# Concurrent Collections (CnC)

**Kath Knobe**
Kath.knobe@intel.com

2012/03/29

# Outline

- Intro to the CnC
- Tuning CnC
- CnC for distributed memory

# Outline

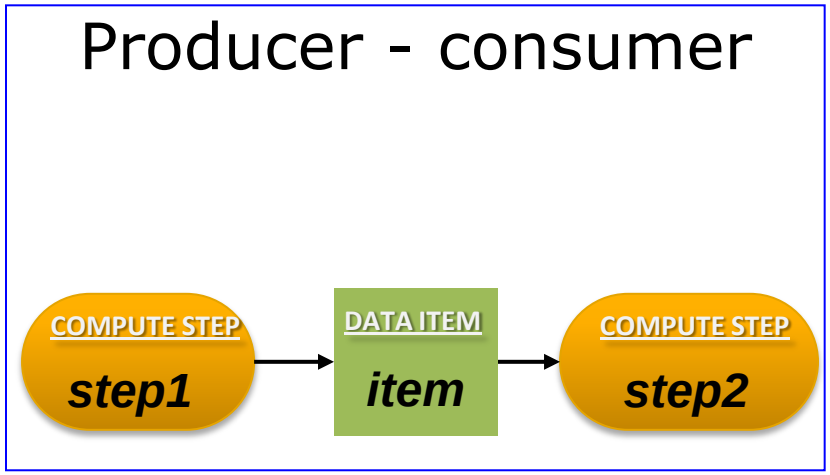
- Intro to the CnC
- Tuning CnC
- CnC for distributed memory

# The Big Idea

- Don’t think about what operations can run in parallel
  - *Difficult and depends on target*
- Think about the semantic ordering constraints only
  - *Easier and depends only on application*

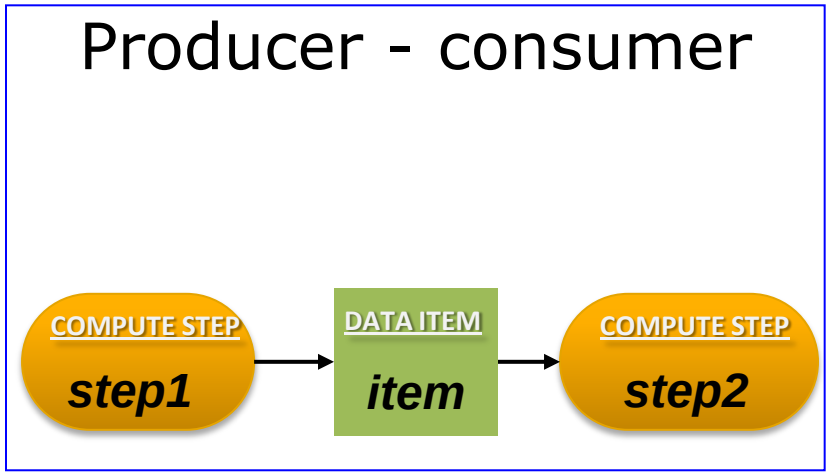
# Exactly Two Sources Of Ordering Requirements

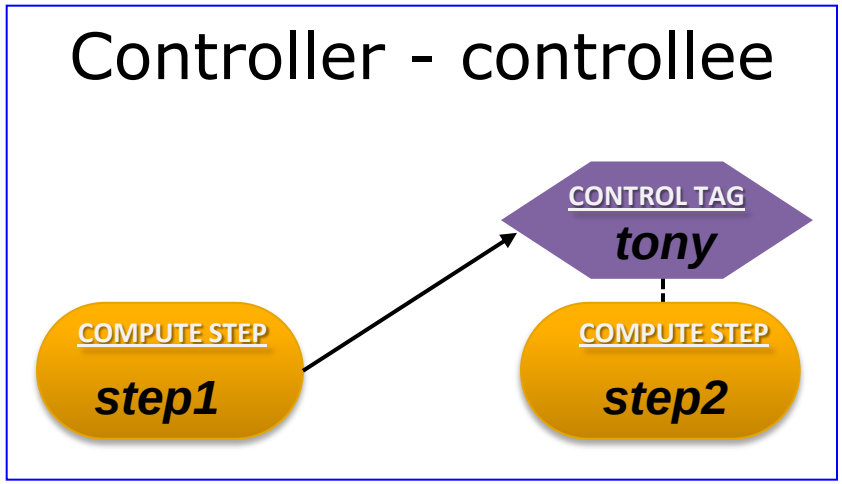
- **Producer must execute before consumer**



# Exactly Two Sources Of Ordering Requirements

- **Producer must execute before consumer**
- **Controller must execute before controllee**



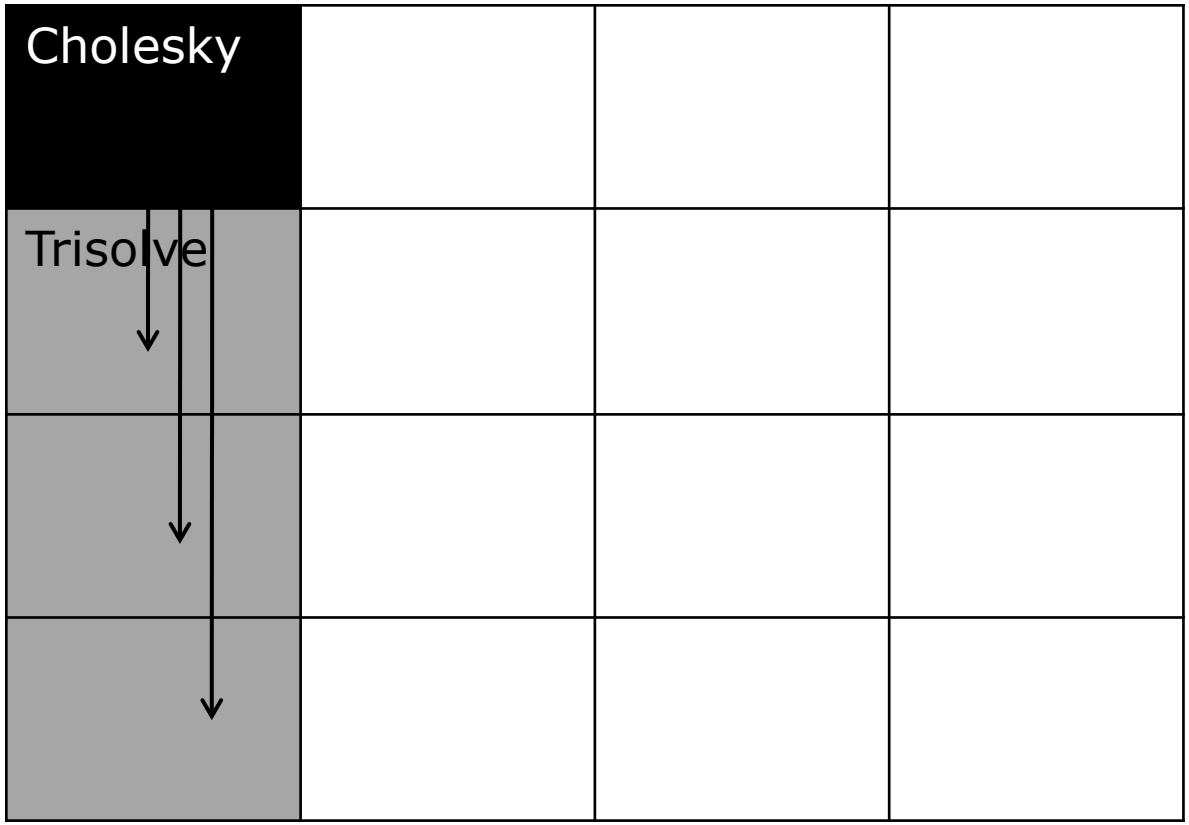
# Cholesky factorization

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

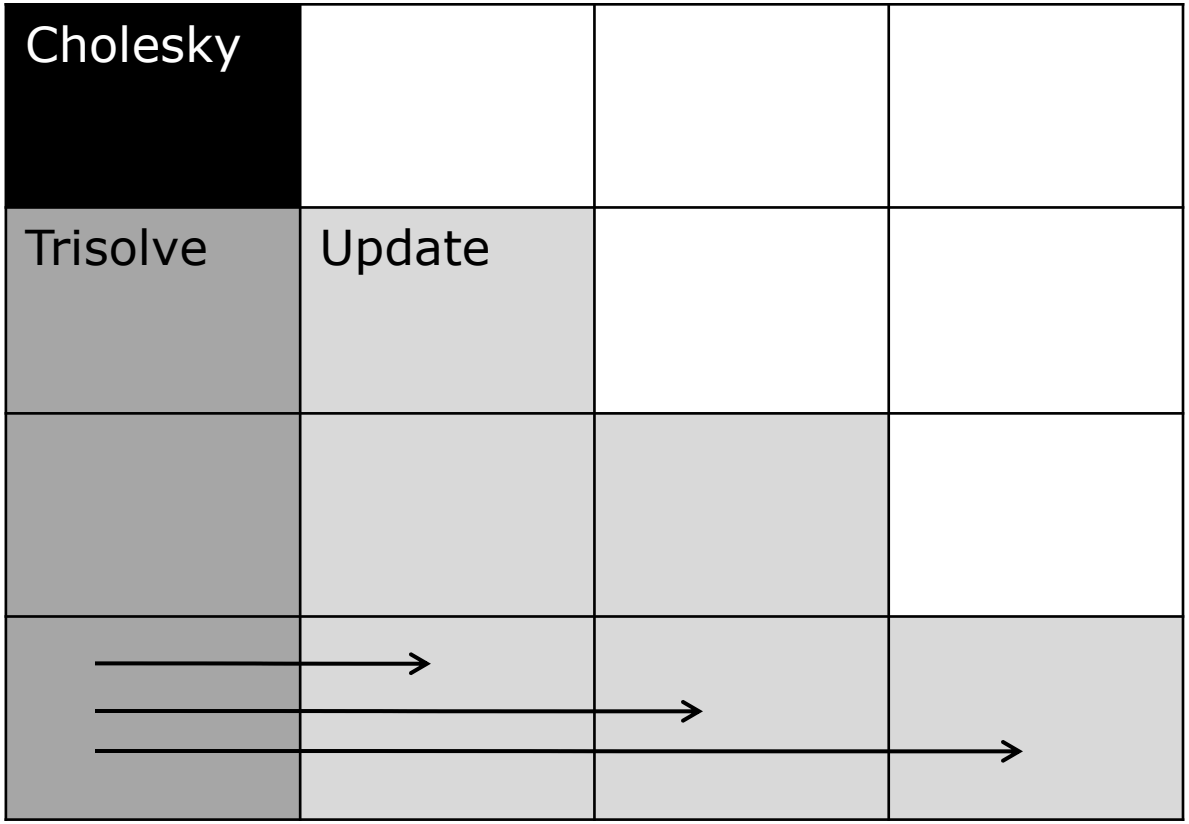
# Cholesky factorization

| Cholesky | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

# Cholesky factorization

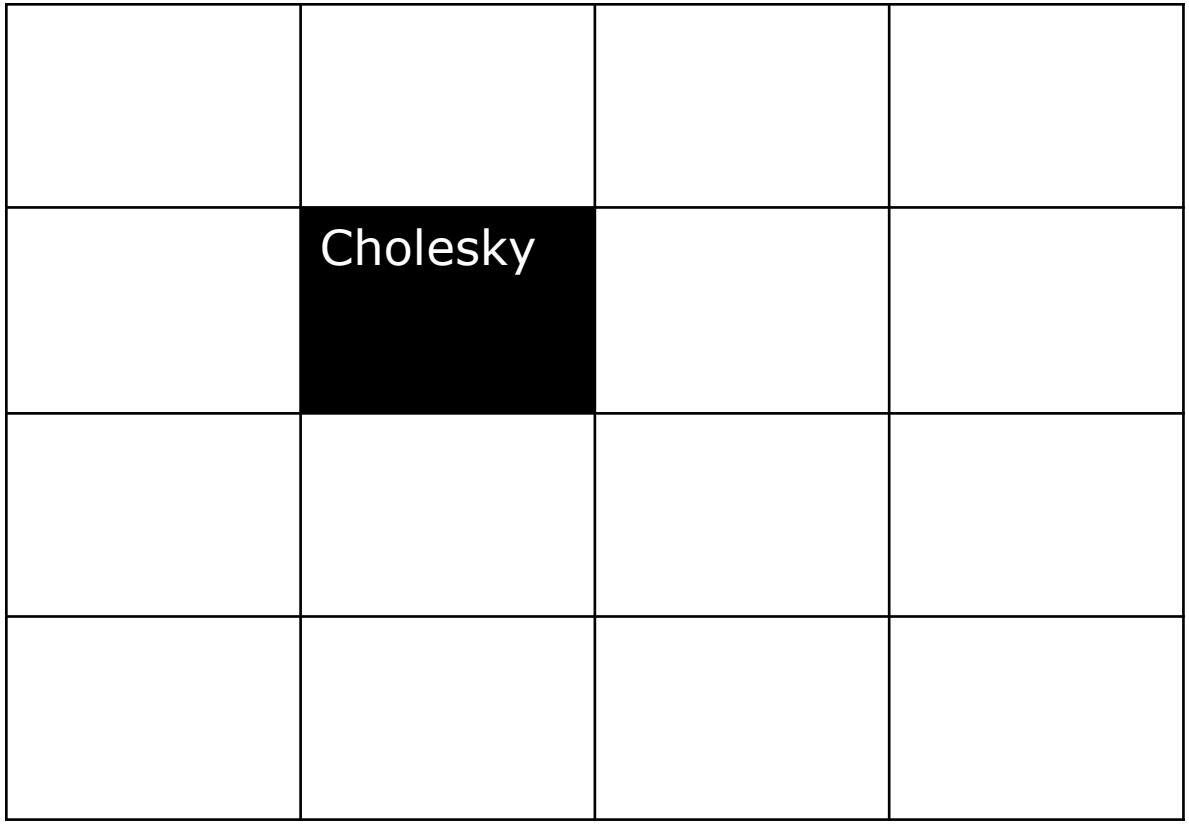


# Cholesky factorization




**Slightly simplified**

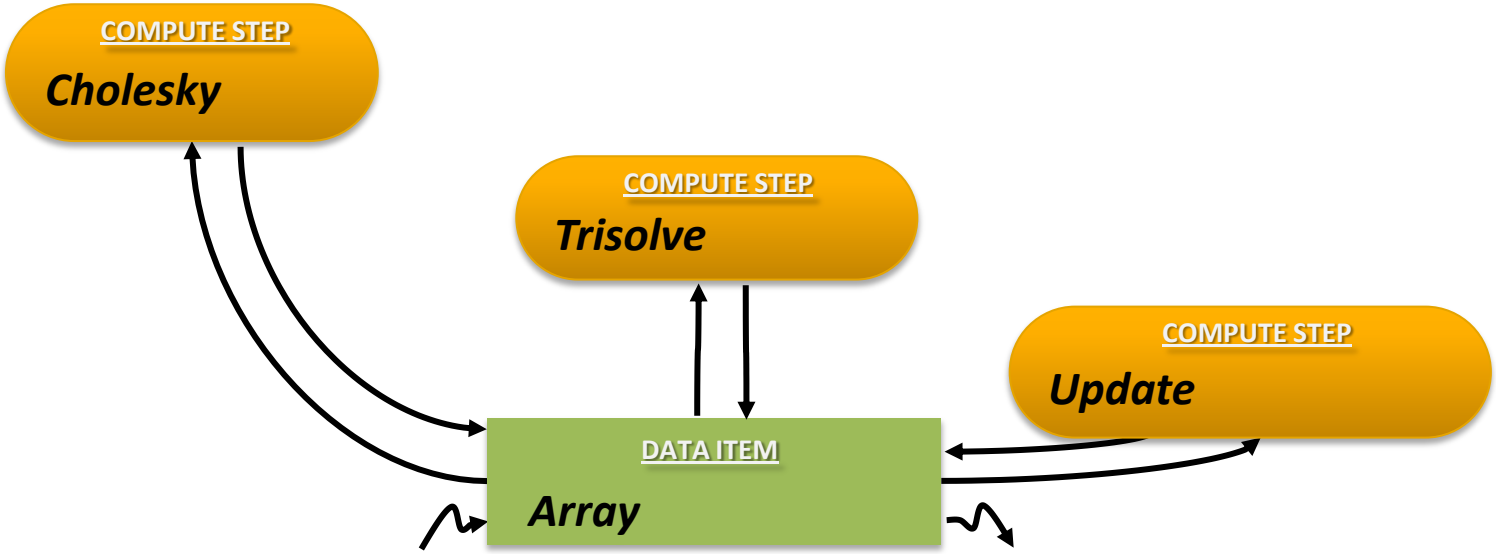
# Cholesky factorization



# 4 simple steps to a CnC application

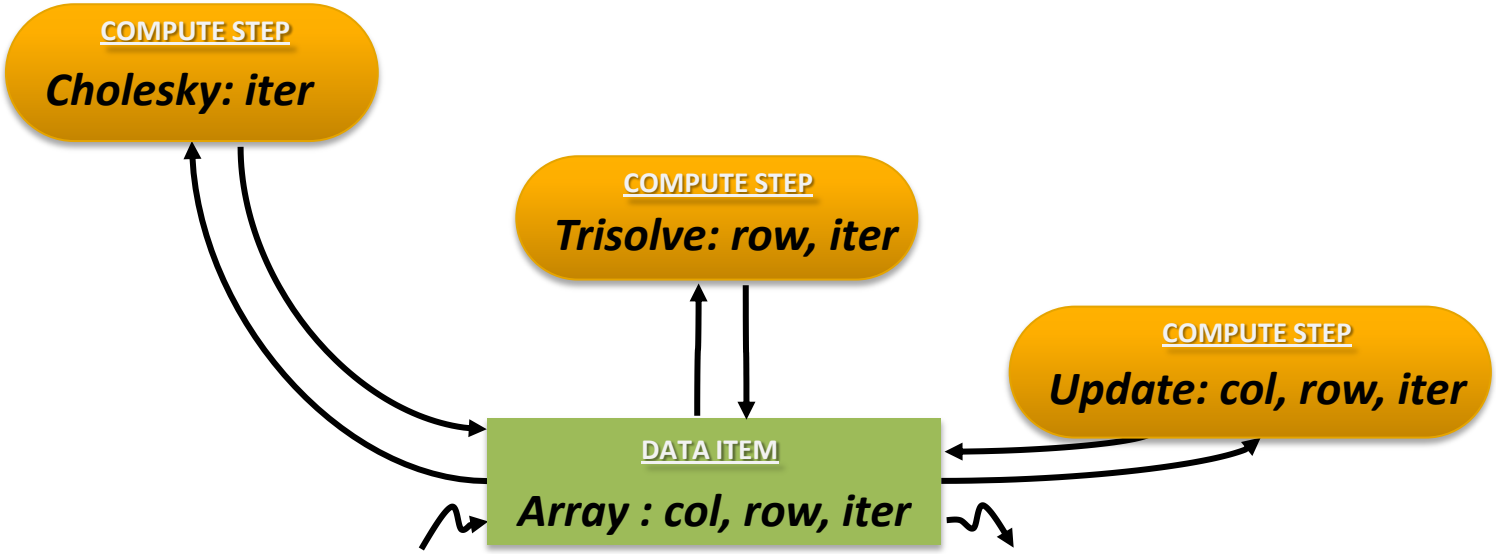
# 1:The white board level

- Data
- Computations
- Producer/consumer relations  among them
- I/O



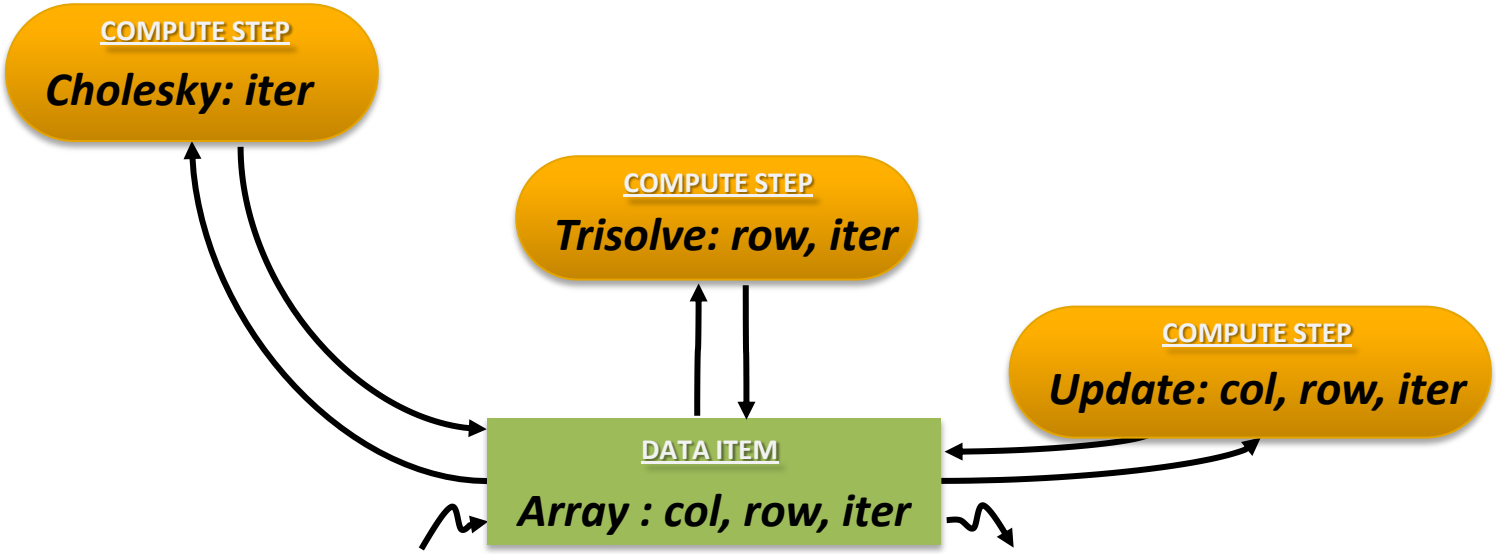
# 2: Distinguish among the instances

- Distinct instances (chunks of data or computation) may be distinctly scheduled and mapped
- A collection is a statically named set of dynamic instances

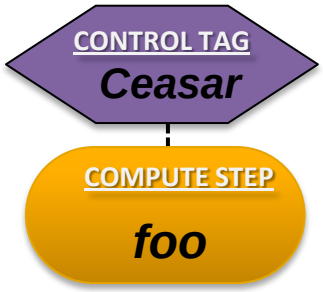


# 2: Distinguish among the instances

- Distinct instances (chunks of data or computation) may be distinctly scheduled and mapped
- A collection is a statically named set of dynamic instances
- Optional: tag functions relating the tags
  e.g., ***Cholesky: iter→ Array: iter, iter, iter***



# Control tag

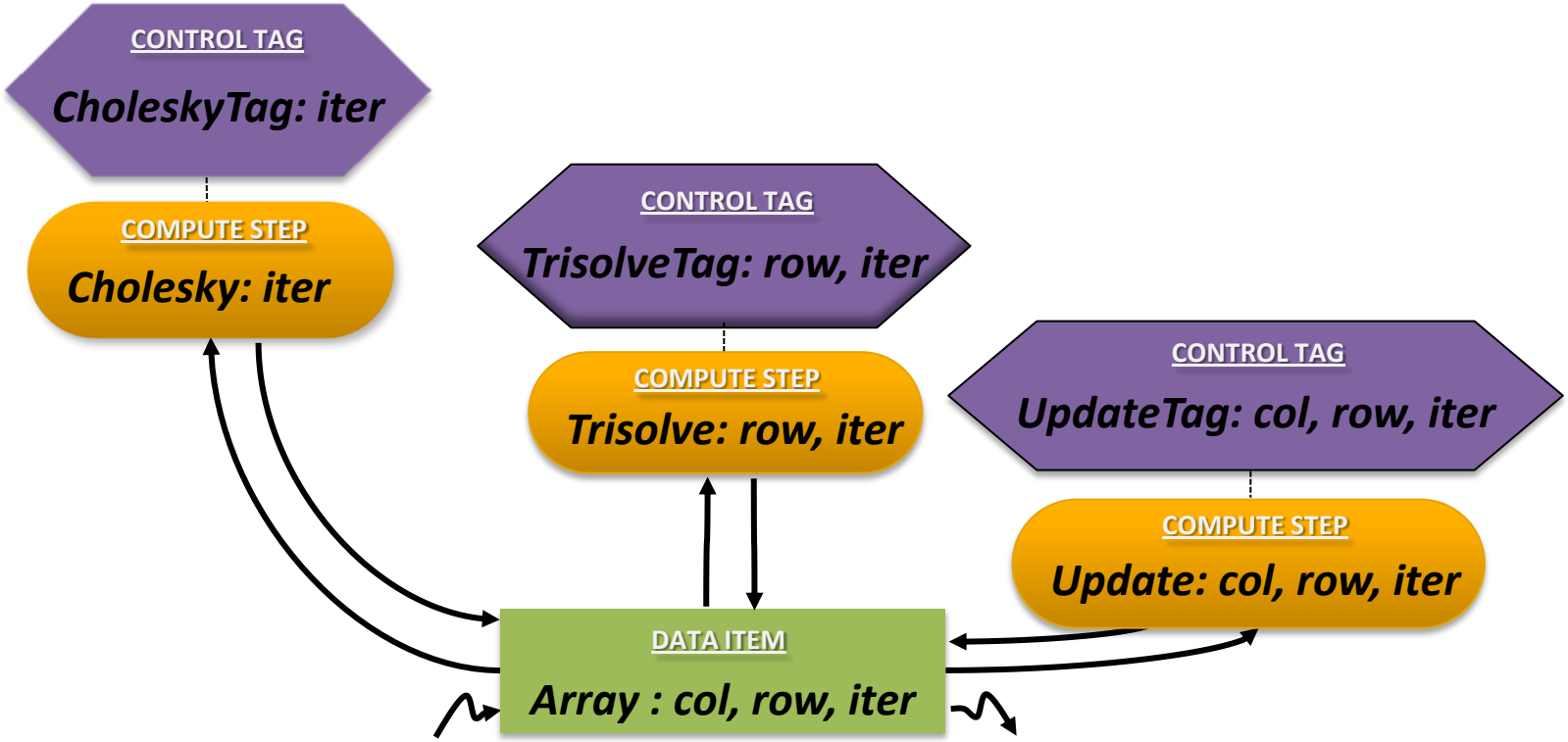



- Tuple – unique identifier
  - <row, col, iteration>
  - <socialSecurityNumber, date>
  - <treeNode>

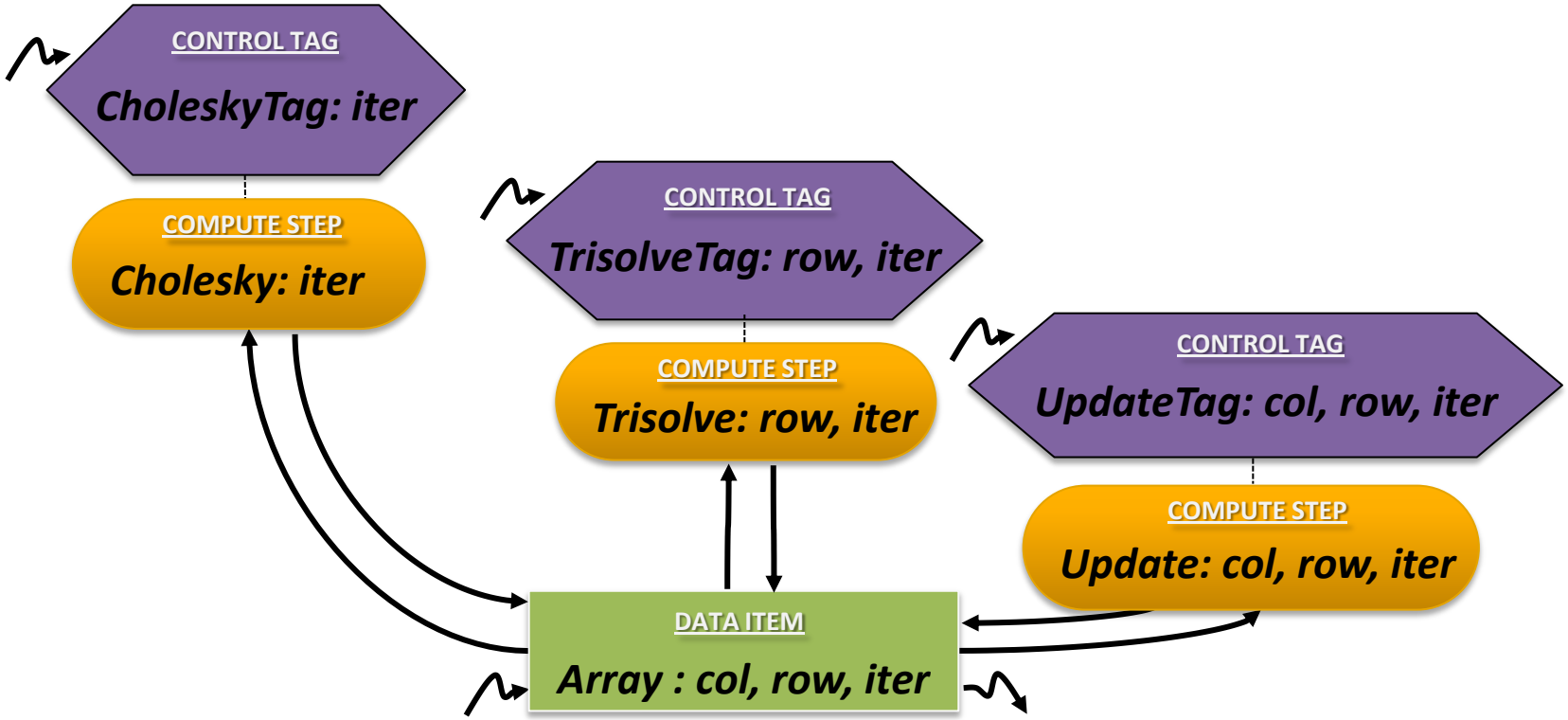
- Meaning of control relationship
  - For each row/col/iteration tuple in the collection Ceasar, the corresponding instance of foo will execute sometime.
  - An instance of foo has access to the values of row, col and iteration.

- Control dependence and data dependence are on the same footing

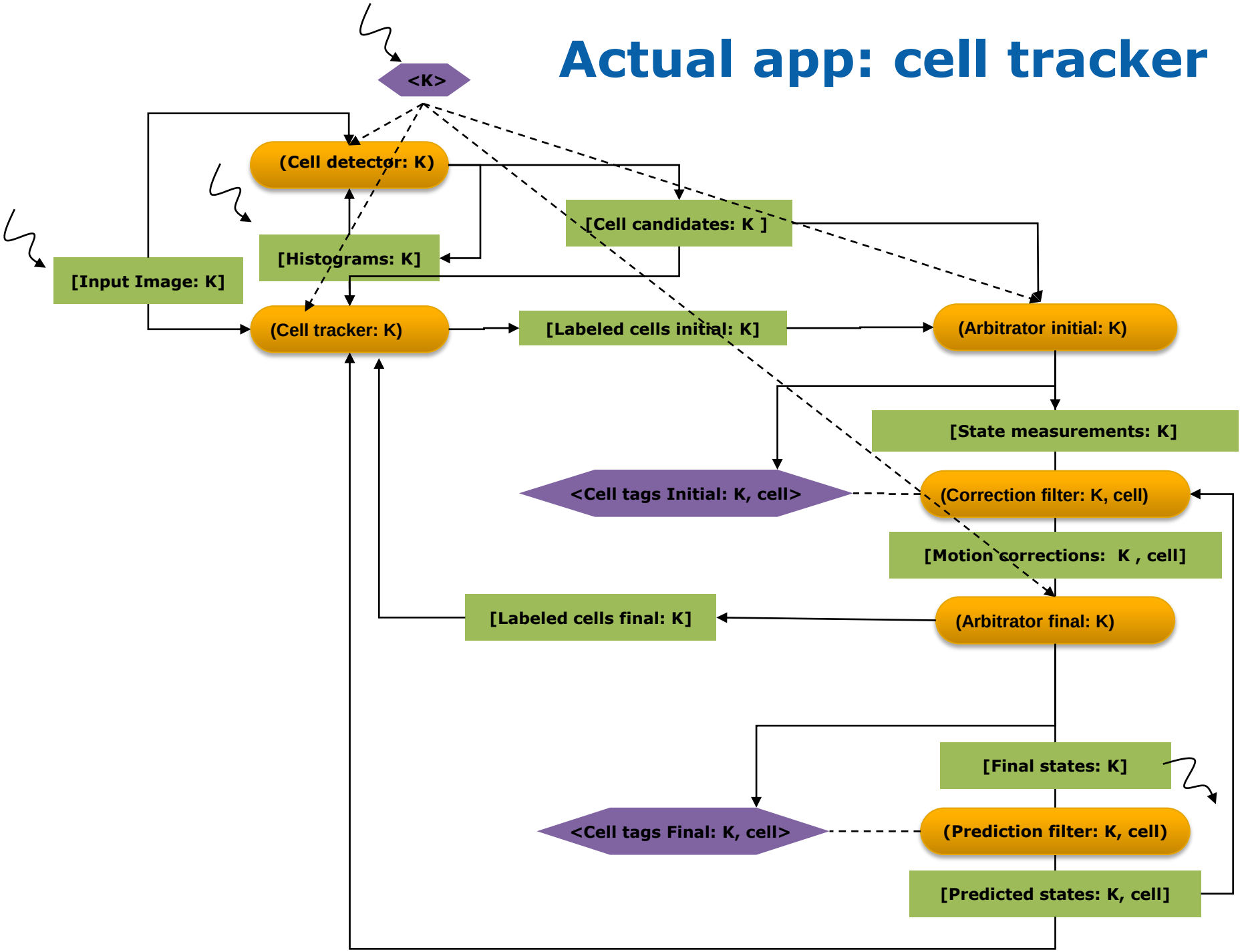
# 3: What are the control tag collections



# 4: Who produces control



# Actual app: cell tracker



# Sample step (cholesky)

```
// Perform unblocked Cholesky factorization on the input block
// Output is a lower triangular matrix.
int cholesky::execute( const int & t, cholesky_context & c ) const
{
    tile_const_ptr_type A_block;
    tile_ptr_type       L_block;

    int b = c.b;
    const int iter = t;

    c.Array.get(triple(iter,col,row), A_block); // Get the input tile.

    L_block = std::make_shared< tile_type >( b ); // allocate output tile

    for(int k_b = 0; k_b < b; k_b++) {
        // do all the math on this tile
    }

    c.Array.put(triple(iter+1,col,row), L_block); // Write the output
  tile

    return CnC::CNC_Success;
}
```

Get data used by this step

Normal computation

Put data produced by this step

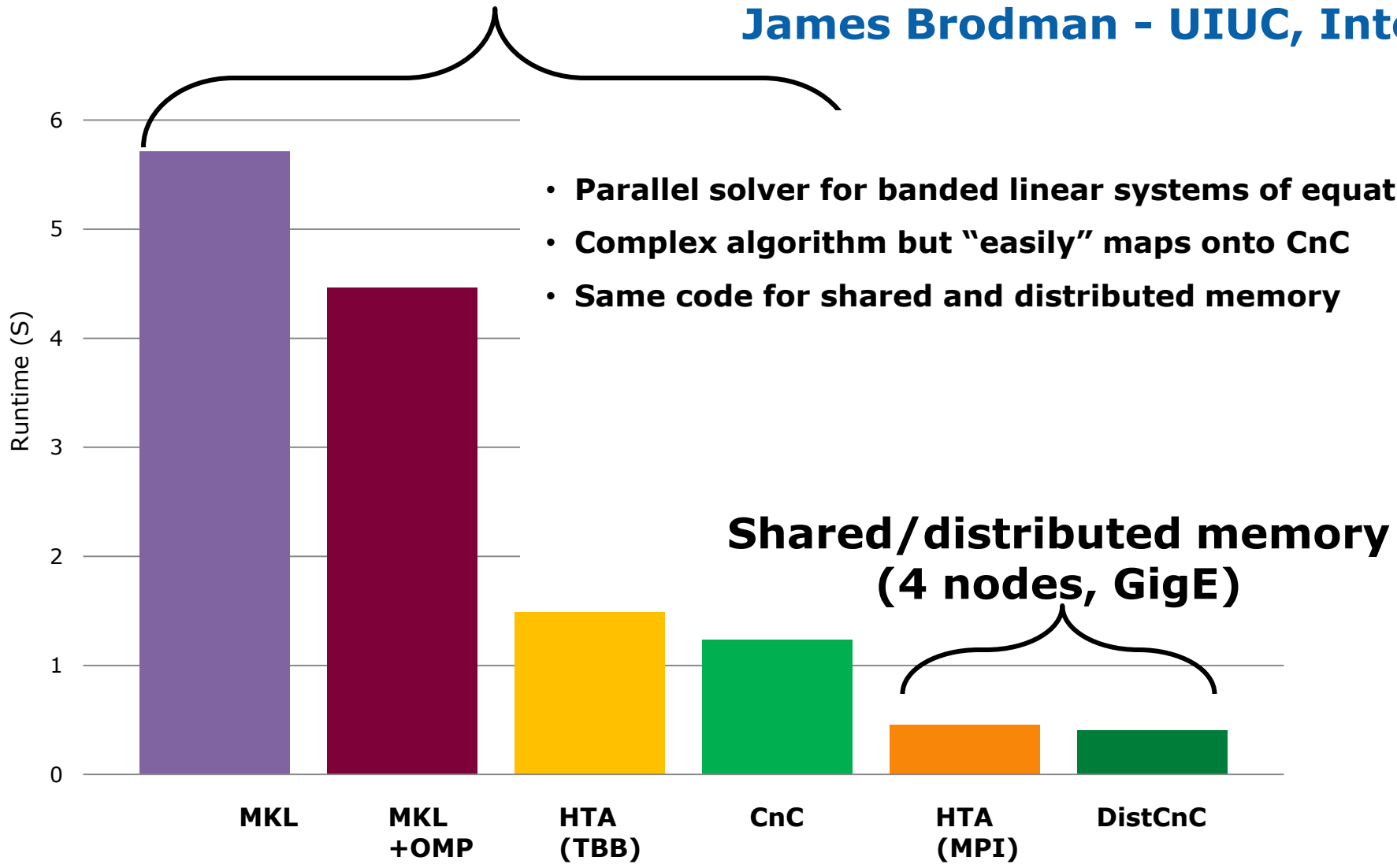
**Steps have no side-effects**

**Steps are atomic**

**Items are dynamic single assignment**

# Spike Performance

## James Brodman - UIUC, Intel

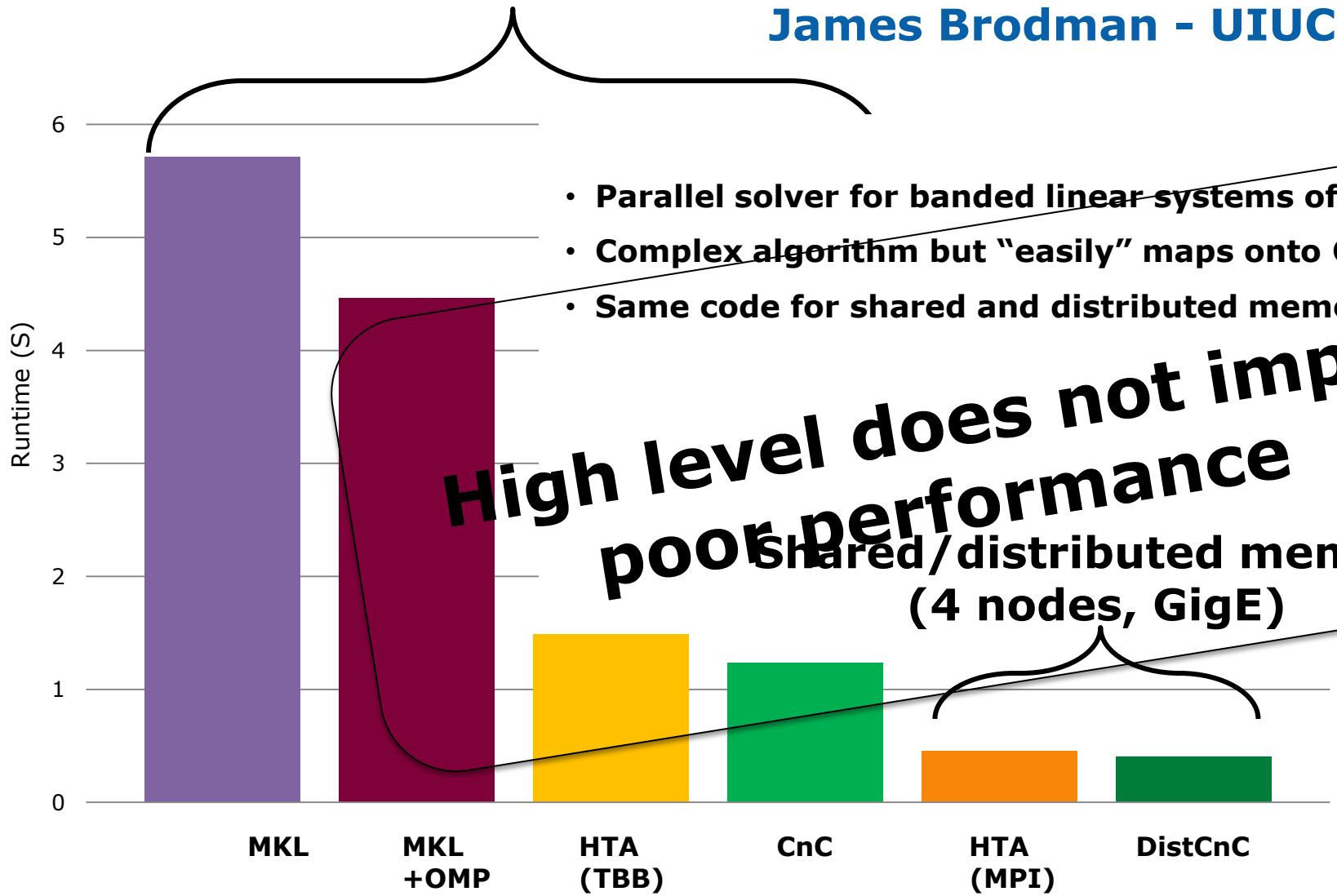



- Parallel solver for banded linear systems of equations
- Complex algorithm but "easily" maps onto CnC
- Same code for shared and distributed memory

**1048576 w/ 128 Super/Sub Diagonals, 32 partitions**

# Spike Performance

James Brodman - UIUC, Intel

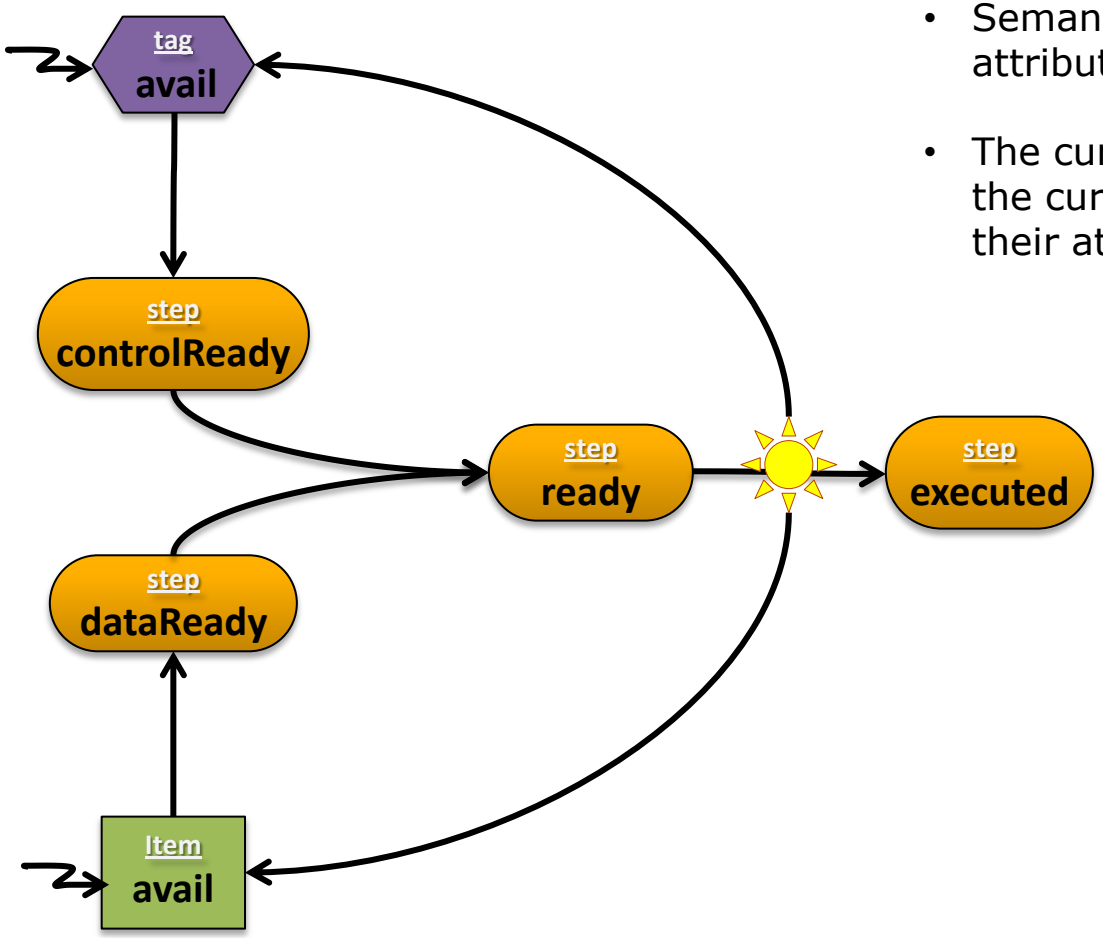



- Parallel solver for banded linear systems of equations
- Complex algorithm but "easily" maps onto CnC
- Same code for shared and distributed memory

**High level does not imply poor performance**

**1048576 w/ 128 Super/Sub Diagonals, 32 partitions**

# Semantics of CnC




- Semantics defined in terms of attributes of instances.
- The current state is described by the current set of instances and their attributes.

# CnC is a coordination language: paired with a computation language

- Existing:
  - C++ (intel)
  - Java (Rice)
  - Haskell (Indiana)
  - Python (Rice)
  - Scala (Rice)
  - APL (Indiana)
- In the works:
  - Fortran (Rice)
  - Chapel (UIUC)
- Next up:
  - Matlab (Rice)

Rice = Vivek Sarkar's group
Indiana = Ryan Newton's group
UIUC = David Padua's group

# CnC is a high level model

## allows for a wide variety of runtime approaches

| | grain | distribution | schedule |
|---|---|---|---|
| HP TStreams | static | static | static |
| HP TStreams | static | static | dynamic |
| Intel CnC | static | dynamic | dynamic |
| Rice CnC | static | dynamic | dynamic |
| Georgia Tech CnC | dynamic | dynamic | dynamic |

# Possible expressions of the model

- Graphical interface
- Textual representation
  - Rice version
  - previous Intel versions
- API
  - Current Intel release

# Key properties

- Think about the meaning of your program
  - Not about parallelism
- Deterministic & serializable
  - Easy to debug
- Separation of concerns simplifies everything
  - Computation from coordination
  - Domain spec from tuning spec
    - Domain spec is free of tuning and largely independent of architecture
- Unified programming for shared and distributed memory
  - Not like MPI/OpenMP combo

# Intel® Concurrent Collections for C++

- New whatif-release
  - Header-files, shared libs
  - Samples
  - Documentation
    - CnC concepts
    - Using CnC
    - Tutorials

- Template library with runtime
  - For Windows(IA-32/Intel-64) and Linux (Intel-64)
  - Shared and distributed memory

# Outline

- Intro to the CnC
- Tuning CnC
- CnC for distributed memory

# Tuning expert

**with Rice: Mike Burke, Sanjay Chatterjee, Zoran Budimlic**

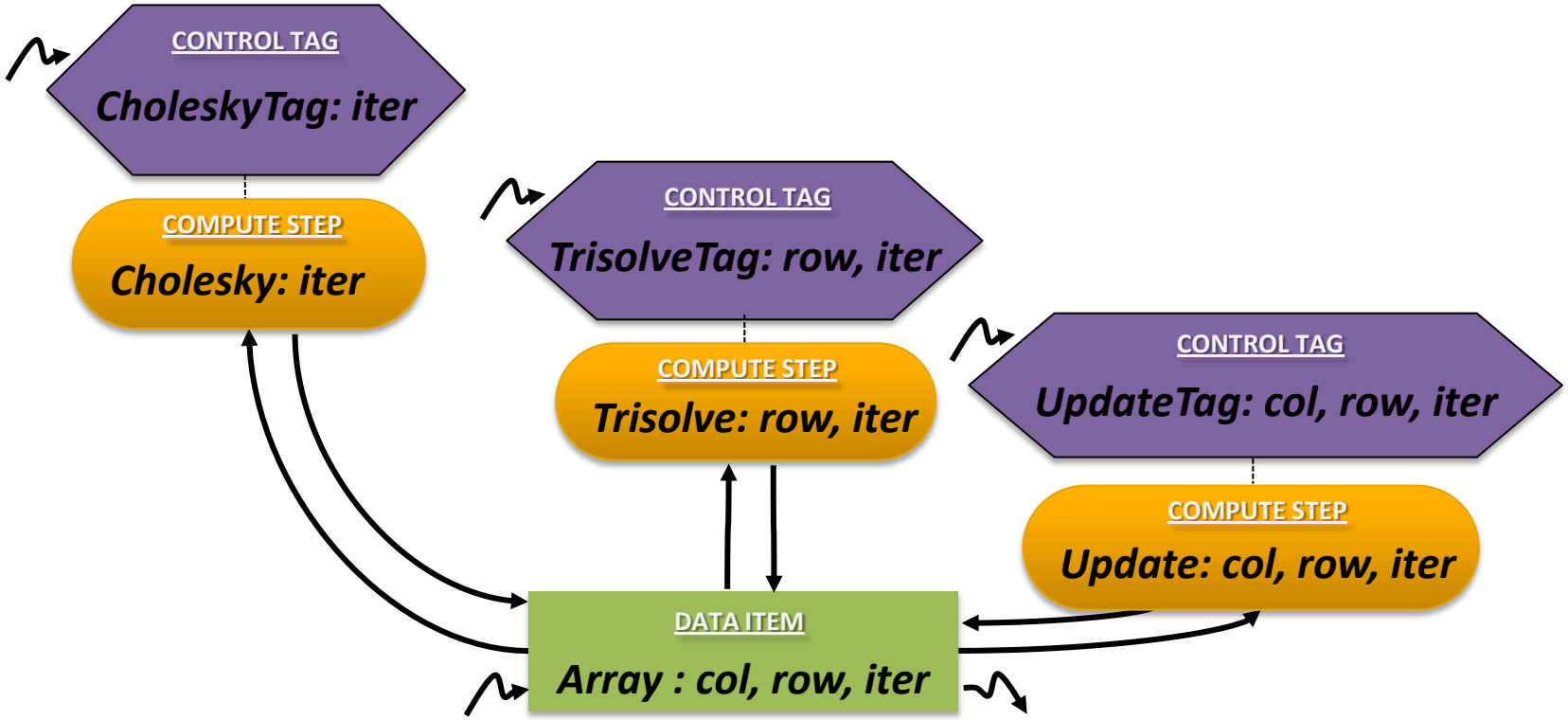
- Without tuning
  - Very good performance
  - Typically more than enough parallelism
  - Asynchronous execution

- Multiple tuning specs for each domain spec
  - Tuning spec is in addition and separate
  - Semantic constraints from the domain spec are still requirements
  - A tuned execution is a one of the many legal executions of the domain spec. Goal: guide away from bad ones.

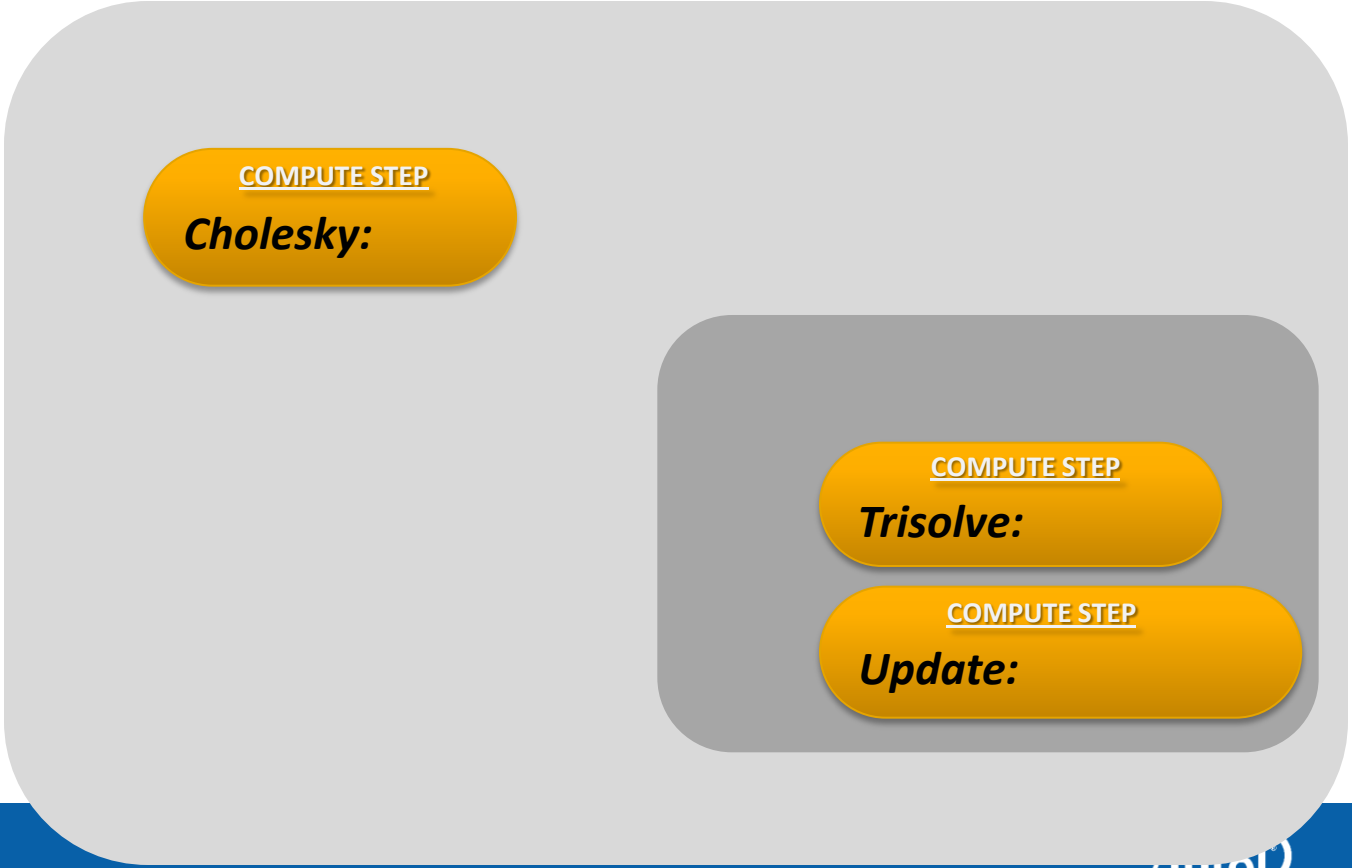
# Foundation for tuning: Hierarchical Affinity groups

- Affinity groups
  - Avoid mappings with poor locality (temporal + spatial)
    - No benefit if too far away in either time or space
    - Foundation doesn't distinguish space from time
    - On top of this foundation we allow (but don't require) time-specific and space-specific control
- Hierarchy
  - Same low-level group => tight affinity
  - Same higher-level group => weaker affinity

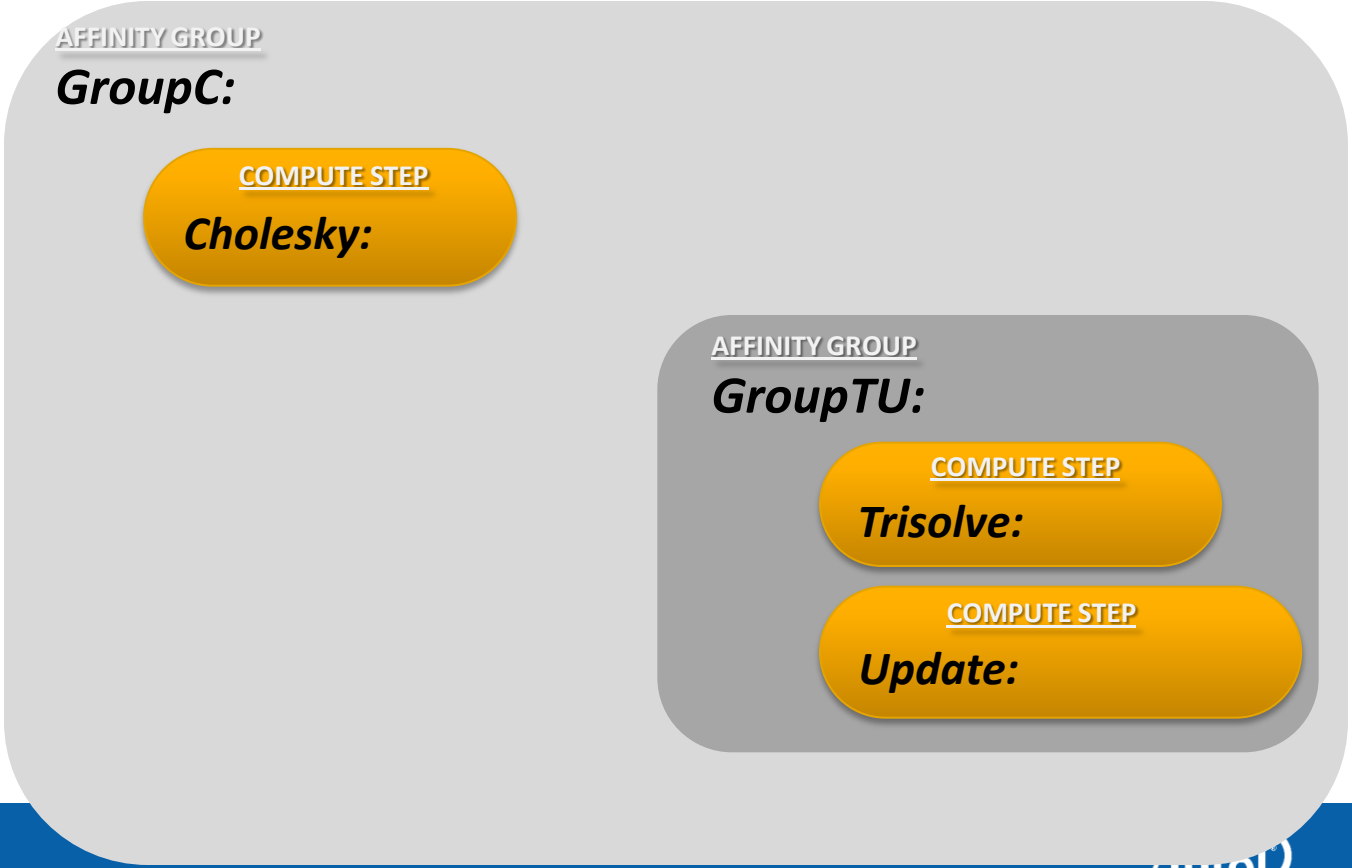
# Cholesky domain spec



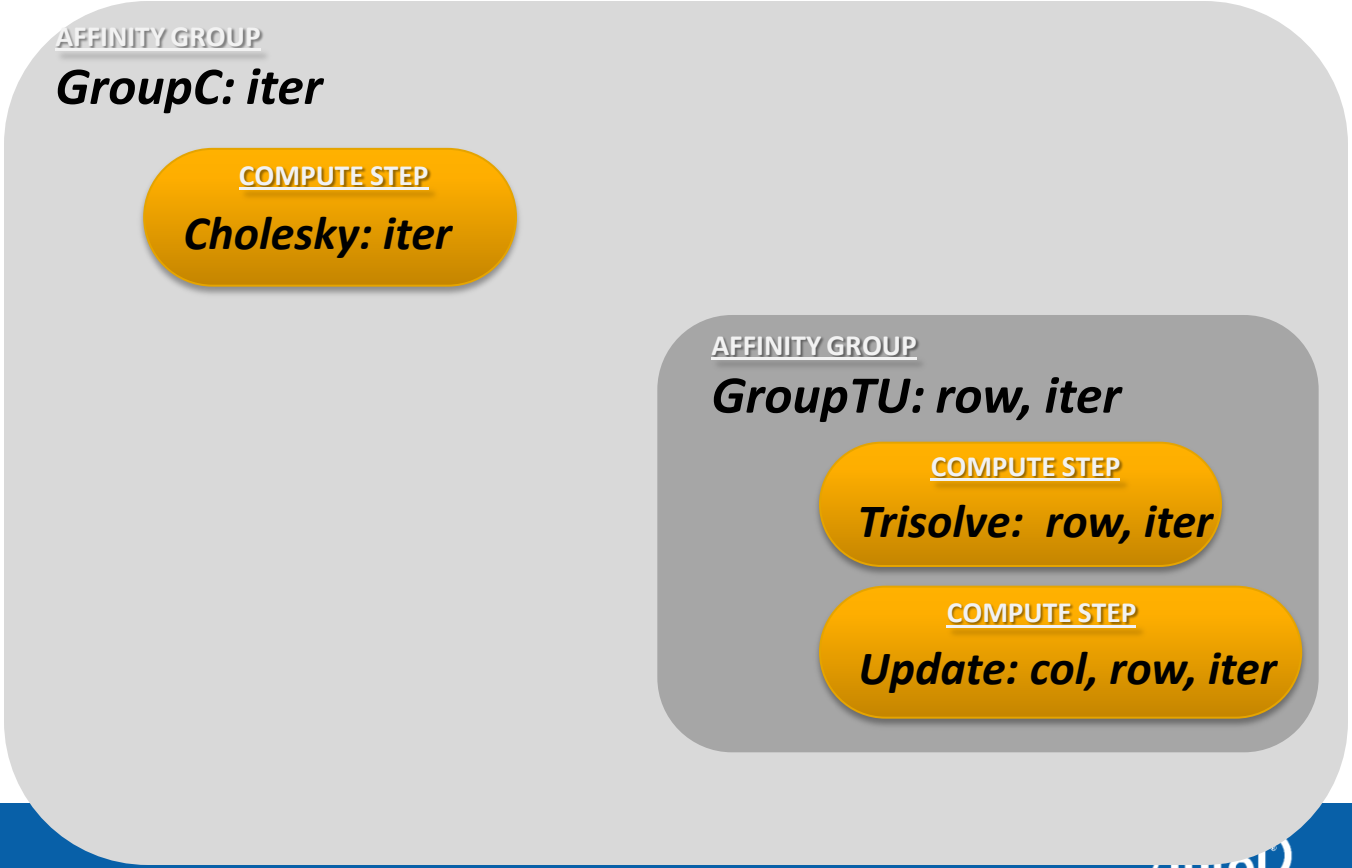
# Hierarchical affinity structure



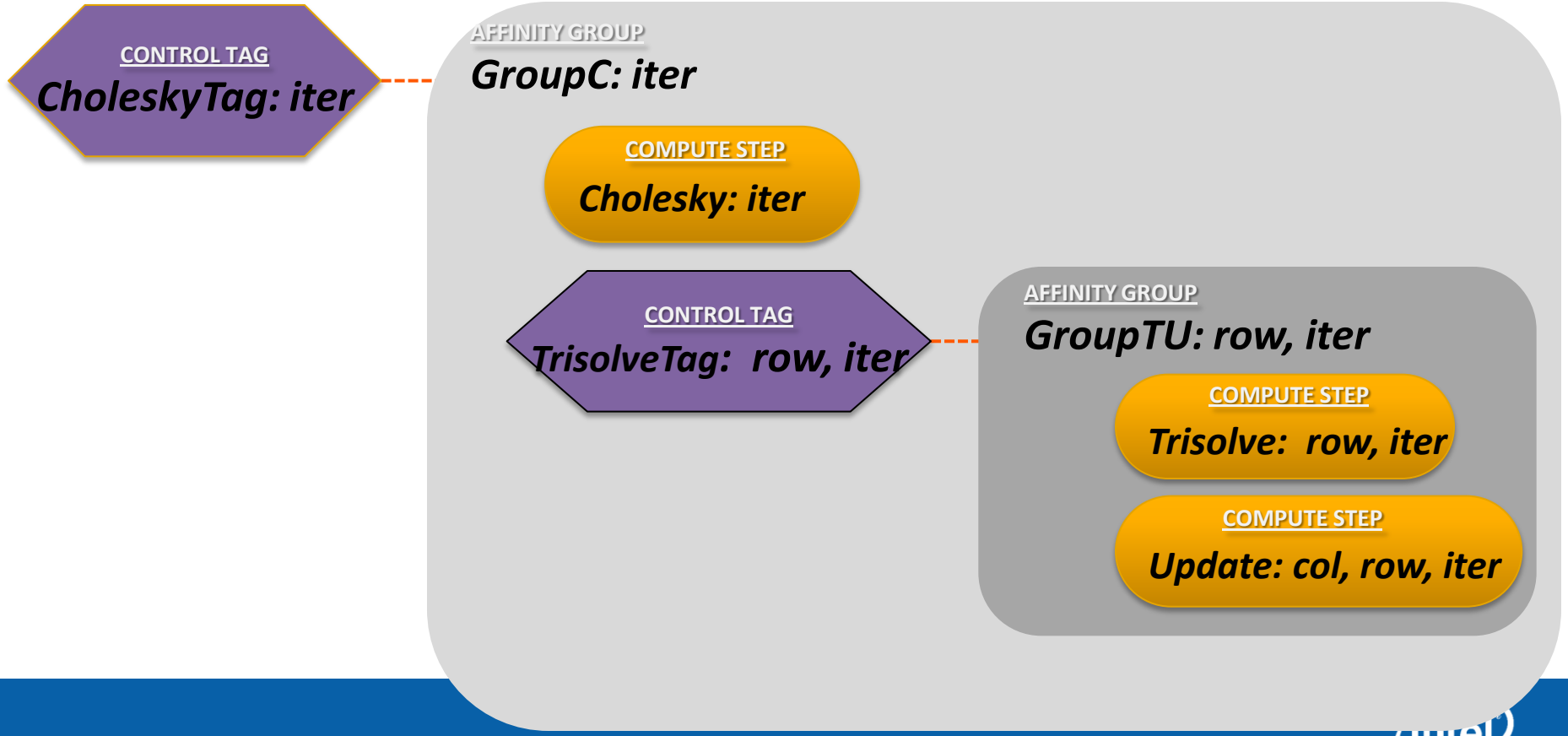
# Hierarchical affinity groups have names



# We distinguish among instances of affinity groups



# Specify the instances of each affinity group



# Cholesky: instances of the outer group

| | | | |
|---|---|---|---|
| Cholesky | | | |
| Trisolve | Update | | |
| Trisolve | Update | Update | |
| Trisolve | Update | Update | Update |

# Cholesky: instances of the outer group

| | | | |
|---|---|---|---|
| | | | |
| | Cholesky | | |
| | Trisolve | Update | |
| | Trisolve | Update | Update |

# Cholesky: instances of the inner group

| | | | |
|---|---|---|---|
| Cholesky | | | |
| Trisolve | Update | | |
| Trisolve | Update | Update | |
| Trisolve | Update | Update | Update |

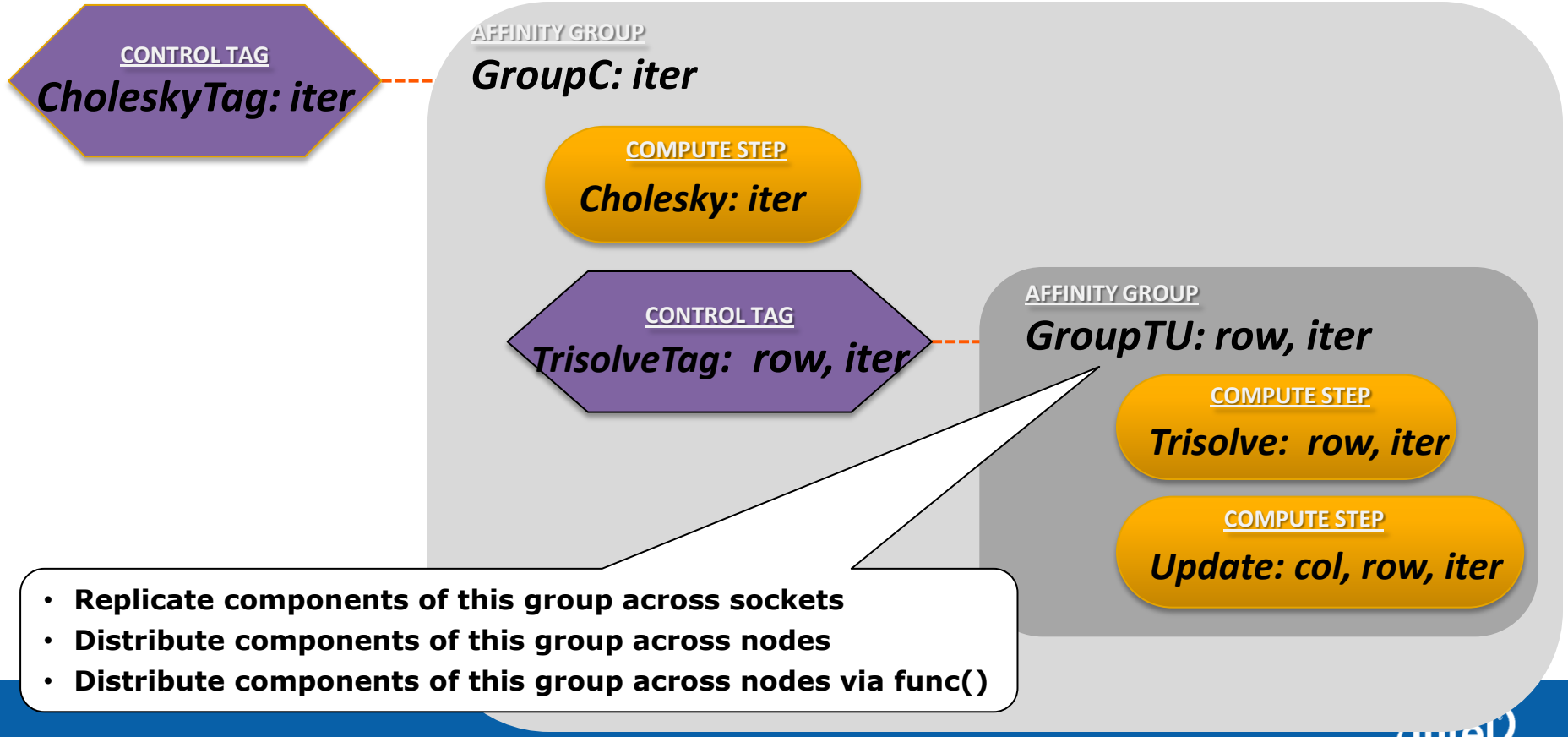
# Cholesky: instances of the inner group

| | | | |
|---|---|---|---|
| Cholesky | | | |
| Trisolve | Update | | |
| Trisolve | Update | Update | |
| Trisolve | Update | Update | Update |

# Cholesky: instances of the inner group

| | | | |
|---|---|---|---|
| Cholesky | | | |
| Trisolve | Update | | |
| Trisolve | Update | Update | |
| Trisolve | Update | Update | Update |

# Space-specific mappings




- Replicate components of this group across sockets
- Distribute components of this group across nodes
- Distribute components of this group across nodes via func()

# Time-specific mappings

**Relative time among components within a group.**

| | ordered | unordered |
|---|---|---|
| **Non-overlapping** | serial/barrier | exclusive |
| **overlapping** | priority | arbitrary |

# Space-time combinations

{big_memory_footprint: i}

Distribute across address spaces (space)
Within each address space exclusive (time)

{video_processing: frameID}

cyclic distribution among address spaces(space)
ordered & overlapping within address spaces (time)

# Related work

- Control of time
  - Various types of parallelism
    - Data parallel, task parallel, streaming, fork-join, …
- Control of space
  - Support for distributions
    - Hierarchical Place Trees, HPF, …

CnC can control both time and space and in a unified way

We allow (but don't require) the tuning expert to blur the distinction between them

Knobe & Burke "Tuning Language for Concurrent Collections" Compilers for Parallel Computing (CPC 2012)

# Outline

- Intro to the CnC
- Tuning CnC
- CnC for distributed memory

# Distributed CnC

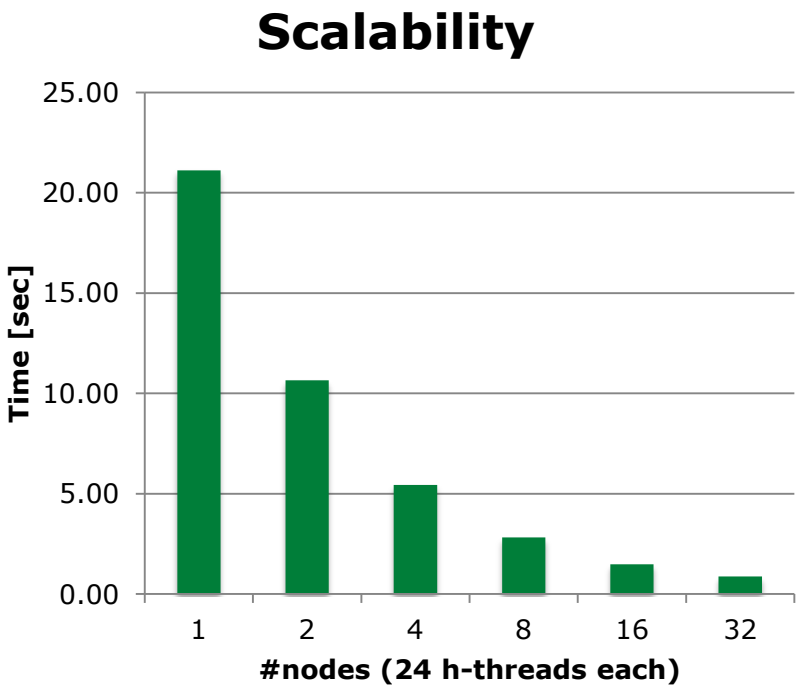
## Intel: Frank Schlimbach

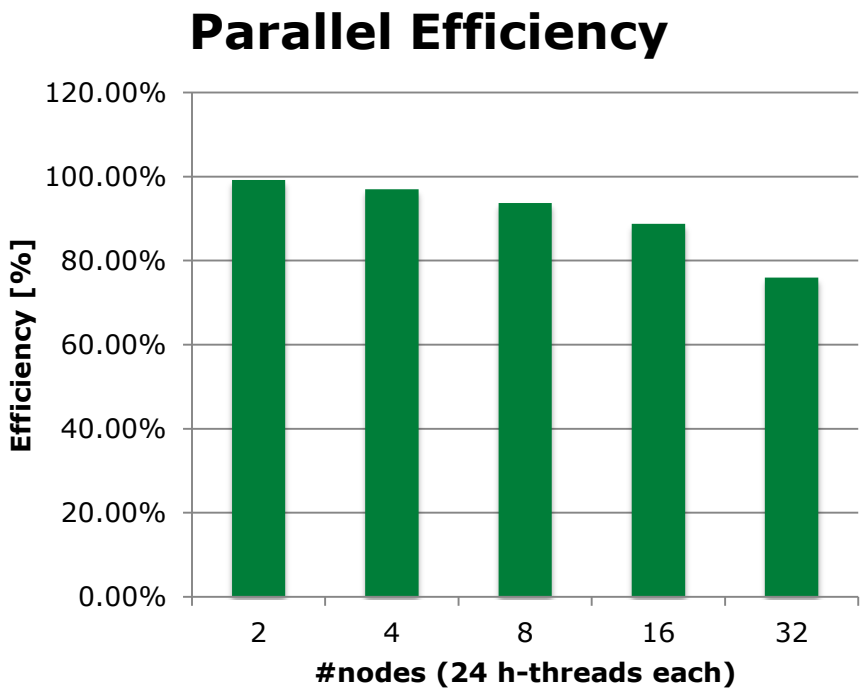
- CnC is a declarative and high level model
  - Abstracts from memory model
- Provides a unified programming model for shared and distributed memory
  - Same binary runs on both
  - Easy to switch between shared and distributed memory (no explicit message passing needed)
- **CnC comes with a control methodology**
  - **Allows controlling work distribution**
- **CnC comes with a data methodology**
  - **Provides hooks for selective data distribution**

# Limitations of distributed computing apply

- Usual caveats for distributed memory apply
  - E.g. ratio between data-exchange and computation
- Different algorithms might be needed for distributed or shared memory
  - Programming methodology and framework stays the same in any case
  - Over a wide class of applications the algorithm stays the same

# Scalability and efficiency: Spike

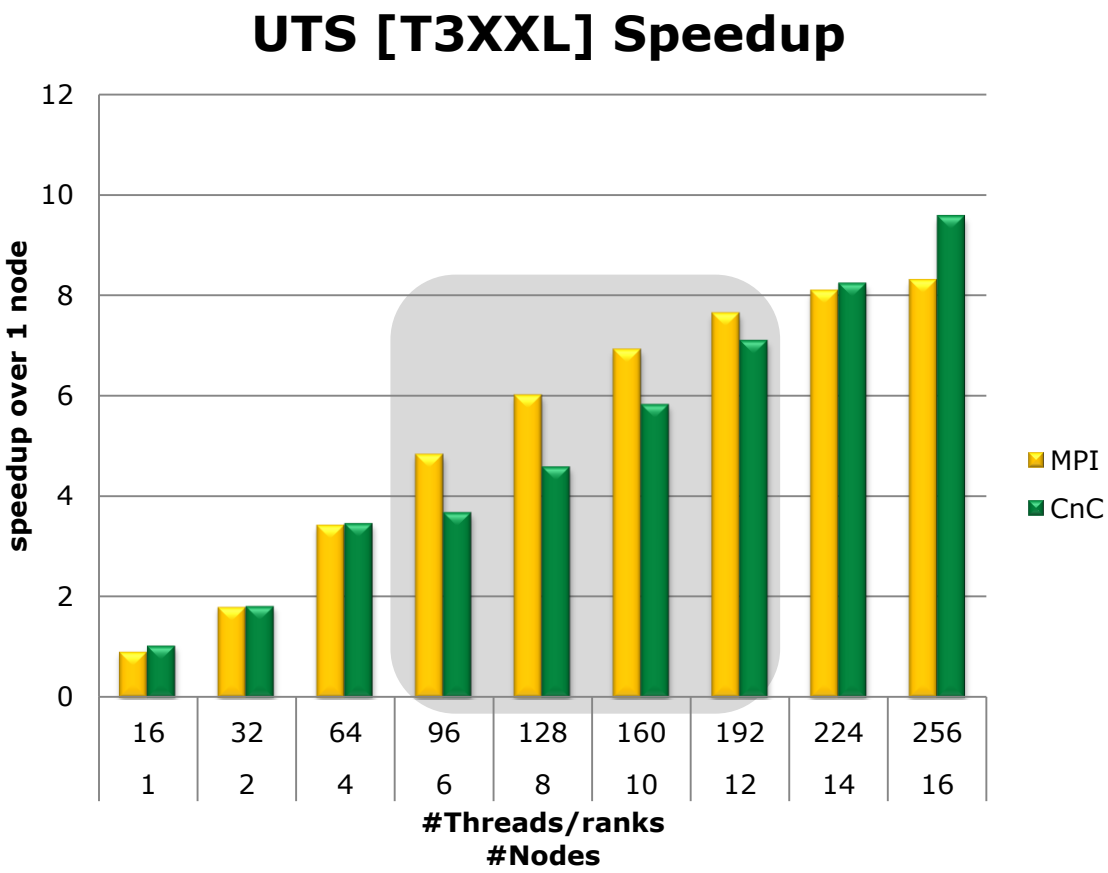


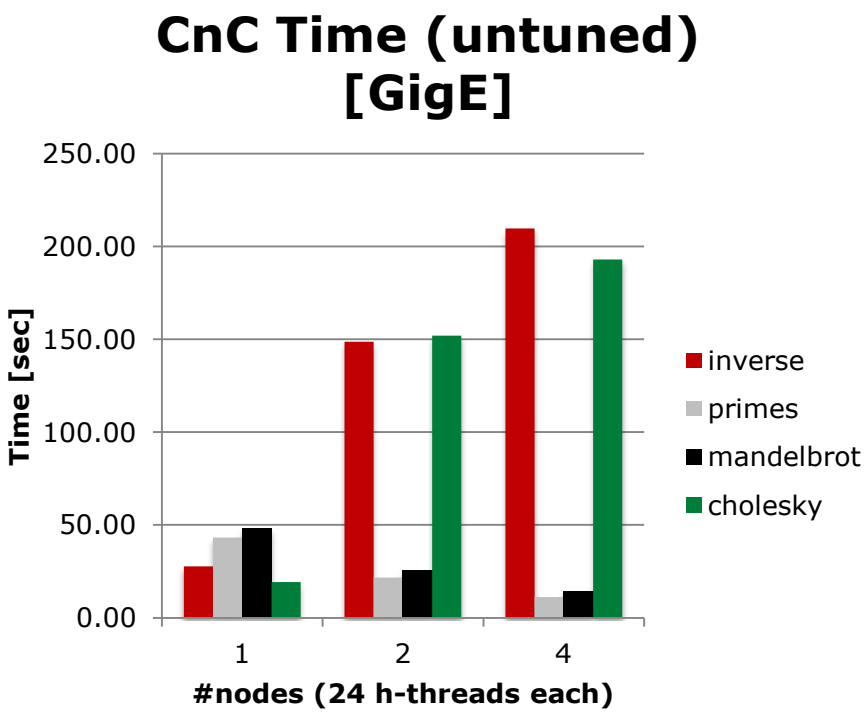

**(matrix: 10Mx10M, 257 bands)**

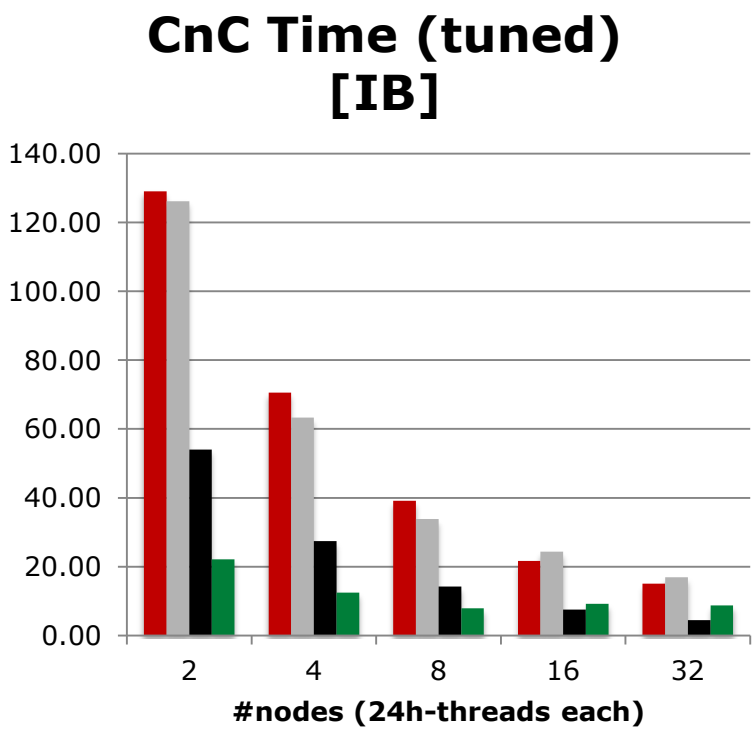
# Distributed CnC: without tuning (UTS)

- Unbalanced tree search
  - Tree shape unknown
- CnC code is trivial
  - CnC: 151 loc
  - Shmem: ~1000 loc
  - MPI: ~800 loc
- Performance gap in the middle region is a load-balancing issue
  - Used scheduler available on our release
  - Solved by different scheduler



# Untuned performance often bad: tuning



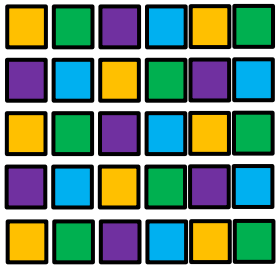
# CnC tuning easy and efficient

- The general default distribution of steps
  - Data is sent to where needed (requested)
  - Not always efficient
- Tuner can declare functions
  - **`consumed_by:`** Which steps use this data item (semantics)
  - **`compute_on:`** Where the step will execute (tuning)
  - To redistribute, modify **`compute_on:`**
    data items and steps automatically adjusted
- Mapping tag->rank can be computed
  - Compile time
  - Initialization time
  - Execution time

# Distribution functions

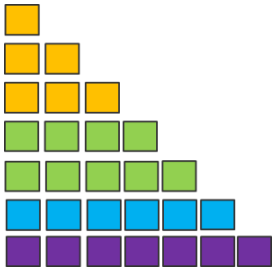
## Cyclic - Simple

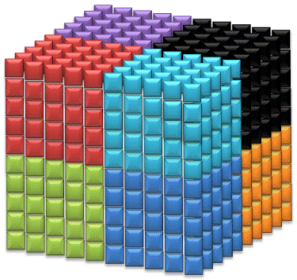
```
int tuner::compute_on( const int & i ) const
{
     return i % numProcs();
}
```



## Blocked - Minimizes data transfer

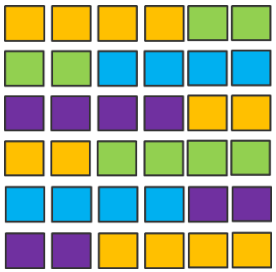
```
int tuner::compute_on( const int& x, y, z ) const
{
  return  x * NBLOCKS_X / nx
        + ( y * NBLOCKS_Y / ny ) * nx
      + ( z * NBLOCKS_Z / nz ) * nx * ny
}
```

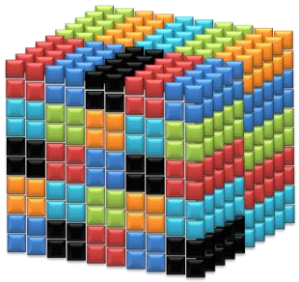




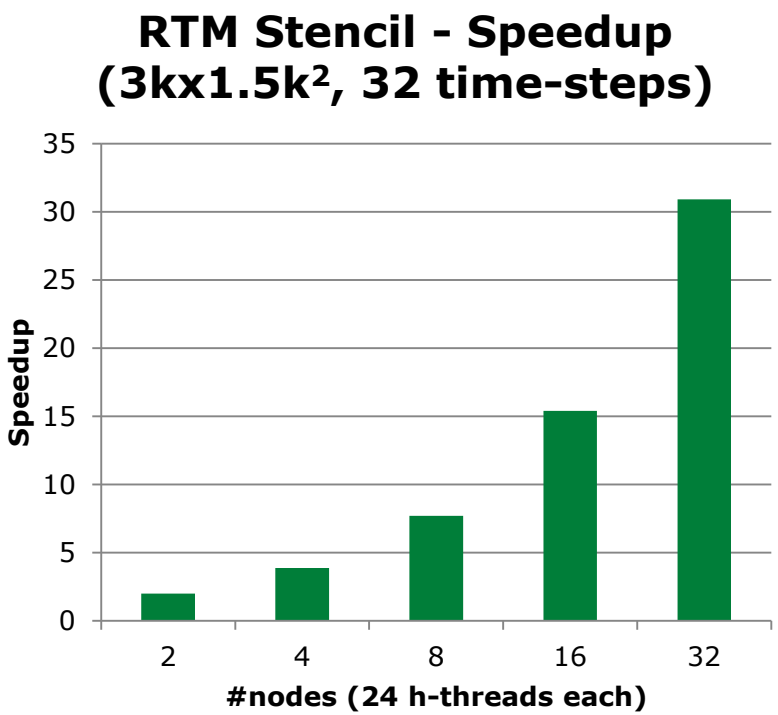
## Blocked cyclic – Block size/shape is relevant

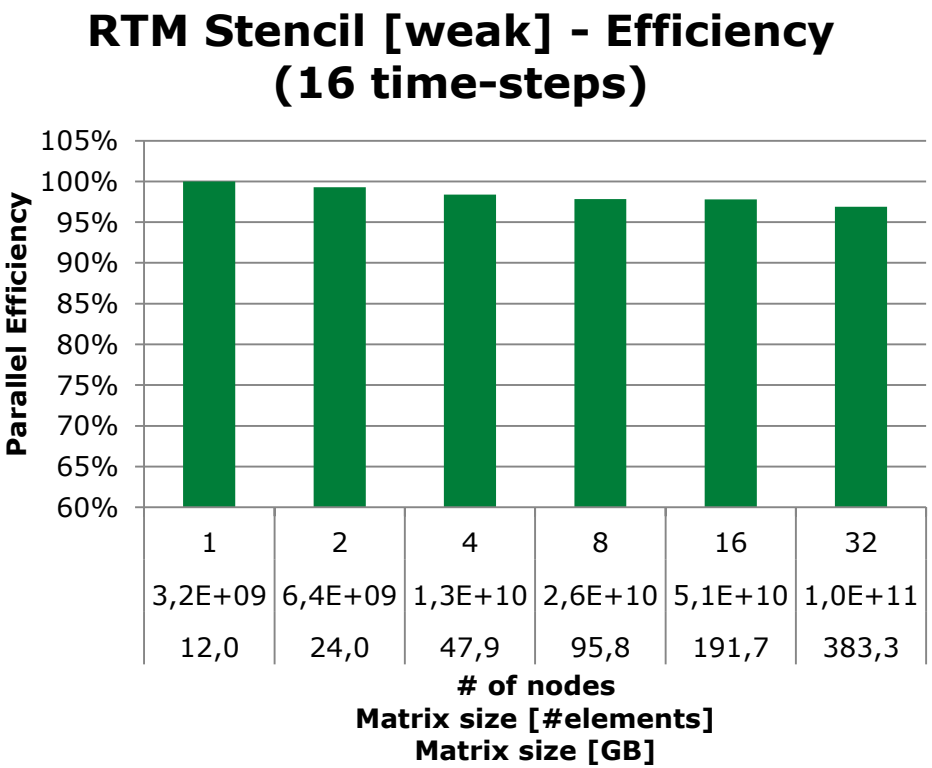
```
int tuner::compute_on( int x, y ) const
{
     return ((x + y * m_nx) / BLOCKSIZE) % numProcs();
}
```





# Scalability and efficiency: stencil



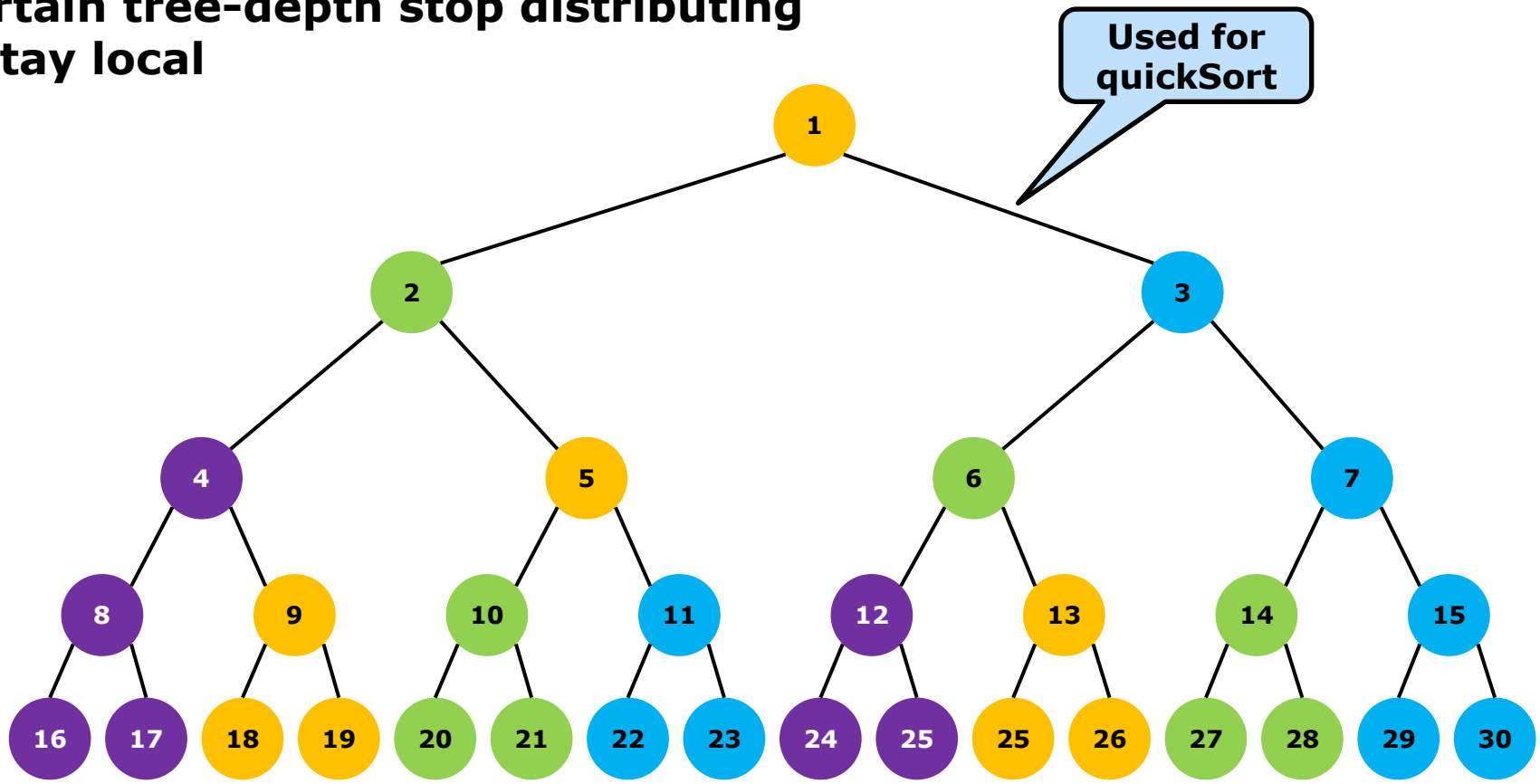




**Efficiency stays above 97%!**

**3D 25-point stencil**
**reads from 2 previous time-steps**
**Shmem performance similar to cache-oblivious algorithms**

# Distribution function for tree

**At certain tree-depth stop distributing and stay local**



# Summary

- Separation of concerns
  - Computation from coordination
  - Domain spec from tuning spec
- Domain spec
  - High-level: hides all the difficulties of low-level techniques
  - Exposes potential parallelism
  - Easy to debug (deterministic)
  - Efficient even with no tuning
- Tuning spec
  - Isolated
  - Simple
  - Flexible
- Support for distributed memory
  - Distribution is easy because the domain spec already identifies chunks of work and data.
  - Same spec runs on shared and/or distributed memory
  - Distribution functions: isolated to support easy experimentation

# Status

- General
  - Available
  - Yearly workshops
  - Used in some intro programming classes as a gentle introduction to parallelism
- Domain language
  - Very good performance on shared memory even without tuning language
    - Bunch of benchmarks
    - Comparable performance to other tools
- Tuning language
  - Ubiquitous High Performanc Computing (UPHC)
    - Vivek Sarkar's group at Rice is currently implementing the tuning language for CnC.
- Distributed CnC
  - Available
  - Next: static analysis to generate distribution functions

**New Intel CnC release:**
**http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/**

**New Intel CnC Yahoo Group:**
**http://tech.groups.yahoo.com/group/intel_concurrent_collections/**

**Rice CnC:**
**http://habanero.rice.edu/cnc**


This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.


intel
Software

# Optimization Notice

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2®, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

# Legal Disclaimer



**http://intel.com/software/products**