Declarative Aspects of Memory Management in the Concurrent Collections Parallel Programming Model

Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, Leo Treggiari
Background

• Mem within step and within env handled by serial language
• The lifetime of an item is longer than a step but shorter than the whole graph execution
• Items, taken together may occupy more memory than is available
• Not like typical GC
  – No pointers
  – CnC spec itself can not determine dead items because when an item becomes dead depends on the schedule
  – Even if a step is not yet prescribed and it’s input is not all available, it may do a get on the item in question.
Memory transitions
Garbage Collection of Dead Items

• Our approach
  – Constrained to a class of applications
  – Declarative slicing annotation within CnC
  – Runtime uses these annotations to determine when no future steps could possibly access the item

• Context
  – Annotations presented as user-supplied
  – For the class of applications handled, these can be automatically generated
Cholesky factorization

- Input matrix $B$ of size $n \times n$ where $n = p*b$ for some $b$ which denotes tile size
- Output lower triangular matrix $L$

1. for $k = 1$ to $p$ do
2. conventionalCholesky($B_{kk}$, $L_{kk}$);
3. for $j = k + 1$ to $p$ do
4. triangularSolve($L_{kk}$, $B_{jk}$, $L_{jk}$);
5. for $i = k+1$ to $j$ do
6. symmetricRank-kUpdate($L_{jk}$, $L_{ik}$, $B_{ij}$);

Aparna Chandramolishvaran
Rich Vuduc
(Georgia Tech)
Cholesky Factorization Example

Compute steps

Tag collection
Step collection
Item collection
Producer / consumer relation
Prescription relation
Input / output
Line 2: Conventional Cholesky

K = 1
Line 4: Triangular Solve

<table>
<thead>
<tr>
<th>K = 1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>K = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Line 6: Symmetric Rank k Update

<table>
<thead>
<tr>
<th>K = 1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>K = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>K = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>K = 1</td>
</tr>
</tbody>
</table>
CnC asynchronous execution

| K = 1 |   |   |   |
CnC asynchronous execution

<table>
<thead>
<tr>
<th>K = 1</th>
</tr>
</thead>
</table>
CnC asynchronous execution

<table>
<thead>
<tr>
<th>K = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>K = 1</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CnC asynchronous execution: k=1 not done. k=2 starts.
Given:

an item instance and a step collection

**slice constraint:** What are the set of all possible step instances such that, if they execute, they will *get* this item instance?

**tag constraint:** What are the set of step instances that will actually execute?

Intersect the two sets
Use of Slicing Annotations for Memory Management

• Transform these constraints into expressions for getCounts
• Transform expressions for getCounts into getCounts
• Free item when all gets of item have completed
Slice Constraints

getS1A
from step: (s1: k) gets [L: k, k, k]
from item: [L: t1, t2, t3] is gotten by (s1: k)
slice constraint: t1 = t2 ∧ t2 = t3 ∧ t1 = k

getS2A
from step: (s2: k, j) gets [L: j, k, k]
from item: [L: t1, t2, t3] is gotten by (s2: k, j)
slice constraint: t2 = t3 ∧ t3 = k ∧ t1 = j

getS2B
from step: (s2: k, j) gets [L: k, k, k+1]
from item: [L: t1, t2, t3] is gotten by (s2: k, j)
slice constraint: t1 = t2 ∧ t2 = k ∧ t3 = k+1

getS3A
from step: (s3: k, j, i) gets [L: j, i, k]
from item: [L: t1, t2, t3] is gotten by (s3: k, j, i)
slice constraint: t1 = j ∧ t2 = i ∧ t3 = k

getS3B
from step: (s3: k, j, i) gets [L: j, k, k+1]
from item: [L: t1, t2, t3] is gotten by (s3: k, j, i)
slice constraint: t1 = j ∧ t2 = k ∧ t3 = k+1

getS3C
from step: if i != j (s3: k, j, i) gets [L: i, k, k+1]
from item: [L: t1, t2, t3] is gotten by (s3: k, j, i)
slice constraint: t1 = i ∧ t2 = k ∧ t3 = t2+1 ∧ (i ≤ j ∨ i ≥ j)
for $k = 1$ to $p$ do
for $j = k + 1$ to $p$ do
for $i = k+1$ to $j$ do
Tag Constraints

getS1A
from step: (s1: k) gets [L: k, k, k]
from item: [L: t1, t2, t3] is gotten by (s1: k)
slice constraint: t1 = t2 ∧ t2 = t3 ∧ t1 = k
tag constraint: 0 ≤ t1 ≤ p-1
→ simplified constraint: t1 = t2 = t3
number of instances: 1

getS2A
from step: (s2: k, j) gets [L: j, k, k]
from item: [L: t1, t2, t3] is gotten by (s2: k, j)
slice constraint: t2 = t3 ∧ t3 = k ∧ t1 = j
tag constraint: 0 ≤ t3 ≤ p-1 ∧ t3+1 ≤ t1 ≤ p-1
→ simplified constraint: t2 = t3 ∧ t1 > t3
number of instances: 1

getS2B
from step: (s2: k, j) gets [L: k, k, k+1]
from item: [L: t1, t2, t3] is gotten by (s2: k, j)
slice constraint: t1 = t2 ∧ t2 = k ∧ t3 = k+1
tag constraint: 0 ≤ t2 ≤ p-1 ∧ t2+1 ≤ t1 ≤ p-1
→ simplified constraint: t1 = t2 ∧ t3 = t2 + 1
number of instances: p-t2-1

getS3A
from step: (s3: k, j, i) gets [L: j, i, k]
from item: [L: t1, t2, t3] is gotten by (s3: k, j, i)
slice constraint: t1 = j ∧ t2 = i ∧ t3 = k
tag constraint: 0 ≤ t3 ≤ p-1 ∧ t3 ≤ t1 ≤ p-1 ∧ t3+1 ≤ t2 ≤ t1
→ simplified constraint: t1 > t3 ∧ t2 > t3
number of instances: 1

getS3B
from step: (s3: k, j, i) gets [L: j, k, k+1]
from item: [L: t1, t2, t3] is gotten by (s3: k, j, i)
slice constraint: t1 = j ∧ t2 = k ∧ t3 = k+1
tag constraint: 0 ≤ t2 ≤ p-1 ∧ t2+1 ≤ t1 ≤ p-1
→ simplified constraint: t3 = t2 + 1 ∧ t1 > t2
number of instances: 1

getS3C
from step: if i != j (s3: k, j, i) gets [L: i, k, k+1]
from item: [L: t1, t2, t3] is gotten by (s3: k, j, i)
slice constraint: t1 = i ∧ t2 = k ∧ t3 = t2+1 ∧ (i ≤ j ∨ i ≤ j)
tag constraint: 0 ≤ t2 ≤ p-1 ∧ t2+1 ≤ t1 ≤ p-1
→ simplified constraint: t3 = t2 + 1 ∧ t1 > t2
number of instances: p-t2-2

*Intel and the Intel logo are registered trademarks of Intel Corporation. Other brands and names are the property of their respective owners
Function to compute `getCount` for an item instance from simplified constraints

```plaintext
getCount = 0;

// getS1A
if(t1 == t2 && t2 == t3)
    getCount = getCount + 1;

// getS2A
if(t2 == t3 && t1 > t3)
    getCount = getCount + 1;

// getS2B
if(t1 == t2 && t3 == t2 + 1)
    getCount = getCount + p - t2 - 1;

// getS3A
if(t1 > t3 && t2 > t3)
    getCount = getCount + 1;

// getS3B
if(t2 + 1 == t3 && t1 > t2)
    getCount = getCount + 1;

// getS3C
if(t2 + 1 == t3 && t1 > t2)
    getCount = getCount + p - t2 - 2;
```

- When an item instance is produced, execute this function to determine the `getCount` of this specific instance.
- The count is decremented at each `get`.
- It is removed as garbage when count hits zero.
Futures

• Simplify the user-supplied slice specification automatically
• Generate the getCount function from the simplified specification
• Generate the slice specification automatically from the CnC graph spec
• Extend the approach to more dynamic applications