Colorado State University

Executing Scientific Task Graphs Using CnC

Christopher D. Krieger & Michelle Mills Strout
krieger@cs.colostate.edu

CnC 2011

Context & Related Work

- Evaluation of a DAG with CnC [Hampton 2009]
  - Today's presentation extends that work by implementing task graphs in CnC in additional ways and by providing an extensive performance comparison
- Performance Evaluation of CnC on High Performance Multicore Computing Systems [Chandramowlishwaran et al. 2010]
  - Today's talk covers a different class of scientific problems (sparse vs dense) and compares the same parallelization scheme using task graphs across different parallel models, rather than varying factorizations and parallel models

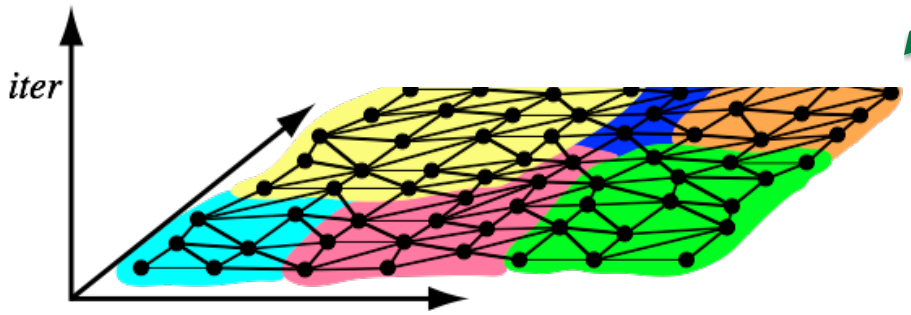
Outline

- Representing Sparse Scientific Algorithms using Task Graphs
- Implementing Task Graphs in CnC
- Performance & Scalability of CnC Task Graph Execution vs "the competition"

Target Algorithms

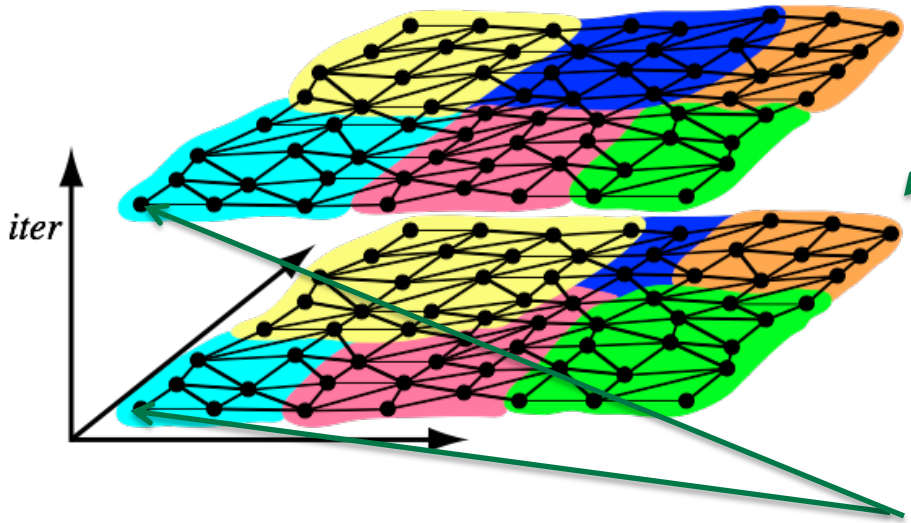
- In this work, we are investigating full sparse tiled:
  - Jacobi Sparse Matrix Solver
  - $A^k x$ Matrix Powers Kernel
  - MolDyn Molecular Dynamics Simulation

From DOALL to Task Graph



1) DOALL parallelism exists within a single timestep/ iteration/operation

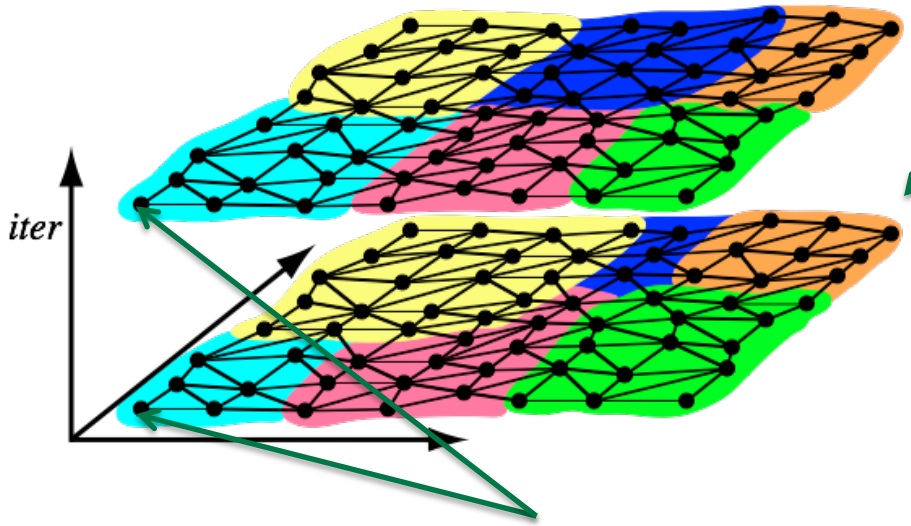
From DOALL to Task Graph



1) DOALL parallelism exists within a single timestep/ iteration/operation

2) Tiling across iterations turns data reuse into temporal locality

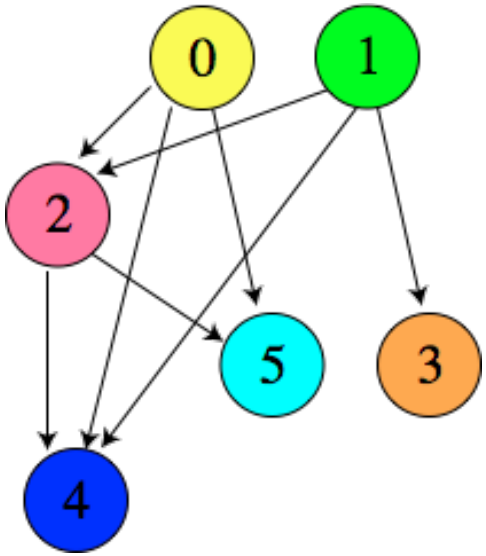
From DOALL to Task Graph



1) DOALL parallelism exists within a single timestep/ iteration/operation

2) However, tiling across iterations turns data reuse into temporal locality

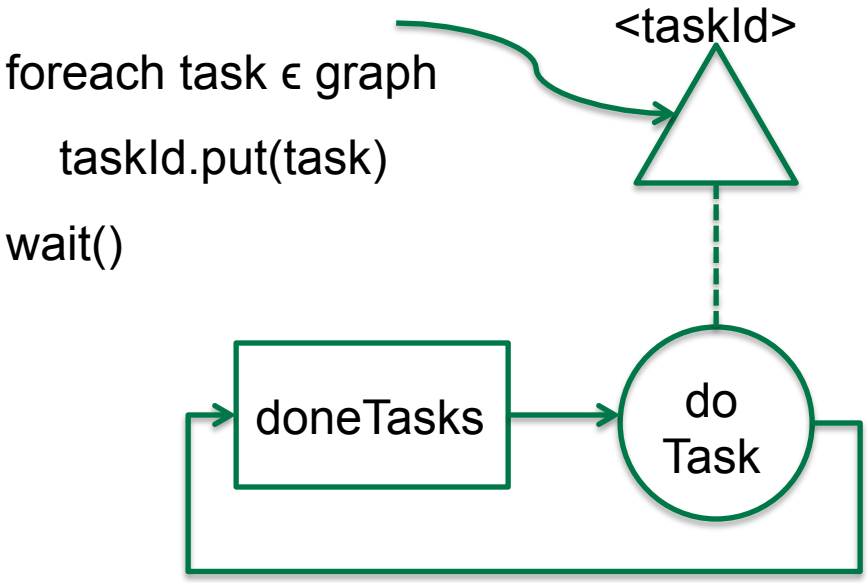
3) Tiling across the outer loop results in a partial ordering between tiles due to dependences between outer loop iterations, shown in the task graph



Outline

- Representing Sparse Scientific Algorithms using Task Graphs
- **Implementing Task Graphs in CnC**
- Performance & Scalability of CnC Task Graph Execution vs “the competition”

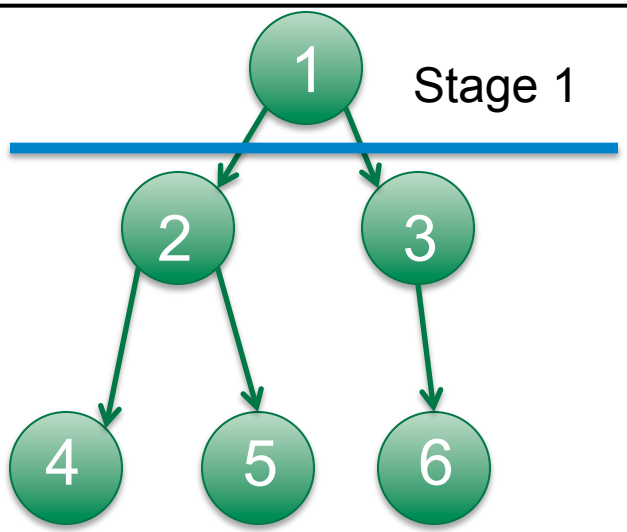
Implementing Task Graphs in CnC: Universal Engine Approach




```
doTask(<tag> id) {
  foreach (pred ϵ predecessors(id))
            doneTasks.get(pred)
  doAlgorithmWork()
  doneTasks.put(id,DONE)
}
```

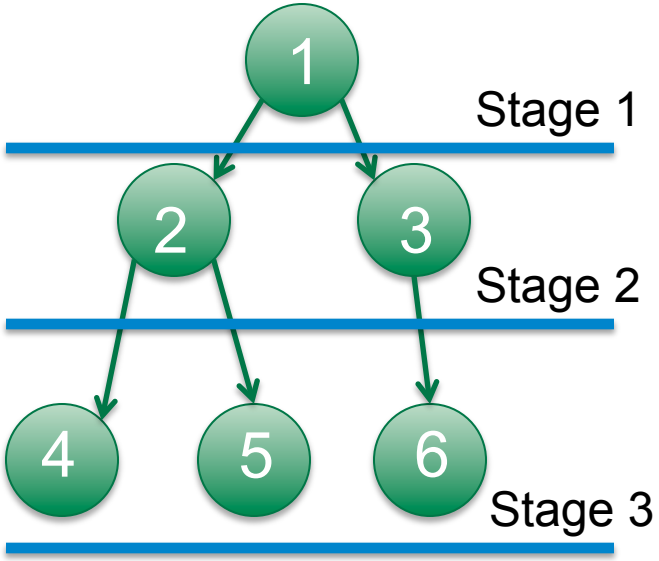
- No real data is passed via item collections
  - [doneTasks] is a collection of "completion tokens"
  - Step instance requires predecessor list for that node
- Each task may be requeued multiple times
- For large graphs, many step instances in flight

Frontier Scheduling



1) Put entry nodes into stage 1

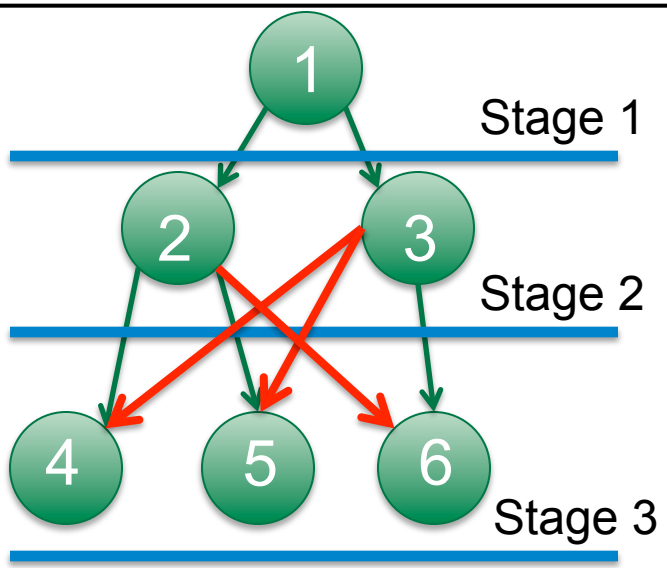
Frontier Scheduling



1) Put entry nodes into stage 1

2) Determine nodes in other stages by putting each node in earliest stage in which all predecessors are in previous stages

Frontier Scheduling



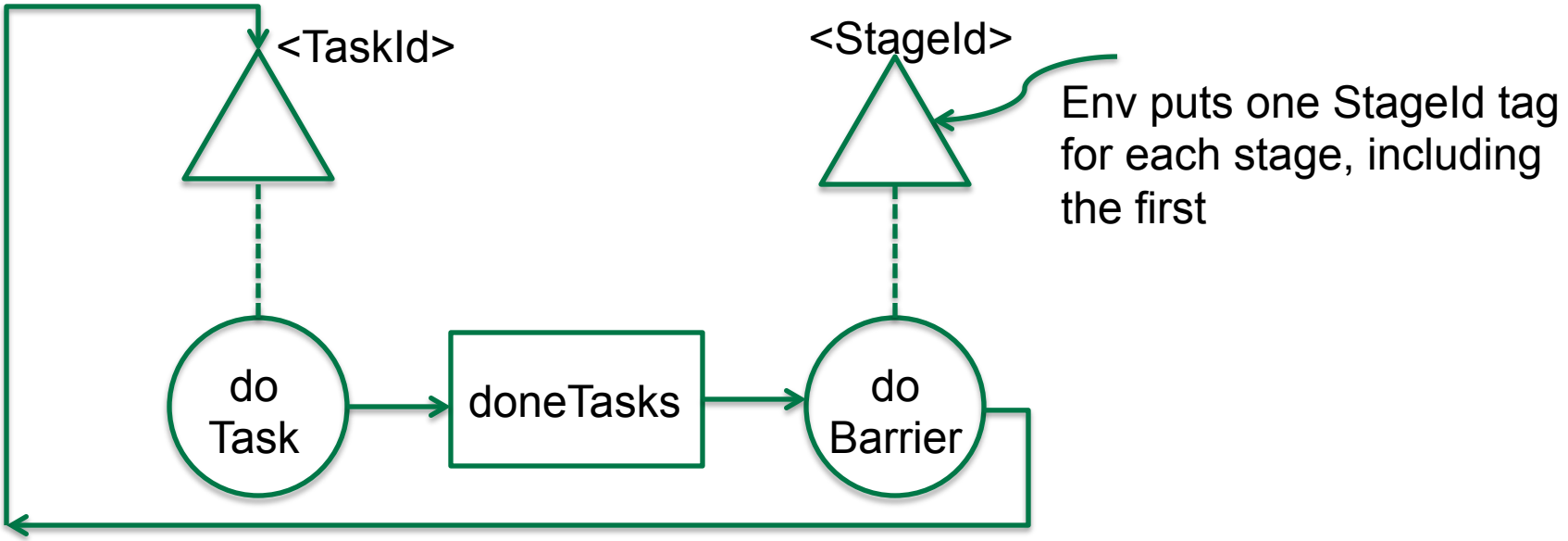
Greatly reduces number of prescribed step instances

- Averages N/(critical path length)

Disadvantages:

- False dependencies decrease async parallelism

Implementing Task Graphs in CnC: Frontier Approach



- doBarrier
  - gets() a doneTasks item for each task in the stage preceding the stageId stage
  - puts() a TaskId tag for each task in stageId stage

Use of Tuners

- Tuners
  - Created a subclass of default_tuner
  - Implemented depends()
- Created variants of both the Universal and Frontier Engines, using depends() on all prescriptions
  - All the depends information is determined before graph execution begins and is available from the task graph

Outline

- Representing Sparse Scientific Algorithms using Task Graphs
- Implementing Task Graphs in CnC
- **Performance & Scalability of CnC Task Graph Execution vs Other Models**

Current Status

- Implemented a common task graph infrastructure:
  - Task graphs, algorithms, engines
  - Allows swapping in and out different parts
- Created task graph execution engines using:
  - CnC universal and frontier approaches (w/ & w/o tuners)
  - TBB graph model (Community Preview)
  - OpenMP 3.0 task model and node parallel for
  - Cilk Plus
  - pThreads
  - Serial

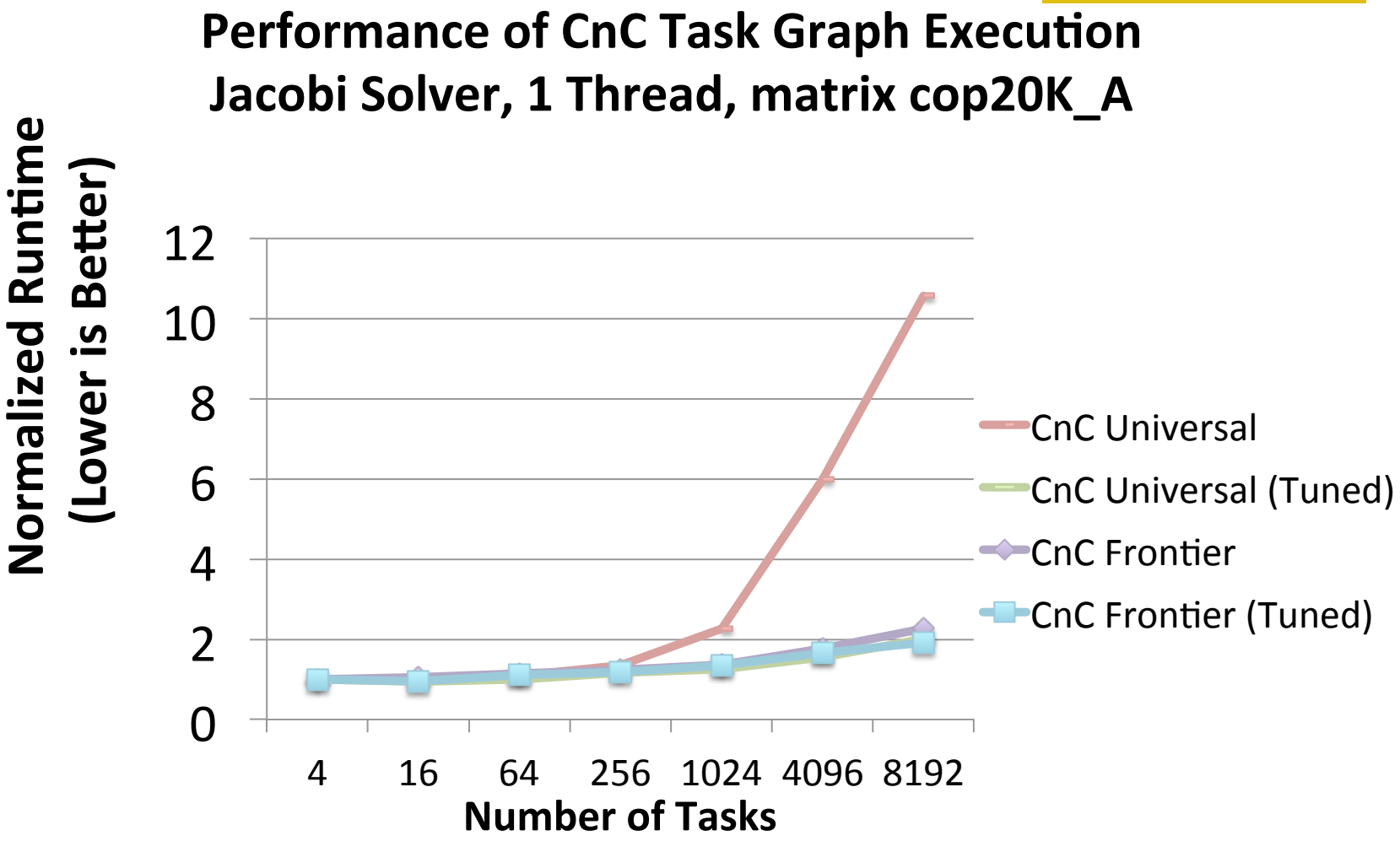
Evaluation Software and Hardware

Processor	Freq	Sockets x Cores	First Level Cache	Mid Level Cache	Last Level Cache	Total Mem (GB)	Stream Triad Score
Core i7 920 (Bloomfield)	2.66 GHz	1 x 4	32 kB	256 kB	8 MB shared / 4 cores	6	13.2 GB/s
Xeon E5450 (Harpertown)	3.0 GHz	2 x 4	32 kB	256 kB	6 MB shared / core pair	16	4.02 GB/s
Xeon E7-4860 (Westmere) *	2.26 GHz	4 x 10	32 kB	256 kB	24 MB shared / 10 cores	256	75.6 GB/s

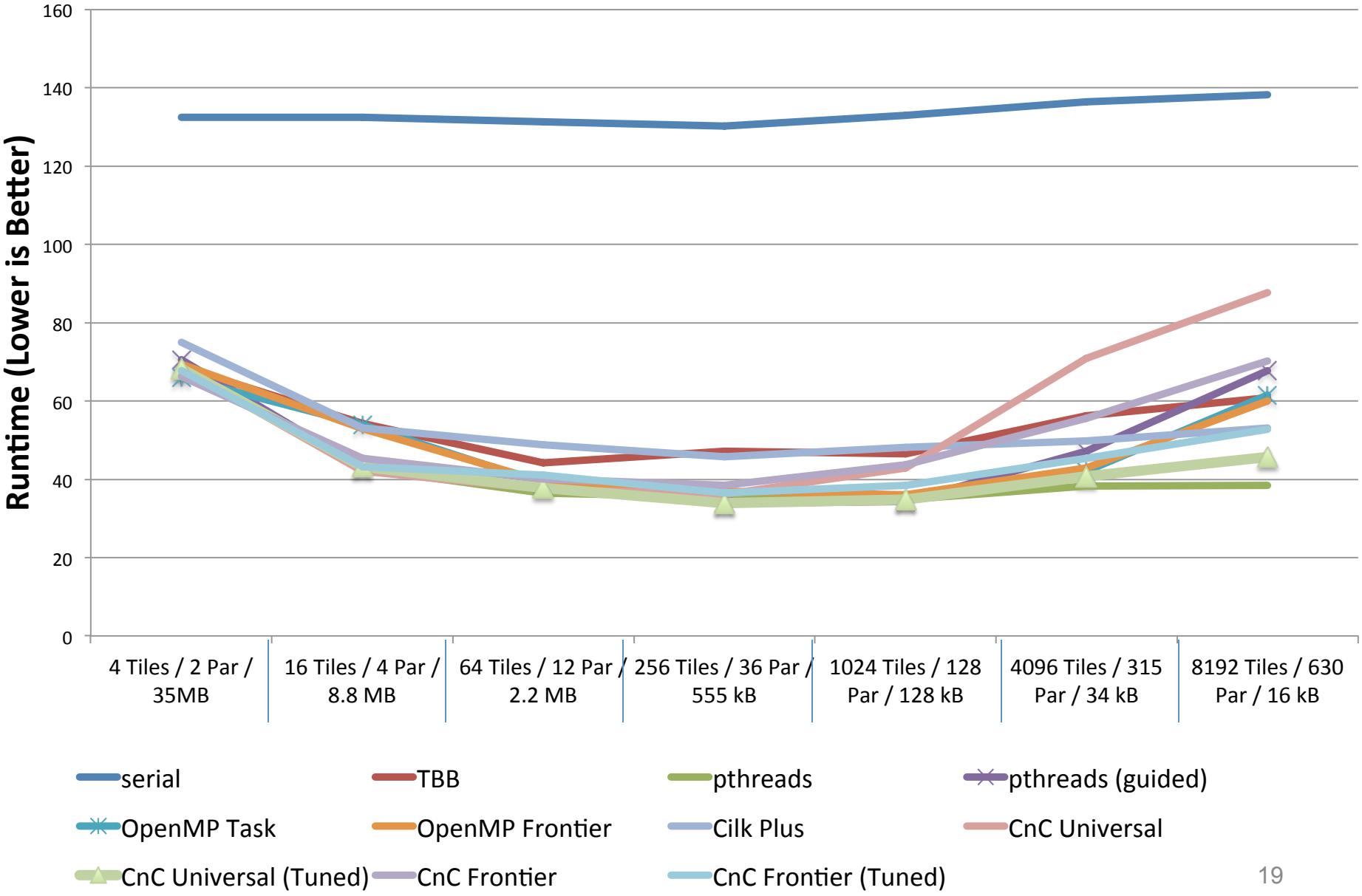
Intel icpc 12.0.2 20110112, TBB 3.0, OpenMP 3.0, CnC 0.6_03

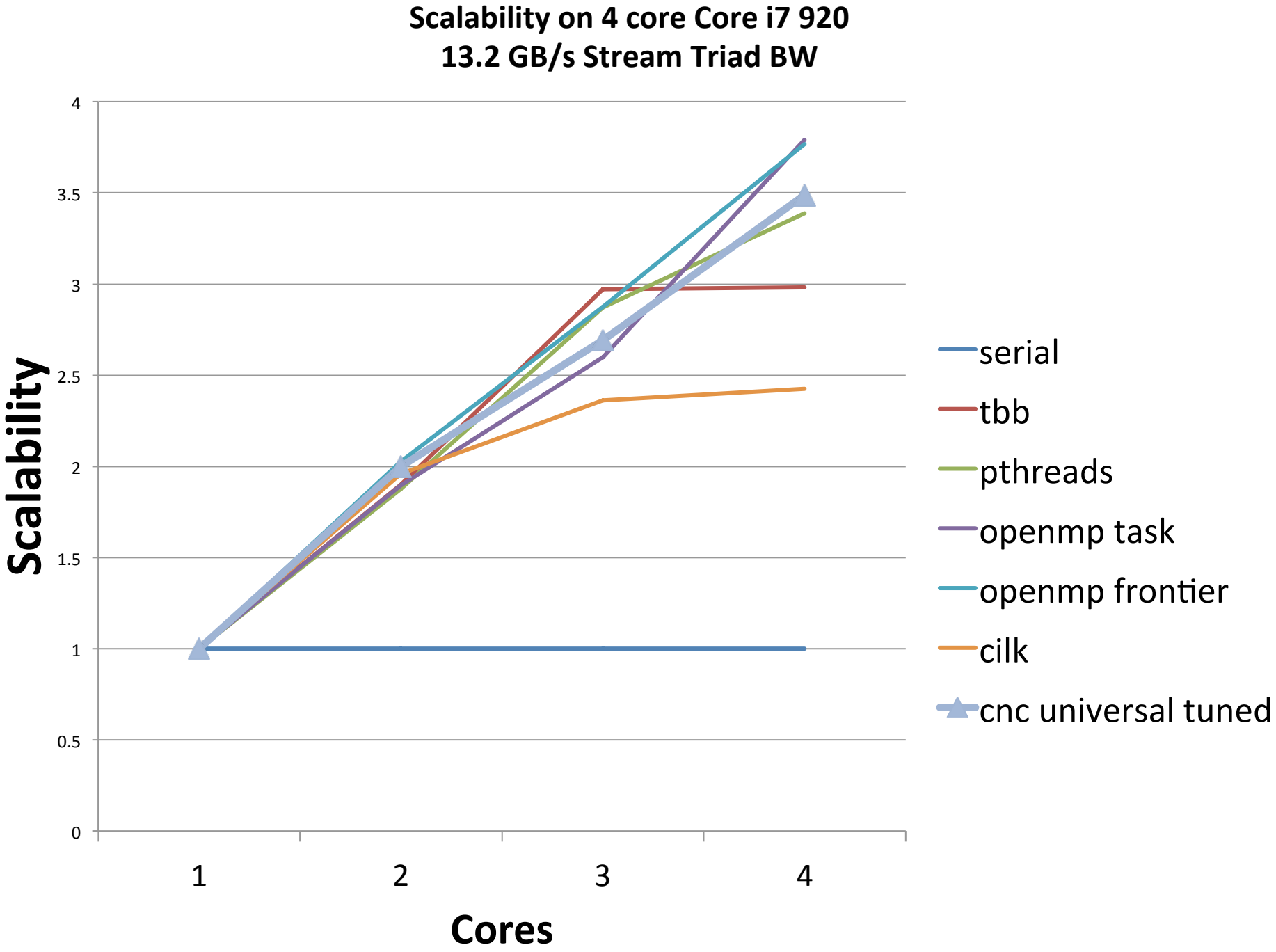
* Special thanks to Mike Pearce and Intel's Multicore Testing Lab

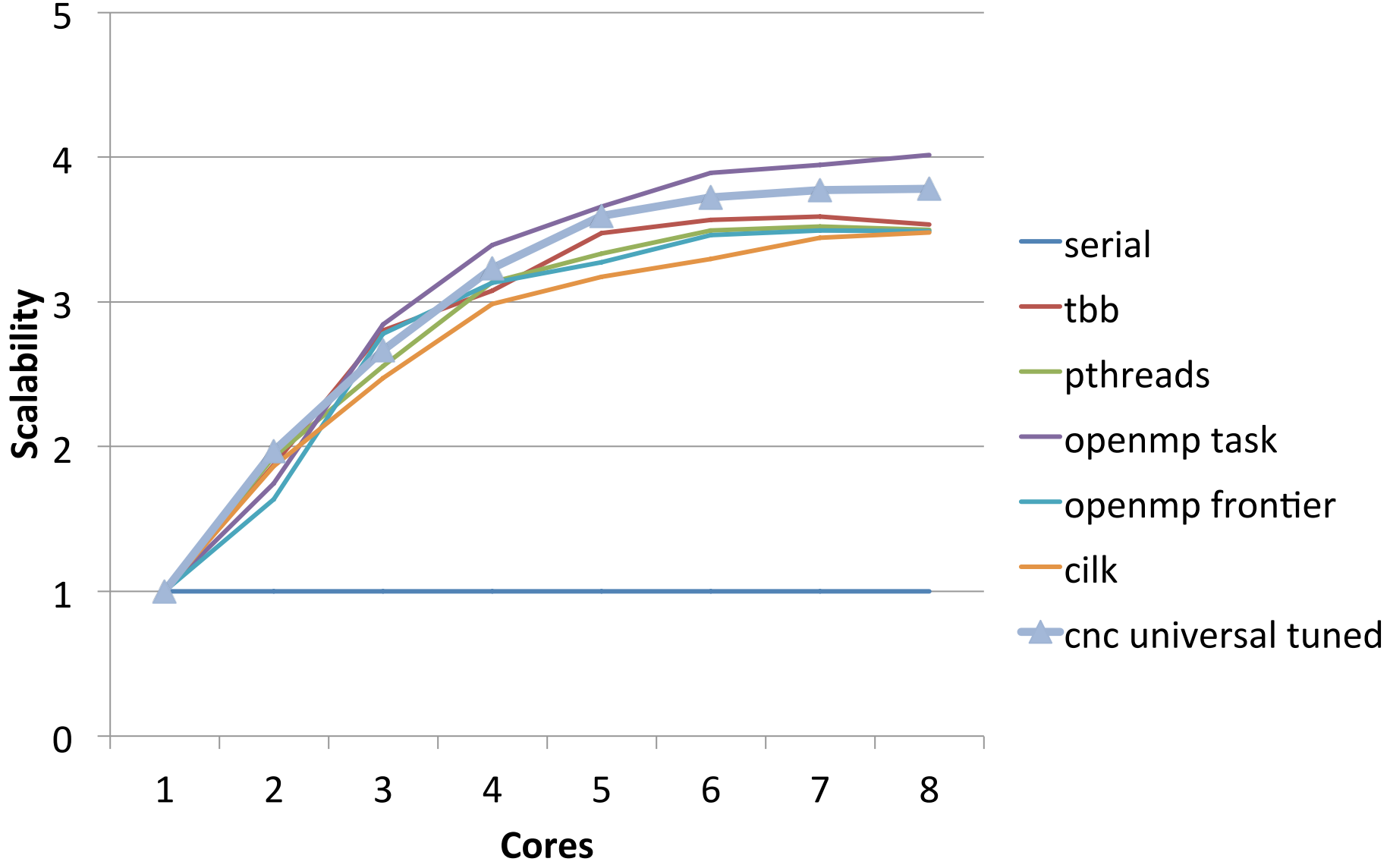
CnC: Dealing with the Overhead



Task Graph Execution Time As Tasks Are Increased
Jacobi Solver, 4 threads, matrix pwtk




Scalability on 4 core Core i7 920
13.2 GB/s Stream Triad BW
Scalability
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
Cores
serial
tbb
pthreads
openmp task
openmp frontier
cilk
cnc universal tuned


Scalability on 8 core Harpertown
4.02 GB/sec Stream Triad BW
Scalability
0
1
2
3
4
5
1
2
3
4
5
6
7
8
Cores
serial
tbb
pthreads
openmp task
openmp frontier
cilk
cnc universal tuned

Issues

- The CnC scheduler statistics when using depends() do not appear to be correct
- TBB graph model does not seem to correctly set the number of threads

Conclusions & Future Directions

- With simple tuning, CnC task graph execution is competitive with other common parallel programming models
- Overhead is acceptable for these scientific algorithms
- Looking at inter-tile affinity to further improve locality

Questions