Chapel Comes of Age:
A Language for Productivity, Parallelism, and Performance

Brad Chamberlain
HPC Knowledge Meeting `19
June 14, 2019
Who am I?

Education:

• Earned Ph.D. from University of Washington CSE in 2001
  • focused on the ZPL data-parallel array language
  • Remain associated with UW CSE as an Affiliate Professor

Industry:

• A Principal Engineer at Cray Inc.
  • The technical lead / a founding member of the Chapel project
What is Chapel?

**Chapel**: A modern parallel programming language

- portable & scalable
- open-source & collaborative

**Goals:**

- Support general parallel programming
  - “any parallel algorithm on any parallel hardware”
- 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 express my parallel computations 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.”
## 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
  • Arkouda: NumPy over Chapel
  • Summary and Resources
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]
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):**

\[
\begin{align*}
A & = \\
B & = + \\
C & = \cdot \\
& \alpha
\end{align*}
\]
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):

![Diagram showing parallel computation of STREAM Triad]
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):
#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 );

    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;
}
#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

```
#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;
}
```

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
/* Perform updates to main table. The scalar equivalent is: */

for (j=0; j<UPDATE); j++)
  imag = LocalRecvBuffer[bufferBase+j];
LocalOffset = (imag & (tparams.TableSize - 1)) -
  tparams.GlobalStartMyProc;
HPCC_Table[LocalOffset] ^= imag;
} else if (status.MPI_TAG == FINISHED_TAG)
  NumberReceiving--;
else
  MPI_Abort(MPI_COMM_WORLD, -1);
HPCC_Recv(LocalRecvBuffer, localBufferSize, tparams.dtype64, 
  MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, inmsg);

while (have_done || NumberReceiving > 0) {
  if (pendingUpdates == maxPendingUpdates) |
    Rand = (Rand << 1) & (65536); |
  if (GlobalOffset < tparams.Top)
    WhichPw = (GlobalOffset / (tparams.MinLocalTableSize + 1)) -
      tparams.GlobalStartMyProc;
HPCC_Table[LocalOffset] ^= Rand;
}

/* receive messages */
do {

  if (have_done) |
    if (status.MPI_TAG == UPDATE_TAG) |
      MPI_Get_count((status, tparams.dtype64, irecvUpdates);
      bufferBase = 0;
    for (j=0; j < recvUpdates; j++)
      imag = LocalRecvBuffer[bufferBase+j];
    LocalOffset = (imag & (tparams.TableSize - 1)) -
      tparams.GlobalStartMyProc;
    HPCC_Table[LocalOffset] ^= imag;
  } else if (status.MPI_TAG == FINISHED_TAG)
    NumberReceiving--;
  else
    MPI_Abort(MPI_COMM_WORLD, -1);

  if (have_done)
    if (status.MPI_TAG == UPDATE_TAG) |
      MPI_Get_count((status, tparams.dtype64, irecvUpdates);
      bufferBase = 0;
    for (j=0; j < recvUpdates; j++)
      imag = LocalRecvBuffer[bufferBase+j];
    LocalOffset = (imag & (tparams.TableSize - 1)) -
      tparams.GlobalStartMyProc;
    HPCC_Table[LocalOffset] ^= imag;
  } else if (status.MPI_TAG == FINISHED_TAG)
    NumberReceiving--;
  else
    MPI_Abort(MPI_COMM_WORLD, -1);

  if (have_done & NumberReceiving > 0) {
    MPI_Test(inreq, have_done, status);
    if (have_done)
      if (status.MPI_TAG == UPDATE_TAG) |
        MPI_Get_count((status, tparams.dtype64, irecvUpdates);
        bufferBase = 0;
      for (j=0; j < recvUpdates; j++)
        imag = LocalRecvBuffer[bufferBase+j];
      LocalOffset = (imag & (tparams.TableSize - 1)) -
        tparams.GlobalStartMyProc;
      HPCC_Table[LocalOffset] ^= imag;
    } else if (status.MPI_TAG == FINISHED_TAG)
      NumberReceiving--;
    else
      MPI_Abort(MPI_COMM_WORLD, -1);
  } while (have_done & NumberReceiving > 0);

  if (have_done & NumberReceiving > 0) {
    MPI_Test(inreq, have_done, status);
    if (have_done)
      if (status.MPI_TAG == UPDATE_TAG) |
        MPI_Get_count((status, tparams.dtype64, irecvUpdates);
        bufferBase = 0;
      for (j=0; j < recvUpdates; j++)
        imag = LocalRecvBuffer[bufferBase+j];
      LocalOffset = (imag & (tparams.TableSize - 1)) -
        tparams.GlobalStartMyProc;
      HPCC_Table[LocalOffset] ^= imag;
    } else if (status.MPI_TAG == FINISHED_TAG)
      NumberReceiving--;
    else
      MPI_Abort(MPI_COMM_WORLD, -1);
  } 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, 
    ipsUpdates);
  HPCC_Recv(LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe, 
    UPDATE_TAG, MPI_COMM_WORLD, outreq);
  pendingUpdates -= peUpdates;
} else
/* send remaining updates in buckets */
do {
  if (have_done)
    if (status.MPI_TAG == UPDATE_TAG) |
      MPI_Get_count((status, tparams.dtype64, irecvUpdates);
      bufferBase = 0;
    for (j=0; j < recvUpdates; j++)
      imag = LocalRecvBuffer[bufferBase+j];
    LocalOffset = (imag & (tparams.TableSize - 1)) -
      tparams.GlobalStartMyProc;
    HPCC_Table[LocalOffset] ^= imag;
  } else if (status.MPI_TAG == FINISHED_TAG)
    NumberReceiving--;
  else
    MPI_Abort(MPI_COMM_WORLD, -1);

  if (have_done) |
    if (status.MPI_TAG == UPDATE_TAG) |
      MPI_Get_count((status, tparams.dtype64, irecvUpdates);
      bufferBase = 0;
    for (j=0; j < recvUpdates; j++)
      imag = LocalRecvBuffer[bufferBase+j];
    LocalOffset = (imag & (tparams.TableSize - 1)) -
      tparams.GlobalStartMyProc;
    HPCC_Table[LocalOffset] ^= imag;
  } else if (status.MPI_TAG == FINISHED_TAG)
    NumberReceiving--;
  else
    MPI_Abort(MPI_COMM_WORLD, -1);

  if (have_done & NumberReceiving > 0) {
    MPI_Test(inreq, have_done, status);
    if (have_done)
      if (status.MPI_TAG == UPDATE_TAG) |
        MPI_Get_count((status, tparams.dtype64, irecvUpdates);
        bufferBase = 0;
      for (j=0; j < recvUpdates; j++)
        imag = LocalRecvBuffer[bufferBase+j];
      LocalOffset = (imag & (tparams.TableSize - 1)) -
        tparams.GlobalStartMyProc;
      HPCC_Table[LocalOffset] ^= imag;
    } else if (status.MPI_TAG == FINISHED_TAG)
      NumberReceiving--;
    else
      MPI_Abort(MPI_COMM_WORLD, -1);
  } 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, 
    ipsUpdates);
  HPCC_Recv(LocalSendBuffer, peUpdates, tparams.dtype64, (int)pe, 
    UPDATE_TAG, MPI_COMM_WORLD, outreq);
  pendingUpdates -= peUpdates;
}
Perform updates to main table. The scalar equivalent is:

```chapel
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;
}"
HPCC RA: Chapel vs. C+MPI

RA Performance (GUPS)

<table>
<thead>
<tr>
<th>Locales (x 36 cores / locale)</th>
<th>Chapel 1.19</th>
<th>MPI (bucketing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.19</td>
<td>0.2</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>0.4</td>
</tr>
<tr>
<td>64</td>
<td>3.33</td>
<td>0.6</td>
</tr>
<tr>
<td>128</td>
<td>6.67</td>
<td>1.2</td>
</tr>
<tr>
<td>256</td>
<td>13</td>
<td>2.4</td>
</tr>
</tbody>
</table>
HPCC RA: MPI vs. Chapel

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

Chapel Kernel
for all (r) in zip(Updates, RAstream()) do
T[r & indexMask].xor(r);
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

- **LCALS**
  - Local loop kernels

- **HPCC RA**
  - Global Random Updates

- **ISx**
  - PRK Stencil

- **HPCC STREAM Triad**
  - Embarrassing/Pleasing Parallelism

- **Bucket-Exchange Pattern**

- **PRK Stencil**
  - Stencil Boundary Exchanges

Nightly performance tickers online at: [https://chapel-lang.org/perf-nightly.html](https://chapel-lang.org/perf-nightly.html)
HPC Patterns: Chapel vs. Reference

- **LCALS**: Chapel vs. Reference
  - Serial Kernels (long)
  - Parallel Kernels (long)

- **HPCC STREAM Triad**: Chapel vs. Reference
  - STREAM Performance (G&H)

- **HPCC RA**: Chapel vs. C+MPI
  - RA Performance (GUPS)

- **ISx**: Chapel vs. Reference
  - Six Time (seconds)

- **PRK Stencil**: Chapel vs. Reference
  - PRK Stencil Performance (G&H)

Nightly performance tickers online at: [https://chapel-lang.org/perf-nightly.html](https://chapel-lang.org/perf-nightly.html)
A Brief Tour of Chapel Features
Chapel Feature Areas

Chapel language concepts

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
Base Language
Base Language Features, by example

```plaintext
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**

```plaintext
iter fib(n) {
    var current = 0,
        next = 1;
    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```plaintext
config const n = 10;
for f in fib(n) do writeln(f);
```

Configuration declarations (support command-line overrides)
`.fib --n=1000000`
Base Language Features, by example

Iterators

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

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

```clu-lang
config const n = 10;
for f in fib(n) do
  writeln(f);
```

```
0
1
1
2
3
5
8...
```
Base Language Features, by example

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

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

```c
config const n = 10;
for f in fib(n) do
    writeln(f);
```

Static type inference for:
- arguments
- return types
- variables
Base Language Features, by example

```
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);
```

Explicit types also supported

Static Type Inference for:
- arguments
- return types
- variables

Explicit types also supported

Static Type Inference for:
- arguments
- return types
- variables

```
0
1
1
2
3
5
8
...  
```
Base Language Features, by example

```plaintext
iter fib(n) {
    var current = 0,
        next = 1;
    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```

```plaintext
config const n = 10;
for f in fib(n) do
    writeln(f);
```

0
1
1
2
3
5
8
...

Base Language Features, by example

```javascript
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);
```

Zippered iteration

```
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

```c
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
data
    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
...
```

Range types and operators
Base Language Features, by example

```plaintext
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);
```

Tuples

```
<table>
<thead>
<tr>
<th>fib</th>
<th>#0 is 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#1 is 1</td>
</tr>
<tr>
<td></td>
<td>#2 is 1</td>
</tr>
<tr>
<td></td>
<td>#3 is 2</td>
</tr>
<tr>
<td></td>
<td>#4 is 3</td>
</tr>
<tr>
<td></td>
<td>#5 is 5</td>
</tr>
<tr>
<td></td>
<td>#6 is 8</td>
</tr>
</tbody>
</table>

...
Base Language Features, by example

```plaintext
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, briefly

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

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

Locales:

- Locale 0
- Locale 1
- Locale 2
- Locale 3

User’s main() executes on locale #0
Task Parallelism and Locality, by example

```chapel
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  printf("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

```chpl
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  printf("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

High-Level Task Parallelism

taskParallel.chpl

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

Abstraction of System Resources

```chpl
coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("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

```chpl
coforall loc in Locales do
    on loc {
        const numTasks = here.numPUs();
        coforall tid in 1..numTasks do
            printf("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

```chpl
taskParallel.chpl

coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("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

```chapl
coforall loc in Locales do
  on loc {
    const numTasks = here.numPUs();
    coforall tid in 1..numTasks do
      printf("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

- **atomic / synchronized variables**: for sharing data & coordination
- **begin / cobegin statements**: other ways of creating tasks
Data Parallelism in Chapel

Chapel language concepts
- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine

Higher-level 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)

```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
```
Arrays

Data Parallelism, by example

```
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

```
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

```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);
```

So far, this is a shared memory program
Nothing refers to remote locales, explicitly or implicitly
Distributed Data Parallelism, by example

```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);
```

Domain Maps
(Map Data Parallelism to the System)
Distributed Data Parallelism, by example

```
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**

![Diagram](dense, strided, sparse, associative)
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;

forall (_, r) in zip(Updates, RAStream()) do
    T[r & indexMask].xor(r);
Arkouda: NumPy over Chapel
Motivation: Say you’ve got…
   …a bunch of Python programmers
   …HPC-scale problems to solve
   …access to HPC systems
   How should you leverage these Python programmers to get your work done?

Concept: Develop Python libraries that are implemented in Chapel
   ⇒ get performance, as with Python-over-C, but also parallelism + scalability
   Even Better: use NumPy interfaces to make it trivial / familiar for users
Sample use from Jupyter

```
In [1]: import arkouda as ak

In [2]: ak.v = False
   ak.connect(server="localhost", port=5555)
   4.2.5
   http://localhost:5555

In [3]: ak.v = False
   N = 10**8 # 10**8 = 100M * 8 = 800MiB # 2**25 = 8 = 256MiB
   A = ak.arange(0,N,1)
   B = ak.arange(0,N,1)
   C = A+B
   print(ak.info(C),C)
   name:"id_3" dtype:"int64" size:100000000 ndim:1 shape:(100000000) itemsize:8
   [0 2 4 ... 199999994 199999996 199999998]

In [4]: S = (N*(N-1))/2
   print(2*S)
   print(ak.sum(C))
   9999999999999999.0
   9999999999999999

In [ ]:
```
Arkouda Accomplishments

By taking this approach, this user was able to:
• interact with a distributed, running Chapel program from Python within Jupyter
• run the same back-end program on...
  ...a Cray XC
  ...an HPE Superdome X
  ...an Infiniband cluster
  ...a Mac laptop
• compute on TB-sized arrays in seconds
• with ~1 month of effort
“Why Chapel?”

• High level — makes for less code
  • Great support for array operations and distributed arrays
  • Direct support for synchronized/atomic variables
  • Close to “Pythonic” (for a statically typed language)
    • Provides a gateway for data scientists ready to go beyond Python

• Portability and Scalability
  • Same code runs on shared- or distributed-memory systems
  • “From Raspberry Pi to Cray XC”

• Integrates with (distributed) numerical libraries (e.g., FFTW, FFTW-MPI)
Summary and Resources
Summary of this Talk

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 Python programmers
   • as a native language: similarly readable / writeable, yet scalable
   • as an implementation option for Python libraries
Chapel Central

https://chapel-lang.org

- downloads
- presentations
- papers
- resources
- documentation
Chapel Online Documentation

https://chapel-lang.org/docs: ~200 pages, including primer examples
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)
Chapel Social Media (no account required)

http://twitter.com/ChapelLanguage

http://facebook.com/ChapelLanguage

https://www.youtube.com/channel/UCHmm27bYjhknK5mU7ZzPGsQ/
Suggested Reading: Chapel history and 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
Chapel Comes of Age: Making Scalable Programming Productive

Bradford L. Chamberlain, Elliot Rouxen, Ben Alumbaugh, Lydia Duncan, Michael Ferguson, Ben Harmsbarger, David Ion, David Kasten, Vasuli Littirinen, Prasun Salhara, and Greg Titus

Chapel Team
Cray Inc.
Seattle, WA, USA
chapel-inf@cray.com

Abstract—Chapel is a programming language whose goal is to support productive, parallel computing at scale. Chapel’s approach can be thought of as combining the strengths of Fortran, Fortran-90, C++, and MPI in a single language. Five years ago, the DARPA High Productivity Computing Systems (HPCS) program launched Chapel project with the goal of developing a new programming system for high-performance computer science. This paper describes the evolution of Chapel and its progress toward the goal of making Chapel a useful tool for scientists. The paper reviews the challenges facing scientific computing, and how Chapel was designed to meet those challenges. The development of Chapel was led by Cray Inc. as part of its participation in the DARPA HPCS program. Chapel is an open-source language that is freely available to all users. Chapel is used in many scientific applications, including climate modeling, astrophysics, and high-energy physics. Chapel is also used in industry, with applications ranging from oil and gas exploration to autonomous vehicles. Chapel is a high-level language that supports both parallel and sequential programming. It is designed to be easy to learn and use, with a simple syntax and a rich set of features. Chapel is compiled to native code, which allows it to achieve high performance. Chapel is also designed to be portable, with support for a wide range of hardware platforms. Chapel is a general-purpose language that can be used for a wide range of applications, from scientific computing to data analysis. Chapel is a promising language for the future of high-performance computing.

The 6th Annual Chapel Implementers and Users Workshop

- sponsored by ACM SIGPLAN
- held in conjunction with PLDI 2019, FCRC 2019
- June 22-23, Phoenix AZ

**Keynote:** Anshu Dubey (Argonne)

*Programming Abstractions for Orchestration of HPC Scientific Computing*
Community Talks:

- Arkouda
- hybrid CPU-GPU computations with Chapel
- Stencil computations: getting performance for a student exercise
- Chapel Graph Library

Cray talks:

- Chapel’s use in AI / HPO
- interoperability improvements for Python, C, and Fortran
- recent performance optimizations
- radix sort in Chapel

Also: state of the project, lightning talks, coding day
Summary of this Talk

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 Python programmers
  • as a native language: similarly readable / writeable, yet scalable
  • as an implementation option for Python libraries
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-