Asynchronous Parallel DLA in Concurrent Collections

Aparna Chandramowlishwaran, Richard Vuduc – Georgia Tech

Kathleen Knobe – Intel

May 14, 2009

Workshop on Scheduling for Large-Scale Systems @ UTK

hpcgarage

Motivation and goals

- Motivating recent work for multicore systems
  - Tile algorithms for DLA, *e.g.*, Buttari, *et al*. (2007); Chan, *et al*. (2007)
  - General parallel programming models suited to this algorithmic style, *e.g.*, Concurrent Collections (CnC) by Knobe & Offner (2004)
- Goals
  - Study: Apply and evaluate CnC using PDLA examples
  - Talk: CnC tutorial crash course; platform for your work?

To download CnC, see: whatif.intel.com

Outline

- Overview of the Concurrent Collections (CnC) language
- Asynchronous parallel Cholesky & symmetric eigensolver in CnC
- Experimental results (preliminary)

3

Concurrent Collections (CnC) programming model

- Separates computation semantics from expression of parallelism
- Program = components + scheduling constraints
  - Components: **Computation**, **control**, **data**
  - Constraints: **Relations** among components
  - No overwriting of data, no arbitrary serialization, and no side-effects
- Combines tuple-space, streaming, and dataflow models

CnC example: Outer product

$$Z \leftarrow x \cdot y^T$$

5

CnC example: Outer product

$$Z \leftarrow x \cdot y^T$$

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Example only; coarser grain may be more realistic in practice.

CnC example: Outer product

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

CnC example: Outer product

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step Unit of execution

Set of all (dynamic) multiplications

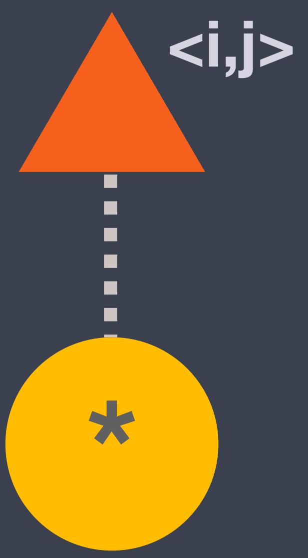
CnC example: Outer product

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Unit of execution

Control



<*a*, *b*, …> = tuple of tag components

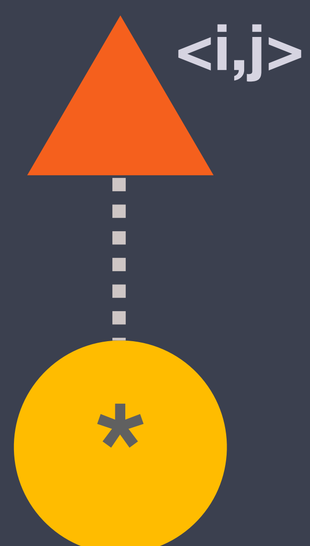
CnC example: Outer product

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step Unit of execution

Tag Control



Says ***whether***, not ***when***, step executes

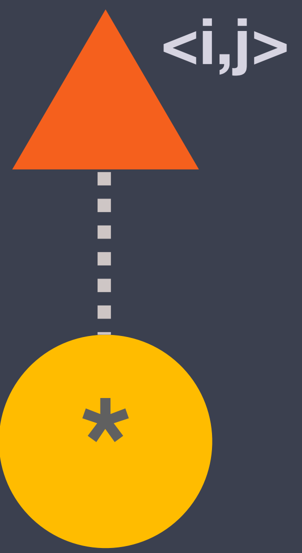
CnC example: Outer product

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step Unit of execution

Tag Control



Tags ***prescribe*** steps

CnC example: Outer product

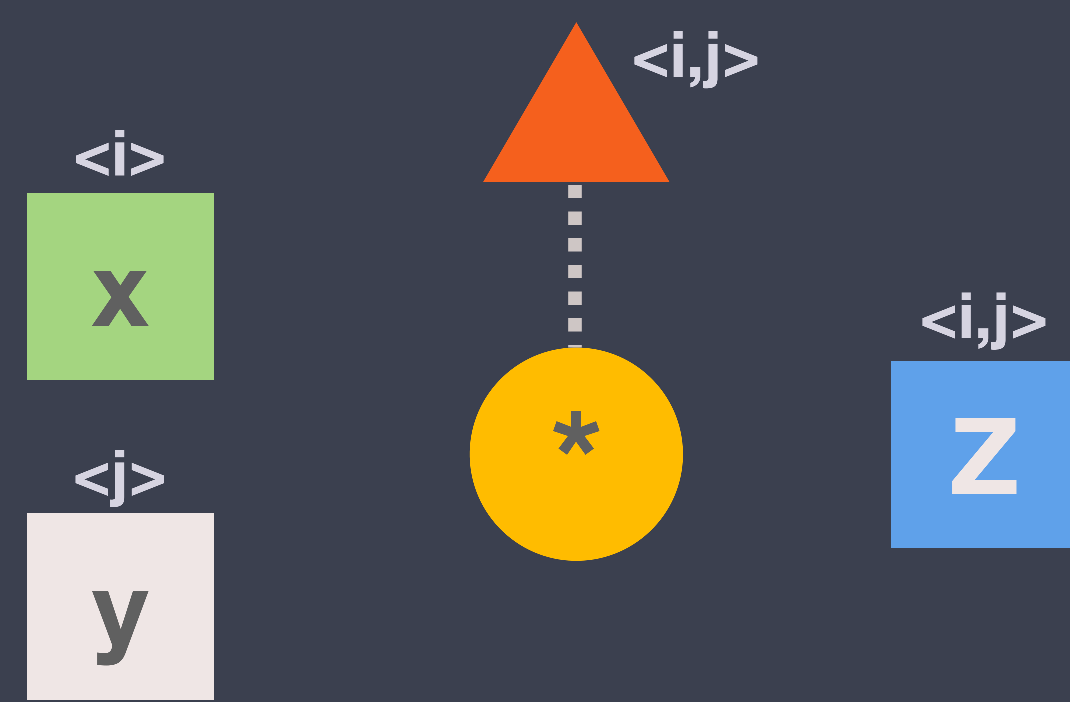
$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step — Unit of execution

Tag — Control

Item — Data



12

CnC example: Outer product

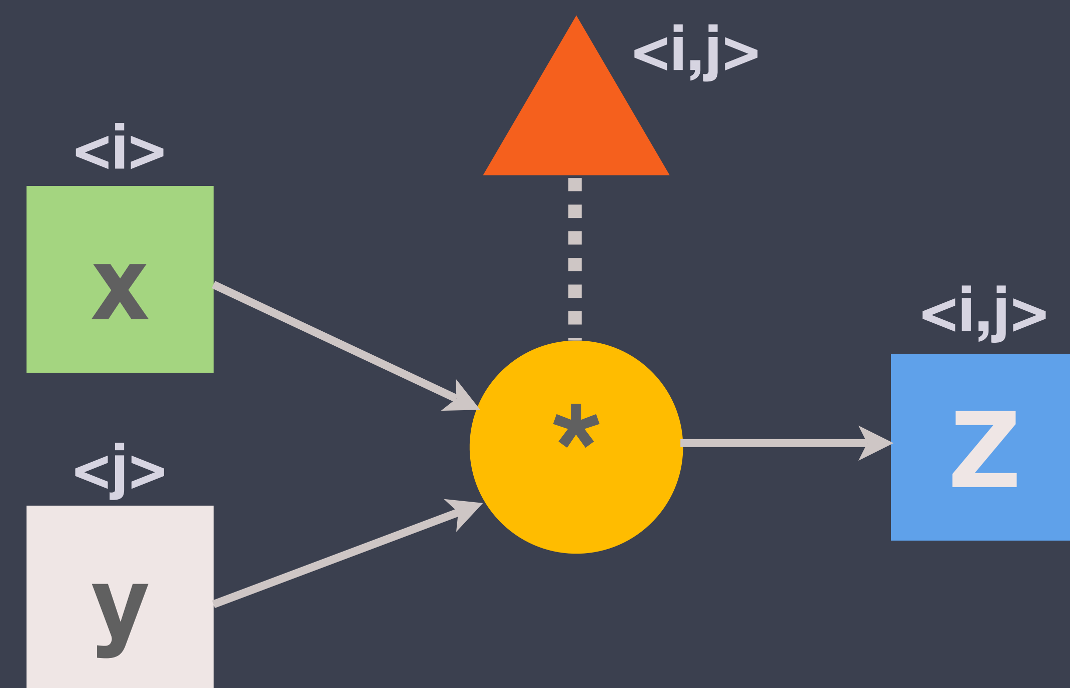
$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step Unit of execution

Tag Control

Item Data



→ shows producer/consumer relations

CnC example: Outer product

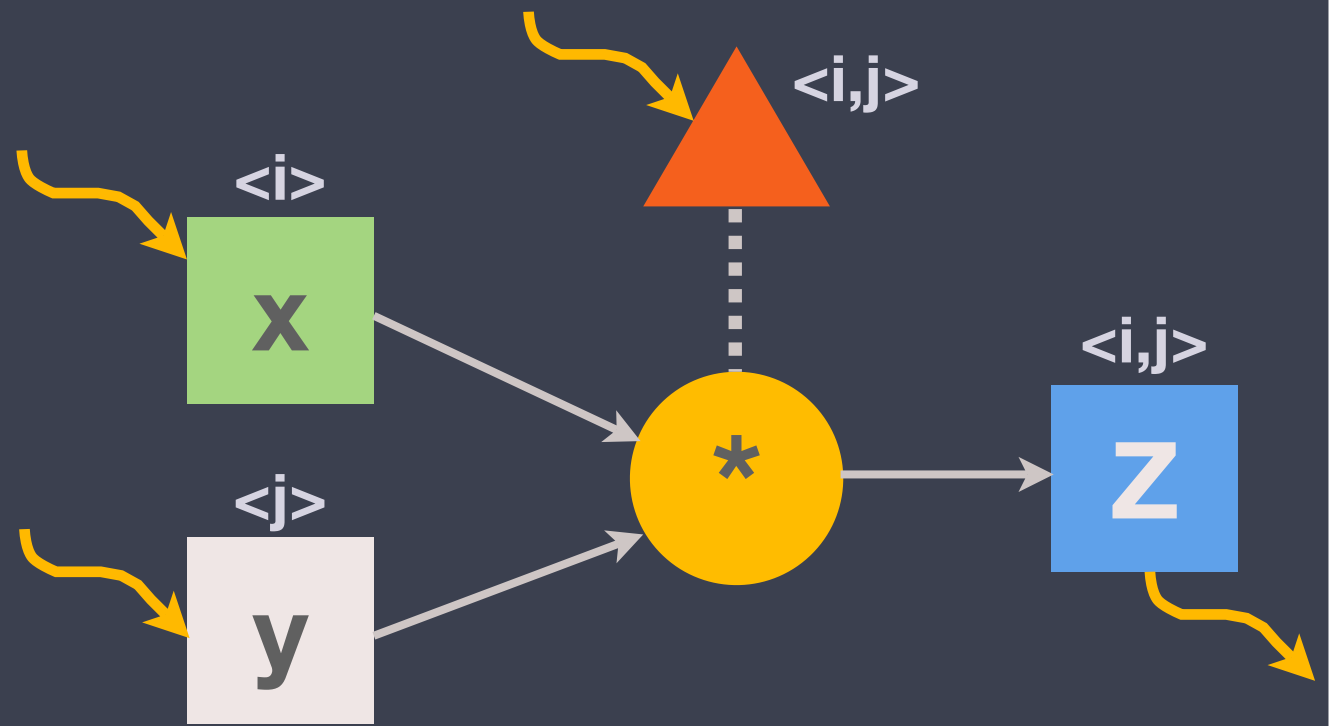
$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step Unit of execution

Tag Control

Item Data



"**Environment**" may produce/consume

Essential properties of a CnC program

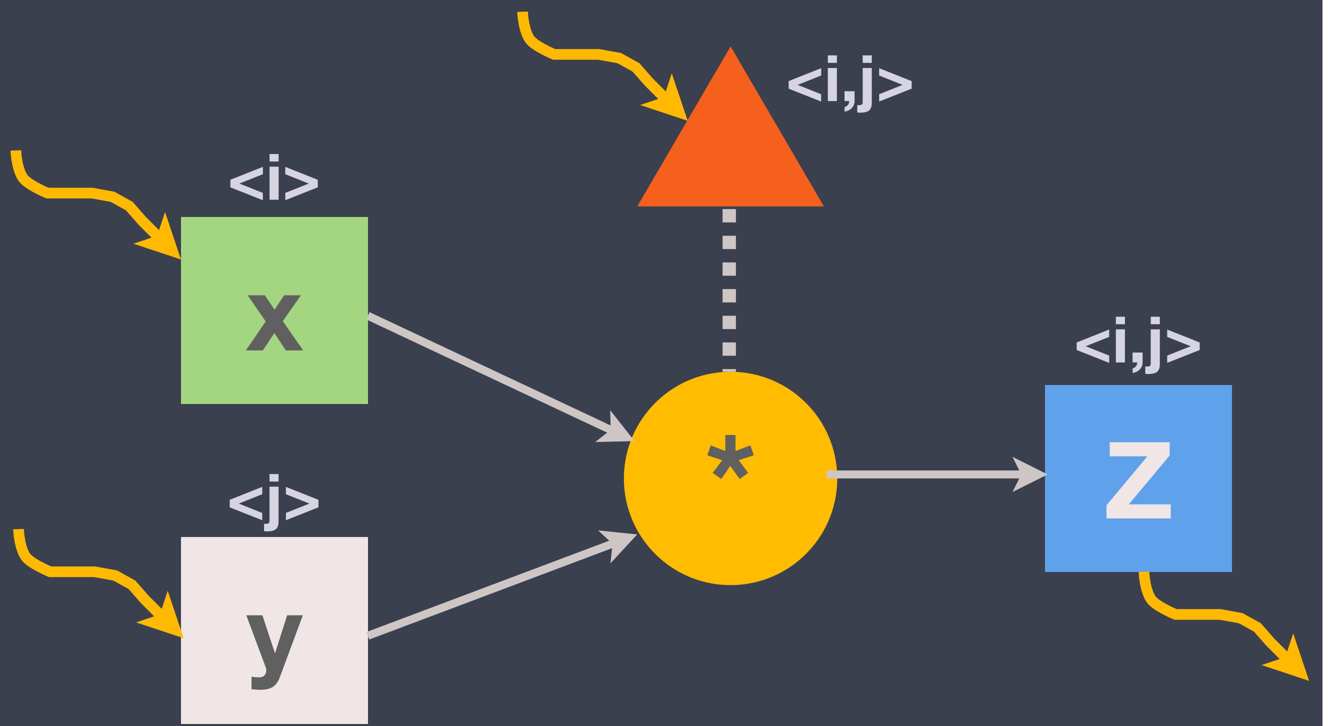
- Written in terms of values, without overwriting ⇒ race-free

  (*dynamic single assignment*)

- No arbitrary serialization, only explicit ordering constraints
  (*avoids analysis*)

- Steps are side-effect free (*functional*)

$$z_{i,j} \leftarrow x_i \cdot y_j$$



CnC example: Tree search

match ← **find** (value **x** in tree **T**)

Collections: Static representation of dynamic ***instances***

Step Unit of execution

Tag Control

Item Data

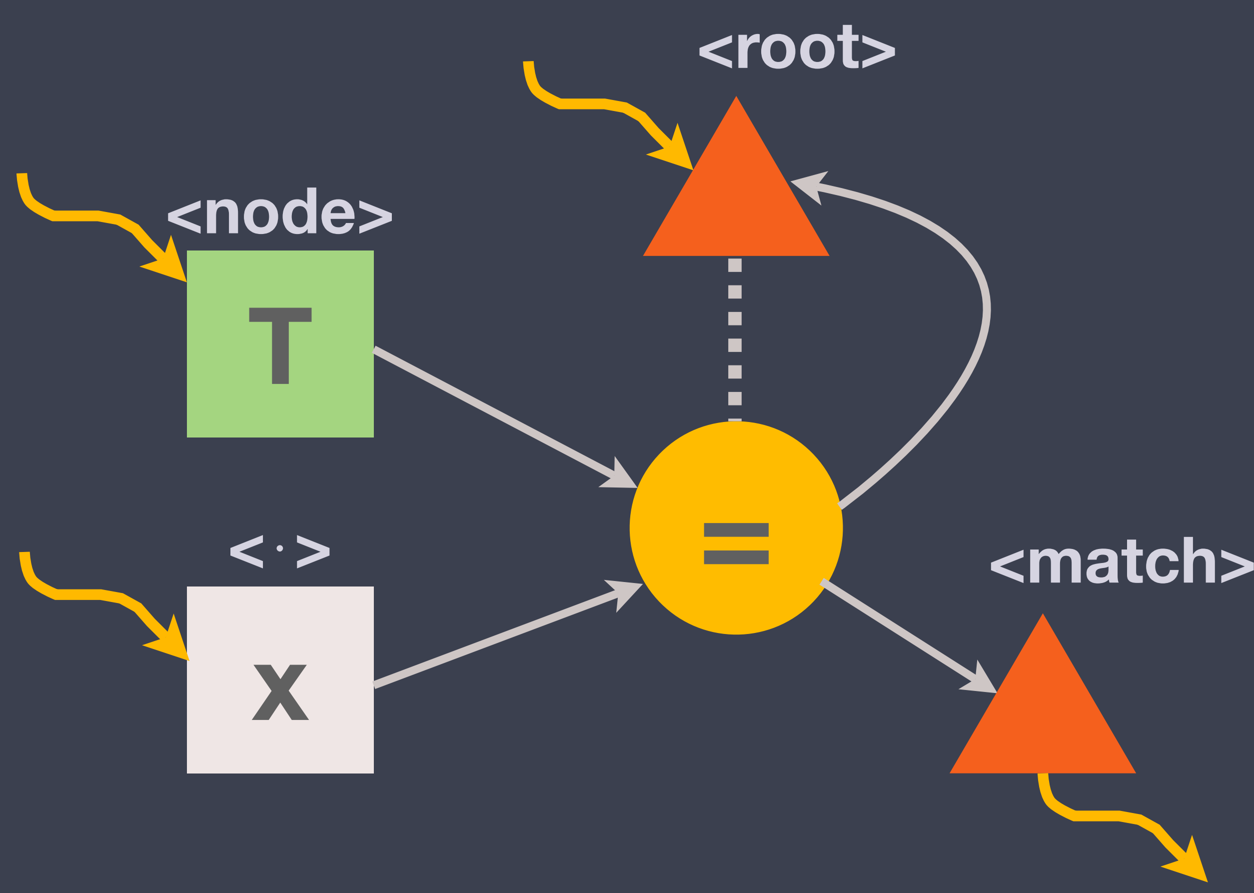
16

CnC example: Tree search Controller/controlee relations

match ← **find** (value **x** in tree **T**)

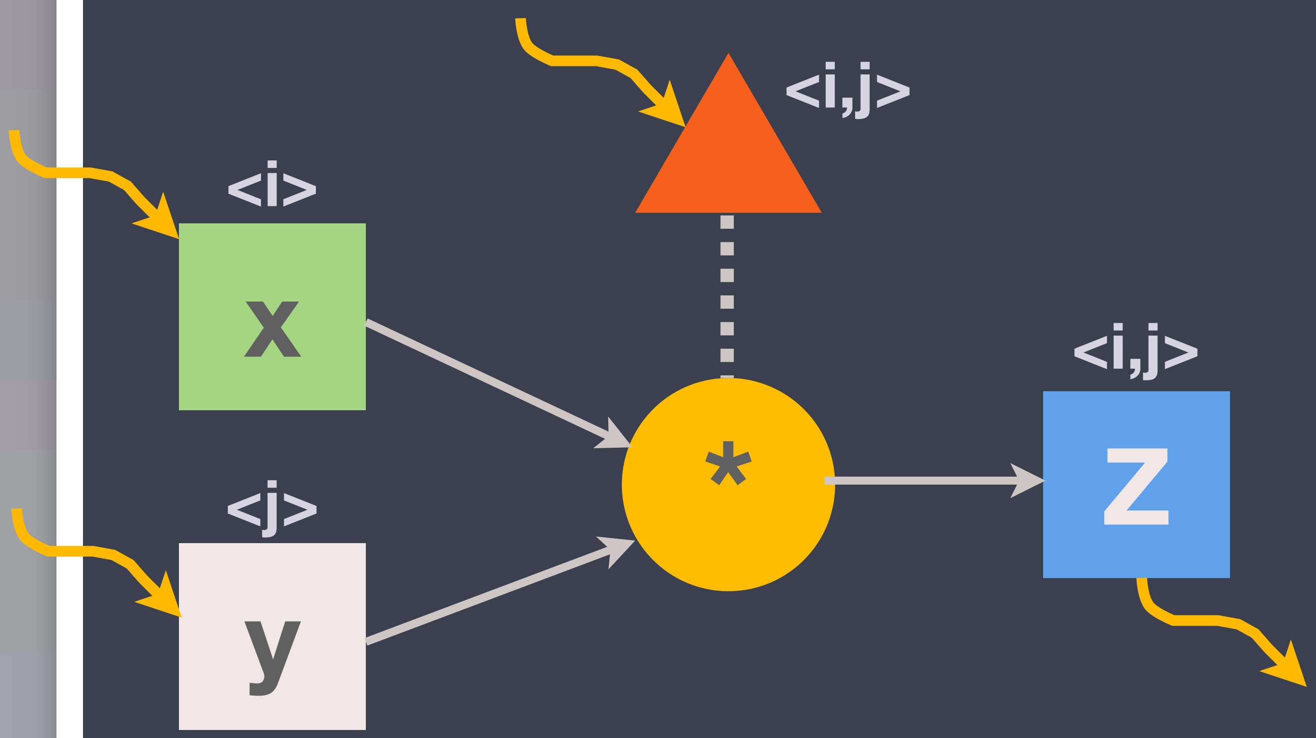
Collections: Static representation of dynamic ***instances***

Step	Unit of execution
Tag	Control
Item	Data



Execution model

$$z_{i,j} \leftarrow x_i \cdot y_j$$



Recall: Outer product example

Execution model

$$z_{i,j} \leftarrow x_i \cdot y_j$$

- Tag <i=2, j=5> ***available***

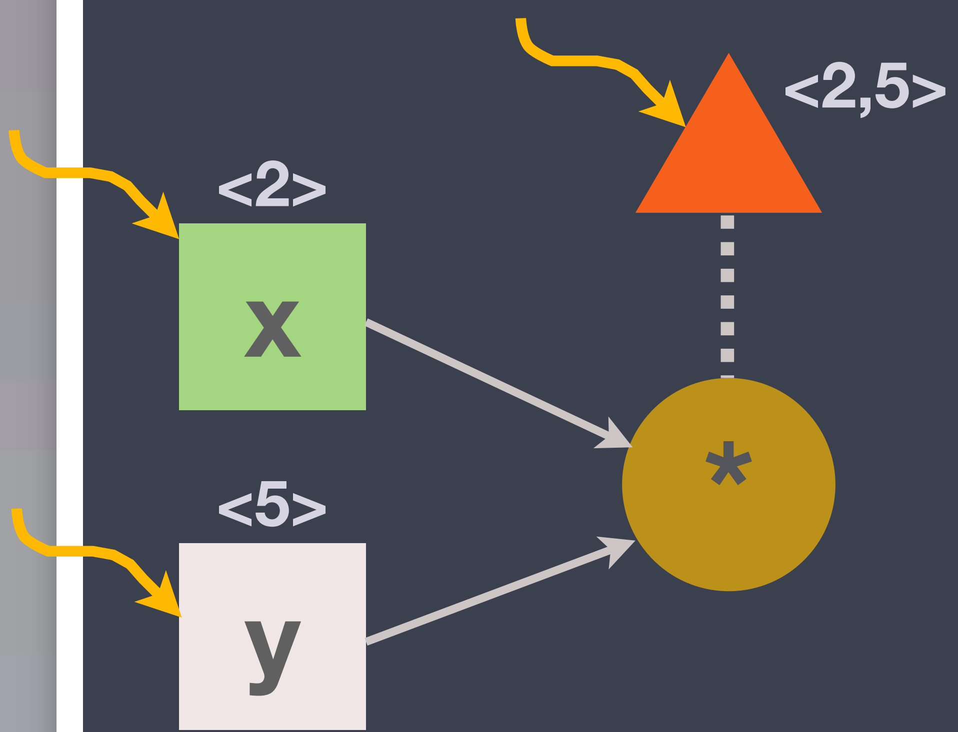
Execution model

$$z_{i,j} \leftarrow x_i \cdot y_j$$

- Tag <i=2, j=5> *available*
  ⇒ Step ***prescribed***

Execution model

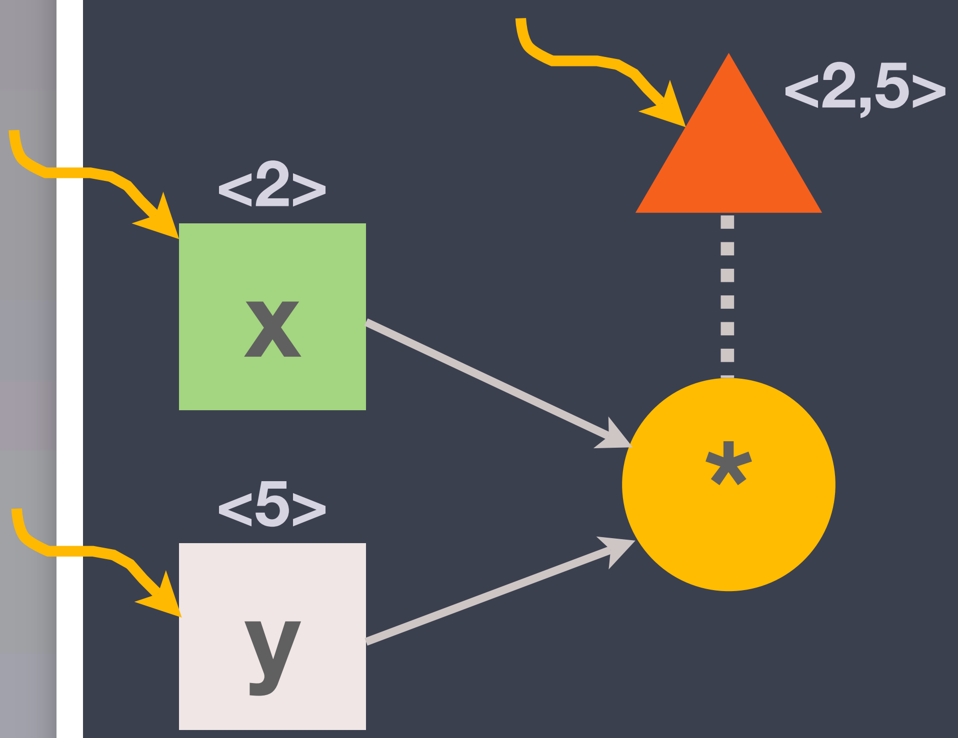
$$z_{i,j} \leftarrow x_i \cdot y_j$$



- Tag <2,5> *available*
  ⇒ Step *prescribed*
- Items x:<2>, y:<5> available
  ⇒ Step ***inputs-available***

Execution model

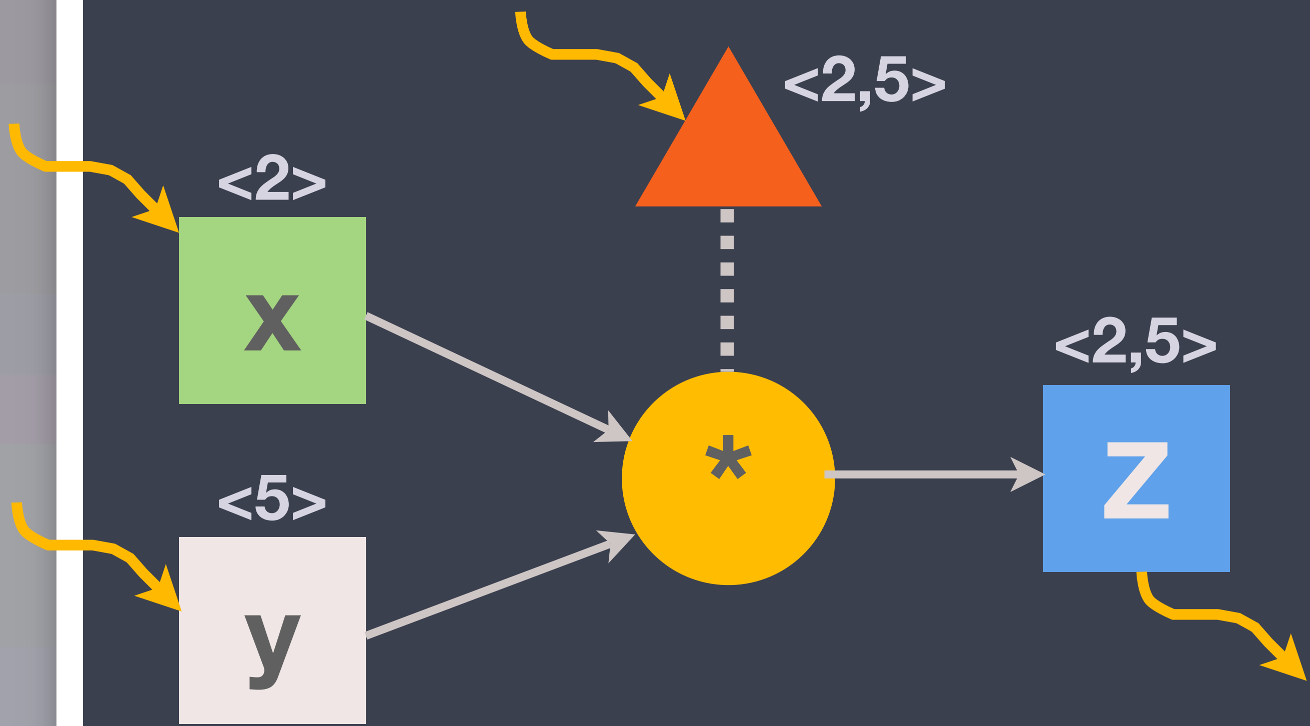
$$z_{i,j} \leftarrow x_i \cdot y_j$$



- Tag <2,5> *available*
  ⇒ Step *prescribed*
- Items x:<2>, y:<5> available
  ⇒ Step *inputs-available*
- *Prescribed* + *inputs-available*
  ⇒ ***enabled***

Execution model

$$z_{i,j} \leftarrow x_i \cdot y_j$$



- Tag <2,5> *available*
  ⇒ Step *prescribed*
- Items x:<2>, y:<5> available
  ⇒ Step *inputs-available*
- *Prescribed* + *inputs-available*
  ⇒ *enabled*
- Executes ⇒ Z:<2,5> ***available***

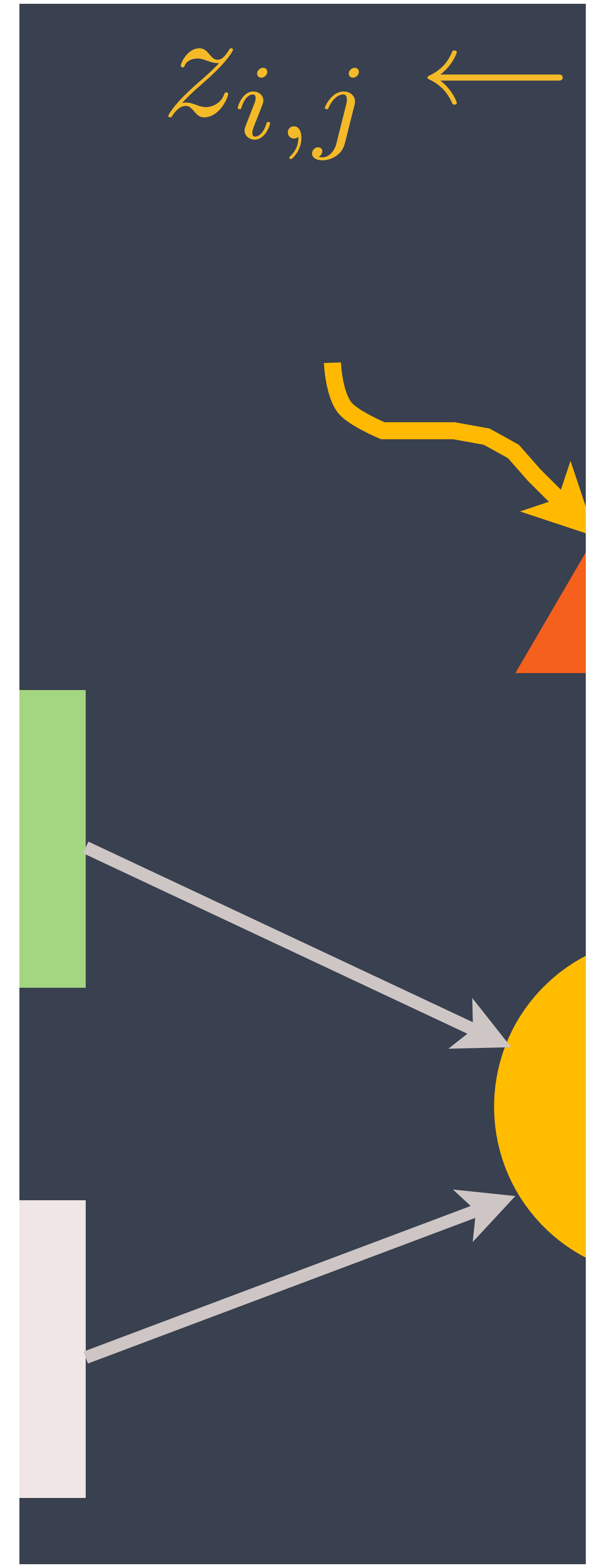
Coding and execution

[1] Write the specification (graph).

[2] Implement steps in a "base" language (C/C++).

[3] Build using CnC translator + compiler.

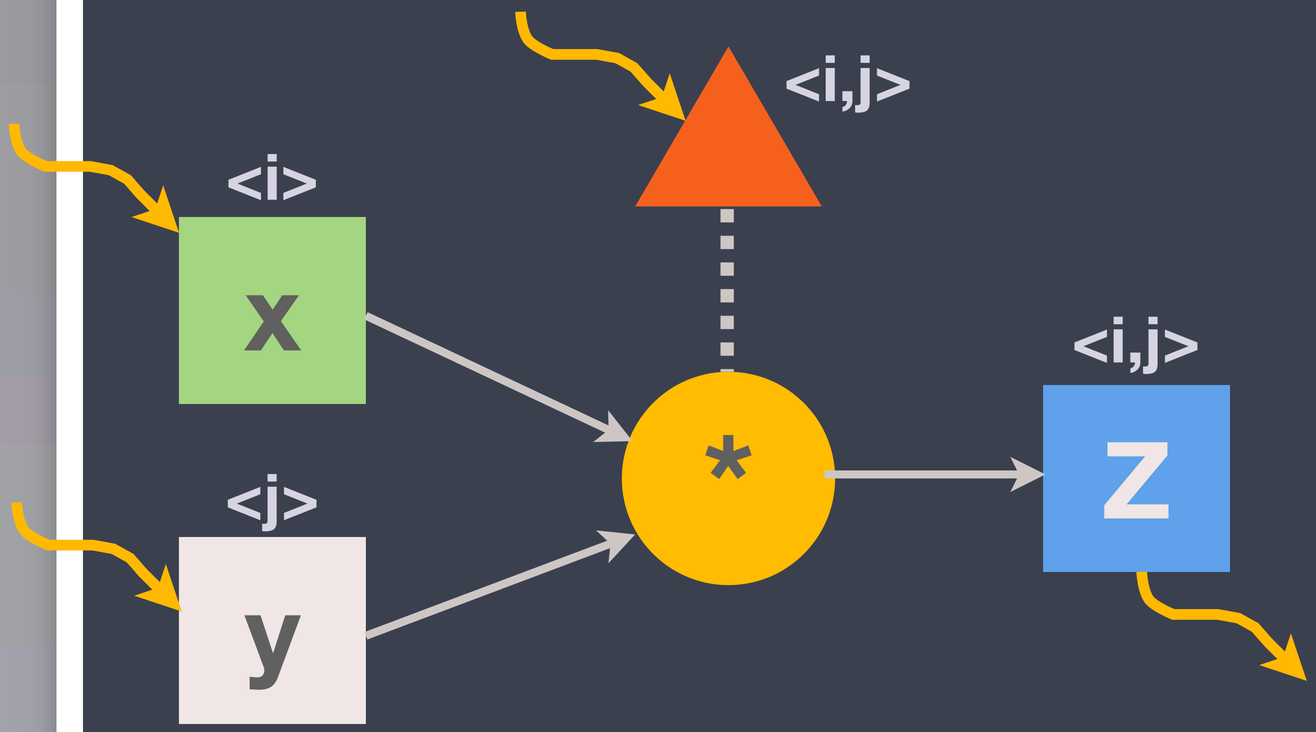
[4] Run-time system maintains collections and schedules step execution.



24

Textual notation

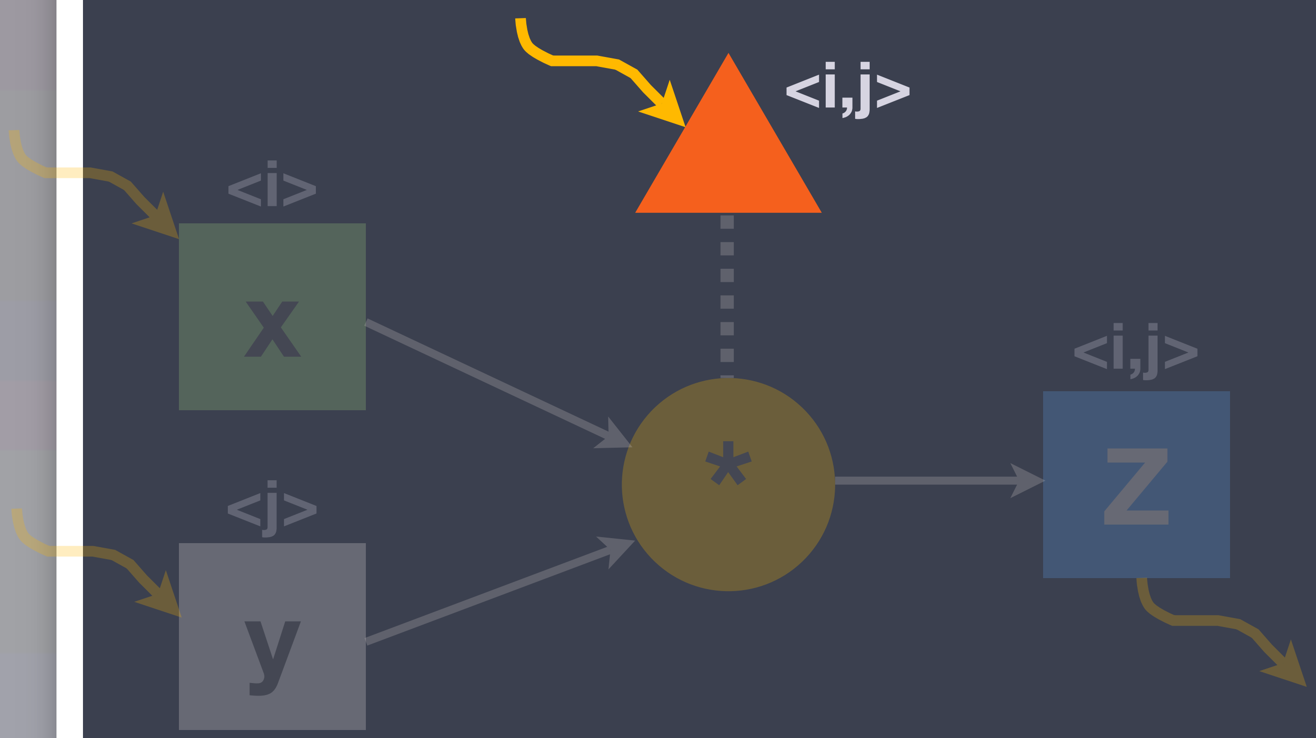
$$z_{i,j} \leftarrow x_i \cdot y_j$$



Recall: Outer product example

Textual notation

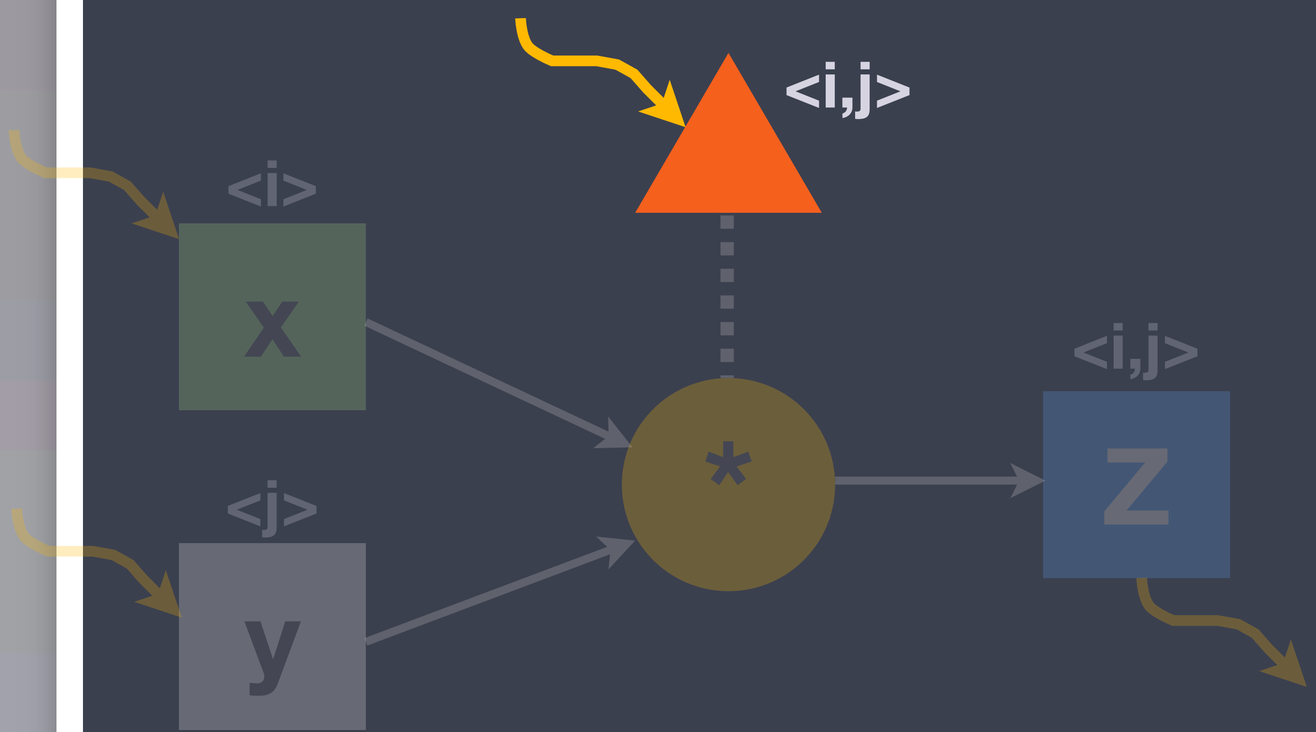
$$z_{i,j} \leftarrow x_i \cdot y_j$$



Textual notation

$$z_{i,j} \leftarrow x_i \cdot y_j$$

```
// Input:

   env → <*: i,j>;
```

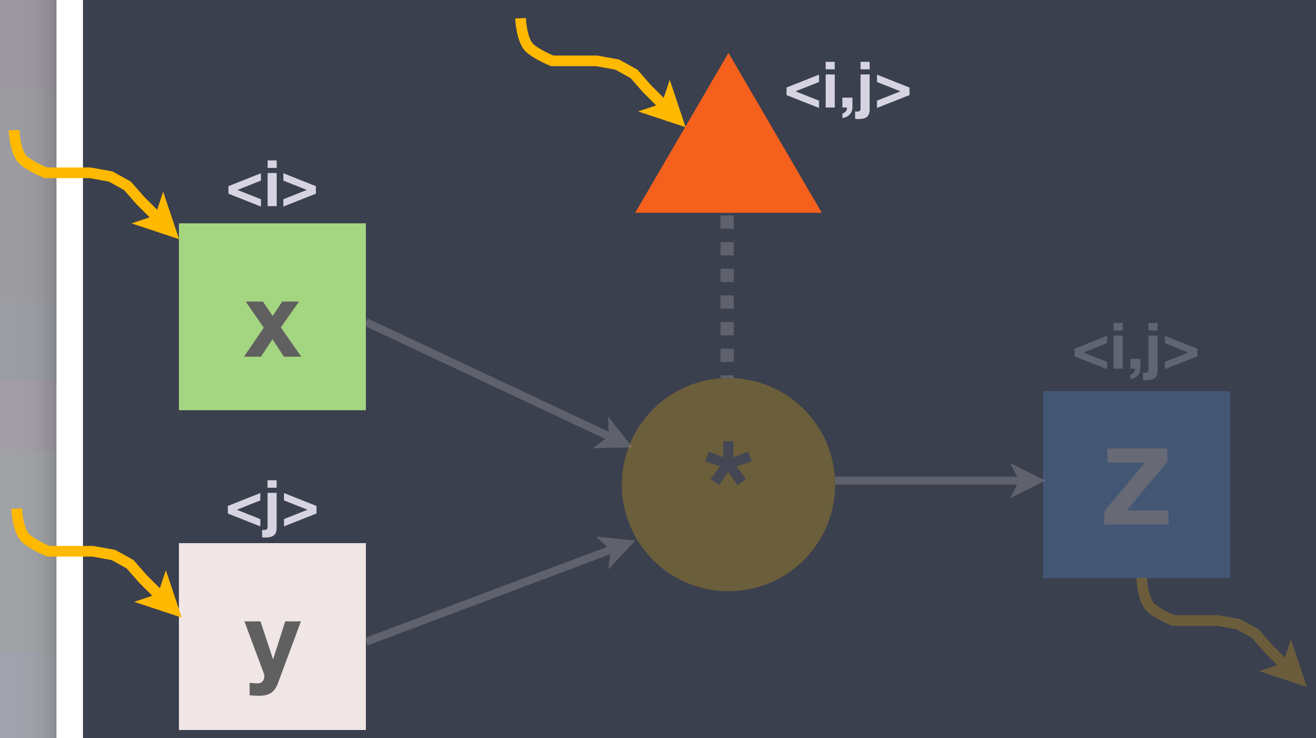


27

Textual notation

$$z_{i,j} \leftarrow x_i \cdot y_j$$

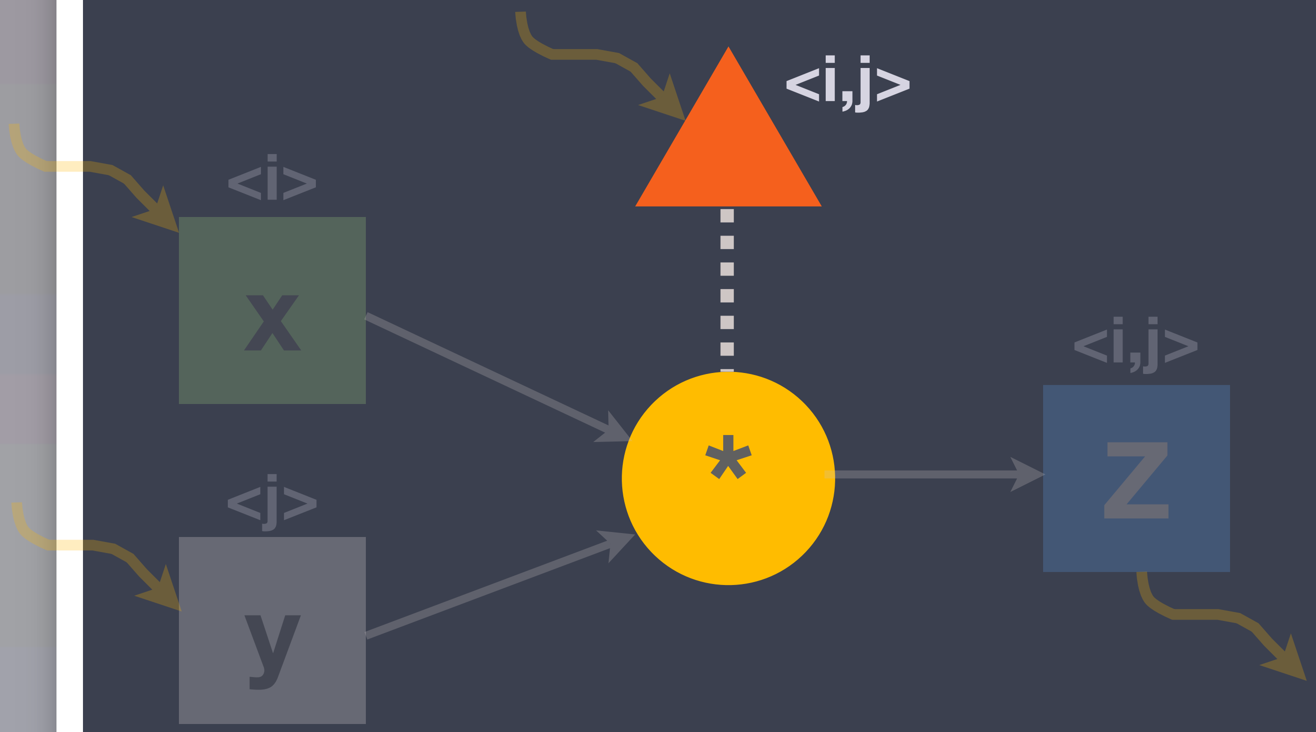
```
// Input:

   env → <*: i,j>, [x: i], [y: j];
```



Textual notation

$$z_{i,j} \leftarrow x_i \cdot y_j$$

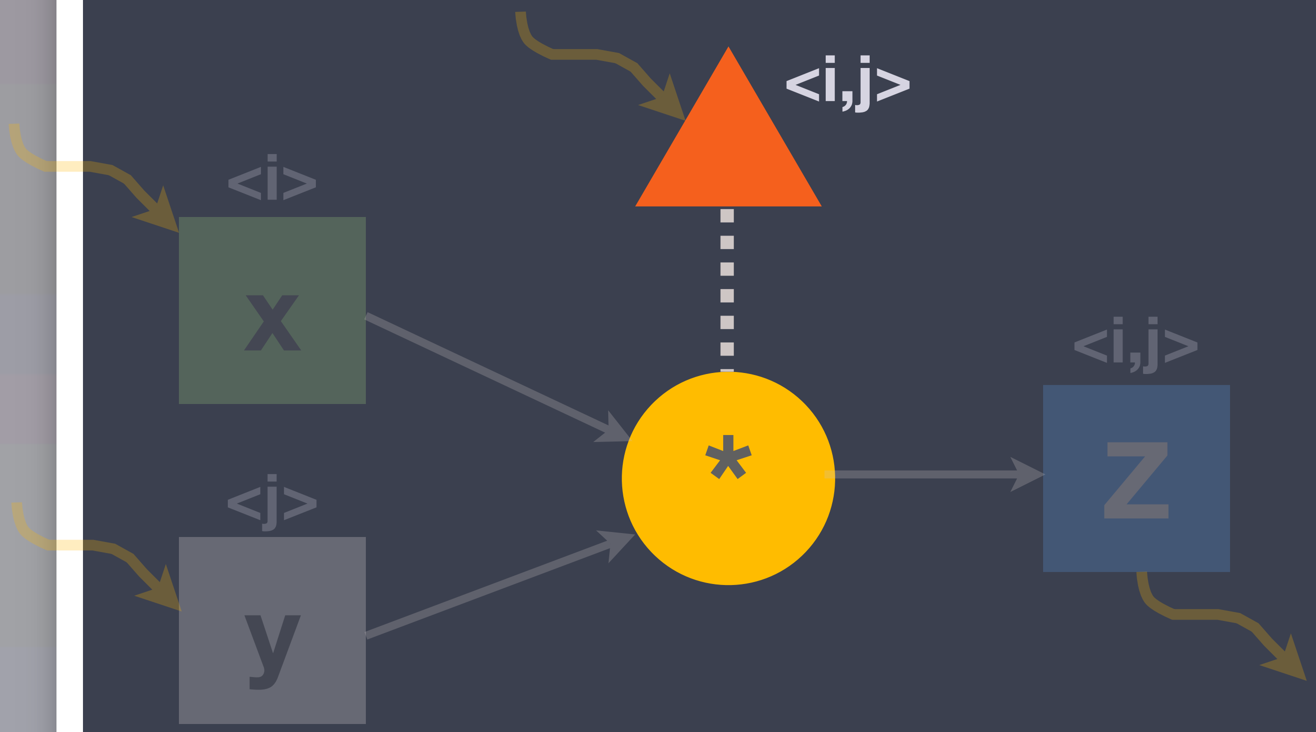
```
// Input:

   env → <*: i,j>, [x: i], [y: j];
```



29

Textual notation

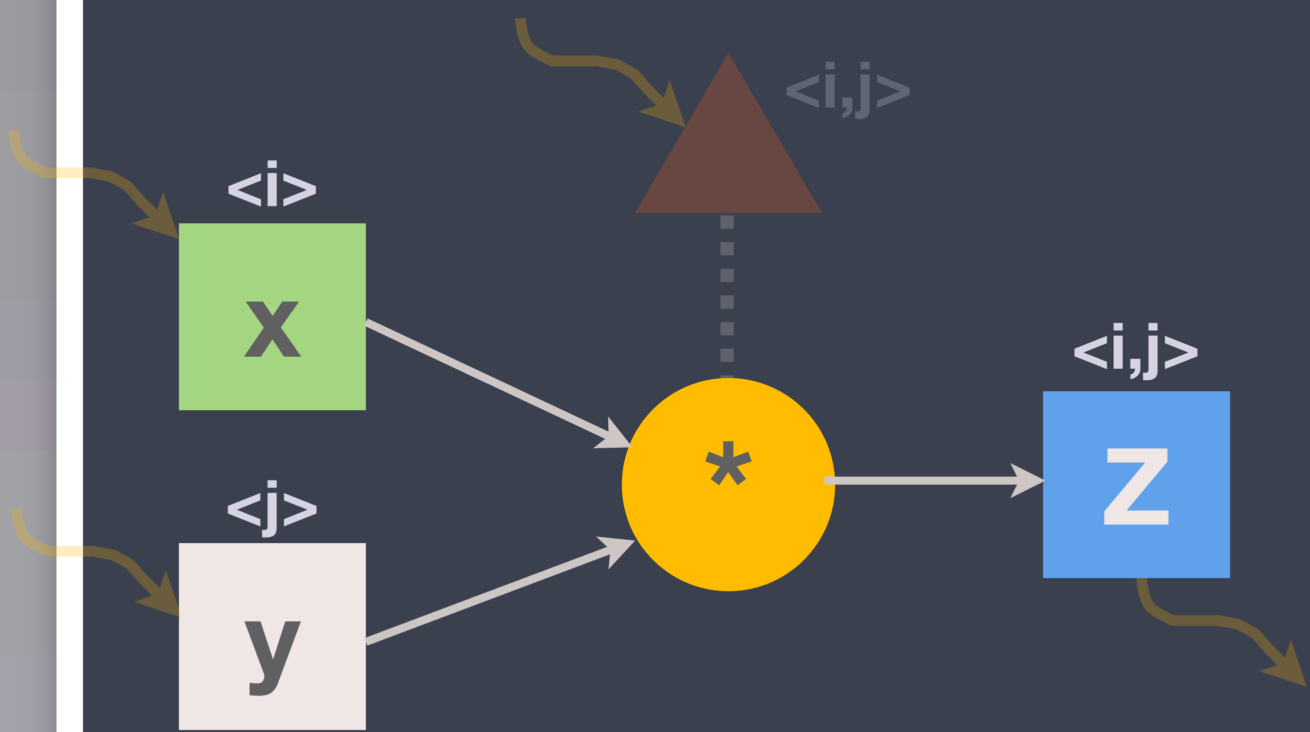
$$z_{i,j} \leftarrow x_i \cdot y_j$$




```
// Input:

    env → <*: i,j>, [x: i], [y: j];

// Prescription relations:

    <*: i,j> :: (*: i,j);
```

Textual notation

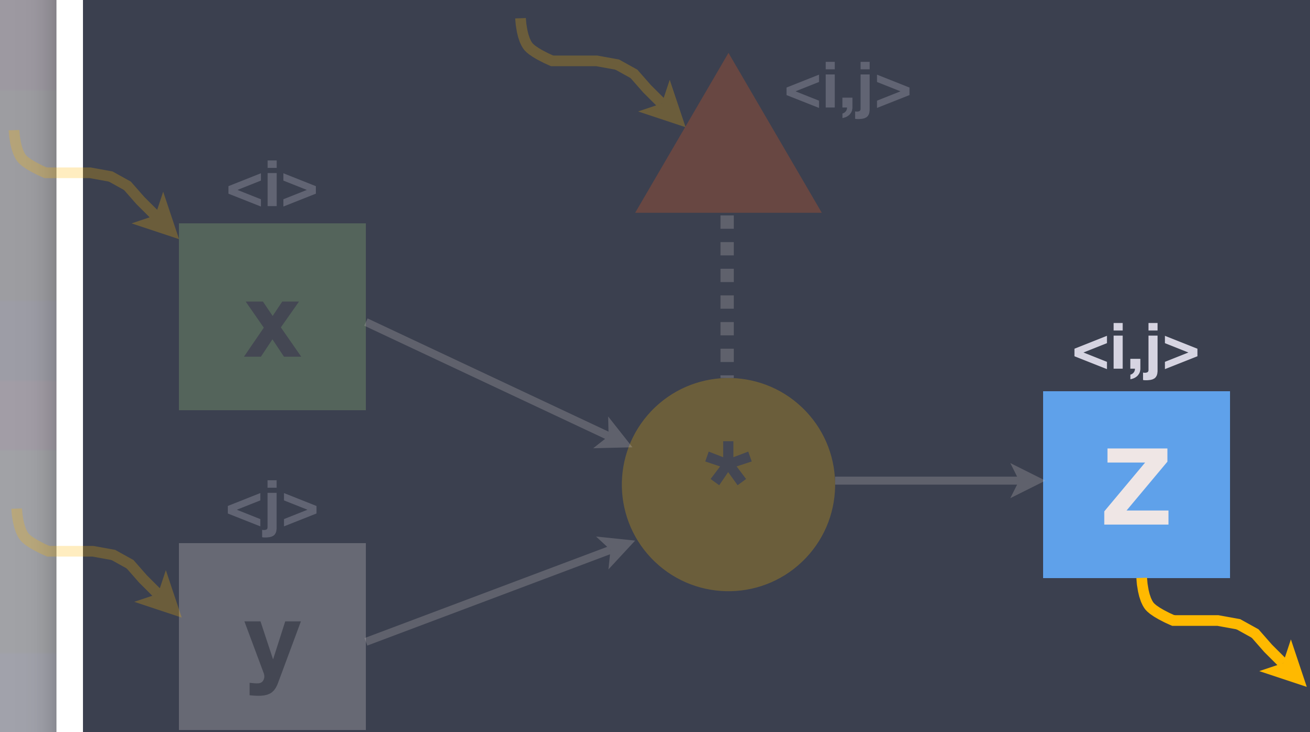
$$z_{i,j} \leftarrow x_i \cdot y_j$$




```
// Input:

    env → <*: i,j>, [x: i], [y: j];

// Prescription relations:

    <*: i,j> :: (*: i,j);

// Producer/consumer relations:

    [x: i], [y: j] → (*: i, j);

    (*: i, j) → [Z: i, j];
```

Textual notation

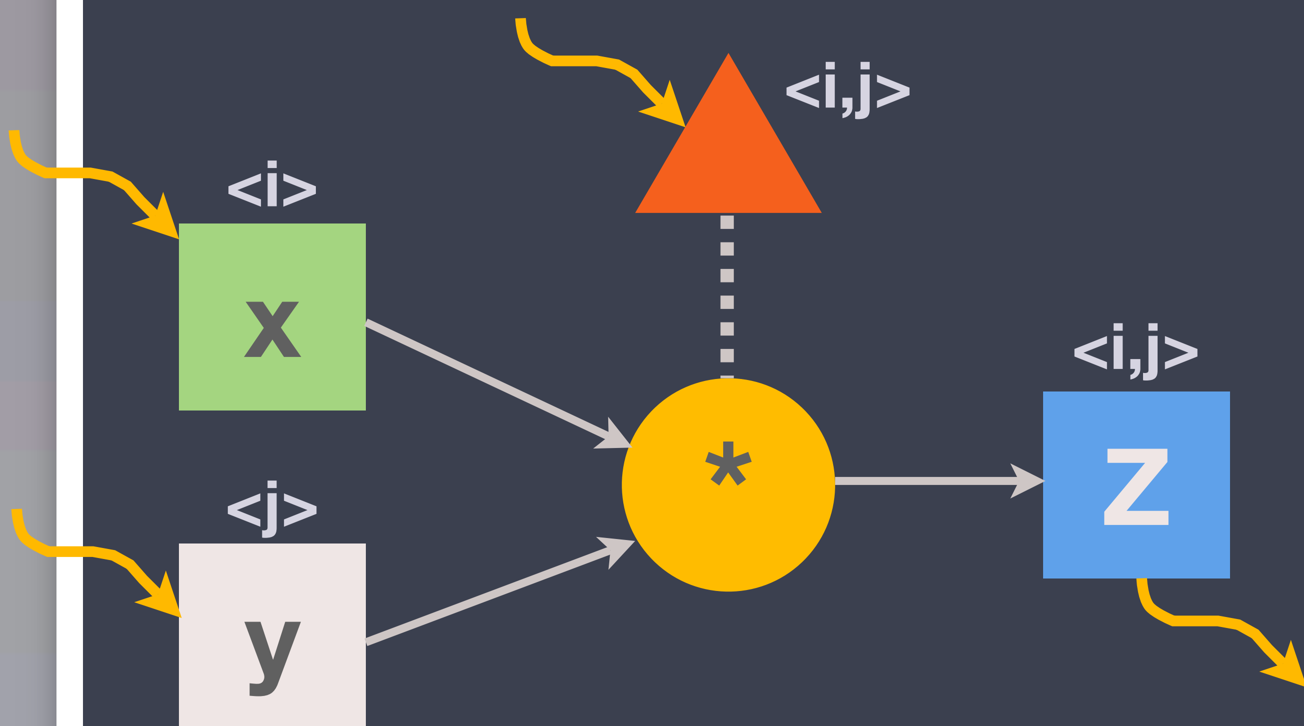
$$z_{i,j} \leftarrow x_i \cdot y_j$$




```
// Input:

   env → <*: i,j>, [x: i], [y: j];

// Prescription relations:

   <*: i,j> :: (*: i,j);

// Producer/consumer relations:

   [x: i], [y: j] → (*: i, j);

   (*: i, j) → [Z: i, j];

// Output:

   [Z: i, j] → env;
```

Textual notation

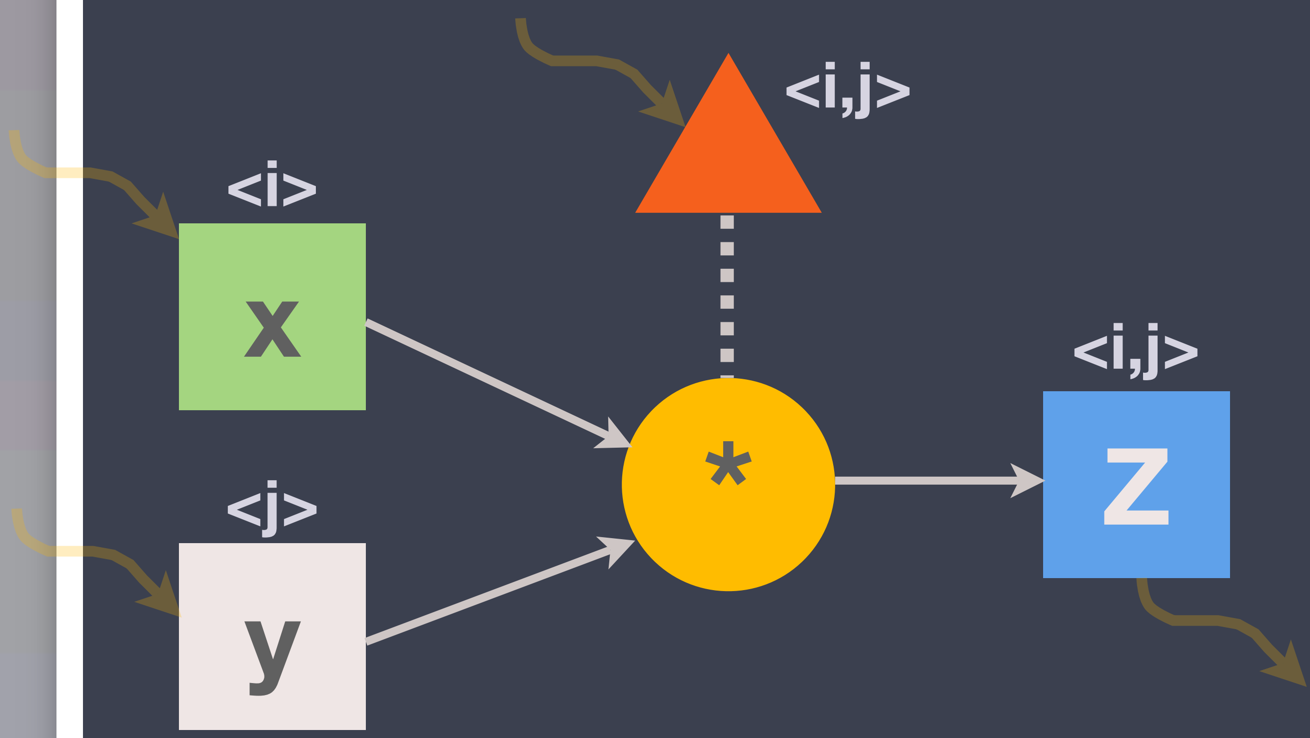
$$z_{i,j} \leftarrow x_i \cdot y_j$$




```
// Input:

   env → <*: i,j>, [x: i], [y: j];

// Prescription relations:

   <*: i,j> :: (*: i,j);

// Producer/consumer relations:

   [x: i], [y: j] → (*: i, j);

   (*: i, j) → [Z: i, j];

// Output:

   [Z: i, j] → env;
```

Step code written in a sequential base language

$$z_{i,j} \leftarrow x_i \cdot y_j$$

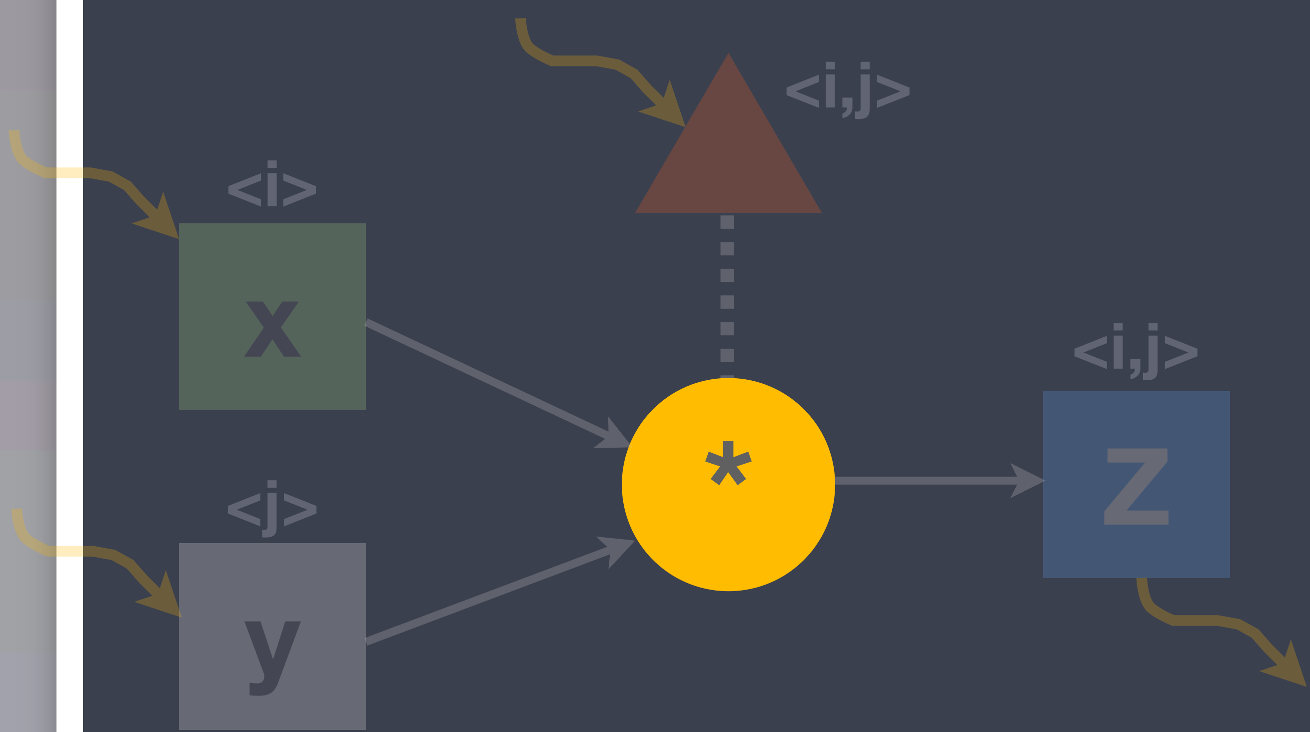



```
Return_t mult (Graph_t& G,
               const Tag_t& t)
{
  int i = t[0], j = t[1];
  double x_i = G.x.Get (Tag_t(i));
  double y_j = G.y.Get (Tag_t(j));
  G.Z.Put (Tag_t(i, j), x_i*y_j);
  return CNC_Success;
}
```

Intel's implementation uses C++; Rice University's uses Java (Habanero)

Step code written in a sequential base language

$$z_{i,j} \leftarrow x_i \cdot y_j$$



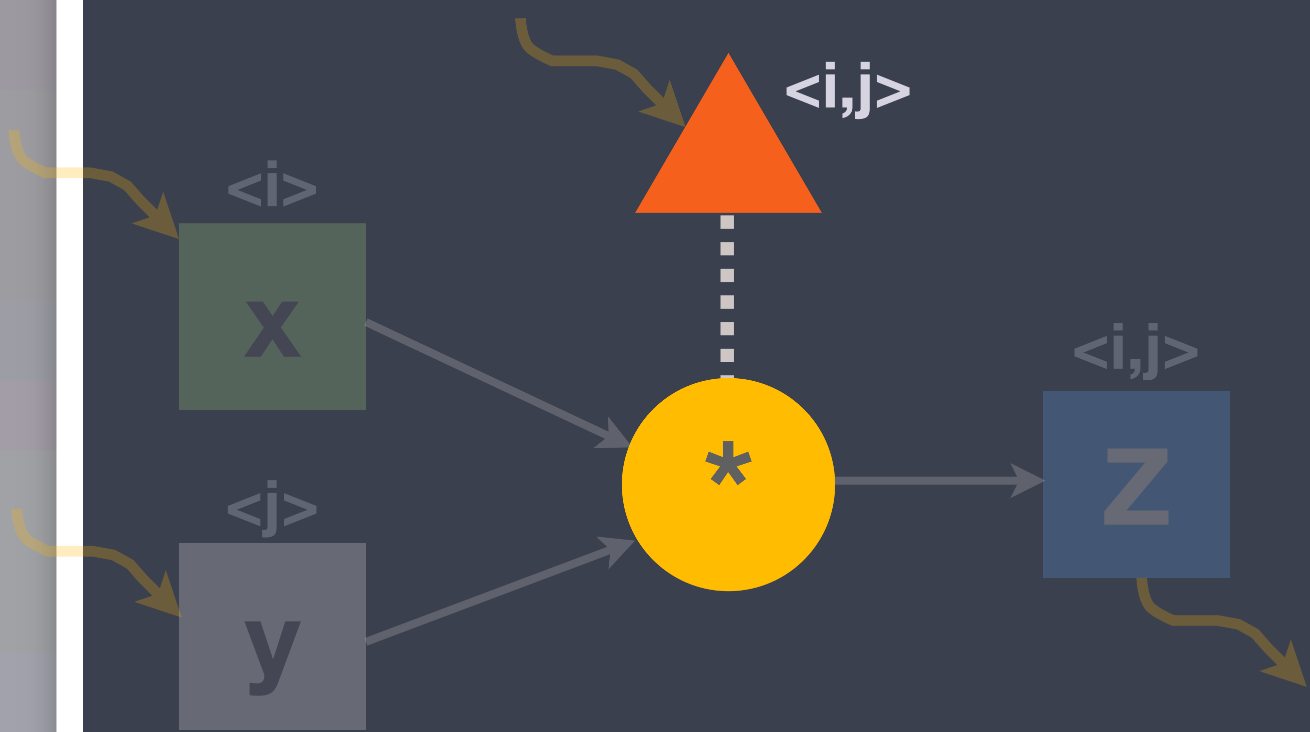

```
Return_t mult (Graph_t& G,
               const Tag_t& t)
{
  int i = t[0], j = t[1];
  double x_i = G.x.Get (Tag_t(i));
  double y_j = G.y.Get (Tag_t(j));
  G.Z.Put (Tag_t(i, j), x_i*y_j);
  return CNC_Success;
}
```

Intel's implementation uses C++; Rice University's uses Java (Habanero)

35

Step code written in a sequential base language

$$z_{i,j} \leftarrow x_i \cdot y_j$$

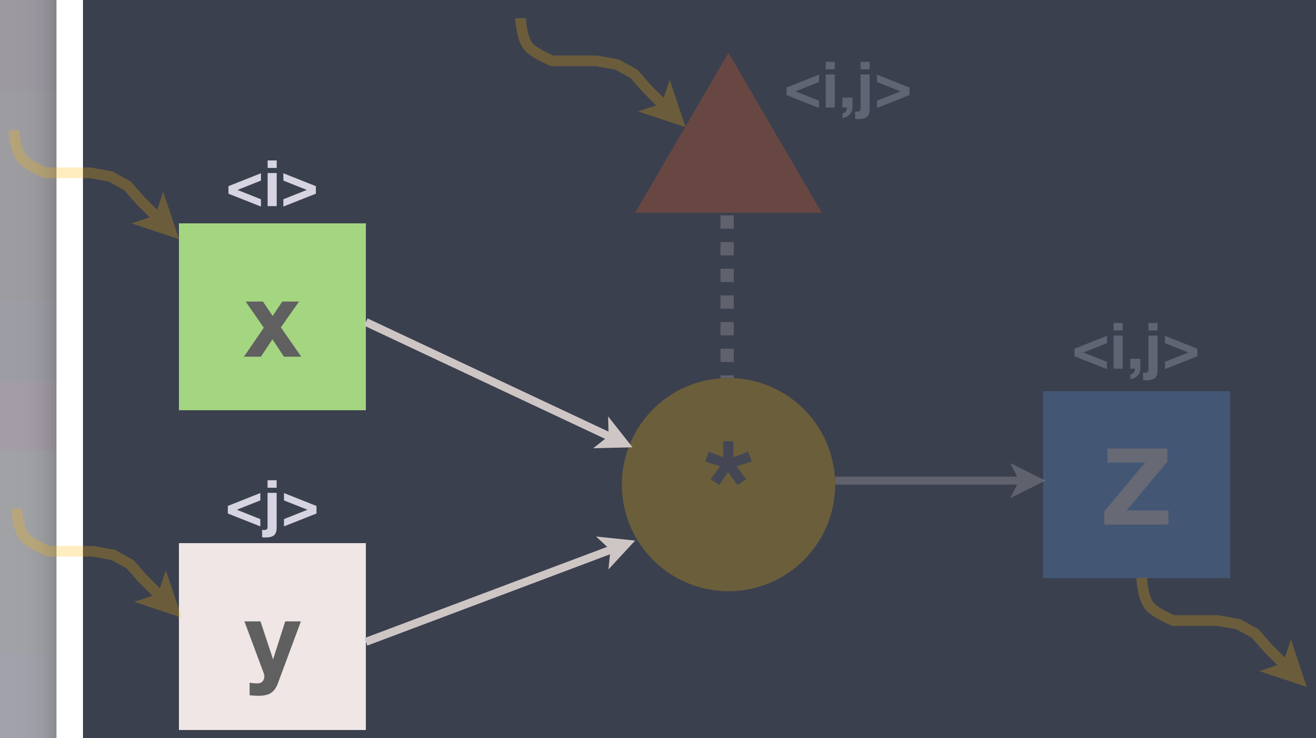



```
Return_t mult (Graph_t& G,
               const Tag_t& t)
{
  int i = t[0], j = t[1];
  double x_i = G.x.Get (Tag_t(i));
  double y_j = G.y.Get (Tag_t(j));
  G.Z.Put (Tag_t(i, j), x_i*y_j);
  return CNC_Success;
}
```

Intel's implementation uses C++; Rice University's uses Java (Habanero)

Step code written in a sequential base language

$$z_{i,j} \leftarrow x_i \cdot y_j$$

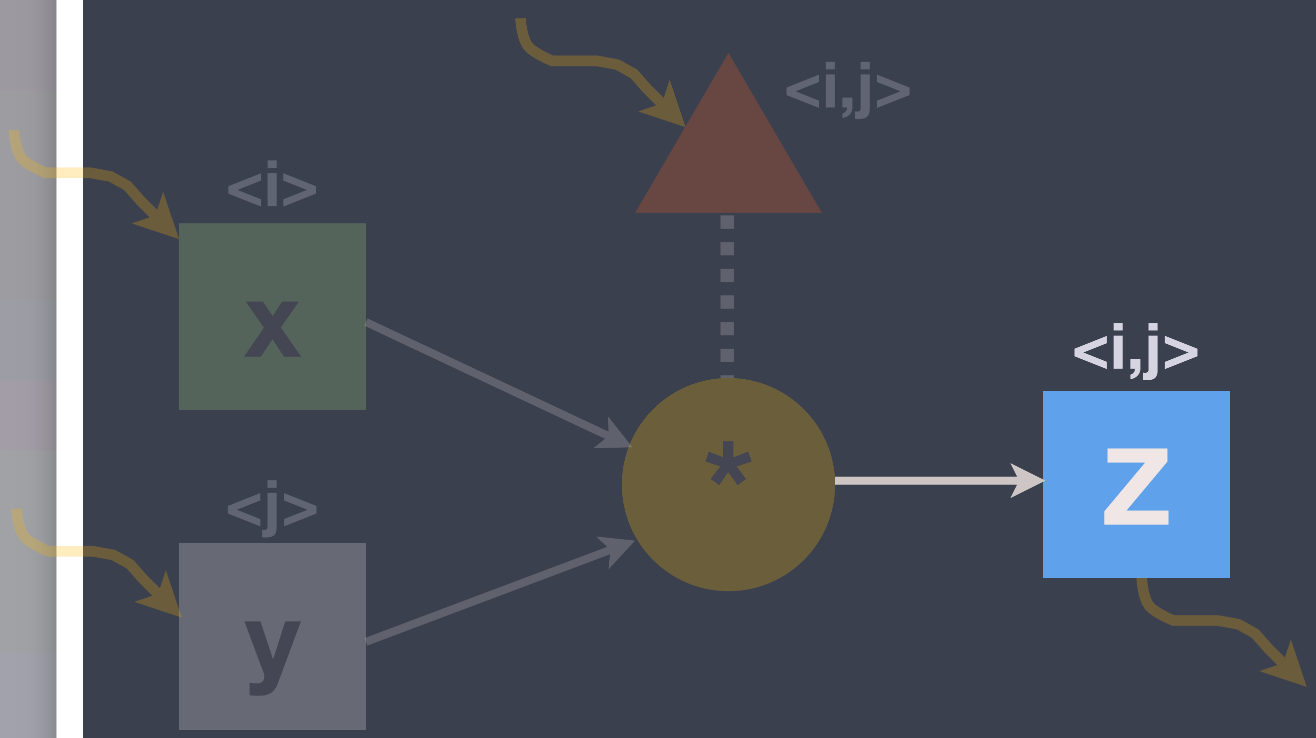



```
Return_t mult (Graph_t& G,
               const Tag_t& t)
{
  int i = t[0], j = t[1];
  double x_i = G.x.Get (Tag_t(i));
  double y_j = G.y.Get (Tag_t(j));
  G.Z.Put (Tag_t(i, j), x_i*y_j);
  return CNC_Success;
}
```

Intel's implementation uses C++; Rice University's uses Java (Habanero)

Step code written in a sequential base language

$$z_{i,j} \leftarrow x_i \cdot y_j$$

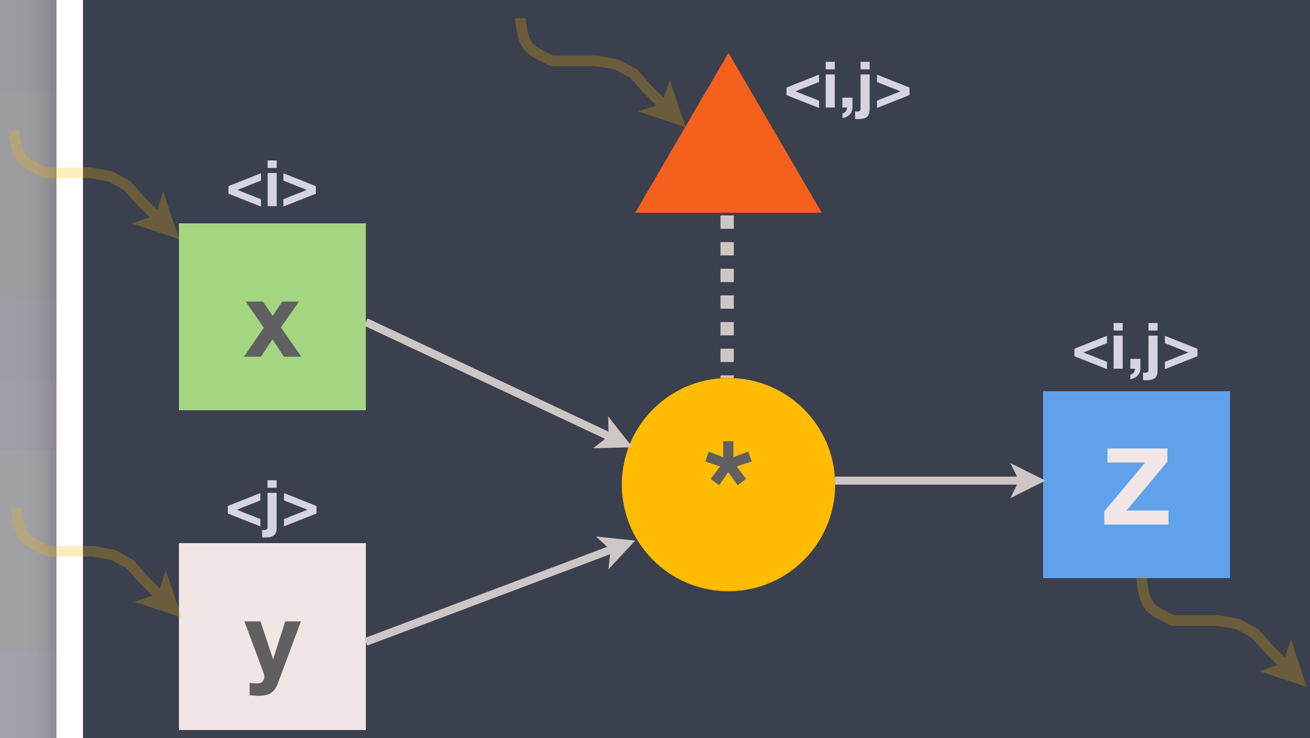



```
Return_t mult (Graph_t& G,
               const Tag_t& t)
{
  int i = t[0], j = t[1];
  double x_i = G.x.Get (Tag_t(i));
  double y_j = G.y.Get (Tag_t(j));
  G.Z.Put (Tag_t(i, j), x_i*y_j);
  return CNC_Success;
}
```

Intel's implementation uses C++; Rice University's uses Java (Habanero)

Step code written in a sequential base language

$$z_{i,j} \leftarrow x_i \cdot y_j$$




```
Return_t mult (Graph_t& G,
               const Tag_t& t)
{
  int i = t[0], j = t[1];
  double x_i = G.x.Get (Tag_t(i));
  double y_j = G.y.Get (Tag_t(j));
  G.Z.Put (Tag_t(i, j), x_i*y_j);
  return CNC_Success;
}
```

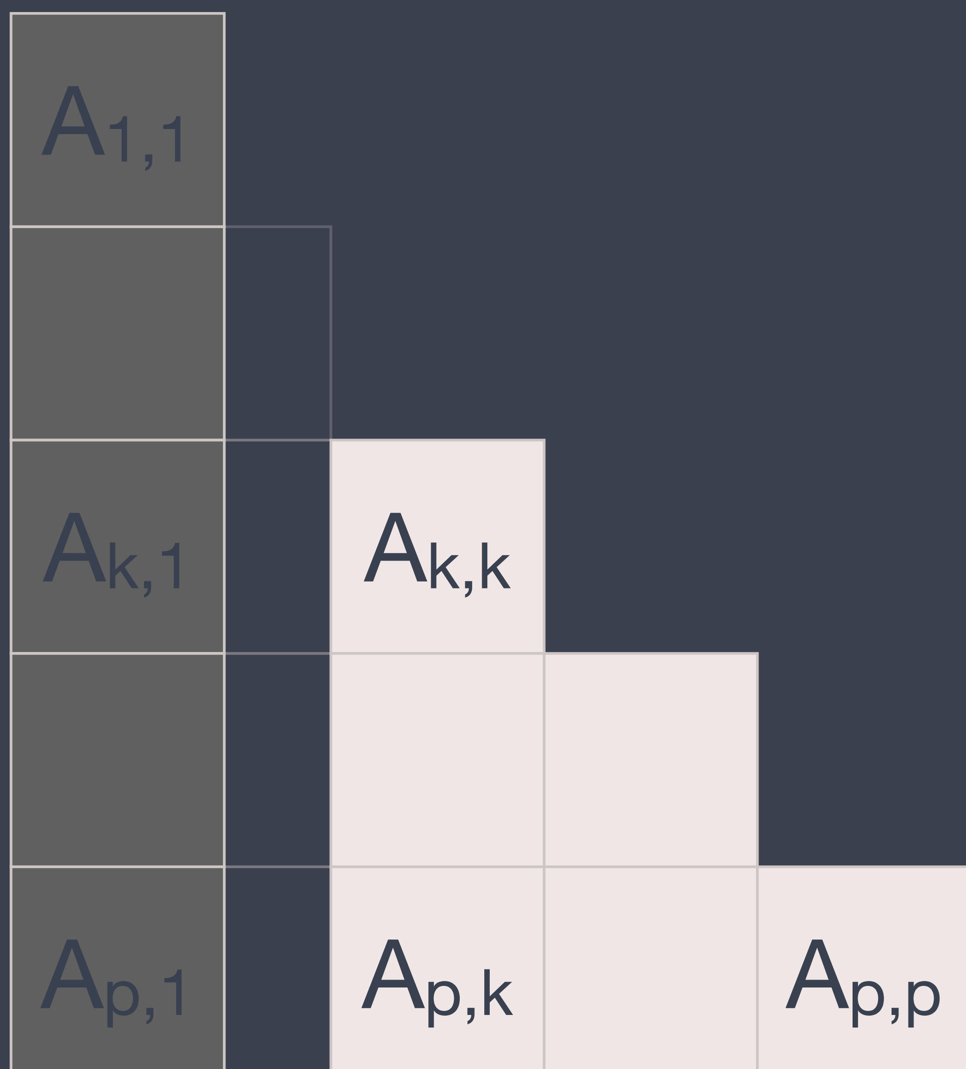
Intel's implementation uses C++; Rice University's uses Java (Habanero)

Run-time system

- Built on top of Intel Threading Building Blocks (TBB)
  - Implements Cilk-style work stealing scheduler
  - Work queues use LIFO, but FIFO and other strategies in development
- Other run-times possible
  - DEC/HP TStreams on MPI; Rice U. Habanero uses Java threads
  - Intel-specific issues with queuing (more later)

Tile Cholesky: $A \rightarrow L \cdot L^T$

Buttari, *et al*. (2007)



Iteration *k*: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

41

Tile Cholesky: $A \rightarrow L \cdot L^T$

Buttari, *et al*. (2007)

Iteration *k*: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

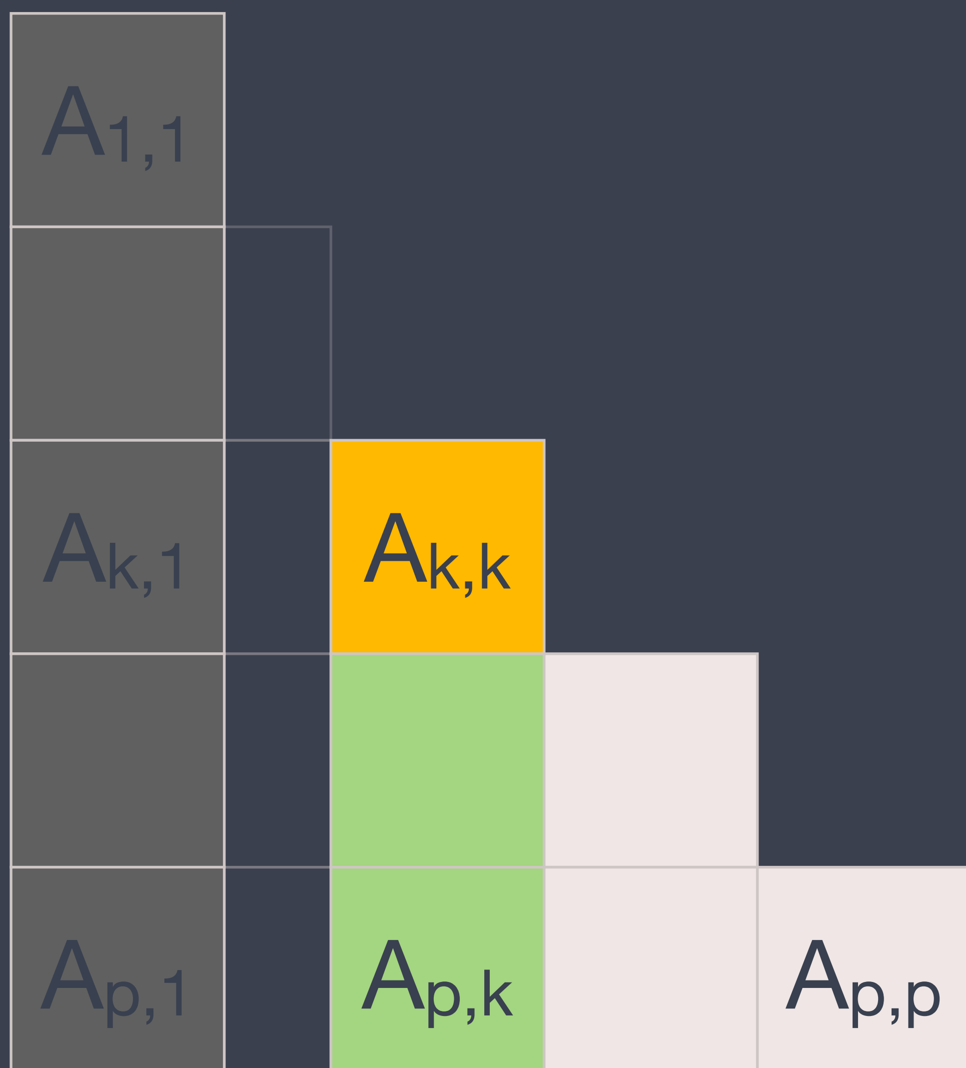
Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

42

Tile Cholesky: $A \rightarrow L \cdot L^T$

Buttari, *et al*. (2007)



Iteration *k*: // Over diagonal tiles

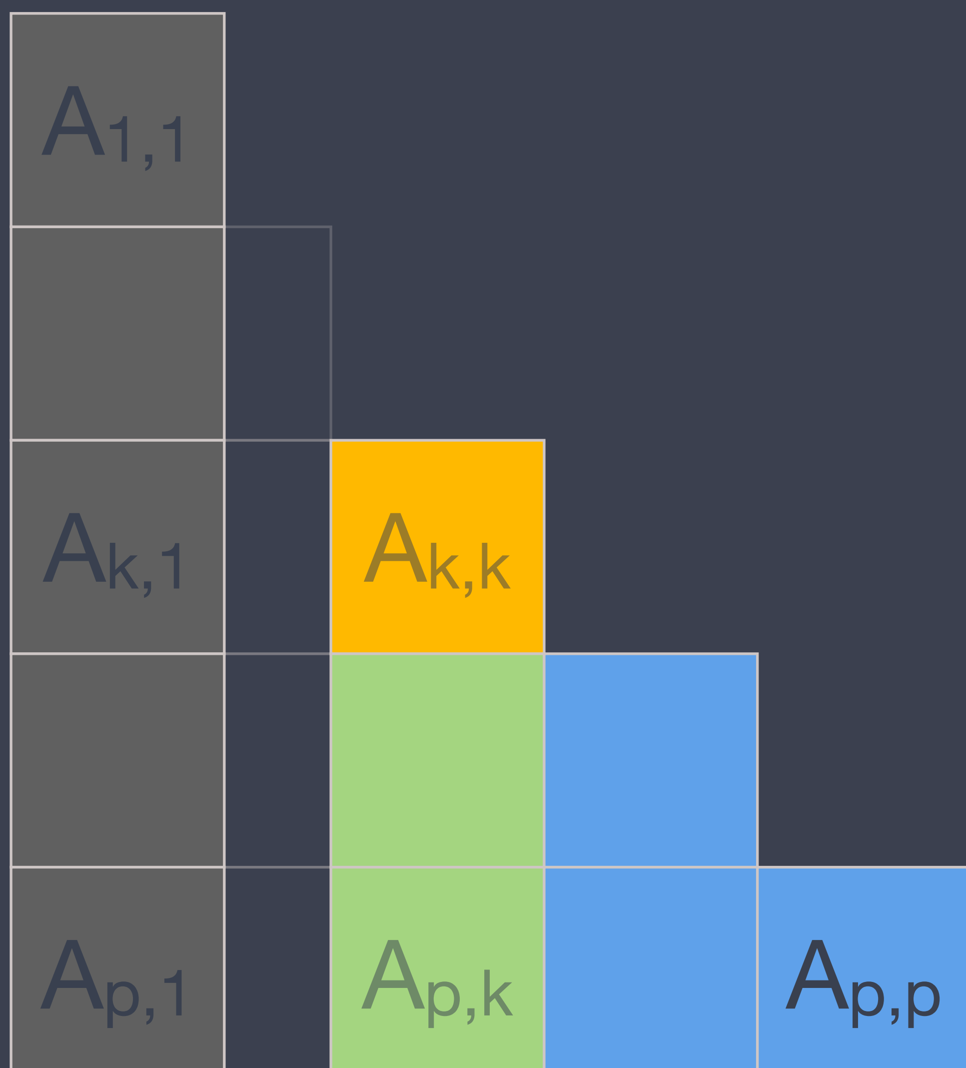
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

Tile Cholesky: $A \rightarrow L \cdot L^T$

Buttari, *et al*. (2007)



Iteration *k*: // Over diagonal tiles

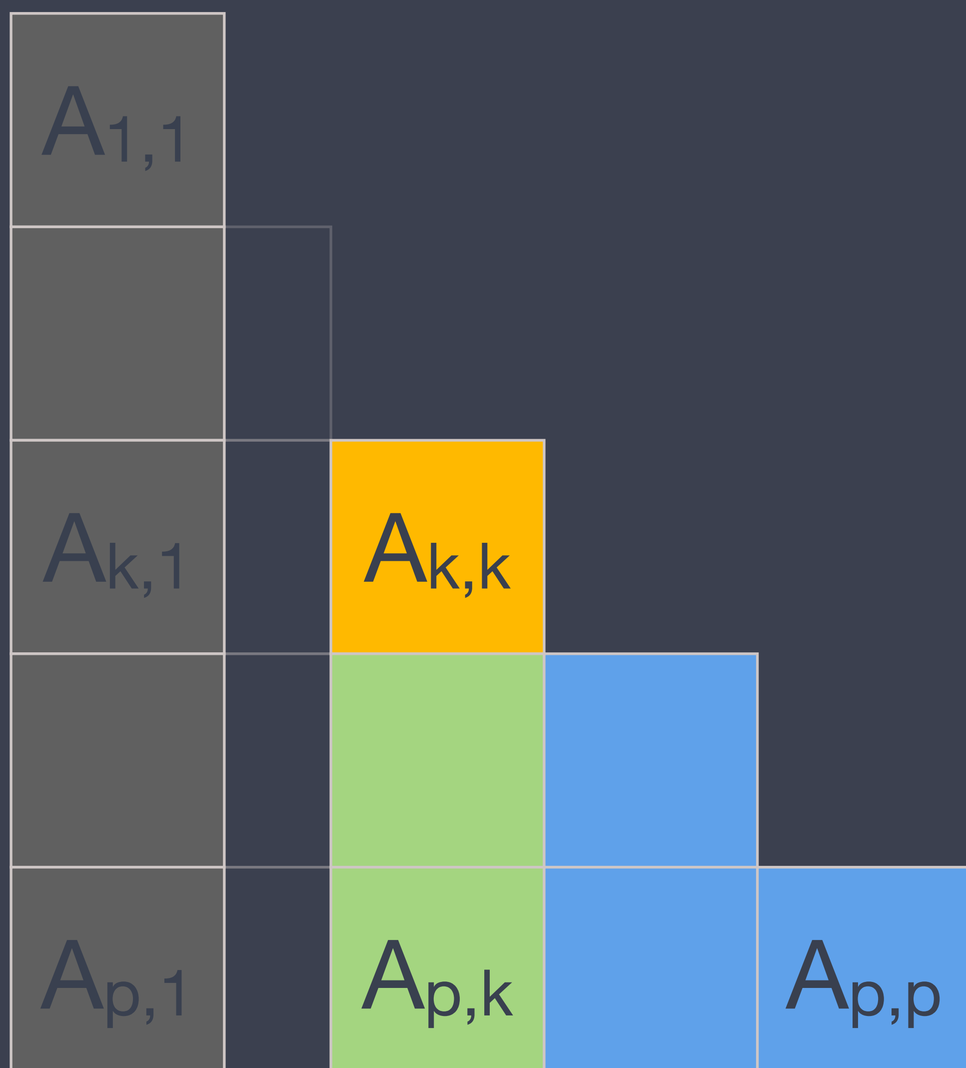
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

Tile Cholesky: $A \rightarrow L \cdot L^T$

Buttari, *et al*. (2007)



Iteration k: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

45

Tile Cholesky in CnC

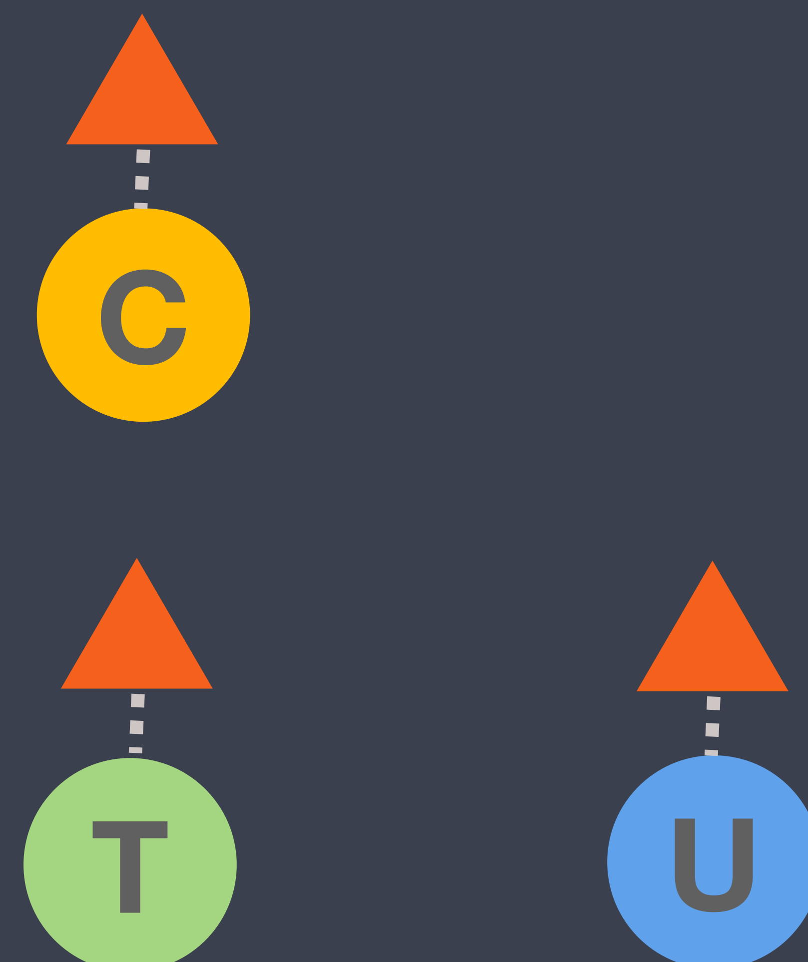
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

46

Tile Cholesky in CnC



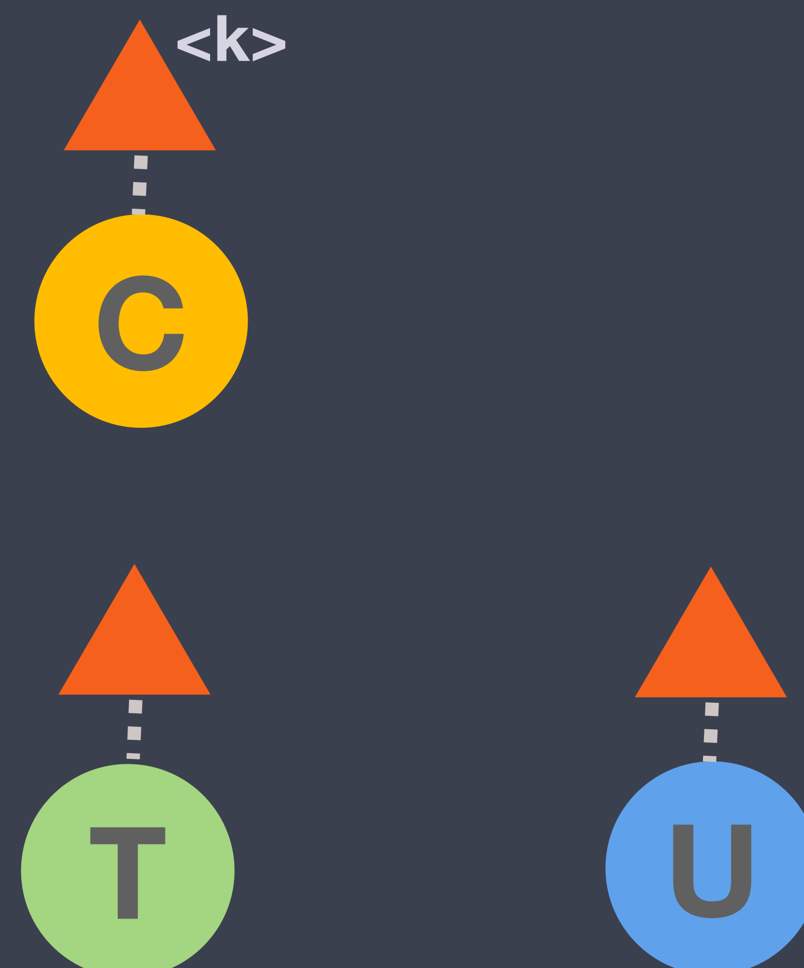
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

Omitted: Items

Tile Cholesky in CnC



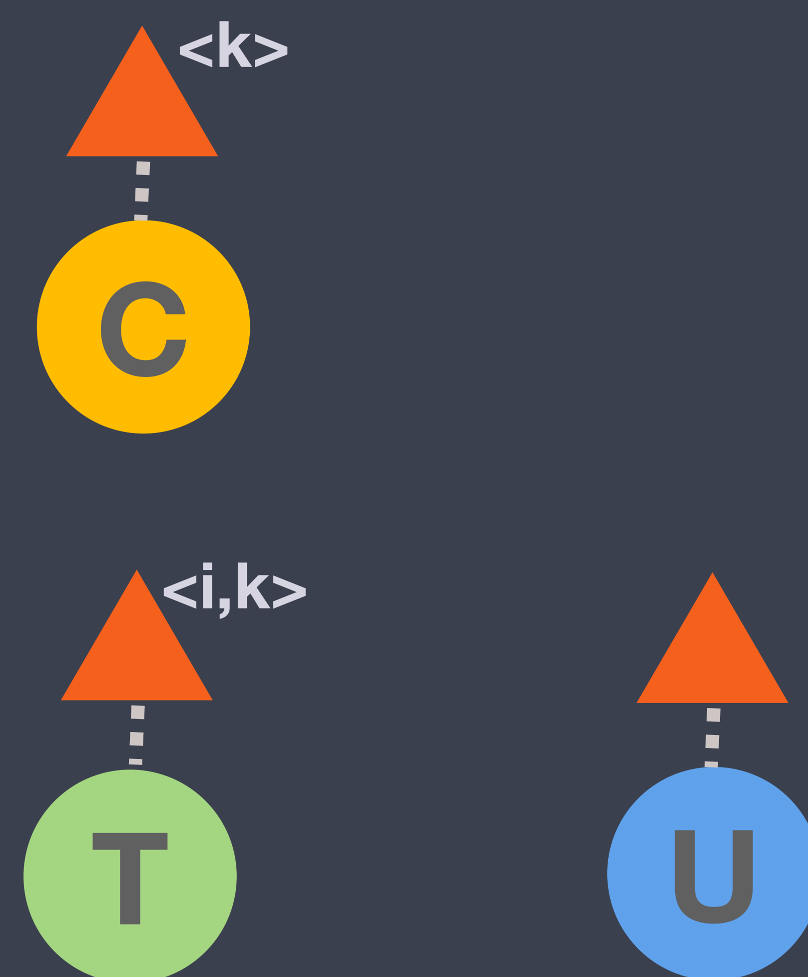
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

Iteration index is a natural tag

Tile Cholesky in CnC



SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

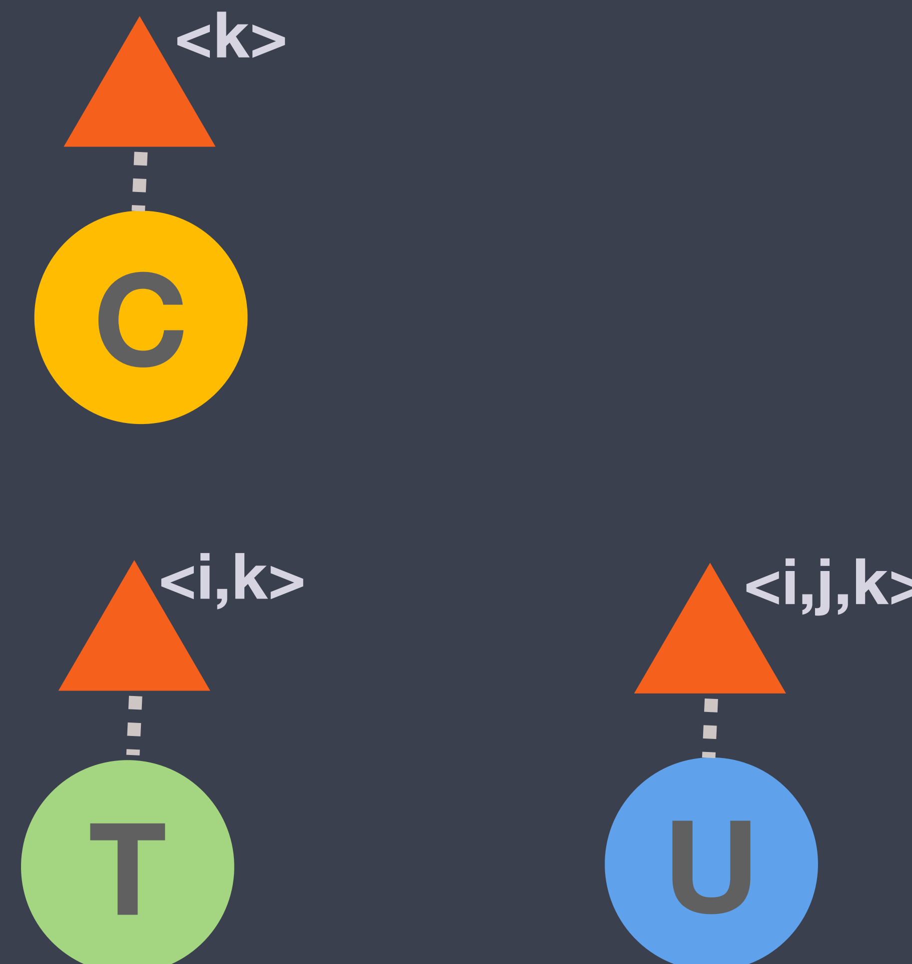
Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

Given k, multiple T steps could go ⇒ 2-D tag

49

Tile Cholesky in CnC



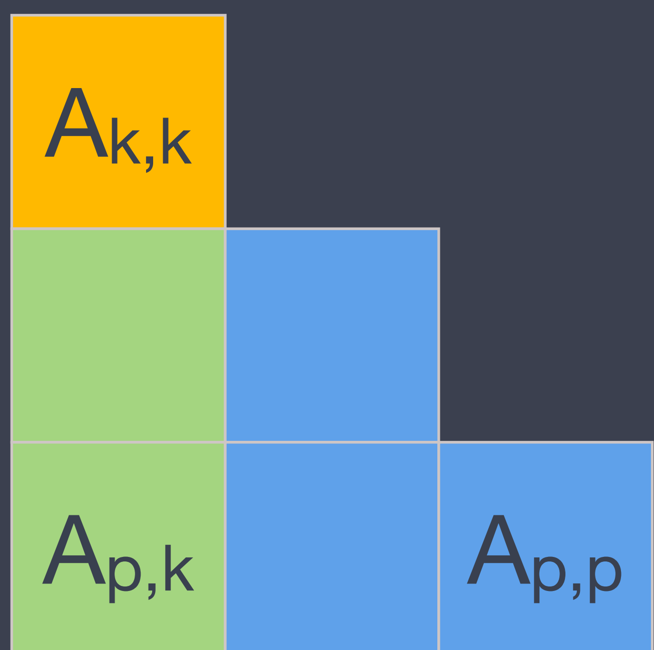
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

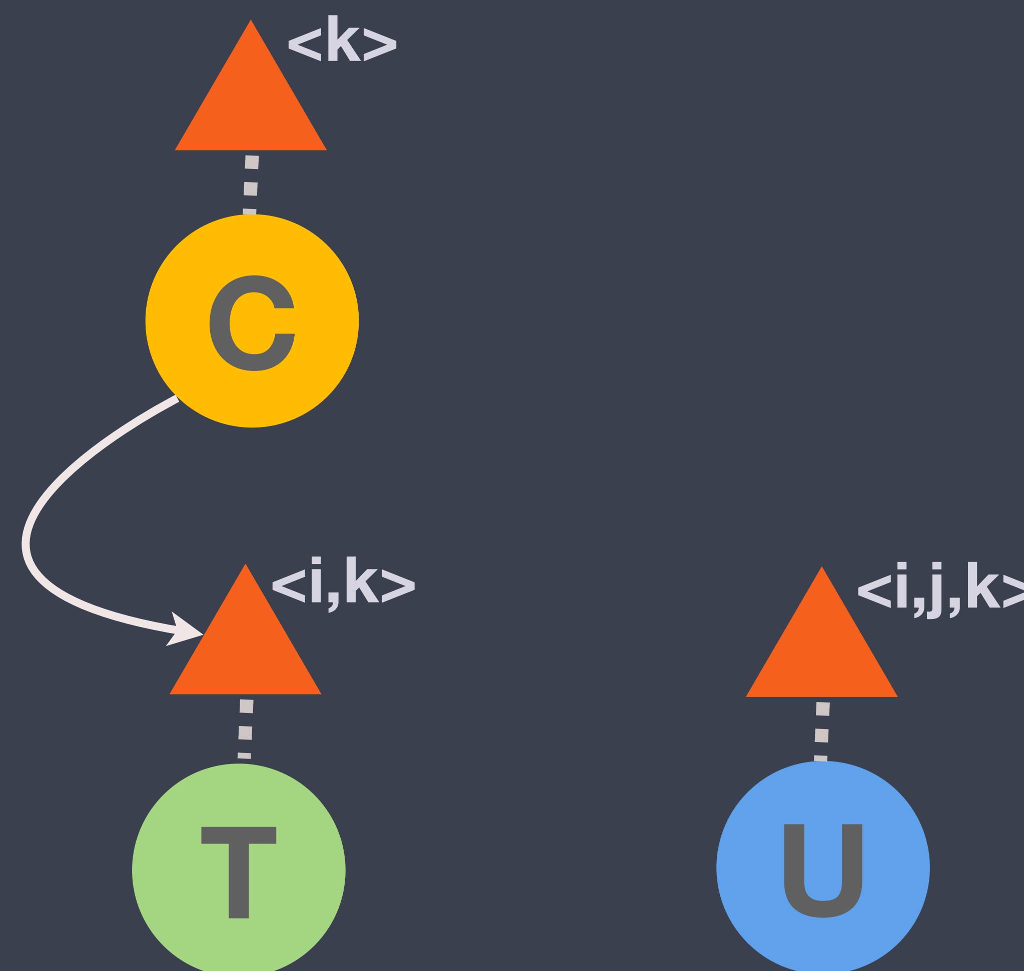
Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

Given *k*, 2-D iteration space of update steps could go

Tile Cholesky in CnC





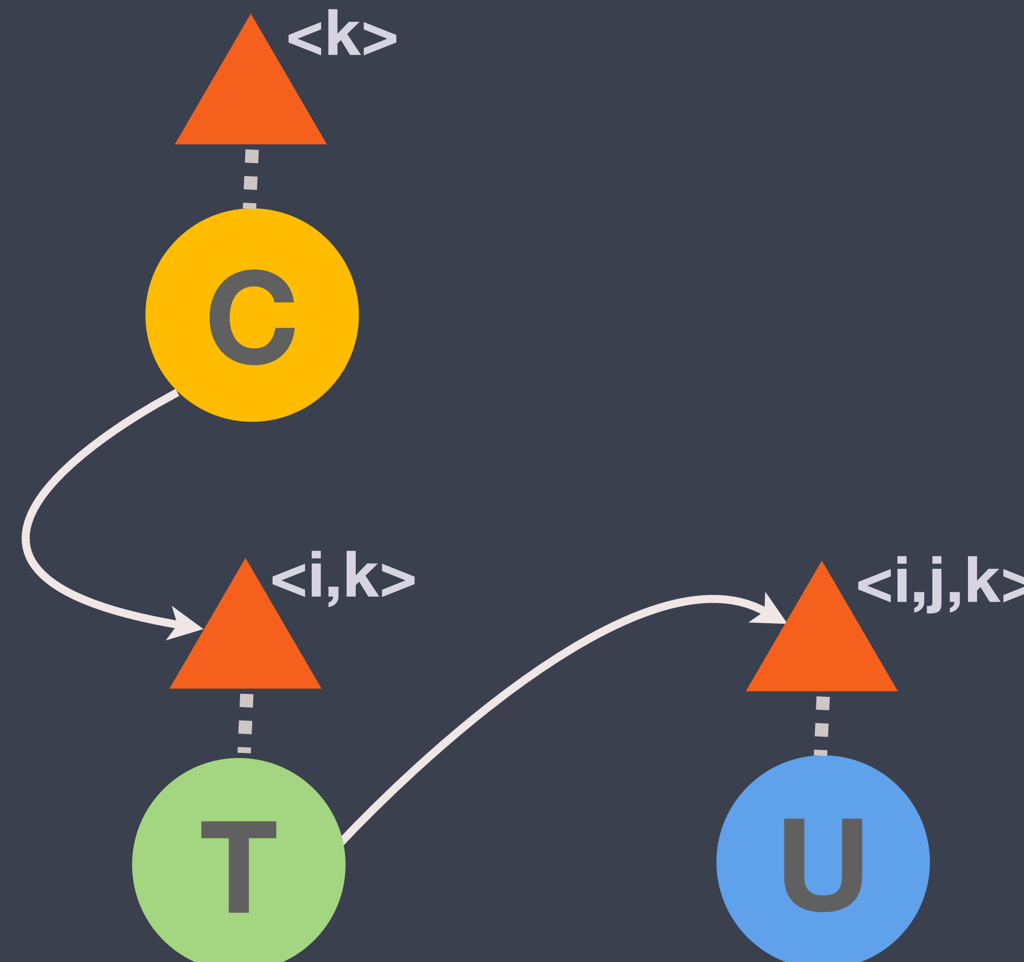
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

Sequential Cholesky step enables Trisolve steps

Tile Cholesky in CnC



SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

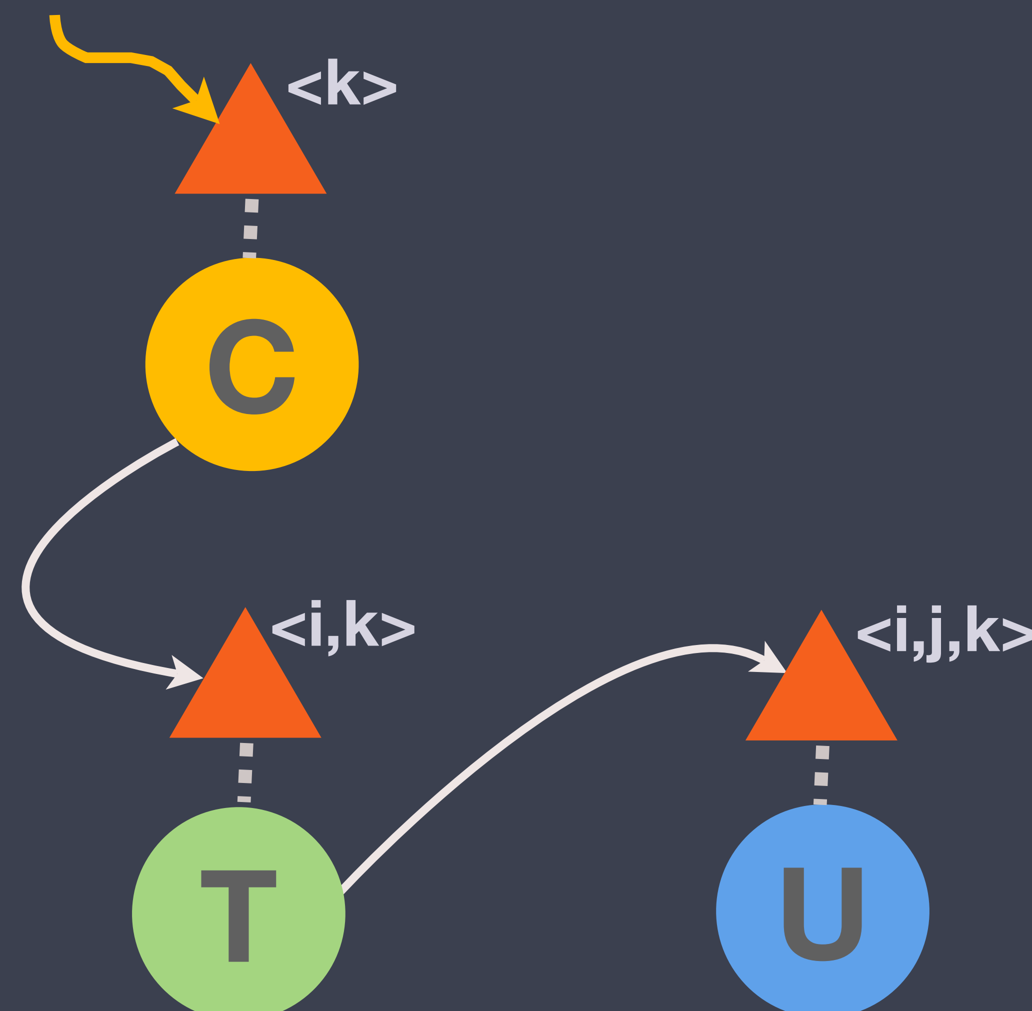
Similarly, Trisolve step enables Update steps

Tile Cholesky in CnC

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)



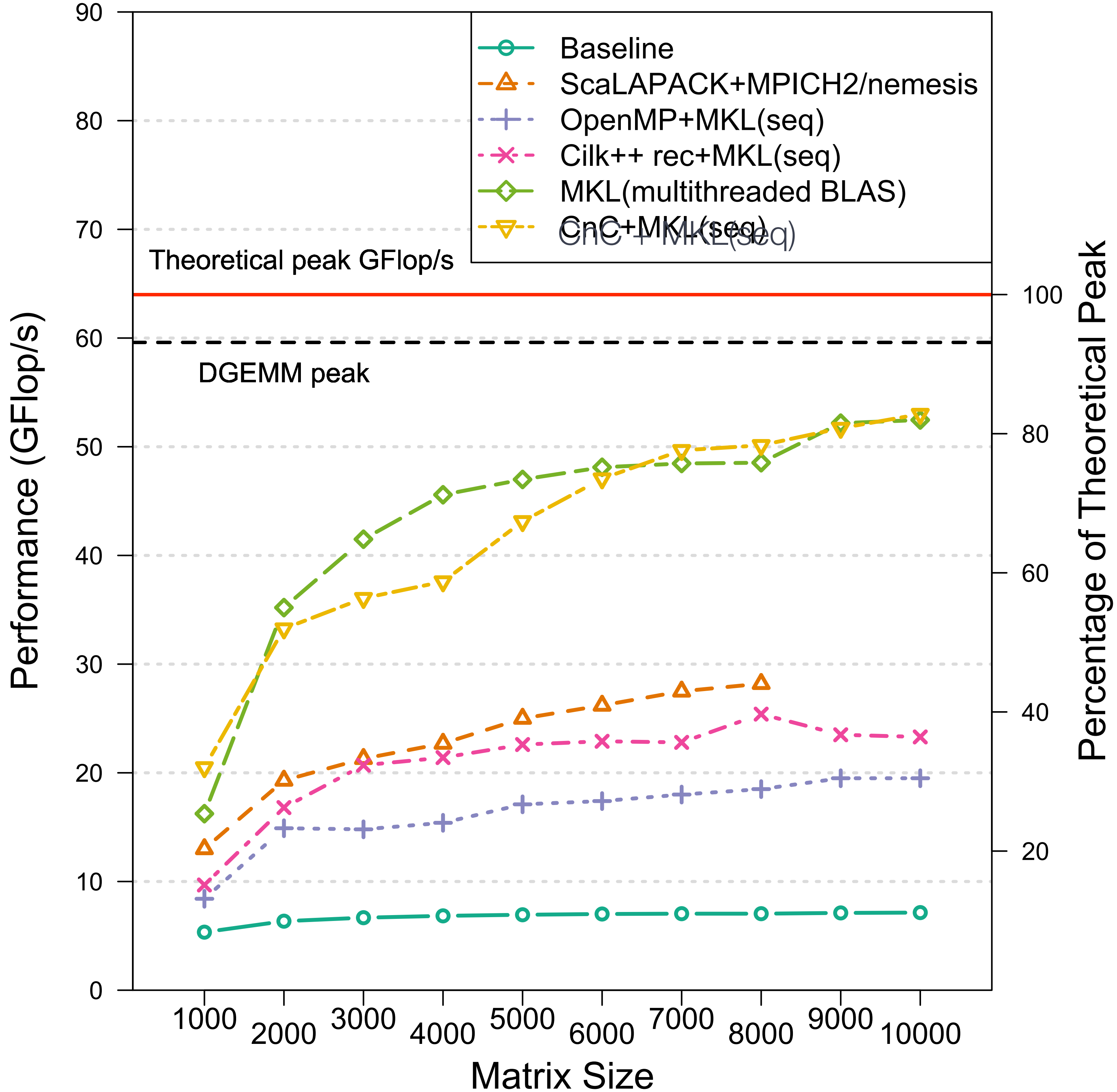
Other arrangements possible, *e.g.*, pre-generate all tags.

Dense symmetric generalized eigensolver

- "Straightforward" translation of LAPACK's `_sygvx` for $Az = \lambda Bz$
  - Pieces: Cholesky / reduction to standard form; tridiag reduction
  - Only partly "asynchronous," but useful proof-of-concept
  - Performance limited by tridiagonal reduction step (BLAS-2)

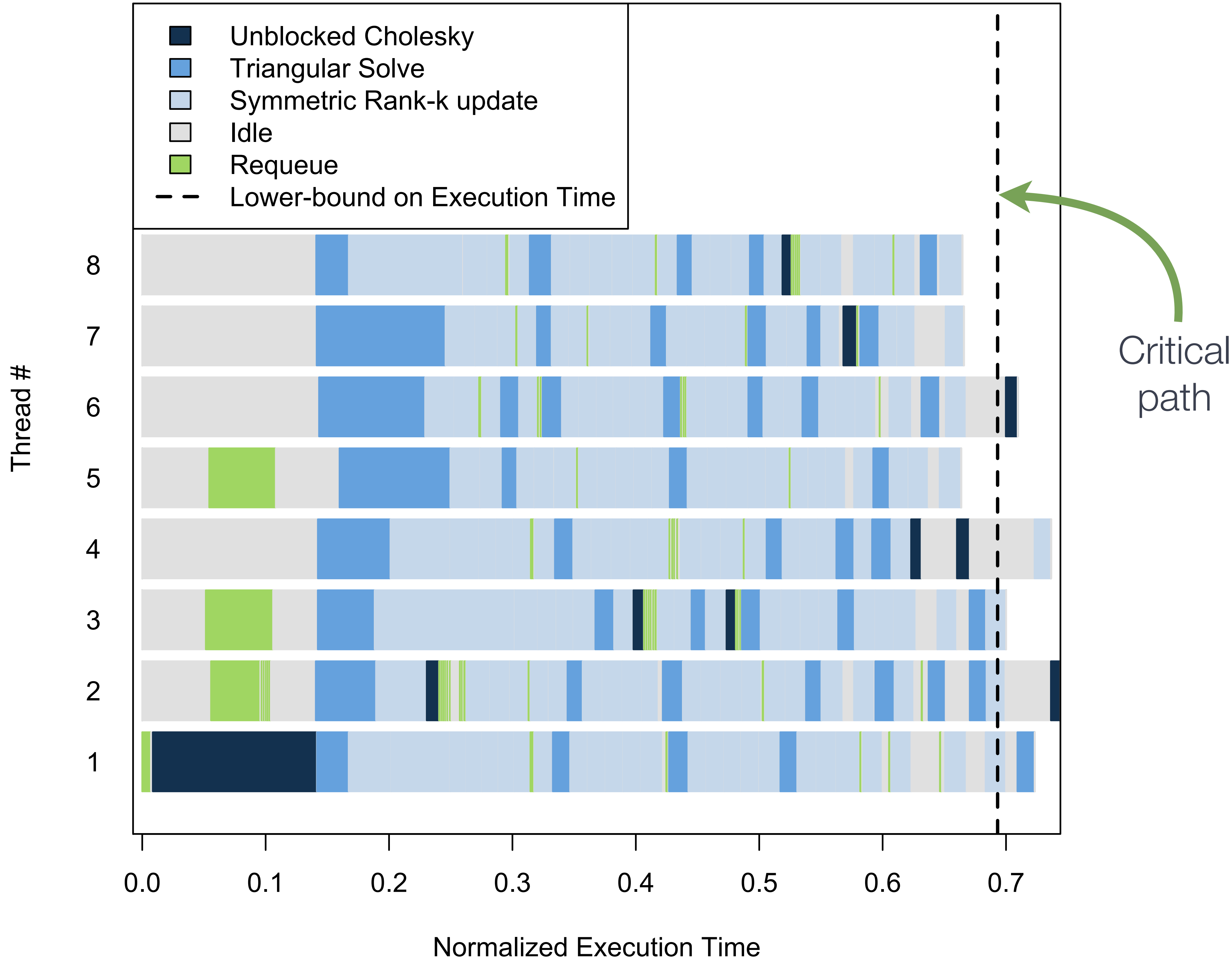
54

Experimental results



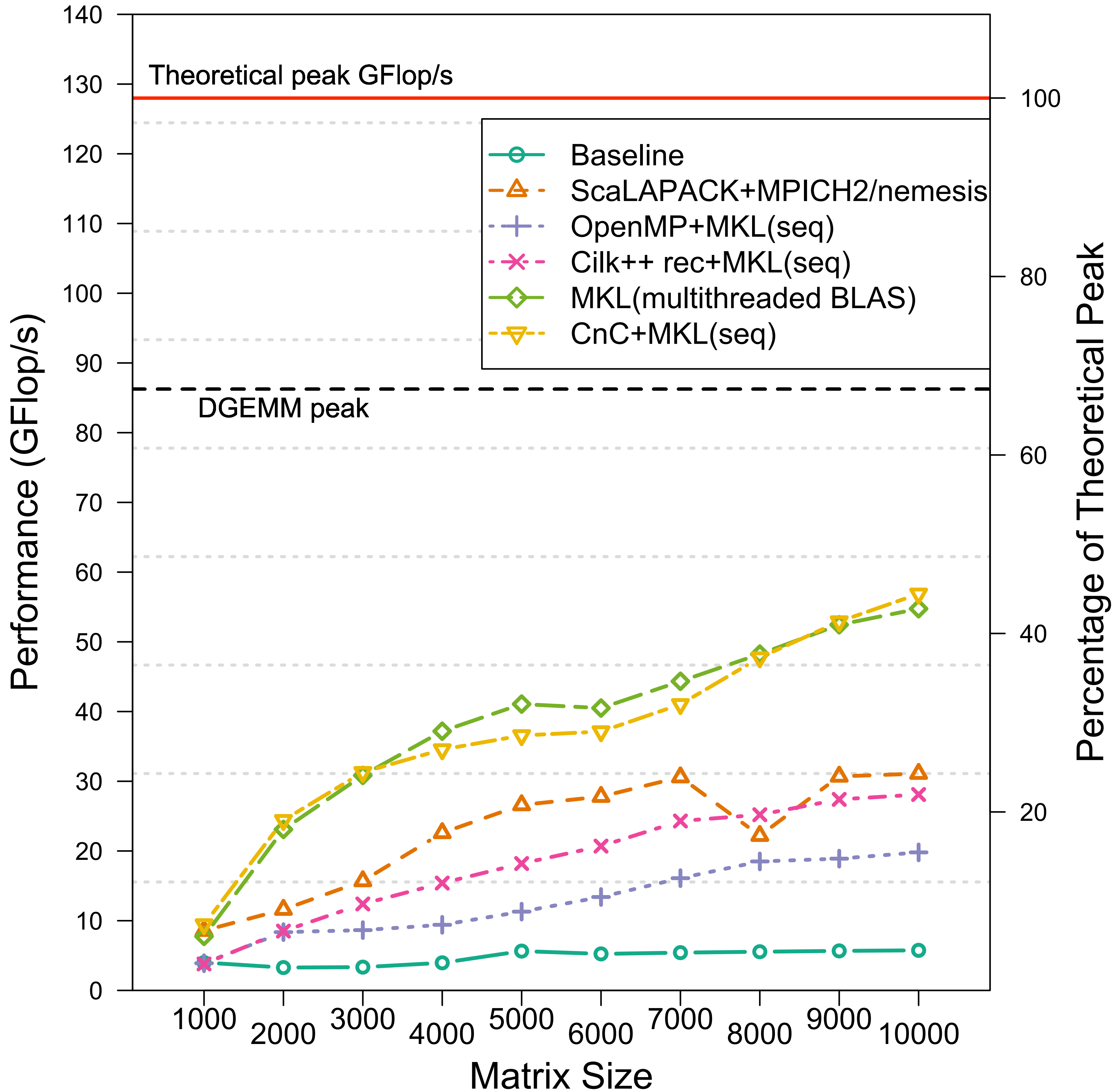
Cholesky performance:

Intel 2-socket x 4-core Harpertown @ 2 GHz + Intel MKL 10.1

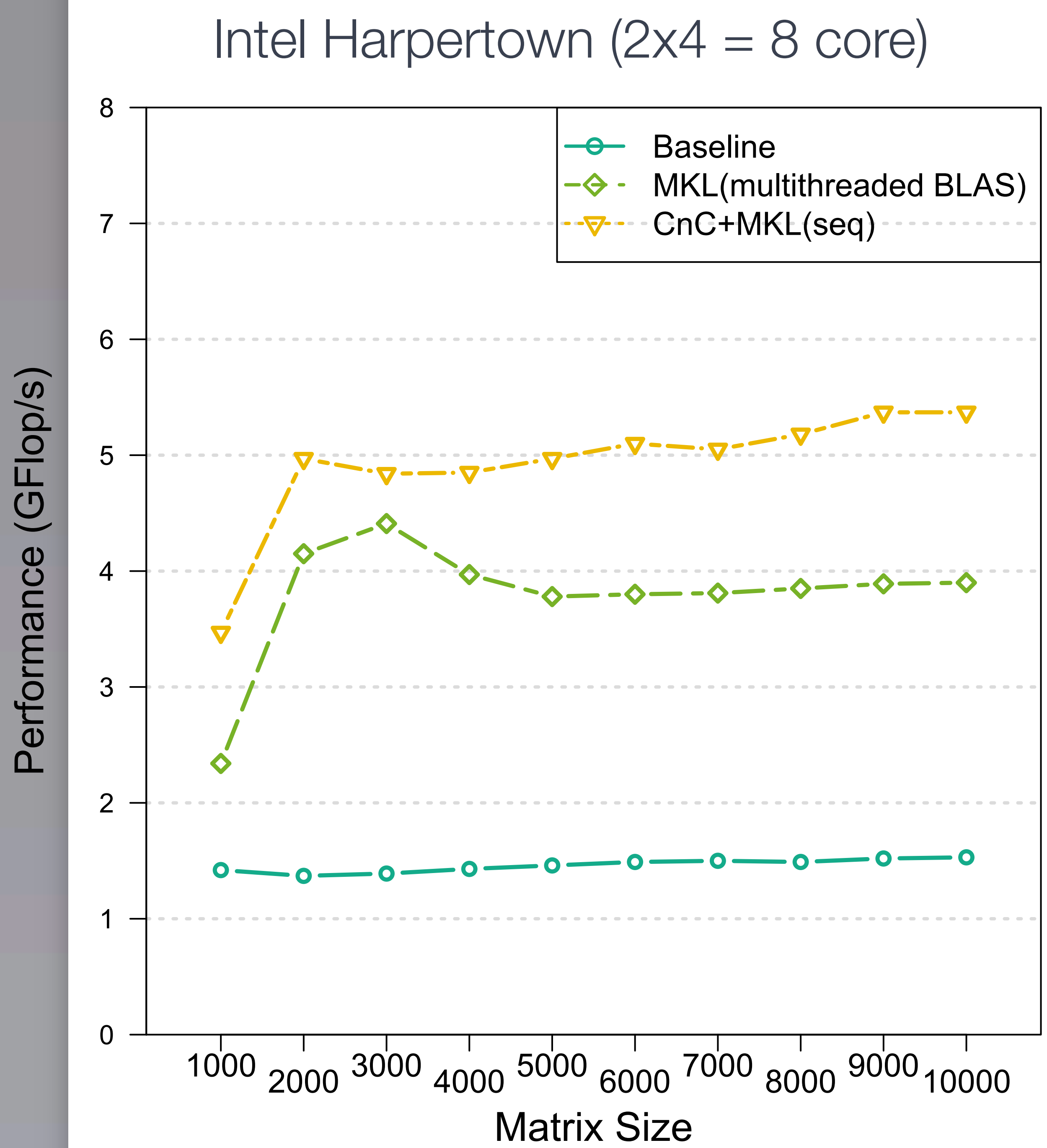


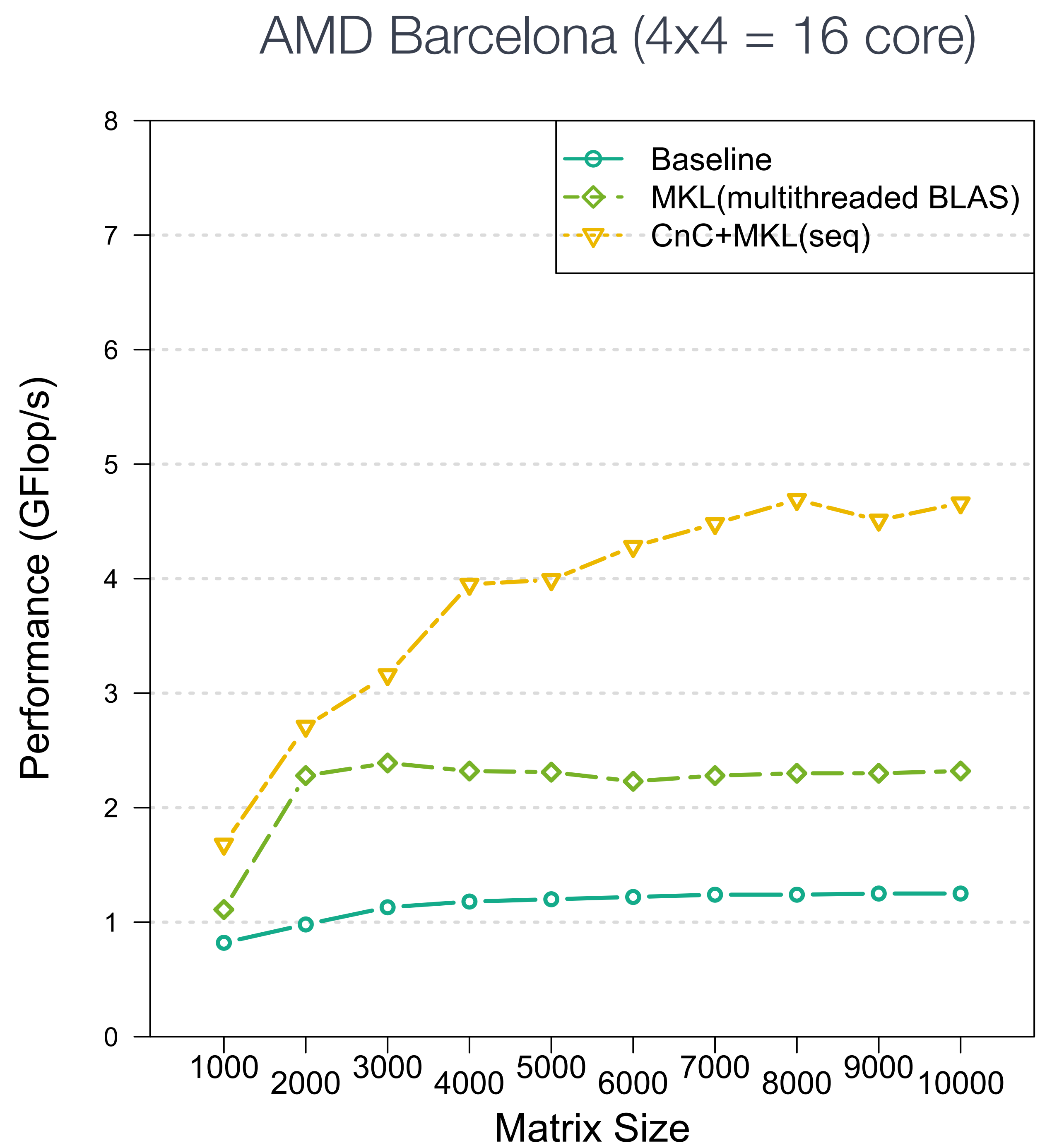
CnC-based Cholesky timeline (n=1000):

Intel 2-socket x 4-core Harpertown @ 2 GHz + Intel MKL 10.1 for sequential components



Cholesky performance: AMD 4-socket x 4-core Barcelona @ 2 GHz





Eigensolver performance (dsygvx)

Summary and future work

- CnC's key ideas
  - Decompose computation into steps + (data) items + (control) tags, with constraint relations among these components – dataflow-like
  - Goal: Separate computation semantics (orderings) from parallelism
- Ongoing
  - "Finish" proof-of-concept example by adding, *e.g.*, blocked data layouts
  - New language primitives to simplify tag management & improve modularity, performance
  - Extending run-time scheduling infrastructure
  - Other applications & architectures

Additional limitations

- Tag types: integers only
- Cannot handle continuous (streaming) input
- More natural support for in-place algorithms
- Tools, *e.g.*, debugging