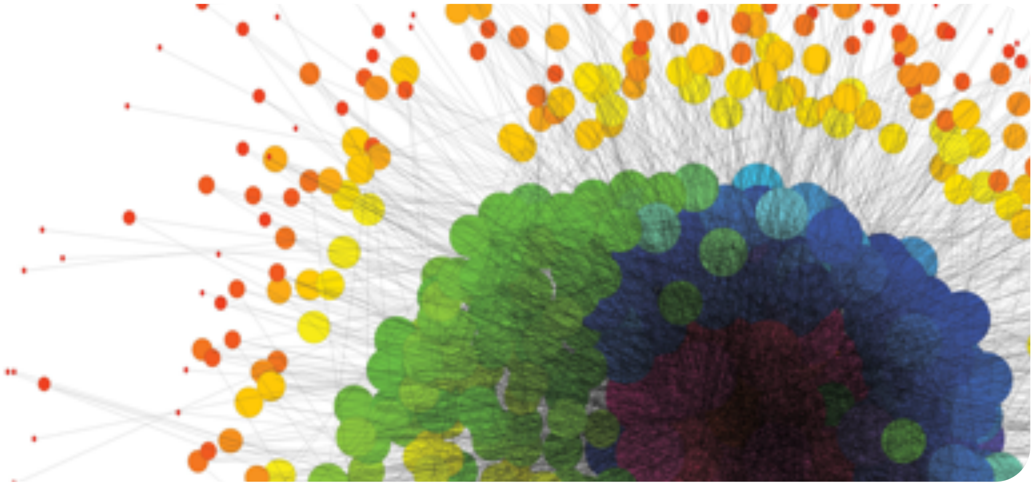
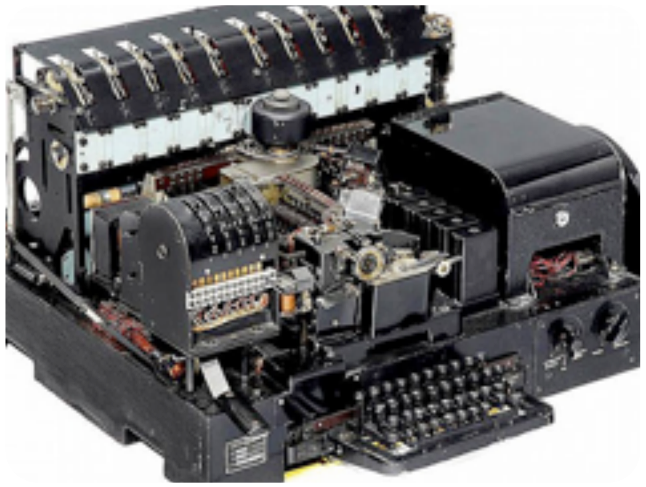
SAND2012-6985P

Exceptional service in the national interest

Sandia National Laboratories





Chapel, Qthreads, and Eurekas

Chapel Projects Review

August 23, 2012

U.S. DEPARTMENT OF ENERGY

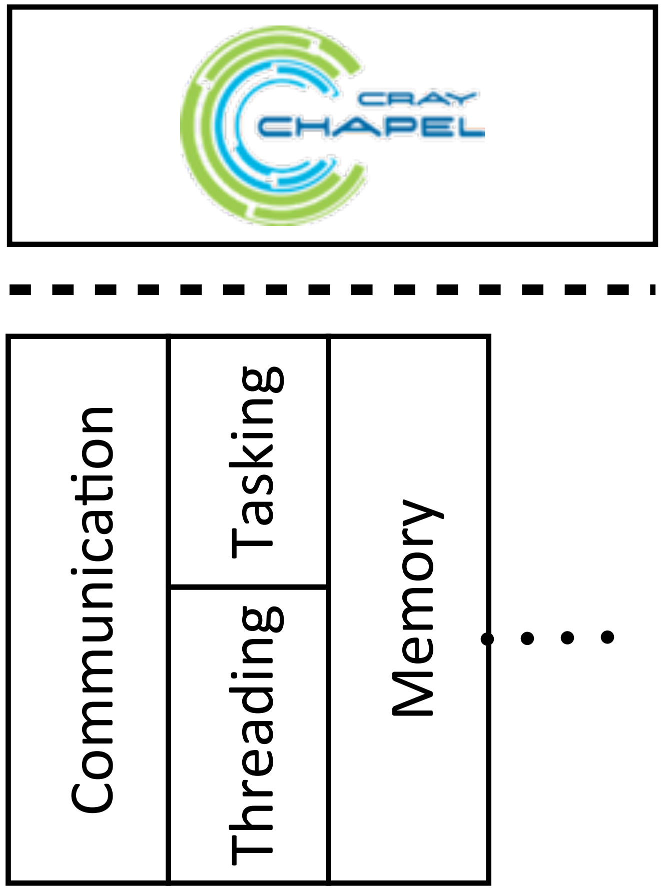
NNSA National Nuclear Security Administration

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.

Chapel's Runtime Layer

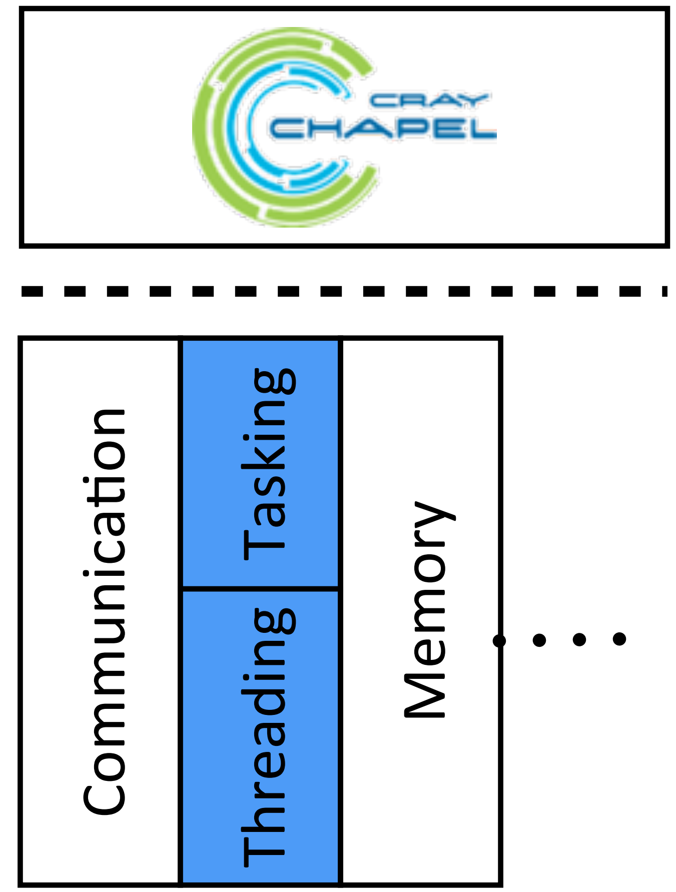
Sandia National Laboratories

- Abstract interface compiler target
  - C API calls
  - Implementations selected when building `chpl`
- Default implementations
  - Tasking/threading: FIFO/Pthreads
  - Communication: GASNet
- Tasking interface supports
  - Sync variables
  - Begin, cobegin, coforall, ...



Challenges in Tasking Runtimes

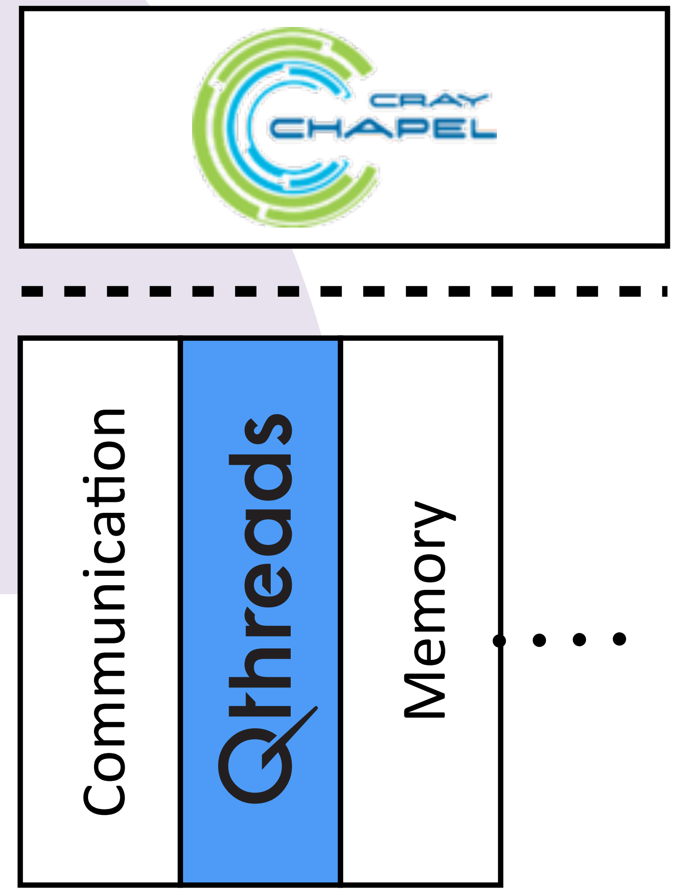
- Per-thread state
- Locality
  - An afterthought in standard threading models
  - Communication and synchronization expensive (easy to use accidentally)
- Synchronization
  - Hard to make portable, maintain guarantees
- Scheduling
  - Part of the critical path
  - Controls resource contention (cache, memory, network, etc.)
  - Adaptivity to load, power, resource contention critical for performance
  - Most schedulers ignore non-computational activity
    - Networking and I/O
- Every machine is different
  - Granularity of sharing (cacheline size)
  - Optimal number of threads (PU count)
  - Communication topology
  - Cache structure
  - Memory model



Sandia's Qthreads for Tasking

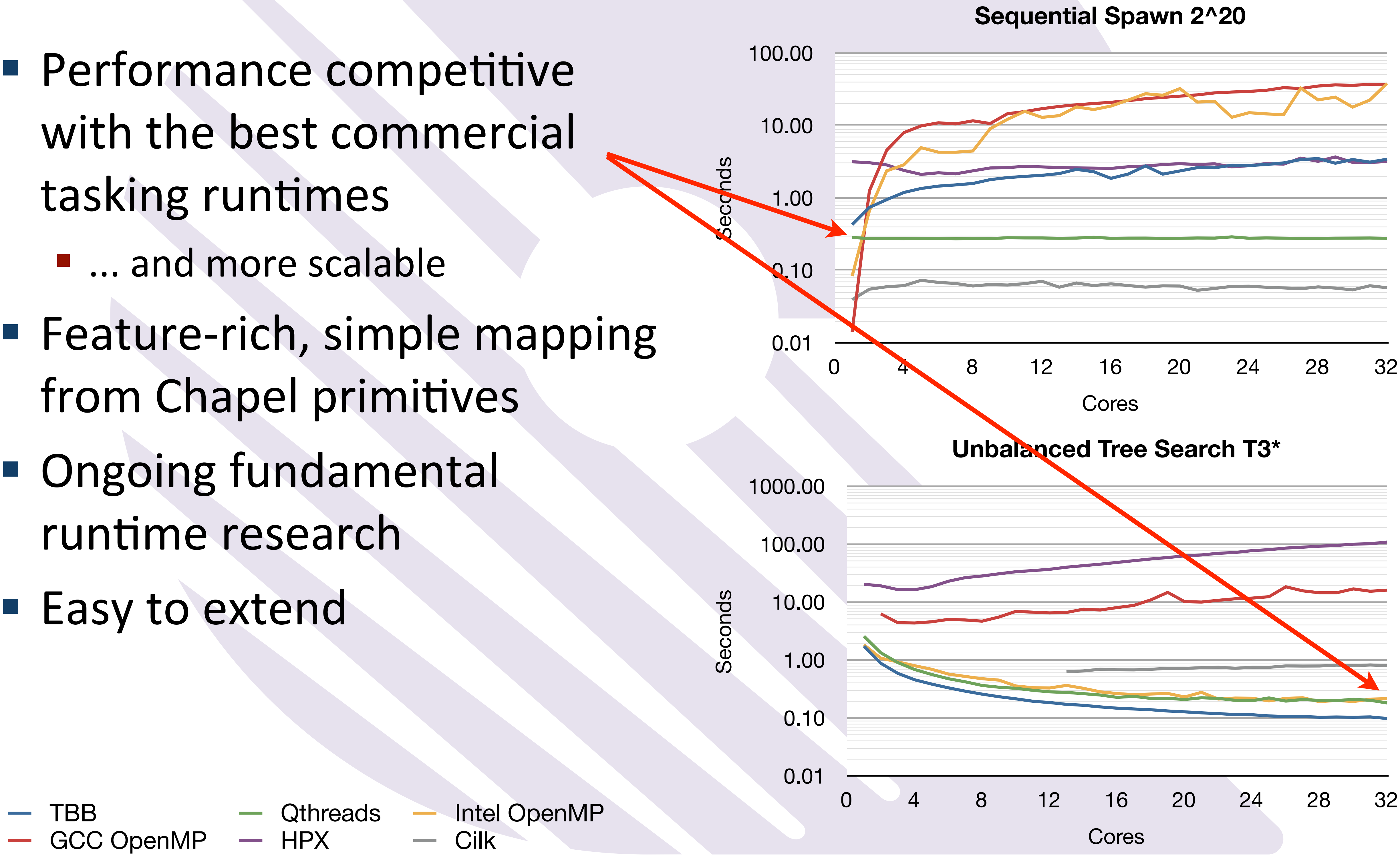
Sandia National Laboratories

- Lightweight user-level tasking
- Platform portability
  - ARM, Tilera, IA32/64, AMD64, PPC32/64, SparcV9
  - Linux, BSD, Solaris, MacOSX, Cygwin
- Locality fundamental to model
  - "Shepherd" a thread mobility domain
- Fine-grained synchronization semantics
  - Full-empty bits (64-bit & 60-bit)
  - Mutexes
  - Atomic operations (integer incr, float incr, & CAS)
  - Collective and reduction operations (sincs)
- Locality-, cache-, I/O-aware work-stealing scheduler model
- Open source research platform



Why Qthreads?

- Performance competitive with the best commercial tasking runtimes
  - ... and more scalable
- Feature-rich, simple mapping from Chapel primitives
- Ongoing fundamental runtime research
- Easy to extend



*T3 dataset generates 4112897 vertices

Recent Chapel Support Overview

- Regression Tests
  - Local Nightlies!
- Synchronization Improvements
  - More direct support of sync variables (Matt Baker)
  - Support for (more efficient) oversubscription
- I/O Subsystem
- Eureka Moment Infrastructure
  - Sincs
  - Task Teams & Subteams

I/O Subsystem Design

- Queue of I/O operations
- Servicing kernel threads (pthreads)
  - Dynamic up to user-configurable maximum
  - Persistent up to user-configurable time limit

- Overheads
  - 1-2 context swaps
  - Queueing latency
  - I/O threadstart

I/O Subsystem Features

- Generic Blocking Operations
  - Basis for fundamental blocking operations
  - Provides inter-operation with external blocking operations (TPLs)
  - 2 context swaps
  - Potential for abuse
- System Call Interception
  - 1 context swap
  - Hard and soft interception
  - Capabilities limited by OS support for `syscall()`
- Networking Operations
  - Collaboration with Portals4
  - Asynchronous operations needn't involve subsystem
  - Progress thread management an area of active research (see SPR)

Eureka Moments

Sandia National Laboratories

- Asynchronous preemptive termination of a set of tasks
  - Number of tasks working towards some goal
  - One task is first to reach special state, signals "eureka"
  - Only first task continues execution, all others terminated

- Use cases
  - Algorithm races
    - Multiple versions of the same kernel
    - Redundant execution
  - Recursive tree search algorithms
    - Many parallel tasks searching for a specific condition or datum
  - Parallel breaks
    - Break out of parallel loops

Eureka Requirements

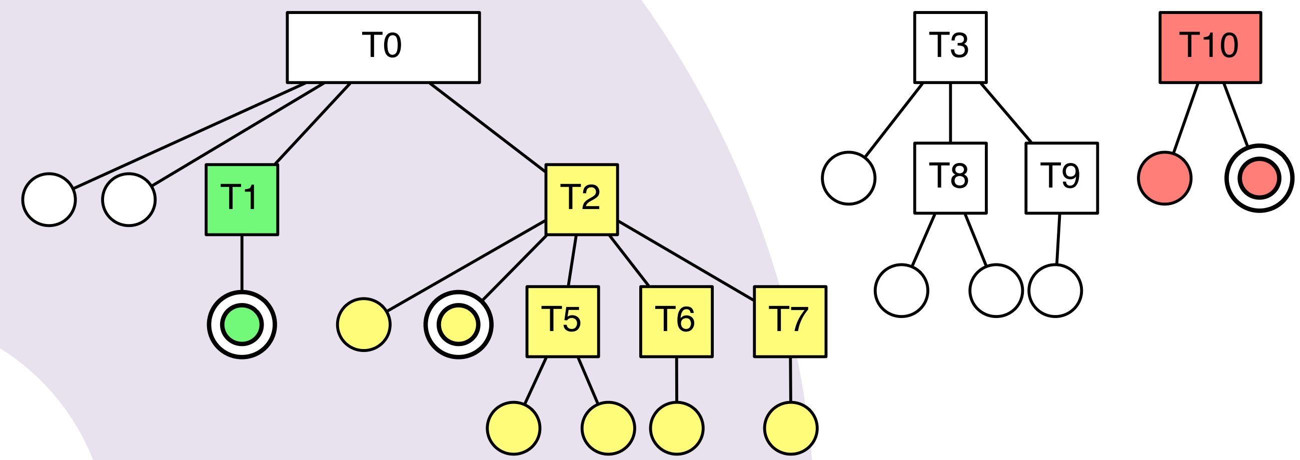
- Preemptive task kill
  - Stop running tasks
  - Filter work queues
  - Track down blocked tasks
- Task collections
  - Maintain membership
  - Scope extent of kill
  - Implemented in Qthreads as “teams”
  - Nested eurekas require nested teams (scope of death)

Task Kill Algorithms

- Option #1: Task-centric
  - Send termination signal to all tasks in team X
  - O(T) operation
  - Must maintain explicit membership list
  - Requires that tasks be able to receive and/or handle signals
    - Even when blocked!
- Option #2: Worker-centric
  - Send termination signal to all worker threads, who then collaborate to eliminate tasks matching some description
  - O(P) operation
  - Do not need explicit membership list
    - Allows simpler mostly-anonymous tasks
  - Tasks do not need to “handle” or “receive” signals

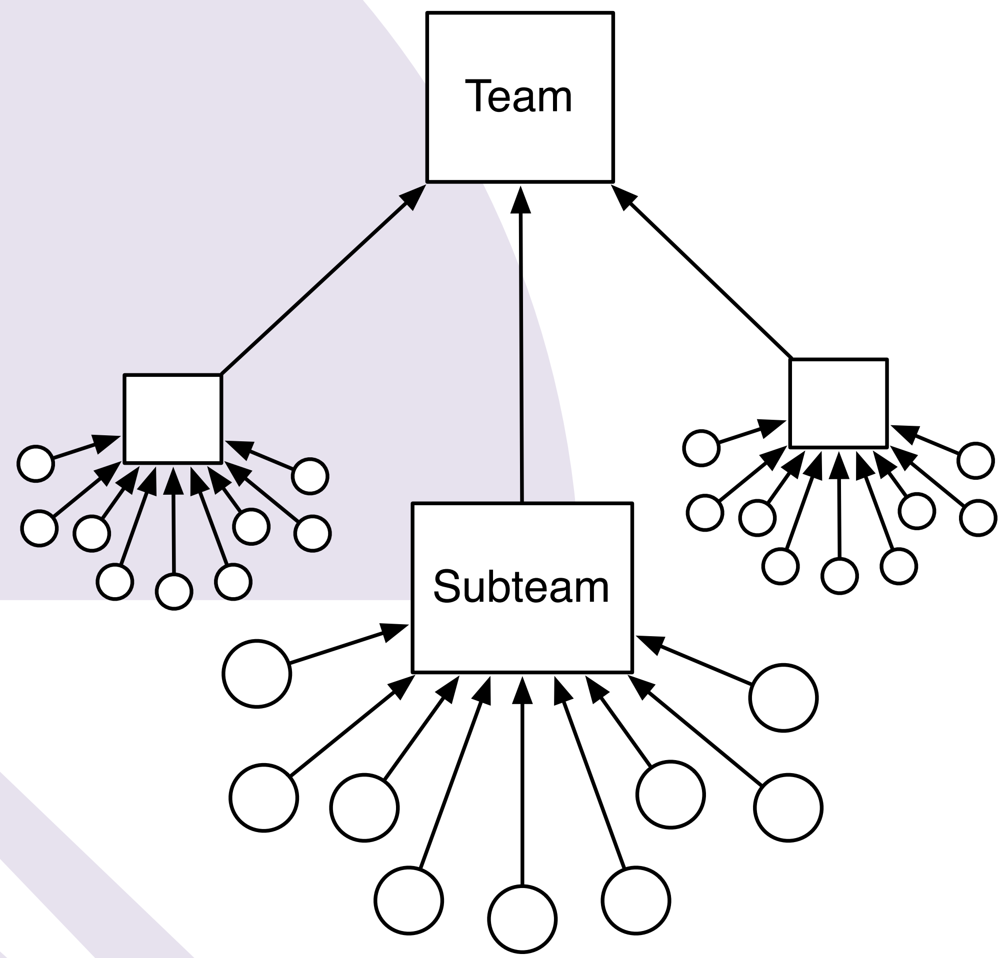
Teams Concept

- All tasks belong to a team
  - Team "0" always runs first task
- A task only belongs to one team
- New tasks can be spawned into:
  - Same team as parent
  - A new team
  - A new team dependent on the parent team's existence (subteam)
- An execution can comprise a forest of team trees
  - Dynamically growing and contracting
- A eureka event propagates down a team tree
  - Tree structure encodes dependence
  - Recursive cascading kill of all subteam tasks in parallel



Teams Implementation in Qthreads

- Each task has an associated team ID (pointer to team struct)
- Team struct
  - No list of references to members or subteams (SPEED)
  - Sincs for synchronizing collections of tasks and subteams
- Team “0” has no associated struct (tasks in “0” store NULL pointer)
  - Minimizes impact on Qthreads apps not using teams
- Subteams have special (invisible) “watcher” member thread
  - Trigger eureka iff the parent team is destroyed



Sinc Synchronization

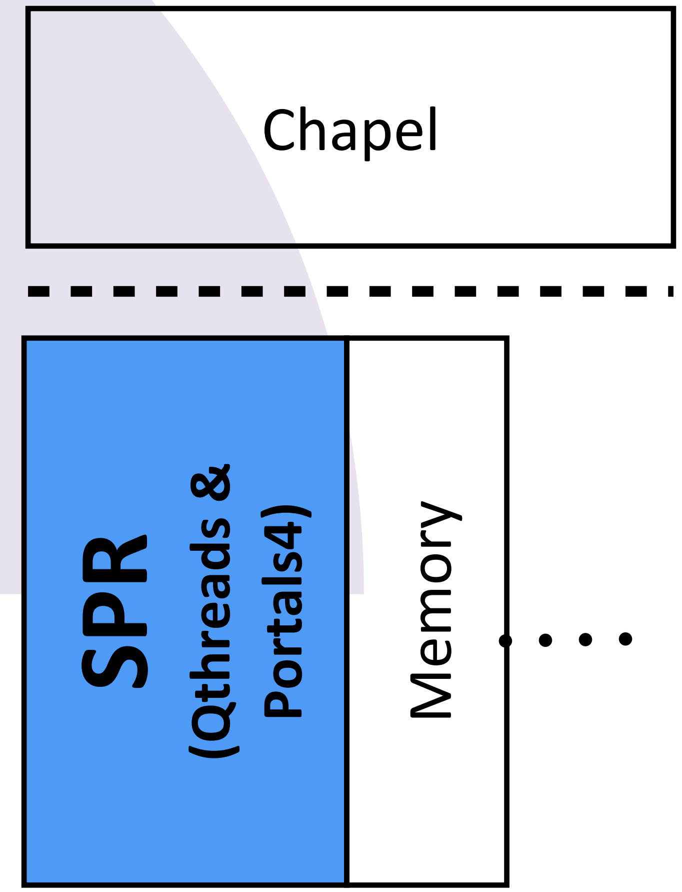
- Collective and reduction operations
  - Dynamic set of anonymous participants
  - Task barrier
  - User-provided reduction operations
    - Do not require synchronization!
- Basic usage
  - Create expecting N submissions (participants)
  - Increase/decrease participation with `qt_sinc_expect()` and `_submit()`
  - Tasks may block until sinc is "ready" with `qt_sinc_wait()`
- Multiple implementations
  - Central counter (both incr and CAS variants)
  - Distributed counter (snzi-style)

Status and the Future

- Status
  - Multiple implementations of sincs construct
    - Used in many places in Qthreads internals and applications
  - Teams and subteams fully implemented
  - Eureka design completed
- The Future
  - Multinode!
  - SPR: A more perfect union of parallelism scopes
  - Distributed task teams

Scalable Parallel Runtime (SPR)

- Integrate Qthreads and Portals for mutual benefit
- Remote task spawn
  - Currently explicit, potential for load balance under certain conditions
  - Continuation-style programming
- Data movement and collectives
  - Can attach (input) data to tasks OR send data directly (RMA-style)
  - Can leverage MPI collectives
- Synchronization
  - Currently done via remote spawn, plan to do better (as needed)
- Progress
  - Portals4 provides strong (asynchronous) progress
  - Currently multiple progress threads (needs development)



Thank you!

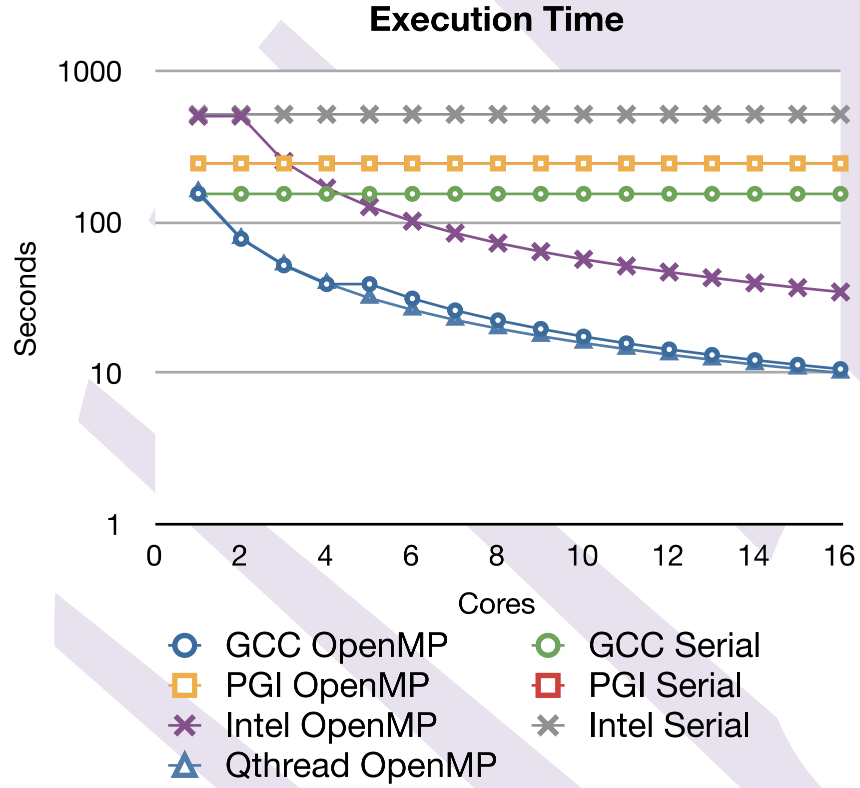
QUESTIONS?

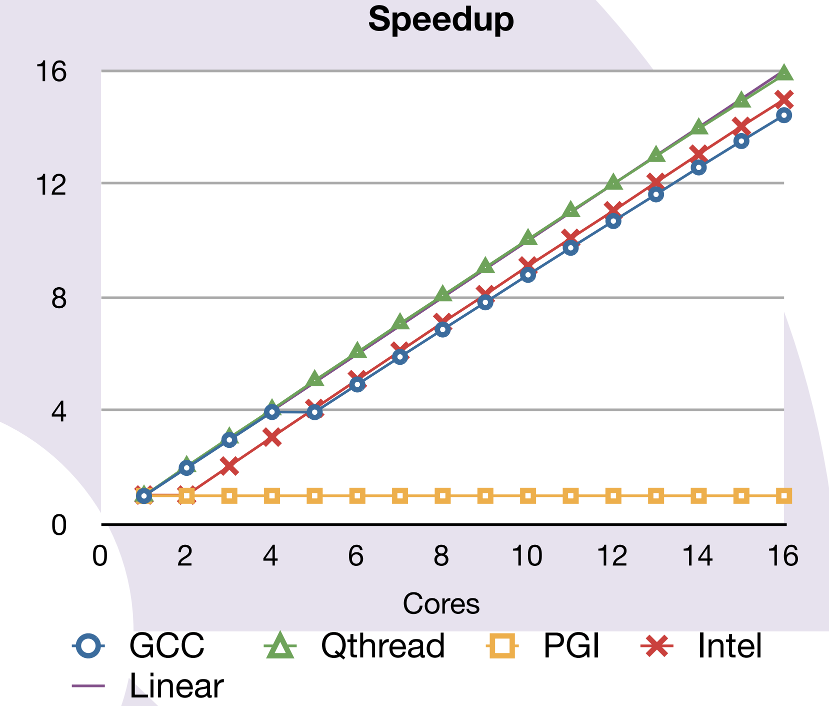
End of presentation...

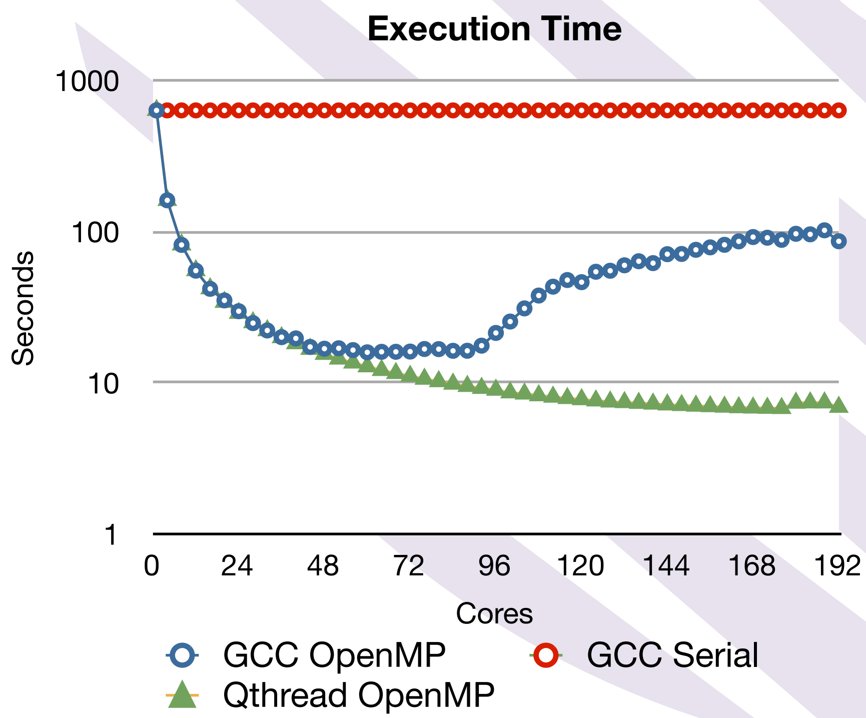
SPARE SLIDES

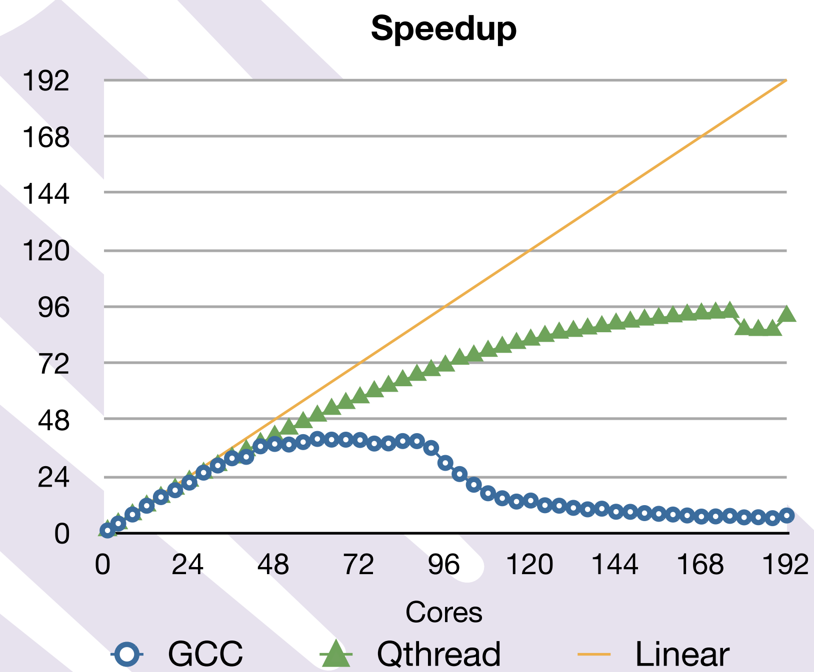
Implementing Other Models

OpenMP SparseLU Factorization (BOTS)









The Others in the Field

	SPR	Cilk	TBB	IOMP	GOMP	HPX	Cuda	Nanox	Tascel	Scioto	H-C	H-J
Loop Parallelism	✔	✔	✔	✔	✔		✔	✔			~	✔
Data Parallelism	✔		✔			~	✔		✔			✔
Any-to-any synch	✔					✔		✔	~	✔		✔
Reductions	✔	✔	✔	~	~		✔	✔	~			✔
Collectives	✔	~	~	~	~	✔	~	~			✔	✔
Data-directed Synchronization	✔					✔					✔	✔
Triggered Tasks	✔										✔	✔
Cache-aware Scheduler	✔	✔	✔	✔		~				✔	✔	✔
NUMA-aware Scheduler	✔					✔			✔	~	✔	~
Task Pinning	✔			✔	✔			✔			✔	~
Spawn Cache	✔					~			✔			
Task Teams	✔								✔	~		
I/O Handling	✔					?			✔			
Modifiable Parallelism	✔			~	~	✔						
Reactive Parallelism	✔											
Compiler Independent	✔		✔					✔		✔		
Multinode-only Features												
Remote task spawn	✔					✔						
SPMD	✔									~		
MIMD	✔					✔			✔	✔		