Chapel: Productive Parallel Programming
(under the influence of jetlag)

Brad Chamberlain, Chapel Team, Cray Inc.
ETH Zürich
April 25th, 2014*

* = Happy birthday, Mom!
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.
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

<table>
<thead>
<tr>
<th>Year</th>
<th>System</th>
<th>Processors</th>
<th>Technology Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988</td>
<td>Cray Y-MP</td>
<td>8</td>
<td>Static finite element analysis, Fortran77 + Cray autotasking + vectorization</td>
</tr>
<tr>
<td>1998</td>
<td>Cray T3E</td>
<td>1,024</td>
<td>Modeling of metallic magnet atoms, Fortran + MPI (Message Passing Interface)</td>
</tr>
<tr>
<td>2008</td>
<td>Cray XT5</td>
<td>150,000</td>
<td>Superconductive materials, C++/Fortran + MPI + vectorization</td>
</tr>
<tr>
<td>~2018</td>
<td>Cray ____</td>
<td>~10,000,000</td>
<td>TBD, TBD: C/C++/Fortran + MPI + CUDA/OpenCL/OpenMP/OpenACC?</td>
</tr>
</tbody>
</table>

Or, perhaps something completely different?
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:
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):

\[
\begin{align*}
A & = = = = \\
B & + + + + \\
C & \cdot \cdot \cdot \cdot \\
\alpha & \square \square \square \square 
\end{align*}
\]
STREAM Triad: a trivial parallel computation

Given: $m$-element vectors $A, B, C$

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

In pictures, in parallel (distributed memory multicore):
STREAM Triad: 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;
  }

  scalar = 3.0;

  for (j=0; j< VectorSize; j++) {
    b[j] = 2.0;
    c[j] = 0.0;
  }

  for (j=0; j< VectorSize; j++)
    a[j] = b[j]+scalar*c[j];

  HPCC_free(c);
  HPCC_free(b);
  HPCC_free(a);

  return 0;
}
```c
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );

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

    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;

    VectorSize = HPCC_LocalVectorSize( params, 3, sizeof(double), 0 );

    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );

    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d). \n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }
    
    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j< VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 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);

    return 0;
}
```

Copyright 2014 Cray Inc.
 STREAM Triad: MPI+OpenMP vs. CUDA

HPC suffers from too many distinct notations for expressing parallelism and locality.
Why so many programming models?

HPC has traditionally given users…
…low-level, *control-centric* programming models
…ones that are closely tied to the underlying hardware
…ones that support only a single type of parallelism

**Examples:**

<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>MPI</td>
<td>executable</td>
</tr>
<tr>
<td>Intra-node/multicore</td>
<td>OpenMP/pthreads</td>
<td>iteration/task</td>
</tr>
<tr>
<td>Instruction-level vectors/threads</td>
<td>pragmas</td>
<td>iteration</td>
</tr>
<tr>
<td>GPU/accelerator</td>
<td>CUDA/OpenCL/OpenACC</td>
<td>SIMD function/task</td>
</tr>
</tbody>
</table>

**benefits:** lots of control; decent generality; easy to implement  
**downsides:** lots of user-managed detail; brittle to changes
Prototypical Next-Gen Processor Technologies

Intel MIC

AMD APU

Nvidia Echelon

Tilera Tile-Gx

General Characteristics of These Architectures

- Increased hierarchy and/or sensitivity to locality
- Potentially heterogeneous processor/memory types

⇒ Next-gen programmers will have a lot more to think about at the node level than in the past
(“Glad I’m not an HPC Programmer!”)

A Possible Reaction:
“This is all well and good for HPC users, but I’m a mainstream desktop programmer, so this is all academic for me.”

The Unfortunate Reality:
- Performance-minded mainstream programmers will increasingly need to deal with parallelism and locality too
Rewinding a few slides...

MPI + OpenMP

```c
#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 0;
}
```

CUDA

```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);
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality.
**STREAM Triad: Chapel**

```plaintext
# 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);
}

__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];
}
```

---

**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.
Outline

✓ Motivation
➢ Chapel Background and Themes
● Survey of Chapel Concepts
● 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**
Chapel's Implementation

● Being developed as open source at SourceForge

● Licensed as BSD software

● **Target Architectures:**
  ● Cray architectures
  ● multicore desktops and laptops
  ● commodity clusters and the cloud
  ● systems from other vendors
  ● *in-progress:* CPU+accelerator hybrids, manycore, …
Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap
Motivating Chapel Themes

1) General Parallel Programming
2) Global-View Abstractions
3) Multiresolution Design
4) Control over Locality/Affinity
5) Reduce HPC ↔ Mainstream Language Gap
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/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>
3) Multiresolution Design: Motivation

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

“Why don’t I have more control?”
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
5) Reduce HPC ↔ Mainstream Language Gap

Consider:
- Students graduate with training in Java, Matlab, Python, etc.
- 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
➢ Survey of Chapel Concepts

● Project Status and Next Steps

Table:

<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>
**Static Type Inference**

```chapel
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

```chapel
const r = 1..10;

printVals(r);
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 4 5 6 7 8 9 10
1 2 3
1 3 5 7 9
10 8 6 4 2
1 3 5
1 3
0 1 2 3 4 ... n-1
```
Iterators

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

```chapel
for f in fibonacci(7) do writeln(f);
```

0
1
1
2
3
5
8

```chapel
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;
}
```

```chapel
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)
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
➢ Survey of Chapel Concepts

• Project Status and Next Steps
Task Parallelism: Begin Statements

// create a fire-and-forget task for a statement
begin writeln(“hello world”);
writeln(“goodbye”);

Possible outputs:

hello world  goodbye
goodbye   hello world
Task Parallelism: Coforall Loops

// 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
Other Task Parallel Concepts

- **cobegins**: create tasks using compound statements
- **sync statements**: join unstructured tasks
- **serial statements**: conditionally squash parallelism
- **atomic variables**: support atomics ops, similar to modern C++
- **sync/single variables**: support producer/consumer patterns
Outline

✓ Motivation
✓ Chapel Background and Themes
➢ Survey of Chapel Concepts

● Project Status and Next Steps

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

Theme 4: Control over Locality/Affinity
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 = ...;
  ```

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

- Locale methods support queries about the target system:

  - `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");
  begin on A[i,j] do
    bigComputation(A);
  begin on node.left do
    search(node.left);
  ```
Chapel: Scoping and Locality

var i: int;
var i: int;

on Locales[1] {

}
Chapel: Scoping and Locality

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


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

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

      // within this scope, i, j, and k can be referenced;
      // the implementation manages the communication for i and j
    }
  }
}
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;
    }
  }
}
```

Copyright 2014 Cray Inc.
Outline

✓ Motivation
✓ Chapel Background and Themes
➢ Survey of Chapel Concepts

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

- Target Machine

● Project Status and Next Steps
Domains

**Domain:**
- A first-class index set
- The fundamental Chapel concept for data parallelism

```chapel
config const m = 4, n = 8;
const D = {1..m, 1..n};
const Inner = {2..m-1, 2..n-1};
```
Domains

**Domain:**
- A first-class index set
- The fundamental Chapel concept for data parallelism
- Useful for declaring arrays and computing with them

```chapel
config const m = 4, n = 8;
const D = {1..m, 1..n};
const Inner = {2..m-1, 2..n-1};
var A, B, C: [D] real;
```
Chapel Domain/Array Operations

- Data Parallel Iteration
  
  ```chapel
  forall (i,j) in D do
  A[i,j] = i + j/10.0;
  ```

- Array Slicing; Domain Algebra
  
  ```chapel
  A[InnerD] = B[InnerD+(0,1)];
  ```

- Promotion of Scalar Functions and Operators
  
  ```chapel
  A = exp(B, C);
  A = foo("hi", B, C);
  A = B + alpha * C;
  ```

- And many others: reductions, scans, reallocation, reshaping, remapping, set operations, aliasing, …
Notes on Forall Loops

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

Typically:
- \(1 \leq \#\text{Tasks} \ll \#\text{Iterations}\)
- \(\#\text{Tasks} \approx \text{amount of HW parallelism}\)

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

Like for loops, forall-loops may be zippered, and corresponding iterations will match up
Chapel supports several types of domains (index sets):

- **dense**
- **strided**
- **sparse**
- **associative**
- **unstructured**
Chapel Array Types

- dense
- strided
- sparse
- associative
- unstructured

Copyright 2014 Cray Inc.
STREAM Triad in Chapel

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

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

A = B + alpha * C;
```
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
- Survey of Chapel Concepts

- Project Status and Next Steps

Theme 2: Global-view Abstractions

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
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:
```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 domain indices and array values
- computation will execute using local processors only

Copyright 2014 Cray Inc.
STREAM Triad: Chapel (multilocation, blocked)

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

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

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

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

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

\[ \text{var A, B, C: [ProblemSpace]} \text{ real;} \]
\[ A = B + \alpha \cdot C; \]
LULESH: a DOE Proxy Application

**Goal:** Solve one octant of the spherical Sedov problem (blast wave) using Lagrangian hydrodynamics for a single material

pictures courtesy of Rob Neely, Bert Still, Jeff Keasler, LLNL
LULESH in Chapel

1288 lines of source code

plus 266 lines of comments
487 blank lines

(the corresponding C+MPI+OpenMP version is nearly 4x bigger)

This can be found in Chapel v1.9 in examples/benchmarks/lulesh/*.chpl
LULESH in Chapel

This is all of the representation dependent code. It specifies:

- data structure choices
  - structured vs. unstructured mesh
  - local vs. distributed data
  - sparse vs. dense materials arrays
- a few supporting iterators
LULESH in Chapel

Here is some sample representation-independent code

IntegrateStressForElems()

LULESH spec, section 1.5.1.1 (2.)
Representation-Independent Physics

```plaintext
proc IntegrateStressForElems(sigxx, sigyy, sigzz, determ) {
  forall k in Elems {
    var b_x, b_y, b_z: 8*real;
    var x_local, y_local, z_local: 8*real;
    localizeNeighborNodes(k, x, x_local, y, y_local, z, z_local);

    var fx_local, fy_local, fz_local: 8*real;

    local {
      /* Volume calculation involves extra work for numerical consistency. */
      CalcElemShapeFunctionDerivatives(x_local, y_local, z_local, b_x, b_y, b_z, determ[k]);

      CalcElemNodeNormals(b_x, b_y, b_z, x_local, y_local, z_local);

      SumElemStressesToNodeForces(b_x, b_y, b_z, sigxx[k], sigyy[k], sigzz[k],
                                 fx_local, fy_local, fz_local);
    }

    for (noi, t) in elemToNodesTuple(k) {
      fx[noi].add(fx_local[t]);
      fy[noi].add(fy_local[t]);
      fz[noi].add(fz_local[t]);
    }
  }
}
```

Because of domain maps, this code is independent of:
- structured vs. unstructured mesh
- shared vs. distributed data
- sparse vs. dense representation
Chapel’s Domain Map Philosophy

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

2. Expert users can write their own domain maps in Chapel
   ● to cope with any 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

Copyright 2014 Cray Inc.
All Chapel domain types support domain maps.

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

Copyright 2014 Cray Inc.
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:

- Current domain maps:
  - $CHPL_HOME/modules/dists/*.chpl
  - layouts/*.chpl
  - internal/Default*.chpl

- Technical notes detailing the domain map interface for implementers:
  - $CHPL_HOME/doc/technotes/README.dsi
Two Other Thematically Similar Features

1) **leader-follower iterators**: Permit users to specify the parallelism and work decomposition used by forall loops
   - including zippered forall loops

2) **locale models**: Permit users to model the target architecture and how Chapel should be implemented on it
   - e.g., how to manage memory, create tasks, communicate, …

Like domain maps, these are…

…written in Chapel by expert users using lower-level features
   - e.g., task parallelism, on-clauses, base language features, …

…available to the end-user via higher-level abstractions
   - e.g., forall loops, on-clauses, lexically scoped PGAS memory, …
Summary of this Section

- Chapel avoids locking crucial implementation decisions into the language specification
  - local and distributed parallel array implementations
  - parallel loop scheduling policies
  - target architecture models

- Instead, these can be…
  …specified in the language by an advanced user
  …swapped between with minimal code changes

- The result cleanly separates the roles of domain scientist, parallel programmer, and compiler/runtime
Outline

♥ Motivation
♥ Chapel Background and Themes
♥ Survey of Chapel Concepts
♥ Project Status and Next Steps
Implementation Status -- Version 1.9.0 (Apr 2014)

Overall Status:

- **User-facing Features:** generally in good shape
  - some require additional attention (e.g., strings, OOP)
- **Multiresolution Features:** in use today
  - their interfaces are likely to continue evolving over time
- **Error Messages:** not always as helpful as one would like
  - correct code works well, incorrect code can be puzzling
- **Performance:** hit-or-miss depending on the idioms used
  - Chapel designed to ultimately support competitive performance
  - effort to-date has focused primarily on correctness

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
When teaching parallel programming, I like to cover:
- data parallelism
- task parallelism
- concurrency
- synchronization
- locality/affinity
- deadlock, livelock, and other pitfalls
- performance tuning
- …

I don’t think there’s been a good language out there…
- for teaching all of these things
- for teaching some of these things well at all
- until now: We believe Chapel can play a crucial role here

(see http://chapel.cray.com/education.html for more information and http://cs.washington.edu/education/courses/csep524/13wi/ for my use of Chapel in class)
Chapel: the next five years

- **Harden prototype to production-grade**
  - add/improve lacking features
  - optimize performance

- **Target more complex/modern compute node types**
  - e.g., Intel MIC, CPU+GPU, AMD APU, …

- **Continue to grow the user and developer communities**
  - including nontraditional circles: desktop parallelism, “big data”
  - transition Chapel from Cray-managed to community-governed
Chapel Execution Time Trends (v1.5–1.9)
(available for browsing at: http://chapel.sourceforge.net/perf/)

Copyright 2014 Cray Inc.
The Cray Chapel Team (Summer 2013)

Chapel USA

Chapel Seattle
Chapel...

...is a collaborative effort — join us!
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 principally in domain maps
  - leader-follower iterators and locale models are other examples
  - these avoid locking crucial policy decisions into the language

We believe Chapel can greatly improve productivity

...for current and emerging HPC architectures
...for emerging mainstream needs for parallelism and locality
For More Information: Online Resources

Chapel project page: http://chapel.cray.com
- overview, papers, presentations, language spec, …

Chapel SourceForge page: https://sourceforge.net/projects/chapel/
- release downloads, community mailing lists, repository, …

MailingAliases:
- chapel_info@cray.com: contact the team at Cray
- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: developer discussion
- chapel-education@lists.sourceforge.net: educator discussion
- chapel-bugs@lists.sourceforge.net: public bug forum

Copyright 2014 Cray Inc.
For More Information: Suggested Reading

Overview Papers:

  ● a more detailed overview of Chapel’s history, motivating themes, features

  ● a high-level overview of the project summarizing the HPCS period

Blog Series:

  ● a series of technical opinion pieces designed to combat standard arguments against the development of high-level parallel languages
Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA, and YARCDATA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

Copyright 2014 Cray Inc.