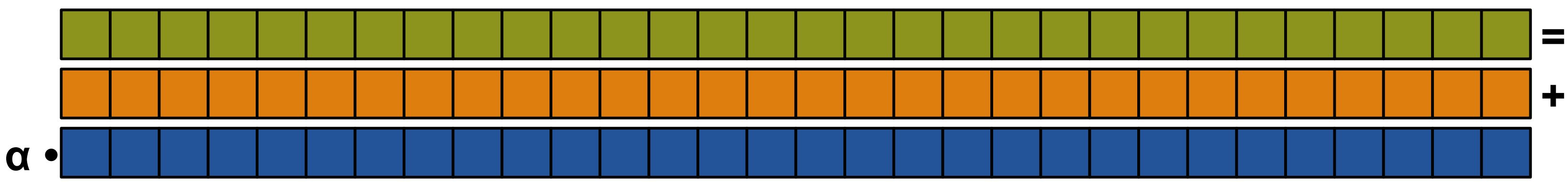
# Data Parallelism with Locality:
# Domain Maps / Distributions (4x3 slides)

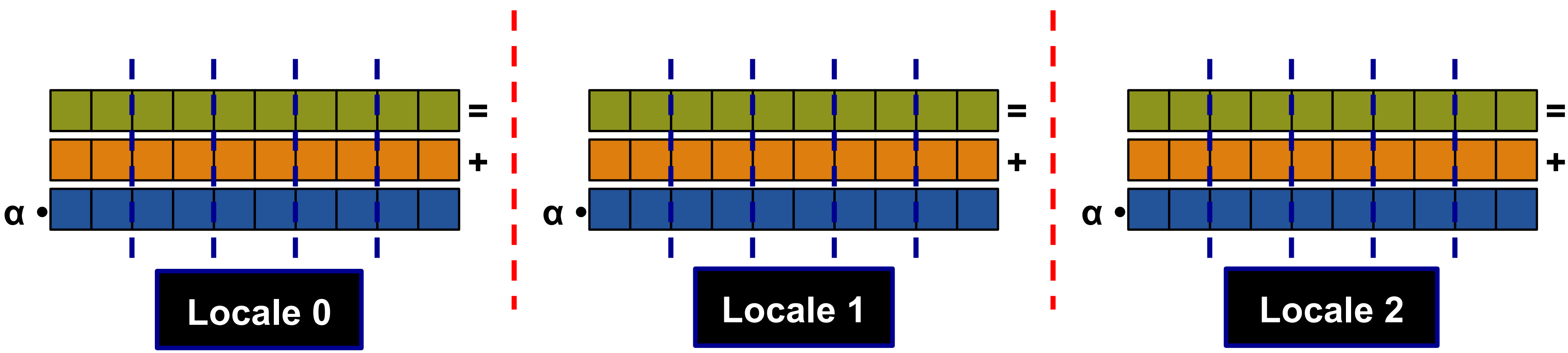
# Domain Maps

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




```
A = B + alpha * C;
```

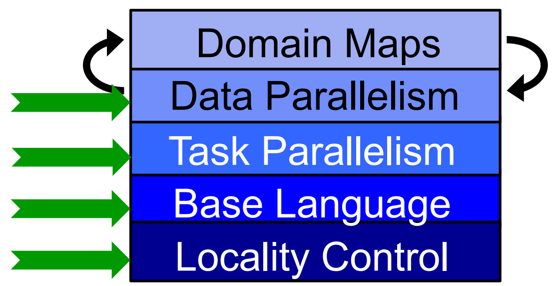
…to the target locales’ memory and processors:



# 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

# Domain Map Roles

**They define data storage:**

- Mapping of domain indices and array elements to locales
- Layout of arrays and index sets in each locale's memory

**…as well as operations:**

- random access, iteration, slicing, reindexing, rank change, …
- the Chapel compiler generates calls to these methods to implement the user's array operations

# Layouts and Distributions

## Domain Maps fall into two major categories:

***layouts:***

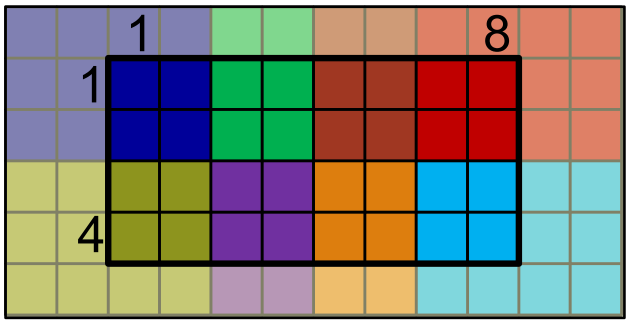
- e.g., a desktop machine or multicore node
- **examples:** row- and column-major order, tilings, compressed sparse row, space-filling curves

***distributions:***

- e.g., a distributed memory cluster or supercomputer
- **examples:** Block, Cyclic, Block-Cyclic, Recursive Bisection, …

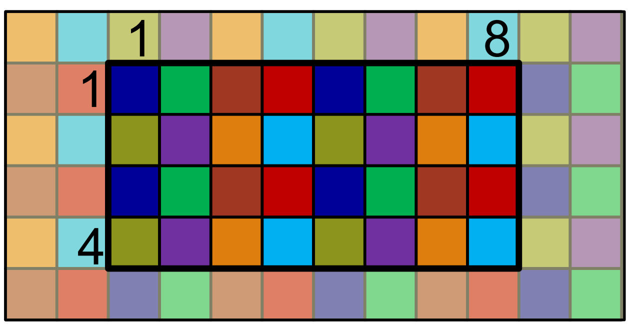
# Sample Distributions: Block and Cyclic

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



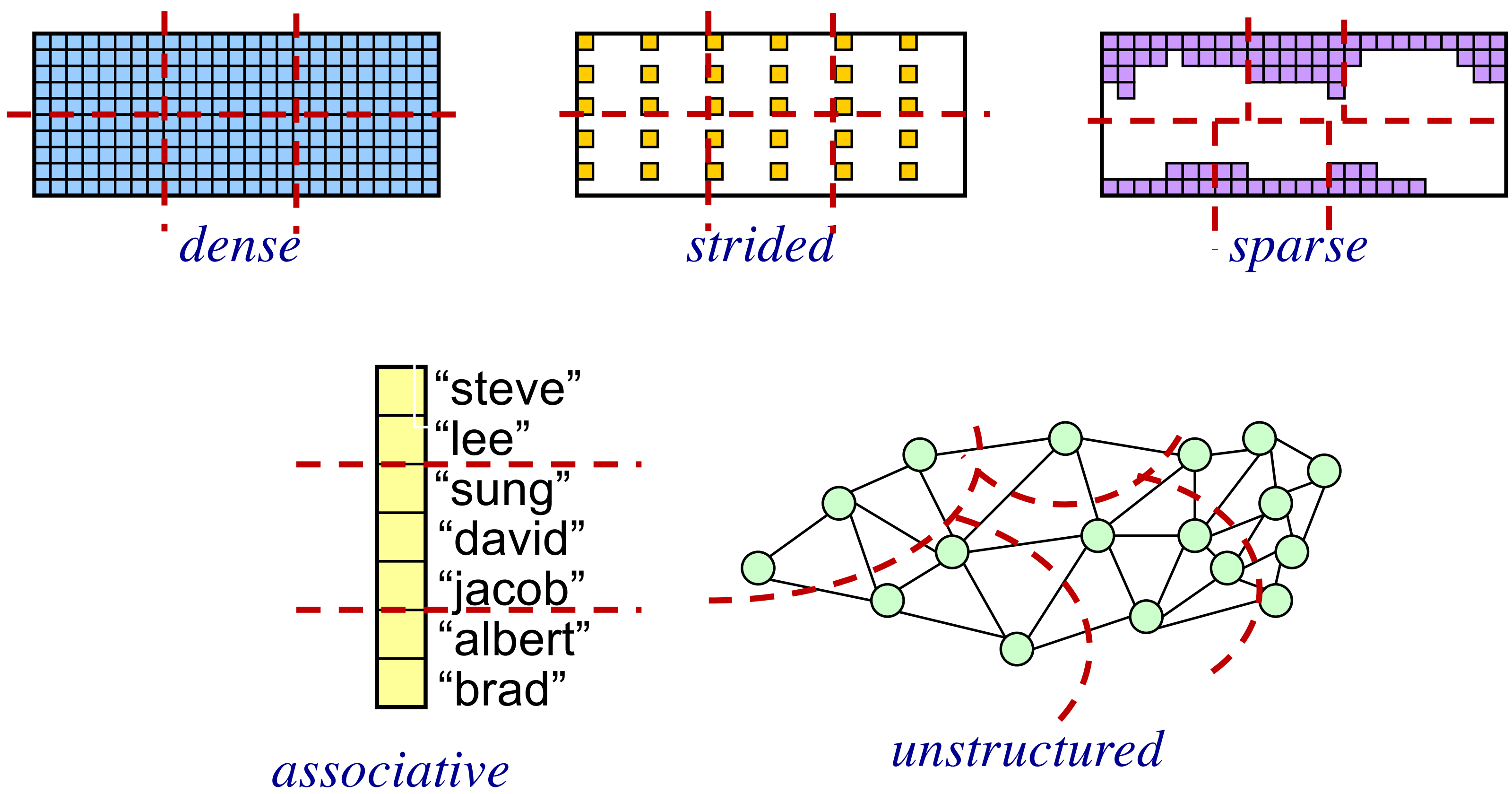

*distributed to*


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




*distributed to*

# All Domain Types Support Domain Maps



# Data Parallelism Implementation Qs

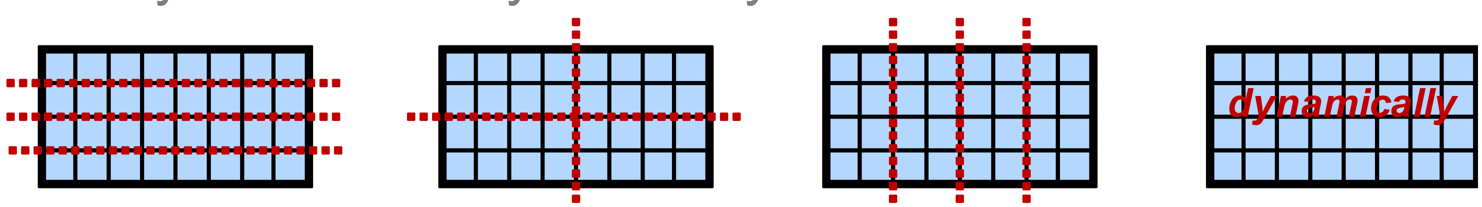
## Q1: How are arrays laid out in memory?

- 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?

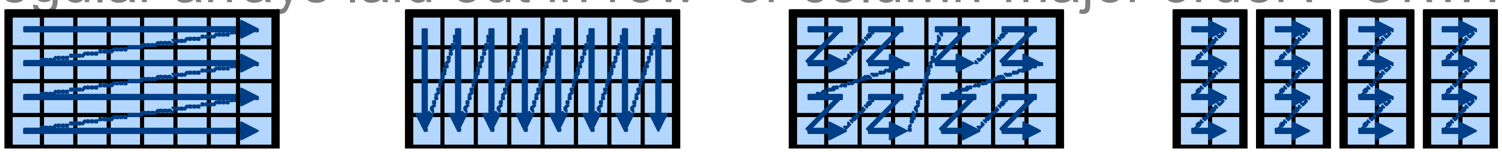
- Completely local to one locale?  Or distributed?
- If distributed… In a blocked manner?  cyclically?  block-cyclically?  recursively bisected?  dynamically rebalanced?  …?



# 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

# 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;
```

By default, domains and their arrays are mapped to a single locale.
Any data parallelism over such domains/ arrays will be executed by the cores on that locale.
Thus, this is a shared-memory parallel program.

```
      Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

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

writeln(A);
```

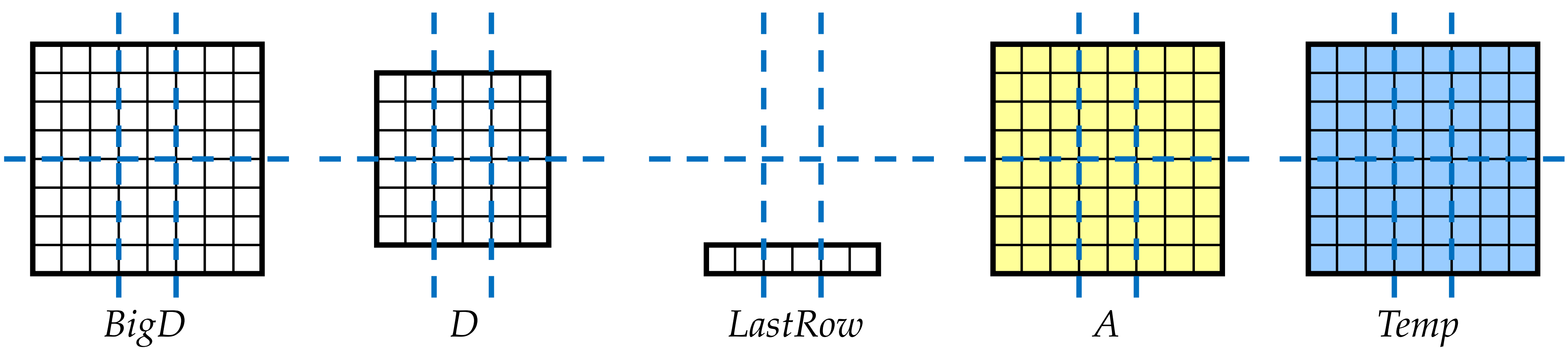
# Jacobi Iteration in Chapel

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

const BigD = {0..n+1, 0..n+1} 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

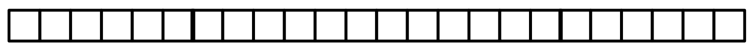


# Jacobi Iteration in Chapel

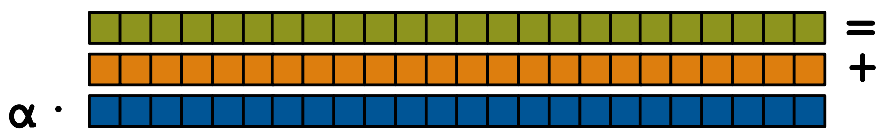
```
config const n = 6,
             epsilon = 1.0e-5;

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
    Temp[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1]) / 4;

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

writeln(A);

use BlockDist;
```

# STREAM Triad in Chapel

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



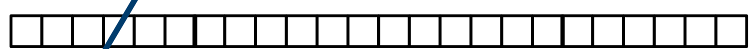
```
var A, B, C: [ProblemSpace] real;
```



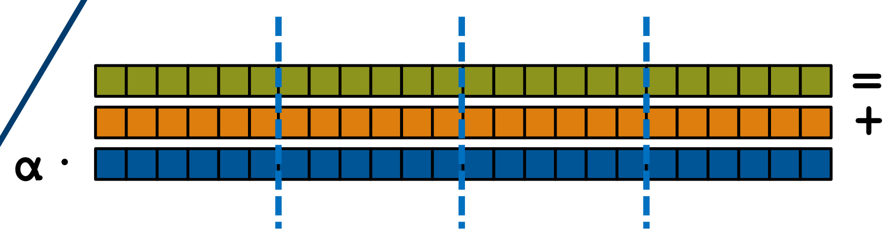

```
A = B + alpha * C;
```

# STREAM Triad in Chapel (multicore)

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



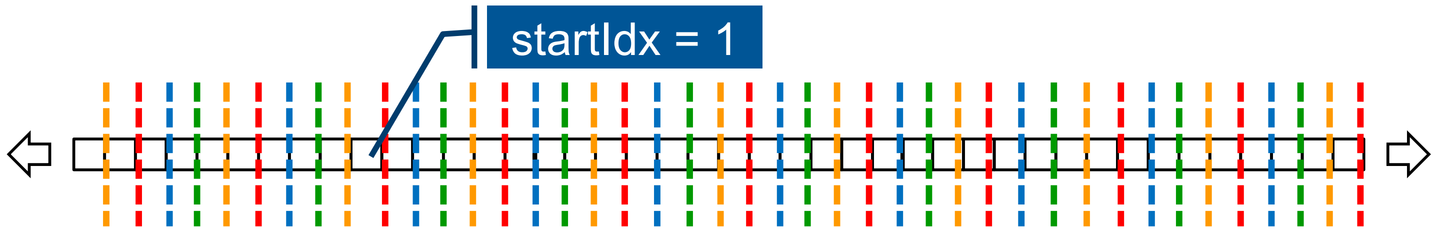
```
var A, B, C: [ProblemSpace] real;
```



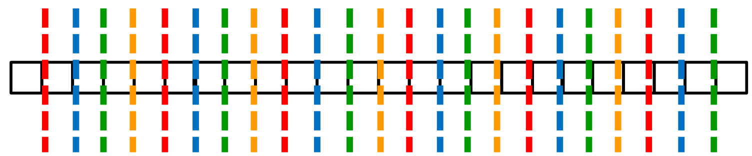

```
A = B + alpha * C;
```

No domain map specified ⇒ use default layout
- current locale owns all domain indices and array values
- computation will execute using local processors only

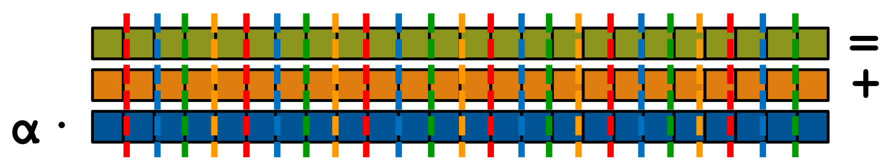
# STREAM Triad in Chapel (multilocale, cyclic)




```
const ProblemSpace = {1..m}
                         dmapped Cyclic(startIdx=1);
```

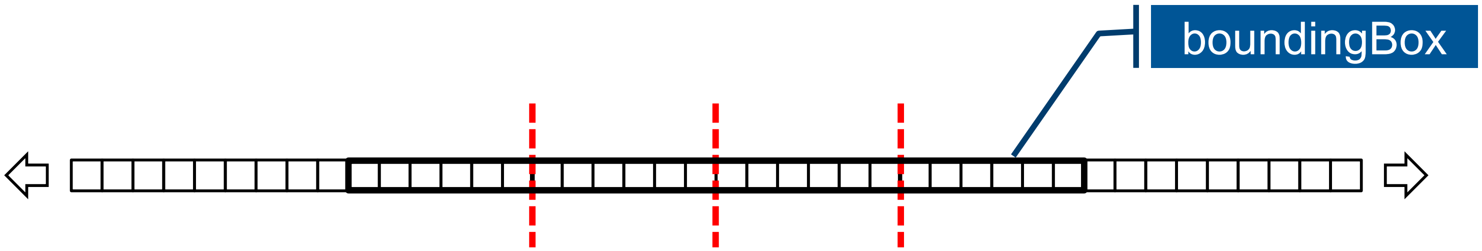


```
var A, B, C: [ProblemSpace] real;
```

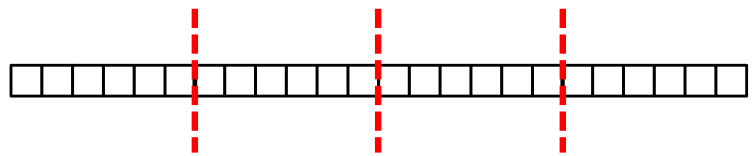



```
A = B + alpha * C;
```

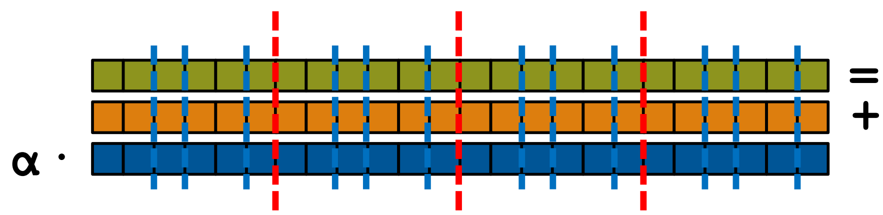
# STREAM Triad in Chapel (multilocale, blocked)




```
const ProblemSpace = {1..m}
                 dmapped Block(boundingBox={1..m});
```



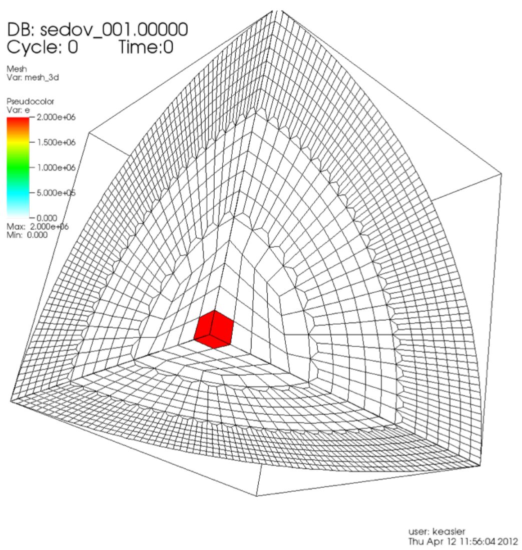
```
var A, B, C: [ProblemSpace] real;
```

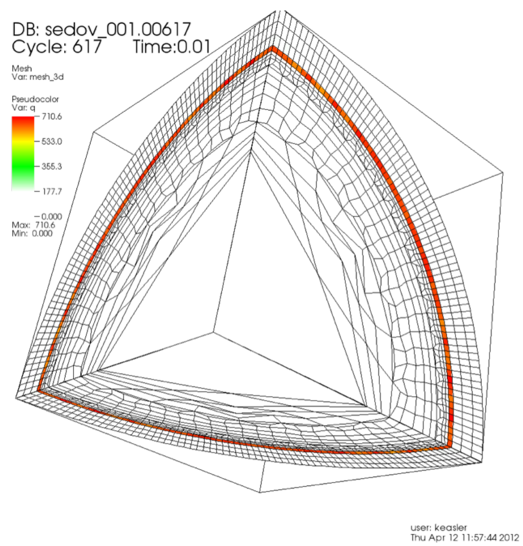


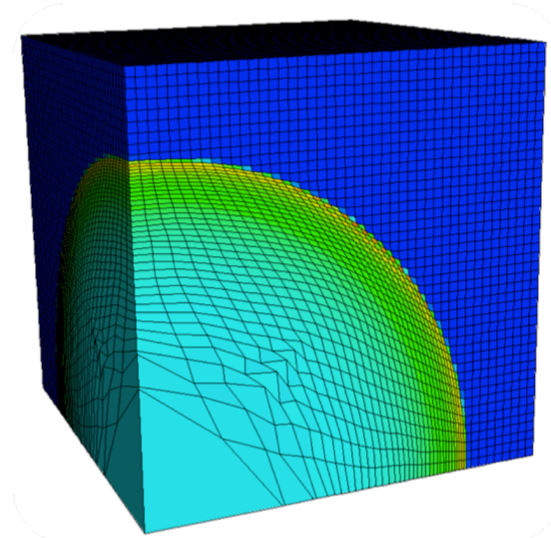

```
A = B + alpha * C;
```

# LULESH: a DOE Proxy Application

**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

# 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)**

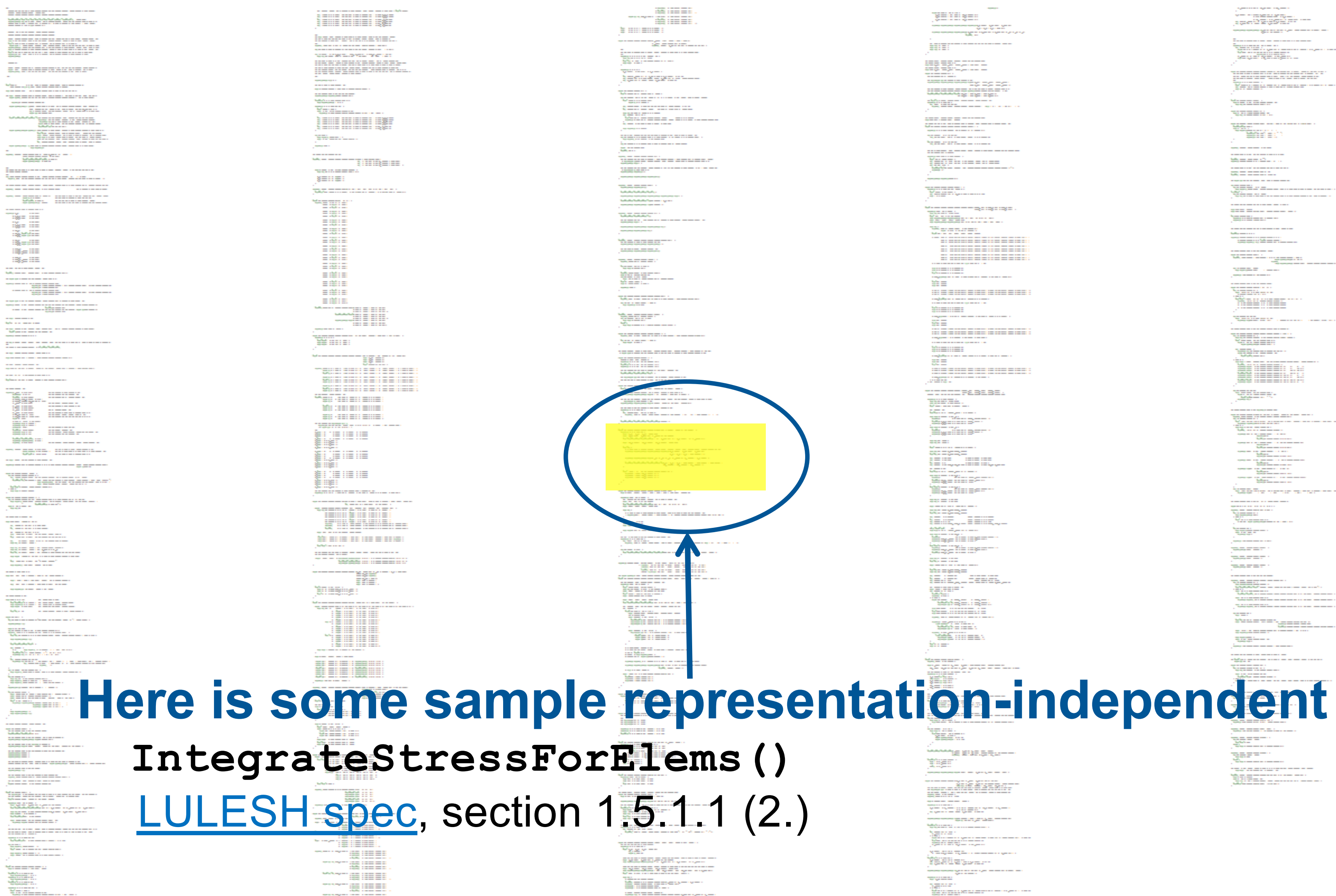
This can be found in the Chapel release under examples/benchmarks/lulesh/*.chpl

# LULESH in Chapel

**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

# LULESH in Chapel



**Here is some sample representation-independent code**

`IntegrateStressForElems()`

LULESH spec, section 1.5.1.1 (2.)

# Representation-Independent Physics

```
proc IntegrateStressForElems(sigxx, sigyy, sigzz, determ) {
 forall k in Elems {
  var b_x, b_y, b_z: 8*real;
  var x_local, y_local, z_local: 8*real;
  localizeNeighborNodes(k, x, x_local, y, y_local, z, z_local);

  var fx_local, fy_local, fz_local: 8*real;

  local {
   /* Volume calculation involves extra work for numerical consistency. */
   CalcElemShapeFunctionDerivatives(x_local, y_local, z_local,
                                    b_x, b_y, b_z, determ[k]);

   CalcElemNodeNormals(b_x, b_y, b_z, x_local, y_local, z_local);

   SumElemStressesToNodeForces(b_x, b_y, b_z, sigxx[k], sigyy[k], sigzz[k],
                               fx_local, fy_local, fz_local);
  }

  for (noi, t) in elemToNodesTuple(k) {
   fx[noi].add(fx_local[t]);
   fy[noi].add(fy_local[t]);
   fz[noi].add(fz_local[t]);
  }
 }
}
```

parallel loop over elements

collect nodes neighboring this element; localize node fields

update node forces from element stresses

Because of domain maps, this code is independent of:
- structured vs. unstructured mesh
- shared vs. distributed data
- sparse vs. dense representation

# 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:**

- Documentation of current domain maps:
  http://chapel.cray.com/docs/latest/modules/layoutdist.html
- Technical notes detailing the domain map interface for implementers:
  http://chapel.cray.com/docs/latest/technotes/dsi.html

# 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, …

# 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**

# Any Questions about Domain Maps?

# Overarching Example:

# Smith-Waterman Algorithm for Sequence Alignment

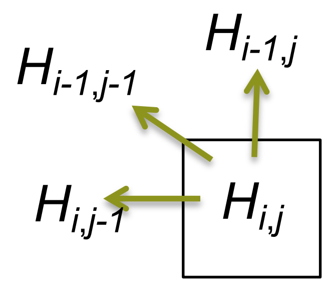
# Smith-Waterman

**Goal: Determine the similarities/differences between two protein sequences/nucleotides.**

- e.g., ACACACTA and AGCACACA*

**Basis of Computation: Defined via a recursive formula:**

$$H(i,0) = 0$$
$$H(0,j) = 0$$
$$H(i,j) = f(H(i\text{-}1, j\text{-}1), H(i\text{-}1, j), H(i, j\text{-}1))$$




**Caveat:** *This is a classic, rather than cutting-edge sequence alignment algorithm, but it illustrates an important parallel paradiagm: wavefront computation*

*Source of running example: Wikipedia

# Smith-Waterman

## Naïve Task-Parallel Approach:

```
proc computeH(i, j) {
  if (i == 0 || j == 0) then
    return 0;
  else
    var h_NW, h_N, h_W: int;

    cobegin {
      h_NW = computeH(i-1, j-1);
      h_N  = computeH(i-1, j);
      h_W  = computeH(i,   j-1);
    }

    return f(h_NW, h_N, h_W);
}
```

Note: Recomputes most subexpressions redundantly

This is a case for dynamic programming!

# Smith-Waterman

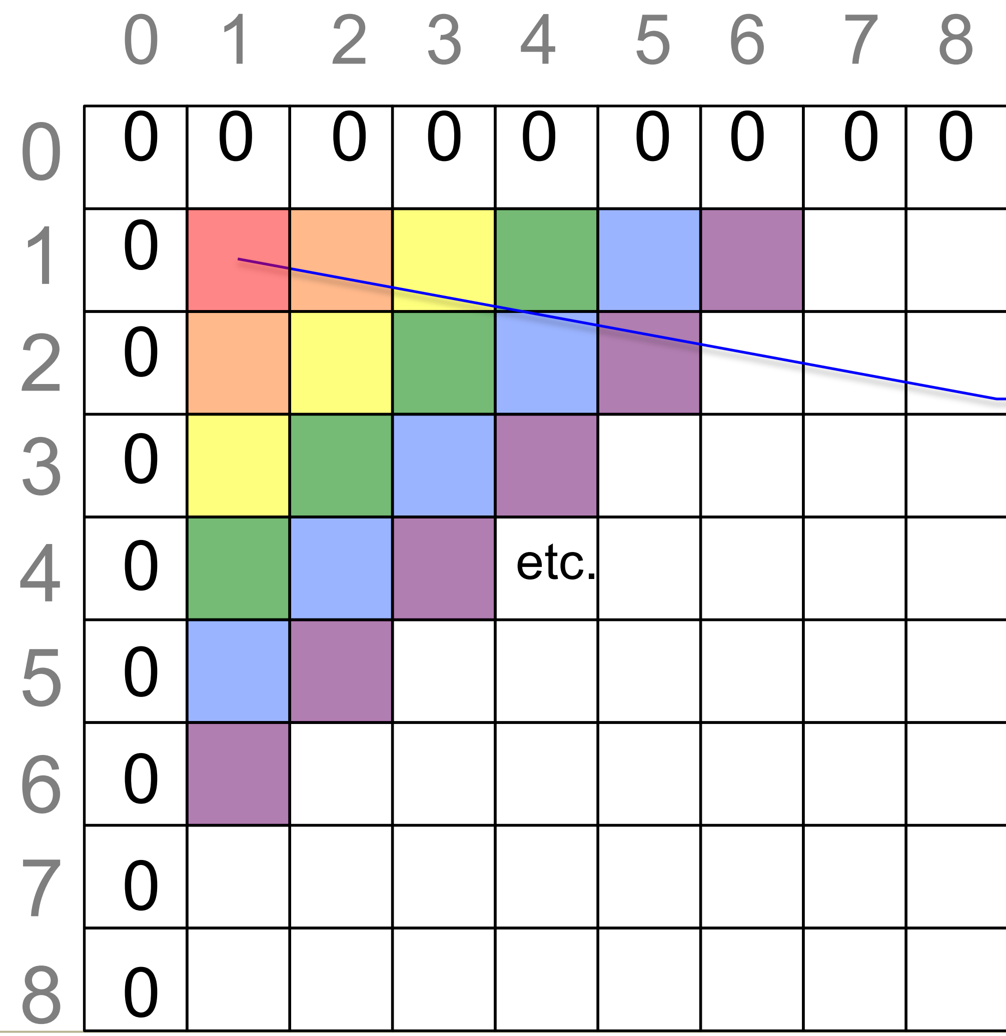
## Dynamic Programming Approach:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | | | | | | | | |
| 2 | 0 | | | | | | | | |
| 3 | 0 | | | | | | | | |
| 4 | 0 | | | | | | | | |
| 5 | 0 | | | | | | | | |
| 6 | 0 | | | | | | | | |
| 7 | 0 | | | | | | | | |
| 8 | 0 | | | | | | | | |

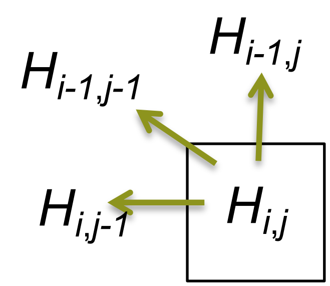
Step 1: Initialize boundaries to 0

# Smith-Waterman

## Dynamic Programming Approach:




Step 2: Compute cells when we're able to



# Smith-Waterman

## Dynamic Programming Approach:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 2 | 1 | 2 | 1 | 2 | 1 | 0 | 2 |
| 2 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 3 | 2 | 3 | 2 | 3 | 2 | 1 |
| 4 | 0 | 2 | 2 | 5 | 4 | 5 | 4 | 3 | 4 |
| 5 | 0 | 1 | 4 | 4 | 7 | 6 | 7 | 6 | 5 |
| 6 | 0 | 2 | 3 | 6 | 6 | 9 | 8 | 7 | 8 |
| 7 | 0 | 1 | 4 | 5 | 8 | 8 | 11 | 10 | 9 |
| 8 | 0 | 2 | 3 | 6 | 7 | 10 | 10 | 10 | 12 |

Step 3: Follow trail of breadcrumbs back

# Smith-Waterman

## Dynamic Programming Approach:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 2 | 1 | 2 | 1 | 2 | 1 | 0 | 2 |
| 2 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 3 | 2 | 3 | 2 | 3 | 2 | 1 |
| 4 | 0 | 2 | 2 | 5 | 4 | 5 | 4 | 3 | 4 |
| 5 | 0 | 1 | 4 | 4 | 7 | 6 | 7 | 6 | 5 |
| 6 | 0 | 2 | 3 | 6 | 6 | 9 | 8 | 7 | 8 |
| 7 | 0 | 1 | 4 | 5 | 8 | 8 | 11 | 10 | 9 |
| 8 | 0 | 2 | 3 | 6 | 7 | 10 | 10 | 10 | 12 |

Step 3: Follow trail of breadcrumbs back

# Smith-Waterman

## Dynamic Programming Approach:

Step 4: Interpret the path against the original sequences

| | | A | C | A | C | A | C | T | A |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 2 | 1 | 2 | 1 | 2 | 1 | 0 | 2 |
| G | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| C | 0 | 0 | 3 | 2 | 3 | 2 | 3 | 2 | 1 |
| A | 0 | 2 | 2 | 5 | 4 | 5 | 4 | 3 | 4 |
| C | 0 | 1 | 4 | 4 | 7 | 6 | 7 | 6 | 5 |
| A | 0 | 2 | 3 | 6 | 6 | 9 | 8 | 7 | 8 |
| C | 0 | 1 | 4 | 5 | 8 | 8 | 11 | 10 | 9 |
| A | 0 | 2 | 3 | 6 | 7 | 10 | 10 | 10 | 12 |

```
AGCACAC-A
A-CACACTA
```

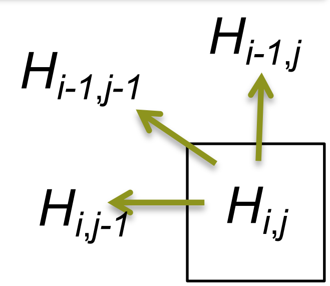
# Smith-Waterman

## Dynamic Programming Approach:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | | | | | | | | |
| 2 | 0 | | | | | | | | |
| 3 | 0 | | | | | | | | |
| 4 | 0 | | | | etc. | | | | |
| 5 | 0 | | | | | | | | |
| 6 | 0 | | | | | | | | |
| 7 | 0 | | | | | | | | |
| 8 | 0 | | | | | | | | |

Step 2: Compute cells when we're able to

How could we do this in parallel?



# Smith-Waterman

## Data-Parallel Approach:

Loop over upper diagonals serially

Process each diagonal in parallel

Repeat for lower diagonals

```
proc computeH(H: [0..n, 0..n] int) {
  for upperDiag in 1..n do
    forall diagPos in 0..#upperDiag {
      const (i,j) = (diagPos+1, upperDiag-diagPos);
      H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    }
  for lowerDiag in 1..n-1 do
    forall diagPos in lowerDiag..n-1 by -1 {
      const (i,j) = (diagPos+1, lowerDiag+diagPos);
      H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    }
}
```

Advantages:

- Reasonably clean
  (if I got my indexing correct)

Disadvantages:

- Not so great in terms of cache use
- A bit fine-grained
  - small number of iterations per task

# Smith-Waterman

## Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int;
  var Ready$: [ProbSpace] sync int;

  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  Ready$[1,1] = 1;

  coforall (i,j) in ProbSpace {
    const goNow = Ready$[i,j];
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady =  NeighborsDone[i,   j+1].fetchAdd(1);
    const seReady =    NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j  ].fetchAdd(1);
    if (eastReady == 2)  then Ready$[i,   j+1] = 1;
    if (seReady == 2)    then Ready$[i+1,j+1] = 1;
    if (southReady == 2) then Ready$[i+1,j  ] = 1;
  }
}
```

Create a domain describing shifted version of H's domain

Arrays to count how many of our 3 neighbors are done; and to signal when we can compute

Set up boundaries: north and west elements have a neighbor done; top-left is ready

Create a task per matrix element and have it block until ready

Compute our element

Increment our neighbors' counts

Signal our neighbors as ready if we're the third

# Smith-Waterman

## Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int;
  var Ready$: [ProbSpace] sync int;

  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  Ready$[1,1] = 1;

  coforall (i,j) in ProbSpace {
    const goNow = Ready$[i,j];
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady =  NeighborsDone[i,  j+1].fetchAdd(1);
    const seReady =    NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j  ].fetchAdd(1);
    if (eastReady == 2)  then Ready$[i,  j+1] = 1;
    if (seReady == 2)    then Ready$[i+1,j+1] = 1;
    if (southReady == 2) then Ready$[i+1,j  ] = 1;
  }
}
```

Disadvantages:

- Still not great in cache use
- Uses $n^2$ tasks
- Most spend most of their time blocking

# Smith-Waterman

## Slightly Less Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int;


  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  sync { computeHHelp(1,1); }


  proc computeHHelp(i,j) {
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady =  NeighborsDone[i,  j+1].fetchAdd(1);
    const seReady =    NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j  ].fetchAdd(1);
    if (eastReady == 2)  then begin computeHHelp(i,  j+1);
    if (seReady == 2)    then begin computeHHelp(i+1,j+1);
    if (southReady == 2) then begin compu
  }
}
```

sync to ensure they're all done before we go on

Rather than create the tasks *a priori*, fire them off once we know they're ready to compute

# Smith-Waterman

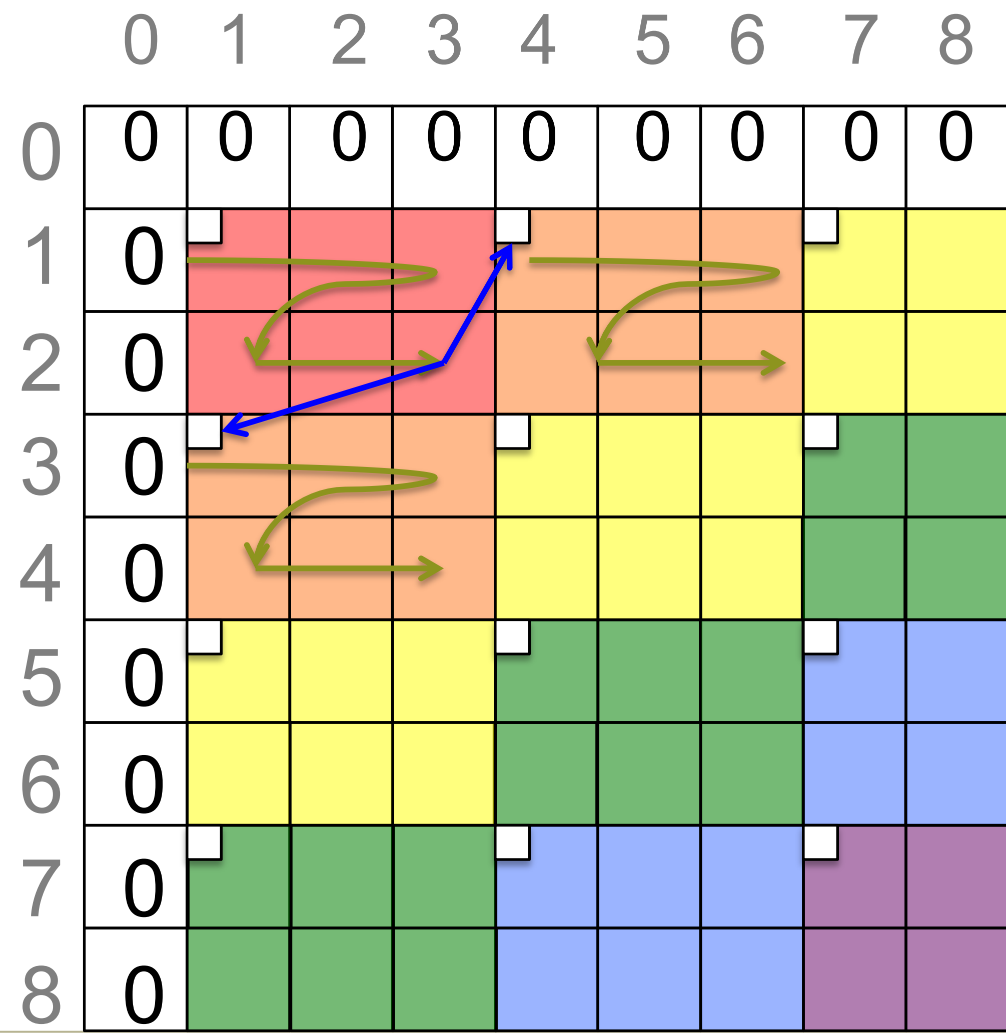
## Slightly Less Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int;


  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  sync { computeHHelp(1,1); }


  proc computeHHelp(i,j) {
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady =  NeighborsDone[i,   j+1].fetchAdd(1);
    const seReady =    NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j  ].fetchAdd(1);
    if (eastReady == 2)  then begin computeHHelp(i,   j+1);
    if (seReady == 2)    then begin computeHHelp(i+1,j+1);
    if (southReady == 2) then begin computeHHelp(i+1,j  );
  }
}
```

Disadvantages:
- Still uses a lot of tasks
- Each task is very fine-grained

# Smith-Waterman

## Coarsening the Parallelism into Chunks:



# Smith-Waterman

## Chunked Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  const StrProbSpace = ProbSpace by (rowsPerChunk, colsPerChunk);
  var NeighborsDone: [StrProbSpace] atomic int;


  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  sync { computeHHelp(1,1); }

  proc computeHHelp(x,y) {
    for (i,j) in ProbSpace[x..#rowsPerChunk, y..#colsPerChunk] do
      H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady =  NeighborsDone[x,             y+colsPerChunk].fetchAdd(1);
    const seReady =    NeighborsDone[x+rowsPerChunk, y+colsPerChunk].fetchAdd(1);
    const southReady = NeighborsDone[x+rowsPerChunk, y             ].fetchAdd(1);
    if (eastReady == 2)  then begin computeHHelp(x,            y+colsPerChunk);
    if (seReady == 2)    then begin computeHHelp(x+rowsPerChunk, y+colsPerChunk);
    if (sou                                      rowsPerChunk, y             );
  }
}
```

Use strided array for atomics

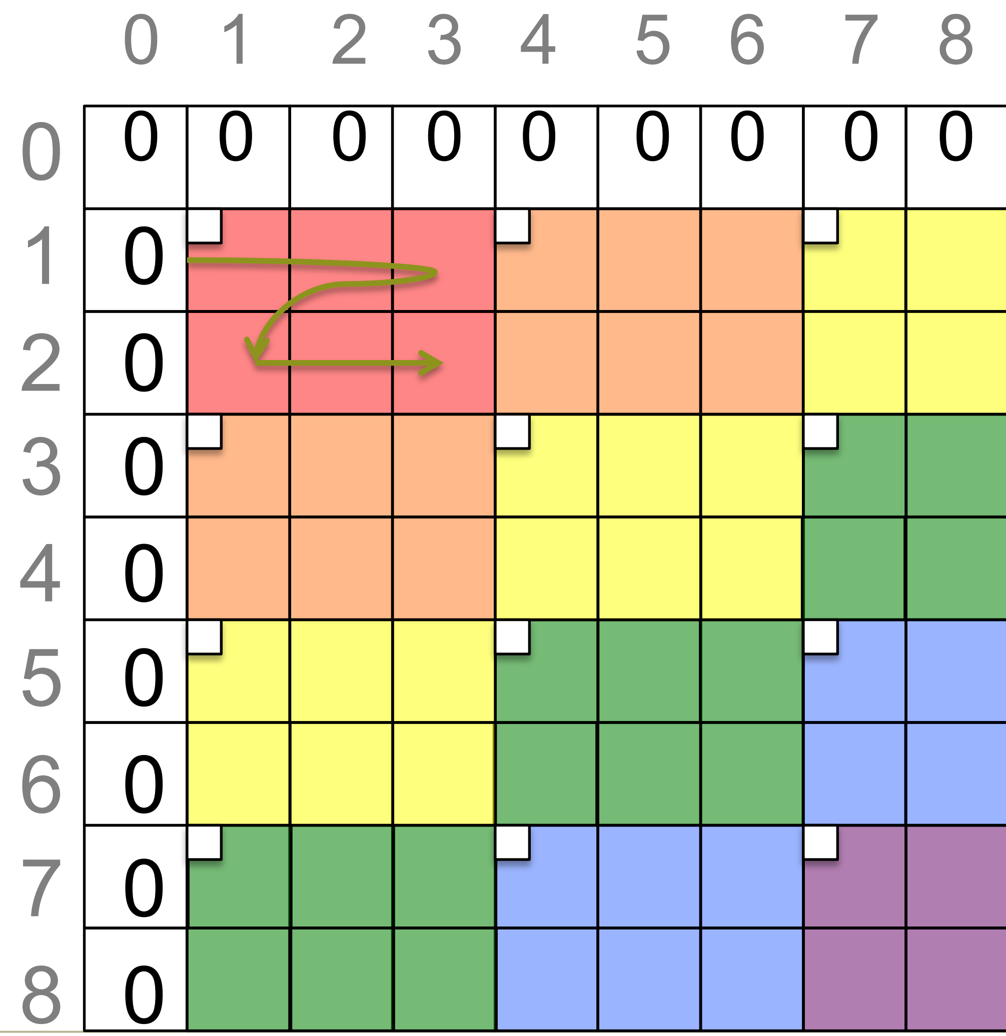
Change helper to iterate over a chunk serially

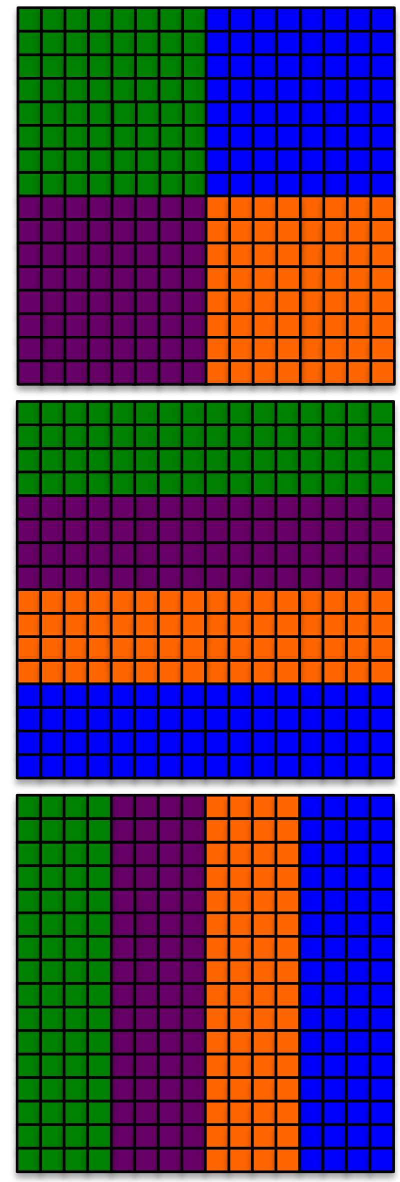
Stride indices to get to next chunk's origin

# Distributed Smith-Waterman

# Distributed Smith-Waterman

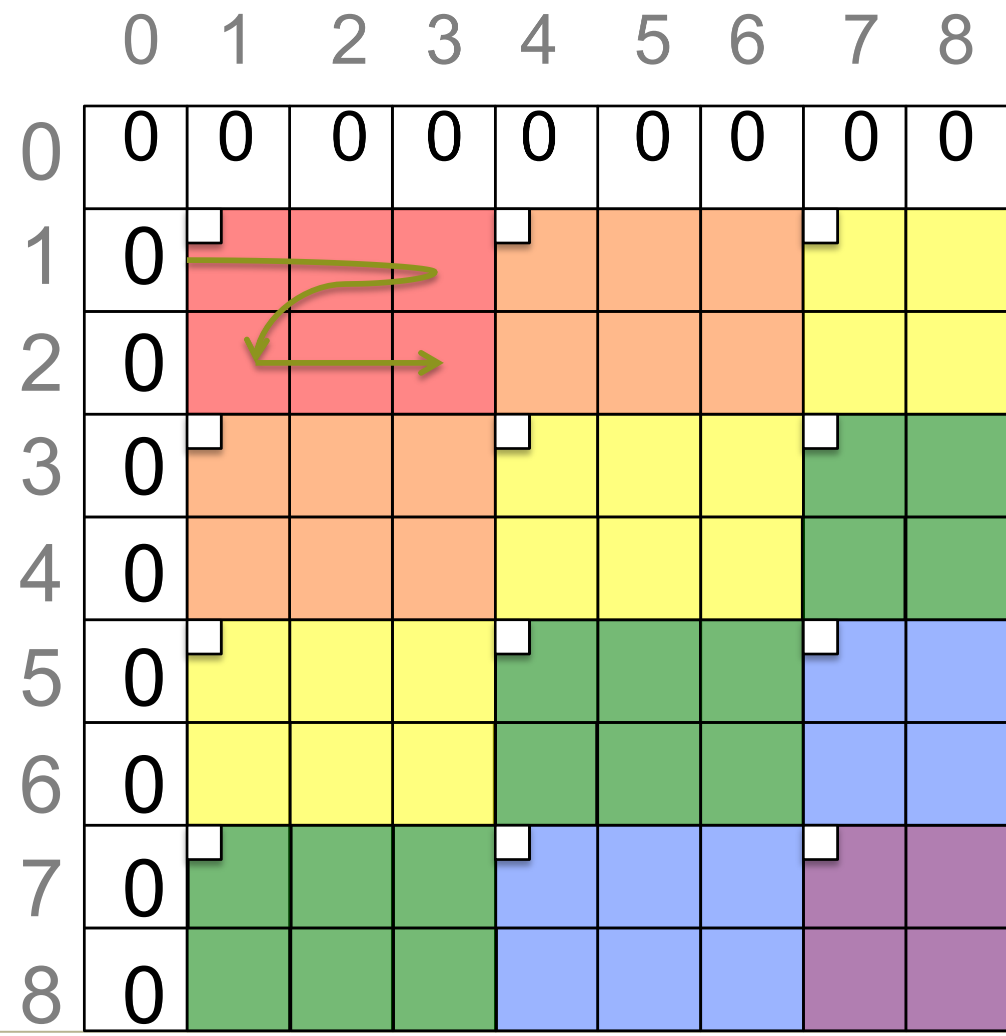
## Now, what about distributed memory?

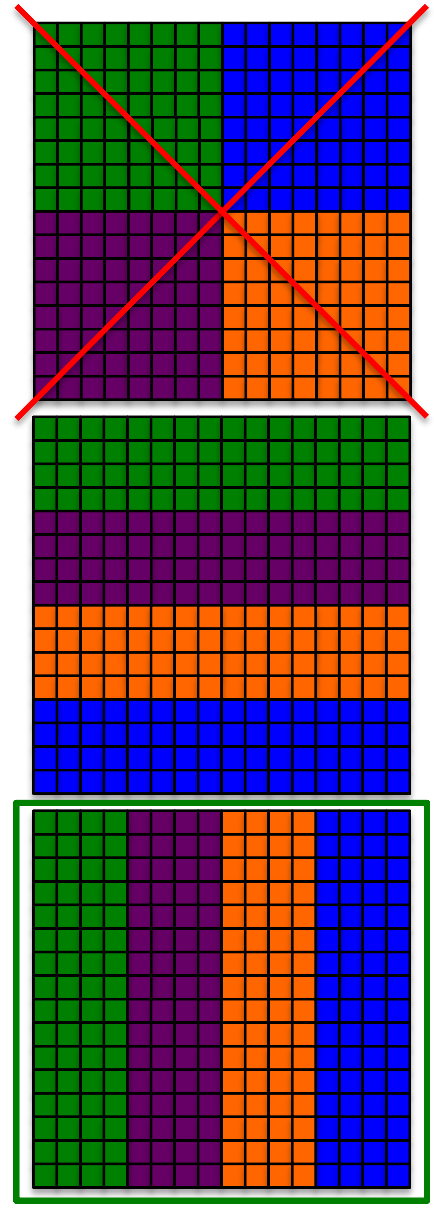


# Distributed Smith-Waterman

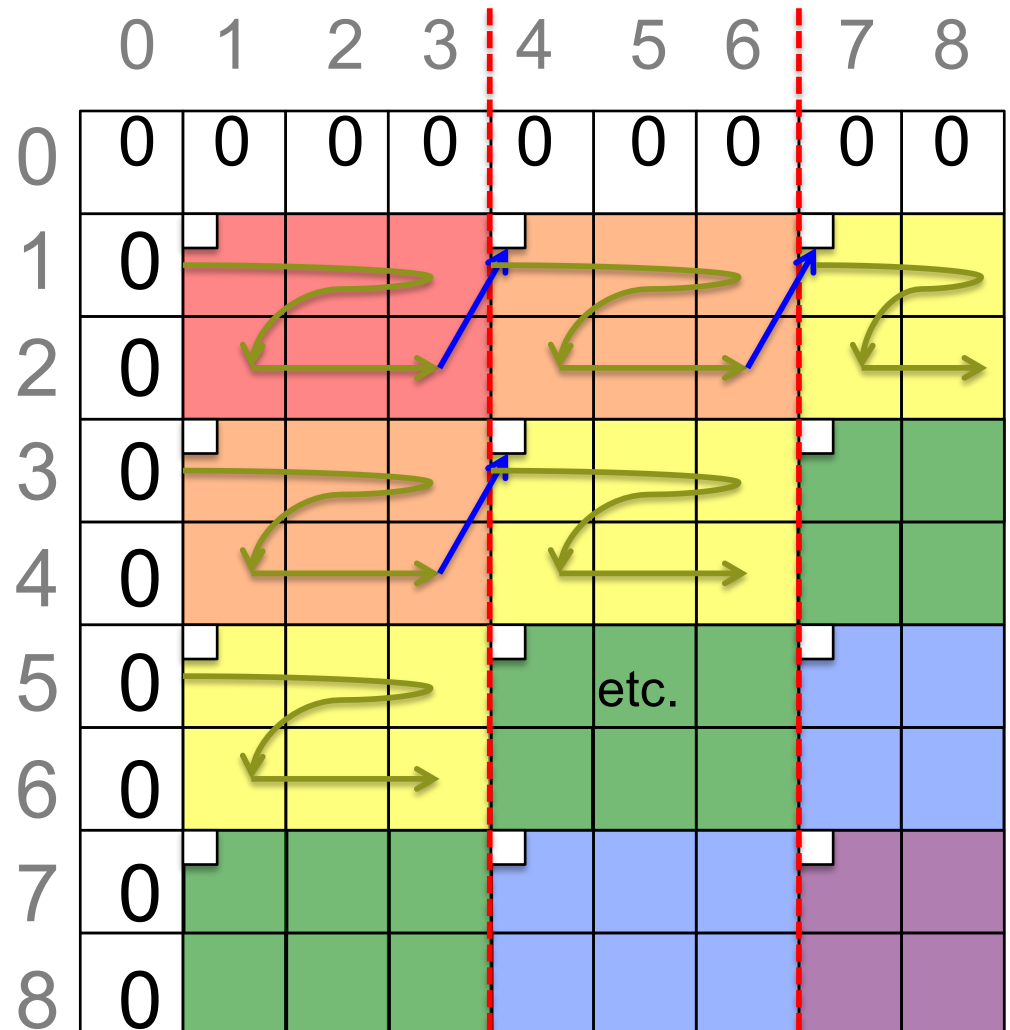
## Now, what about distributed memory?



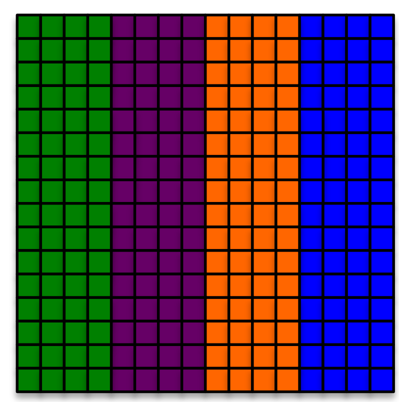
# Distributed Smith-Waterman

## Now, what about distributed memory?




<u>Advantages:</u>

- Good cache behavior: Nice fat blocks of data touchable in memory order
- Pipeline parallelism: Good utilization once pipeline is filled



# Distributed Smith-Waterman

## Distributed Chunked Data-Driven Task-Parallel Approach:

```
const Hspace = {0..n, 0..n};
const LocaleGrid = Locales.reshape({0..#numLocales, 0..0});
const DistHSpace = Hspace dmapped Block(Hspace, LocaleGrid);
var H: [DistHSpace] int;

proc computeH(H: [] int) {
  const ProbSpace = H.domain.translate(1,1);
  const StrProbSpace = ProbSpace by (rowsPerChunk, colsPerChunk);
  var NeighborsDone: [StrProbSpace] atomic int;
…
  proc computeHHelp(x,y) {
    on H[x,y] {
      for (i,j) in ProbSpace[x..#rowsPerChunk, y..#colsPerChunk] do
        H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady =  NeighborsDone[x,             y+colsPerChunk].fetchAdd(1);
    …etc…
    if (eastReady == 2)  then begin computeHHelp(x,              y+colsPerChunk);
    …etc…
} } }
```

Reshape the 1D Locales array into a 2D column

Block-distribute the data space across the column of locales

Compute each chunk on the locale that owns its initial element

# Any Questions about Smith-Waterman?

# 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, and URIKA. 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.*