Exceptional service in the national interest

Sandia National Laboratories

A tale of two schedulers

Noah Evans, Richard Barrett, Stephen Olivier, George Stelle
nevans@sandia.gov
6/26/17

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

Outline

- Making Parallel Programming Easier
- Qthreads Chapel Support
- The two Qthreads schedulers (plus the old one)
  - Sherwood
  - Nemesis
  - Distrib
- Performance evaluation
- Future work
- Conclusions

Making Parallel programming easier

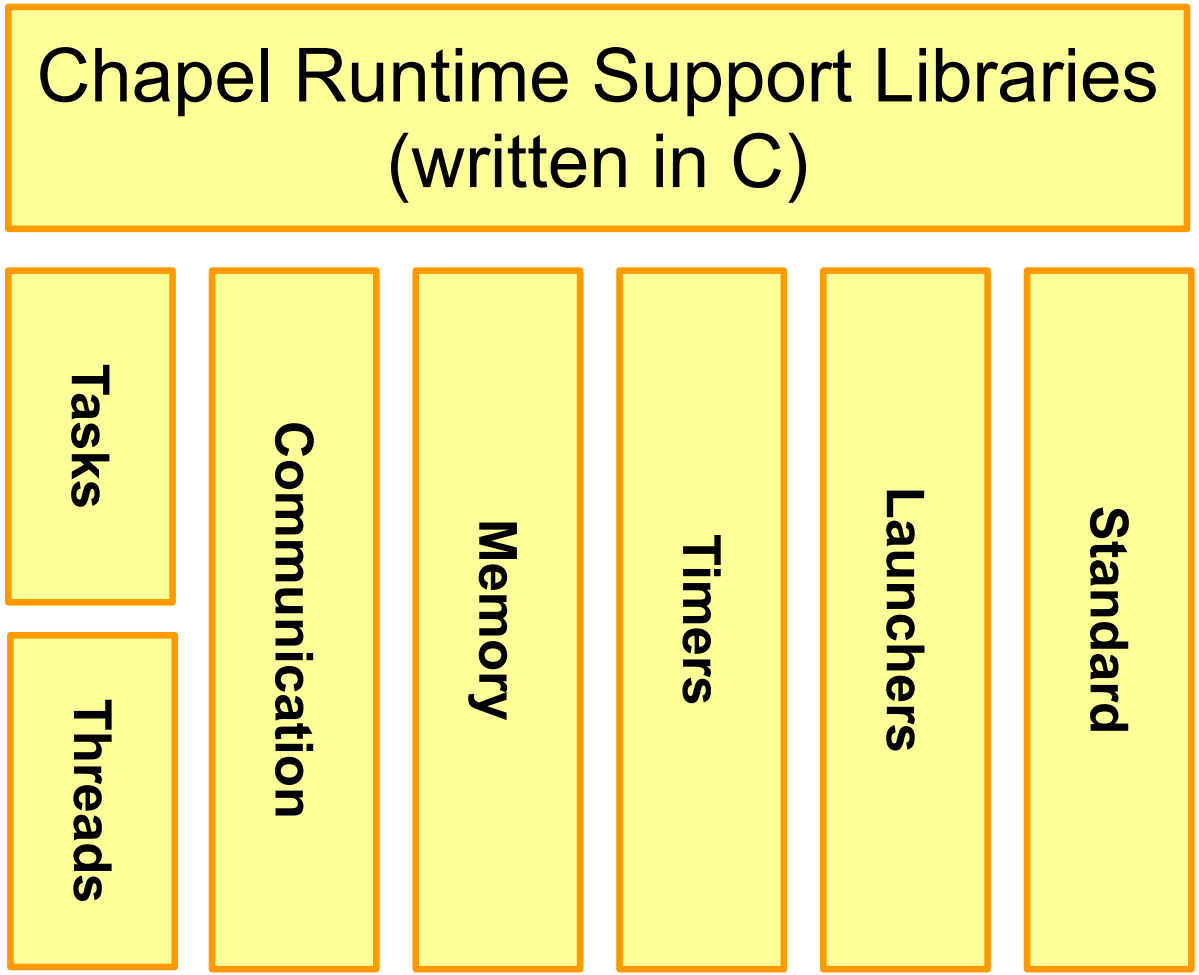
Sandia National Laboratories

- Typical Parallel Programming: MPI and BSP
  - Downside: fiddly, lots of application programming effort
- Another Strategy: Push complexity of parallel programs into the runtime
- Programmer specifies data dependencies and smallest units of work.
- This is the approach taken by the HPCS language Chapel

Solution: Multiresolution

- Ability to change underlying aspects of language
- Write one program, compile in different ways based on environment variables
- Choose abstraction at compile time rather than in the code.
- Goal: enable performance portability, reduce programmer effort

Chapel structure



Qthreads Chapel Support

Sandia National Laboratories

- Qthreads
  - user level tasking model
  - low level, anonymous threads, no signal handling cooperative.
  - lighter than pthreads
- Distinguishing feature Full Empty Bits (FEBs)
  - models the Cray XMT FEB, primitives can be in hardware or software
- Default for Chapel
- Qthreads tasking model is also multiresolution, can choose schedulers at configure time

Objective: Qthreads scheduler for many-core chapel

Sandia National Laboratories

- Our old default scheduler built for NUMA multicore machines using mutexes. Our mutex based schedulers don't scale for many-core.
- We've been working on schedulers to use lock-free methods and different scheduling strategies for many-core.
- Evaluating two schedulers, Nemesis and a new scheduler distrib. Nemesis good for simple streaming tasks. Distrib is good for irregular jobs using work stealing.

Sherwood

- Original work stealing scheduler for Qthreads
- Idea was queue to optimize for NUMA multicore
  - Front: LIFO scheduling for cache locality
  - Back: Bulk transfer of stolen jobs between NUMA domains
  - Design: Mutex lock at both ends of double ended queue
- However, Looking at both ends of queue prevents lock free approaches
- So good for older multicore, poor performance on manycore.

Nemesis

- Alternative to Sherwood
- Took an idea from MPICH2, the “Nemesis” lock free queue [Buntinas et al, 2006]
- Scheduling is simple FIFO, no load balancing
- Optimized for performance of streaming jobs
- No concept of work stealing or load balancing
- spin based backoff mechanism

Newest Distrib

- Take advantage of lessons learned from Nemesis, but take advantage of work stealing
- Minimize cache contention by spreading queues across cache lines
- LIFO scheduling
- At the same time lightweight work stealing, steal one at a time using a predefined “steal ratio” of how many times to check the local queue, before attempting to steal from other queues
- If nothing to steal backoff after a certain number of iterations

Summary

Table 1: Qthreads schedulers

Scheduler	Queue	Workstealing	Performance
Sherwood	One per NUMA domain OR one per worker thread	Yes	Good for multicore with big caches
Nemesis	Only one per worker thread	No	Good for streaming and contended workloads
Distrib	Only one per worker thread	Yes	Good for contended workloads with imbalance

Performance Evaluation

- Want to see how much overhead using LIFO scheduling and our minimal work stealing contributes vs Nemesis
- Also use Sherwood as a baseline
- Questions to answer:
  - What is the overhead of work stealing?
  - How much does backoff matter?
  - When should we use Nemesis and when should we use Distrib?

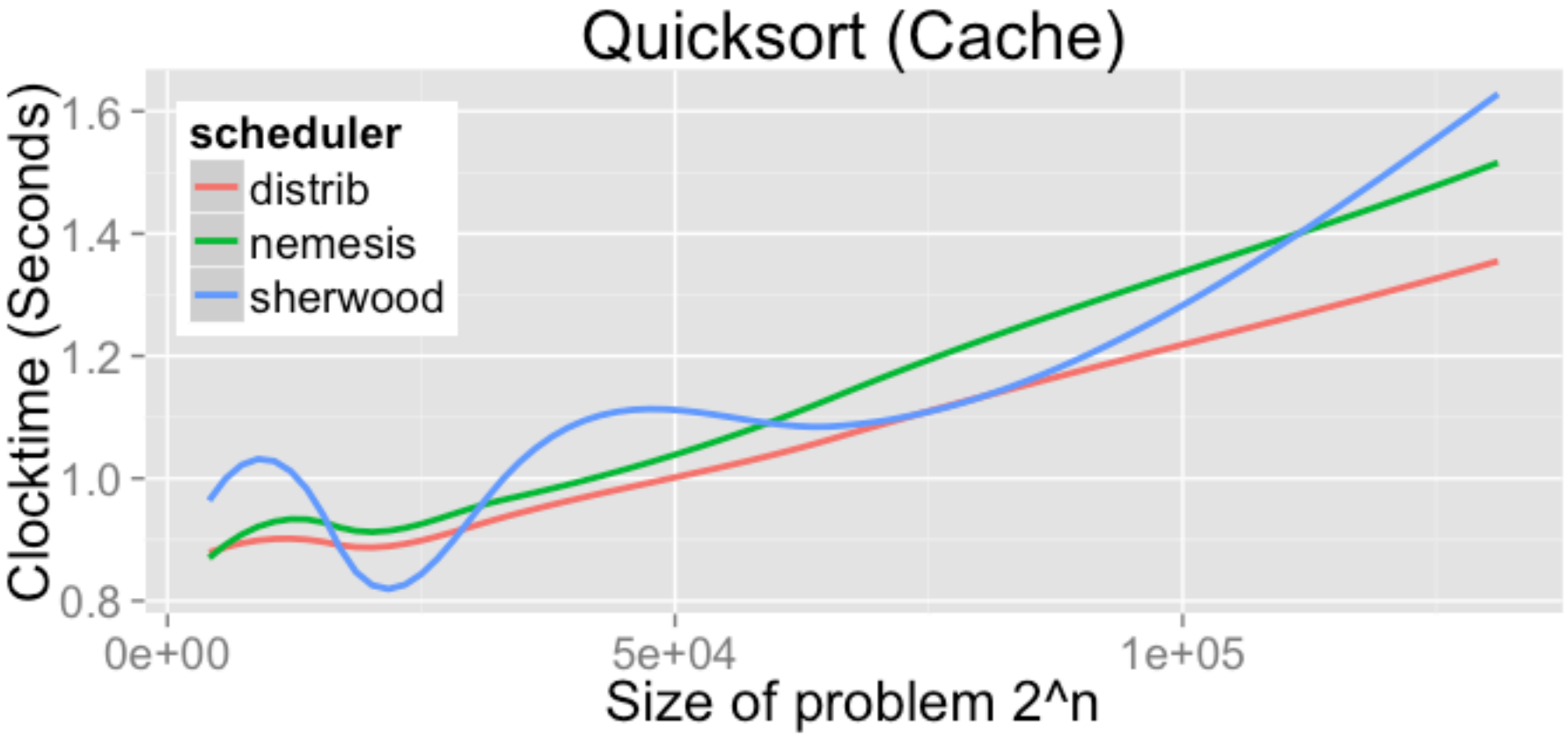
Experimental Design

- Knights Landing Processor 7250
  - 68 cores, 272 hardware threads, 1.6 GHz.
  - 16GB of high bandwidth memory (MC-DRAM) on package
  - operate in cache mode.
- Chapel 1.14, GCC version 4.8.3 using -O3 and -march=native
- Performance comparisons using Linux's perftools suite for full system profiling

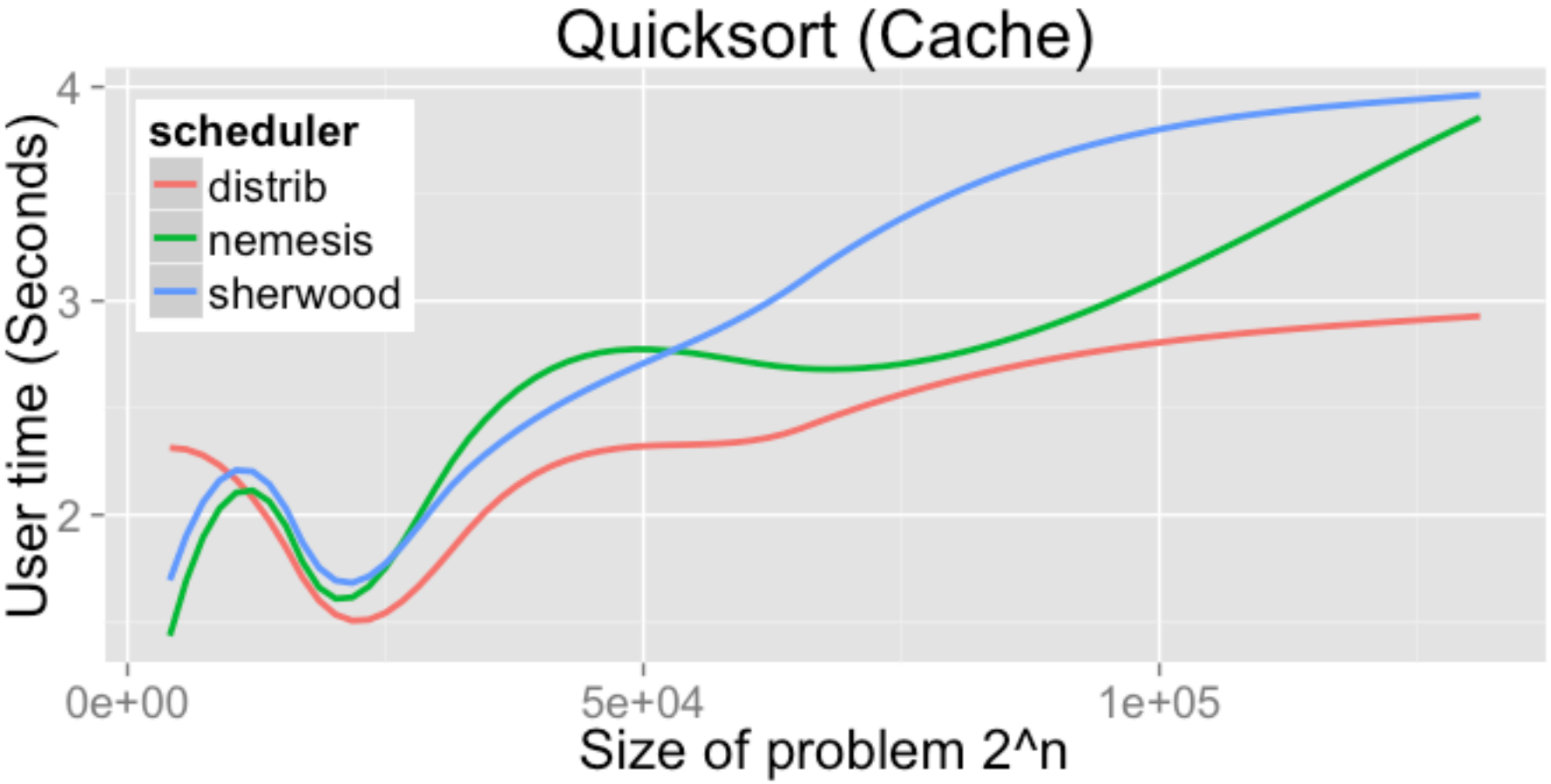
Benchmark overview

- Quicksort: simple distributed quick sort
- HPCS Scalable Synthetic Compact Applications graph analysis (SSCA#2)
- Stream: memory streaming benchmark
- Tree: constructs and sums a binary tree in parallel
- Graph500: two benchmarks, breadth first search (BFS) and shortest path, chapel only does BFS

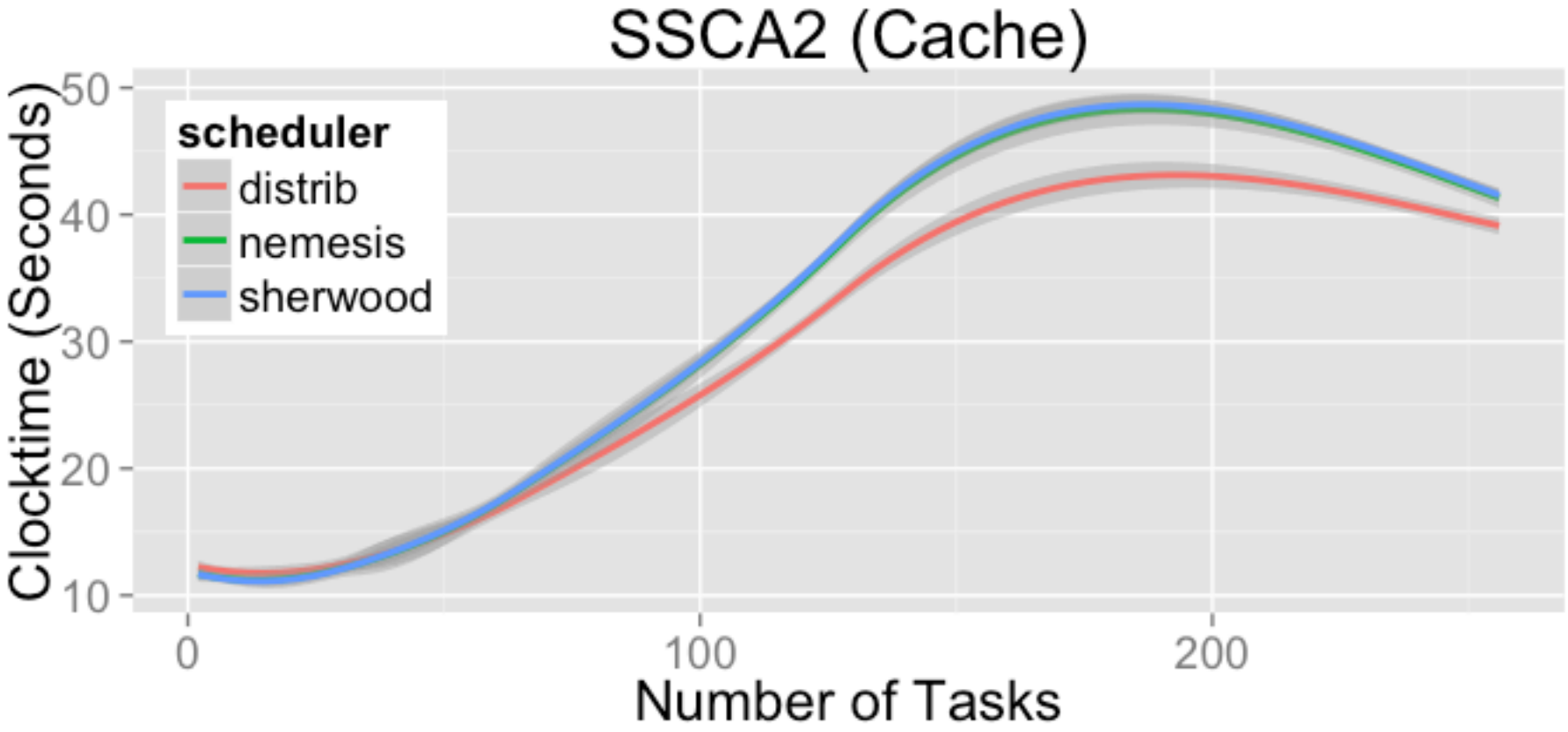
Quicksort: distrib load balancing better (lower is better)



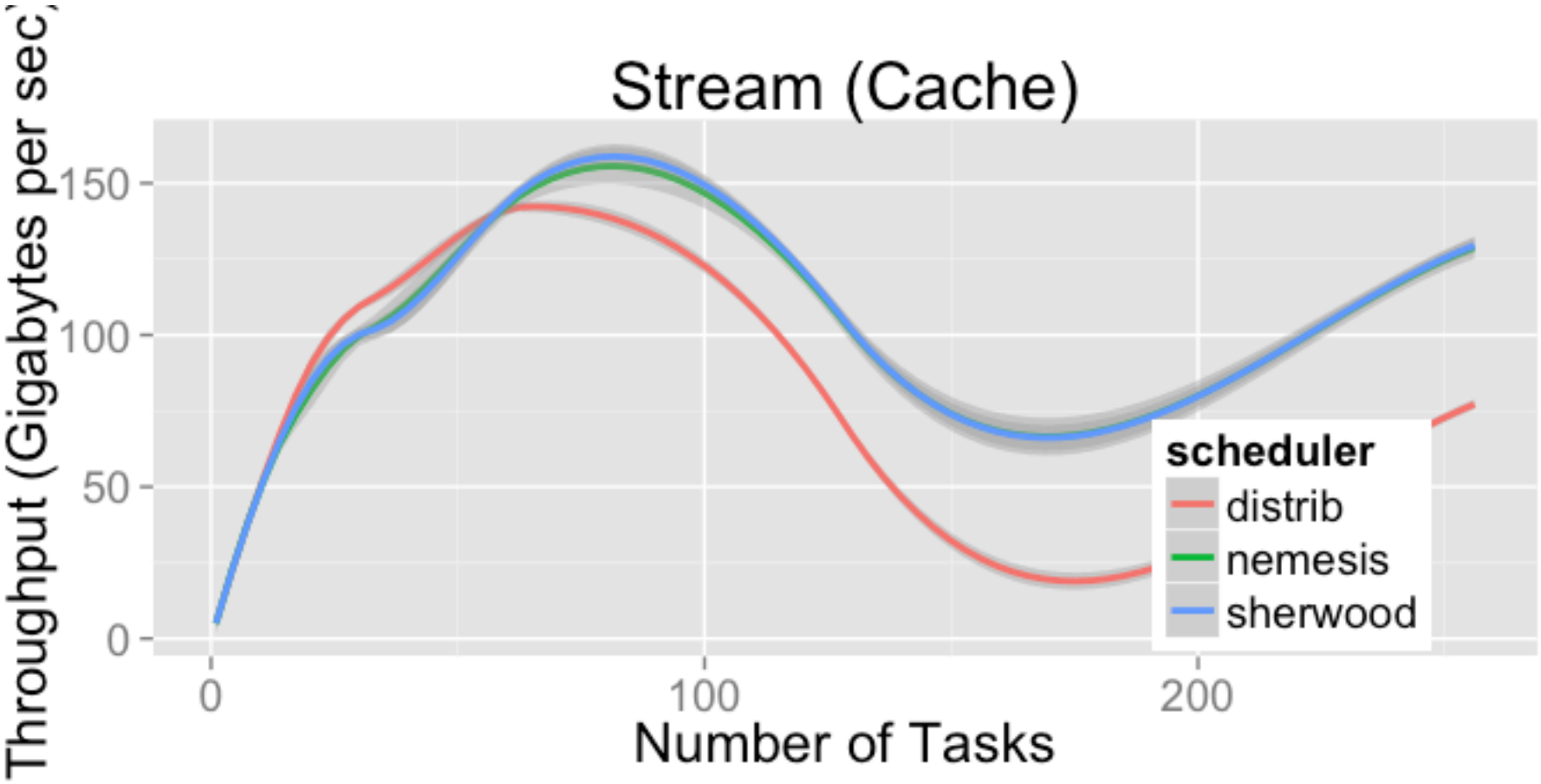
Same amount of actual work done, just better distributed



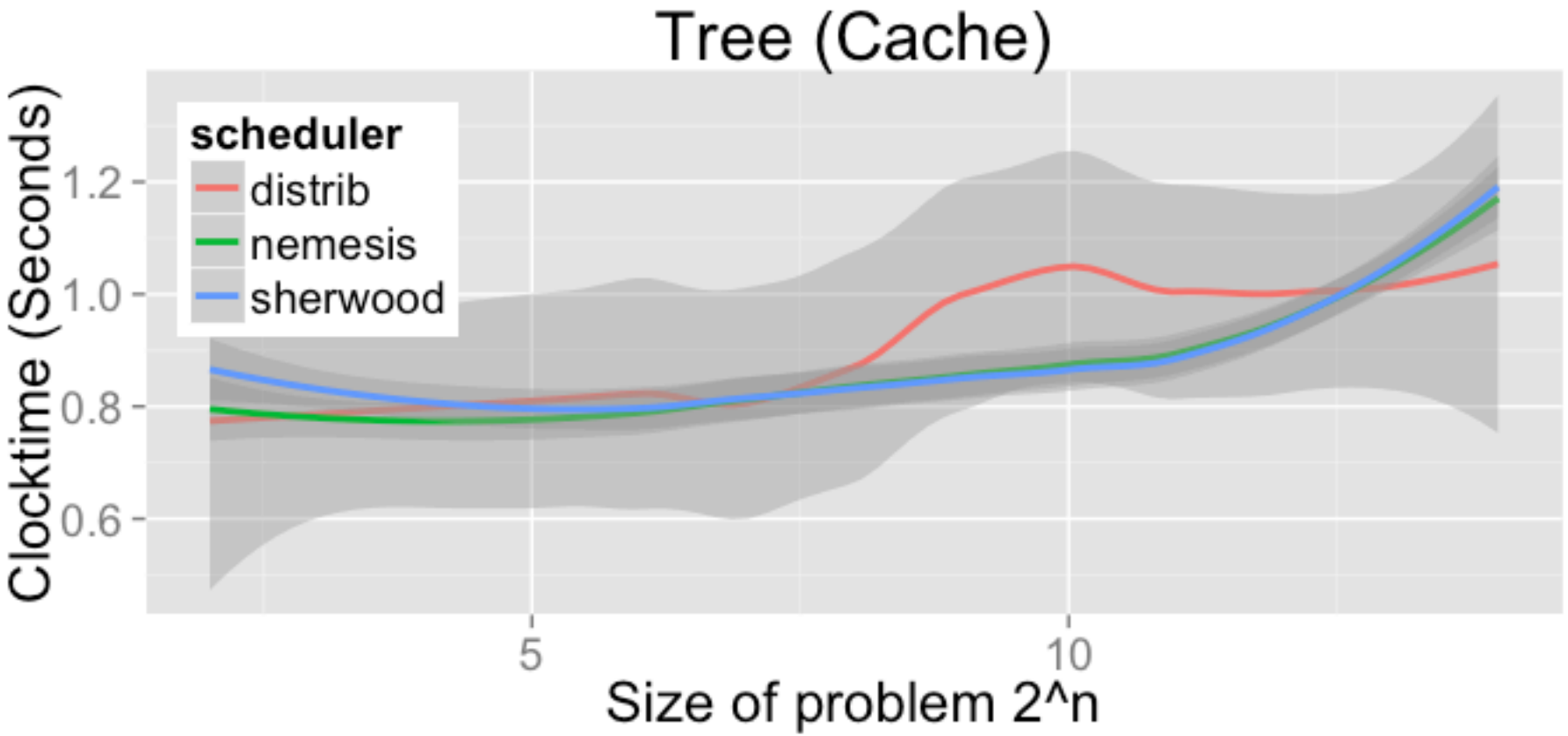
Distrib better for SSCA2 (lower is better)



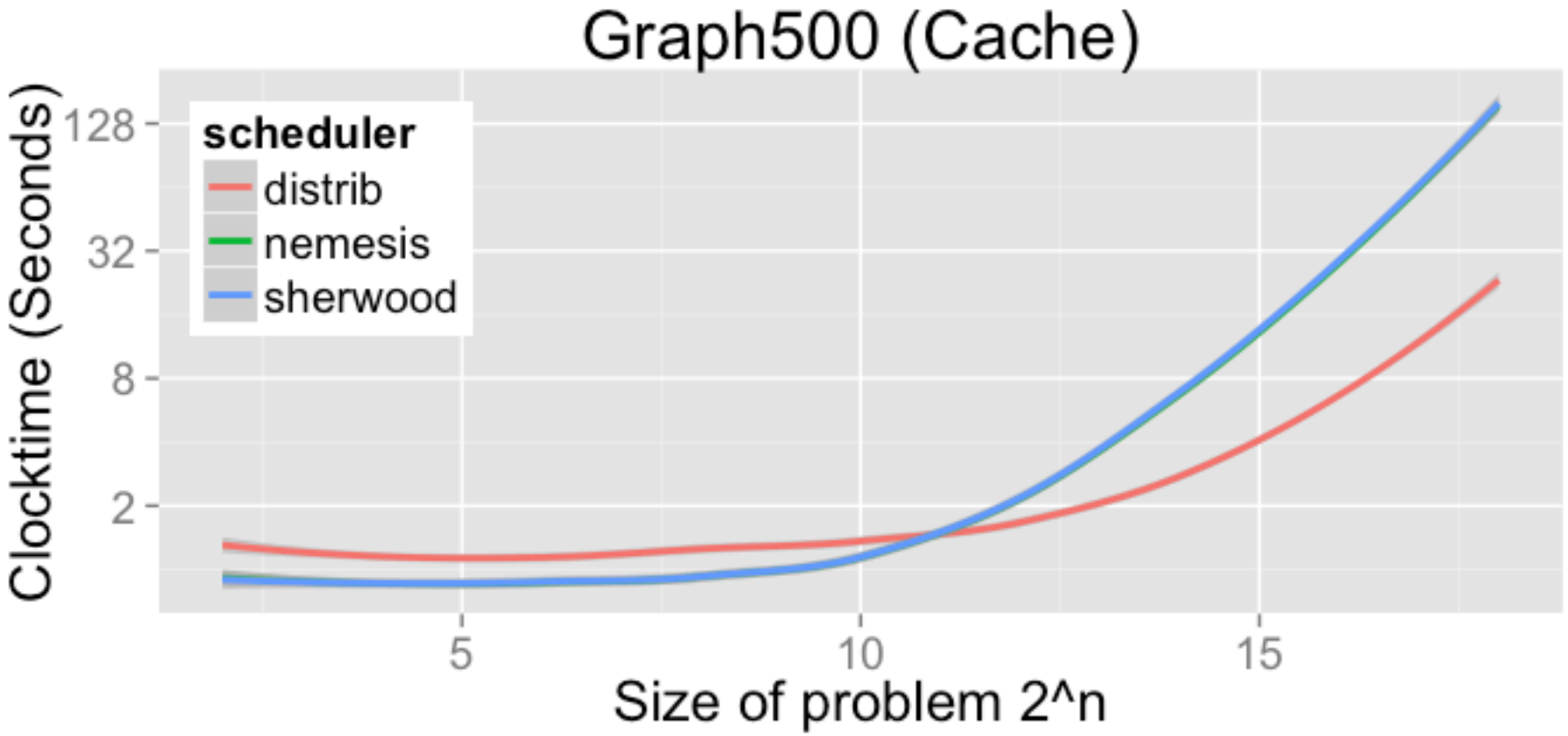
Nemesis FIFO better for Stream (higher is better)



Tree: Distrib better at scale



Distrib better for graph500 (lower is better)



Experimental conclusions

- Distrib is better for most cases at scale
- Lightweight workstealing major reason
- Overhead makes it slower for small problems
- Nemesis is still better for streaming jobs with simple workflows

Future work

- All application progress threads in Qthreads
  - (eg. MPI and Openfabrics asynchronous network threads)
  - Right now nemesis and distrib have a backoff to make time for progress threads
  - If all components of app use runtime, no need to backoff
- Is it possible to make distrib perform better than Nemesis in *all* cases?
  - Make work stealing zero cost (turn off w/ no overhead)
  - Switch LIFO/FIFO
- Dynamic schedulers?

Conclusions

- For most HPC use cases distrib is better
- For heavy streaming nemesis is more performant
- Can choose best tool for best job, fitting into Chapel's multi resolution approach
- Helps solving a wide variety of HPC problems

Sandia National Laboratories

Thank You