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
- 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?

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

[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
  • Recent Uses of 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):
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):

![Diagram showing the computation of the STREAM Triad](image-url)
#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;
}
STREAM Triad: C + MPI + OpenMP

```c
#include <hpcc.h>
#include <omp.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;
}

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

STREAM Performance (GB/s)

GB/s

Reference MPI+OpenMP
Chapel 1.19

Locales (x 36 cores / locale)

© 2019 Cray, a Hewlett Packard Enterprise company
**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<MAXUPDATE; ++j) {
 *  Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
 *  else
 *  Ran = (Ran << 1) ^ ((s64Int) Ran < ZERO64B ? POLY : ZERO64B);
 *}
 */

if (recvUpdates > 0) {
  while (recvUpdates > 0) {
    if (have_done) {
      if (status.MPI_TAG == UPDATE_TAG) {
        MPI_Get_count(inmsg, &status, tparams.dtype64, &recvUpdates);
        bufferBase = 0;
        for (j=0; j < recvUpdates; j++) {
          inmsg = LocalRecvBuffer[bufferBase++];
          LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                        tparams.GlobalStartMyProc;
          HPCC_Table[LocalOffset] ^= inmsg;
        }
      } else if (status.MPI_TAG == FINISHED_TAG) {
        NumberReceiving--;
        MPI_Abort(MPI_COMM_WORLD, -1);
      }
    }
    while (have_done && NumberReceiving > 0) {
      if (pendingUpdates < maxPendingUpdates) {
        Ran = (Ran << 1) ^ ((64*count) Ran < ZEROS64B ? POLY : ZEROS64B);
        GlobalOffset = Ran & (tparams.TableSize-1);
        if (GlobalOffset < tparams.Top)
          WhichProc = (GlobalOffset / (tparams.MinLocalTableSize + 1));
        else
          WhichProc = (GlobalOffset - tparams.Top) / tparams.MinLocalTableSize;
        if (WhichProc == tparams.MyProc ||
            LocalOffset = (Ran & (tparams.TableSize - 1)) -
                        tparams.GlobalStartMyProc;
            HPCC_Table[LocalOffset] ^= Ran;
        }
      }
    }
  }
}

MPI_Test(outreq, &have_done, MPI_STATUS_IGNORE);
if (have_done) {
  outreq = MPI_REQUEST_NULL;
  proc_count = HPCC_GetUpdates(Buckets, LocalSendBuffer, localBufferSize, $ipUpdates);
  MPI_ Isend(&LocalSendBuffer, procUpdates, tparams.dtype64, (int)pe, $updated_TAG, MPI_COMM_WORLD, outreq);
  pendingUpdates += procUpdates;
}

/* send our done messages */
for (proc_count = 0; proc_count < tparams.procNums; ++proc_count) {
  if (proc_count == tparams.MyProc) {
    tparams.finish_req = MPI_REQUEST_NULL; continue;
  } /* send garbage - who cares, no one will look at it */
  MPI_Isend(4&Ran, 0, tparams.dtype64, proc_count, FINISHED_TAG, MPI_COMM_WORLD, tparams.finish_req + proc_count);
}

/* Finish everyone else up... */
while (NumberReceiving > 0) {
  if (status.MPI_TAG == UPDATE_TAG) {
    MPI_Get_count(inmsg, &status, tparams.dtype64, &recvUpdates);
    bufferBase = 0;
    for (j=0; j < recvUpdates; j++) {
      inmsg = LocalRecvBuffer[bufferBase++];
      LocalOffset = (inmsg & (tparams.TableSize - 1)) -
                    tparams.GlobalStartMyProc;
      HPCC_Table[LocalOffset] ^= inmsg;
    }
  } else if (status.MPI_TAG == FINISHED_TAG) {
    NumberReceiving--;
    MPI_Abort(MPI_COMM_WORLD, -1);
  }
  #if we got a done message. Thanks for playing... */
  #else
  if (status.MPI_TAG == FINISHED_TAG) {
    #endif
    NumberReceiving--;
    if (have_done) {
      if (status.MPI_TAG == UPDATE_TAG) {
        MPI_Get_count(inmsg, &status, tparams.dtype64, &recvUpdates);
        bufferBase = 0;
        for (j=0; j < recvUpdates; j++) {
          inmsg = LocalRecvBuffer[bufferBase++];
          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... */
    } else
      MPI_Abort(MPI_COMM_WORLD, -1);

  HPCC_Irecvv(LocalRecvBuffer, localBufferSize, tparams.dtype64, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, sinreq);
  while (have_done && NumberReceiving > 0);
/* Perform updates to main table. The scalar equivalent is: */

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

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

*;
  * for (i=0 ; i<NUPDATE; i++) {
    *  Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
    *  Table[Ran & (TABSIZE-1)] ^= Ran;
  * }
/* */
HPCC RA: Chapel vs. C+MPI

RA Performance (GUPS)

- Chapel 1.19
- MPI (bucketing)

Locales (x 36 cores / locale)

GUPS

0 2 4 6 8 10 12 14
16 32 64 128 256

better
forall (_, r) in zip(Updates(), RAStream()) do
T[r & indexMask].xor(r);
/* Perform updates to main table. The scalar equivalent is:
   for (r=0; r<NUMUPDATES; r++) {
   Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
   if (Ran & (tparams.TableSize - 1)) = Ran;
   }
   */

MPI_Irecv(LocalRecvBuffer, localBufferSize, tparams.dtype64, 
   MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, 
   while (i < SendCell) |
/* receive messages */
do |
   else {
   if (status.MPI_TAG == UPDATE_TAG) {
   MPI_Get_count(&recvUpdates, tparams.dtype64, &status);
   R = recvUpdates;
   proc_count = 0;
   for (j=0; j < recvUpdates; j++) {
   /* receive messages */
   inmsg = LocalRecvBuffer[j].inmsg & (tparams.TableSize - 1) - 
   tparams.GlobalStartMyProc;
   if (inmsg < tparams.MinLocalTableSize) |
/* send remaining updates in buckets */
do |
   MPI_Test(inreq, have_done, &status); |
   if (have_done) |
   else {
   if (status.MPI_TAG == FINISHED_TAG) |
   NumberReceiving--;|
   else |
   MPI_Abort(MPI_COMM_WORLD, -1); |
   } |
/* send our done messages */
do |
   else if (status.MPI_TAG == FINISHED_TAG) |
   NumberReceiving--;
   else |
   MPI_Abort(MPI_COMM_WORLD, -1); |
}while (have_done && NumberReceiving > 0); |
if (pendingUpdates < maxPendingUpdates) |
   Ran = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
   GlobalOffset = (Ran & (tparams.TableSize - 1)) - 
   tparams.GlobalStartMyProc;
   if (GlobalOffset < tparams.Top) |
   Whichr = (GlobalOffset / (tparams.MinLocalTableSize + 1)); |
   else |
   Whichr = (GlobalOffset / (tparams.Remainder / 
   tparams.MinLocalTableSize) + 1); |
   if (Whichr == (whichProc + tparams.Remainder) / 
   tparams.MinLocalTableSize) |
   (LocalOffset = (Ran & (tparams.TableSize - 1)) - 
   tparams.GlobalStartMyProc;
   MPI_T[r & indexMask].xor(r);
   } |.

forall (r, r in zip(Updates, RStream()) 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**: Chapel vs. Reference
  - Local loop kernels

- **HPCC RA**: Chapel vs. C+MPI
  - Global Random Updates

- **HPCC STREAM Triad**: Chapel vs. Reference
  - Embarrassing/Pleasing Parallelism

- **ISx**: Chapel vs. Reference
  - Bucket-Exchange Pattern

- **PRK Stencil**: Chapel vs. Reference
  - 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**

**HPCC RA**

**STREAM Triad**

**ISx**

**PRK Stencil**

**HCPC STREAM Triad: Chapel vs. Reference**

**ISx: Chapel vs. Reference**

**PRK Stencil: Chapel vs. Reference**

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

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

Lower-level Chapel
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

```
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);
```

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

0 1 1 2 3 5 8 ...

Iterators

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

```clu
cfg 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;
  }
}
```

```
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

```plaintext
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

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

```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 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;
  }
}

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
...
```

Zippered iteration
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
...
```
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;
    }
}
```

```javascript
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

Diagram:

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
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:

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

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

```chapl
const numTasks = here.numPUs();
coforall tid in 1..numTasks do
  printf("Hello from task %d of %d 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

```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

So far, this is a shared memory program
Nothing refers to remote locales, explicitly or implicitly

```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
      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 %d of %d 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
            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 n1033
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
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

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

```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)

```
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

```
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);
```

So far, this is a shared memory program
Nothing refers to remote locales, explicitly or implicitly

```
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

```
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)

```
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

```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**

![Diagram showing dense, strided, sparse, and associative arrays]
STREAM Triad and HPCC RA Kernel, revisited

```plaintext
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);
```
Recent Uses of Chapel
Recent and Notable Chapel Use Cases

Simulation of Ultralight Dark Matter
Nikhil Padmanabhan et al.
Yale University

Chapel Hypergraph Library (CHGL)
Louis Jenkins, Marcin Zalewski, et al.
PNNL

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

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

© 2019 Cray, a Hewlett Packard Enterprise company
Simulating Ultralight Dark Matter

Who: Nikhil Padmanabhan, et al., Yale University

What: numerically intensive simulations that make predictions for the phenomenology of Ultralight Dark Matter (ULDM) using pseudo-spectral methods based on FFTs

Why: reproduce the expressiveness of an existing Python version, yet with performance and scalability

For more information:
• see Nikhil’s upcoming presentation at the PAW-ATM 2019 workshop at SC19
Chapel CFD Computations

Who: Simon Bourgault-Côté, Matthieu Parenteau, …, École Polytechnique Montréal

What: a 3D unstructured Reynolds Average Navier-Stokes (RANS) flow solver for use in aircraft design research

Why Chapel:

• simplifies development of distributed memory code:
  • easy to write and read
  • much more compact and flexible than MPI
• results in performance equivalent to other distributed C++ flow solvers
• interfaces with external libraries fairly easily

Challenges: requires care to avoid performance pitfalls (e.g., unintentional or excessive remote accesses)
Chapel Hypergraph Library

Who: Louis Jenkins, Marcin Zalewski, et al., Pacific Northwest National Laboratory

What: a library supporting computations on HyperGraphs using a distributed, scalable representation

For more information:

- GitHub repository: [https://github.com/pnnl/chgl](https://github.com/pnnl/chgl)

Example Use:

Arkouda: NumPy at Massive Scale

Who: Mike Merrill, Bill Reus, et al., U.S. Department of Defense

What: a Python library supporting NumPy operations running over Chapel on large-scale systems

For more information, see:

- the following slides
- Bill’s CLSAC 2019 presentation from which they were excerpted
- Mike’s upcoming presentation at the PAW-ATM 2019 workshop at SC19
- the open source release, coming this month
(Data) Science is Interactive

“Hypothesis Testing”

Data Science is Interactive

Data

Summarize

Enrich

Inspect

Filter

Model

Transform

Output

presented at CLSAC 2019, October 9, 2019
Implications for Computing

• Stay in memory
• Compute in small, reversible steps
• Enable introspection (code and state)
• Use other people’s code
• Avoid boilerplate

So, basically Python...

...but fast

\[
\text{Maximize } \frac{t_{\text{thinking}}}{t_{\text{thinking}} + t_{\text{coding}} + t_{\text{waiting}}}
\]

presented at CLSAC 2019, October 9, 2019
Interactive Computational Ladder

- We need the upper tier
  - Cybersecurity data >> 6 TB
- But hardware is the easy part
  - Need serious data engineering
  - Need to rethink job scheduling
  - Need an HPC shell

presented at CLSAC 2019, October 9, 2019
An HPC Shell for Data Science

Load Terabytes of data...
... into a familiar, interactive UI ...
... where standard data science operations ...
... execute within the human thought loop ...
... and interoperate with optimized libraries.

presented at CLSAC 2019, October 9, 2019
Arkouda

Load Terabytes of data...
... into a familiar, interactive UI ...
... where standard data science operations ...
... execute within the human thought loop ...
... and interoperate with optimized libraries.

Arkouda: an HPC shell for data science
• Chapel backend (server)
• Jupyter/Python frontend (client)
• NumPy-like API

presented at CLSAC 2019, October 9, 2019
1) In terminal:

   > arkouda_server -nl 96

   server listening on hostname:port

2) In Jupyter:

   ```python
   import arkouda as ak
   ak.connect(hostname, port)
   ```

   4.2.5
   psp = tcp://nid00104:5555
   connected to tcp://nid00104:5555
Data Exploration with Arkouda and NumPy

In [9]:
A = ak.randint(0, 10, 10**11)
B = ak.randint(0, 10, 10**11)
C = A * B
hist = ak.histogram(C, 20)
Cmax = C.max()
Cmin = C.min()

In [10]:
bins = np.linspace(Cmin, Cmax, 20)
plt.bar(bins, hist.to_ndarray(), width=(Cmax-Cmin)/20)

presented at CLSAC 2019, October 9, 2019
Hypothesis Testing on 50 Billion Records

- A, B are 50 billion-element arrays
- Timings measured on real data
- Hardware: Cray XC40
  - 96 nodes
  - 3072 cores
  - 24 TB
  - Lustre filesystem

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Approximate Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read from disk</td>
<td>A = ak.read_hdf()</td>
<td>30-60</td>
</tr>
<tr>
<td>Scalar Reduction</td>
<td>A.sum()</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Histogram</td>
<td>ak.histogram(A)</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Vector Ops</td>
<td>A + B, A == B, A &amp; B</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Logical Indexing</td>
<td>A[A == val]</td>
<td>1 - 10</td>
</tr>
<tr>
<td>Set Membership</td>
<td>ak.in1d(A, set)</td>
<td>1</td>
</tr>
<tr>
<td>Gather</td>
<td>B = Table[A]</td>
<td>30 - 300</td>
</tr>
<tr>
<td>Group by Key</td>
<td>G = ak.GroupBy(A)</td>
<td>60</td>
</tr>
<tr>
<td>Aggregate per Key</td>
<td>G.aggregate(B, ‘sum’)</td>
<td>15</td>
</tr>
<tr>
<td>Get Item</td>
<td>print(A[42])</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Export to NumPy</td>
<td>A[:10**6].to_ndarray()</td>
<td>2</td>
</tr>
</tbody>
</table>

presented at CLSAC 2019, October 9, 2019
What about Model?

- **Vision**: Expose HPC libraries to Python via Arkouda
  - FFT
  - Tensor decomposition
  - Graph algorithms
  - Solvers
  - CHGL (Chapel HyperGraph Library from PNNL)
  - Anything you could link into a Chapel application (via C or LLVM)
- **Need to standardize a distributed array interface with the HPC community**

presented at CLSAC 2019, October 9, 2019
Arkouda Design

• Why Chapel?
  • High-level language with C-comparable performance
  • Parallelism is a first-class citizen
  • Great distributed array support
  • Portable code: from laptop up to supercomputer

presented at CLSAC 2019, October 9, 2019
Future Directions

• Open source release
• Tactical functionality
  • Strings and/or categorical dtype
  • Actual DataFrame class
  • Segmented arrays for sparse linear algebra (e.g. GraphBLAS)
• Strategic goals
  • Integration of Parallel Libraries
  • Multi-user support
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 scientists and 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

The Chapel Parallel Programming Language

What is Chapel?
Chapel is a modern programming language that is...

- parallel: contains first-class concepts for concurrent and parallel computation
- productive: designed with programmability and performance in mind
- portable: runs on laptops, clusters, the cloud, and HPC systems
- scalable: supports locality-oriented features for distributed memory systems
- open-source: hosted on GitHub, permissively licensed

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

- read a blog article or book chapter
- watch an overview talk or browse its slides
- download the release
- browse sample programs
- view other resources to learn how to trivially write distributed programs like this:

  ```
  use CyclicDist; // use the cyclic distribution library
  config const n = 100; // use --n=0 as default when executing to override this default
  forall l in [1..n] do
    writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
  ```

What's Hot?

- Chapel 1.17 is now available—download a copy or browse its release notes
- The advance program for CHIUW 2018 is now available—hope to see you there!
- Chapel is proud to be a Rails Girls Summer of Code 2018 organization
- Watch talks from ACCU 2017, CHIUW 2017, and ATPESC 2016 on YouTube
- Browse slides from SIAM PP18, NWCPP, SeaLang, SC17, and other recent talks
- Also see: What's New?
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)
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, Elliott Reinbacher, Ben Alworth, Lydia Duncan, Michael Ferguson, Ben Harmsen, David Irwin, David Kasting, Vassilis Lirivelis, Pratya Sahai, and Greg Titus

Chapel was a grammar-driven language designed to support productive, general-purpose parallel computing at scale. Chapel’s approach was thought of as creating a language whose code is as readable and easy to 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 an open-source community. Though developed at Cray, Chapel’s design and implementation are parallel, permitting it to support multiple programming paradigms and many different data structures in the Chapel system. In addition, Chapel programs can be run on a variety of platforms and HPC systems from other vendors. Chapel is being developed as an open-source project under the Apache 2.0 license and is hosted on GitHub.

The development of the Chapel language was undertaken by Cray Inc. as part of its participation in the DARPA High Productivity Computing Systems program (HiPS). HiPS wrapped up in late 2010, at which point Chapel was a competing prototype, having successfully demonstrated several key features. The HiPS experience has enabled Cray to make significant improvements to Chapel’s syntax and underlying compiler. Chief among these was supporting date 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, layouts, and tasks.

Under HiPS, Chapel also successfully supported the inclusion of SIMD instructions using distinct language features from those used to control locality and affinity. It is Chapel programmers specify which computations should run in parallel; Chapel determines from that specification where those computations should be run. This permits Chapel programs to support data-parallel, MPI, and heterogeneous computing within a single unified language.

Chapel’s implementation under HiPS 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 Adara™ networks. This allows Chapel to take advantage of native hardware support for remote pair, gather, and atomic memory operations.

Despite these successes, at the close of HiPS, Chapel was not at all ready to support production codes. To this end, the HiPS team worked with the language’s open-source community to develop a production-quality implementation. The team also worked with companies like Intel and Cray to help with transitioning the new language into their production environment.

This paper’s contribution is to describe the results of this two-year effort, providing updates with an understanding of Chapel’s progress and achievements since the end of the HiPS program. In doing so, we directly compare the state of Chapel version 1.1.3, released last month, with Chapel version 1.1.2, which was released five years ago in April 2010.

© 2019 Cray, a Hewlett Packard Enterprise Company
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 scientists and Python programmers
  • as a native language: similarly readable / writeable, yet scalable
  • as an implementation option for Python libraries
This presentation may contain forward-looking statements that involve risks, uncertainties and assumptions. If the risks or uncertainties ever materialize or the assumptions prove incorrect, the results of Hewlett Packard Enterprise Company and its consolidated subsidiaries (“Hewlett Packard Enterprise”) may differ materially from those expressed or implied by such forward-looking statements and assumptions. All statements other than statements of historical fact are statements that could be deemed forward-looking statements, including but not limited to any statements regarding the expected benefits and costs of the transaction contemplated by this presentation; the expected timing of the completion of the transaction; the ability of HPE, its subsidiaries and Cray to complete the transaction considering the various conditions to the transaction, some of which are outside the parties’ control, including those conditions related to regulatory approvals; projections of revenue, margins, expenses, net earnings, net earnings per share, cash flows, or other financial items; any statements concerning the expected development, performance, market share or competitive performance relating to products or services; any statements regarding current or future macroeconomic trends or events and the impact of those trends and events on Hewlett Packard Enterprise and its financial performance; any statements of expectation or belief; and any statements of assumptions underlying any of the foregoing. Risks, uncertainties and assumptions include the possibility that expected benefits of the transaction described in this presentation may not materialize as expected; that the transaction may not be timely completed, if at all; that, prior to the completion of the transaction, Cray’s business may not perform as expected due to transaction-related uncertainty or other factors; that the parties are unable to successfully implement integration strategies; the need to address the many challenges facing Hewlett Packard Enterprise’s businesses; the competitive pressures faced by Hewlett Packard Enterprise’s businesses; risks associated with executing Hewlett Packard Enterprise’s strategy; the impact of macroeconomic and geopolitical trends and events; the development and transition of new products and services and the enhancement of existing products and services to meet customer needs and respond to emerging technological trends; and other risks that are described in our Fiscal Year 2018 Annual Report on Form 10-K, and that are otherwise described or updated from time to time in Hewlett Packard Enterprise’s other filings with the Securities and Exchange Commission, including but not limited to our subsequent Quarterly Reports on Form 10-Q. Hewlett Packard Enterprise assumes no obligation and does not intend to update these forward-looking statements.
THANK YOU

QUESTIONS?

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

cray.com
@cray_inc
linkedin.com/company/cray-inc/