Chapel 101

Brad Chamberlain

CHIUW 2020, May 22, 2020

chapel_info@cray.com
chapel-lang.org
@ChapelLanguage

CRAY
a Hewlett Packard Enterprise company

CRAY 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

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 that I don’t need because I ~~was born to suffer~~ want full control to ensure 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 want, implemented in a language as attractive as recent graduates want.”

Comparing Chapel to Other Languages

Chapel aims 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]

Why Consider New Languages at all?

Syntax	• High level, elegant syntax • Improve programmer productivity
Semantics	• Static analysis can help with correctness • We need a compiler (front-end)
Performance	• If optimizations are needed to get performance • We need a compiler (back-end)
Algorithms	• Language defines what is easy and hard • Influences algorithmic thinking

[Source: Kathy Yelick, CHIUW 2018 keynote: *Why Languages Matter More Than Ever*]

Outline

✓ Context and Motivation

➢ Chapel and Productivity

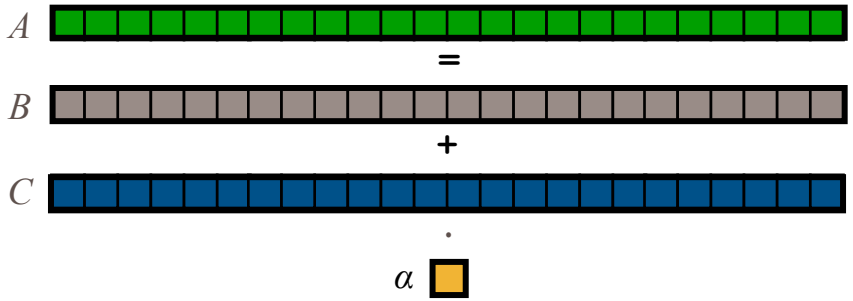
- A Brief Tour of Chapel Features
- Summary and Resources

STREAM Triad: a trivial parallel computation

Given: m-element vectors A, B, C

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

In pictures:

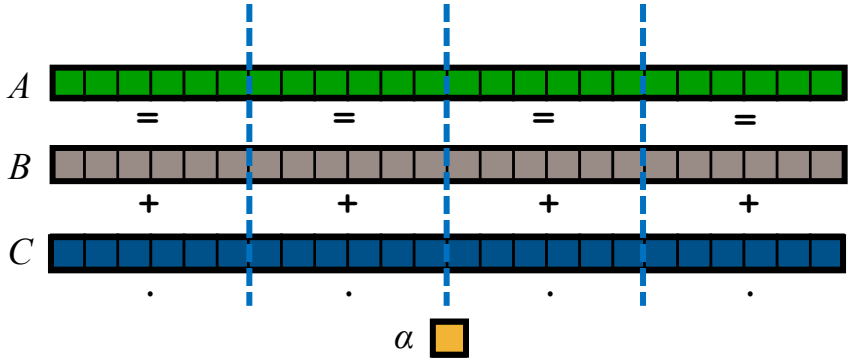


STREAM Triad: a trivial parallel computation

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):

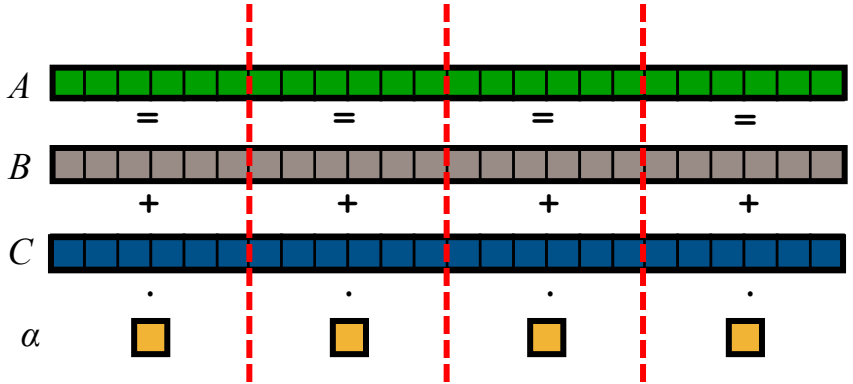


STREAM Triad: a trivial parallel computation

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):

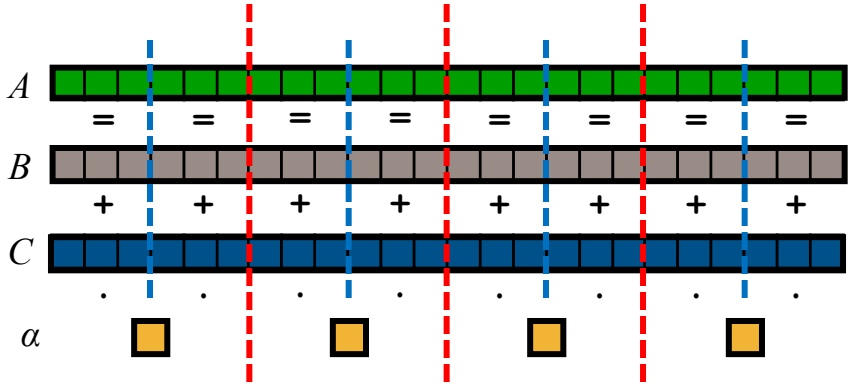


STREAM Triad: a trivial parallel computation

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: C + MPI

```c
#include <hpcc.h>


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 );
```

```c
  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;
}
```

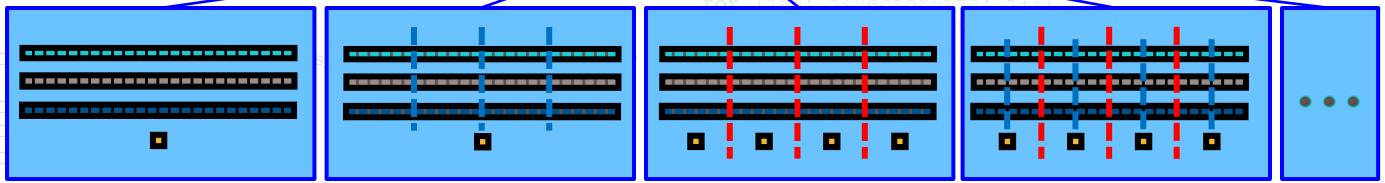
STREAM Triad: C + MPI + OpenMP

```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;
}
```

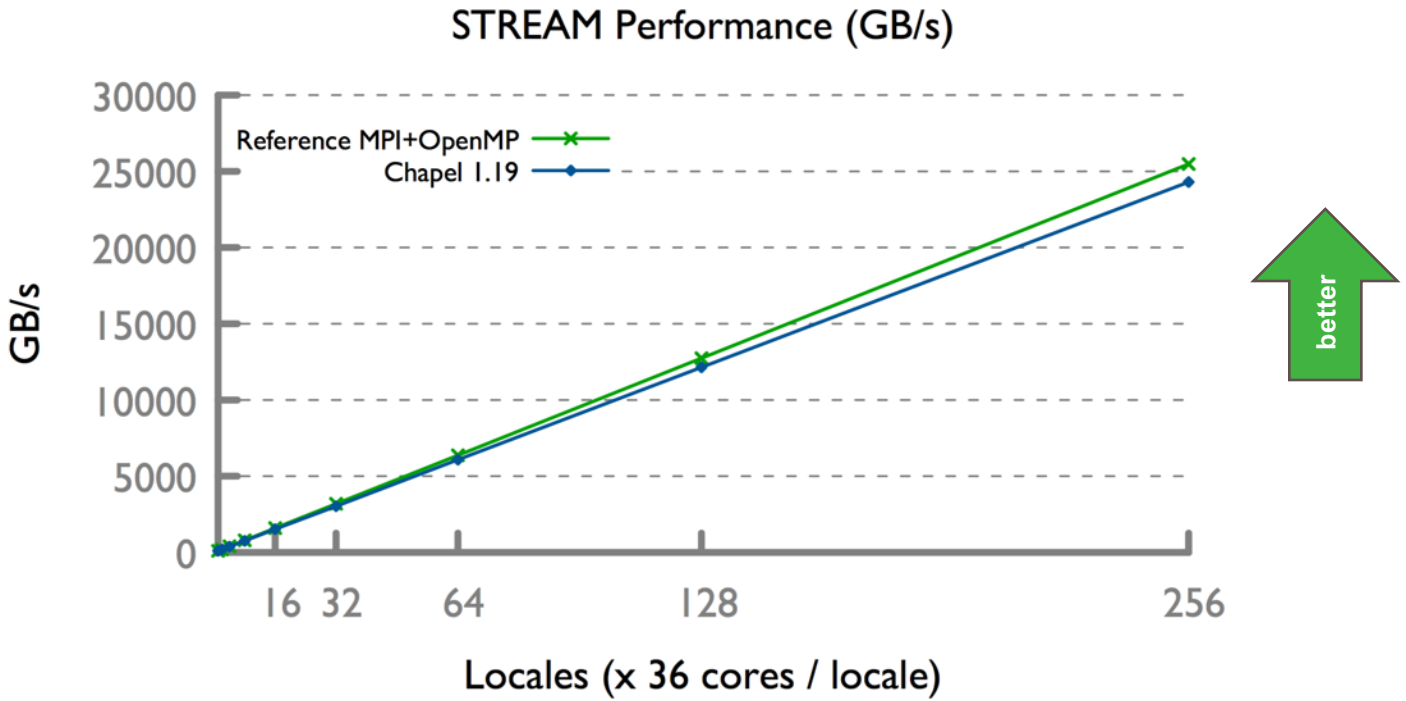
STREAM Triad: Chapel

```
use …;

config const m = 1000,
             alpha = 3.0;

const ProblemSpace = {1..m} dmapped …;

var A, B, C: [ProblemSpace] real;

B = 2.0;
C = 1.0;

A = B + alpha * C;
```

The special sauce:
How should this index set—and the arrays and computations over it—be mapped to the system?

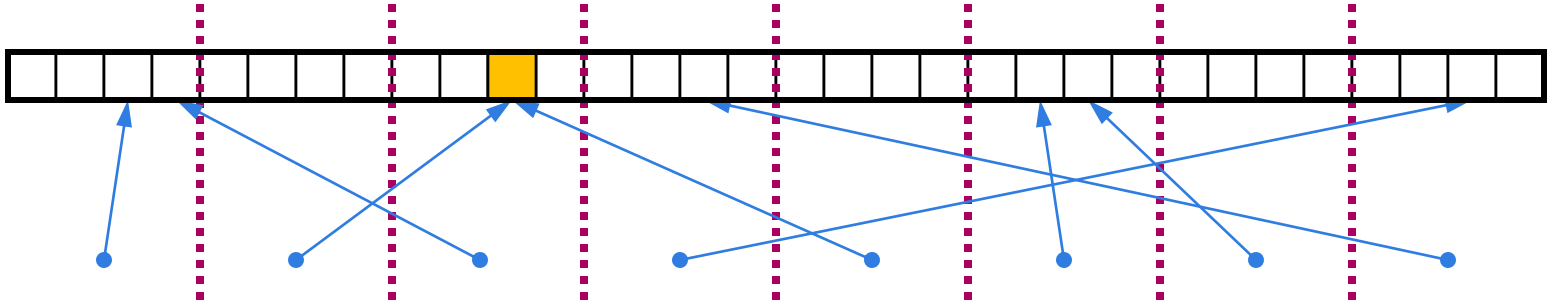


HPCC STREAM Triad: Chapel vs. C+MPI+OpenMP



HPCC Random Access (RA)

Data Structure: distributed table



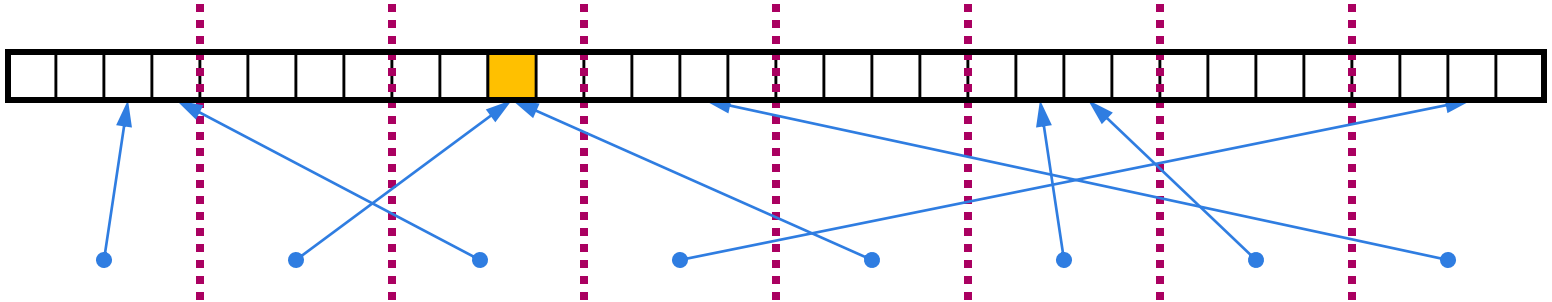
Computation: update random table locations in parallel

Two variations:

- **lossless:** don't allow any updates to be lost
- **lossy:** permit some fraction of updates to be lost

HPCC Random Access (RA)

Data Structure: distributed table



Computation: update random table locations in parallel

Two variations:

- **lossless:** don't allow any updates to be lost
- **lossy:** permit some fraction of updates to be lost

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: MPI kernel comment vs. Chapel

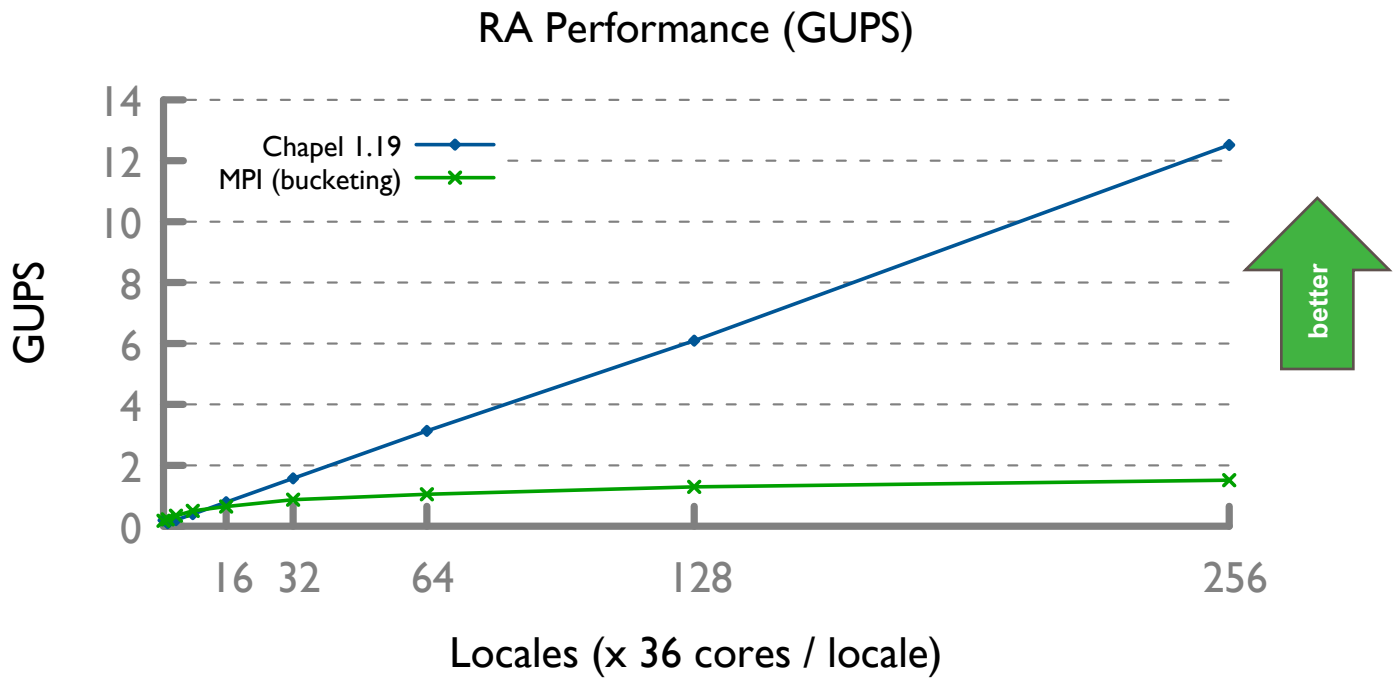
Chapel Kernel

```
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: MPI vs. Chapel

Chapel Kernel

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

HPCC RA: MPI vs. Chapel

Chapel Kernel

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

```
/* 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.dty
          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
 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);
                                        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);
```

```
                                                          e,
                                                          pe,
}
/* 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);
```

Why Consider New Languages at all?

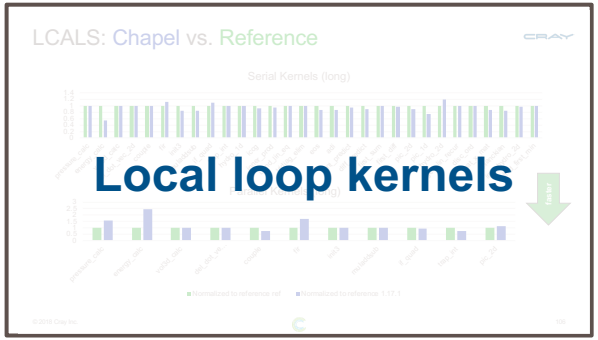
Syntax	• High level, elegant syntax • Improve programmer productivity
Semantics	• Static analysis can help with correctness • We need a compiler (front-end)
Performance	• If optimizations are needed to get performance • We need a compiler (back-end)
Algorithms	• Language defines what is easy and hard • Influences algorithmic thinking

[Source: Kathy Yelick, CHIUW 2018 keynote: *Why Languages Matter More Than Ever*]

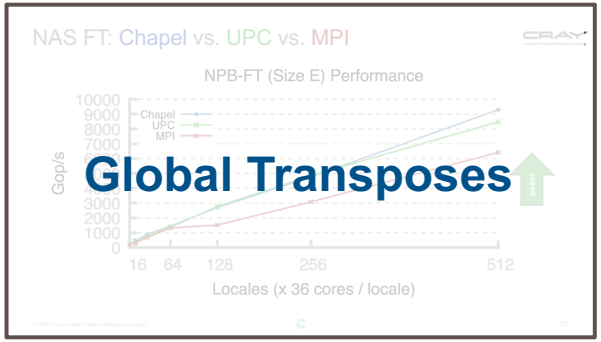
HPC Patterns: Chapel vs. Reference

CRAY
a Hewlett Packard Enterprise company

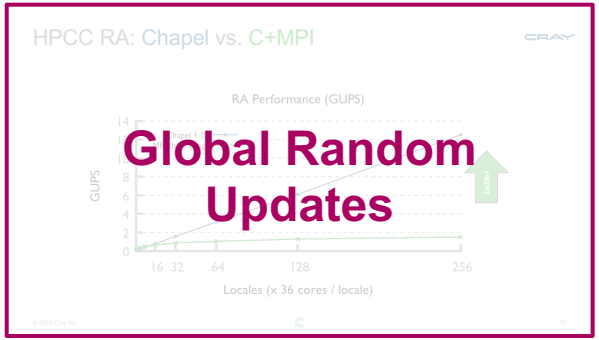
LCALS



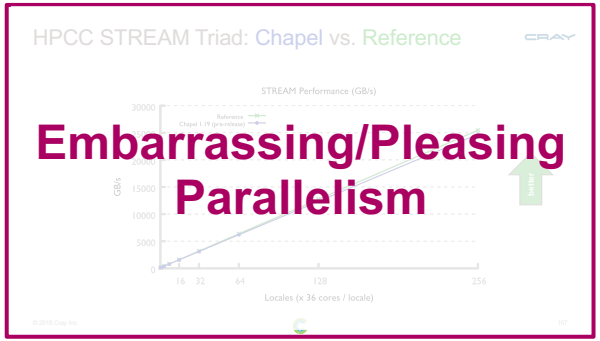
NAS FT



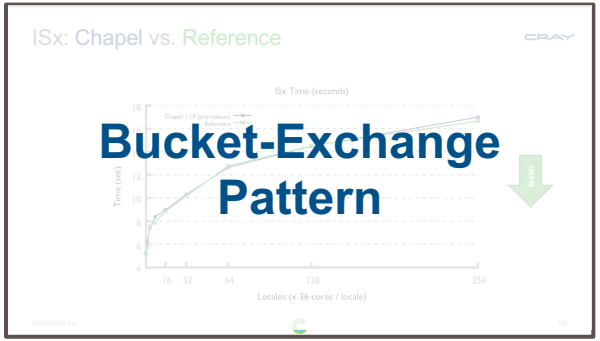
HPCC RA



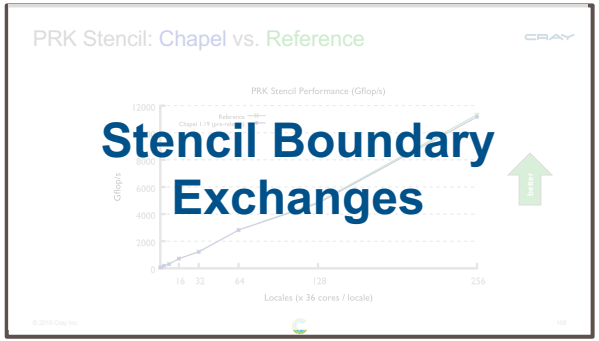
STREAM Triad



ISx

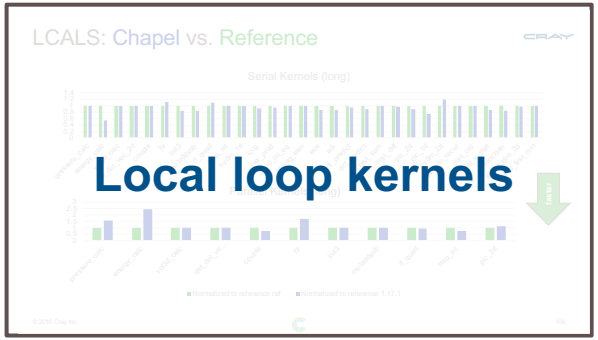


PRK Stencil

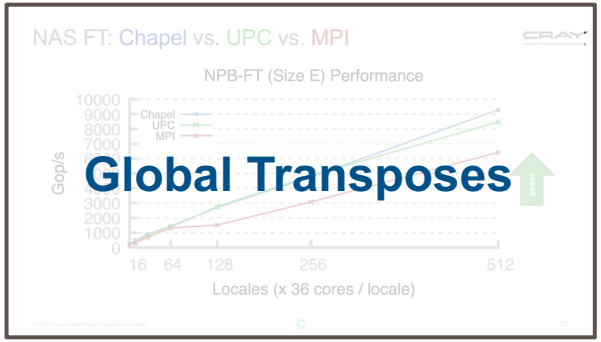


HPC Patterns: Chapel vs. Reference

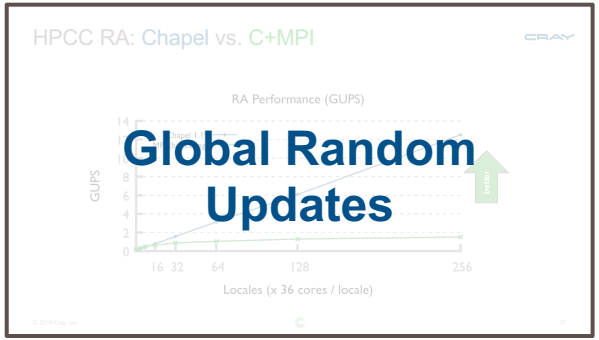
LCALS



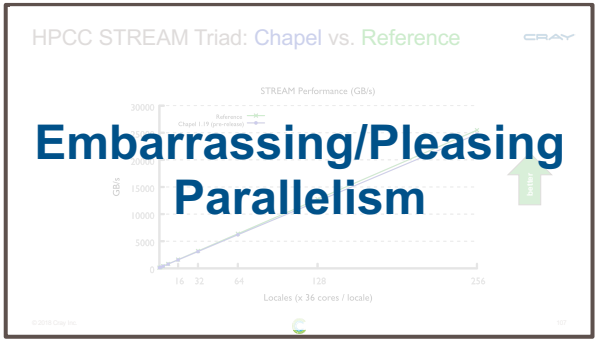
NAS FT



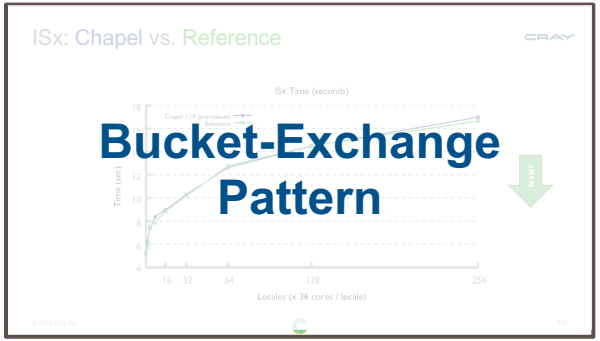
HPCC RA



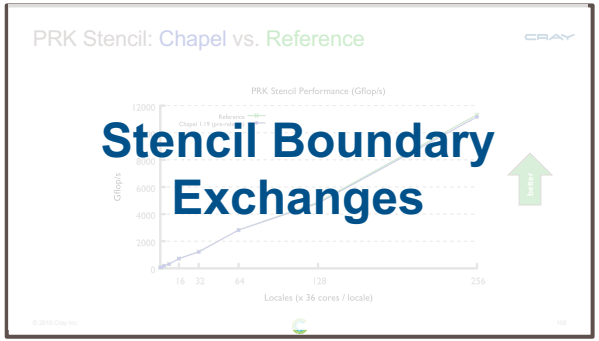
STREAM Triad



ISx

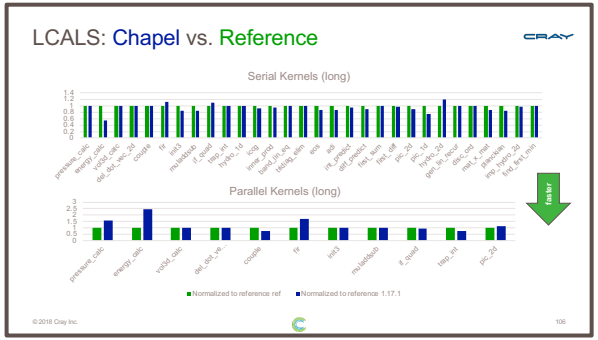


PRK Stencil

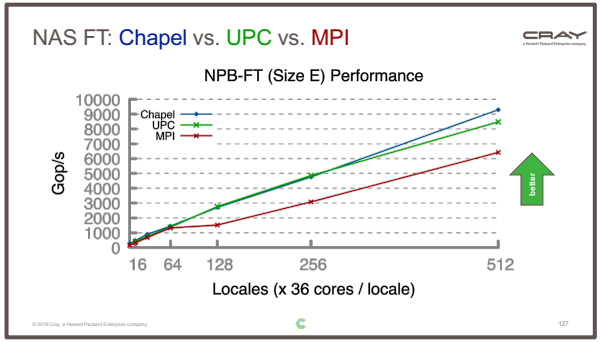


HPC Patterns: Chapel vs. Reference

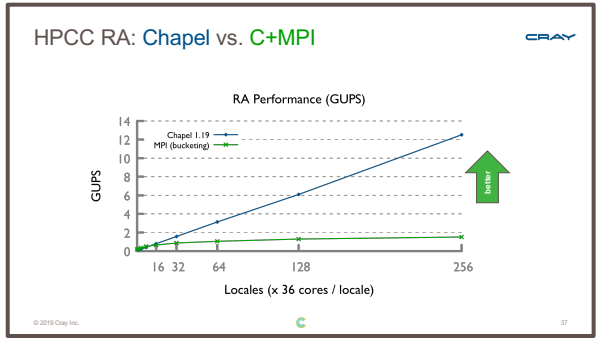
LCALS

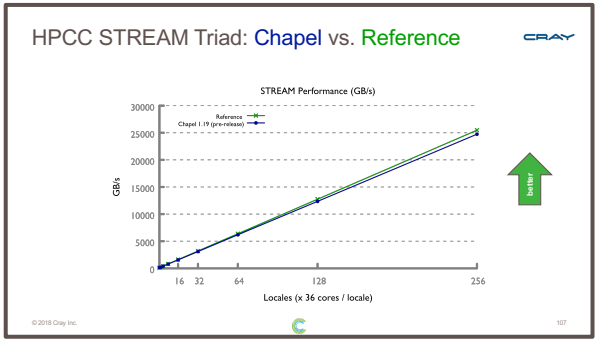


NAS FT

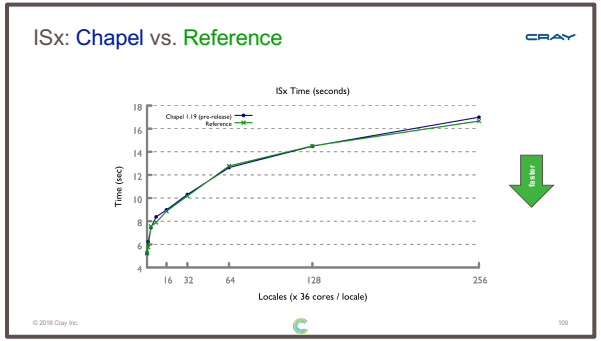


HPCC RA

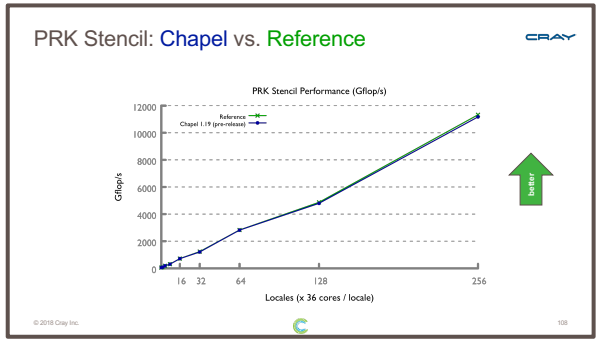




STREAM Triad



ISx



PRK Stencil

More on Chapel performance online at:
https://chapel-lang.org/performance.html

Notable Applications of Chapel

ChplUltra: Simulating Ultralight Dark Matter
Nikhil Padmanabhan et al.
Yale University

CHAMPS: 3D Computational Fluid Dynamics
Simon Bourgault-Côté, Matthieu Parenteau, et al.
École Polytechnique Montréal

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

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

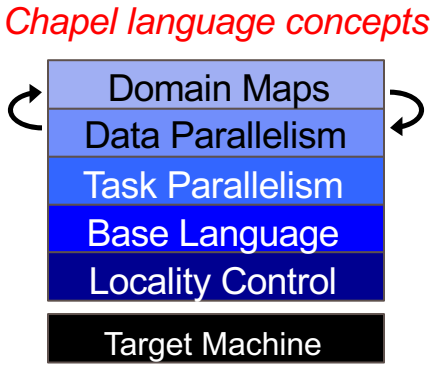
ChOp: Chapel-based Optimization
Tiago Carneiro, Nouredine Melab, et al.
INRIA Lille, France

CrayAI: Distributed Machine Learning
Cray, a Hewlett Packard Enterprise Company

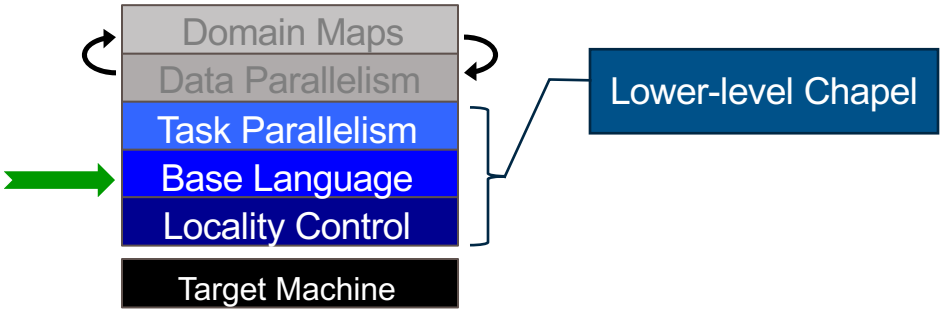
For more information, see: https://chapel-lang.org/poweredby.html

A Brief Tour of Chapel Features

Chapel Feature Areas



Base Language



Base Language Features, by example

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
config const n = 10;

for f in fib(n) do
  writeln(f);
```

```
0
1
1
2
3
5
8
…
```

Base Language Features, by example

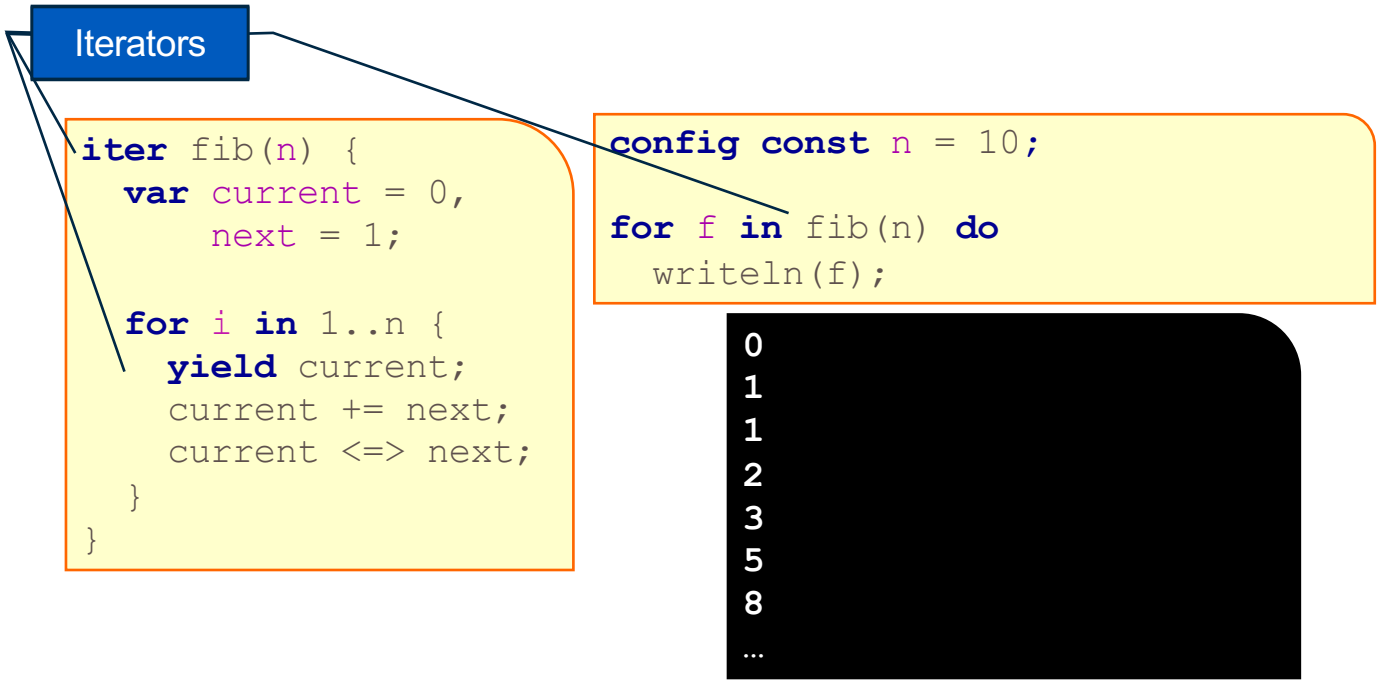
Configurable declarations
(support command-line overrides)
`./fib --n=1000000`

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
config const n = 10;

for f in fib(n) do
  writeln(f);
```

```
0
1
1
2
3
5
8
...
```

Base Language Features, by example



Base Language Features, by example

Static type inference for:
- arguments
- return types
- variables

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
config const n = 10;

for f in fib(n) do
  writeln(f);
```

```
0
1
1
2
3
5
8
...
```

Base Language Features, by example

Explicit types also supported

```
iter fib(n: int): int {
  var current: int = 0,
      next: int = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
config const n: int = 10;

for f in fib(n) do
  writeln(f);
```

```
0
1
1
2
3
5
8
...
```

Base Language Features, by example

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

```
config const n = 10;

for f in fib(n) do
  writeln(f);
```

```
0
1
1
2
3
5
8
…
```


Base Language Features, by example

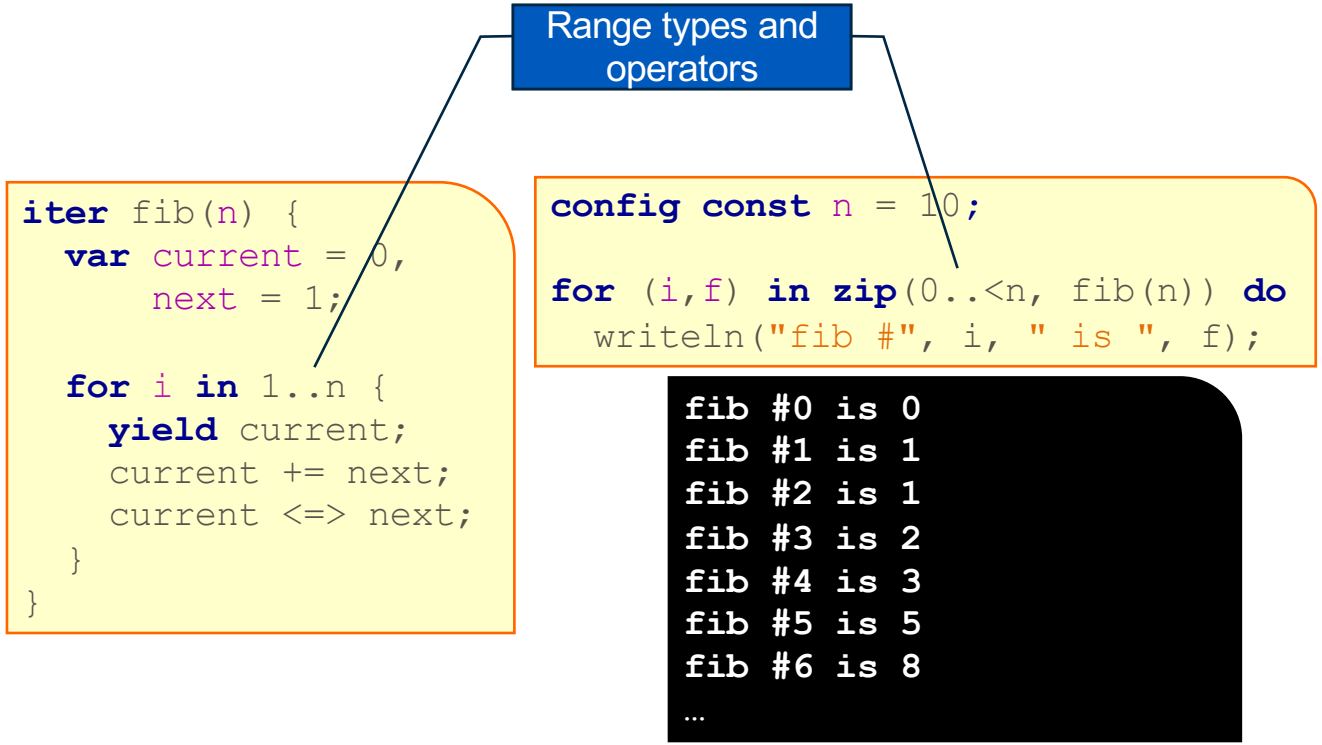
```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

Zippered iteration

```
config const n = 10;

for (i,f) in zip(0..<n, fib(n)) do
  writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
…
```


Base Language Features, by example



Base Language Features, by example

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

Tuples

```
config const n = 10;

for (i,f) in zip(0..<n, fib(n)) do
  writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
…
```

Base Language Features, by example

```
iter fib(n) {
  var current = 0,
      next = 1;

  for i in 1..n {
    yield current;
    current += next;
    current <=> next;
  }
}
```

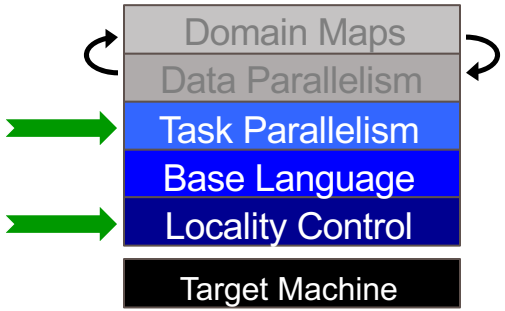
```
config const n = 10;

for (i,f) in zip(0..<n, fib(n)) do
  writeln("fib #", i, " is ", f);
```

```
fib #0 is 0
fib #1 is 1
fib #2 is 1
fib #3 is 2
fib #4 is 3
fib #5 is 5
fib #6 is 8
…
```


Other Base Language Features

- **Object-oriented programming** (value- and reference-based)
  - Managed objects and lifetime checking
  - Nilable vs. non-nilable class variables
- **Generic programming / polymorphism**
- **Error-handling**
- **Compile-time meta-programming**
- **Modules** (supporting namespaces)
- **Procedure overloading / filtering**
- **Arguments:** default values, intents, name-based matching, type queries
- and more…

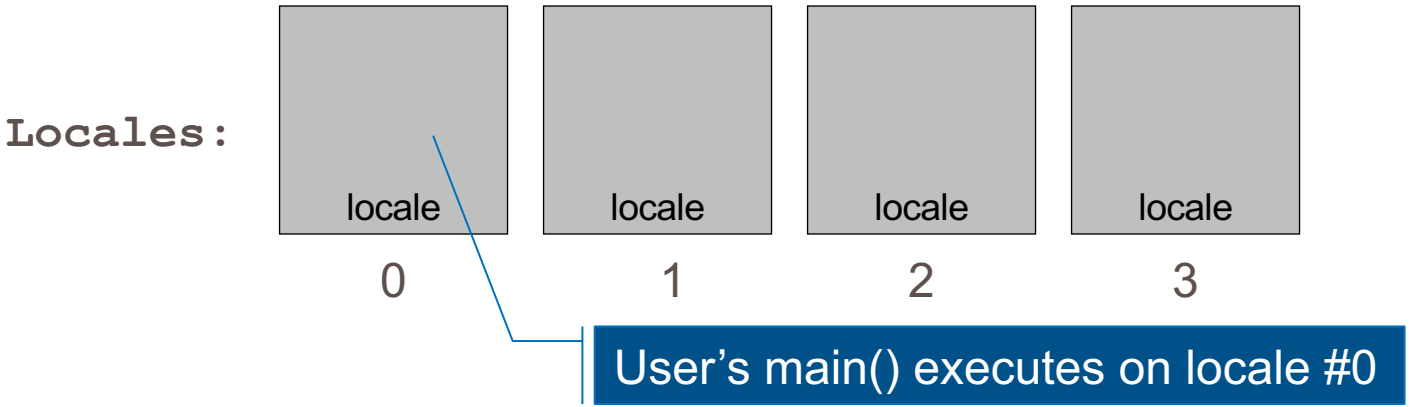
Task Parallelism and Locality Control



Locales in Chapel

- Locales can run tasks and store variables
  - Think “compute node”
  - Number of locales specified on executable’s command-line

```
> ./myProgram --numLocales=4     # or `-nl 4`
```



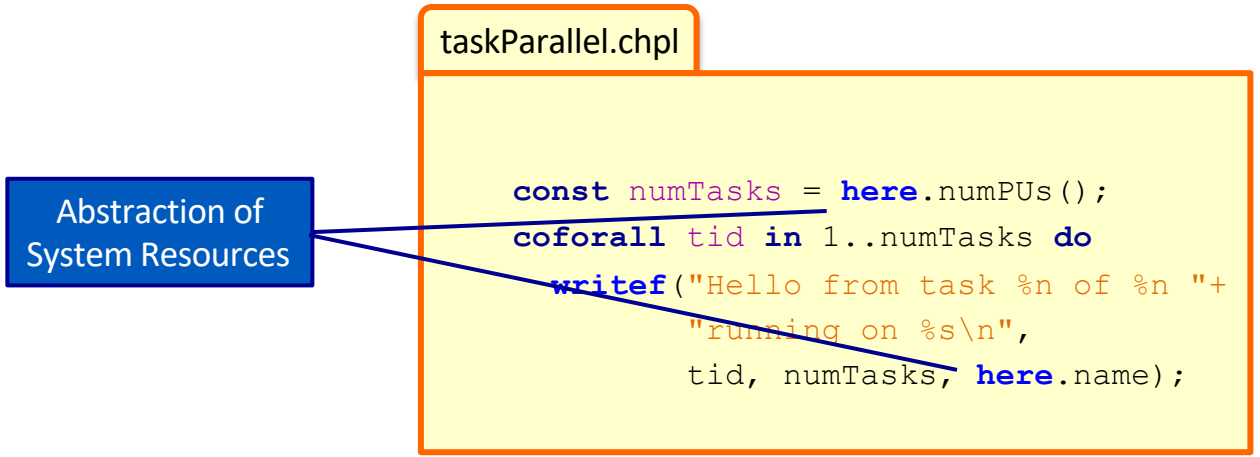
Task Parallelism and Locality, by example

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```


Task Parallelism and Locality, by example




```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example

High-Level Task Parallelism

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```


Task Parallelism and Locality, by example

So far, this is a shared memory program

Nothing refers to remote locales, explicitly or implicitly

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel
Hello from task 2 of 2 running on n1032
Hello from task 1 of 2 running on n1032
```

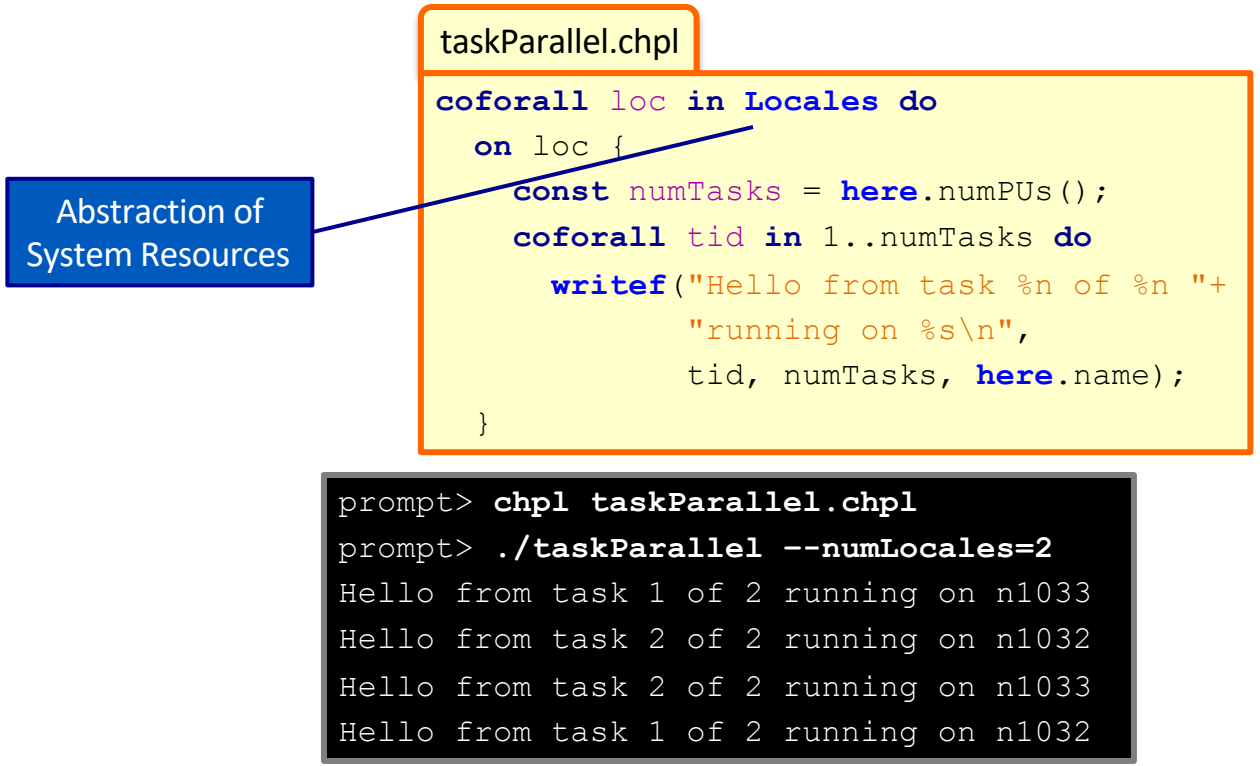
Task Parallelism and Locality, by example

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Task Parallelism and Locality, by example



Task Parallelism and Locality, by example

High-Level Task Parallelism

taskParallel.chpl

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

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```


Task Parallelism and Locality, by example

taskParallel.chpl

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

Control of Locality/Affinity

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```


Task Parallelism and Locality, by example

taskParallel.chpl

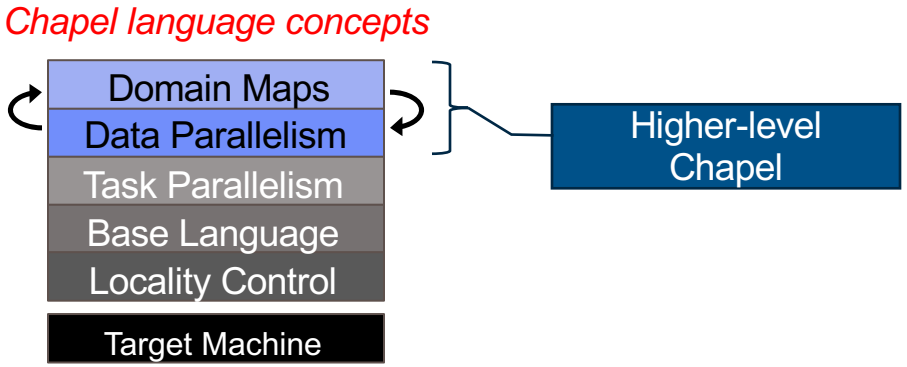
```
coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      writef("Hello from task %n of %n "+
             "running on %s\n",
             tid, numTasks, here.name);
  }
```

```
prompt> chpl taskParallel.chpl
prompt> ./taskParallel --numLocales=2
Hello from task 1 of 2 running on n1033
Hello from task 2 of 2 running on n1032
Hello from task 2 of 2 running on n1033
Hello from task 1 of 2 running on n1032
```

Other Task Parallel Features

- **begin / cobegin statements:** other ways of creating tasks
- **atomic / synchronized variables:** for sharing data & coordination
- **task intents / task-private variables:** ways of having tasks refer to variables

Data Parallelism in Chapel



Data Parallelism, by example

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```


Data Parallelism, by example

Domains (Index Sets)

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```


Data Parallelism, by example

Arrays

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```


Data Parallelism, by example

Data-Parallel Forall Loops

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```


Data Parallelism, by example

So far, this is a shared memory program

Nothing refers to remote locales, explicitly or implicitly

dataParallel.chpl

```
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```


Distributed Data Parallelism, by example

Domain Maps
(Map Data Parallelism to the System)

dataParallel.chpl

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
        dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```


Distributed Data Parallelism, by example

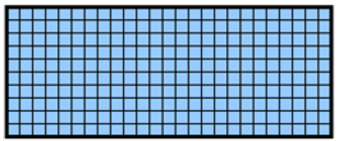
dataParallel.chpl

```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
        dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

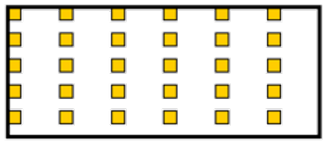
```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```


Other Data Parallel Features

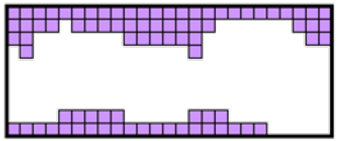
- **Parallel Iterators and Zippering**
- **Slicing:** refer to subarrays using ranges / domains
- **Promotion:** execute scalar functions in parallel using array arguments
- **Reductions:** collapse arrays to scalars or subarrays
- **Scans:** parallel prefix operations
- **Several Domain/Array Types:**



dense



strided



sparse

associative

Summary and Resources

Summary

Chapel cleanly and orthogonally supports…

…expression of parallelism and locality

…specifying how to map computations to the system

Chapel is powerful:

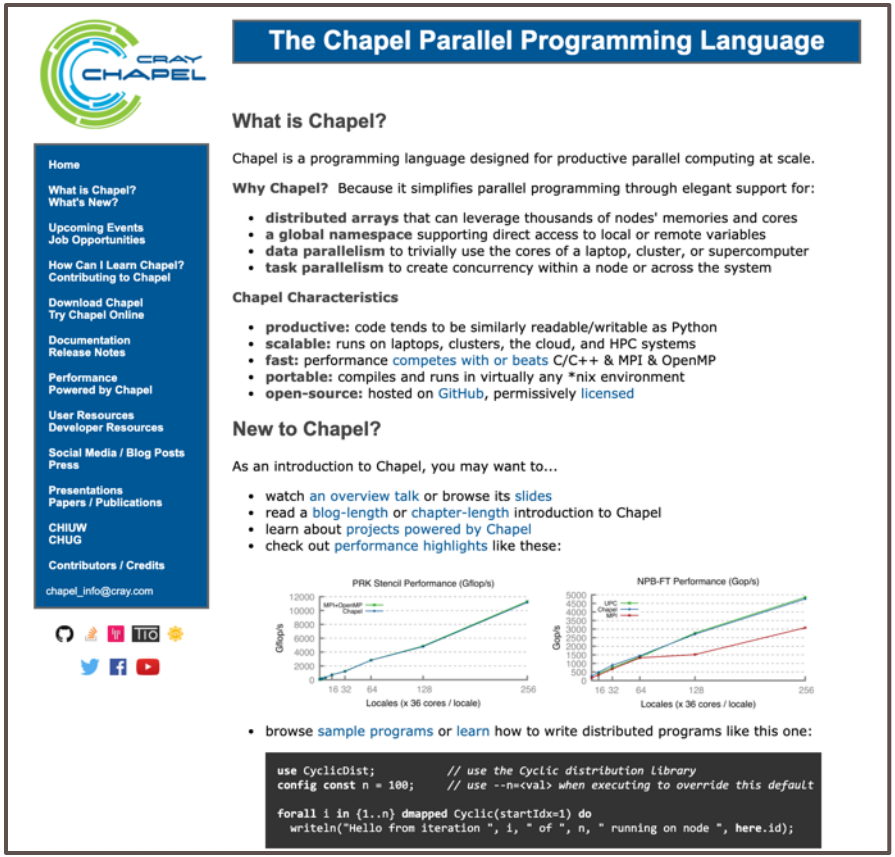
- supports succinct, straightforward code
- can result in performance that competes with (or beats) C+MPI+OpenMP

Chapel is attractive to computational scientists and Python programmers

Chapel Homepage

https://chapel-lang.org

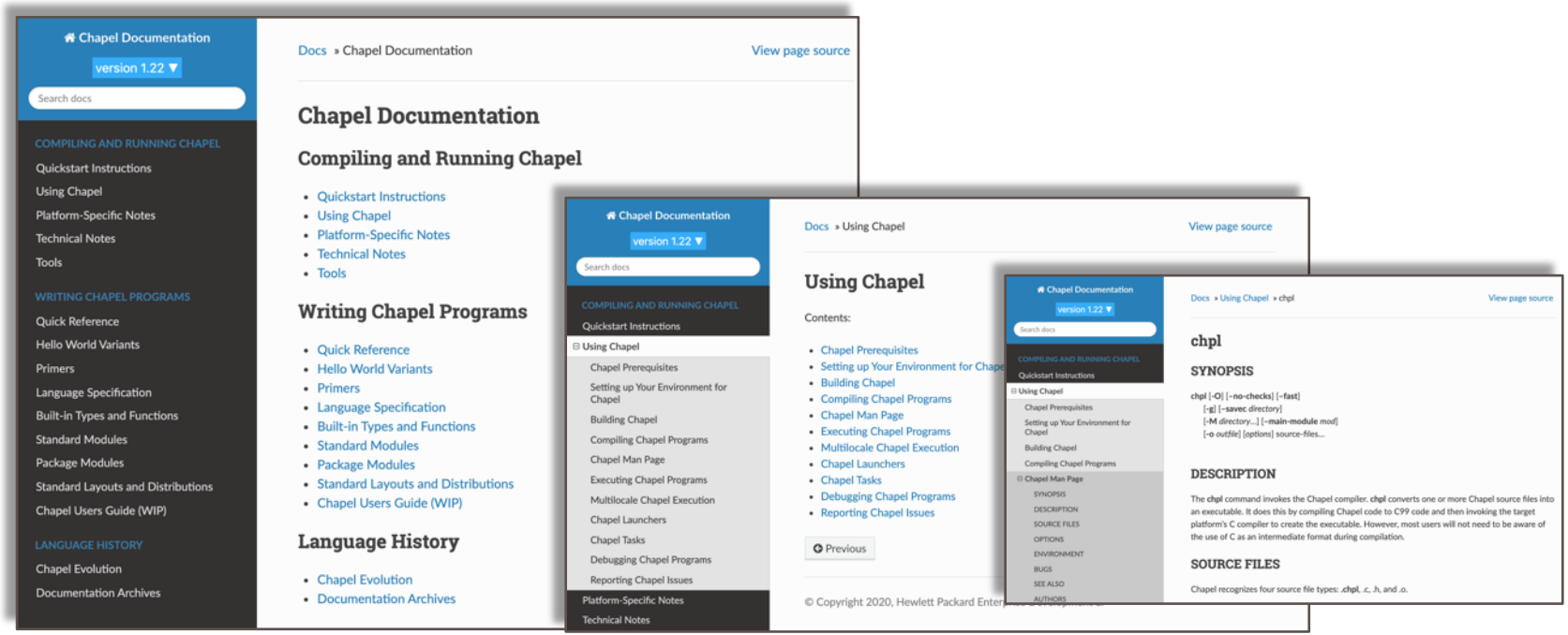
- downloads
- presentations
- papers
- resources
- documentation



Chapel Documentation

CRAY
a Hewlett Packard Enterprise company

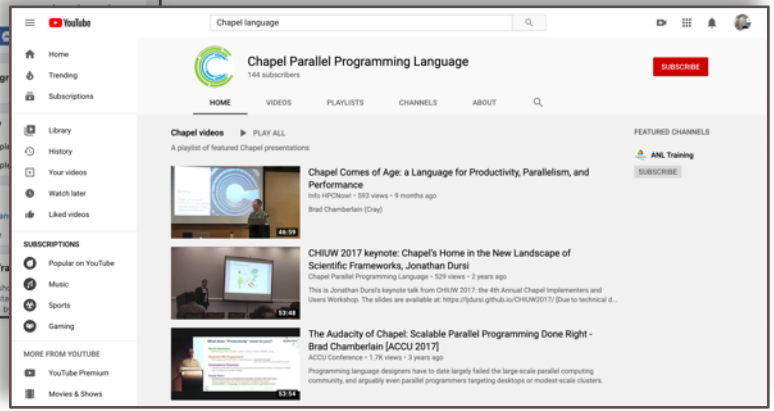
https://chapel-lang.org/docs: ~270 pages, including primer examples



Chapel Social Media (no account required)

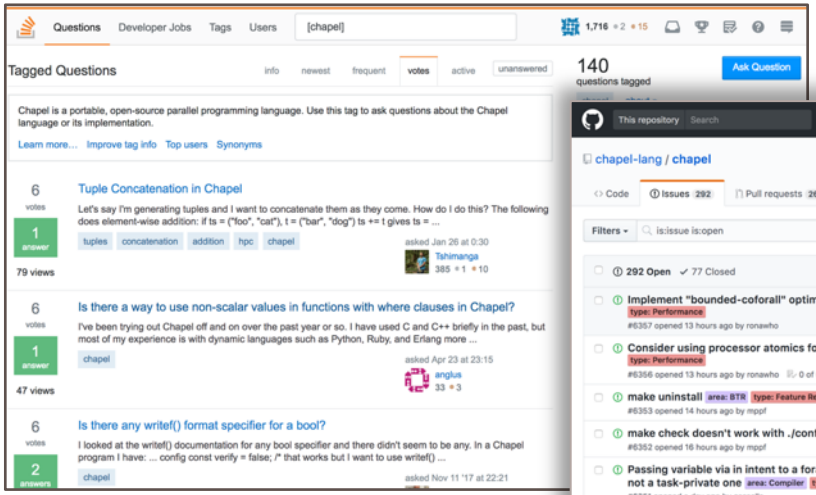
http://twitter.com/ChapelLanguage

http://facebook.com/ChapelLanguage

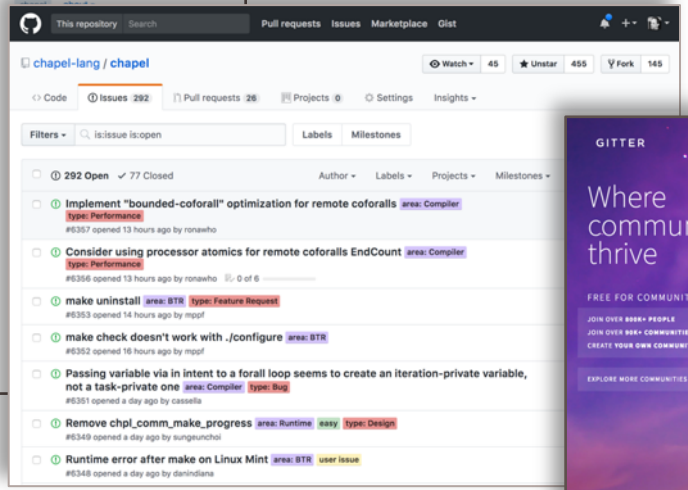


https://www.youtube.com/channel/UCHmm27bYjhknK5mU7ZzPGsQ/

Chapel Community



https://stackoverflow.com/questions/tagged/chapel



https://github.com/chapel-lang/chapel/issues

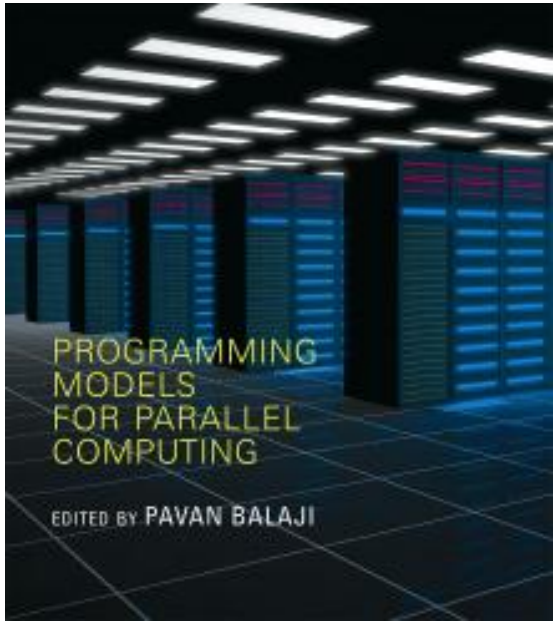
https://gitter.im/chapel-lang/chapel

read-only mailing list: chapel-announce@lists.sourceforge.net (~15 mails / year)

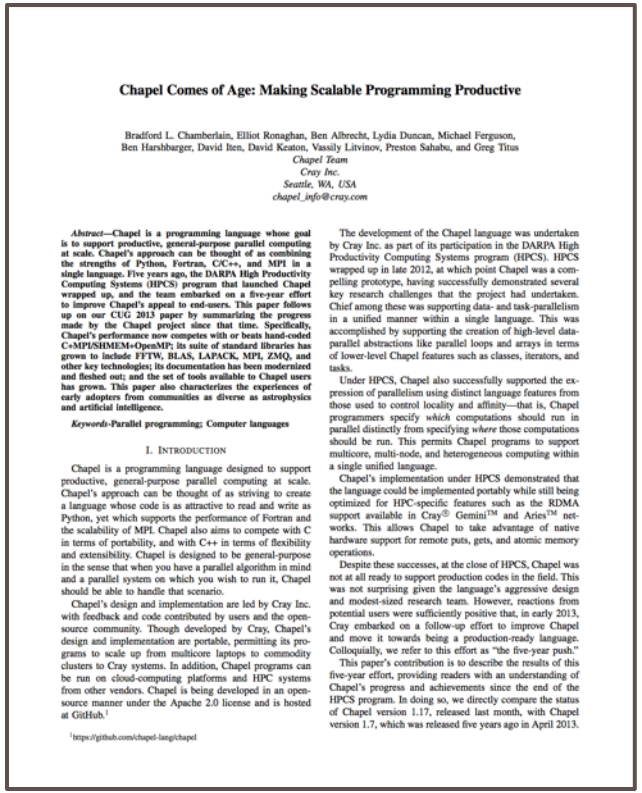
Suggested Reading: Historical Overview

Chapel chapter from *Programming Models for Parallel Computing*

- a detailed overview of Chapel's history, motivating themes, features
- published by MIT Press, November 2015
- edited by Pavan Balaji (Argonne)
- chapter is also available online



Suggested Reading: Mid-project Progress (2013–2018)



Chapel Comes of Age: Making Scalable Programming Productive

Bradford L. Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson,
Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, and Greg Titus
Chapel Team
Cray Inc.
Seattle, WA, USA
chapel_info@cray.com

***Abstract*—Chapel is a programming language whose goal is to support productive, general-purpose parallel computing at scale. Chapel's approach can be thought of as combining the strengths of Python, Fortran, C/C++, and MPI in a single language. Five years ago, the DARPA High Productivity Computing Systems (HPCS) program that launched Chapel wrapped up, and the team embarked on a five-year effort to improve Chapel's appeal to end-users. This paper follows up on our CUG 2013 paper by summarizing the progress made by the Chapel project since that time. Specifically, Chapel's performance now competes with or beats hand-coded C+MPI/SHMEM+OpenMP; its suite of standard libraries has grown to include FFTW, BLAS, LAPACK, MPI, ZMQ, and other key technologies; its documentation has been modernized and fleshed out; and the set of tools available to Chapel users has grown. This paper also characterizes the experiences of early adopters from communities as diverse as astrophysics and artificial intelligence.**

***Keywords*-Parallel programming; Computer languages**

I. Introduction

Chapel is a programming language designed to support productive, general-purpose parallel computing at scale. Chapel's approach can be thought of as striving to create a language whose code is as attractive to read and write as Python, yet which supports the performance of Fortran and the scalability of MPI. Chapel also aims to compete with C in terms of portability, and with C++ in terms of flexibility and extensibility. Chapel is designed to be general-purpose in the sense that when you have a parallel algorithm in mind and a parallel system on which you wish to run it, Chapel should be able to handle that scenario.

Chapel's design and implementation are led by Cray Inc. with feedback and code contributed by users and the open-source community. Though developed by Cray, Chapel's design and implementation are portable, permitting its programs to scale up from multicore laptops to commodity clusters to Cray systems. In addition, Chapel programs can be run on cloud-computing platforms and HPC systems from other vendors. Chapel is being developed in an open-source manner under the Apache 2.0 license and is hosted at GitHub.[1]

[1]https://github.com/chapel-lang/chapel

The development of the Chapel language was undertaken by Cray Inc. as part of its participation in the DARPA High Productivity Computing Systems program (HPCS). HPCS wrapped up in late 2012, at which point Chapel was a compelling prototype, having successfully demonstrated several key research challenges that the project had undertaken. Chief among these was supporting data- and task-parallelism in a unified manner within a single language. This was accomplished by supporting the creation of high-level data-parallel abstractions like parallel loops and arrays in terms of lower-level Chapel features such as classes, iterators, and tasks.

Under HPCS, Chapel also successfully supported the expression of parallelism using distinct language features from those used to control locality and affinity—that is, Chapel programmers specify *which* computations should run in parallel distinctly from specifying *where* those computations should be run. This permits Chapel programs to support multicore, multi-node, and heterogeneous computing within a single unified language.

Chapel's implementation under HPCS demonstrated that the language could be implemented portably while still being optimized for HPC-specific features such as the RDMA support available in Cray® Gemini™ and Aries™ networks. This allows Chapel to take advantage of native hardware support for remote puts, gets, and atomic memory operations.

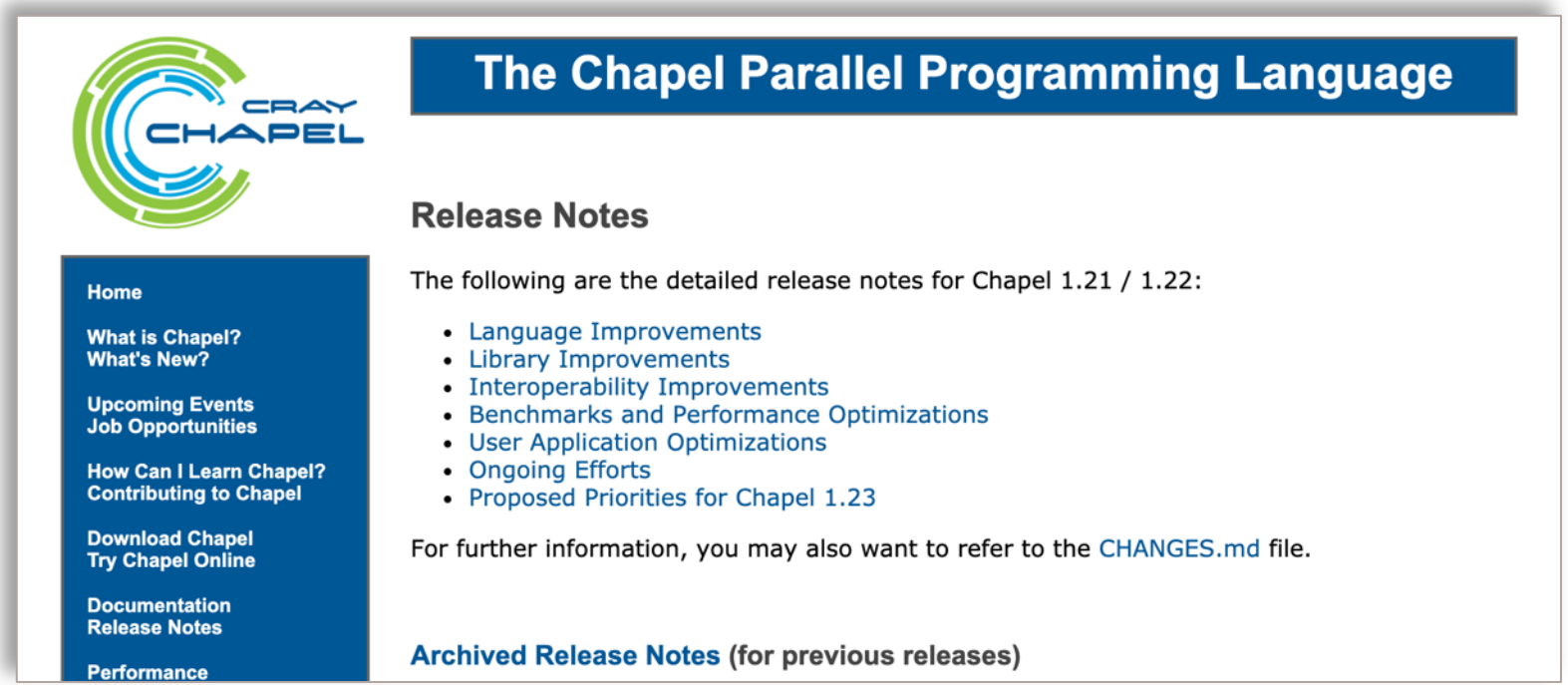
Despite these successes, at the close of HPCS, Chapel was not at all ready to support production codes in the field. This was not surprising given the language's aggressive design and modest-sized research team. However, reactions from potential users were sufficiently positive that, in early 2013, Cray embarked on a follow-up effort to improve Chapel and move it towards being a production-ready language. Colloquially, we refer to this effort as "the five-year push."

This paper's contribution is to describe the results of this five-year effort, providing readers with an understanding of Chapel's progress and achievements since the end of the HPCS program. In doing so, we directly compare the status of Chapel version 1.17, released last month, with Chapel version 1.7, which was released five years ago in April 2013.

[paper](#) and [slides](#) available at chapel-lang.org

Suggested Reading: The Very Latest

- Chapel release notes: https://chapel-lang.org/releaseNotes.html



Summary

Chapel cleanly and orthogonally supports…

…expression of parallelism and locality

…specifying how to map computations to the system

Chapel is powerful:

- supports succinct, straightforward code
- can result in performance that competes with (or beats) C+MPI+OpenMP

Chapel is attractive to computational scientists and Python programmers

SAFE HARBOR STATEMENT

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts.

These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

THANK YOU

QUESTIONS?

bradc@cray.com

@ChapelLanguage

chapel-lang.org

cray.com

@cray_inc

linkedin.com/company/cray-inc-/