Chapel: The Design and Implementation of a Multiresolution Language

Brad Chamberlain, Chapel Team, Cray Inc.
Keynotes on HPC Languages, Lyon, June 30th, 2013
Slides presented by Kathy Yelick in CS294, Fall 2013 with permission
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors
   • Static finite element analysis

1 TF – 1998: Cray T3E; 1,024 Processors
   • Modeling of metallic magnet atoms

1 PF – 2008: Cray XT5; 150,000 Processors
   • Superconductive materials

1 EF – ~2018: Cray ____; ~10,000,000 Processors
   • TBD
Sustained Performance Milestones

1 GF – 1988: Cray Y-MP; 8 Processors
• Static finite element analysis
• Fortran77 + Cray autotasking + vectorization

1 TF – 1998: Cray T3E; 1,024 Processors
• Modeling of metallic magnet atoms
• Fortran + MPI (Message Passing Interface)

1 PF – 2008: Cray XT5; 150,000 Processors
• Superconductive materials
• C++/Fortran + MPI + vectorization

1 EF – ~2018: Cray ____; ~10,000,000 Processors
• TBD
• TBD: C/C++/Fortran + MPI + CUDA/OpenCL/OpenMP/OpenACC?
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
   \alpha
```
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:
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):

```
A  =  =  =  =  =  =  =  =  =
B  +  +  +  +  +  +  +  +
C  .  .  .  .  .  .  .  .
\alpha . . . . . . . . .
```
#include <hpcc.h>

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    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] = 0.0;
    }

    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);
STREAM Triad: MPI+OpenMP

```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

static int VectorSize;
static double *a, *b, *c;

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

    int rv, errCount;
    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) {  //pragma omp parallel for
    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] = 0.0;
    }
    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);
```

STREAM Triad: MPI+OpenMP vs. CUDA

HPC suffers from too many distinct notations for expressing parallelism and locality

CUDA

#include <hpcc.h>
#define N 2000000

int main() {
    float *d_a, *d_b, *d_c;
    float scalar;
    cudaMemcpy((void**)d_a, sizeof(float)*N);
    cudaMemcpy((void**)d_b, sizeof(float)*N);
    cudaMemcpy((void**)d_c, sizeof(float)*N);

    dim3 dimBlock(128);
    dim3 dimGrid(N/dimBlock.x);
    if (N % dimBlock.x != 0) dimGrid.x+=1;
    set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
    set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);
    scalar=3.0f;
    STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar, N);
    cudaMemcpy((void**)d_a, sizeof(float)*N);
    cudaMemcpy((void**)d_b, sizeof(float)*N);
    cudaMemcpy((void**)d_c, sizeof(float)*N);

    return 0;
}

__global__ void set_array(float *a, float value, int len) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < len) a[idx] = value;
}

__global__ void STREAM_Triad(float *a, float *b, float *c, float scalar, int len) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
**STREAM Triad: Chapel**

```c
#define N       2000000
int main() {
  float *d_a, *d_b, *d_c;
  float scalar;
  cudaMalloc((void**)&d_a, sizeof(float)*N);
  cudaMalloc((void**)&d_b, sizeof(float)*N);
  cudaMalloc((void**)&d_c, sizeof(float)*N);
  dim3 dimBlock(128);
  dim3 dimGrid(N/dimBlock.x );
  if( N % dimBlock.x != 0 ) dimGrid.x+=1;
  set_array<<<dimGrid,dimBlock>>>(d_b, .5f, N);
  set_array<<<dimGrid,dimBlock>>>(d_c, .5f, N);
  scalar=3.0f;
  STREAM_Triad<<<dimGrid,dimBlock>>>(d_b, d_c, d_a, scalar,  N);
  cudaThreadSynchronize();
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
}
```

```c
__global__ void set_array(float *a,  float value, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) a[idx] = value;
}
```

```c
__global__ void STREAM_Triad( float *a, float *b, float *c, 
  float scalar, int len) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if (idx < len) c[idx] = a[idx]+scalar*b[idx];
}
```

---

**Philosophy:** Good language design can tease details of locality and parallelism away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.
Motivation

Chapel Background and Themes

- Tour of Chapel Concepts and Implementation
- Project Status and Next Steps
What is Chapel?

● An emerging parallel programming language
  ● Design and development led by Cray Inc.
    ● in collaboration with academia, labs, industry
    ● Initiated under the DARPA HPCS program

● Overall goal: Improve programmer productivity
  ● Improve the programmability of parallel computers
  ● Match or beat the performance of current programming models
  ● Support better portability than current programming models
  ● Improve the robustness of parallel codes

● A work-in-progress
1) General Parallel Programming

With a unified set of concepts...

...express any parallelism desired in a user’s program
  ● **Styles**: data-parallel, task-parallel, concurrency, nested, …
  ● **Levels**: model, function, loop, statement, expression

...target any parallelism available in the hardware
  ● **Types**: machines, nodes, cores, instructions

<table>
<thead>
<tr>
<th>Type of HW Parallelism</th>
<th>Programming Model</th>
<th>Unit of Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-node</td>
<td>Chapel</td>
<td>executable/task</td>
</tr>
<tr>
<td>Intra-node/m multicore</td>
<td>Chapel</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level vectors/threads</td>
<td>Chapel</td>
<td>iteration</td>
</tr>
<tr>
<td>GPU/accelerator</td>
<td>Chapel</td>
<td>SIMD function/task</td>
</tr>
</tbody>
</table>
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”

Global-View

\[
\begin{align*}
&\begin{array}{c}
( & \ldots & \ldots & \ldots \\
+ & \ldots & \ldots & \ldots \\
= & \ldots & \ldots & \ldots \\
\end{array} \\
& \text{Local-View}
\end{align*}
\]
2) Global-View Abstractions

In pictures: “Apply a 3-Point Stencil to a vector”
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

Global-View

```chapel
proc main() {
    var n = 1000;
    var A, B: [1..n] real;
    forall i in 2..n-1 do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Local-View (SPMD)

```chapel
proc main() {
    var n = 1000;
    var p = numProcs(),
        me = myProc(),
        myN = n/p,
    var A, B: [0..myN+1] real;
    if (me < p-1) {
        send(me+1, A[myN]);
        recv(me+1, A[myN+1]);
    }
    if (me > 0) {
        send(me-1, A[1]);
        recv(me-1, A[0]);
    }
    forall i in 1..myN do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Bug: Refers to uninitialized values at ends of A
2) Global-View Abstractions

In code: “Apply a 3-Point Stencil to a vector”

Global-View

```
proc main() {
    var n = 1000;
    var A, B: [1..n] real;
    forall i in 2..n-1 do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Local-View (SPMD)

```
proc main() {
    var n = 1000;
    var p = numProcs(),
        me = myProc(),
        myN = n/p,
        myLo = 1,
        myHi = myN;
    var A, B: [0..myN+1] real;

    if (me < p-1) {
        send(me+1, A[myN]);
        recv(me+1, A[myN+1]);
    } else
        myHi = myN-1;

    if (me > 0) {
        send(me-1, A[1]);
        recv(me-1, A[0]);
    } else
        myLo = 2;

    forall i in myLo..myHi do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

Communication becomes geometrically more complex for higher-dimensional arrays

Assumes p divides n
2) Global-View Programming: A Final Note

- A language may support both global- and local-view programming — in particular, Chapel does.

```chapel
proc main() {
    coforall loc in Locales do
        on loc do
            MySPMDProgram(loc.id, Locales.numElements);
    }

proc MySPMDProgram(myImageID, numImages) {
    ...
}
```
3) Multiresolution Design: Motivation

“Why is everything so tedious/difficult?”
“Why don’t my programs port trivially?”

“Why don’t I have more control?”

Low-Level Implementation Concepts
- MPI
- OpenMP
- Pthreads

High-Level Abstractions
- HPF
- ZPL
3) Multiresolution Design

**Multiresolution Design:** Support multiple tiers of features
- higher levels for programmability, productivity
- lower levels for greater degrees of control

*Chapel language concepts*

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

- build the higher-level concepts in terms of the lower
- permit the user to intermix layers arbitrarily
4) Control over Locality/Affinity

Consider:
- Scalable architectures package memory near processors
- Remote accesses take longer than local accesses

Therefore:
- Placement of data relative to tasks affects scalability
- Give programmers control of data and task placement

Note:
- Over time, we expect locality to matter more and more within the compute node as well
Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

abstract concept:

- support a shared namespace on distributed memory
- permit any parallel task to access any lexically visible variable
- doesn’t matter if it’s local or remote
Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

abstract concept:

- support a shared namespace on distributed memory
  - permit any parallel task to access any lexically visible variable
  - doesn’t matter if it’s local or remote
- establish a strong sense of ownership
  - every variable has a well-defined location
  - local variables are cheaper to access than remote ones
### PGAS: What’s in a Name?

<table>
<thead>
<tr>
<th>PGAS Languages</th>
<th>memory model</th>
<th>programming model</th>
<th>execution model</th>
<th>data structures</th>
<th>communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>distributed memory</td>
<td>cooperating executables (often SPMD in practice)</td>
<td>manually fragmented</td>
<td>APIs</td>
<td></td>
</tr>
<tr>
<td>OpenMP</td>
<td>shared memory</td>
<td>global-view parallelism</td>
<td>shared memory multithreaded</td>
<td>shared memory arrays</td>
<td>N/A</td>
</tr>
<tr>
<td>CAF</td>
<td>PGAS</td>
<td>Single Program, Multiple Data (SPMD)</td>
<td>co-arrays</td>
<td>co-array refs</td>
<td></td>
</tr>
<tr>
<td>UPC</td>
<td>PGAS</td>
<td>global-view parallelism</td>
<td>distributed memory multithreaded</td>
<td>global-view distributed arrays</td>
<td>implicit</td>
</tr>
<tr>
<td>Titanium</td>
<td>PGAS</td>
<td>global-view parallelism</td>
<td>distributed memory multithreaded</td>
<td>global-view distributed arrays</td>
<td>implicit</td>
</tr>
</tbody>
</table>

**1D block-cyc arrays**/distributed pointers **method-based**
5) Reduce HPC ↔ Mainstream Language Gap

Consider:
- Students graduate with training in Java, Matlab, Perl, Python
- Yet HPC programming is dominated by Fortran, C/C++, MPI

We’d like to narrow this gulf with Chapel:
- to leverage advances in modern language design
- to better utilize the skills of the entry-level workforce...
- ...while not alienating the traditional HPC programmer
  - e.g., support object-oriented programming, but make it optional
Outline

- Motivation
- Chapel Background and Themes
- Tour of Chapel Concepts and Implementation

- Project Status and Next Steps
const pi = 3.14, // pi is a real
coord = 1.2 + 3.4i, // coord is a complex...
coord2 = pi*coord, // ...as is coord2
name = "brad", // name is a string
verbose = false; // verbose is boolean

proc addem(x, y) { // addem() has generic arguments
  return x + y; // and an inferred return type
}

var sum = addem(1, pi), // sum is a real
    fullname = addem(name, "ford"); // fullname is a string

writeln((sum, fullname));

(4.14, bradford)
Range Types and Algebra

```plaintext
const r = 1..10;
printVals(r # 3);
printVals(r # -3);
printVals(r by 2);
printVals(r by -2);
printVals(r by 2 # 3);
printVals(r # 3 by 2);
printVals(0.. # n);

proc printVals(r) {
  for i in r do
    write(r, " ");
    writeln();
}
```

1 2 3
8 9 10
1 3 5 7 9
10 8 6 4 2
1 3 5
1 3
0 1 2 3 4 … n-1
Iterators

iter fibonacci(n) {
    var current = 0,
        next = 1;
    for 1..n {
        yield current;
        current += next;
        current <=> next;
    }
}

iter tiledRMO(D, tileSize) {
    const tile = {0..#tileSize, 0..#tileSize};
    for base in D by tileSize do
        for ij in D[tile + base] do
            yield ij;
}

for f in fibonacci(7) do writeln(f);
0
1
1
2
3
5
8

for ij in tiledRMO({1..m, 1..n}, 2) do
    write(ij);
(1, 1) (1, 2) (2, 1) (2, 2)
(1, 3) (1, 4) (2, 3) (2, 4)
(1, 5) (1, 6) (2, 5) (2, 6)
...
(3, 1) (3, 2) (4, 1) (4, 2)
Zippered Iteration

```chapel
for (i, f) in zip(0..#n, fibonacci(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

- tuple types and values
- rank-independent programming features
- interoperability features
- compile-time features for meta-programming
  - e.g., compile-time functions to compute types, parameters
- OOP (value- and reference-based)
- argument intents, default values, match-by-name
- overloading, where clauses
- modules (for namespace management)
- …
Outline

✔ Motivation
✔ Chapel Background and Themes
➢ Tour of Chapel Concepts and Implementation

● Project Status and Next Steps
Task Parallelism: Begin Statements

// create a fire-and-forget task for a statement
begin writeln("hello world");
writeln("good bye");

Possible outputs:

- hello world
- good bye
- good bye
- hello world
Task Parallelism: Cobegin Statements

```c
// create a task per child statement
cobegin {
    producer(1);
    producer(2);
    consumer(1);
} // implicit join of the three tasks here
```
Task Parallelism: Coforall Loops

```plaintext
// create a task per iteration
coforall t in 0..#numTasks {
    writeln("Hello from task ", t, " of ", numTasks);
} // implicit join of the numTasks tasks here

writeln("All tasks done");
```

Sample output:

```
Hello from task 2 of 4
Hello from task 0 of 4
Hello from task 3 of 4
Hello from task 1 of 4
All tasks done
```
Task Parallelism: Data-Driven Synchronization

1) **atomic variables**: support atomic operations (as in C++)
   - e.g., compare-and-swap; atomic sum, mult, etc.

2) **single-assignment variables**: reads block until assigned

3) **synchronization variables**: store full/empty state
   - by default, reads/writes block until the state is full/empty
Bounded Buffer Producer/Consumer Example

```chapel
cobegin {
    producer();
    consumer();
}

// 'sync' types store full/empty state along with value
var buff$: [0..#buffersize] sync real;

proc producer() {
    var i = 0;
    for ... {
        i = (i+1) % buffersize;
        buff$[i] = ...;  // writes block until empty, leave full
    }
}

proc consumer() {
    var i = 0;
    while ... {
        i = (i+1) % buffersize;
        ...buff$[i]...;  // reads block until full, leave empty
    }
}
```
Chapel Compiler Architecture

- Chapel Source Code
  - Chapel-to-C Compiler
    - Generated C Code
      - Standard C Compiler & Linker
        - Chapel Executable
          - Chapel Compiler
  - Standard Modules (in Chapel)
  - Internal Modules (in Chapel)
    - Runtime Support Library (in C)
      - Tasks/Threads
      - Communication
      - Memory
        - ...

Chapel-to-C Compiler

Chapel Compiler

Generated C Code

Standard C Compiler & Linker

Chapel Executable
Outline

✓ Motivation
✓ Chapel Background and Themes
➢ Tour of Chapel Concepts and Implementation

- Project Status and Next Steps
The Locale Type

Definition:
- Abstract unit of target architecture
- Supports reasoning about locality
- Capable of running tasks and storing variables
  - i.e., has processors and memory

Typically: A compute node (multicore processor or SMP)
Defining Locales

● Specify # of locales when running Chapel programs

```
% a.out --numLocales=8  % a.out -nl 8
```

● Chapel provides built-in locale variables

```
config const numLocales: int = ...;
const Locales: [0..#numLocales] locale = ...;
```

```
Locales  L0 L1 L2 L3 L4 L5 L6 L7
```

● User’s `main()` begins executing on locale #0
Locale Operations

- Locale methods support queries about the target system:

```chapel
proc locale.physicalMemory(...) { ... }
proc locale.numCores { ... }
proc locale.id { ... }
proc locale.name { ... }
```

- **On-clauses** support placement of computations:

```chapel
writeln("on locale 0");

on Locales[1] do
  writeln("now on locale 1");
writeln("on locale 0 again");
```

```chapel
cobegin {
  on A[i,j] do
    bigComputation(A);

  on node.left do
    search(node.left);
}
```
Chapel and PGAS

- Chapel is PGAS, but unlike UPC/CAF, it’s not SPMD
  - never think about “the other copies of the program”
  - “global name-/address space” comes from lexical scoping
    - rather than: “We’re all running the same program, so we must all have a variable named $x$”
    - as in traditional languages, each declaration yields one variable
    - stored on locale where task executes, not everywhere/thread 0
Chapel and PGAS

```chapel
var i: int;
```
Chapel and PGAS

```chapel
var i: int;
on Locales[1] {
```
Chapel and PGAS

```chapel
var i: int;
on Locales[1] {
  var j: int;
}
```
Chapel and PGAS

```chapel
var i: int;
on Locales[1] {
    var j: int;
    coforall loc in Locales {
        on loc {
```

Chapel and PGAS

```chapel
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k: int;
    }
  }
}
```
Chapel and PGAS: Public vs. Private

How public a variable is depends only on scoping
- who can see it?
- who actually bothers to refer to it non-locally?

```chapel
var i: int;
on Locales[1] {
  var j: int;
  coforall loc in Locales {
    on loc {
      var k = i + j;
    }
  }
}
```
Runtime Communication Layer

Chapel Runtime Support Library (in C)

Communication

Communication layer interface:
- single-sided communication (gets/puts)
  - for remote reads/writes
- remote forks (active messages)
  - for on-clauses
  - blocking, non-blocking, and “fast”
- optionally, remote atomic memory ops (AMOs)
Runtime Communication Layer

Chapel Runtime Support Library (in C)

Communication

none (single locale)  gasnet  ugni

GASNet (universal)  Cray uGNI (Cray networks)
Custom Runtime Impacts on Random Access

Summary: Chapel’s custom runtime demonstrates how a portable, high-level language can take advantage of architecture-specific productivity features like Cascade’s...
Motivation

Chapel Background and Themes

Tour of Chapel Concepts and Implementation

Project Status and Next Steps
Chapel supports several types of domains (index sets):

- **Dense**
- **Strided**
- **Sparse**

**Associative**

- “steve”
- “lee”
- “sung”
- “david”
- “jacob”
- “albert”
- “brad”

**Unstructured**
Chapel Array Types

- **dense**
- **strided**
- **sparse**
- **associative**
- **unstructured**
Chapel Domain/Array Operations

- Data Parallel Iteration (as well as serial and coforall)
  
  \[
  A = \texttt{forall} (i,j) \texttt{ in } D \texttt{ do } (i + j/10.0);
  \]

- Array Slicing; Domain Algebra
  
  \[
  A[\text{InnerD}] = B[\text{InnerD}+(0,1)];
  \]

- Promotion of Scalar Operators and Functions
  
  \[
  A = B + \alpha \times C;
  \]

- And several others: indexing, reallocation, set operations, remapping, aliasing, queries, …
Notes on Forall Loops

```chapel
forall a in A do
  writeln("Here is an element of A: ", a);
```

Typically $1 \leq \#\text{Tasks} \ll \#\text{Iterations}$

```chapel
forall (a, i) in zip(A, 1..n) do
  a = i/10.0;
```

Forall-loops may be zippered, like for-loops

- Corresponding iterations will match up
Promotion Semantics

Promoted functions/operators are defined in terms of zippered forall-loops in Chapel. For example…

```
A = B;
```

…is equivalent to:

```
forall (a, b) in zip(A, B) do
  a = b;
```
Benefits of Zippered Promotion Semantics

Whole-array operations are implemented element-wise...

\[
A = B + \alpha \cdot C; \quad \Rightarrow \quad \text{forall } (a,b,c) \text{ in } (A,B,C) \text{ do } a = b + \alpha \cdot c;
\]

...rather than operator-wise.

\[
A = B + \alpha \cdot C; \quad \times \quad T_1 = \alpha \cdot C; \quad A = B + T_1;
\]

\[\Rightarrow \text{No temporary arrays required by semantics}\]
\[\Rightarrow \text{No surprises in memory requirements}\]
\[\Rightarrow \text{Friendlier to cache utilization}\]
Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?
- Are regular arrays laid out in row- or column-major order? Or…?
- How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

Q2: How are arrays stored by the locales?
- Completely local to one locale? Or distributed?
- If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?
Data Parallelism Implementation Qs

Q1: How are arrays laid out in memory?
   ● Are regular arrays laid out in row- or column-major order? Or…?

   ● How are sparse arrays stored? (COO, CSR, CSC, block-structured, …?)

Q2: How are arrays stored by the locales?
   ● Completely local to one locale? Or distributed?

   ● If distributed… In a blocked manner? cyclically? block-cyclically? recursively bisected? dynamically rebalanced? …?

A: Chapel’s domain maps are designed to give the user full control over such decisions
Outline

✓ Motivation
✓ Chapel Background and Themes
➢ Tour of Chapel Concepts and Implementation

● Project Status and Next Steps
Domain Maps

Domain maps are “recipes” that instruct the compiler how to map the global view of a computation...

\[ A = B + \alpha \cdot C; \]

...to the target locales’ memory and processors:
STREAM Triad: Chapel (multicore)

\[
\text{const ProblemSpace} = \{1..m\};
\]

\[
\text{var A, B, C: [ProblemSpace] real;}
\]

\[
A = B + \alpha \cdot \text{alpha} \times C;
\]
STREAM Triad: Chapel (multicore)

```chapel
const ProblemSpace = {1..m};

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

A = B + alpha * C;
```

No domain map specified => use default layout
- current locale owns all indices and values
- computation will execute using local processors only
STREAM Triad: Chapel (multilocale, blocked)

\[
\text{const ProblemSpace} = \{1..m\}
\]

\[
\text{dmapped Block}\left(\text{boundingBox} = \{1..m\}\right);
\]

\[
\text{var A, B, C: [ProblemSpace] real};
\]

\[
A = B + \alpha \cdot C;
\]
STREAM Triad: Chapel (multilocale, cyclic)

\[
\text{const ProblemSpace} = \{1..m\} \\
\text{dmapped Cyclic(startIdx=1)};
\]

\[
\text{var A, B, C: [ProblemSpace] real;}
\]

\[
A = B + \alpha \cdot C;
\]
Sample Distributions: Block and Cyclic

```
var Dom = {1..4, 1..8} dmapped Block( {1..4, 1..8} );
```

```
var Dom = {1..4, 1..8} dmapped Cyclic( startIdx=(1,1) );
```
Domain Map Types

- **dense**
- **strided**
- **sparse**
- **associative**
- **unstructured**

Examples:
- "steve"
- "lee"
- "sung"
- "david"
- "jacob"
- "albert"
- "brad"
Chapel’s Domain Map Philosophy

1. Chapel provides a library of standard domain maps
   ● to support common array implementations effortlessly

2. Advanced users can write their own domain maps in Chapel
   ● to cope with shortcomings in our standard library

3. Chapel’s standard domain maps are written using the same end-user framework
   ● to avoid a performance cliff between “built-in” and user-defined cases
Domain Map Descriptors

**Domain Map**
- **Represents**: a domain map value
- **Generic w.r.t.**: index type
- **State**: the domain map’s representation
- **Typical Size**: $\Theta(1)$
- **Required Interface**:
  - create new domains

**Domain**
- **Represents**: a domain
- **Generic w.r.t.**: index type
- **State**: representation of index set
- **Typical Size**: $\Theta(1) \rightarrow \Theta(\text{numIndices})$
- **Required Interface**:
  - create new arrays
  - queries: size, members
  - iterators: serial, parallel
  - domain assignment
  - index set operations

**Array**
- **Represents**: an array
- **Generic w.r.t.**: index type, element type
- **State**: array elements
- **Typical Size**: $\Theta(\text{numIndices})$
- **Required Interface**:
  - (re-)allocation of elements
  - random access
  - iterators: serial, parallel
  - slicing, reindexing, aliases
  - get/set of sparse “zero” values
HPCC Stream Performance on Jaguar (XT5)

![Graph showing performance vs. cores for Chapel EP, Chapel Global, and MPI EP.]

- Chapel EP
- Chapel Global
- MPI EP
For More Information on Domain Maps

HotPAR’10: *User-Defined Distributions and Layouts in Chapel: Philosophy and Framework*
Chamberlain, Deitz, Iten, Choi; June 2010

CUG 2011: *Authoring User-Defined Domain Maps in Chapel*
Chamberlain, Choi, Deitz, Iten, Litvinov; May 2011

Chapel release:
- Technical notes detailing domain map interface for programmers:
  \$CHPL_HOME/doc/technotes/README.dsi
- Current domain maps:
  \$CHPL_HOME/modules/dists/*.*.chpl
  layouts/*.*.chpl
  internal/Default/*.*.chpl
Domain Maps: Next Steps

- More advanced uses of domain maps:
  - Dynamically load balanced domains/arrays
  - Resilient data structures
  - *in situ* interoperability with legacy codes
  - out-of-core computations

- Further compiler optimization via optional interfaces
  - particularly communication idioms (stencils, reductions, …)
More Data Parallelism Implementation Qs

Q1: How are forall loops implemented?

```plaintext
forall i in B.domain do B[i] = i/10.0;
```

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?

Q2: How are parallel zippered loops implemented?

```plaintext
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha * c;
```

- Particularly given that the iterands might have incompatible distributions, memory layouts, and parallelization strategies
More Data Parallelism Implementation Qs

Q1: How are forall loops implemented?

```
forall i in B.domain do B[i] = i/10.0;
```

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?

Q2: How are parallel zippered loops implemented?

```
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha * c;
```

- Particularly given that the iterands might have incompatible distributions, memory layouts, and parallelization strategies

A: Chapel’s leader-follower iterators are designed to give users full control over such decisions
Leader-Follower Iterators: Definition

- Chapel defines all forall loops in terms of leader-follower iterators:
  - leader iterators: create parallelism, assign iterations to tasks
  - follower iterators: serially execute work generated by leader

- Given...

```plaintext
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha * c;
```

...A is defined to be the leader

...A, B, and C are all defined to be followers
Leader-Follower Iterators: Rewriting

Conceptually, the Chapel compiler translates:

```
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha * c;
```

into:

```
inlined A.lead() iterator, which creates tasks that yield work {
  for (a,b,c) in zip(A.follow(work), B.follow(work), C.follow(work)) do
    a = b + alpha * c;
}
```
Writing Leaders and Followers

Leader iterators are defined using task/locality features:

```plaintext
iter BlockArr.lead() { 
  coforall loc in Locales do 
    on loc do 
      coforall tid in here.numCores do 
        yield computeMyChunk(loc.id, tid); 
  }
```

Follower iterators simply use serial features:

```plaintext
iter BlockArr.follow(work) { 
  for i in work do 
    yield accessElement(i); 
}
```
Leader-Follower Iterators: Rewriting

- Putting it all together, the following loop...

```plaintext
forall (a, b, c) in zip(A, B, C) do
  a = b + alpha * c;
```

...would get rewritten by the Chapel compiler as:

```plaintext
coforall loc in Locales do
  on loc do
    coforall tid in here.numCores {
      const work = computeMyChunk(loc.id, tid);
      for (a, b, c) in zip(A.follow(work), B.follow(work), C.follow(work)) do
        a = b + alpha * c;
    }
```
Controlling Data Parallelism

Q: “What if I don’t like the approach implemented by an array’s leader iterator?”

A: Several possibilities…
forall \((b, a, c)\) in \(\text{zip}(B, A, C)\) do
\[
a = b + \text{alpha} \times c;
\]
Controlling Data Parallelism

```chapel
const ProblemSize = {1..n} dmapped BlockCyclic(start=1, blocksize=64);

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

forall (a,b,c) in zip(A,B,C) do
  a = b + alpha * C;
```

Change the array’s default leader by changing its domain map (perhaps to one that you wrote yourself).
Controlling Data Parallelism

```plaintext
forall (a, b, c) in zip(dynamic(A, chunk=64), B, C) do
    a = b + alpha * c;
```

Explicitly invoke a standalone leader iterator (perhaps one that you wrote yourself).
Chapel Adaptive vs. OpenMP Guided

Adaptive Speedups

- Chapel (adaptive): triangular
- OpenMP (guided): triangular
- Chapel (adaptive): random
- OpenMP (guided): random

Speedup

tasks

Chapel Adaptive vs. OpenMP Guided
Leader/Follower Experimental Takeaways

Chapel loops can be competitive with OpenMP

- OpenMP’s parallel schedules are baked into the language/compiler/runtime
- Chapel’s are specified in the language at the user level
  - This permits us to write more advanced iterators like work-stealing
For More Information on Leader-Follower Iterators

PGAS 2011: *User-Defined Parallel Zippered Iterators in Chapel*, Chamberlain, Choi, Deitz, Navarro; October 2011

Chapel release:
- Primer example introducing leader-follower iterators:
  - examples/primers/leaderfollower.chpl
- Library of dynamic leader-follower range iterators:
  - *AdvancedIters* section in language specification
Summary of this Domain Maps Section

● Chapel avoids locking crucial implementation decisions into the language specification
  ● local and distributed array implementations
  ● parallel loop implementations

● Instead, these can be…
  …specified in the language by an advanced user
  …swapped in and out with minimal code changes

● The result separates the roles of domain scientist, parallel programmer, and implementation cleanly
Outline

✓ Motivation
✓ Chapel Background and Themes
✓ Tour of Chapel Concepts and Implementation
➢ Project Status and Next Steps
Implementation Status -- Version 1.7.0 (Apr 2013)

Overall Status:
- Most features work at a functional level
  - some features need to be improved or re-implemented (e.g., OOP)
- Many performance optimizations remain
  - particularly for distributed memory (multi-locale) execution

This is a good time to:
- Try out the language and compiler
- Use Chapel for non-performance-critical projects
- Give us feedback to improve Chapel
- Use Chapel for parallel programming education
The Cray Chapel Team (Summer 2012)
Chapel Community
(see chapel.cray.com/collaborations.html for further details and ideas)

● Lightweight Tasking using Qthreads: Sandia (Kyle Wheeler, Dylan Stark, Rich Murphy)
  ● paper at CUG, May 2011
● Parallel File I/O, Bulk-Copy Opt: U Malaga (Rafael Asenjo, Maria Angeles Navarro, et al.)
  ● papers at ParCo, Aug 2011; SBAC-PAD, Oct 2012
● I/O, LLVM back-end, etc.: LTS (Michael Ferguson, Matthew Lentz, Joe Yan, et al.)
● Interoperability via Babel/BRAID: LLNL/Rice (Tom Epperly, Adrian Prantl, Shams Imam)
  ● paper at PGAS, Oct 2011
● Application Studies: LLNL (Rob Neely, Bert Still, Jeff Keasler)
● Interfaces/Generics/OOP: CU Boulder (Jeremy Siek, Jonathan Turner, et al.)
● Futures/Task-based Parallelism: Rice (Vivek Sarkar, Shams Imam, Sagnak Tasırlar, et al.)
● Lightweight Tasking using MassiveThreads: U Tokyo (Kenjiro Taura, Jun Nakashima)
● CPU-accelerator Computing: UIUC (David Padua, Albert Sidelnik, Maria Garzarán)
  ● paper at IPDPS, May 2012
● Model Checking and Verification: U Delaware (Stephen Siegel, T. Zirkel, T. McClory)
● Chapel-MPI Compatibility: Argonne (Pavan Balaji, Rajeev Thakur, Rusty Lusk, Jim Dinan)
● and several others...
Next Steps

● Evolve from Prototype- to Production-grade
  ● Add/Improve Lacking Features
  ● Performance Optimizations

● Target more complex compute node types
  ● e.g., CPU+GPU, Intel MIC, …
  ● via Hierarchical Locales

● Continue to grow the user and developer communities
  ● Work toward transitioning Chapel from Cray-controlled to community-governed
Summary

Higher-level programming models can help insulate algorithms from parallel implementation details

- yet, without necessarily abdicating control
- Chapel does this via its multiresolution design
  - Here, we saw it in domain maps and leader-follower iterators
  - These avoid locking crucial performance decisions into the language

We believe Chapel can greatly improve productivity

...for current and emerging HPC architectures
...and for the growing need for parallel programming in the mainstream
CS294: Discussion Topics (not a slide from Chamberlain)

- Mixed task and data parallelism
- Data parallelism vs. collectives
- Generality of domains
- Domain-maps:
  - Locality is specified as part of domain
  - Sparse structures
- Implementation and scaling