An Overview of Chapel: a productive parallel programming language

Sung-Eun Choi
Cray, Inc.

KIIFE-KOCSEA HPC SIG Joint Workshop @ SC’12
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):
STREAM Triad: a trivial parallel computation

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

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

**In pictures, in parallel (distributed memory multicore):**

![Diagram showing parallel computation]
```c
#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] = 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);

    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).
" , 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;
}
#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;
    c[idx] = a[idx]+scalar*b[idx];
}

#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 = HPCC_Sync( params, 0 == myRank);
    int errCount;
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );
    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) { return 0;
    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
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 0.0;
    }
    scalar = 3.0f;
    #ifdef _OPENMP
    #pragma omp parallel for
    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];
    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);
    return 0;
    #endif
    #endif
}
#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;
    c[idx] = a[idx]+scalar*b[idx];
}

#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;i
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );
    MPI_Comm comm_world = MPI_COMM_WORLD;
    if ( params->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);
    return 0;
}
### Chapel

```chapel
config const m = 1000,
    alpha = 3.0;

const ProblemSpace = {1..m} dmapped ...;

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

B = 2.0;
C = 3.0;
A = B + alpha * C;
```
#define N 2000000

```c
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];
}
```

#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;
    }

define _OPENMP
#pragma omp parallel for
#endif
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 0.0;
    }
    scalar = 3.0;

define _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;
}
```

MPI + OpenMP

config const m = 1000,
alpha = 3.0;

const ProblemSpace = {1..m} dmapped ...;

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

B = 2.0;
C = 3.0;
A = B + alpha * C;
Motivation

- Chapel Background and Themes
  - Tour of Chapel Concepts
  - Project Status
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
- 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
Recall from our STREAM example..

<table>
<thead>
<tr>
<th>Style 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</td>
<td>iteration/task</td>
</tr>
<tr>
<td>GPU/accelerator</td>
<td>CUDA</td>
<td>SIMD function/task</td>
</tr>
</tbody>
</table>
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 all parallelism available in the hardware

- **Types**: machines, nodes, cores, instructions

<table>
<thead>
<tr>
<th>Style 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>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?”
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
Outline

✓ Motivation
✓ Chapel Background and Themes
➢ Tour of Chapel Concepts
● Project Status
Base Language Features

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

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
Task: a unit of (parallel) work in Chapel

- All parallelism is implemented using tasks
- `main()` is the only task when a program begins
begin writeln(“Hello from task 0 of 4”);
begin writeln(“Hello from task 1 of 4”);
begin writeln(“Hello from task 2 of 4”);
begin writeln(“Hello from task 3 of 4”);
writeln(“All tasks done”);
cobegin {
    writeln("Hello from task 0 of 4");
    writeln("Hello from task 1 of 4");
    writeln("Hello from task 2 of 4");
    writeln("Hello from task 3 of 4");
}

writeln("All tasks done");
Coforall Loops

```coforall
coforall t in 0..numTasks-1 do
    writeln("Hello from task ", t, " of ", numTasks);

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

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
Locality Features

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
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
- Can be queried for characteristics like amount of memory or number of cores

**Typically:** A multi-core processor or an SMP
Controlling Locality

- Users specify the number of locales to use at program launch
  - The locale variables are available to the user in an built-in array called \texttt{Locales}

- \textit{On-clauses} support placement of computations:

  writeln(“on locale 0”);
  on Locales[1] do
    writeln(“now on locale 1”);
  writeln(“on locale 0 again”);
Parallelism and Locality are distinct concepts

- e.g., begin statements create tasks
  - new task will run on the current locale
- e.g., on-clauses place computation
  - no parallelism introduced

Composing these concepts can be very powerful
Data Parallel Features

- Domain Maps
- Data Parallelism
- Task Parallelism
- Base Language
- Locality Control
- Target Machine
Chapel supports several types of domains (index sets):

- Dense
- Strided
- Sparse
- Associative
- Unstructured
Chapel Array Types

Each domain type can be used to declare arrays:

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

- Parallel and Serial Iteration
  
  ```chapel
  for a in A do a = 0.0;
  for all (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 Operators and Functions
  
  ```chapel
  A = B + alpha * C;
  A = exp(B, C);
  ```

- And several others: indexing, reallocation, set operations, remapping, aliasing, queries, ...
Important Note #2

Operations on arrays are the same regardless of the domain that is used to declare the arrays

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

- A, B, and C could be dense, strided, sparse, associative, or unstructured
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? ...

...?
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
const ProblemSpace = {1..m};

\[
\begin{align*}
\alpha \cdot C + \beta
\end{align*}
\]

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

\[
A = B + \alpha \cdot C;
\]
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
const ProblemSpace = {1..m}

\textbf{dmapped} Block(boundingBox={1..m});

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

A = B + alpha * C;
const ProblemSpace = {1..m}

dmapped Cyclic(startIdx=1);

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

A = B + alpha * C;
Important Note #3

All Chapel domain types support domain maps
Standard Domain Maps

Completed domain maps:
- Block, Cyclic, Replicated
- Sparse COO and CSR
- Quadratic probing associative

In the works:
- Block-Cyclic, 2D dimensional
- Distributed associative and sparse
More Domain Maps

Users can write their own domain maps

- GPU computations
- Dynamically load balanced domains/arrays
- Resilient data structures
- *in situ* interoperability with legacy codes
- out-of-core computations
- ...
Chapel is a new parallel programming language aimed at drastically improving programmer productivity

- Chapel avoids locking crucial implementation decisions into the language specification
  - Separates the roles of domain scientist and HPC expert
  - Results in much cleaner, maintainable code
Outline

- Motivation
- Chapel Background and Themes
- Tour of Chapel Concepts
- Project Status
In a nutshell:

- Most features work
- Many performance optimizations remain

This is a good time to:

- Try out the language and compiler
- Give us feedback on the language
- Use Chapel for parallel programming education
- Use Chapel for non-performance-critical projects
Some Next Steps

• Grow the set of architectures we can target effectively
• Grow the set of codes we are evaluating
• Performance optimizations
• Evolve from prototype- to production-grade
For More Information

Chapel project page: [http://chapel.cray.com](http://chapel.cray.com)
- overview, papers, presentations, language spec, ...

Chapel SourceForge page: [https://sourceforge.net/projects/chapel/](https://sourceforge.net/projects/chapel/)
- release downloads, public mailing lists, code repository, ...

Mailing Lists:
- chapel_info@cray.com: contact the team
- chapel-users@lists.sourceforge.net: user-oriented discussion list
- chapel-developers@lists.sourceforge.net: dev.-oriented discussion
- chapel-education@lists.sourceforge.net: educator-oriented discussion
- chapel-bugs@lists.sourceforge.net: public bug forum
- chapel_bugs@cray.com: private bug mailing list