Hierarchical CnC

Kath Knobe
Thanks to ...

• Zoran Budimlić – Rice
• Nick Vrvilo – Rice
• Frank Schlimbach – Intel
• Milind Kulkarni – Purdue
• Gary Delp – Mayo Clinic
High level motivation

• Software engineering
• Hierarchical understanding
• Hierarchical optimizations
• Hierarchical mapping
• Reuse
  – Within a single app or from a library
  – Communicating runtimes
• Hierarchy is not only for computation but also hierarchical
  – Documentation, development, testing, debugging, checkpoint-continue, static & dynamic analysis, static & dynamic tuning, etc.
No assumption about the implementation

• Some languages know about arrays, lists, strings
• CnC knows about collections, graphs, tags
• We have a variety of different implementations of these
• Hierarchical CnC will also know about hierarchy
• We can have a variety of very different implementation of hierarchy
• Even the runtimes can be different at different places or levels in the graph
Outline

• Background via an app
• Introduction to hierarchy
• Constraints and optimizations
Background via an app
<table>
<thead>
<tr>
<th>Cholesky</th>
<th>Trisolve</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cholesky factorization

<table>
<thead>
<tr>
<th>Cholesky</th>
<th>Trisolve</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cholesky factorization

<table>
<thead>
<tr>
<th></th>
<th>Cholesky</th>
<th>Trisolve</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Cholesky factorization

<table>
<thead>
<tr>
<th>Cholesky</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Trisolve</td>
<td>Update</td>
</tr>
</tbody>
</table>

![Diagram showing the process]

**Diagram Legend:**
- **CnC:** Indicates a central node or core element.
# Cholesky factorization

<table>
<thead>
<tr>
<th>Cholesky</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Trisolve</td>
<td>Update</td>
</tr>
</tbody>
</table>

- [Arrows pointing right] indicate the process flow.
Cholesky factorization

<table>
<thead>
<tr>
<th>Cholesky</th>
<th>Trisolve</th>
<th>Update</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CnC
Cholesky CnC graph spec
Cholesky CnC graph spec

Producer relation

Consumer relation

(C iter)

[C: iter]

(T: iter, row)

[T: iter, row]

(U: iter, row, col)

[U: iter, row, col]
Cholesky CnC graph spec

Input:
Produced by env

Output:
Consumed by env
Cholesky CnC graph spec

<control tags>
Which instances will execute
Cholesky CnC graph spec

Control relationship
Semantics of CnC: specifies a partial order of execution
Semantics of CnC: specifies a partial order of execution
Semantics of CnC:
specifies a partial order of execution
Semantics of CnC: specifies a partial order of execution
Semantics of CnC: specifies a partial order of execution

Termination:
- No step is executing
- All ready steps have executed
- All inputs have arrived

Clean termination:
- Termination
- All control-ready steps have executed
Introduction to hierarchy
Basic idea

- Every level appears to be a normal CnC app
- But now includes the relationships between adjacent levels
Current fixed 3 level hierarchy

This is at a high level Implementations can vary dramatically
General hierarchy looks like flat CnC at every level

This is at a high level
Implementations can vary dramatically

Full app

Collection name

Collection name

Collection name

Instance tag

Instance tag

Instance tag

Instance tag

Instance tag

Instance tag

Instance tag

Instance tag

Instance tag
# Types of hierarchical relationships

4 possibilities

- **Applied to:**
  - Computation / data

- **Types of decomposition:**
  - Heterogeneous / homogeneous

<table>
<thead>
<tr>
<th>Homogeneous</th>
<th>Data</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>([x: j, k] = [x: j', k'])</td>
<td>((\text{foo: } j, k) = (\text{foo: } j', k'))</td>
</tr>
<tr>
<td></td>
<td>Special case: ([x: j] = [x: j, k])</td>
<td>Special case: ((\text{foo: } j) = (\text{foo: } j, k))</td>
</tr>
</tbody>
</table>

**SIMD:** Single instruction / multiple data

Usually means “in parallel”.

Here it says nothing about parallelism.
Types of hierarchical relationships

4 possibilities
- Applied to:
  Computation / data
- Types of decomposition:
  Heterogeneous / homogeneous

<table>
<thead>
<tr>
<th></th>
<th>Data</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homogeneous</td>
<td>[x: j, k] = [x: j', k']</td>
<td>(foo: j, k) = (foo: j', k')</td>
</tr>
<tr>
<td></td>
<td>Special case:</td>
<td>Special case:</td>
</tr>
<tr>
<td></td>
<td>[x: j] = [x: j, k]</td>
<td>(foo: j) = (foo: j, k)</td>
</tr>
<tr>
<td>Heterogeneous</td>
<td>[x: j] = [y: j], [z: j]</td>
<td>(foo: j) = {graph: j}</td>
</tr>
</tbody>
</table>

MIMD: Multiple instruction / multiple data
Usually means “in parallel”.
Here it says nothing about parallelism.

Like an array
Like SIMD
Like a struct
Like MIMD
Decompositions

distinct collections
Same color = same name = same collection
Cholesky CnC graph spec
This is a homogeneous decomposition of (CTU) into children (CTU:iter). These children all look the same for different values of iter.
This is a further (heterogeneous) decomposition of (CTU: iter)
Into 2 distinct computation steps (CT: iter) and (U: iter, row, col)
This is a further (heterogeneous) decomposition of (CT: iter) into 2 distinct computation steps (C: iter) and (T: iter, row)
Semantics of flat CnC:

Termination:
- No step is executing
- All ready steps have executed
- All inputs have arrived

Clean termination:
- Termination
- All control-ready steps have executed
Semantics of hierarchical CnC: specifies a partial order of execution

**Termination:**
- No step is executing
- All ready steps have executed
- All inputs have arrived

**Clean termination:**
- Termination
- All control-ready steps have executed

Lower level graph
Semantics of hierarchical CnC:
specifies a partial order of execution

Termination:
- No step is executing
- All ready steps have executed
- All inputs have arrived

Clean termination:
- Termination
- All control-ready steps have executed

Lower level graph
*terminated*
Semantics of hierarchical CnC:
specifies a partial order of execution

Termination:
- No step is executing
- All ready steps have executed
- All inputs have arrived

Clean termination:
- Termination
- All control-ready steps have executed

Lower level graph
terminated
Semantics of hierarchical CnC: specifies a partial order of execution

Termination:
- No step is executing
- All ready steps have executed
- All inputs have arrived

Clean termination:
- Termination
- All control-ready steps have executed

Lower level graph

terminated
Semantics of hierarchical CnC: specifies a partial order of execution

Termination:
- No step is executing
- All ready steps have executed
- All inputs have arrived

Clean termination:
- Termination
- All control-ready steps have executed

Lower level graph terminated
Hierarchical nodes that are not step-like

Like a sub-graph of a larger graph or like an app

• Doesn’t need to live by the in/compute/out rule
  – Most of our current apps are step-like

• The Intel system supports non-step like nodes as subgraphs of a larger graph
  – Examples reductions, joins
Reuse

• We want to reuse hierarchical nodes
  – Multiple times within the same app or from libraries
  – Step-like or graph-like
  – Either case innards can be public or private

• If public
  – As if it were built for the app itself
  – Can analyze and optimized wrt its position

• If private
  – It’s a black box
  – Can optimize it as a whole (move or delete)
Constraints and optimizations
Constraints on hierarchy

• Every level of a hierarchical CnC spec is a legal CnC spec:
  – Steps at every level must be step-like:
    • Can get all their input, compute, put their output and terminate
  – Data items at every level must item-like
    • Are dynamic single-assignment
• The meaning of the parent node is identical to the meaning of the children taken as a whole
• Implication:
  – Parent/child relationship of steps and the parent/child relationship of items must be consistently determined
One example of an optimization: Interchange

Computations

• 4 parent/child combinations
  – SIMD of SIMD
  – MIMD of MIMD
  – SIMD of MIMD
  – MIMD of SIMD

Interchange is legal
if the result is step-like at both levels

Data

• 4 parent/child combinations
  – Struct of structs
  – Struct of arrays
  – Array of structs
  – Array of arrays

Interchange is legal
if the result DSA at both levels
Conclusions

• Hierarchy is useful for the domain expert and for the tuning expert

• Hierarchy is not only for computation but also hierarchical
  – Discussed: Static & dynamic analysis, static & dynamic tuning, etc.
  – But also: Documentation, development, testing, debugging, checkpoint-continue,
End

Thanks to ...

Zoran Budimlić – Rice
Nick Vrvilo – Rice
Frank Schlimbach – Intel
Milind Kulkarni – Purdue
Gary Delp – Mayo Clinic