Data Parallelism with Locality:
Domain Maps / Distributions (16x9 slides)
Scalable Parallel Programming Concerns

Q: What should scalable parallel programmers focus on?
A: **Parallelism**: What should execute simultaneously?

**Locality**: Where should those tasks execute?
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 + \alpha \cdot C
```
STREAM Triad: a trivial parallel computation

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

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

In pictures, in parallel (shared memory / multicore):
STREAM Triad: a trivial parallel computation

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

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

In pictures, in parallel (distributed memory):
STREAM Triad: a trivial parallel computation

**Given:** \( m \)-element vectors \( A, B, C \)

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

In pictures, in parallel (distributed memory multicore):
#include "hpcc.h"

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

int HPCC_StarStream(HPCC_Params *params) {
    int myRank, commSize;
    int rv, errCount;
    MPI_Comm comm = MPI_COMM_WORLD;
    
    MPI_Comm_size( comm, &commSize );
    MPI_Comm_rank( comm, &myRank );
    
    rv = HPCC_Stream( params, 0 == myRank);
    MPI_Reduce( &rv, &errCount, 1, MPI_INT, MPI_SUM, 0, comm );
    return errCount;
}

int HPCC_Stream(HPCC_Params *params, int doIO) {
    register int j;
    double scalar;
    
    VectorSize = HPCC_LocalVectorSize( params, 3,
    sizeof(double), 0 );
    
    a = HPCC_XMALLOC( double, VectorSize );
    b = HPCC_XMALLOC( double, VectorSize );
    c = HPCC_XMALLOC( double, VectorSize );
    
    if (!a || !b || !c) {
        if (c) HPCC_free(c);
        if (b) HPCC_free(b);
        if (a) HPCC_free(a);
        if (doIO) {
            fprintf( outFile, "Failed to allocate memory (%d).\n",
                    VectorSize );
            fclose( outFile );
        }
        return 1;
    }
    
    for (j=0; j<VectorSize; j++) {
        b[j] = 2.0;
        c[j] = 1.0;
    }
    scalar = 3.0;
    
    for (j=0; j<VectorSize; j++)
    a[j] = b[j]+scalar*c[j];
    
    return 0;  }
#include <hpcc.h>
#endif

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).
                     ^
                     %zd_a, sizeof(float)*N);  
            cudaThreadSynchronize();
            return 1;
        }
    }
    return 0;  
}

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;
}
STREAM Triad: MPI+OpenMP

```c
#include <hpcc.h>
#include <omp.h>
#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];
}
```

HPC suffers from too many distinct notations for expressing parallelism and locality. This tends to be a result of bottom-up language design.
Why so many programming models?

HPC tends to approach programming models bottom-up:

Given a system and its core capabilities…

…provide features that permit users to access the available performance.

- portability? generality? programmability? These are second- or third-order concerns, if that.

<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 / Open[MP</td>
<td>CL</td>
</tr>
</tbody>
</table>

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

**downsides:** lots of user-managed detail; brittle to changes
#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];
}

#include <hpcc.h>
#include <omp.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);
    int 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;
    for (j=0; j<VectorSize; j++)
        a[j] = b[j]+scalar*c[j];
    HPCC_free(c);
    HPCC_free(b);
    HPCC_free(a);
    return 0;  }

Philosophy: Good, top-down language design can tease system-specific implementation details away from an algorithm, permitting the compiler, runtime, applied scientist, and HPC expert to each focus on their strengths.
Data Parallelism, by example

This is a shared memory program
Nothing has referred to remote locales, explicitly or implicitly

dataParallel.chpl

```chpl
config const n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5

```
1.1 1.3 1.5 1.7 1.9  
2.1 2.3 2.5 2.7 2.9  
3.1 3.3 3.5 3.7 3.9  
4.1 4.3 4.5 4.7 4.9  
5.1 5.3 5.5 5.7 5.9  
```
Distributed Data Parallelism, by example

dataParallel.chpl

```chpl
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
    dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=4

1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
Distributed Data Parallelism, by example

Not at all…
• Lowering of code is well-defined
• User can control details
• Part of Chapel's multiresolution philosophy…

```chpl
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i, j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

```
prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
```
Chapel’s Multiresolution Design: Motivation

“Why is everything so tedious/difficult?”

“Why don’t my programs trivially port to new systems?”

“Why don’t I have more control?”
**Multiresolution Design**:
Support multiple tiers of features

- higher levels for programmability, productivity
- lower levels for greater degrees of control

- build the higher-level concepts in terms of the lower
- permit users to intermix layers arbitrarily
Distributed Data Parallelism, by example

Chapel’s prescriptive approach:

```chpl
forall (i,j) in D do...
```

⇒ invoke and inline D’s default parallel iterator
• defined by D’s type / domain map

```chpl
cfg cons n = 1000;
var D = {1..n, 1..n};

var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
writeln(A);
```

default domain map
• create a task per local core
• block indices across tasks

prompt> chpl dataParallel.chpl
prompt> ./dataParallel --n=5 --numLocales=4
1.1 1.3 1.5 1.7 1.9
2.1 2.3 2.5 2.7 2.9
3.1 3.3 3.5 3.7 3.9
4.1 4.3 4.5 4.7 4.9
5.1 5.3 5.5 5.7 5.9
Distributed Data Parallelism, by example

Chapel’s prescriptive approach:

```
forall (i,j) in D do...
```

⇒ invoke and inline D’s default parallel iterator
- defined by D’s type / domain map

```
dataParallel.chpl
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
    dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
    A[i,j] = i + (j - 0.5)/n;
writeln(A);
```
Distributed Data Parallelism, by example

Chapel’s prescriptive approach:

```chapel
forall (i,j) in D do...
```

```chapel
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
```

What if I don’t like D’s iteration strategy?

- Write and call your own parallel iterator:
  ```chapel
  forall (i,j) in myParIter(D) do...
  ```

Distributed Data Parallelism, by example

Chapel's prescriptive approach:

```chapel
forall (i,j) in D do...
```

```chapel
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
```

What if I don’t like D’s iteration strategy?

- Write and call your own parallel iterator:
  ```chapel
  forall (i,j) in myParIter(D) do...
  ```

```chapel
parallel --n=5 --numLocales=4
```
Distributed Data Parallelism, by example

Chapel’s prescriptive approach:

```
forall (i, j) in D do...
```

What if I don’t like D’s iteration strategy?

- Write and call your own parallel iterator:
  ```
  forall (i, j) in myParIter(D) do...
  ```
- Or, use a different domain map:
  ```
  var D = {1..n, 1..n} dmapped Block(...);
  ```

```
dataParallel.chpl
```
```
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
  dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i, j) in D do
  A[i, j] = i + (j - 0.5)/n;
```

```
dataParallel.chpl
```
```
Distributed Data Parallelism, by example
Copyright 2018 Cray Inc.
```
Distributed Data Parallelism, by example

Chapel’s prescriptive approach:

```chapel
forall (i,j) in D do...
```

What if I don’t like D’s iteration strategy?

- Write and call your own parallel iterator:
  ```chapel
  forall (i,j) in myParIter(D) do...
  ```

- Or, use a different domain map:
  ```chapel
  var D = {1..n, 1..n} dmapped Block(...);
  ```

- Or, write and use your own domain map:
  ```chapel
  var D = {1..n, 1..n} dmapped MyDomMap(...);
  ```

```chapel
use CyclicDist;
config const n = 1000;
var D = {1..n, 1..n}
  dmapped Cyclic(startIdx = (1,1));
var A: [D] real;
forall (i,j) in D do
  A[i,j] = i + (j - 0.5)/n;
```
Use a Different Domain Map
Domain Maps: A Multiresolution Feature

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:
Domain Maps: A Multiresolution Feature

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

...to the target locales’ memory

Domain Maps specify...
...mapping of indices to locales
...layout of domains / arrays in memory
...parallel iteration strategies
...core operations on arrays / domains
Sample Domain Maps: Block and Cyclic

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

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

distributed to

![Block map]

L0 L1 L2 L3
L4 L5 L6 L7

![Cyclic map]

L0 L1 L2 L3
L4 L5 L6 L7
Write and Use Your Own Domain Map
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 end-user framework
   ● to avoid a performance cliff between “built-in” and user-defined cases
   ● in fact every Chapel array is implemented using this framework
# Domain Map Descriptors

<table>
<thead>
<tr>
<th>Domain Map</th>
<th>Domain</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Represents:</strong></td>
<td>a domain map value</td>
<td>a domain</td>
</tr>
<tr>
<td><strong>Generic w.r.t.:</strong></td>
<td>index type</td>
<td>index type, element type</td>
</tr>
<tr>
<td><strong>State:</strong></td>
<td>the domain map’s representation</td>
<td>representation of index set</td>
</tr>
<tr>
<td><strong>Typical Size:</strong></td>
<td>$\Theta(1) \rightarrow \Theta(numLocales)$</td>
<td>$\Theta(1) \rightarrow \Theta(numIndices)$</td>
</tr>
<tr>
<td><strong>Required Interface:</strong></td>
<td>create new domains, which locale owns index $i$?</td>
<td>create new arrays, queries: size, members, iterators: serial, parallel, domain assignment, index set operations</td>
</tr>
</tbody>
</table>

**Array**

- Represents: an array
- Generic w.r.t.: index type, element type
- State: array elements
- Typical Size: $\Theta(numIndices)$
- Required Interface:
  - (re-)allocation of elements
  - random access
  - iterators: serial, parallel
  - get/set of sparse “zero” values
  - …
Two Other Thematically Similar Features

1) **parallel 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, …
Chapel and Performance Portability

- Avoid locking key policy decisions into the language
  - Array memory layout?
  - Sparse storage format?
  - Parallel loop policies?
Chapel and Performance Portability

● Avoid locking key policy decisions into the language
  ● Array memory layout? not defined by Chapel
  ● Sparse storage format? not defined by Chapel
  ● Parallel loop policies? not defined by Chapel
  ● Abstract node architecture? not defined by Chapel

● Instead, permit users to specify these in Chapel itself
  ● support performance portability through…
    …a separation of concerns
    …abstractions—known to the compiler, and therefore optimizable
  ● goal: make Chapel a future-proof language
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
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 publicly 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.: CHAPEL, CLUSTER CONNECT, CLUSTERSTOR, CRAYDOC, CRAYPAT, CRAYPORT, DATAWARP, ECOPHLEX, LIBSCI, NODEKARE, REVEAL. 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 on this website are the property of their respective owners.