Multiresolution Parallel Programming with Chapel

Vassily Litvinov and the Chapel team, Cray Inc.

HPC Advisory Council Conference
13 September 2012
Outline

- Motivation: Programming Models
  - Multiresolution Programming
  - Empirical Evaluation
  - About the Project
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:**

- $A$
- $B$
- $C$
- $\alpha$

$A = A_i = B_i + \alpha \cdot C_i$
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:**

```
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, distributed memory:**

\[
\begin{align*}
A &= = = = = \\
B &= + + + + \\
C &= \cdot \cdot \cdot \cdot \\
\alpha &= \cdot \cdot \cdot \cdot 
\end{align*}
\]
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:
#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\n", VectorSize );
            fclose( outFile );
        }
        return 1;
    }

    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 3.0;
    }

    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] = 3.0;
    }
    
    scalar = 3.0;
    
    #ifdef _OPENMP
    #pragma omp parallel for
    #endif
    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];
    
    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);
    
    return 0;
}
```
#include <hpcc.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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

int HPCC_StarStream(HPCC_Params *params) { int myRank, commSize;
int rv, errCount;
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Comm_size( comm, &commSize );
MPI_Comm_rank( comm, &myRank );
rv = HPCC_StarStream( 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;
}
#endif
#pragma omp parallel for
#endif
for (j=0; j<VectorSize; j++) {
  b[j] = 2.0;
  c[j] = 3.0;
}
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;
}  #ifdef _OPENMP
#endif

__global__ void set_array(float *a, float *b, float *c, 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];
}  
```
Why so many programming models?

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>process</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>

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

**benefits:** lots of control; decent generality; easy to implement

**downsides:** lots of user-managed detail; brittle to changes
“Why is everything so tedious/difficult?”
“Why don’t my programs port trivially?”

“Why don’t I have more control?”
Outline

✓ Motivation

➤ Multiresolution programming
  • Empirical Evaluation
  • About the Project
**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

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

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)
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] = 3.0;
}
scalar = 3.0;
#ifdef _OPENMP
#pragmaomp 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;
```

CUDA

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

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

int HPCC_StarStream(HPCC_Params *params, *par)
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] = 3.0;
}
scalar = 3.0;
#ifdef _OPENMP
#pragmaomp 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;
```

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;
```
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
Q: How are loops implemented?

A = B + alpha * C; // an implicit loop

- How many tasks? Where do they execute?
- How is the iteration space divided between the tasks?
  - statically? dynamically? what algorithm?

“leader-follower” iterators (PGAS 2011 paper)

A: Chapel’s domain maps are designed to give the user full control here, too
const ProblemSpace = [1..m];

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

A = B + alpha * C;

No domain map is specified => use the default one
• current locale owns all indices and values
• computation will execute using local processors only, in parallel
const ProblemSpace = [1..m];

dmapped Block(boundingBox=[1..m]);

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

A = B + alpha * C;

Block domain map is chosen explicitly
• indices and values are distributed over all locales
• computation will execute on all locales and processors, in parallel
const ProblemSpace = [1..m];

dmapped Cyclic(startIdx=1);

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

A = B + alpha * C;

Cyclic domain map is chosen explicitly
• similarly, distributed values, distributed+parallel computation
Given an implicit loop...

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

or an equivalent explicit loop

- `forall` indicates it is parallel

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

the compiler converts it to

\[
\text{for } \text{followThis in } A.\text{domain_map}.\text{these}(\ldots) \{ \\
\text{for } (a, b, c) \text{ in } (A.\text{domain_map}.\text{these}(\text{followThis}, \ldots), \\
B.\text{domain_map}.\text{these}(\text{followThis}, \ldots), \\
C.\text{domain_map}.\text{these}(\text{followThis}, \ldots)) \{ \\
a = b + \alpha \times c; \\
\}
\}
\]

Chapel’s `iterator` – here enables user to introduce distribution and parallelism

“leader/follower” scheme (not in this talk)
Domain Maps: The User Can

- ... and the domain map author implements these iterators, for example:

```plaintext
iter MyDomainMap.these(...) {
  coforall loc in Locales {
    on loc {
      coforall task in 1..here.numCores {
        yield computeMyChunk(loc.id, task);
      }
    }
  }
}
```

\[ \alpha \cdot \text{Something} = + \text{Something} \]
Domain maps are “recipes” that instruct the compiler how to implement global-view computations.

- Unless requested explicitly, a reasonable default domain map/implementation is used.
- Chapel provides a library of standard domain maps to support common array implementations effortlessly.
- Advanced users can write their own domain maps in Chapel to cope with shortcomings in the standard library using Chapel – all of the language is fully available.

switching to other resolution levels for more control
- not required, but available when desired
• 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 cleanly separates the roles of domain scientist, parallel programmer, and implementation
Outline

✓ Motivation
✓ Multiresolution programming
➢ Empirical Evaluation
• About the Project
User-Defined Parallel Iterators

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

- Implemented various scheduling policies
  - OpenMP-style dynamic and guided
  - adaptative, with work stealing
  - available as iterators
- Compared performance against OpenMP
  - Chapel is competitive

Chapel’s multi-resolution design allows HPC experts to implement desired policies and scientists to incorporate them with minimal code changes
Chapel vs. OpenMP Guided

Guided scheduling Speedups

- Base (16)
- Chapel (16)
- OMP (16)
- Base (32)
- Chapel (32)
- OMP (32)
Chapel Adaptive vs. OpenMP Guided

![Adaptive Speedups Diagram](image-url)
The domain map allows the user to target GPUs with minimal code changes
Parboil Benchmark Suite

- CUDA Tuned Chapel w/ Explicit Transfer
- Tuned Chapel w/ Implicit Transfer

Execution Time (ms)

- CUDA: 60 ms
- Tuned Chapel w/ Explicit Transfer: 60 ms
- Tuned Chapel w/ Implicit Transfer: 67 ms

MRI-FHD

I/O
Compute
Two Point Angular Correlation Function (TPACF)

- CUDA
- Tuned Chapel w/ Explicit Transfer
- Tuned Chapel w/ Implicit Transfer

Execution Time (ms)

- I/O
- Compute
## Code Size Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Lines (CUDA)</th>
<th># Lines (Chapel)</th>
<th>% difference</th>
<th># of Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP</td>
<td>186</td>
<td>154</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>MRI-FHD</td>
<td>285</td>
<td>145</td>
<td>49</td>
<td>2</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>250</td>
<td>125</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>RPES</td>
<td>633</td>
<td>504</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>TPACF</td>
<td>329</td>
<td>209</td>
<td>36</td>
<td>1</td>
</tr>
</tbody>
</table>
Outline

✓ Motivation
✓ Multiresolution programming
✓ Empirical Evaluation
➤ About the Project
Chapel's Implementation

- Being developed as open source at SourceForge
  - BSD license

- **Target Architectures:**
  - Cray architectures
  - multicore desktops and laptops
  - commodity clusters
  - systems from other vendors
  - *in-progress*: CPU+accelerator hybrids, manycore, ...

- Try it out and give us feedback!
Some Next Steps

- Hierarchical Locales
- Resilience Features
- Performance Optimizations
- Evolve from Prototype- to Production-grade
- Evolve from Cray- to community-language
- and much more...
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