SAND2014-19476PE



Run Time Systems R&D with the Qthreads Multithreading Library

Dylan Stark, Stephen Olivier
Dept. 1423, Sandia National Laboratories, NM

Technical Seminar

Sandia - California Site

November 4, 2014

Sandia National Laboratories

Exceptional service in the national interest

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.

Outline

- Introduction
- Node-level work
  - OpenMP interface
  - Locality and Power-awareness
  - Kokkos interface
- Distributed memory work
  - Chapel interface
  - Unified Scalable Parallel Runtime
  - MPI+Qthreads integration

Qthreads Philosophy

- Qthreads as a vehicle for run time system research
  - Co-design efforts with architecture and applications
  - Modular for flexibility and extensibility
    - Interfaces to OpenMP, Kokkos, Chapel, <your language here>
    - Different schedulers, e.g., work stealing, hierarchical
- Crowded space of run time system solutions
  - Don't claim to have the best, but strive to improve
  - Still many unsolved problems
    - Want to gain understanding
    - Seek collaboration

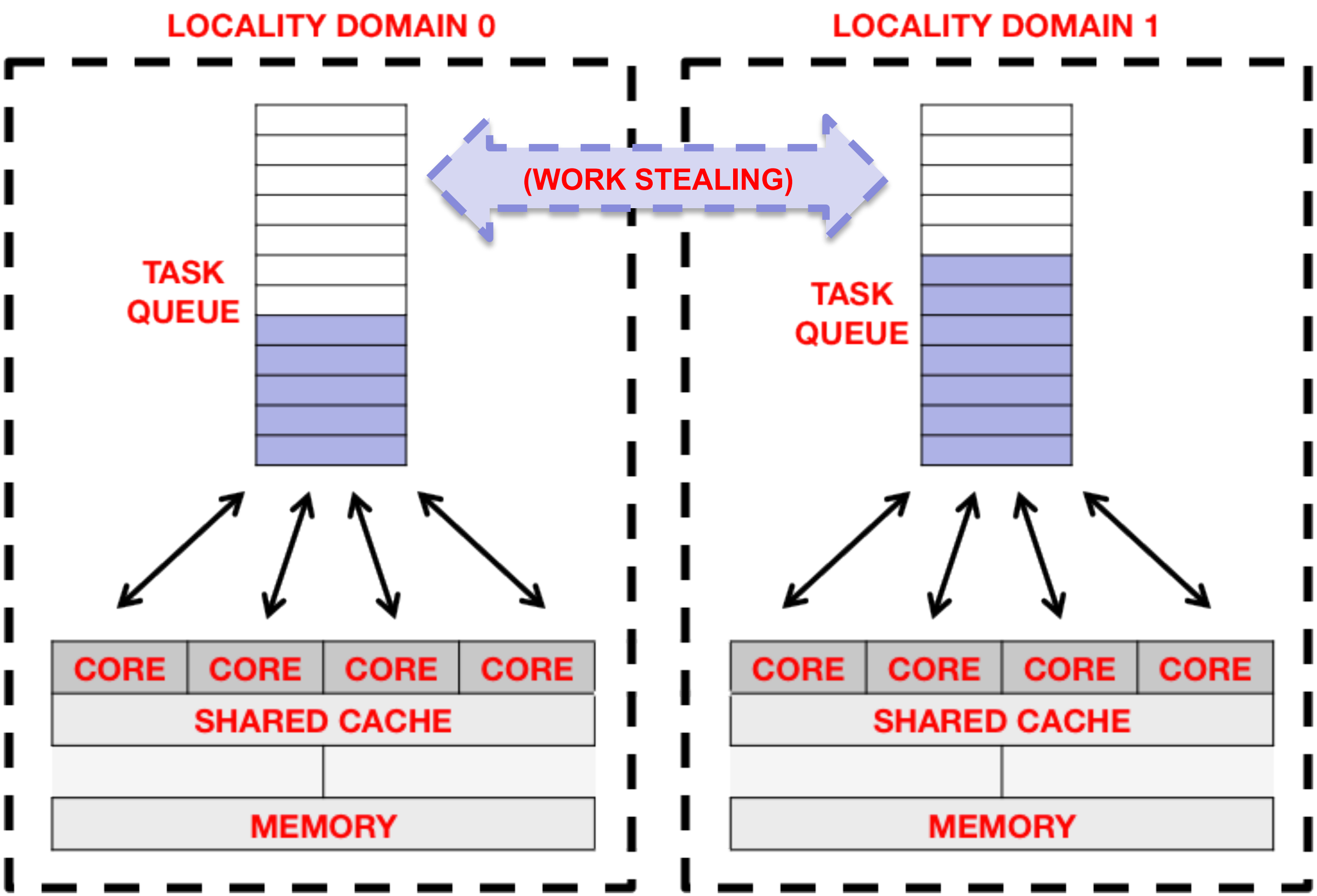
Qthreads Overview

- Programmer exposes application parallelism as massive numbers of lightweight tasks (qthreads)
  - Problem-centric rather than processor-centric decomposition enhances productivity, transparent scaling
  - Both loop-based and task-based parallelism supported
  - Full/empty bit primitives for powerful, lightweight synchronization (emulates Tera/Cray MTA/XMT behavior)
  - C API with no special compiler support required
- Dynamic run time system manages the scheduling of tasks for locality and performance
  - Heavyweight worker pthreads execute the tasks
  - Worker pthreads pinned to underlying hardware

Qthreads Capabilities

- Locality-aware load balancing of tasks to support NUMA and complex cache hierarchies
  - Locality domain with work queue shared among worker threads that share cache and memory
  - Work stealing between locality domains for global load balancing
- Lightweight task context switching
- Ported to x86, Phi, PPC, Sparc, Tilera

Qthreads Run Time View of Locality



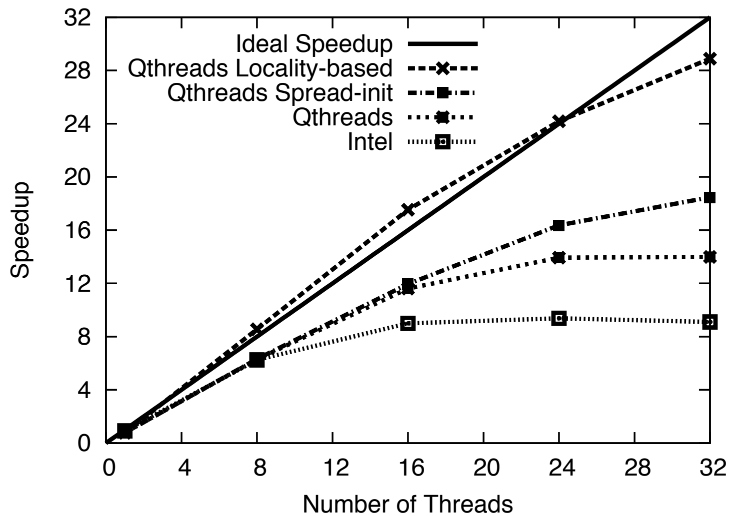
OpenMP-over-Qthreads

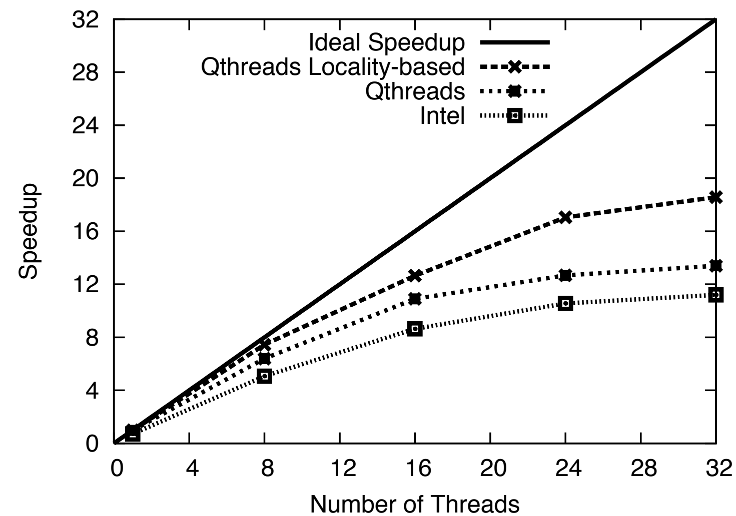
- Qthreads as run time for OpenMP
  - Allows execution of OpenMP codes without porting to Qthreads API
  - Enables experimentation with potential new OpenMP features
- Leverage existing OpenMP front-ends
  - ROSE / XOMP interface (LLNL -- Quinlan/Liao)
    - Mappings for OpenMP constructs to run time library functions
    - Supports OpenMP 3.1
  - Intel's OpenMP interface (open-sourced at openmprtl.org)
    - Early investigations
    - OpenMP 4.0 and beyond

OpenMP Locality Extensions

- Added support for placing tasks onto locality domains
  - Map to NUMA regions to avoid remote memory accesses
  - Builds on hierarchical scheduler in Qthreads

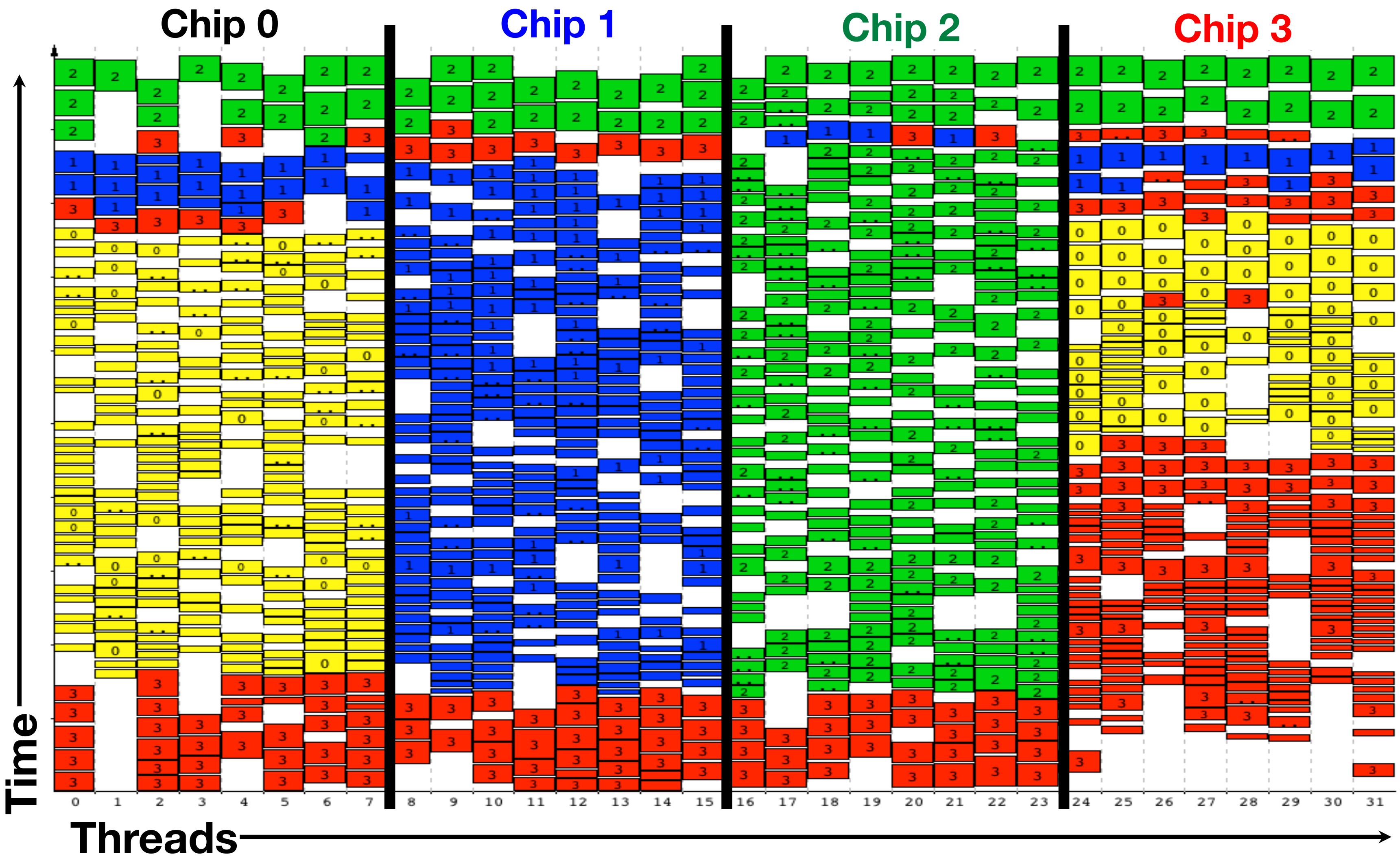
- Increased performance on Health and Heat benchmarks



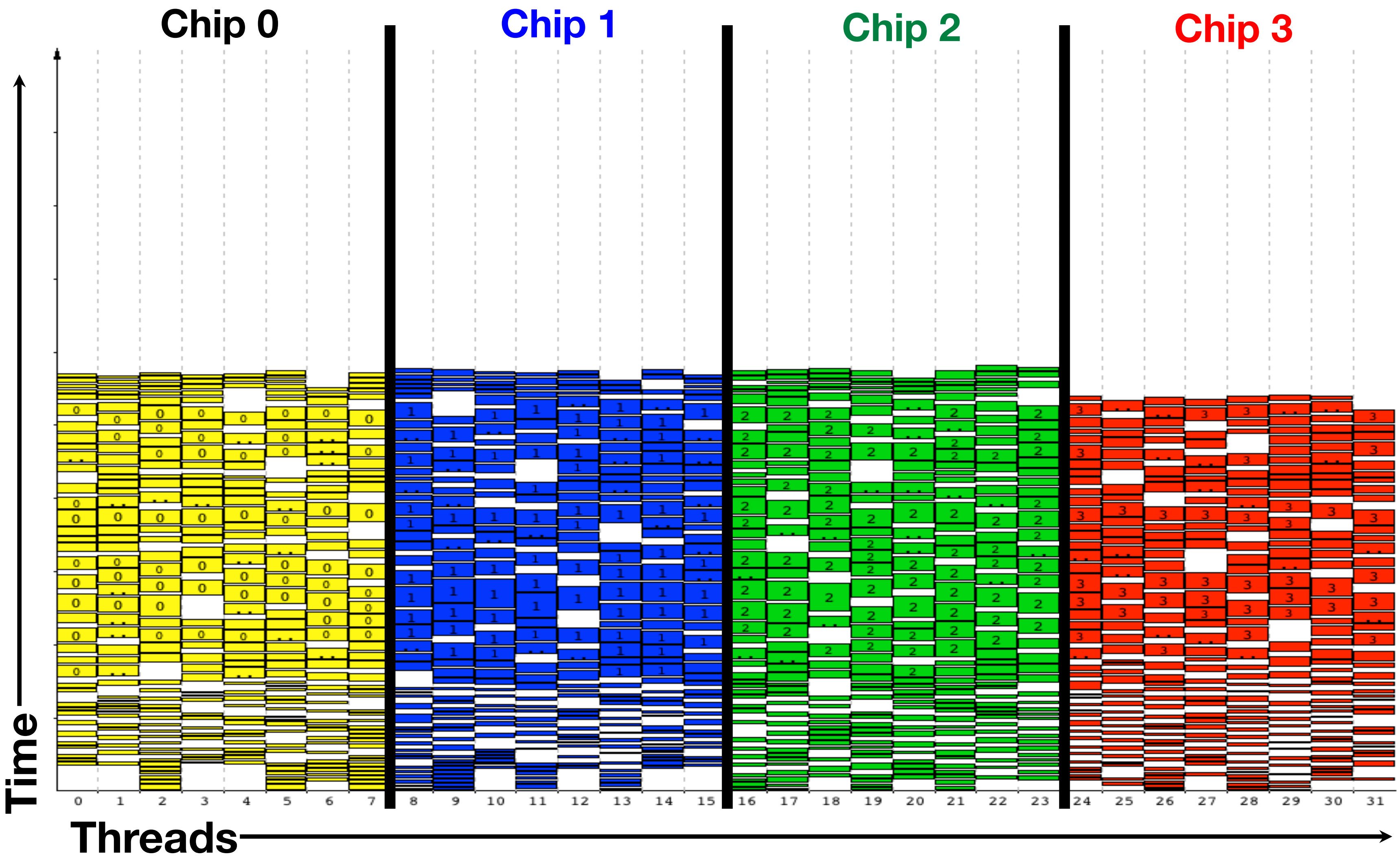


[SC12 paper with Martin Schulz, Bronis de Supinski, Jan Prins]

Sample Schedule: Locality Oblivious



Locality-Based Schedule



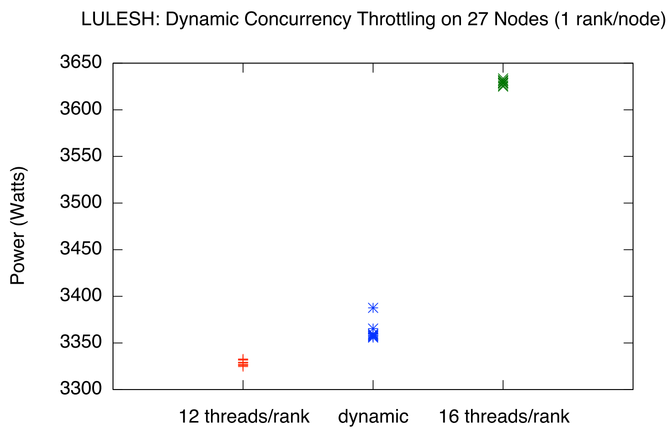
Dynamic Concurrency Throttling

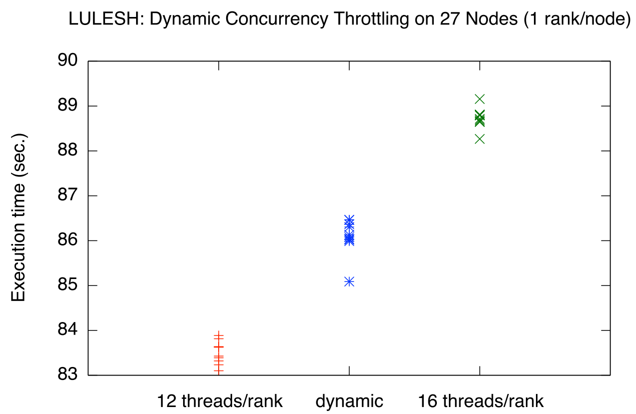
Sandia National Laboratories

- Observe memory saturation in some OpenMP codes
  - Could use fewer than the maximum available cores
  - Save power by shutting down unused cores
- RCRTool (Allan Porterfield of RENCI)
  - Monitors hardware performance counters
  - Reports CPU, memory, power data on a blackboard
- Maestro (power-aware Qthreads, also with Porterfield)
  - Qthreads queries RCR blackboard data when scheduling tasks
  - If memory saturated, shuts down some worker threads
    - Corresponding cores clocked down
    - Spin back up later if conditions change

Dynamic Concurrency Throttling

- Evaluation on LULESH
  - OpenMP+MPI hybrid code
    - Independent Qthreads instance on each node
    - Unmodified MPI across 27 nodes
  - Power savings on 16-core SandyBridge by throttling to 12 cores
  - Relief of memory pressure improves performance over 16-core runs





[HP-PAC14 and HP-PAC13 (Grant et al., Porterfield et al.)]

Kokkos Task Parallel API (LDRD)

Sandia National Laboratories

Existing SNL Technologies: Kokkos & Qthreads

Kokkos C++ API for efficient manycore data-vector parallelism	Qthreads multithreading library for scalable task parallelism

Development of New Capabilities

Extend Kokkos API for task parallelism and graph processing	Extend Qthreads for nested data parallelism, Phi, GPU tasks

Goal: Unified Task-Data-Vector Manycore API

Performance portable C++ API for CSE and graph applications

Kokkos Task Parallel API Design

- Expand Kokkos API with *future* objects
  - Handles to either serial or data parallel tasks
  - Templated on return type, execution space (e.g., host or accelerator)

- Targeting Multi-Threaded Graph Library (Berry), hybrid matrix factorization (Rajamanickam), Finite Element codes (Edwards)
  - Just starting Year 2 of LDRD

Kokkos/Qthread LDRD: Task Parallelism

- **TaskPolicy< Space > and Future< type , Space >**
  - Task policy object for a group of potentially concurrent tasks

```
TaskPolicy<> manager( ... ); // default Space
Future<type> fa = manager.spawn( functor_a ); // single-thread task
Future<type> fb = manager.spawn( functor_b ); // may be concurrent
```

  - Tasks may be data parallel via data parallel pattern and policy

```
Future<>       fc = manager.foreach(RangePolicy(0,N)).spawn( functor_c );
Future<type> fd = manager.reduce(TeamPolicy(N,M)).spawn( functor_d );
wait( tm ); // Host can wait for all tasks to complete
```

  - Destruction of task manager object waits for concurrent tasks to complete
- **Task Manager : TaskPolicy< Space = Qthread >**
  - Defines a scope for a collection of potentially concurrent tasks
  - Have configuration options for task management and scheduling
  - Manage resources for scheduling queue

Kokkos/Qthread LDRD: Task Parallelism

- **Tasks may have execution dependences**
  - Start a task only after other tasks have completed

```
Future<> array_of_dep[ M ] = { /* futures for other tasks */ };
```

  - Single threaded task:

```
Future<> fx = manager.spawn( functor_x , array_of_dep , M );
```

  - Tasks and their dependences define a directed acyclic graph (dag)
- **Challenge: A GPU task cannot 'wait' on dependences**
  - An executing GPU task cannot be suspended – waiting blocks a processor
  - Other future light-weight core architecture may not be able to block as well
  - A task may spawn nested tasks and need to wait for their completion
  - Solution: 'respawn' the task with new dependences

```
manager.respawn( this , array_of_dep , M );
return ; // 'this' returns to be called after new dependences complete
```

Distributed Memory & Qthreads

- Three use cases:

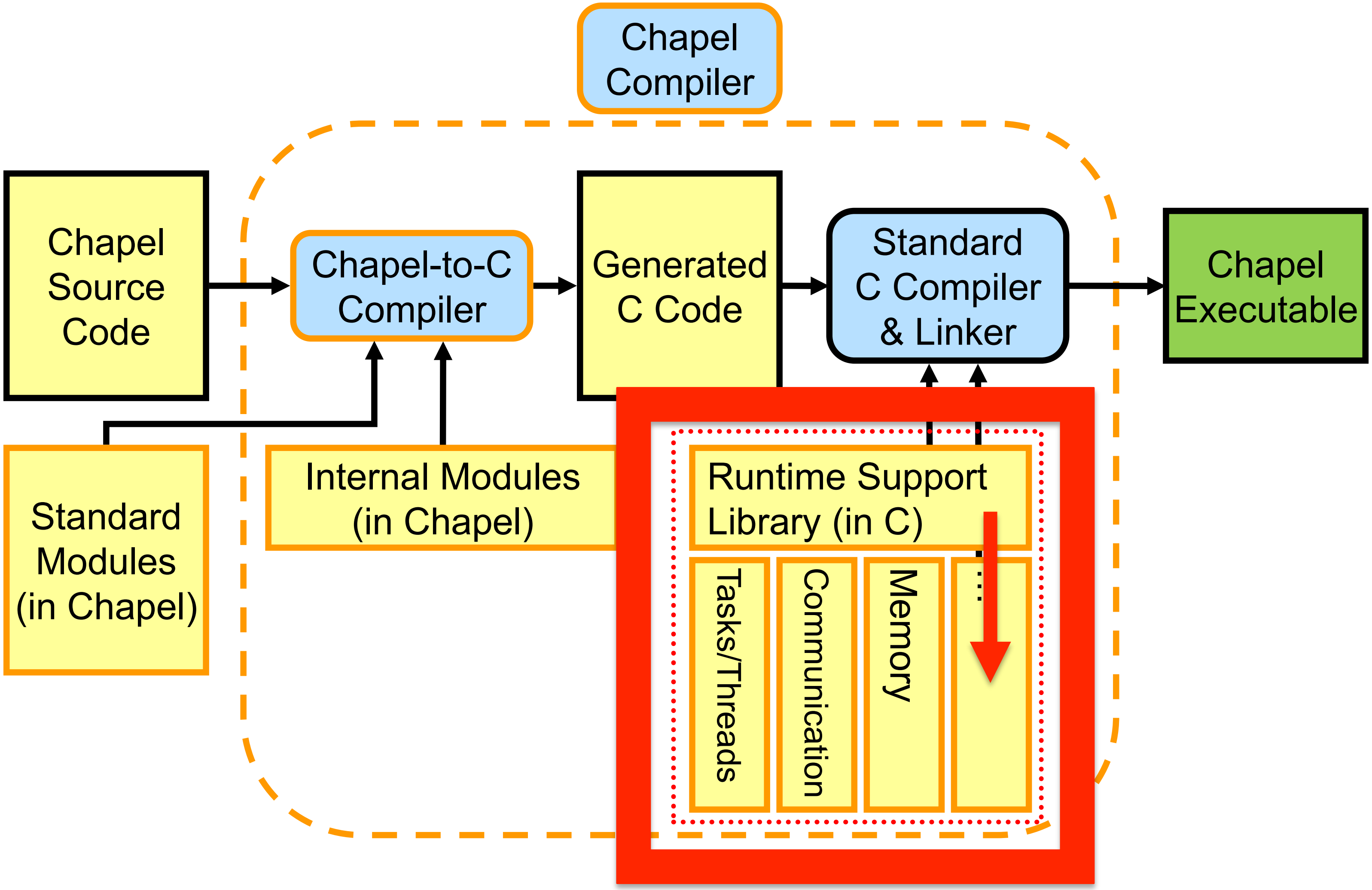
  1. Stove pipe model for Chapel

  2. Unified runtime model with Portals 4 for Chapel

  3. Managed model for MPI+X

1) STOVE PIPE MODEL FOR CHAPEL

Chapel compilation and runtime



Chapel compilation and runtime

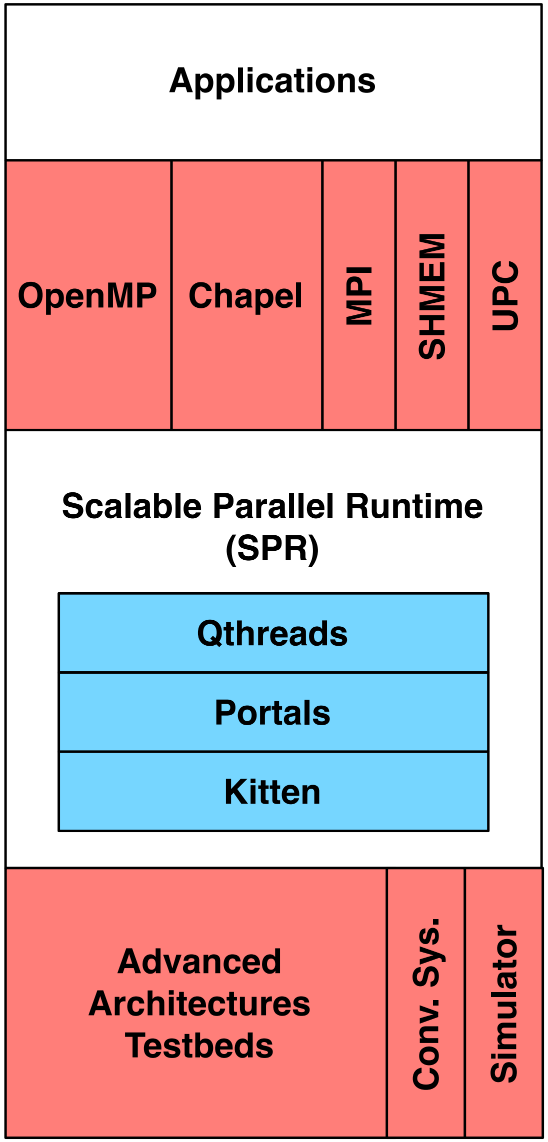
- Straightforward
- Works out of the box
- Easy to mix and match TPLs

- But comes at a cost:
  - Information must be collected and managed by Chapel shim
  - Blocked cores for certain comm. operations
  - Slow-path for task creation and synchronization

2) UNIFIED RUNTIME MODEL WITH PORTALS 4 FOR CHAPEL

A Unified Runtime Example

- **Qthreads: Lightweight threading interface**
  - Scalable, lightweight scheduling on NUMA platforms
  - Supports a variety of synchronization mechanisms, including full/empty bits and atomic operations
  - Potential for direct hardware mapping
- **Portals 4: Lightweight communication interface**
  - Semantics for supporting both one-sided and tagged message passing
  - Small set of primitives, allows offload from main CPU
  - Supports direct hardware mapping
- **Kitten: Lightweight OS kernel**
  - Builds on lessons from ASCI Red, Cplant, Red Storm
  - Utilizes scalable parts of Linux environment
  - Primarily supports direct hardware mapping



Chapel with a Unified Runtime

- Replaced Qthreads & GASNet with SPR (Qthreads + Portals4)
  - Single point for initializing both platforms: spr_init(SPMD,...)
  - spr_unify() used to transition to single thread of control before application starts
  - Most other interface functions are no-ops (e.g., chpl_task_init(), chpl_comm_post_task_init(), chpl_comm_rollcall(), ...)
  - Direct mappings for data movement and work migration
- Now both layers share ...
  - Platform information discovery (to make room for progress engine)
  - Memory management (for activation records, stacks, network packets)
  - Synchronization mechanisms (such as full-empty support)
  - Direct task spawning and management

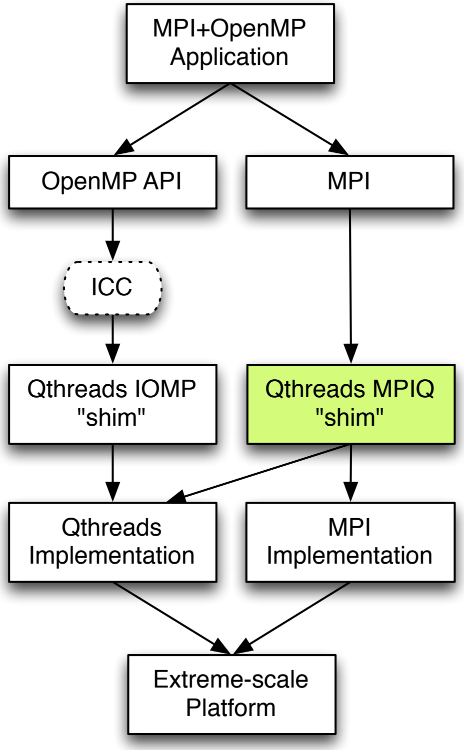
3) MANAGED MODEL FOR MPI+X

Early exploration with MPI+Qthreads (MPIQ)

- Task-parallel runtime for resource management
  - Extension of Sandia Qthreads library
  - Low-level C API, supports other PMs (OpenMP, Sandia Kokkos, etc.)

- Practical target for C/C++ mini-apps
  - Concurrent MPI calls from any context
  - Communication is just “long latency event”

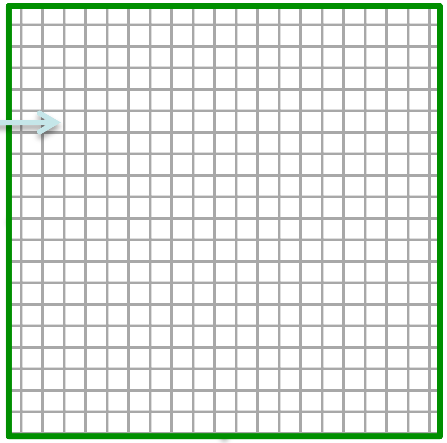
- Requirements on runtime:
  - Support possible over-subscription of concurrent blocking MPI calls
  - Manage long-latency events in cooperatively scheduled tasks
  - And co-schedule work and communication



Code modification for miniGhost

Data parallel model:

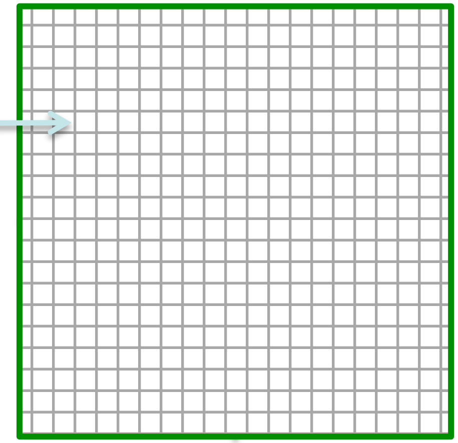
```
int stencil (…) {
   Exchange_boundary_data ( … );
   Apply_boundary_conditions ( … );
   Apply_stencil ( …);
}
```



Code modification for miniGhost

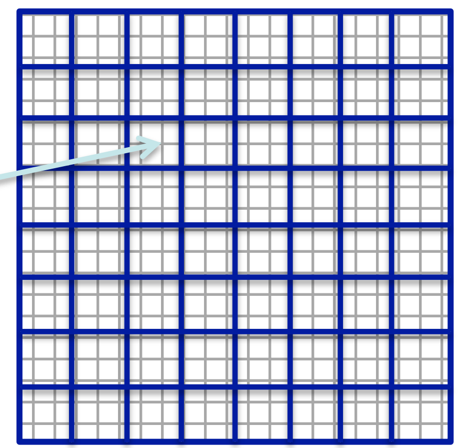
Data parallel model:

```
int stencil (…) {
   Exchange_boundary_data ( … );
   Apply_boundary_conditions ( … );
   Apply_stencil ( …);
}
```

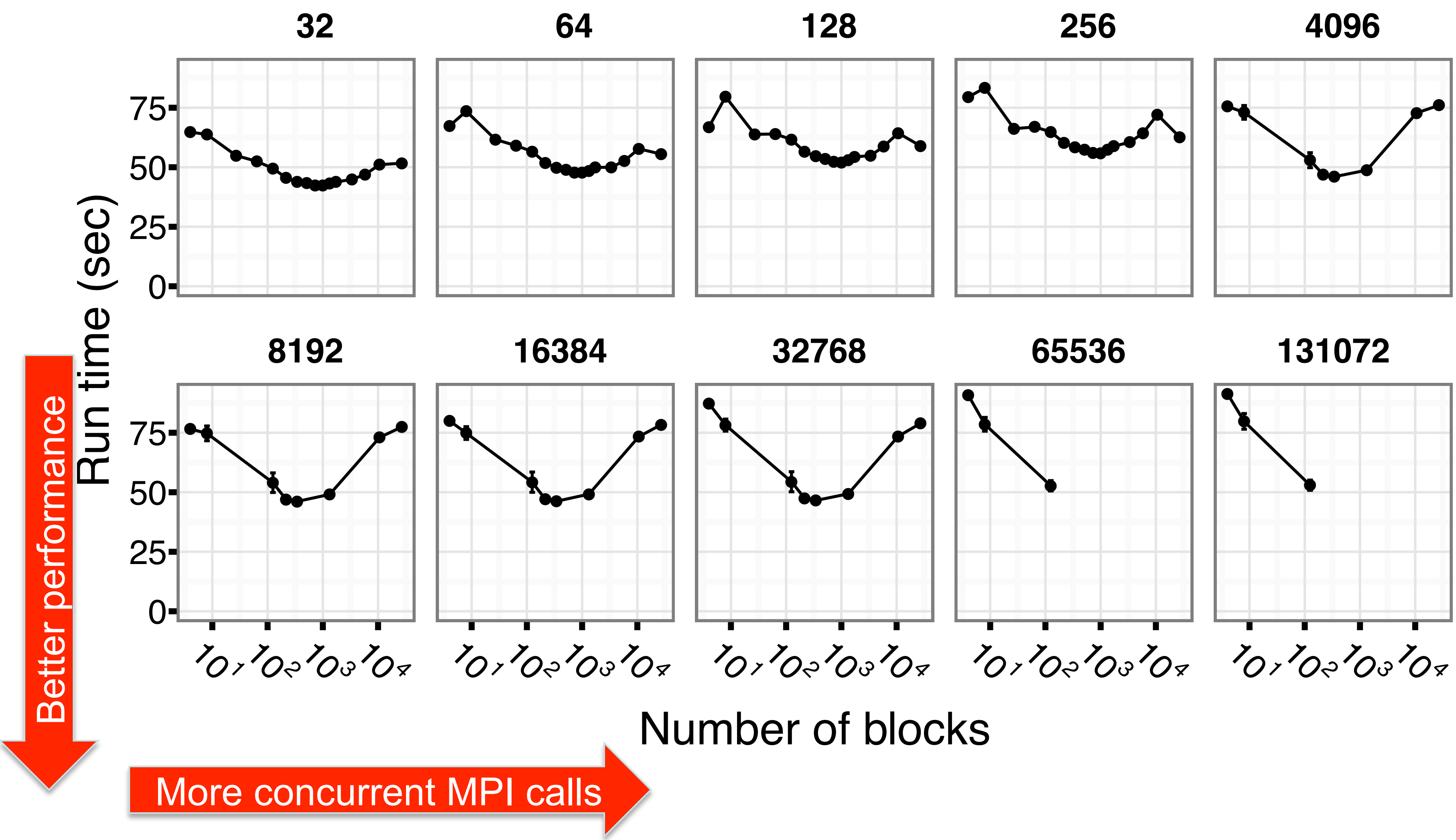


Task parallel model: Loop over blocks; spawned code is the usual data parallel model.

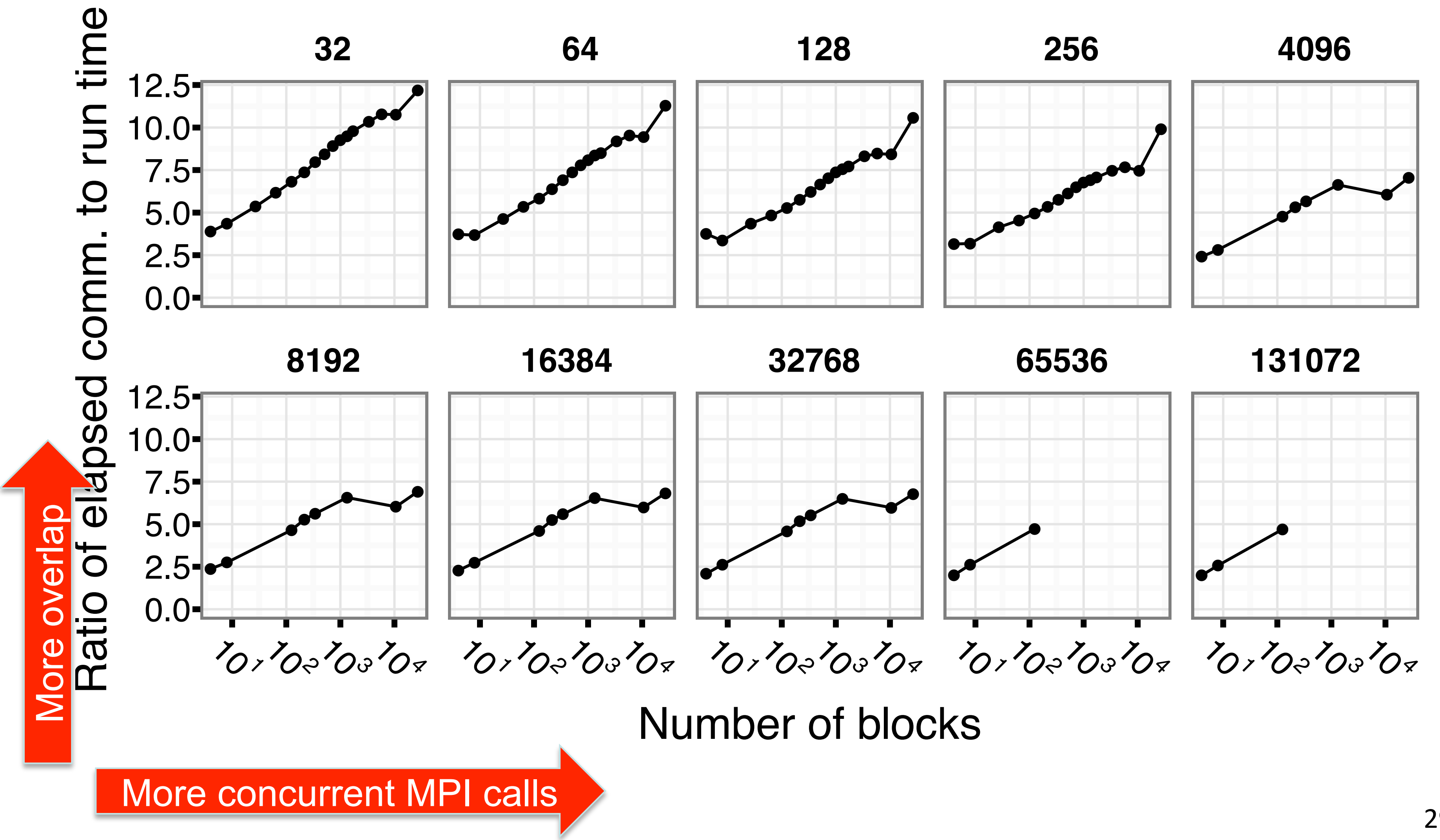
```
ierr = MG_Block_init ( blks, … );

for ( i=0; i<numblks; i++ ) {
    spawn ( stencil ( i, … ));
}
```



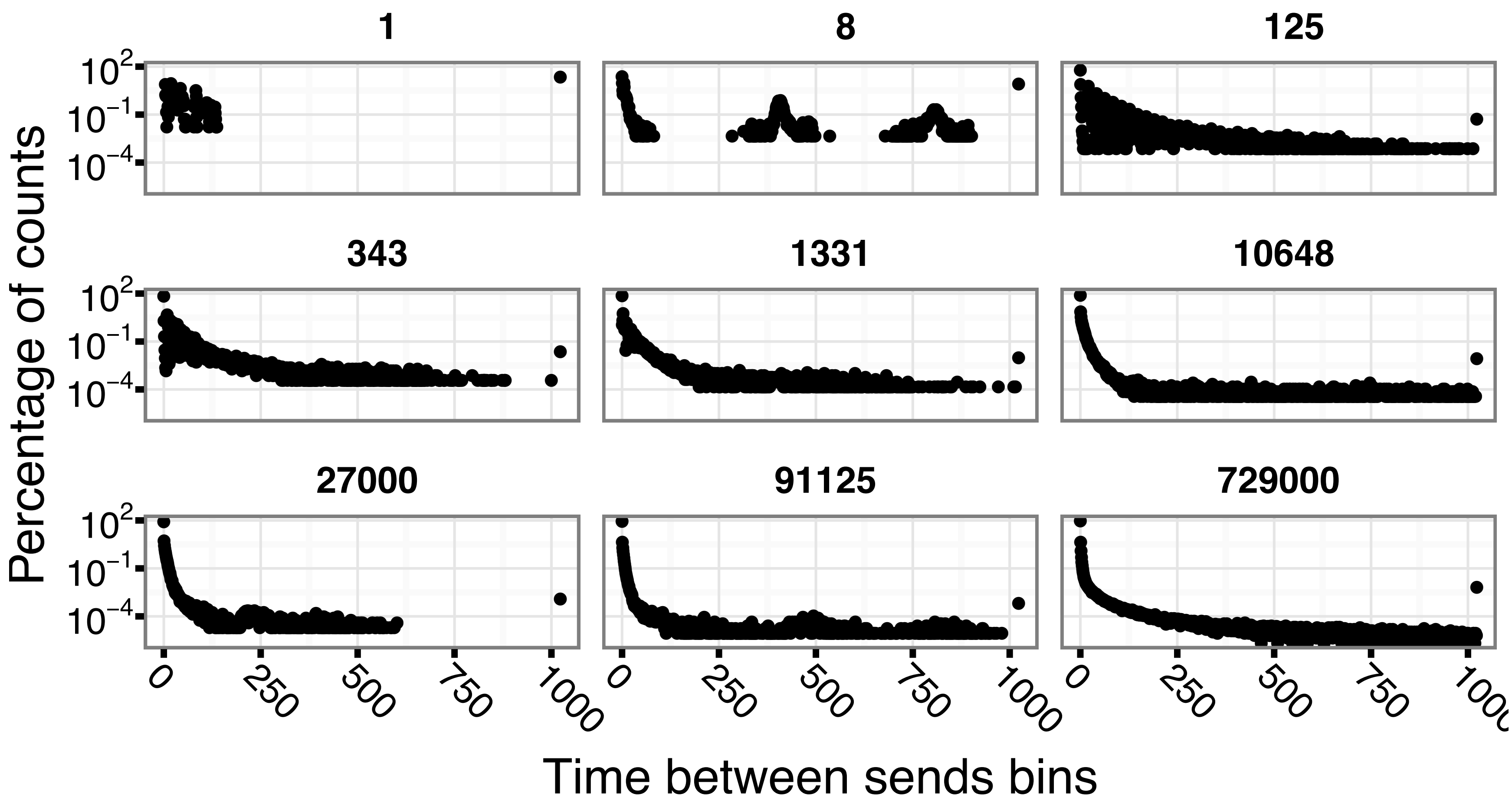
Increasing performance with over-subscription



Overlapping communication and computation



Spreading message injection (256 cores, or 64 ranks)



Longer between sends

Summary

- Qthreads: a vehicle for threaded runtime research
- Node-level work
  - OpenMP interface
  - Locality and Power-awareness
  - Kokkos interface
- Distributed memory work
  - Chapel interface
  - Unified Scalable Parallel Runtime
  - MPI+Qthreads integration

Contributors to Qthreads Research

- Richard Barrett, Carter Edwards, Ryan Grant, Courtenay Vaughan, Kevin Pedretti, Jon Berry, Siva Rajamanickam (SNL)
- Kyle Wheeler and Rich Murphy (now at Micron)
- Brian Barrett (now at Amazon)
- George Stelle (UNM)
- Alina and Dragos Sbirlea (Rice)
- Brad Chamberlain and Greg Titus (Cray)
- Allan Porterfield and Jan Prins (UNC/RENCI)
- Bronis de Supinski and Martin Schulz (LLNL)
- Marc Snir and Alex Brooks (UIUC)

Available Online

Qthreads

More info: http://www.cs.sandia.gov/qthreads/

Source: https://code.google.com/p/qthreads/