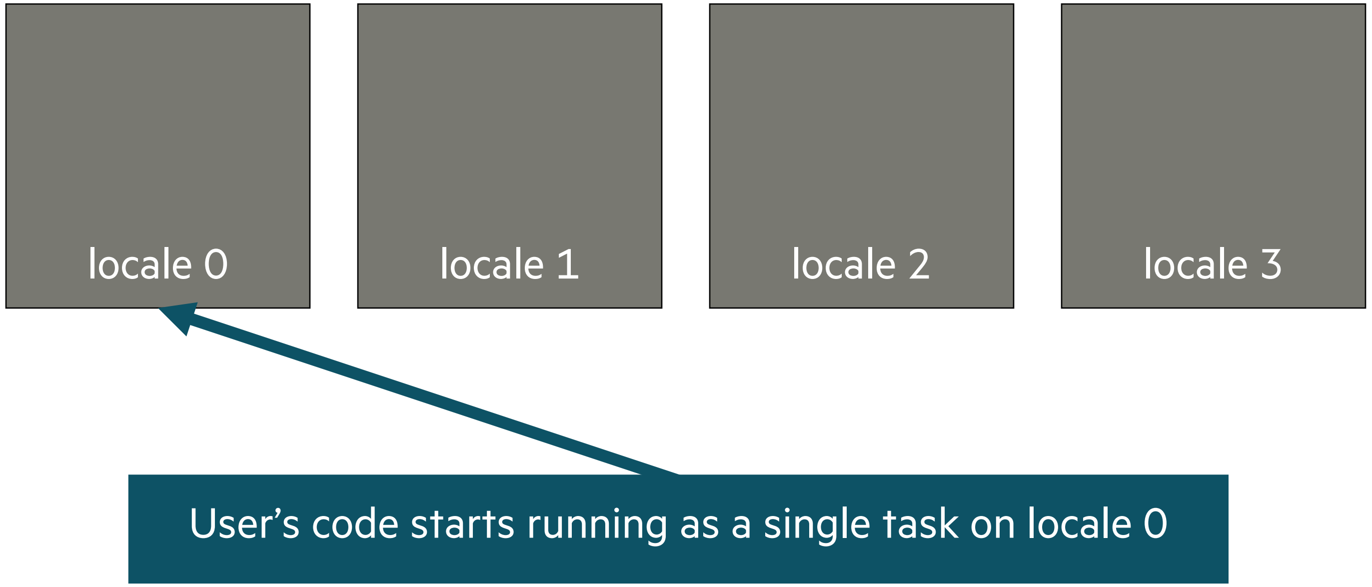
- Chapel tasks are asynchronous in sense that they are:
  - launched dynamically
  - managed by the runtime
  - no pre-determined start or end conditions
- Asynchrony gives expressiveness benefits in that any parallel pattern can be expressed
- In practice, many patterns use groups of tasks doing similar things
  - such as the coforall examples in this talk
- Asynchrony improves performance by spreading network load and avoids synchronization bottlenecks
  - aggregators in this talk benefit highly from this

CHAPEL TERMINOLOGY: LOCALES

- Locales 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:



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 of %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
…
```

DIFFERENCES BETWEEN CHAPEL AND TRADITIONAL PGAS / SHMEM

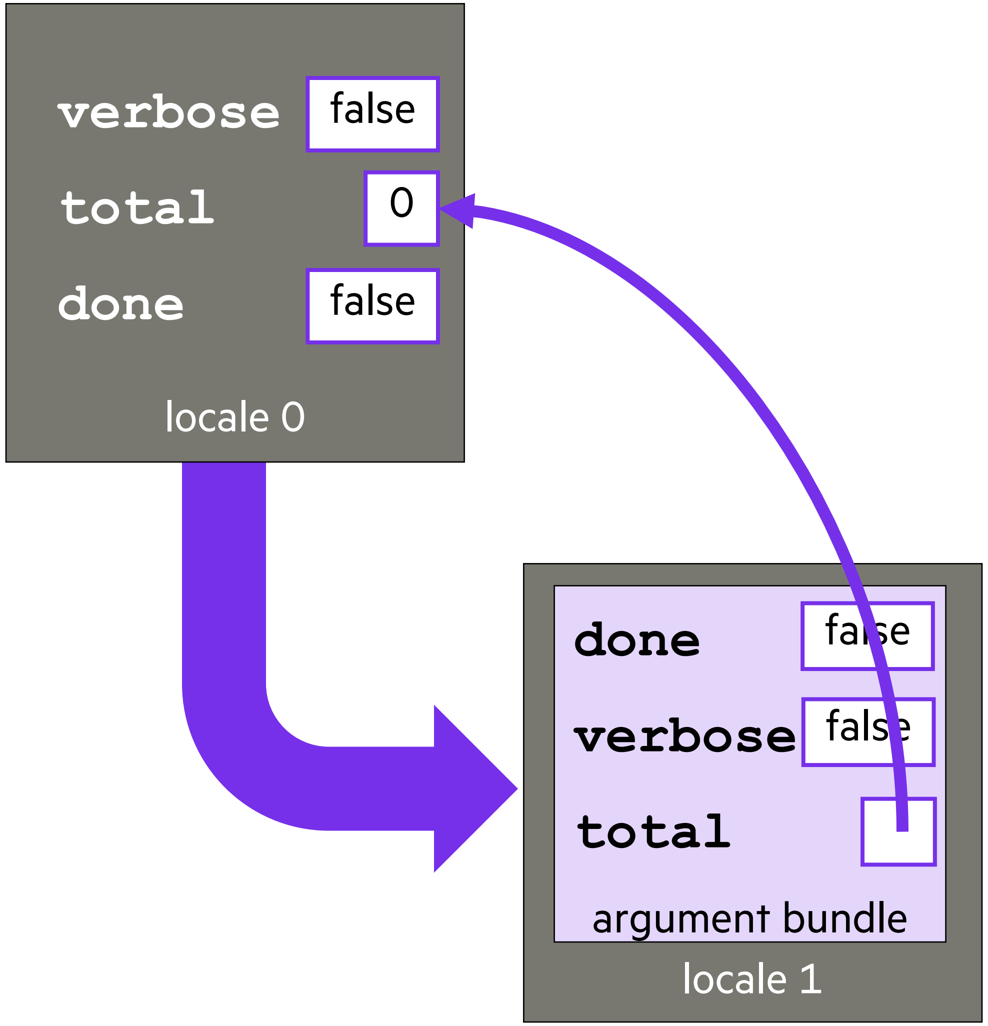
1. Chapel supports a post-SPMD execution model
   - **traditional PGAS:** all PEs/ranks/threads start by executing 'main'
   - **Chapel:** a single task executes 'main' on locale 0 and additional parallelism* is introduced from there

(* = local or distributed)

CHAPEL'S PARTITIONED GLOBAL NAMESPACE

onClause.chpl

```
const verbose = false;
var total = 0,
    done = false;

…

on Locales[1] {
  if !done {
    if verbose then
      writef("Adding locale 1's contribution");
    total += computeMyContribution();
  }
}
```



DIFFERENCES BETWEEN CHAPEL AND TRADITIONAL PGAS / SHMEM

1. Chapel supports a post-SPMD execution model
   - **traditional PGAS:** all ranks/threads/PEs start by executing 'main'
   - **Chapel:** a single task executes 'main' on locale 0 and additional parallelism* is introduced from there

   (* = local or distributed)

2. Chapel's partitioned global address space is also post-SPMD
   - **traditional PGAS:** "I have a variable named 'x', so you must too, and therefore I can refer to yours"
   - **Chapel:** "I see variable 'x' in my lexical scope, so I can refer to it, whether it's local or remote"

One outcome of these differences is that Chapel feels much more like traditional programming

BULK COMMUNICATION IN CHAPEL: A TOOL FOR MANUAL AGGREGATION

bulkComm.chpl

```
var Buff: [0..<buffSize] real;

on Locales[1] {
  var LocBuff = Buff;

  processData(LocBuff);
  Buff = LocBuff;
}
```

- allocate an array on locale 0
- move computation to locale 1
- bulk ‘get’ from remote array
- bulk ‘put’ to remote array

CHAPEL AGGREGATORS

BALE INTRODUCTION

- Bale is a collection of mini applications and aggregation libraries
  - Chapel has several ports of Bale applications, including index gather
  - We use Bale to evaluate the productivity of our aggregators and to compare performance to SHMEM

- From the description in https://github.com/jdevinney/bale, Bale is a:

  "Vehicle for discussion for parallel programming productivity. The bale effort attempts to:
  - *demonstrate some challenges of implementing interesting (i.e. irregular) scalable distributed parallel applications.*
  - *demonstrate an approach (aggregation) to achieve high performance for the internode communication in such applications*
  - *explore concepts that make it easier to write, maintain, and get top performance from such applications*

  We use bale to evolve our thinking on parallel programming in the effort to make parallel programming easier, more productive, and more fun. Yes, we think making it fun is a worthy goal!"

BALE INDEX GATHER IN CHAPEL

```
use BlockDist, Random;

const numTasks = numLocales * here.maxTaskPar;
config const N = 1000000,    // number of updates per task
             M = 10000;      // number of entries in the table per task

const D = newBlockDom(0..<M*numTasks);
var Src: [D] int = D;
const UpdatesDom = newBlockDom(0..<N*numTasks);
var Dst, Inds: [UpdatesDom] int;

fillRandom(Inds, min=0, max=Src.size);

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

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?

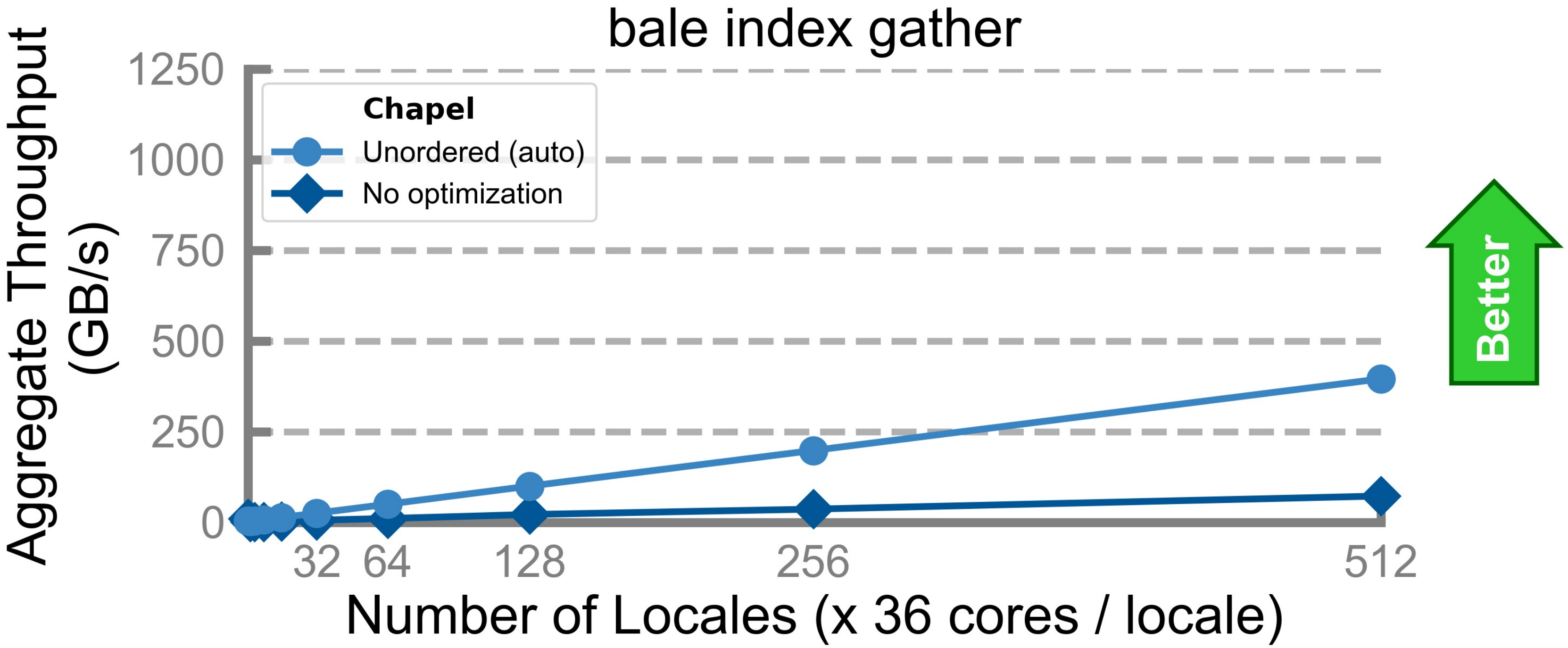
So, our compiler rewrites the inner loop to perform the ops asynchronously

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

- Implemented by Michael Ferguson, 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

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

```
use CopyAggregation;

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

‘use’ the module providing the aggregators

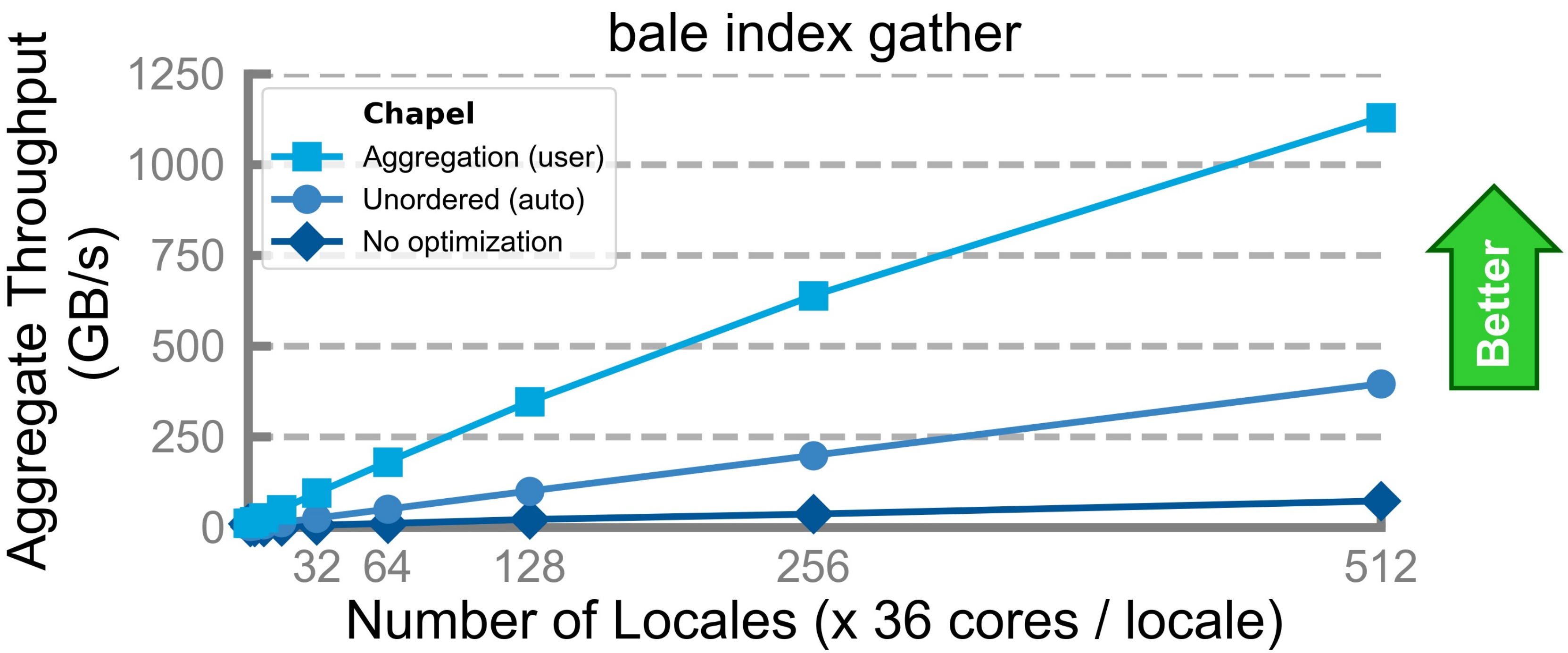
To use it, we simply replace the assignment with ‘agg.copy’

Give 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]);
```



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]);
```

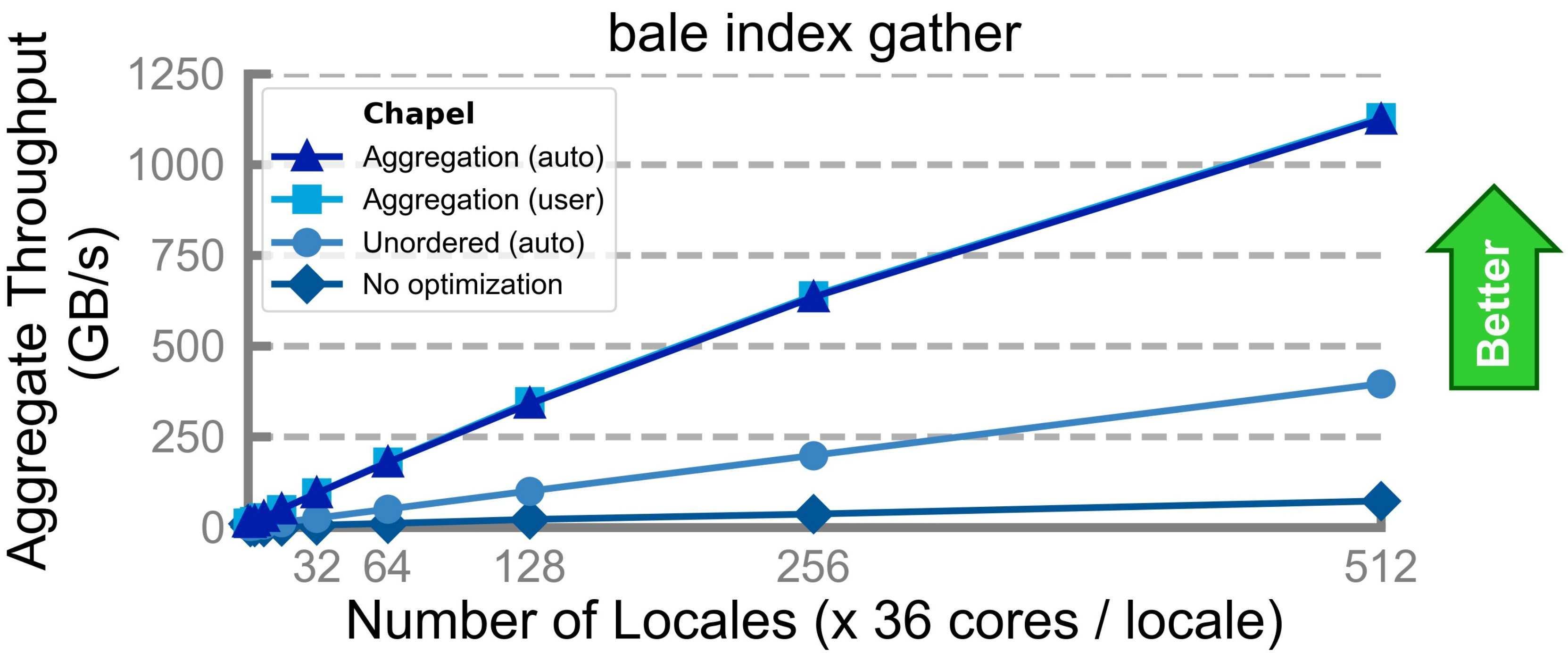
A: In many cases, yes

- 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

Elegant SHMEM version

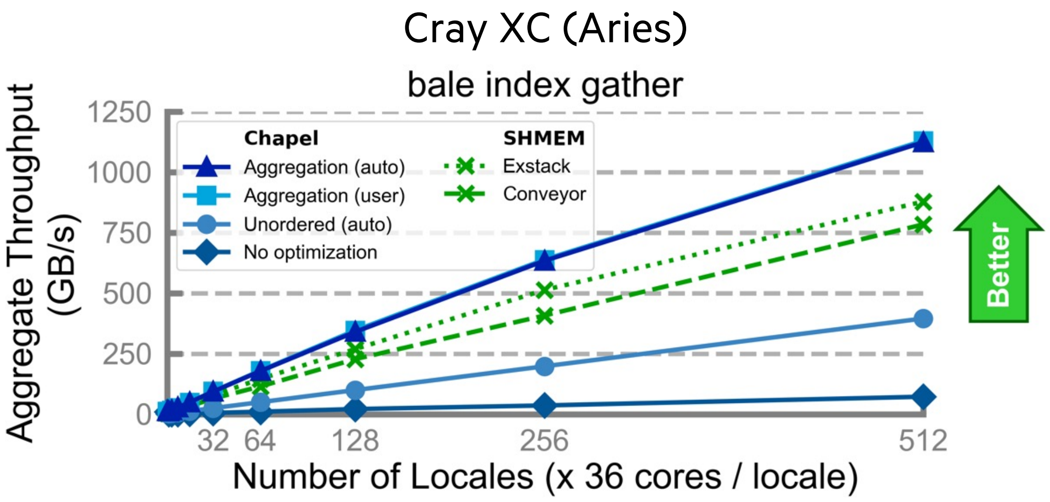
```
for (i = 0; i < N; i++)
  shmem_get(&target[i], &table[index[i]], sizeof(long), index[i] % NPES);
```

Exstack version

```
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;
}
```



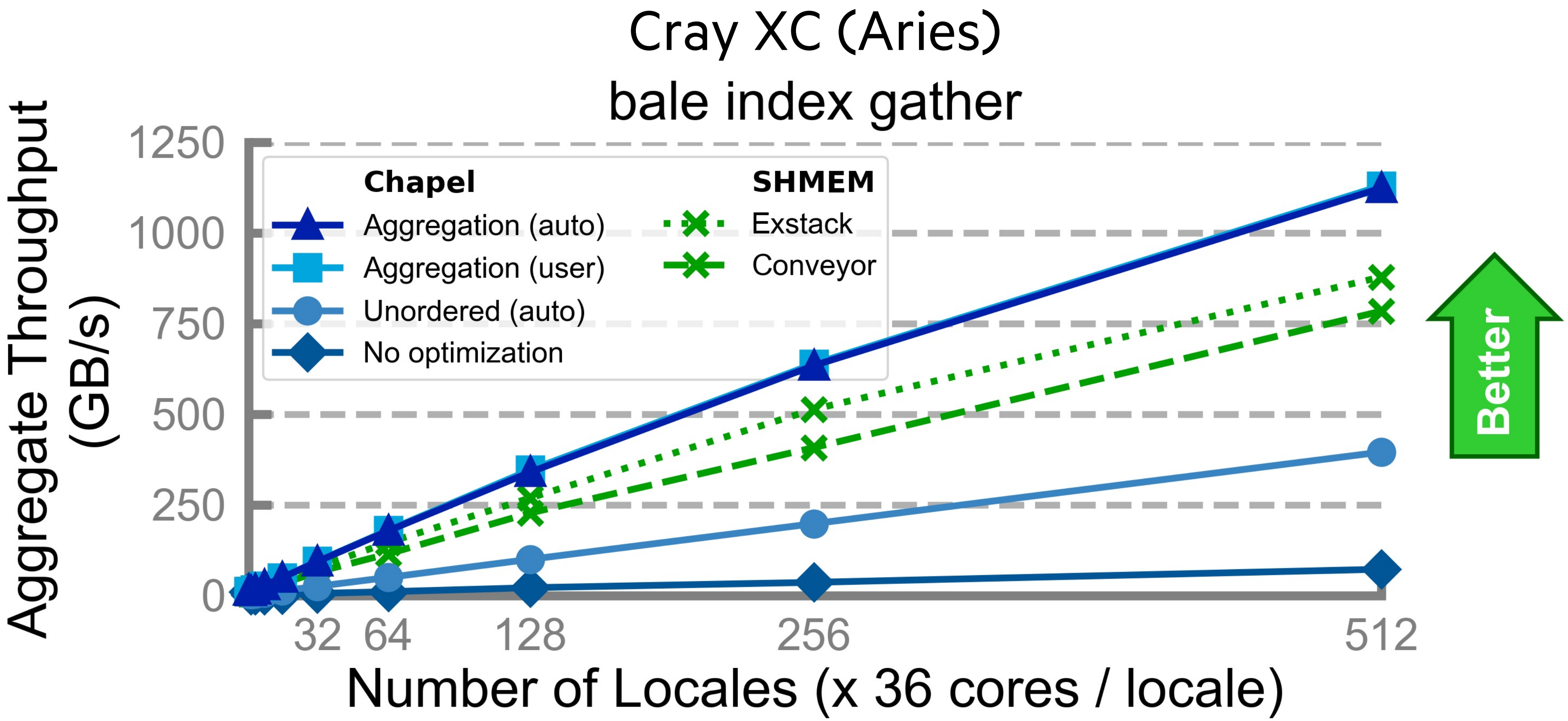
Elegant Chapel version (compiler-optimized w/ '--auto-aggregation')

```
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]);
```

BALE INDEX GATHER: CHAPEL VS. EXSTACK VS. CONVEYORS



IMPLEMENTING CHAPEL'S AGGREGATORS

- Chapel's aggregators are implemented as Chapel source code
  - no language or compiler changes were required
  - initial implementation only relied on high-level features
    - current optimized version calls into lower-level put/get routines

  - Relies upon:
    - standard language features:
      - OOP: records, initializers, de-initializers
      - arrays
      - access to C-level pointers and dereferences
    - Chapel features that you've seen:
      - global namespace
      - task-local variables

  - ~100 lines of reasonably straightforward code to implement SrcAggregator
    - (~420 lines for entire 'CopyAggregation' module)

INITIAL SRC AGGREGATOR IMPLEMENTATION: EXCERPTS

```
record SrcAggregator {
  type elemType;
  var dstAddrs, srcAddrs: [LocaleSpace] [0..<bufferSize] addr;
  var bufferIdxs: [LocaleSpace] int;
  …
  proc flushBuffer(loc: int, ref bufferIdx) {
    var srcVals: [0..<bufferIdx] elemType;

    on Locales[loc] {
      const locSrcAddrs = srcAddrs[loc][0..<bufferIdx];
      var locSrcVals: [0..<bufferIdx] elemType;
      …   // fill the locSrcVals array
      srcVals = locSrcVals;
    }
    …   // assign the srcVals to the dstAddrs
  }
}
```

Arrays for buffering per-locale src/dest addresses

time to flush a buffer?

allocate a landing spot for the remote src values

move to the remote node to buffer src vals and copy them back

Bulk array copy to ‘get’ the src addresses

Declare a buffer to store the local src vals and fill it

Bulk array copy to ‘put’ src values back to original locale

Store src vals to dest addresses

INITIAL SRC AGGREGATOR IMPLEMENTATION: EXCERPTS

```
record SrcAggregator {
  type elemType;
  var dstAddrs, srcAddrs: [LocaleSpace] [0..<bufferSize] addr;
  var bufferIdxs: [LocaleSpace] int;
  …
  proc flushBuffer(loc: int, ref bufferIdx) {
    var srcVals: [0..<bufferIdx] elemType;

    on Locales[loc] {
      const locSrcAddrs = srcAddrs[loc][0..<bufferIdx];
      var locSrcVals: [0..<bufferIdx] elemType;
      …   // fill the locSrcVals array
      srcVals = locSrcVals;
    }
    …   // assign the srcVals to the dstAddrs
  }
}
```

Bulk array copy to 'get' the src addresses

Bulk array copy to 'put' src values back to original locale

CHAPEL AGGREGATORS: ATTRACTIVE PROPERTIES

- More flexible than traditional aggregators:
  - **traditional aggregators:** like barriers or collectives, tend to assume everyone is involved and quasi-lockstep
  - **Chapel aggregators:** Chapel's post-SPMD nature relaxes traditional BSP constraints
    - tasks communicate with remote locales asynchronously, once a given buffer fills up
    - any subset of tasks/locales can utilize aggregators that target any locales *without those locales being involved*

- User-level tasks make the implementation efficient
  - Chapel leverages Sandia's Qthreads

- Performance is competitive with conventional techniques

SOUNDS GREAT, WHAT'S THE CATCH?

Q: Clean code, competitive performance and scalability, no modifications to the language or compiler…

…so, what's the catch?

A: Not a 'catch' per se, but currently, Chapel's aggregators only support copy-style and atomic operations

- Ultimately, want/need to support general operations ("user-defined aggregators")
  - In principle, not so different from the existing ones
  - **Limiting factor:** These would most naturally be expressed with first-class functions (FCFs)

    …but Chapel's support for FCFs is currently a bit weak

- That said, many interesting computations can be written with copy-style aggregation…

  …like Arkouda!

ARKOUDA AND AGGREGATION

MOTIVATION FOR ARKOUDA

Motivation: Say you've got…

…HPC-scale data science problems to solve

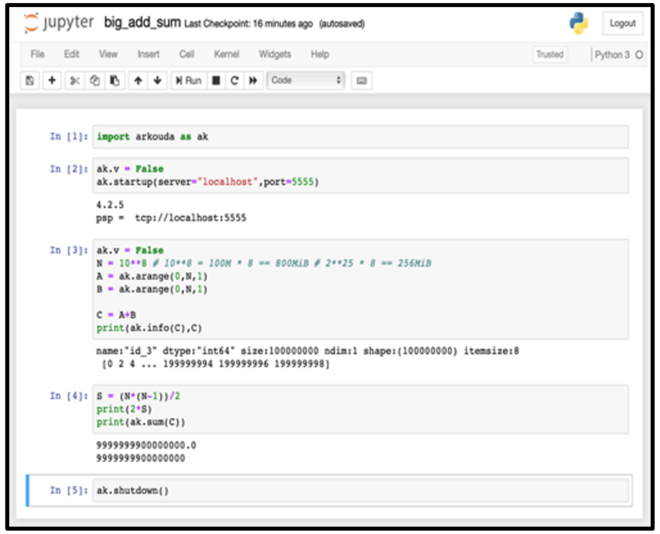
…a bunch of Python programmers

…access to HPC systems

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

ARKOUDA'S HIGH-LEVEL APPROACH

Arkouda Client
(written in Python)



Arkouda Server
(written in Chapel)

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

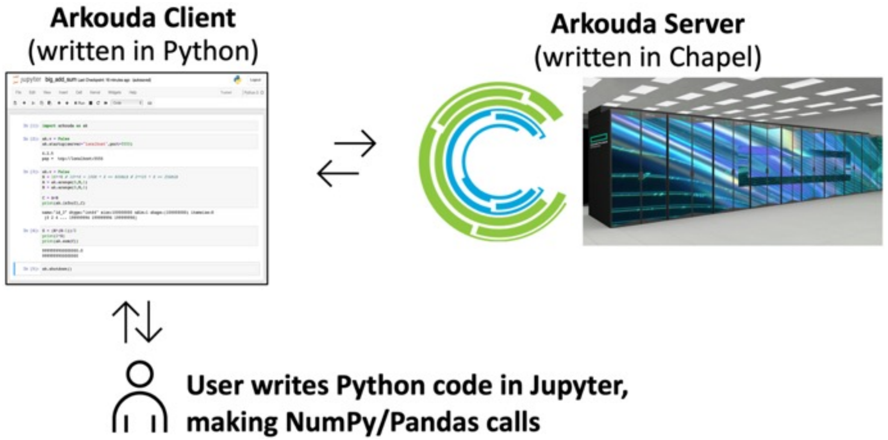
ARKOUDA SUMMARY

What is it?

- A Python library supporting a key subset of NumPy and Pandas for Data Science
  - Uses a Python-client/Chapel-server model to get scalability and performance
  - Computes massive-scale results (multi-TB-scale arrays) within the human thought loop (seconds to a few minutes)
- ~22k 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

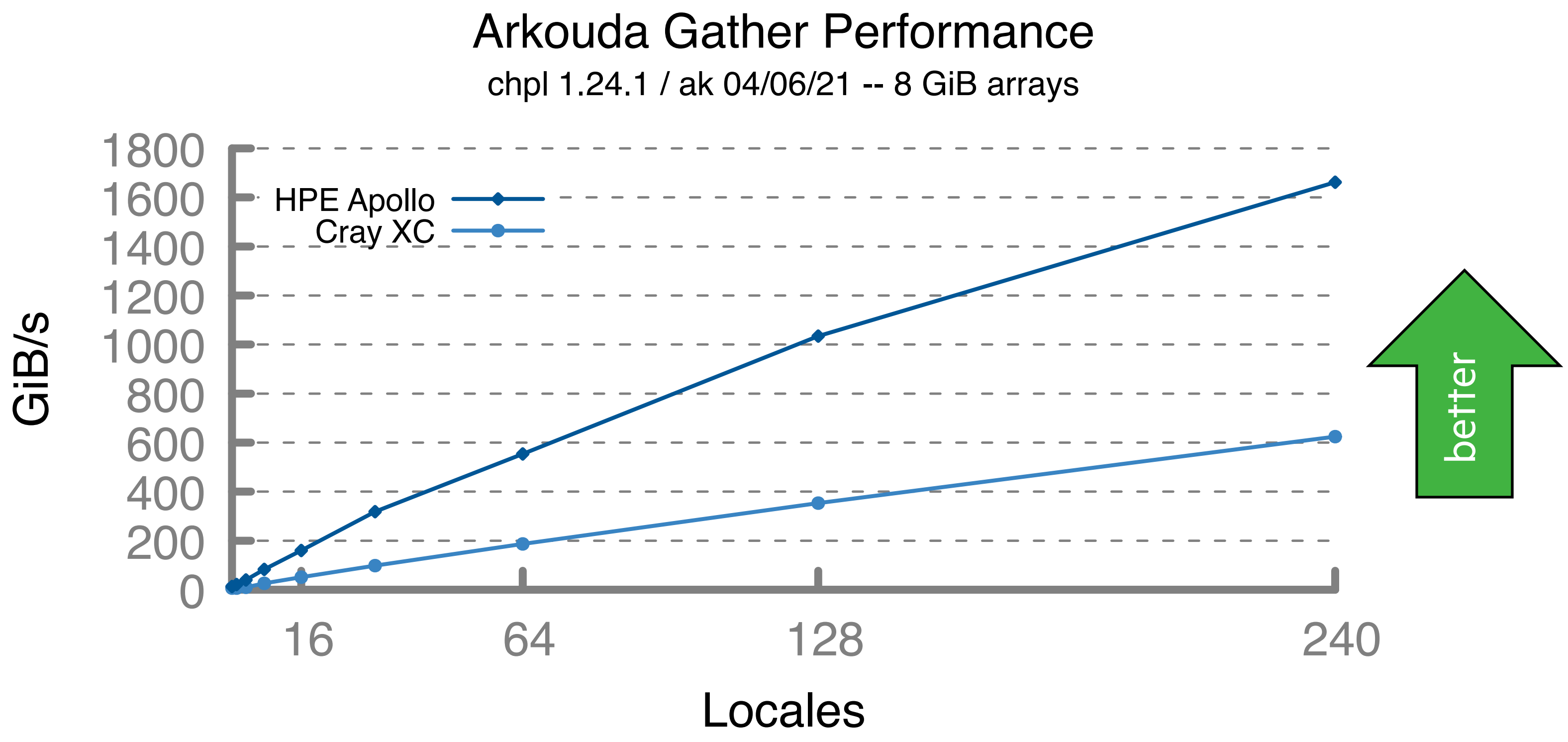


Why Chapel?

- high-level language with performance and scalability
- close to Pythonic
  - enabled writing Arkouda rapidly
  - doesn't repel Python users who look under the hood
- ports from laptop to supercomputer

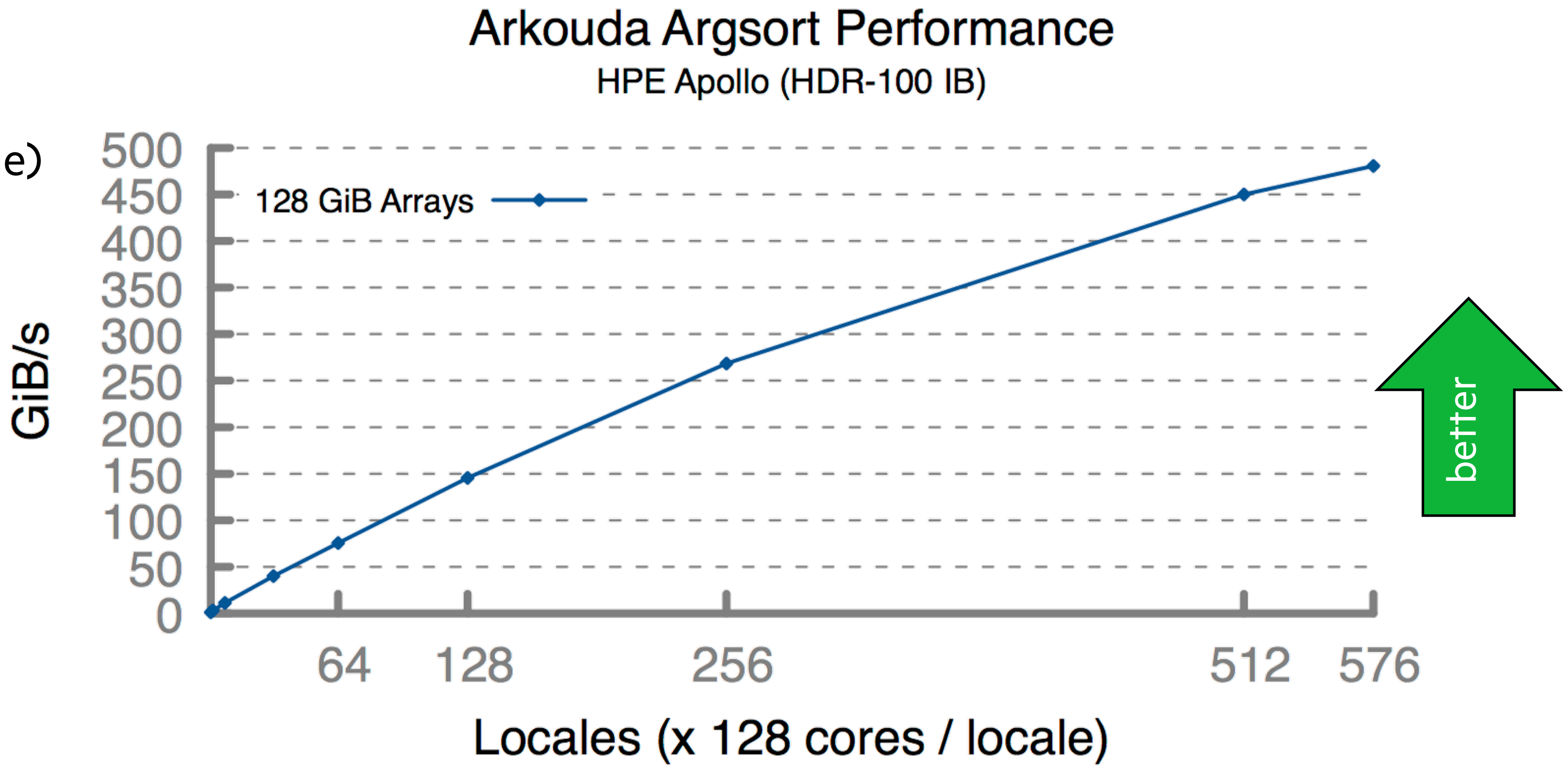
ARKOUDA GATHER

- For Arkouda's gather kernel, Chapel performance on a recent HPE Apollo system is well ahead of XC
  - These timings were taken in April 2021
  - System-level bugs hurt reference SHMEM performance, so no direct comparisons here



ARKOUDA ARGSORT AT SCALE

- Ran on a large Apollo system, summer 2021
  - 73,728 cores of AMD Rome
  - 72 TiB of 8-byte values
  - 480 GiB/s (2.5 minutes elapsed time)
  - ~100 lines of Chapel code



Close to world-record performance—quite likely a record for performance/SLOC

WRAP-UP

CHAPEL RESOURCES

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

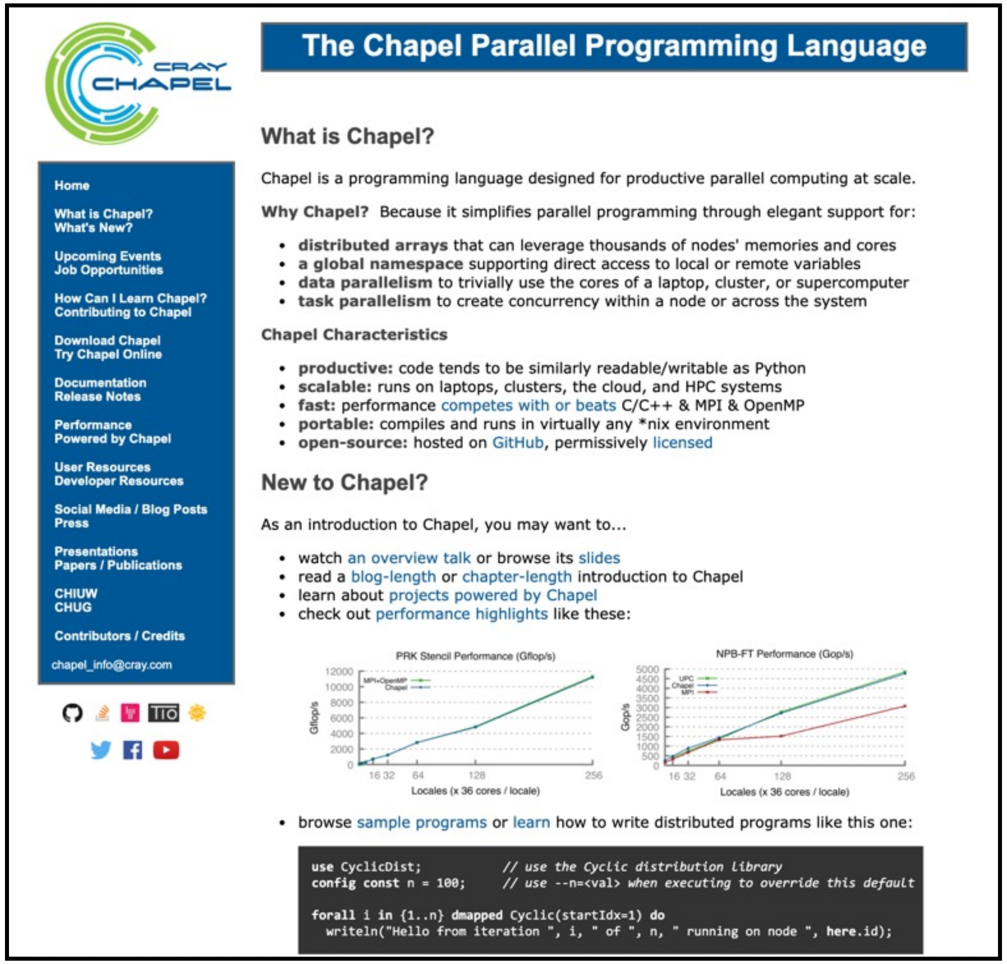
- (points to all other resources)

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



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 1.24, April 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

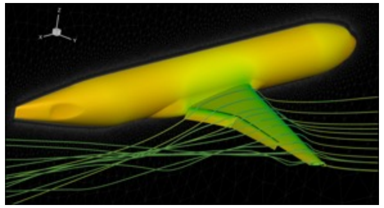
CHAMPS:

- 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")

SUMMARY

Chapel is designed for productive parallel programming at scale

- recent users have reaped these benefits in large applications



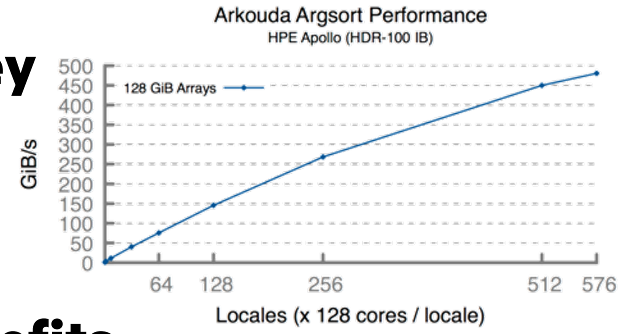
Though PGAS in nature, Chapel avoids SPMD / BSP assumptions

- parallelism is expressed in the source code starting from a single task
- lexical scoping simplifies PGAS-based communication
- the net result is a far more approachable distributed parallel language

```
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);
  }
}
```

For gather/scatter/sort in Arkouda and Bale, copy aggregators are key

- Chapel's are implemented concisely and elegantly within the language
- performance rivals that of Exstack / Conveyors



Chapel's design and language-based nature provide optimization benefits

- e.g., automatic asynchronous operations and automatic aggregation (as in Arkouda / Bale)

THANK YOU

https://chapel-lang.org
@ChapelLanguage