Productive Programming in Chapel:
A Computation-Driven Introduction

Short Introduction to Task Parallelism

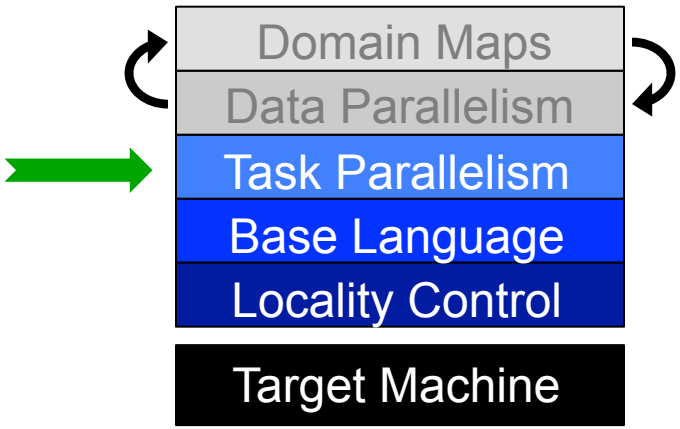
Michael Ferguson and Lydia Duncan
Cray Inc,
SC15 November 15th, 2015

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.

Outline

- ✓ Motivation
- ✓ Chapel Background and Themes
- ✓ Learning the Base Language with n-body
- ➢ **Short Introduction to Task Parallelism**
- **Hands-On 1: Hello World**
- **Short Introduction to Locality**
- **Data Parallelism with Jacobi**
- **Hands-On 2: Mandelbrot**
- **Project Status, Next Steps**



Defining our Terms

Task: a unit of computation that can/should execute in parallel with other tasks

Thread: a system resource that executes tasks

- not exposed in the language
- occasionally exposed in the implementation

Task Parallelism: a style of parallel programming in which parallelism is driven by programmer-specified tasks

(in contrast with):

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

Task Parallelism: Begin Statements

```
// create a fire-and-forget task for a statement
begin writeln("hello world");
writeln("goodbye");
```

Possible outputs:

```
hello world
goodbye
```

```
goodbye
hello world
```

Cobegins/Serial by Example: QuickSort

```
proc quickSort(arr: [?D],
               thresh = log2(here.maxTaskPar),
               depth = 0,
               low: int = D.low,
               high: int = D.high) {
  if high - low < 8 {
    bubbleSort(arr, low, high);
  } else {
    const pivotVal = findPivot(arr, low, high);
    const pivotLoc = partition(arr, low, high, pivotVal);
    serial (depth >= thresh) do cobegin {
      quickSort(arr, thresh, depth+1, low, pivotLoc-1);
      quickSort(arr, thresh, depth+1, pivotLoc+1, high);
    }
  }
}
```

Cobegins/Serial by Example: QuickSort

```
proc quickSort(arr: [?D],
               depth = 0,
               low: int = D.low,
               high: int = D.high) {
  if high - low < 8 {
    bubbleSort(arr, low, high);
  } else {
    const pivotVal = findPivot(arr, low, high);
    const pivotLoc = partition(arr, low, high, pivotVal);
    serial (here.runningTasks > here.maxTaskPar) do
      cobegin {
        quickSort(arr, depth+1, low, pivotLoc-1);
        quickSort(arr, depth+1, pivotLoc+1, high);
      }
  }
}
```

Task Parallelism: Cobegin Statements

```
// create a task per child statement
cobegin {
  producer(1);
  producer(2);
  consumer(1);
}  // implicit join of the three tasks here
```

Task Parallelism: Coforall Loops

```
// create a task per iteration
coforall t in 0..#numTasks {
  writeln("Hello from task ", t, " of ", numTasks);
} // implicit join of the numTasks tasks here

writeln("All tasks done");
```

Sample output:

```
Hello from task 2 of 4
Hello from task 0 of 4
Hello from task 3 of 4
Hello from task 1 of 4
All tasks done
```

Comparison of Begin, Cobegin, and Coforall

begin:

- Use to create a dynamic task with an unstructured lifetime
- "fire and forget"

cobegin:

- Use to create a related set of heterogeneous tasks
- …or a small, finite set of homogenous tasks
- The parent task depends on the completion of the tasks

coforall:

- Use to create a fixed or dynamic # of homogenous tasks
- The parent task depends on the completion of the tasks

Note: All these concepts can be composed arbitrarily

Task Parallelism: Data-Driven Synchronization

1) ***atomic variables:*** support atomic operations (as in C++)
   - e.g., compare-and-swap; atomic sum, etc.

2) ***single-assignment variables:*** reads block until assigned

3) ***synchronization variables:*** store full/empty state
   - by default, reads/writes block until the state is full/empty

Bounded Buffer Producer/Consumer Example

```
begin producer();
consumer();

// 'sync' types store full/empty state along with value
var buff$: [0..#buffersize] sync real;

proc producer() {
  var i = 0;
  for … {
    i = (i+1) % buffersize;
    buff$[i] = …;  // writes block until empty, leave full
} }

proc consumer() {
  var i = 0;
  while … {
    i= (i+1) % buffersize;
    …buff$[i]…;   // reads block until full, leave empty
} }
```

Synchronization Variables

- **Syntax**

```
sync-type:
  sync type
```

- **Semantics**
  - Stores *full/empty* state along with normal value
  - Defaults to *full* if initialized, *empty* otherwise
  - Default read blocks until *full,* leaves *empty*
  - Default write blocks until *empty,* leaves *full*
- **Examples: Critical sections and futures**

```
var future$: sync real;

begin future$ = compute();
res = computeSomethingElse();
useComputedResults(future$, res);
```

```
var lock$: sync bool;

lock$ = true;
critical();
var lockval = lock$;
```

Atomic Variables

- **Syntax**

```
sync-type:
  atomic type
```

- **Semantics**
  - Supports operations on variable atomically w.r.t. other tasks
  - Based on C/C++ atomic operations
- **Example: Trivial barrier**

```
var count: atomic int, done: atomic bool;

proc barrier(numTasks) {
  const myCount = count.fetchAdd(1);
  if (myCount < numTasks - 1) then
    done.waitFor(true);
  else
    done.testAndSet();
}
```

Atomic Methods

- `read():t` — return current value
- `write(v:t)` — store *v* as current value
- `exchange(v:t):t` — store *v*, returning previous value
- `compareExchange(old:t,new:t):bool`
  store *new* iff previous value was *old*; returns true on success
- `waitFor(v:t)` — wait until the stored value is *v*
- `add(v:t)` — add *v* to the value atomically
- `fetchAdd(v:t)` — same, returning pre-sum value
  (*sub*, *or*, *and*, *xor* also supported similarly)

- `testAndSet()` — like *exchange(true)* for atomic bool
- `clear()` — like *write(false)* for atomic bool

Comparison of Synchronization Types

sync/single:

- Best for producer/consumer style synchronization
  - "this task should block until something happens"
- Use single for write-once values

atomic:

- Best for uncoordinated accesses to shared state

Other Task Parallel Concepts

- **atomic variables:** support atomics ops, similar to modern C++
- **sync/single variables:** support producer/consumer patterns

- **sync statements:** join unstructured tasks
- **serial statements:** conditionally squash parallelism

Other Task Parallel Features

Current:

- serial statements to conditionally squash parallelism
- sync statements to join dynamically generated tasks

Planned:

- task-private variables
- task teams to support
  - collective operations (barriers, joins, reductions, etc.)
  - thread scheduling policies

Smith-Waterman Algorithm for Sequence Alignment

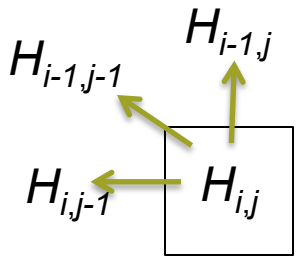
Smith-Waterman

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:

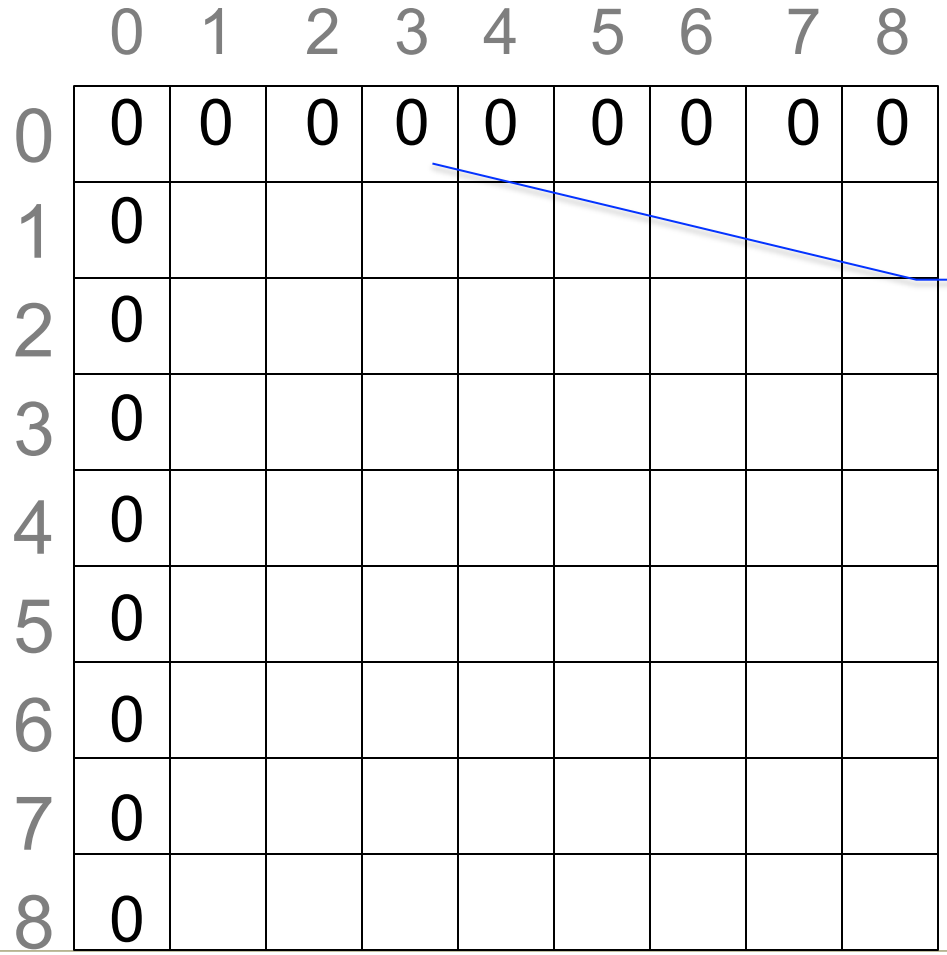
```
proc computeH(i, j) {
  if (i == 0 || j == 0) then
    return 0;
  else
    var h_NW, h_N, h_W: int;

    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:



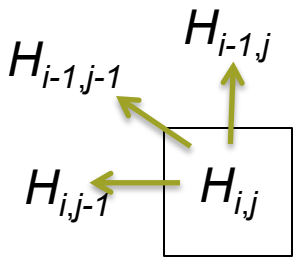
Step 1: Initialize boundaries to 0

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



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

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

		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

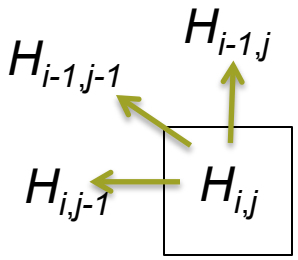
Smith-Waterman

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:

```
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]);
    }
}
```

Loop over upper diagonals serially

Process each diagonal in parallel

Repeat for lower diagonals

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 computeHHelp(i+1,j  );
  }
}
```

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:

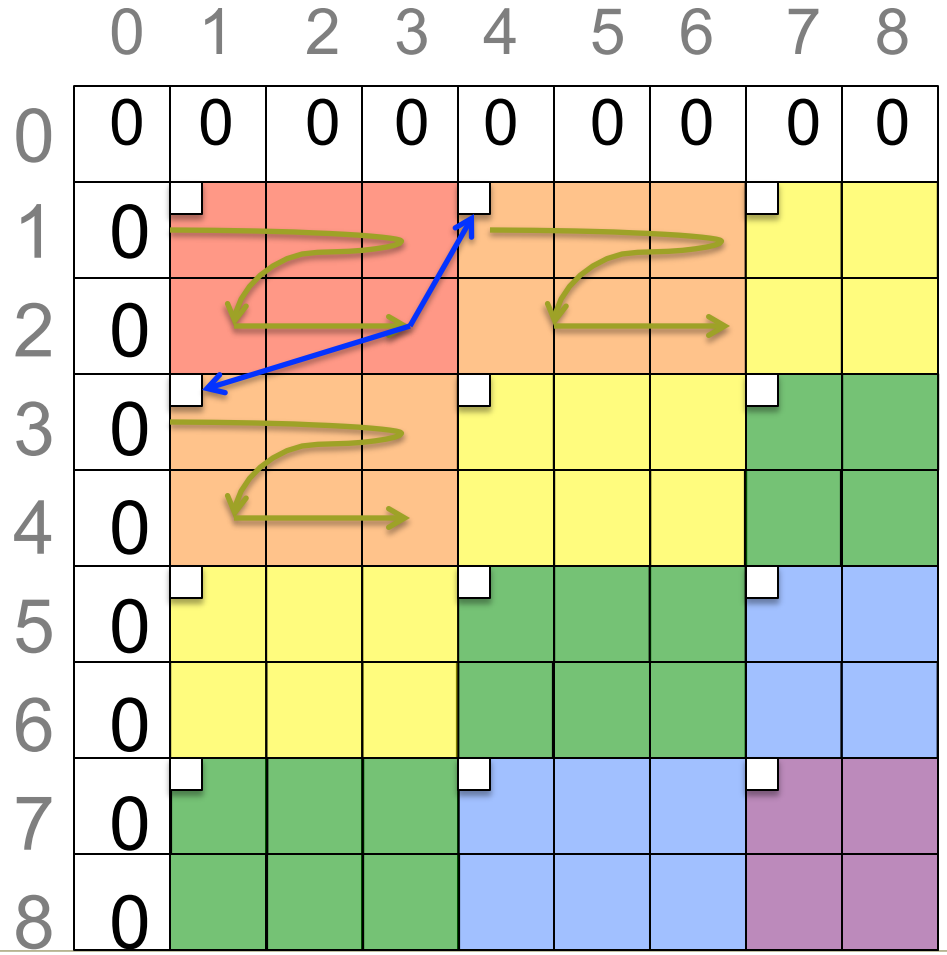
```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int;


  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 (southReady == 2) then begin computeHHelp(x+rowsPerChunk, y              );
  }
}
```

Use strided array for atomics

Change helper to iterate over a chunk serially

Stride indices to get to next chunk's origin

Questions about Task 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, 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.


CRAY
CRAY CHAPEL
COMPUTE | STORE | ANALYZE
http://chapel.cray.com chapel_info@cray.com http://github.com/chapel-lang/chapel/