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:**

```
A = B + C · α
```
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

#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;
    }
}

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;
}
#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

```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 );
r
rv = HPCC_Stream( params, 0 == myRank );
MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm);
return errCount;

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

STREAM Performance (GB/s)

GB/s

Locales (x 36 cores / locale)

Reference MPI+OpenMP
Chapel 1.19

© 2020 Cray, a Hewlett Packard Enterprise company
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 (i = 0; i < UpdateCount; i++) {
   if (status.MPI_TAG == UPDATE_TAG) {
      MPI_Get_count(outreq, &status, tparams.dtype64, &recvUpdates);
      for (j = 0; j < recvUpdates; j++) {
         inmsg = LocalRecBuf[outmsg];
         inp = (int) tparams.TableSize - 1;
         HPCC_Table[LocalOffset] ^= inp;
      }
   }
   else {
      HPCC_InsertUpdate(Ran, WhichProc, Buckets);
      pendingUpdates++;
   }
}

HPCC_Test(outreq, have_done, status);
if (have_done) {
   if (status.MPI_TAG == UPDATE_TAG) {
      MPI_Get_count(outreq, &status, tparams.dtype64, &recvUpdates);
      bufferBase = 0;
      for (j = 0; j < recvUpdates; j++) {
         inmsg = LocalRecBuf[outmsg];
         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);
}
while (have_done || NumberReceiving > 0) {
   if (pendingUpdates > maxPendingUpdates) {
      Ran = (Ran << 1) ^ ((s64Int) Ran < 0) ? POLY : 0;
      GlobalOffset = (GlobalOffset + (int) tparams.TableSize - 1) - tparams.GlobalStartProc;
      HPCC_Table[LocalOffset] ^= Ran;
   }
   else {
      HPCC_InsertUpdate(Ran, WhichProc, Buckets);
      pendingUpdates++;
   }
   i++;
}
*/

HPCC_Irecv(LocalRecBuf, localBufferSize, tparams.dtype64, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
while (i < SendCnt) {
   if (have_done) {
      if (status.MPI_TAG == UPDATE_TAG) {
         MPI_Get_count(outreq, &status, tparams.dtype64, &recvUpdates);
         for (j = 0; j < recvUpdates; j++) {
            inmsg = LocalRecBuf[outmsg];
            inp = (int) tparams.TableSize - 1;
            HPCC_Table[LocalOffset] ^= inp;
         }
      } else if (status.MPI_TAG == FINISHED_TAG) {
         NumberReceiving--;
      } else
         MPI_Abort(MPI_COMM_WORLD, -1);
      HPCC_Irecv(LocalRecBuf, localBufferSize, tparams.dtype64, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &inreq);
   } while (have_done || NumberReceiving > 0);
Perform updates to main table. The scalar equivalent is:

```plaintext
for (i=0; i<NUPDATE; i++) {
    Ran = (Ran << 1) ^ ((s64Int) Ran < 0) ? POLY : 0;
    Table[Ran & (TPSIZET-1)] ^= Ran;
}
```

Chapel Kernel

```plaintext
forall (_, r) in zip(Updates, RAStream()) do
    T[r & indexMask].xor(r);
```
HPCC RA: Chapel vs. C+MPI

RA Performance (GUPS)

GUPS

Locales (x 36 cores / locale)

Chapel 1.19
MPI (bucketing)

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

```
while (i < SendCnt) {
    if (pe == peUpdates[recvUpdates]) {
        /* receive messages */
        for (i=0; i<NUPDATE; i++) {
            Table[Ran & (TABSIZE-1)] ^= Ran;
            GlobalOffset = (Ran << 1) ^ (((s64Int) Ran < 0) ? POLY : 0);
            LocalOffset = (GlobalOffset - proc_count) * tparams.TableSize;
            proc_count += 1;
            /* send remaining updates in buckets */
            HPCC_InsertUpdate(r, GlobalOffset, LocalOffset)
        }
        /* send garbage who cares, no one will look at it */
    } else if (status.MPI_TAG == FINISHED_TAG) {
        /* we got a done message. Thanks for playing... */
        NumberReceiving++; continue; }
    else if (status.MPI_TAG == UPDATE_TAG) {
        /* we got an update message. */
        inmsg = (int) bufferBase + i;
        inmsg = bufferBase + i;
        status.MPI_TAG = FINISHED_TAG;
        /* send garbage who cares, no one will look at it */
        proc_count += 1;
        /* send remaining updates in buckets */
        HPCC_InsertUpdate(r, bufferBase, inmsg)
    } else if (status.MPI_TAG == FINSIHED_TAG) {
        /* send garbage who cares, no one will look at it */
        NumberReceiving--; continue; }
    else if (status.MPI_TAG == SENDERS_TAG) {
        /* receive messages */
        MPI_Test(&inreq, &status);
        if (have_done) {
            if (status.MPI_TAG == UPDATE_TAG) {
                inmsg = (int) bufferBase + i;
                status.MPI_TAG = FINISHED_TAG;
                /* send garbage who cares, no one will look at it */
                proc_count += 1;
                /* send remaining updates in buckets */
                HPCC_InsertUpdate(r, bufferBase, inmsg)
            } else if (status.MPI_TAG == FINISHED_TAG) {
                /* send garbage who cares, no one will look at it */
                NumberReceiving--; continue; }
            else if (status.MPI_TAG == UPDATE_TAG) {
                /* we got an update message. */
                inmsg = (int) bufferBase + i;
                inmsg = bufferBase + i;
                status.MPI_TAG = FINISHED_TAG;
                /* send garbage who cares, no one will look at it */
                proc_count += 1;
                /* send remaining updates in buckets */
                HPCC_InsertUpdate(r, bufferBase, inmsg)
            } else if (status.MPI_TAG == FINSIHED_TAG) {
                /* send garbage who cares, no one will look at it */
                NumberReceiving--; continue; }
            else if (status.MPI_TAG == SENDERS_TAG) {
                /* receive messages */
                MPI_Test(&inreq, &status);
                if (have_done) {
                    if (status.MPI_TAG == UPDATE_TAG) {
                        inmsg = (int) bufferBase + i;
                        status.MPI_TAG = FINISHED_TAG;
                        /* send garbage who cares, no one will look at it */
                        proc_count += 1;
                        /* send remaining updates in buckets */
                        HPCC_InsertUpdate(r, bufferBase, inmsg)
                    } else if (status.MPI_TAG == FINISHED_TAG) {
                        /* send garbage who cares, no one will look at it */
                        NumberReceiving--; continue; }
                    else if (status.MPI_TAG == UPDATE_TAG) {
                        /* we got an update message. */
                        inmsg = (int) bufferBase + i;
                        inmsg = bufferBase + i;
                        status.MPI_TAG = FINISHED_TAG;
                        /* send garbage who cares, no one will look at it */
                        proc_count += 1;
                        /* send remaining updates in buckets */
                        HPCC_InsertUpdate(r, bufferBase, inmsg)
                    } else if (status.MPI_TAG == FINSIHED_TAG) {
                        /* send garbage who cares, no one will look at it */
                        NumberReceiving--; continue; }
                }
            }
        }
    }
}
```
HPCC RA: MPI vs. Chapel

```
/* Perform updates to main table. The scalar equivalent is:
 * for (i=0; i<UPDATE; i++) {
 *   /* receive messages */
 *   if (have_done) {
 *     if (status.MPI_TAG == UPDATE_TAG) {
 *       MPI_Get_count(status, tparams.dtype64, &recvUpdates);
 *       bufferBase = 0;
 *       for (j=0; j < recvUpdates; j++) {
 *         if (inmsg < (tparams.TableSize - 1)) {
 *           tparams.GlobalStartMyProc =_mpi_token;  // MPITokenProc;
 *           HPCC_Table[LocalOffset + inmsg] -= inmsg;
 *         } else if (status.MPI_TAG == FINISHED_TAG) {
 *           NumberReceiving--;
 *         } else {
 *           MPI_Abort(MPI_COMM_WORLD, -1);
 *         }
 *       } while (have_done && NumberReceiving > 0);
 *   }
 *   pendingUpdates -= pendingUpdates;
 *   if (have_done) {
 *     if (status.MPI_TAG == UPDATE_TAG) {
 *       MPI_Get_count(status, tparams.dtype64, &recvUpdates);
 *       bufferBase = 0;
 *       for (j=0; j < recvUpdates; j++) {
 *         if (inmsg < (tparams.TableSize - 1)) {
 *           tparams.GlobalStartMyProc = mpi_token;  // MPITokenProc;
 *           HPCC_Table[LocalOffset + inmsg] ^= inmsg;
 *         } else if (status.MPI_TAG == FINISHED_TAG) {
 *           NumberReceiving--;
 *         } else {
 *           MPI_Abort(MPI_COMM_WORLD, -1);
 *         }
 *       } while (have_done && NumberReceiving > 0);
 *     }
 *     else if (status.MPI_TAG == RAStream_TAG) {
 *       HPCC_InsertUpdate(RAStream, RAStream, RACount, RAStream);
 *     }
 *     else {
 *       HPCC_InsertUpdate(RAStream, RAStream, RAStream);
 *     }
 *   }
 * }

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

Chapel Kernel
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

**NAS FT**
- Global Transposes

**HPCC RA**
- Global Random Updates

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

**ISx**
- Bucket-Exchange Pattern

**PRK Stencil**
- Stencil Boundary Exchanges
HPC Patterns: Chapel vs. Reference

**LCALS**
- Local loop kernels

**NAS FT**
- Global Transposes
- NAS FT: Chapel vs. UPC vs. MPI
- NPB-FT (Size E) Performance

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

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

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

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

© 2020 Cray, a Hewlett Packard Enterprise company
HPC Patterns: Chapel vs. Reference

LCALS: Chapel vs. Reference

NAS FT: Chapel vs. UPC vs. MPI

HPCC RA: Chapel vs. C+MPI

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

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

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

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

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

```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
...

© 2020 Cray, a Hewlett Packard Enterprise company
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);
```

Configurable declarations (support command-line overrides)

```
./fib --n=1000000
```

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

Iterators

```plaintext
defib(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;
  }
}
```

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

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

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

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

Explicit types also supported

```
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

```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

```python
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

```pseudocode
iter fib(n) {
    var current = 0,
        next = 1;
    for i in 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}
```
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 (i, f) in zip(0..<n, fib(n)) do
    writeln("fib ", i, " is ", f);
```

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

```
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

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

Locales:

<table>
<thead>
<tr>
<th>Locale</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>User's main() executes on locale #0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2020 Cray, a Hewlett Packard Enterprise company
Task Parallelism and Locality, by example

```
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

Abstraction of System Resources

```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
```
### High-Level Task Parallelism

```chpl
class ParallelMain {
    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
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
    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

```
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
```
Abstraction of System Resources

Task Parallelism and Locality, by example

```chpl
coforall loc in Locales do
    on loc {
        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 --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

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

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

© 2020 Cray, a Hewlett Packard Enterprise company
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 %d of %d + ",
             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

Chapel language concepts

<table>
<thead>
<tr>
<th>Domain Maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Parallelism</td>
</tr>
<tr>
<td>Task Parallelism</td>
</tr>
<tr>
<td>Base Language</td>
</tr>
<tr>
<td>Locality Control</td>
</tr>
<tr>
<td>Target Machine</td>
</tr>
</tbody>
</table>

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
```
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

```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

```
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);
```

```bash
chpl dataParallel.chpl
./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

```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

```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
```

<table>
<thead>
<tr>
<th>1.1</th>
<th>1.3</th>
<th>1.5</th>
<th>1.7</th>
<th>1.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>2.3</td>
<td>2.5</td>
<td>2.7</td>
<td>2.9</td>
</tr>
<tr>
<td>3.1</td>
<td>3.3</td>
<td>3.5</td>
<td>3.7</td>
<td>3.9</td>
</tr>
<tr>
<td>4.1</td>
<td>4.3</td>
<td>4.5</td>
<td>4.7</td>
<td>4.9</td>
</tr>
<tr>
<td>5.1</td>
<td>5.3</td>
<td>5.5</td>
<td>5.7</td>
<td>5.9</td>
</tr>
</tbody>
</table>
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);
```
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

The Chapel Parallel Programming Language

What is Chapel?
Chapel is a programming language designed for productive parallel computing at scale.

Why Chapel? Because it simplifies parallel programming through elegant support for:
- distributed arrays that can leverage thousands of nodes' memories and cores
- a global namespace supporting direct access to local or remote variables
- data parallelism to trivially use the cores of a laptop, cluster, or supercomputer
- task parallelism to create concurrency within a node or across the system

Chapel Characteristics
- productive: code tends to be similarly readable/writable as Python
- scalable: runs on laptops, clusters, the cloud, and HPC systems
- fast: performance competes with or beats C/C++ & MPI & OpenMP
- portable: compiles and runs in virtually any *nix environment
- open-source: hosted on GitHub, permissively licensed

New to Chapel?
As an introduction to Chapel, you may want to...
- watch an overview talk or browse its slides
- read a blog-length or chapter-length introduction to Chapel
- learn about projects powered by Chapel
- check out performance highlights like these:

- browse sample programs or learn how to write distributed programs like this one:

```c
use CyclicDist; // use the Cyclic distribution library
cfconf count n = 100; // use --reconf when exercising to override this default
forall i in (1..n) do
depends cyclic(startId=x) do
  writeln("Hello from iteration ", i, " of ", n, " running on node ", here.id);
```
Chapel Documentation

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](#)
Chapel Comes of Age: Making Scalable Programming Productive


Cray Inc.
Seattle, WA, USA
chapel_info@cray.com

Abstract—Chapel is a programming language whose goal is to support productive, parallel computing at scale. Chapel’s approach can be traced to the strength of Chapel since its first version, FSCM, and more recently in Chapel 2.0, a high-level, divide-and-conquer approach to the language. This paper builds on this approach to improve Chapel’s efficiency and productivity. This paper follows up on our CVG 2013 paper by examining the progress made by the Chapel project since that time. Specifically, Chapel’s performance in computer science and hardware benchmarks has been quite impressive, and the language has been extended to include PSTY, PARX, LAMARC, MPI, OpenMP, and other high-level features that have been maintained and tested as the set of tools available to Chapel users has grown. This paper also describes the adoption of Chapel’s programming model in a diverse array of academic and commercial environments.

Keywords: Parallel programming, Computer language

1. INTRODUCTION

Chapel is a parallel programming language designed to support productive, general-purpose parallel computing at scale. Chapel’s approach can be traced to the strength of Chapel since its first version, FSCM, and more recently in Chapel 2.0, a high-level, divide-and-conquer approach to the language. This paper builds on this approach to improve Chapel’s efficiency and productivity. This paper follows up on our CVG 2013 paper by examining the progress made by the Chapel project since that time. Specifically, Chapel’s performance in computer science and hardware benchmarks has been quite impressive, and the language has been extended to include PSTY, PARX, LAMARC, MPI, OpenMP, and other high-level features that have been maintained and tested as the set of tools available to Chapel users has grown. This paper also describes the adoption of Chapel’s programming model in a diverse array of academic and commercial environments.

2. CHapel Comes of Age: Productive Parallelism at Scale

CUG 2018
Brad Chamberlain, Chapel Team, Cray Inc.

The development of the Chapel language was undertaken by Cray Inc. as part of its participation in the DARPA High Productivity Computing Systems programs (HPCS). HPCS was a multi-year program that included Chapel and other languages for high-performance computing. The goal of Chapel was to create a language that would support parallel computation at scale. The language was designed to be easy to use, with a simple, intuitive syntax that made it easy to write parallel programs. The language was also designed to be efficient, with a low overhead for parallel execution.

The language has been extended over time, with new features added to support emerging technologies and to improve performance. The language is now used in a variety of applications, from scientific simulation to machine learning and data analytics.

3. Available Resources

Paper and slides available at chapel-lang.org

© 2020 Cray, a Hewlett Packard Enterprise company
Suggested Reading: The Very Latest

• Chapel release notes:  [https://chapel-lang.org/releaseNotes.html](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-/