Data Parallelism, By Example
Safe Harbor Statement

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.
Data Parallel Features

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

Higher-level Chapel

Target Machine

Copyright 2014 Cray Inc.
Data Parallel “Hello, world!”

```chapel
config const numIters = 1000;

forall i in 1..numIters do
    writeln(“Hello, world! “,
             “from iteration “, i, “ of “, numIters);
```

Hello, world! from iteration 500 of 1000
Hello, world! from iteration 501 of 1000
Hello, world! from iteration 502 of 1000
Hello, world! from iteration 503 of 1000
Hello, world! from iteration 1 of 1000
Hello, world! from iteration 2 of 1000
Hello, world! from iteration 3 of 1000
Hello, world! from iteration 4 of 1000
...

Copyright 2014 Cray Inc.
Defining our Terms

Data Parallelism: a style of parallel programming in which parallelism is driven by computations over collections of data elements or their indices.
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 in Chapel

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

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

A = B + alpha * C;
```

Copyright 2014 Cray Inc.
Domains

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

```plaintext
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;
```
Data Parallelism by Example: Jacobi Iteration

\[ A = \left( \begin{array}{c|cc|cc} & & \cdots & & \\ \hline & & & & \\ \hline & & & & \\ \hline & & & & \\ \hline & & & & \\ \end{array} \right) \]

\[ \sum \left( \begin{array}{c|c|c|c} & & \cdots & \\ \hline & & & \\ \hline & & & \\ \hline & & & \\ \end{array} \right) \div 4 \]

repeat until max change < \( \varepsilon \)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Jacobi Iteration in Chapel

```
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do

delta = max reduce abs(A - Temp);
A = Temp;
}
while (delta > epsilon);
writeln(A);
```

Declare program parameters

- **config**: `n` and `epsilon` cannot change values after initialization.
- **const**: can be set on executable command-line

```
prompt> jacobi --n=10000 --epsilon=0.0001
```

Note that no types are given; they’re inferred from initializers

- `n` is a default integer (64 bits)
- `epsilon` is a default real floating-point (64 bits)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD]real;
A[LastRow] = 1.0;
do {
    forall (i,j) in D do
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);
writeln(A);
```

Declare domains (first class index sets)

{lo..hi, lo2..hi2} ⇒ 2D rectangular domain, with 2-tuple indices

Dom1[Dom2] ⇒ computes the intersection of two domains

.exterior() ⇒ one of several built-in domain generators
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;
do {
    forall (i,j) in D do
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
}while (delta > epsilon);
writeln (A);
```

Declare arrays

- `var` ⇒ can be modified throughout its lifetime
- `: [Dom] T` ⇒ array of size `Dom` with elements of type `T`
- `(no initializer)` ⇒ values initialized to default value (0.0 for reals)
Jacobi Iteration in Chapel

```chapel
config const n = 6,
     epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do
    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Set Explicit Boundary Condition:

Arr[Dom] ⇒ refer to array slice (“forall i in Dom do …Arr[i]…”)

Copyright 2014 Cray Inc.
Jacobi Iteration in Chapel

```chapel
config const n = 6,

Compute 5-point stencil
forall ind in Dom ⇒ parallel forall expression over Dom’s indices,
  binding them to ind
  (here, since Dom is 2D, we can de-tuple the indices)

\[ \sum \left( \begin{array}{c}
  A[i-1,j] \\
  A[i+1,j] \\
  A[i,j-1] \\
  A[i,j+1]
\end{array} \right) / 4 \]

forall (i,j) in D do

const delta = max reduce abs(A[D] - Temp[D]);
A[D] = Temp[D];
while (delta > epsilon);
writeln(A);
```

Copyright 2014 Cray Inc.
Jacobi Iteration in Chapel

config const n = 6,
           epsilon = 1.0e-5;

Compute maximum change

\textbf{op reduce} \Rightarrow \text{collapse aggregate expression to scalar using } \textit{op}

**Promotion:** abs() and \(-\) are scalar operators; providing array operands results in parallel evaluation equivalent to:

\texttt{forall (a,t) in zip(A,Temp) do abs(a - t)}

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

const delta = max reduce abs(A[D] - Temp[D]);
A[D] = Temp[D];
while (delta > epsilon);
```

writeln(A);
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;
do { 
   forall (i,j) in D do 

    const delta = max reduce abs(A[D] - Temp[D]); 
    A[D] = Temp[D];
} while (delta > epsilon);
writeln(A);
```

**Copy data back & Repeat until done**

uses slicing and whole array assignment
standard `do...while` loop construct
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```

Write array to console
Jacobi Iteration in Chapel

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;

A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);
```
Chapel Data Parallel 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,** ...
Tiled Row-Major Order Iterator

// iterate over domain D using tilesize x tilesize tiles in row-major order

iter tiledRMO(D, tilesize) {
    const tile = {0..#tilesize, 0..#tilesize};

    for base in D by tilesize do
        for ij in D[tile.translate(base)] do
            yield ij;
}

for ij in tiledRMO({1..m, 1..n}, 2) do
    write(ij);

Output:

(1,1) (1,2) (2,1) (2,2) (1,3) (1,4) (2,3) (2,4) ...
(3,1) (3,2) (4,1) (4,2) (3,3) (3,4) (4,3) (4,4) ...
Chapel Domain Types

dense

strided

sparse

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

associative

unstructured
Chapel Array Types

- **dense**
- **strided**
- **sparse**

- **associative**
- **unstructured**
Domain Maps

Diagram:

1. Domain Maps
2. Data Parallelism
3. Task Parallelism
4. Base Language
5. Locality Control
6. Target Machine

Copyright 2014 Cray Inc.
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:
config const numIters = 100000;

const D = {1..numIters};

forall i in D do
  writeln("Hello, world! ",
         "from iteration ", i, " of ", numIters);
config const numIters = 100000;

const D = {1..numIters} dmapped Cyclic(startIdx=1);

forall i in D do
    writeln(“Hello, world! “, “from iteration “, i, “ of “, numIters);
Layouts and Distributions

Domain Maps fall into two major categories:

*layout*:  
- target a shared memory  
- **examples**: row- and column-major order, tilings, compressed sparse row, space-filling curves

*distribution*:  
- map indices/elements to distributed memories  
- **examples**: Block, Cyclic, Block-Cyclic, Recursive Bisection, …
Sample Distributions: Block and Cyclic

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

```
distributed to
```

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

```
distributed to
```

Copyright 2014 Cray Inc.
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 domain indices and array values
- computation will execute using local processors only
STREAM Triad: Chapel (distributed, blocked)

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

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

A = B + alpha * C;
```
STREAM Triad: Chapel (distributed, cyclic)

\[
\text{const } \text{ProblemSpace} = \{1..m\} \\
\text{dmapped Cyclic(startIdx}=1); \\
\text{var } A, B, C: \text{[ProblemSpace]} \text{ real;}
\]

\[
A = B + \alpha \ast C;
\]
Jacobi Iteration in Chapel (shared memory)

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1},
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do
        const delta = max reduce abs(A[D] - Temp[D]);
        A[D] = Temp[D];
    } while (delta > epsilon);
writeln(A);
```

Copyright 2014 Cray Inc.
Jacobi Iteration in Chapel (distributed memory)

```chapel
config const n = 6,
    epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
```

With this simple change, we specify a mapping from the domains and arrays to locales
Domain maps describe the mapping of domain indices and array elements to *locales*
- specifies how array data is distributed across locales
- specifies how iterations over domains/arrays are mapped to locales

Copyright 2014 Cray Inc.
Jacobi Iteration in Chapel (distributed memory)

```chapel
config const n = 6,
     epsilon = 1.0e-5;

const BigD = {0..n+1, 0..n+1} dmapped Block({1..n, 1..n}),
    D = BigD[1..n, 1..n],
    LastRow = D.exterior(1,0);

var A, Temp : [BigD] real;
A[LastRow] = 1.0;

do {
    forall (i,j) in D do

    const delta = max reduce abs(A[D] - Temp[D]);
    A[D] = Temp[D];
} while (delta > epsilon);

writeln(A);

use BlockDist;
```
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
All Domain Types Support Domain Maps

- dense
- strided
- sparse

- associative
- unstructured
### Domain Map Descriptor Classes

<table>
<thead>
<tr>
<th><strong>Domain Map</strong></th>
<th><strong>Domain</strong></th>
<th><strong>Array</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Represents:</strong> a domain map value</td>
<td><strong>Represents:</strong> a domain</td>
<td><strong>Represents:</strong> an array</td>
</tr>
<tr>
<td><strong>Generic w.r.t.:</strong> index type</td>
<td><strong>Generic w.r.t.:</strong> index type</td>
<td><strong>Generic w.r.t.:</strong> index type, element type</td>
</tr>
<tr>
<td><strong>State:</strong> the domain map’s representation</td>
<td><strong>State:</strong> representation of index set</td>
<td><strong>State:</strong> array elements</td>
</tr>
<tr>
<td><strong>Typical Size:</strong> $\Theta(1)$</td>
<td><strong>Typical Size:</strong> $\Theta(1) \rightarrow \Theta(\text{numIndices})$</td>
<td><strong>Typical Size:</strong> $\Theta(\text{numIndices})$</td>
</tr>
<tr>
<td><strong>Required Interface:</strong></td>
<td><strong>Required Interface:</strong></td>
<td><strong>Required Interface:</strong></td>
</tr>
<tr>
<td>- create new domains</td>
<td>- create new arrays</td>
<td>- (re-)allocation of elements</td>
</tr>
<tr>
<td></td>
<td>- queries: size, members</td>
<td>- random access</td>
</tr>
<tr>
<td></td>
<td>- iterators: serial, parallel</td>
<td>- iterators: serial, parallel</td>
</tr>
<tr>
<td></td>
<td>- domain assignment</td>
<td>- slicing, reindexing, aliases</td>
</tr>
<tr>
<td></td>
<td>- index set operations</td>
<td>- get/set of sparse “zero” values</td>
</tr>
</tbody>
</table>

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:
  - `$CHPL_HOME/modules/dists/*.chpl`
  - `$CHPL_HOME/layouts/*.chpl`
  - `$CHPL_HOME/internal/Default*.chpl`
- Technical notes detailing the domain map interface for implementers:
  - `$CHPL_HOME/doc/technotes/README.dsi`
LULESH Case Study: Domain Maps in Action
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
Fundamental LULESH Concepts/Terminology

- **mesh**
- **nodes**
- **element**
LULESH in Chapel
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
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
Here is some sample representation-independent code

IntegrateStressForElems()

LULESH spec, section 1.5.1.1 (2.)
Because of domain maps, this code is independent of:
- structured vs. unstructured mesh
- shared vs. distributed data
- sparse vs. dense representation
Data Parallelism in LULESH (Structured)

```plaintext
const Elems = \{0..#elemsPerEdge, 0..#elemsPerEdge\},
                  Nodes = \{0..#nodesPerEdge, 0..#nodesPerEdge\};

var determ: [Elems] real;

forall k in Elems { ...determ[k]... }
```

![Diagram of Elems and Nodes]
Data Parallelism in LULESH (Unstructured)

\begin{verbatim}
const Elems = {0..#numElems}, Nodes = {0..#numNodes};

var determ: [Elems] real;

forall k in Elems { ...determ[k]... }
\end{verbatim}

\begin{itemize}
  \item \textbf{Elems}
  \item \textbf{Nodes}
\end{itemize}
Materials Representation

- Not all elements will contain all materials, and some will contain combinations
Materials Representation (Dense)

naïve approach: store all materials everywhere (reasonable for LULESH 1.0, but not in practice)

\[
\text{const } \text{Mat1Elems} = \text{Elems}, \\
\text{Mat2Elems} = \text{Elems};
\]
Materials Representation (Sparse)

improved approach: use sparse subdomains to only store materials where necessary

```plaintext
var Mat1Elems: sparse subdomain(Elems) = enumerateMat1Locs(),
Mat2Elems: sparse subdomain(Elems) = enumerateMat2Locs();
```
const Elems = {0..#numElems},
Nodes = {0..#numNodes};

var determ: [Elems] real;

forall k in Elems { ... }
LULESH Data Structures (distributed, block)

```chapel
const Elems = {0..#numElems} dmapped Block(...),
Nodes = {0..#numNodes} dmapped Block(...);

var determ: [Elems] real;

forall k in Elems { ... }
```

![Diagram showing distributed data structures]

Elems

Nodes

Copyright 2014 Cray Inc.
**LULESH Data Structures (distributed, cyclic)**

```chapel
const Elems = {0..#numElems} dmapped Cyclic(...),
             Nodes = {0..#numNodes} dmapped Cyclic(...);

var determ: [Elems] real;

forall k in Elems { ... }  
```

**Elems**

![Colorful rectangles representing elements](image)

**Nodes**

![Colorful circles representing nodes](image)
Compile-time Reconfiguration in LULESH

```cpp
config param use3DRepresentation = false,
    useBlockDist = (CHPL_COMM != "none"),
    useSparseMaterials = true;
```
Compile-time Reconfiguration in LULESH

```chapel
const ElemSpace = if use3DRepresentation
    then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
    else {0..#numElems},
NodeSpace = if use3DRepresentation
    then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
    else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
    else ElemSpace,
Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
    else NodeSpace;

var elemToNode: [Elems] nodesPerElem*index(Nodes);

const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
    else Elems;

proc MatElemsType type {
    if useSparseMaterials then
        return sparse subdomain(Elems);
    else
        return Elems.type;
}
```
Compile-time Reconfiguration in LULESH

```
const ElemSpace = if use3DRepresentation
    then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
    else {0..#numElems},
NodeSpace = if use3DRepresentation
    then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
    else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
    else ElemSpace,
Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
    else NodeSpace;

var elemToNode: [Elems] nodesPerElem*index(Nodes);

const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
    else Elems;

proc MatElemsType type {
    if useSparseMaterials then
        return sparse subdomain(Elems);
    else
        return Elems.type;
}
```
Compile-time Reconfiguration in LULESH

```chapel
const ElemSpace = if use3DRepresentation
    then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
    else {0..#numElems},
NodeSpace = if use3DRepresentation
    then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
    else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
else ElemSpace,
Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
else NodeSpace;

var elemToNode: [Elems] nodesPerElem*index(Nodes);

const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
else Elems;

proc MatElemsType type {
    if useSparseMaterials then
        return sparse subdomain(Elems);
    else
        return Elems.type;
}
```

potentially distributed domains for elements and nodes
Compile-time Reconfiguration in LULESH

```chapel
const ElemSpace = if use3DRepresentation
    then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
    else {0..#numElems},
NodeSpace = if use3DRepresentation
    then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
    else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
    else ElemSpace,
Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
    else NodeSpace;

var elemToNode: [Elems] nodesPerElem*index(Nodes);

const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
    else Elems;

proc MatElemsType type {
    if useSparseMaterials then
        return sparse subdomain(Elems);
    else
        return Elems.type;
}
```

nodes adjacent to each element
const ElemSpace = if use3DRepresentation then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge} else {0..#numElems},
NodeSpace = if use3DRepresentation then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge} else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace) else ElemSpace,
Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace) else NodeSpace;

var elemToNode: [Elems] nodesPerElem*index(Nodes);

const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems() else Elems;

proc MatElemsType type {
    if useSparseMaterials then
        return sparse subdomain(Elems);
    else
        return Elems.type;
}
const ElemSpace = if use3DRepresentation
                      then {0..#elemsPerEdge, 0..#elemsPerEdge, 0..#elemsPerEdge}
                      else {0..#numElems},
                NodeSpace = if use3DRepresentation
                            then {0..#nodesPerEdge, 0..#nodesPerEdge, 0..#nodesPerEdge}
                            else {0..#numNodes};

const Elems = if useBlockDist then ElemSpace dmapped Block(ElemSpace)
               else ElemSpace,
                Nodes = if useBlockDist then NodeSpace dmapped Block(NodeSpace)
                         else NodeSpace;

var elemToNode: [Elems] nodesPerElem*index(Nodes);

const MatElems: MatElemsType = if sparseMaterials then enumerateMatElems()
                              else Elems;

proc MatElemsType type {
  if useSparseMaterials then
    return sparse subdomain(Elems);
  else
    return Elems.type;
}
Additional Data Parallel Notes
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.
Motivation for Leader-Follower Iterators

Q: How are parallel loops implemented?
   (how many tasks? executing where? how are iterations divided up?)

   \[
   \text{forall } a \in A \{ \ldots \}
   \]

Q2: What about zippered data parallel operations?
   (how to reconcile potentially conflicting parallel implementations?)

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

A: Via Leader-Follower Iterators…

- see locality section
- like domain maps, can be written by users
  - gives control over crucial scheduling and placement decisions
Chapel Promotion Semantics

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

\[ A = B; \]

…is equivalent to:

\[
\text{forall } (a,b) \text{ in zip}(A,B) \text{ do } \newline \quad a = b;
\]
Implication of Zippered Promotion Semantics

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

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

...rather than operator-wise.

\[ A = B + \alpha \times C; \quad \Rightarrow \quad T_1 = \alpha \times C; \]
\[ A = B + T_1; \]
Implication of Zippered Promotion Semantics

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

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

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

\[ A[D] = A[D\text{-one}] + A[D\text{+one}]; \quad \Rightarrow \quad \text{forall } (a_1, a_2, a_3) \text{ in } (A[D], A[D\text{-one}], A[D\text{+one}]) \text{ do } a_1 = a_2 + a_3; \]

\[ \Rightarrow \text{Read/write race!} \]
Implication of Zippered Promotion Semantics

Whole-array operations are implemented element-wise…

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

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

⇒ No temporary arrays required by semantics
⇒ No surprises in memory requirements
⇒ Friendlier to cache utilization

⇒ Differs from traditional array language semantics

\[ B[D] = A[D\text{-one}] + A[D+one]; \]

\[ \Rightarrow \text{forall } (b, a_2, a_3) \text{ in } (B[D], A[D\text{-one}], A[D+one]) \text{ do } b = a_2 + a_3; \]
Questions about Data Parallelism in Chapel?
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.
http://chapel.cray.com  chapel_info@cray.com  http://sourceforge.net/projects/chapel/