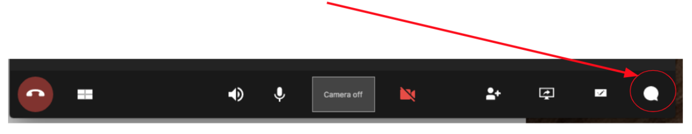
Computing graphs on an HPC cluster: working with distributed unstructured data in Chapel

ALEX RAZOUMOV
alex.razoumov@westgrid.ca

To ask questions

- Websteam: email **info@westgrid.ca**
- Vidyo: use the GROUP CHAT to ask questions



- Please mute your microphone unless you have a question
- Feel free to ask questions via audio at any time

Data
- Unstructured
  - Trees, graphs, . . .
  - Adaptive mesh refinement (AMR) with a space-filling curve
  - Unstructured meshes, e.g. 2D polygonal or 3D tetrahedral
  - Sparse linear algebra (via dynamically allocated subsets of dense arrays)
- Large, a.k.a. distributed across multiple nodes

Chapel programming language

https://chapel-lang.org

- High-level parallel language
  - "Python for parallel programming"
  - much easier to use and learn than MPI
  - abstractions for data and task parallelism
  - optimization for data-driven placement of subcomputations
  - granular ("multi-resolution") design: can bring your code closer to machine level if needed
  - everything you can do in MPI (and OpenMP!), you should be able to do in Chapel
- Focus on performance
  - compiled language; simple Chapel codes perform as fast as optimized C/C++/Fortran codes
  - very complex/production Chapel codes reported to run at ∼70% performance of a similar well-tuned MPI code (room to improve)

Chapel programming language (cont.)

`https://chapel-lang.org`

- Perfect language for learning parallel programming for beginners
- Open-source
  - can compile on all Unix-like platforms
  - precompiled for MacOS (single-locale via Homebrew)
  - Docker image `http://dockr.ly/2vJbi06` (simulates multi-locale environment on your laptop)
- Fairly small community at the moment: too few people know/use Chapel $\iff$ too few libraries
  - you *can* load functions written in other languages

How to compile/run Chapel codes on CC clusters

https://docs.computecanada.ca/wiki/Chapel

- Running single-locale Chapel interactively (same version everywhere)

```
module load gcc chapel-single

salloc --time=0:30:0 --ntasks=1 --mem-per-cpu=3500 --account=def-someprof
chpl test.chpl -o test
./test

 salloc --time=0:30:0 --ntasks=1 --cpus-per-task=3 --mem-per-cpu=3500 --account=...
 chpl test.chpl -o test
 ./test
```

- Running multi-locale Chapel interactively (1.19 on Cedar, 1.17 for now on Graham)

```
. /home/razoumov/startMultiLocale.sh

salloc --time=0:30:0 --nodes=4 --cpus-per-task=3 --mem-per-cpu=3500 --account=...
chpl probeLocales.chpl -o probeLocales
./probeLocales -nl 4
```

  - run production codes via batch jobs
  - 1.19-compiled codes on Graham currently do not run properly (work in progress)
  - multi-locale Chapel on Béluga will be compiled shortly

Domains

A multi-dimensional, rectangular, bounded collection of integer indices:

```
config const n = 5;
var tenIndices: domain(1) = {1..10};
var mesh: domain(2) = {1..n, 1..n};
var thirdDim: range = 1..16;
var threeDimensions: domain(3) = {thirdDim, 1..10, 5..10};

for m in mesh do
  write(m, ' ');
writeln();
```

```
(1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (2, 1) (2, 2) (2, 3) (2, 4)
(2, 5) (3, 1) (3, 2) (3, 3) (3, 4) (3, 5) (4, 1) (4, 2) (4, 3)
(4, 4) (4, 5) (5, 1) (5, 2) (5, 3) (5, 4) (5, 5)
```

Arrays on domains

1. You can define arrays on top of domains:

```
config const n = 5;
var mesh: domain(2) = {1..n, 1..n};
var A: [mesh] string;

for m in mesh do
  A[m] = '' + m[1] + m[2];

writeln(A);
```

```
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
41 42 43 44 45
51 52 53 54 55
```

Distributed domains and arrays

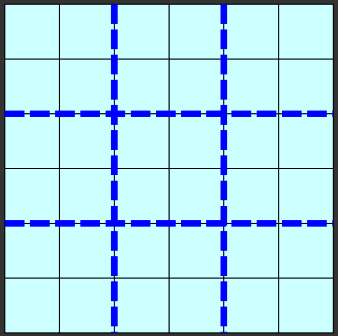
Running this example on 4 nodes

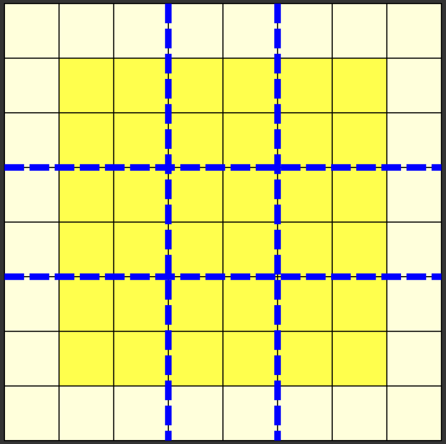
② Domains, along with arrays on top of them, can be distributed across locales:

```
use BlockDist;
config const n = 5;
const mesh: domain(2) = {1..n, 1..n};
const distrib: domain(2) dmapped Block(boundingBox=mesh) = mesh;

var A: [distrib] string;
forall a in A do
  a = '%i'.format(a.locale.id+1) + '-' + here.name[1..5] + '  ';

writeln(A);
```

```
1-node1   1-node1   1-node1   2-node2   2-node2
1-node1   1-node1   1-node1   2-node2   2-node2
1-node1   1-node1   1-node1   2-node2   2-node2
3-node3   3-node3   3-node3   4-node4   4-node4
3-node3   3-node3   3-node3   4-node4   4-node4
```

Block distribution





Domains (cont.)

- 10 standard distributions in Chapel
  `https://chapel-lang.org/docs/modules/layoutdist.html`
  - some standard distributions are quite flexible with mapping (can define your own)
- For expert programmers, Chapel provides tools for creating custom distributions
- An array element is always stored on the same locale as its defining domain index
- Computations on distributed arrays try to follow data
  - often you have a mixture of locales in a single computation (e.g. a distributed finite difference stencil)
  - Chapel tries to minimize communication and at the same time load-balance computation

Sparse domains and arrays

A sparse domain is a dynamic, initially empty subset of a rectangular domain:

```
config var n = 5;
const mesh = {1..n, 1..n};         // 2D rectangular domain
var SD: sparse subdomain(mesh);    // initially an empty subset of `mesh` indices
var A: [SD] real;                  // sparse real array on top of the sparse domain

writeln("Initially, SD = ", SD);
writeln("Initially, A = ", A);

proc writeSparseArr() {
  writeln("A in dense representation:");
  for (i,j) in mesh {
    write(A(i,j), "  ");
    if j == n then writeln();
  }
  writeln();
}

writeSparseArr();
```

```
Initially, SD = {}
Initially, A = 

A in dense representation:
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```

Sparse domains and arrays (code continues)

```
A.IRV = 1e-3;    // change the default Implicitly Replicated Value

SD += (1,n);     // add corners to the sparse domain
SD += (n,n);
SD.add((1,1));   // alternative syntax
SD += (n,1);

A[1,1] = 100;

writeln("With corners, SD = ", SD);
writeln("With corners, A = ", A);

writeSparseArr();
```

```
With corners, SD = {
 (1, 1) (1, 5)
 (5, 1) (5, 5)
}

With corners, A = 100.0 0.001
                  0.001 0.001

A in dense representation:
 100.0  0.001  0.001  0.001  0.001
 0.001  0.001  0.001  0.001  0.001
 0.001  0.001  0.001  0.001  0.001
 0.001  0.001  0.001  0.001  0.001
 0.001  0.001  0.001  0.001  0.001
```

Sparse domains and arrays (code continues)

```
for (i,j) in mesh {
  if SD.member(i,j) then
    write("* "); //   (i,j) is a member in the sparse index set
  else
    write(". "); //   (i,j) is not a member in the sparse index set
  if (j == n) then writeln();
}

var sparseSum = + reduce A;
writeln("sparse elements sum = ", sparseSum);

var denseSum = + reduce [ij in mesh] A(ij);
writeln("dense elements sum = ", denseSum);
```

```
* . . . *
. . . . .
. . . . .
. . . . .
* . . . *

sparse elements sum = 100.003

dense elements sum = 100.024
```

Sparse domains and arrays (code continues)

```
SD.clear();          // empty the sparse index set (the domain and all its arrays)
A.IRV = 0.0;         // reset the Implicitly Replicated Value

for i in 1..n do
  SD += (i,i);       // add the main diagonal to the sparse domain

[(i,j) in SD] A(i,j) = i + j;

writeln("Now, SD = ", SD);
writeln("Now, A = ", A);
writeSparseArr();
```

```
Now, SD = {
 (1, 1) (2, 2) (3, 3) (4, 4) (5, 5)
}

Now, A = 2.0 4.0 6.0 8.0 10.0

A in dense representation:
 2.0  0.0  0.0  0.0  0.0
 0.0  4.0  0.0  0.0  0.0
 0.0  0.0  6.0  0.0  0.0
 0.0  0.0  0.0  8.0  0.0
 0.0  0.0  0.0  0.0  10.0
```

Sparse domains and arrays (code continues)

```
iter antiDiag(n) {
  for i in 1..n do
    yield (i, n-i+1);
}

SD = antiDiag(n);

[(i,j) in SD] A(i,j) = i + j;

writeln("`antiDiag` SD = ", SD);
writeln("`antiDiag` A = ", A);
writeSparseArr();
```

```
`antiDiag` SD = {
 (1, 5) (2, 4) (3, 3) (4, 2) (5, 1)
}

`antiDiag` A = 6.0 6.0 6.0 6.0 6.0

A in dense representation:
 0.0  0.0  0.0  0.0  6.0
 0.0  0.0  0.0  6.0  0.0
 0.0  0.0  6.0  0.0  0.0
 0.0  6.0  0.0  0.0  0.0
 6.0  0.0  0.0  0.0  0.0
```

Distributed sparse domains and arrays

Running this example on 4 nodes

```
use BlockDist;

config const n = 5;
const D = {1..n, 1..n} dmapped Block({1..n, 1..n});  // distributed dense index set
var SD: sparse subdomain(D);                         // distributed sparse subset, initially empty
var A: [SD] int;                                     // distributed sparse array

for i in 1..n do {    // populate the sparse index set
  SD += (i,i);        // main diagonal
  SD += (1,i);        // first row
  SD += (i,n);        // last column
 }

// assign the sparse array elements in parallel
forall a in A do
  a = here.id + 1;

// print a dense view of the array
writeln('A =');
for i in 1..n {
  for j in 1..n do
    write(A[i,j], " ");
  writeln();
}
```

```
A =
 1 1 1 2 2
 0 1 0 0 2
 0 0 1 0 2
 0 0 0 4 4
 0 0 0 0 4
```

Using sparse arrays

Details at `https://chapel-lang.org/docs/modules/packages/LinearAlgebra.html`

```
use LinearAlgebra;
```

- When compiling Chapel codes, you can optionally link to external BLAS or LAPACK, if you need their functions

```
module load openblas/0.3.4
chpl --fast test.chpl -o test -lopenblas
```

- Currently implemented LinearAlgebra functions are still in their infancy
  - only basic matrix operations, eigenvalues and eigenvectors
  - no inverse, no linear system solve ...
- Currently works on local/distributed dense arrays and local sparse arrays
  - ... but not on distributed sparse arrays
- Can call external C/C++ linear solvers, but these will not use Chapel parallelism for computation ...
- Can work directly with distributed sparse arrays in parallel on multiple nodes/cores outside of the linear algebra libraries

Associative domains and arrays

- Recall the definition of a domain in Chapel: *multi-dimensional, rectangular, bounded collection of integer indices*
- **Associative domain** is a 1D finite set of indices (or more precisely, keys) of any type
  - associated domains are similar to Python's sets (unordered, unique indices)
  - associated domains with arrays on top are similar to Python's dictionaries
  - starting with v1.19, can be distributed across multiple nodes

```
var days: domain(string);  // a domain (set) whose indices are strings
var maxTemp: [days] real;  // array of reals

days += 'Mon';             // add a domain index
days += 'Tue';             // another
days.add('Wed');           // another
maxTemp['Mon'] = 25;       // add an array value

writeln('domain = ', days);
writeln('maxTemp = ', maxTemp);

var week = {'Mon', 'Tue', 'Wed', 'Thu', 'Fri',
            'Sat', 'Sun'};      // an associative domain is also a set!
```

```
domain = {Mon, Wed, Tue}
maxTemp = 25.0 0.0 0.0
```

Opaque domains and arrays

- **Opaque domain** is a special case of associative domain whose indices have no values (anonymous) – designed to support unstructured data such as graphs
- As of 1.19, cannot be mapped to locales
  - likely to be implemented in future versions
  - for this reason, we won't be using them today

```
var people: domain(opaque);    // opaque domain, its indices have no values
var name: [people] string;     // array on top of this domain
var connection: [people] index(people);     // another array on top of it

for i in 1..5 {
  var newPerson = people.create(); // inferred to be of type index(people)
  name[newPerson] = 'name%i'.format(i);
}
use Random;
var myRandNums = makeRandomStream(real, seed=314159265, algorithm=RNG.NPB);
for i in people do
  for j in people do
    if i != j && myRandNums.getNext() > 0.5 && connection[i] == nil then
      connection[i] = j;

for person in people do   // no particular order
  writeln(name[person], ' -> ', name[connection[person]]);
```

```
name3 -> name5
name5 -> name1
name2 -> name3
name1 -> name5
name4 -> name1
```

Distributed associative domains and arrays

Running this example on 4 nodes

- Starting with v1.19, associative domains can be distributed across multiple nodes
- Each domain index is mapped to a locale based upon a hash of this index
- As always with domains, all arrays on top of these will also be mapped to locales

```
use HashedDist;

var D: domain(string) dmapped Hashed(idxType=string);   // a distributed associative
D += 'hello';                                           // ... domain (set) of strings
for i in 1..10 do
  D += '%02i'.format(i);
writeln(D);

var A: [D] int; // a distributed associative int array
forall a in A do
  a = a.locale.id + 1;

for i in 1..numLocales {
  write('node ', i, ': ');
  forall (key, value) in zip(D, A) {
    if value == i then write(key, ' ');
  }
  writeln();
}
```

```
{06, 01, hello} {07, 03} {}
{10, 05, 04, 02, 09, 08}
```

```
node 1: 06 01 hello
node 2: 03 07
node 3:
node 4: 10 02 05 04 09 08
```

Building a local graph with associative arrays

```
var vertices, edges: domain(string);                      // two domains with string indices
var degree: [vertices] int, weight: [vertices] real;      // two arrays for each vertex
var from, to: [edges] index(vertices);                    // for each edge two vertices

use Random;
var myRandNums = makeRandomStream(real, seed=314159265, algorithm=RNG.NPB);

config const numVertices = 8;    // allocate vertices, assign them names and random weights
for i in 1..numVertices {
  var thisVertex = '%03i'.format(i);
  vertices += thisVertex;
  weight[thisVertex] = myRandNums.getNext();
}

for i in vertices do    // iterate over all pairs of vertices
  for j in vertices do
    if (myRandNums.getNext() > 0.5 && i != j) { // new directional edge
      var thisEdge = i + '-' + j;
      edges += thisEdge;
      degree[i] += 1;        degree[j] += 1;
      from[thisEdge] = i;    to[thisEdge] = j;
    }

writeln(numVertices, ' vertices: ', vertices);
writeln(edges.shape[1], ' edges: ', edges);
```

```
8 vertices: {005, 004, 007, 006, 001,
             003, 002, 008}
20 edges: {008-002, 008-003, 008-006,
           007-005, 003-007, 005-008,
           006-008, 002-008, 006-007,
           005-007, 005-003, 001-003,
           005-002, 001-005, 002-007,
           002-004, 001-006, 002-005,
           002-003, 004-006}
```

Distributing this graph

Running this example on 4 nodes

```
-var vertices, edges: domain(string);                  // two domains with string indices
+use HashedDist;
+var vertices, edges: domain(string) dmapped Hashed(idxType=string);
 var degree: [vertices] int, weight: [vertices] real;   // two arrays for each vertex
 var from, to: [edges] index(vertices);                 // for each edge two vertices
```

```
8 vertices: {007, 003}  {008}  {005, 001, 002}  {004, 006}
20 edges: {007-008, 007-004, 001-005, 005-008}
          {004-006, 001-007, 006-002, 002-008, 003-005}
          {007-006, 004-002, 006-004, 006-005}
          {004-007, 001-002, 007-002, 004-003, 008-007, 005-006, 004-008}
```

Distributing this graph

Running this example on 4 nodes

```
-var vertices, edges: domain(string);                      // two domains with string indices
+use HashedDist;
+var vertices, edges: domain(string) dmapped Hashed(idxType=string);
 var degree: [vertices] int, weight: [vertices] real;      // two arrays for each vertex
 var from, to: [edges] index(vertices);                    // for each edge two vertices
```

```
8 vertices: {007, 003}  {008}  {005, 001, 002}  {004, 006}
20 edges: {007-008, 007-004, 001-005, 005-008}
          {004-006, 001-007, 006-002, 002-008, 003-005}
          {007-006, 004-002, 006-004, 006-005}
          {004-007, 001-002, 007-002, 004-003, 008-007, 005-006, 004-008}
```

Check where the edge 001-005 is stored ...

Custom domain-to-locale mapping

- When both vertices are local, the edge should be local too
  $\Rightarrow$ use a custom mapper from domain indices to locales
- Our implementation
  - vertices are distributed by their number
  - edges are distributed by their first vertex's number

Custom domain-to-locale mapping (cont.)

Running this example on 4 nodes

```
-var vertices, edges: domain(string) dmapped Hashed(idxType=string);
+
+record vertexMapper {
+  proc this(idx:string, targetLocs: [?D] locale) : D.idxType {
+    const numLocales = targetLocs.domain.size;
+    return (idx:int) % numLocales;    // vertex number % number of locales
+  }
+}
+
+record edgeMapper {
+  proc this(idx:string, targetLocs: [?D] locale) : D.idxType {
+    const numLocales = targetLocs.domain.size;
+    return (idx[1..3]:int) % numLocales;    // # of the first vertex in an edge % number of locales
+  }
+}
+
+var vertices: domain(string) dmapped Hashed(idxType=string, mapper=new vertexMapper());
+var edges: domain(string) dmapped Hashed(idxType=string, mapper=new edgeMapper());
 var degree: [vertices] int, weight: [vertices] real;    // two arrays for each vertex
 var from, to: [edges] index(vertices);                  // for each edge two vertices
```

```
8 vertices: {004, 008}  {005, 001}  {006, 002}  {007, 003}
20 edges: {004-005, 004-007, 008-001, 004-002, 004-003}
          {001-003, 001-005, 005-004}
          {002-005, 006-004, 006-002, 006-001}
          {007-008, 007-005, 007-004, 007-003, 007-002, 003-002, 003-001, 003-007}
```

Summary

- Chapel supports less structured / dynamic data
  - sparse domains and arrays: can be mapped to locales
  - associative domains and arrays: can be mapped to locales as of v1.19
  - opaque domains and arrays: distribution across locales currently not implemented
- `HashedDist` supports custom index mapping to locales
- Chapel on Compute Canada systems
  `https://docs.computecanada.ca/wiki/Chapel`
- The official page `https://chapel-lang.org`

Questions?