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

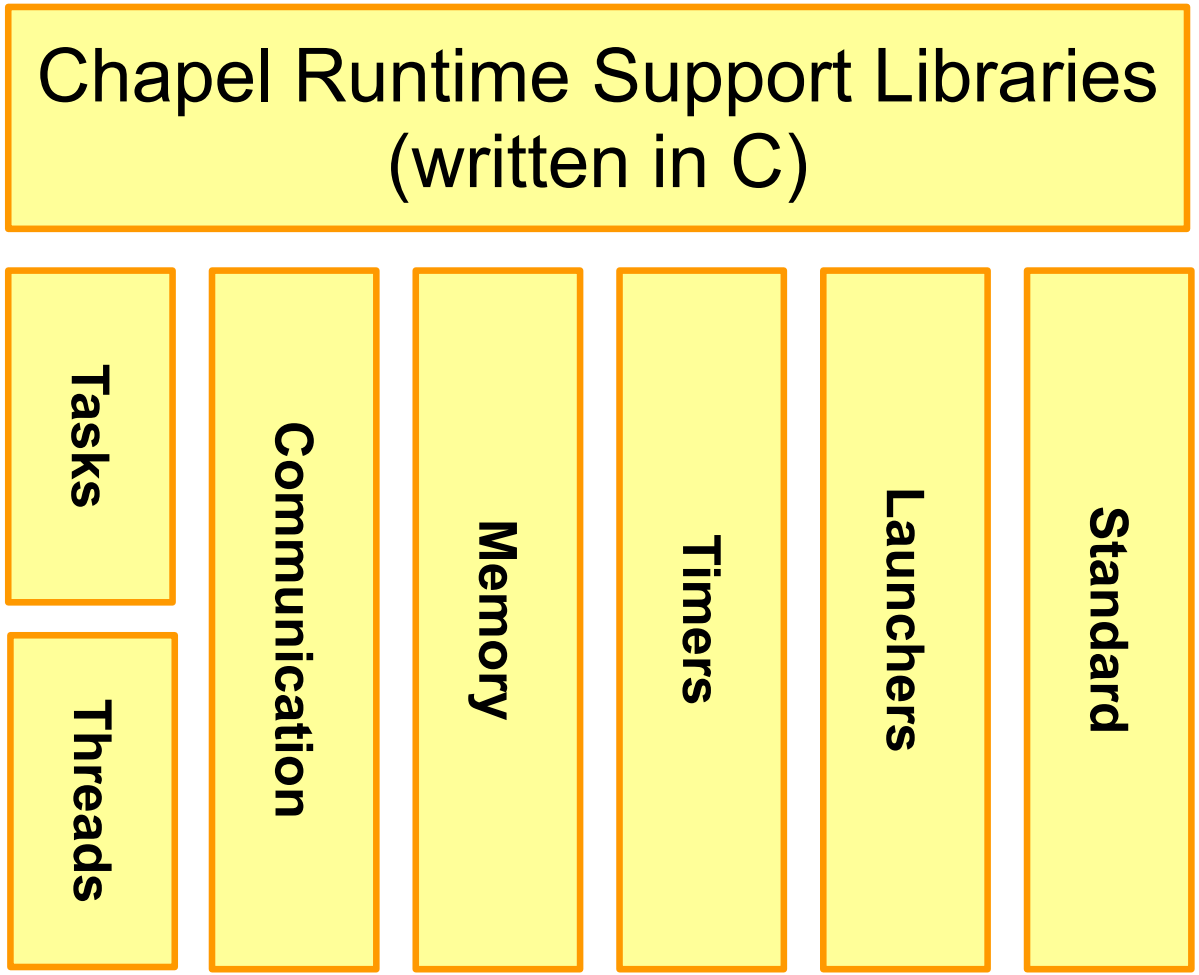
Sandia National Laboratories

- 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
- 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

Objective: scheduler for many-core

- Our old default scheduler built for NUMA multicore machines. Mutexes 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 distrib. Nemesis good for simple streaming tasks. Distrib is good for irregular jobs using work stealing.

Qthreads schedulers

- Qthreads schedulers are defined modularly
- Can change the scheduling behavior at configure time
- Dictate how jobs are added and removed from cooperative thread queue.

Sherwood

- Original work stealing scheduler for Qthreads
- Idea was to optimize for NUMA multicore
  - mutex locking
    - look at both ends of double ended queue
  - LIFO scheduling for cache locality
- Work stealing bulk between NUMA domains
- 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 (cite)
- Scheduling is simple FIFO, no load balancing
- Optimized for performance of streaming jobs
- No concept of work stealing or load balancing

don’t call it new, call it alternative

Newest Distrib

- Take advantage of lessons learned from Nemesis, but take advantage of work stealing
- Use Nemesis-style lock free queue
- 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

Add slide here summarizing the different schedulers — maybe there was a table in the paper?

Define stealing ratio

Summary

Table 1: Qthreads schedulers

Scheduler	Queu	Workstealing
Sherwood	One one p	Yes
Nemesis	Only	No
Distrib	Only one per worker thread	Yes

Add another column about which performs better where

Performance Evaluation

- Want to see how much overhead using LIFO scheduling and our minimal work stealing contributes
- Questions to answer:
  - What is the overhead of work stealing?
  - When should we use Nemesis and when should we use Distrib?

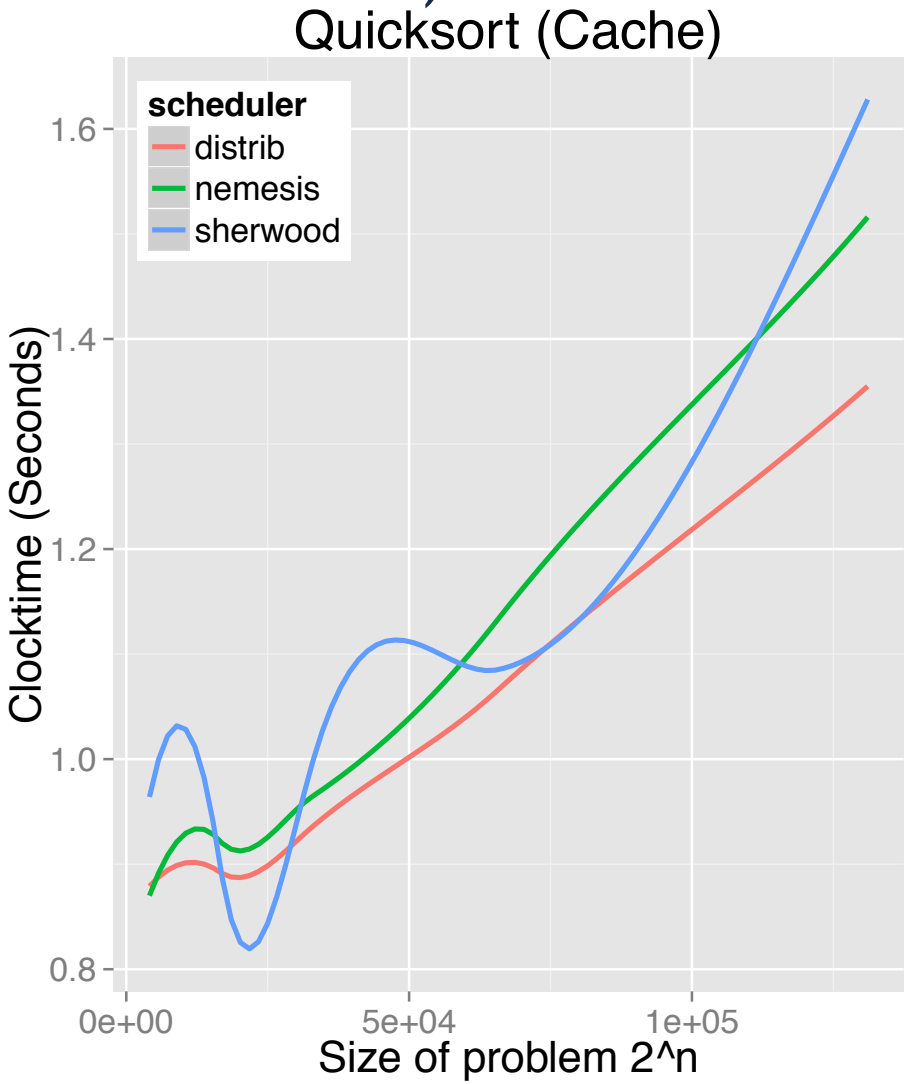
Experimental Design

- 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

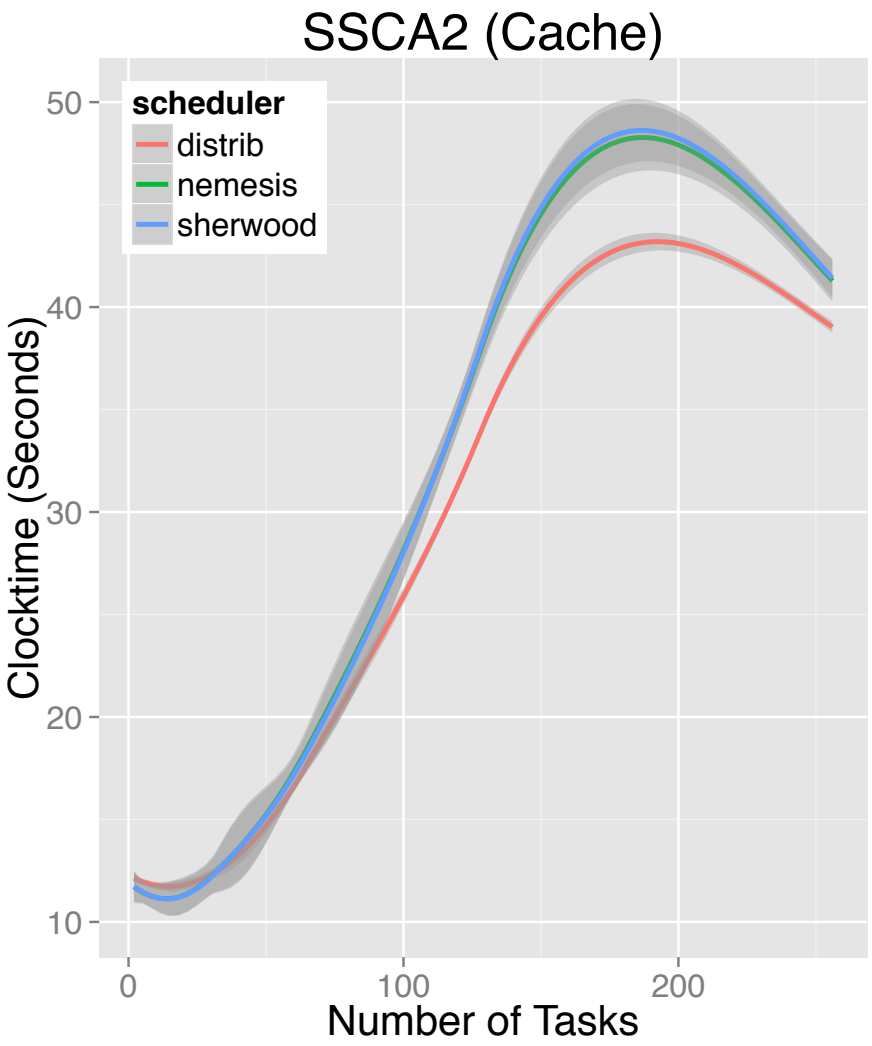
Benchmark overview

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

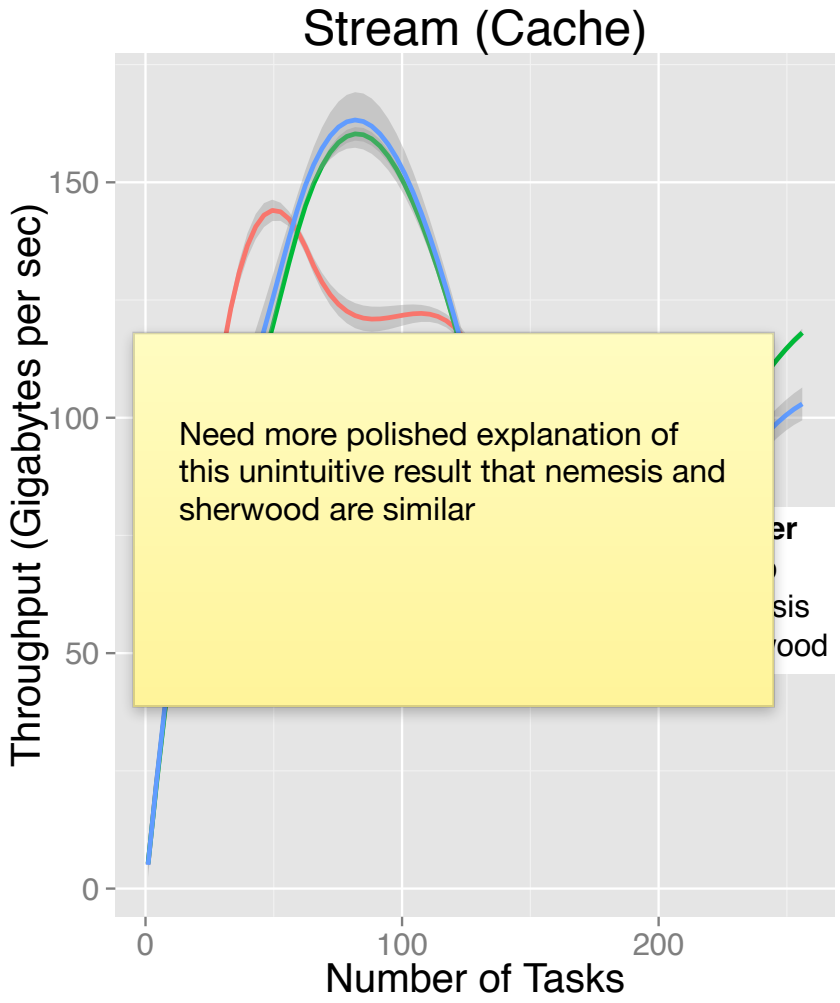
Quicksort: distrib load balancing better (lower is better)



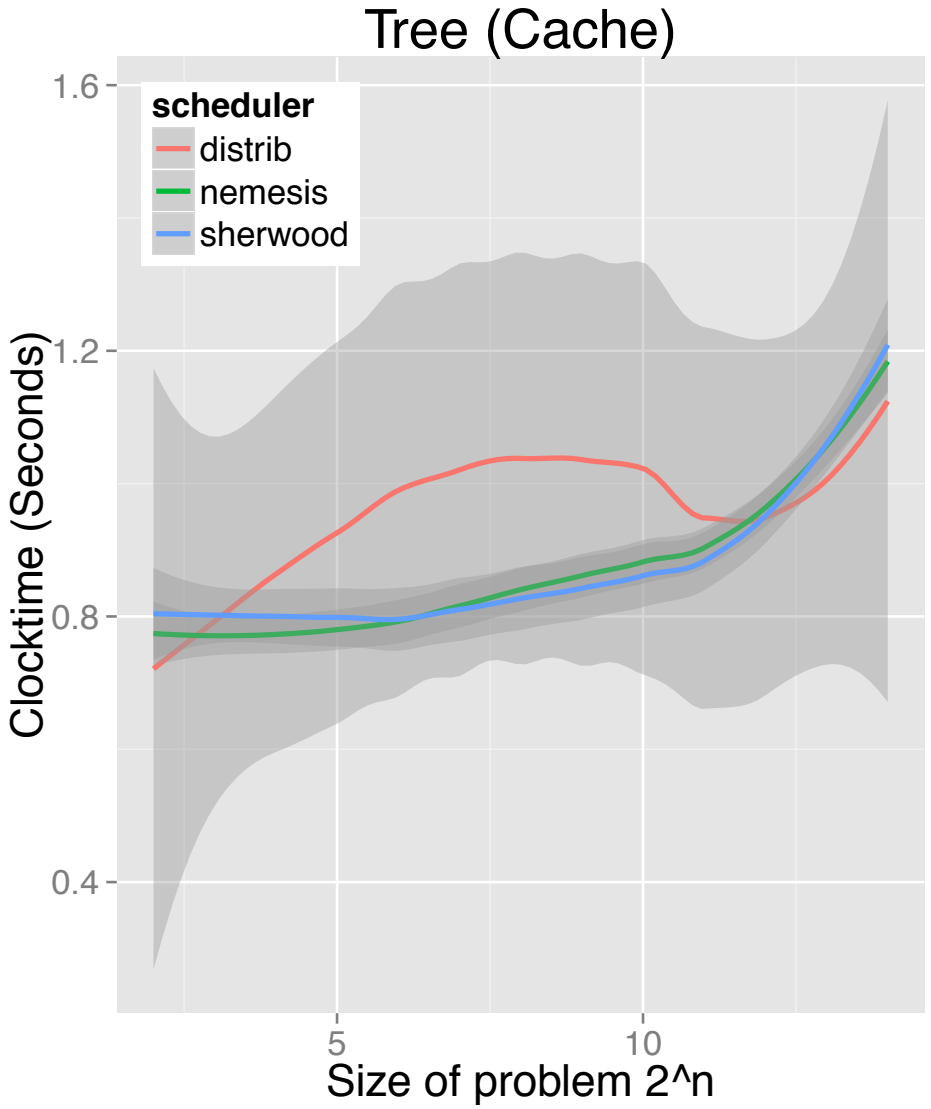
Distrib better for SSCA2 (lower is better)



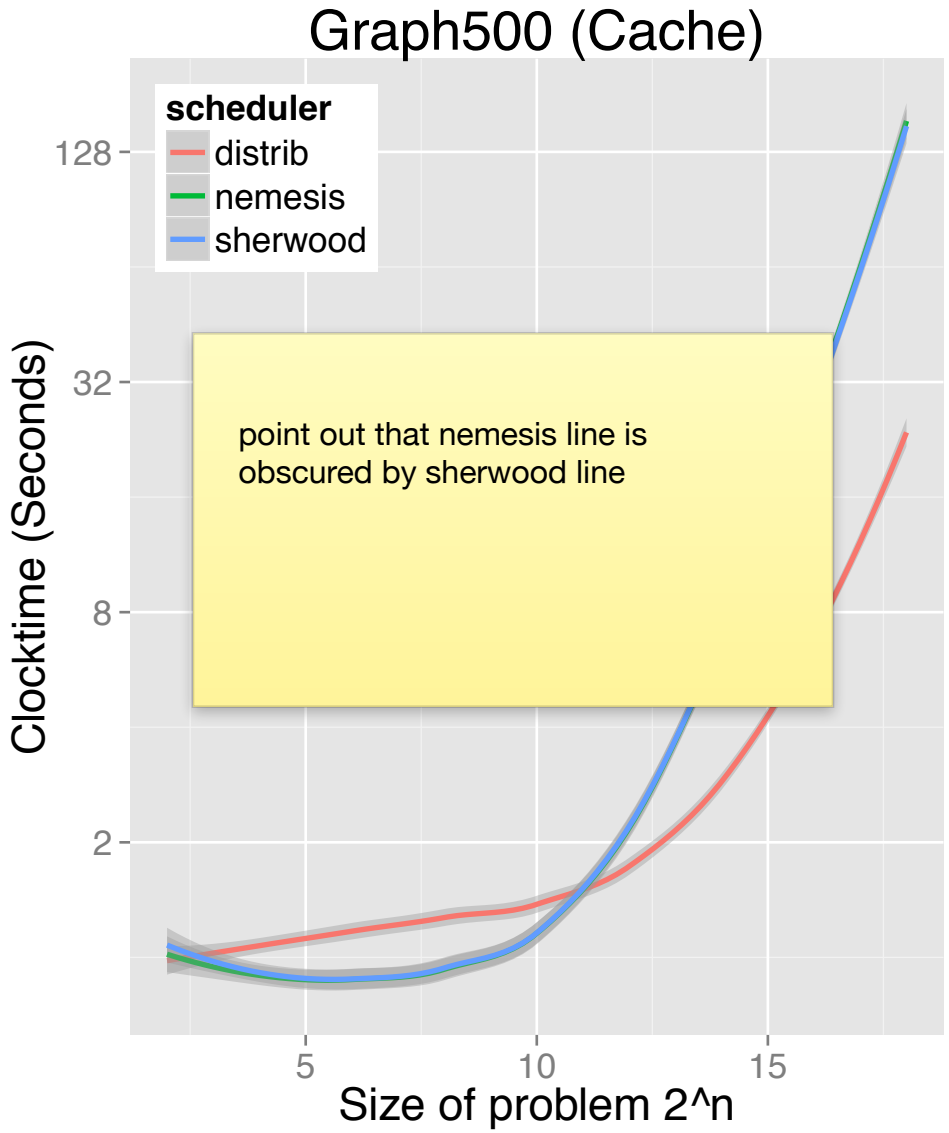
Nemesis FIFO better for Stream (higher is better)



Tree: Distrib better at scale



Distrib better for graph500 (lower is better)



Experimental conclusions

- Distrib is better for most cases at scale
- 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?

lithe like — explain

Conclusions

- For most 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