Parallel Sparse Tensor Decomposition in Chapel

Thomas B. Rolinger, Tyler A. Simon, Christopher D. Krieger

IPDPSW 2018
CHIUW
Outline

1. Motivation and Background
2. Porting SPLATT to Chapel
3. Performance Evaluation: Experiments, modifications and optimizations
4. Conclusions
Motivation and Background
1.) Motivation: **Tensors + Chapel**

- Why focus on Chapel for this work?
  - Tensor decompositions algorithms are complex and immature
    - Expressiveness and simplicity of Chapel would promote maintainable and extensible code
    - High performance is crucial as well
1.) Motivation: Tensors + Chapel

- Why focus on Chapel for this work?
  - Tensor decompositions algorithms are complex and immature
    - Expressiveness and simplicity of Chapel would promote maintainable and extensible code
    - High performance is crucial as well
  - Existing tensor tools are based on C/C++ and OpenMP+MPI
    - No implementations within Chapel (or similar framework)
1.) Background: Tensors

- Tensors: Multidimensional arrays
  - Typically very large and sparse
    - Can have billions of non-zeros and densities on the order of $10^{-10}$
1.) Background: **Tensors**

- **Tensors: Multidimensional arrays**
  - Typically very large and sparse
    - Can have billions of non-zeros and densities on the order of $10^{-10}$

- **Tensor Decomposition:**
  - Higher-order extension of matrix singular value decomposition (SVD)
  - **CP-ALS:** Alternating Least Squares
    - Critical routine: Matricized tensor times Khatri-Rao product (MTTKRP)
1.) Background: SPLATT

- SPLATT: The Surprisingly Parallel spArse Tensor Toolkit
  - Developed by University of Minnesota (Smith, Karypis)
  - Written in C with OpenMP+MPI hybrid parallelism
- Current state of the art in tensor decomp.
- We focus on SPLATT’s the shared-memory (single locale) implementation of CP-ALS for this work
- Porting SPLATT to Chapel serves as a “stress test” for Chapel
  - File I/O, BLAS/LAPACK interface, custom data structures and non-trivial parallelized routines
Porting SPLATT to Chapel
2.) Porting SPLATT to Chapel: Overview

- **Goal:** simplify SPLATT code when applicable but preserve original implementation and design
- Single-locale port
  - Multi-locale port left for future work
- Mostly a straightforward port
  - However, there were some cases that required extra effort to port: **mutex/locks**, work sharing constructs, jagged arrays
2.) Porting SPLATT to Chapel:

**Mutex Pool**

- SPLATT uses a mutex pool for some of the parallel MTTKRP routines to synchronize access to matrix rows.
- Chapel currently does not have a native lock/mutex module.
  - Can recreate behavior with `sync` or `atomic` variables.
  - We originally used `sync` variables, but later switched to `atomic` (see Performance Evaluation section).

```chapel
proc set(pool : [] atomic bool, lockID : int) { 
  while pool[lockID].testAndSet() { 
    chpl_task_yield();
  }
}
proc unset(pool : [] atomic bool, lockID : int) { 
  pool[lockID].clear();
}
```
Performance Evaluation
4.) Performance Evaluation: Set Up

- Compare performance of Chapel port of original C/OpenMP code
- Default Chapel 1.16 build (Qthreads, jemalloc)
- OpenBLAS for BLAS/LAPACK
- Ensured both C and Chapel code utilize same # of threads for each trial
  - OMP_NUM_THREADS
  - CHPL_RT_NUM_THREADS_PER_LOCALE
4.) Performance Evaluation: **Datasets**

<table>
<thead>
<tr>
<th>Name</th>
<th>Dimensions</th>
<th>Non-Zeros</th>
<th>Density</th>
<th>Size on Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>YELP</td>
<td>41k x 11k x 75k</td>
<td>8 million</td>
<td>1.97E-7</td>
<td>240 MB</td>
</tr>
<tr>
<td>RATE-BEER</td>
<td>27k x 105k x 262k</td>
<td>62 million</td>
<td>8.3E-8</td>
<td>1.85 GB</td>
</tr>
<tr>
<td>BEER-ADVOCATE</td>
<td>31k x 61k x 182k</td>
<td>63 million</td>
<td>1.84E-7</td>
<td>1.88 GB</td>
</tr>
<tr>
<td>NELL-2</td>
<td>12k x 9k x 29k</td>
<td>77 million</td>
<td>2.4E-5</td>
<td>2.3 GB</td>
</tr>
<tr>
<td>NETFLIX</td>
<td>480k x 18k x 2k</td>
<td>100 million</td>
<td>5.4E-6</td>
<td>3 GB</td>
</tr>
</tbody>
</table>

See paper for more details on data sets
4.) Performance Evaluation: **Summary**

- Profiled and analyzed Chapel code
  - Initial code exhibited very poor performance
- Identified 3 major bottlenecks
  - MTTKRP: up to **163x slower** than C code
  - Matrix inverse: up to **20x slower** than C code
  - Sorting (refer to paper for details)
- After modifications to initial code
  - Achieved competitive performance to C code
4.) Performance Evaluation:

MTTKRP Optimizations: Matrix Row Accessing

Original C: number of cols is small (35) but number of rows is large (tensor dims)
4.) Performance Evaluation:

**MTTKRP Optimizations:** Matrix Row Accessing

**Original C:** number of cols is small (35) but number of rows is large (tensor dims)

**Initial Chapel:** use slicing to get row reference → very slow since cost of slicing is not amortized by computation on each slice
4.) Performance Evaluation:

**MTTKRP Optimizations: Matrix Row Accessing**

Original C: number of cols is small (35) but number of rows is large (tensor dims)

```
double *mat = ...; // row-major 1D array
for(int j = 0; j < cols; j++) {
    row[j]...
}
```

Initial Chapel: use slicing to get row reference → very slow since cost of slicing is not amortized by computation on each slice

```
var mat [0..rows-1,0..cols-1] = ...;
for j in 0..cols-1 {
    row[j]...
}
```

2D Index: use (i,j) index into original matrix instead of getting row reference → 17x speed up over initial MTTKRP code

```
var mat [0..rows-1,0..cols-1] = ...;
for j in 0..cols-1 {
    row[i,j]...
}
```
4.) Performance Evaluation:

**MTTKRP Optimizations:** Matrix Row Accessing

**Original C:** number of cols is small (35) but number of rows is large (tensor dims)

```c
double *mat = ...; // row-major 1D array
double *row = mat + (i*cols);
for(int j = 0; j < cols; j++) {
    row[j]...
}
```

**Initial Chapel:** use slicing to get row reference → very slow since cost of slicing is not amortized by computation on each slice

```chapel
var mat [0..rows-1,0..cols-1] = ...
ref row = mat[i,0..cols-1];
for j in 0..cols-1 {
    row[j]...
}
```

**2D Index:** use (i,j) index into original matrix instead of getting row reference → **17x** speed up over initial MTTKRP code

```chapel
var mat [0..rows-1,0..cols-1] = ...
for j in 0..cols-1 {
    row[i,j]...
}
```

**Pointer:** more direct C translation → **1.26x** speed up over 2D indexing

```chapel
var mat [0..rows-1,0..cols-1] = ...
var matPtr = c_ptrTo(mat);
var row = matPtr + (i*cols);
for j in 0..cols-1 {
    row[j]...
}
```
MTTKRP Runtime: Chapel Matrix Access Optimizations

**YELP**

- **Initial**
- **2D Index**
- **Pointer**

**NELL-2**

- **Initial**
- **2D Index**
- **Pointer**

The graphs show the time in seconds for different numbers of threads/tasks (1 to 32) for YELP and NELL-2 datasets, with three different initializations: Initial, 2D Index, and Pointer.
MTTKRP Runtime: Chapel Matrix Access Optimizations

YELP

- YELP: virtually no scalability after 2 tasks

NELL-2

- NELL-2: near linear speed-up
4.) Performance Evaluation:

MTTKRP Optimizations: Mutex/Locks

- YELP requires the use of locks during the MTTKRP and NELL-2 does not
  - Decision whether to use locks is highly dependent on tensor properties and number of threads used

- Initially used `sync vars`
  - MTTKRP critical regions are short and fast
- Not well suited for how `sync vars` are implemented in Qthreads
- Switched to `atomic vars`
- Up to 14x improvement on YELP
- FIFO w/ `sync vars` competitive with Qthreads w/ `atomic vars`

- Troubling: simple recompilation of code can drastically alter performance
4.) Performance Evaluation:

**MTTKRP Optimizations: Mutex/Locks**

- YELP requires the use of locks during the MTTKRP and NELL-2 does not
  - Decision whether to use locks is highly dependent on tensor properties and number of threads used
- Initially used `sync` vars
  - MTTKRP critical regions are short and fast
    - Not well suited for how `sync` vars are implemented in Qthreads
  - Switched to `atomic` vars
    - Up to $14x$ improvement on YELP

Troubling: simple recompilation of code can drastically alter performance
4.) Performance Evaluation:

**MTTKRP Optimizations: Mutex/Locks**

- YELP requires the use of locks during the MTTKRP and NELL-2 does not
  - Decision whether to use locks is highly dependent on tensor properties and number of threads used

- Initially used `sync` vars
  - MTTKRP critical regions are short and fast
    - Not well suited for how `sync` vars are implemented in Qthreads
  - Switched to `atomic` vars
    - Up to 14x improvement on YELP

- FIFO w/ `sync` vars competitive with Qthreads w/ `atomic` vars
  - Troubling: just recompiling the code can drastically alter performance
Chapel MTTKRP Runtime
sync vars VS atomic vars
YELP

NO CODE DIFFERENCE: just recompiled for different tasking layer
4.) Performance Evaluation: Matrix Inverse (OpenBLAS/OpenMP)

- SPLATT uses LAPACK routines to compute matrix inverse
  - Experiments used OpenBLAS, parallelized via OpenMP

Issue: interaction of Qthreads and OpenMP is messy
4.) Performance Evaluation: Matrix Inverse (OpenBLAS/OpenMP)

- SPLATT uses LAPACK routines to compute matrix inverse
  - Experiments used OpenBLAS, parallelized via OpenMP
- Observed **15x slow down** in matrix inverse runtime for Chapel when using 32 threads (OpenMP and Qthreads)
- **Issue:** interaction of Qthreads and OpenMP is messy
4.) Performance Evaluation: Matrix Inverse (OpenBLAS/OpenMP) cont.

**Problem:** OpenMP and Qthreads stomp over each other

**Reason:** Default $\rightarrow$ Qthreads pinned to cores
  - OpenMP threads all end up on 1 core due to how Qthreads uses sched_setaffinity

**Result:** Huge performance loss for OpenMP routine
4.) Performance Evaluation:
Matrix Inverse (OpenBLAS/OpenMP) cont.

- **Try:** Explicitly bind OpenMP threads to cores
- **Result:** Chapel will fall back to only using 1 thread

**Reason:** Same as OpenMP in previous slide – Difference: Chapel detects this over subscription and will prevent it by only using 1 thread.

**Problem:** Not always clear to users – If `CHPL_RT_NUM_THREADS_PER_LOCALE` is set, then a warning is displayed about falling back to 1 thread – If not, users expect default (# threads == # cores) but only a single thread is used and no warning given.
4.) Performance Evaluation: Matrix Inverse (OpenBLAS/OpenMP) cont.

- **Try:** Explicitly bind OpenMP threads to cores
- **Result:** Chapel will fall back to only using 1 thread
- **Reason:** Same as OpenMP in previous slide
  - Difference: Chapel detects this over subscription and will prevent it by only using 1 thread
4.) Performance Evaluation:

Matrix Inverse (OpenBLAS/OpenMP) cont.

- **Try:** Explicitly bind OpenMP threads to cores
- **Result:** Chapel will fall back to only using 1 thread
- **Reason:** Same as OpenMP in previous slide
  - Difference: Chapel detects this over subscription and will prevent it by only using 1 thread
- **Problem:** Not always clear to users
  - If `CHPL_RT_NUM_THREADS_PER_LOCALE` is set, then a warning is displayed about falling back to 1 thread
  - If not, users expect default (# threads == # cores) but only a single thread is used and **no warning given**
4.) Performance Evaluation: Matrix Inverse (OpenBLAS/OpenMP) cont.

- **Attempted solutions:**
  1.) QT_AFFINITY=no, QT_SPINCOUNT=300
  2.) Remove Chapel over subscription warning/check and allow both Qthreads and OpenMP threads to bind to cores

- Overall Results:
  (1) and (2) provided roughly equal improvement of OpenMP runtime but still 4x slower than the C code
4.) Performance Evaluation:
Matrix Inverse (OpenBLAS/OpenMP) cont.

• Attempted solutions:
  – 1.) QT_AFFINITY=no, QT_SPINCOUNT=300
  – 2.) Remove Chapel over subscription warning/check and allow both Qthreads and OpenMP threads to bind to cores

• Overall Results:
  – (1) and (2) provided roughly equal improvement of OpenMP runtime but still 4x slower than the C code
4.) Performance Evaluation:
Matrix Inverse (OpenBLAS/OpenMP) cont.

- Another issue:
  - Improving OpenMP runtime caused a 7 to 13x slow down in a Chapel routine that followed
  - Still resource contention on cores
4.) Performance Evaluation:

Matrix Inverse (OpenBLAS/OpenMP) cont.

• Another issue:
  – Improving OpenMP runtime caused a **7 to 13x slow down** in a Chapel routine that followed
  – Still resource contention on cores

• No clear solution to overcome issues
  – We set OMP_NUM_THREADS=1 for Chapel runs since OpenMP runtime is generally negligible
4.) Performance Evaluation:

Matrix Inverse (OpenBLAS/OpenMP) cont.

- **Another issue:**
  - Improving OpenMP runtime caused a 7 to 13x slow down in a Chapel routine that followed
  - Still resource contention on cores

- **No clear solution to overcome issues**
  - We set OMP_NUM_THREADS=1 for Chapel runs since OpenMP runtime is generally negligible

- **Brings up crucial question regarding library integration:**
  - When does it make sense to provide native Chapel implementations rather than integrate with existing libraries?
Final Results

MTTKRP Runtime

YELP

NELL-2

time - seconds

threads/tasks

C
Chapel-initial
Chapel-optimize

0.5
1
2
4
8
16
32
64
128
256
512
1024
2048
2048
1024
512
256
128
64
32
16
8
4
2
1

1
2
4
8
16
32

1
2
4
8
16
32

5.) Conclusions

• Implemented parallel sparse tensor decomposition in Chapel
• Identified bottlenecks in code
  – Array slicing
  – `sync` vs `atomic` variables for locks
  – Conflicts between OpenMP and Qthreads
• Achieved 83-96% of the original C/OpenMP performance after modifications to initial port
• Suggestions for Chapel:
  – Create a mutex/lock library
  – More documentation/experiments with integrating 3rd party code that utilize different threading libraries
• Future work:
  – Multi-locale version
  – Closer inspection of code to make it more Chapel-like
    • Will the performance suffer or improve?
Questions

Contact: tbrolin@cs.umd.edu
Back up Slides
Matricizing a Tensor

\[ X_1 = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}, \quad X_2 = \begin{bmatrix} 13 & 16 & 19 & 22 \\ 14 & 17 & 20 & 23 \\ 15 & 18 & 21 & 24 \end{bmatrix} \]

\[ X_{(1)} = \begin{bmatrix} 1 & 4 & 7 & 10 & 13 & 16 & 19 & 22 \\ 2 & 5 & 8 & 11 & 14 & 17 & 20 & 23 \\ 3 & 6 & 9 & 12 & 15 & 18 & 21 & 24 \end{bmatrix}, \]

\[ X_{(2)} = \begin{bmatrix} 1 & 2 & 3 & 13 & 14 & 15 \\ 4 & 5 & 6 & 16 & 17 & 18 \\ 7 & 8 & 9 & 19 & 20 & 21 \\ 10 & 11 & 12 & 22 & 23 & 24 \end{bmatrix}, \]

\[ X_{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & \cdots & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & \cdots & 21 & 22 & 23 & 24 \end{bmatrix} \]
Kronecker and Khatri-Rao Products

**Kronecker Product**

\[
A \otimes B = \begin{bmatrix}
a_{11}B & a_{12}B & \cdots & a_{1J}B \\
a_{21}B & a_{22}B & \cdots & a_{2J}B \\
\vdots & \vdots & \ddots & \vdots \\
a_{IJ}B & a_{I2}B & \cdots & a_{IJ}B \\
\end{bmatrix}
= \begin{bmatrix}
a_1 \otimes b_1 & a_1 \otimes b_2 & a_1 \otimes b_3 & \cdots & a_J \otimes b_{L-1} & a_J \otimes b_L \\
\end{bmatrix}
\]

**Khatri-Rao Product**

\[
A \odot B = \begin{bmatrix}
a_1 \otimes b_1 & a_2 \otimes b_2 & \cdots & a_K \otimes b_K \\
\end{bmatrix}
\]
4.) Performance Evaluation:

Sorting Optimizations

- Profiled customized sorting routine in Chapel code and found two bottlenecks:
  - Creation of small array in recursive routine
    - Created millions of times due to recursion and large tensors: consumed up to 10% of the sorting runtime
    - **Solution**: just declare local ints rather than an array (possible since this array was only of length 2)
  - Reassignment of array of arrays
    - C code: array of pointers $\rightarrow$ simple pointer assignment
    - Chapel code:
      - Initially 2D matrix $\rightarrow$ used slicing for reassignment (slow due to large size of slices)
      - Changed to array of arrays $\rightarrow$ whole array assignment (slow due to copying the arrays)
      - **Final**: get pointer to arrays and use pointer reassignment (similar to C code)

- Modifications resulted in roughly **4x** improvement
Chapel Sorting Runtime
NELL-2

![Chart showing time in seconds vs. threads/tasks for different optimization options. The chart includes lines for Initial, Array-opt, Slices-opt, and All-opts, with a clear decrease in time as the number of threads/tasks increases.]
Runtimes for CP-ALS Routines

YELP: 1 thread/task

YELP: 32 threads/tasks
## Runtimes for CP-ALS Routines

### NELL-2: 1 thread/task

<table>
<thead>
<tr>
<th>Routine</th>
<th>C Seconds</th>
<th>Chapel-optimize Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTTKRP</td>
<td>109.25</td>
<td>130.55</td>
</tr>
<tr>
<td>INVERSE</td>
<td>0.002</td>
<td>0.003</td>
</tr>
<tr>
<td>MAT MULT</td>
<td>0.78</td>
<td>1.17</td>
</tr>
<tr>
<td>MAT A^TA</td>
<td>0.13</td>
<td>0.14</td>
</tr>
<tr>
<td>MAT NORM</td>
<td>0.06</td>
<td>0.05</td>
</tr>
<tr>
<td>CPD FIT</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>SORT</td>
<td>7.90</td>
<td>9.86</td>
</tr>
</tbody>
</table>

### NELL-2: 32 threads/tasks

<table>
<thead>
<tr>
<th>Routine</th>
<th>C Seconds</th>
<th>Chapel-optimize Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTTKRP</td>
<td>5.81</td>
<td>6.03</td>
</tr>
<tr>
<td>INVERSE</td>
<td>0.003</td>
<td>0.008</td>
</tr>
<tr>
<td>MAT MULT</td>
<td>0.06</td>
<td>0.13</td>
</tr>
<tr>
<td>MAT A^TA</td>
<td>0.24</td>
<td>0.19</td>
</tr>
<tr>
<td>MAT NORM</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>CPD FIT</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>SORT</td>
<td>0.63</td>
<td>1.45</td>
</tr>
</tbody>
</table>
## 4.) Performance Evaluation:

### Initial Results: CP-ALS Routines Runtimes

<table>
<thead>
<tr>
<th>Data set</th>
<th>Threads/tasks</th>
<th>Code</th>
<th>MTTKRP</th>
<th>Sort</th>
<th>Mat $A^TA$</th>
<th>Mat Norm</th>
<th>CPD Fit</th>
<th>Inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>YELP</td>
<td>1</td>
<td>C</td>
<td>13.31</td>
<td>0.82</td>
<td>0.34</td>
<td>0.14</td>
<td>0.04</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chapel-Initial</td>
<td>225.11</td>
<td>7.21</td>
<td>0.36</td>
<td>0.14</td>
<td>0.04</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>C</td>
<td>0.73</td>
<td>0.07</td>
<td>0.41</td>
<td>0.01</td>
<td>0.01</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chapel-Initial</td>
<td>118.93</td>
<td>0.47</td>
<td>0.56</td>
<td>0.06</td>
<td>0.01</td>
<td>0.98</td>
</tr>
<tr>
<td>NELL-2</td>
<td>1</td>
<td>C</td>
<td>109.25</td>
<td>7.9</td>
<td>0.13</td>
<td>0.06</td>
<td>0.01</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chapel-Initial</td>
<td>1999</td>
<td>69.04</td>
<td>0.14</td>
<td>0.06</td>
<td>0.01</td>
<td>0.39</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>C</td>
<td>5.81</td>
<td>0.63</td>
<td>0.24</td>
<td>0.01</td>
<td>0.01</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chapel-Initial</td>
<td>88.3</td>
<td>5.01</td>
<td>0.19</td>
<td>0.02</td>
<td>0.01</td>
<td>0.39</td>
</tr>
</tbody>
</table>

*Times shown in seconds*
4.) Performance Evaluation:

**MTTKRP Optimizations: Mutex/Locks**

- YELP requires the use of locks during the MTTKRP and NELL-2 does not
  - Decision whether to use locks is highly dependent on tensor properties and number of threads used

<table>
<thead>
<tr>
<th>Sync vars (Qthreads)</th>
<th>Atomic vars (Qthreads)</th>
<th>Sync vars (FIFO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Tasks put to sleep</td>
<td>- Tasks spin-wait</td>
<td>- Tasks spin-wait, similar to atomic vars in Qthreads</td>
</tr>
<tr>
<td>- Suitable for long-held heavily contended locks</td>
<td>- Suitable for short, non-intensive critical regions</td>
<td></td>
</tr>
</tbody>
</table>

- Initially used sync vars
  - MTTKRP critical regions are short and fast
  - Switching to atomic vars gave huge improvement for YELP
- FIFO w/ sync vars competitive with Qthreads w/ atomic vars
  - Troubling: simple recompilation of code can drastically alter performance
## 4.) Performance Evaluation:

### Initial Results: CP-ALS Routines Runtimes

<table>
<thead>
<tr>
<th>Data set</th>
<th>Threads/tasks</th>
<th>Code</th>
<th>MTTKRP</th>
<th>Inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>YELP</strong></td>
<td>1</td>
<td>C</td>
<td>13.31</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chapel</td>
<td><strong>225.11</strong> → <strong>15.15</strong></td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>C</td>
<td>0.73</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chapel</td>
<td><strong>118.93</strong> → <strong>0.88</strong></td>
<td>0.98</td>
</tr>
<tr>
<td><strong>NELL-2</strong></td>
<td>1</td>
<td>C</td>
<td>109.25</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chapel</td>
<td><strong>1999</strong> → <strong>130.54</strong></td>
<td>0.39</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>C</td>
<td>5.81</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chapel</td>
<td><strong>88.3</strong> → <strong>6.03</strong></td>
<td>0.39</td>
</tr>
</tbody>
</table>

*Times shown in seconds*
3.) Porting SPLATT to Chapel:

Work Sharing Constructs

```plaintext
#pragma omp parallel shared(A,B) num_threads(2)
int tid = omp_get_thread_num();
A[tid] = foo(tid);
#pragma omp barrier
#pragma omp for
for(int i = 0; i < 16; i++) {
  B[i] = bar(i, A);
}
```

≠

```plaintext
coforall tid in 0..1 {
  A[tid] = foo(tid);
  b.barrier();
  forall i in 0..15 {
    B[i] = bar(i, A);
  }
}
```
3.) Porting SPLATT to Chapel:

Work Sharing Constructs

Solution: Manually compute loop bounds for each task
3.) Porting SPLATTT to Chapel: Work Sharing Constructs (cont.)

Specific case of perfectly nested loops and partial reduction → clean and concise Chapel translation

```chapel
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  double *myVals = thdData[tid];
  #pragma omp for
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      myVals[j] += vals[i][j] * 2;
    }
  }
  // do reduction on myVals
}
```

```chapel
var myVals: [cols] real;
forall r in rows with (+ reduce myVals) do
  forall c in cols do
    myVals[c] += vals[r,c] * 2
```