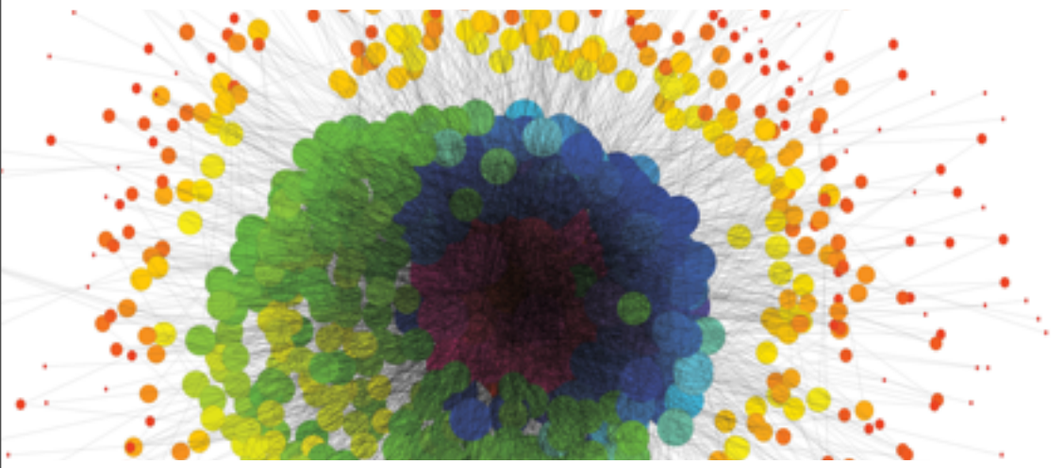
SAND2013-0373P

Exceptional service in the national interest

Sandia National Laboratories



Qthreads Briefing for AMD

Fast-Forward Jan 2013

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

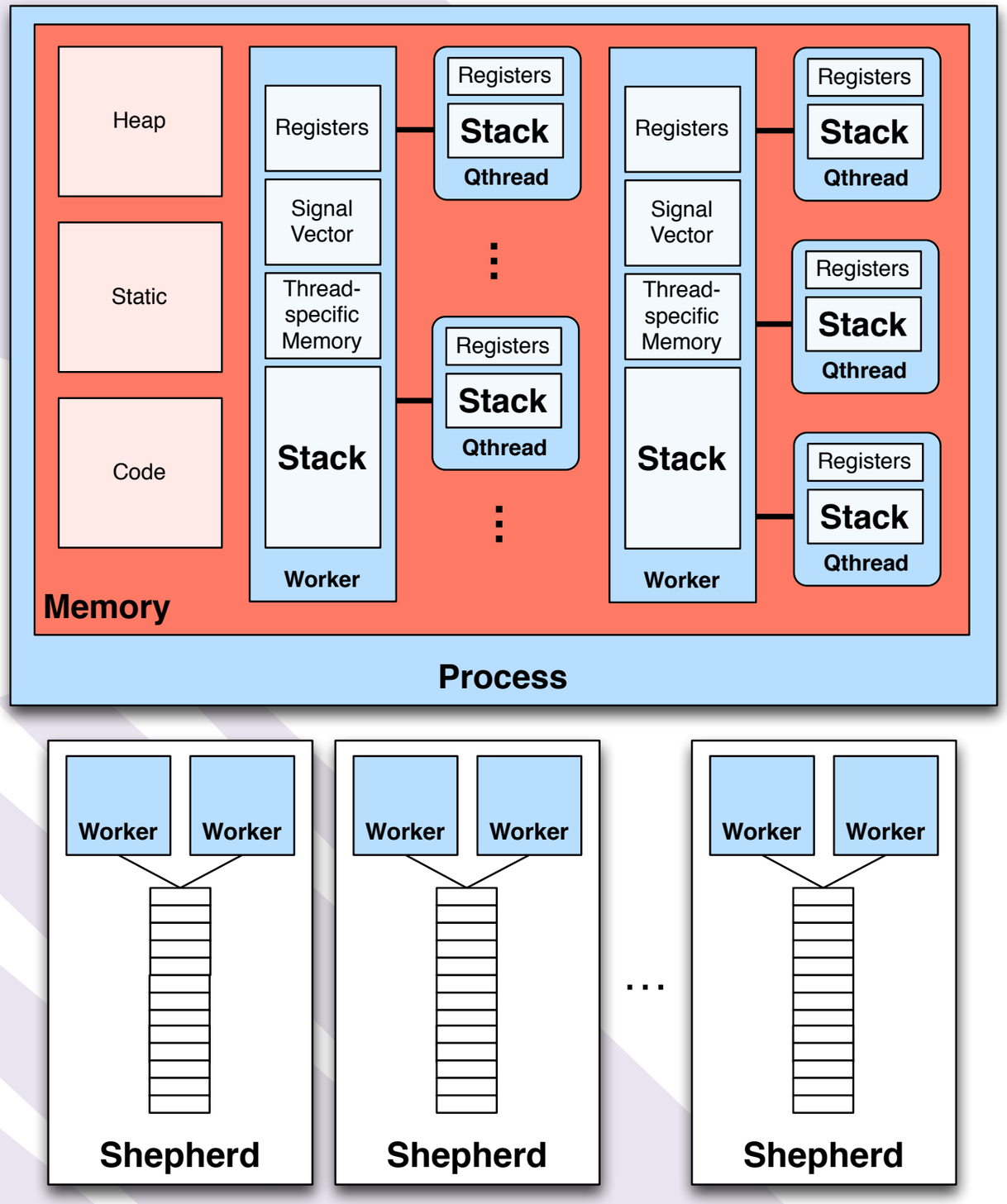
Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Agenda

- General Concepts
  - What's a qthread?
  - What are the "crown jewels"?
- Interface & API
- Synchronization Details
- Scheduler Details

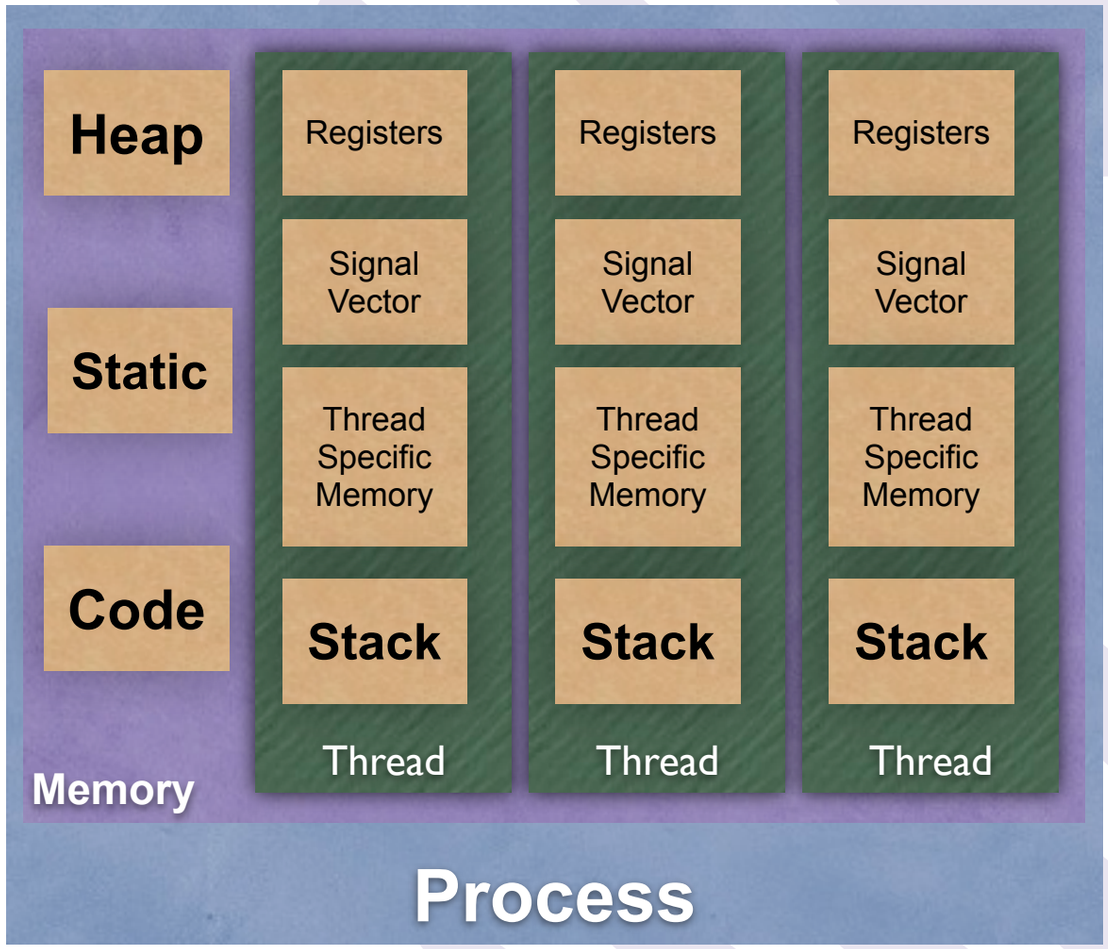
Qthreads Highlights

- Lightweight User-level Threading (tasking)
- Platform Portability
  - IA32/64, AMD64, PPC32/64, SparcV9+, SST, Tilera, ARM
  - Linux, BSD, Solaris, MacOSX, Cygwin
- Locality fundamental to model
  - "Shepherd" as thread-mobility domain
- Fine-grained synchronization
  - Full/Empty Bits (64-bit & 60-bit)
  - Mutexes
  - Atomic Operations (Integer incr, Float incr, & CAS)
- Locality-aware Cache-aware Workstealing Scheduler

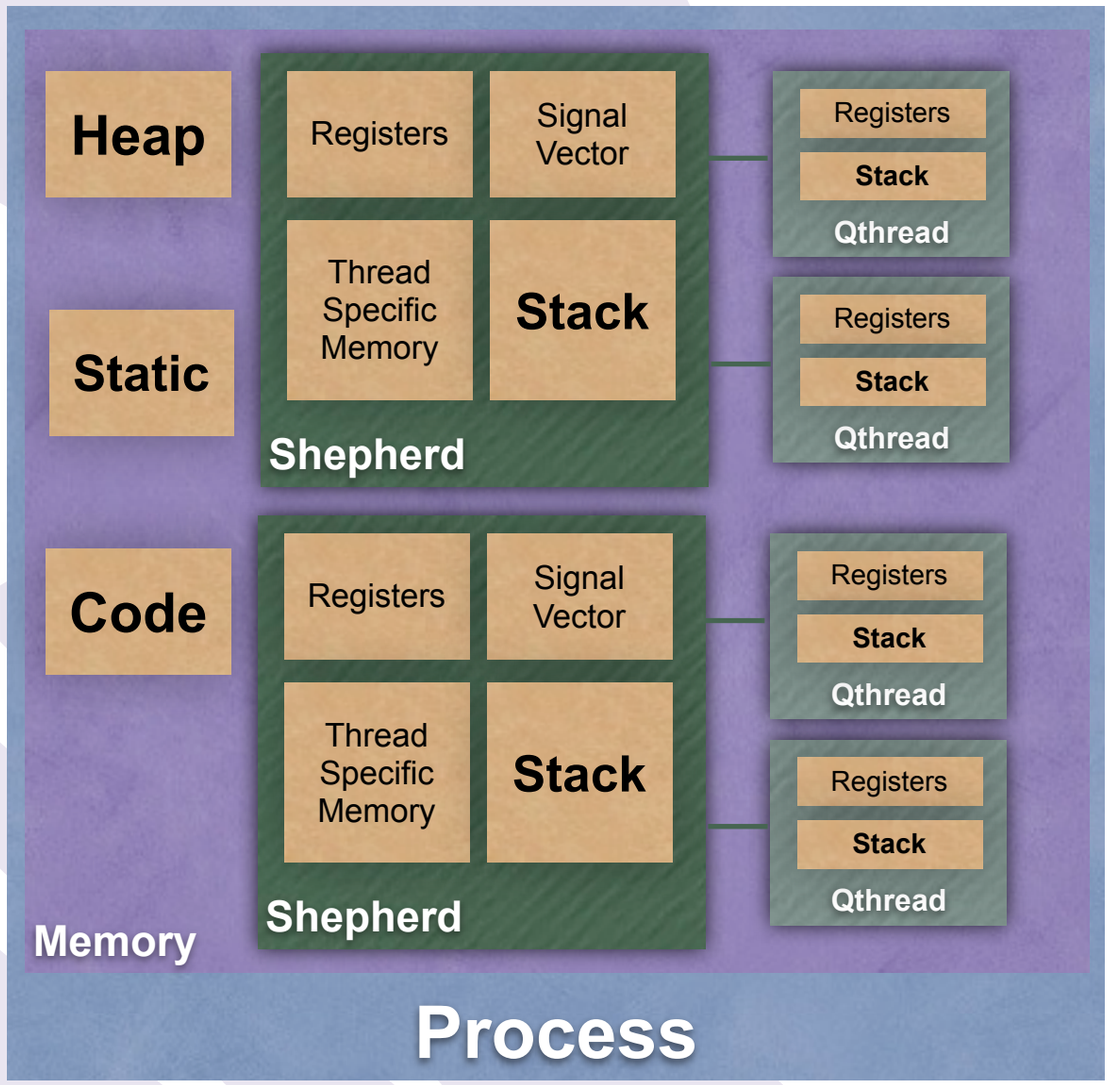


Thread Structure

Standard Pthreads



Qthreads



Qthread Programmer Conveniences

Sandia National Laboratories

- Threaded loop constructs
  - Tree-based task spawning, efficient barriers
  - `qt_loop()` - task per iteration
  - `qt_loop_balance()` - task per worker thread
  - `qt_queue_loop()` - task per dynamic self-scheduled work-chunk
- Data Structures
  - Distributed across shepherds
  - Qpool - distributed memory pool
  - Qarray - distributed arrays
  - Qlfqueue - lock-free MWMR queue
  - Qdqueue - distributed MWMR queue, built on qlfqueue
  - Qswsrqueue - lock-free SWSR queue
  - Qt_dictionary - lock-free key-value store

MTGL PageRank Core: XMT

Sandia National Laboratories

```
template <>
void compute_acc<compressed_sparse_row_graph<bidirectionalS> >(
    compressed_sparse_row_graph<bidirectionalS>& g, rank_info* rinfo)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  unsigned int n = num_vertices(g);

  int i, j;
  vertex_id_map<Graph> vid_map = get(_vertex_id_map, g);
  vertex_descriptor_t* rev_end_points = g.get_rev_end_points();

  #pragma mta assert nodep
  #pragma mta use 100 streams
  for (int i = 0; i < n; i++) {
    double total = 0.0;
    int begin = g[i];
    int end = g[i + 1];

    for (int j = begin; j < end; j++) {
      int src = rev_end_points[j];
      double r = rinfo[src].rank;
      double incr = (r / (double) rinfo[src].degree);
      total += incr;
    }

    rinfo[i].acc = total;
  }
}
```

MTGL PageRank Core: XMT

Sandia National Laboratories

```
template <>
void compute_acc<compressed_sparse_row_graph<bidirectionalS> >(
    compressed_sparse_row_graph<bidirectionalS>& g, rank_info* rinfo)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  unsigned int n = num_vertices(g);

  int i, j;
  vertex_id_map<Graph> vid_map = get(_vertex_id_map, g);
  vertex_descriptor_t* rev_end_points = g.get_rev_end_points();

  #pragma mta assert nodep
  #pragma mta use 100 streams
  for (int i = 0; i < n; i++) {
    double total = 0.0;
    int begin = g[i];
    int end = g[i + 1];

    for (int j = begin; j < end; j++) {
      int src = rev_end_points[j];
      double r = rinfo[src].rank;
      double incr = (r / (double) rinfo[src].degree);
      total += incr;
    }

    rinfo[i].acc = total;
  }
}
```

Defines Thread Behavior

Defines Parallelism

MTGL PageRank Core: XMT

```
template <>
void compute_acc<compressed_sparse_row_graph<bidirectionalS> >(
    compressed_sparse_row_graph<bidirectionalS>& g, rank_info* rinfo)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  unsigned int n = num_vertices(g);

  int i, j;
  vertex_id_map<Graph> vid_map = get(_vertex_id_map, g);
  vertex_descriptor_t* rev_end_points = g.get_rev_end_points();

  #pragma mta assert nodep
  #pragma mta use 100 streams
  for (int i = 0; i < n; i++) {
    double total = 0.0;
    int begin = g[i];
    int end = g[i + 1];

    for (int j = begin; j < end; j++) {
      int src = rev_end_points[j];
      double r = rinfo[src].rank;
      double incr = (r / (double) rinfo[src].degree);
      total += incr;
    }

    rinfo[i].acc = total;
  }
}
```

MTGL PageRank Core: XMT

```
template <>
void compute_acc<compressed_sparse_row_graph<bidirectionalS> >(
    compressed_sparse_row_graph<bidirectionalS>& g, rank_info* rinfo)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  unsigned int n = num_vertices(g);

  int i, j;
  vertex_id_map<Graph> vid_map = get(_vertex_id_map, g);
  vertex_descriptor_t* rev_end_points = g.get_rev_end_points();

  #pragma mta assert nodep
  #pragma mta use 100 streams
  for (int i = 0; i < n; i++) {
    double total = 0.0;
    int begin = g[i];
    int end = g[i + 1];

    for (int j = begin; j < end; j++) {
      int src = rev_end_points[j];
      double r = rinfo[src].rank;
      double incr = (r / (double) rinfo[src].degree);
      total += incr;
    }

    rinfo[i].acc = total;
  }
}
```

```
template<>
void compute_acc_outer<compressed_sparse_row_graph<bidirectionalS> >(
    qthread_t* me, const size_t startat, const size_t stopat,
    void* arg)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  qt_rank_info<Graph>* qt_rinfo = (qt_rank_info<Graph>*)arg;
  vertex_id_map<Graph> vid_map = get(_vertex_id_map, qt_rinfo->g);
  vertex_descriptor_t* rev_end_points =
      qt_rinfo->g.get_rev_end_points();

  for (size_t i = startat ; i < stopat ; ++i) {
    double total = 0.0;
    int begin = qt_rinfo->g[i];
    int end = qt_rinfo->g[i + 1];

    for (int j  = begin ; j < end ; ++j) {
      vertex_descriptor_t src = rev_end_points[j];
      double r = qt_rinfo->rinfo[src].rank;
      double incr = (r / qt_rinfo->rinfo[src].degree);
      total += incr;
    }

    qt_rinfo->rinfo[i].acc = total;
  }
}
```

MTGL PageRank Core: XMT

Sandia National Laboratories

```
template <>
void compute_acc<compressed_sparse_row_graph<bidirectionalS> >(
    compressed_sparse_row_graph<bidirectionalS>& g, rank_info* rinfo)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  unsigned int n = num_vertices(g);

  int i, j;
  vertex_id_map<Graph> vid_map = get(_vertex_id_map, g);
  vertex_descriptor_t* rev_end_points = g.get_rev_end_points();

  #pragma mta assert nodep
  #pragma mta use 100 streams
  for (int i = 0; i < n; i++) {
    double total = 0.0;
    int begin = g[i];
    int end = g[i + 1];

    for (int j = begin; j < end; j++) {
      int src = rev_end_points[j];
      double r = rinfo[src].rank;
      double incr = (r / (double) rinfo[src].degree);
      total += incr;
    }

    rinfo[i].acc = total;
  }
}
```

```
template <>
void compute_acc<compressed_sparse_row_graph<bidirectionalS> >(
    compressed_sparse_row_graph<bidirectionalS>& g, rank_info* rinfo)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  unsigned int n = num_vertices(g);

  qt_rank_info<Graph> ri(g, rinfo);
  qt_loop_balance(0, n, compute_acc_outer<Graph>, &ri);
}

template<>
void compute_acc_outer<compressed_sparse_row_graph<bidirectionalS> >(
    qthread_t* me, const size_t startat, const size_t stopat,
    void* arg)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  qt_rank_info<Graph>* qt_rinfo = (qt_rank_info<Graph>*)arg;
  vertex_id_map<Graph> vid_map = get(_vertex_id_map, qt_rinfo->g);
  vertex_descriptor_t* rev_end_points =
      qt_rinfo->g.get_rev_end_points();

  for (size_t i = startat ; i < stopat ; ++i) {
    double total = 0.0;
    int begin = qt_rinfo->g[i];
    int end = qt_rinfo->g[i + 1];

    for (int j  = begin ; j < end ; ++j) {
      vertex_descriptor_t src = rev_end_points[j];
      double r = qt_rinfo->rinfo[src].rank;
      double incr = (r / qt_rinfo->rinfo[src].degree);
      total += incr;
    }

    qt_rinfo->rinfo[i].acc = total;
  }
}
```

MTGL PageRank Core: Qthreads

Sandia National Laboratories

```
template <>
void compute_acc<compressed_sparse_row_graph<bidirectionalS> >(
    compressed_sparse_row_graph<bidirectionalS>& g, rank_info* rinfo)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  unsigned int n = num_vertices(g);

  qt_rank_info<Graph> ri(g, rinfo);
  qt_loop_balance(0, n, compute_acc_outer<Graph>, &ri);
}
```

```
template<>
void compute_acc_outer<compressed_sparse_row_graph<bidirectionalS> >(
    qthread_t* me, const size_t startat, const size_t stopat,
    void* arg)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  qt_rank_info<Graph>* qt_rinfo = (qt_rank_info<Graph>*)arg;
  vertex_id_map<Graph> vid_map = get(_vertex_id_map, qt_rinfo->g);
  vertex_descriptor_t* rev_end_points =
      qt_rinfo->g.get_rev_end_points();

  for (size_t i = startat ; i < stopat ; ++i) {
    double total = 0.0;
    int begin = qt_rinfo->g[i];
    int end = qt_rinfo->g[i + 1];

    for (int j  = begin ; j < end ; ++j) {
      vertex_descriptor_t src = rev_end_points[j];
      double r = qt_rinfo->rinfo[src].rank;
      double incr = (r / qt_rinfo->rinfo[src].degree);
      total += incr;
    }

    qt_rinfo->rinfo[i].acc = total;
  }
}
```

MTGL PageRank Core: Qthreads

Sandia National Laboratories

```
template <>
void compute_acc<compressed_sparse_row_graph<bidirectionalS> >(
    compressed_sparse_row_graph<bidirectionalS>& g, rank_info* rinfo)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  unsigned int n = num_vertices(g);

  qt_rank_info<Graph> ri(g, rinfo);
  qt_loop_balance(0, n, compute_acc_outer<Graph>, &ri);
}
```

Defines Parallelism

```
template<>
void compute_acc_outer<compressed_sparse_row_graph<bidirectionalS> >(
    qthread_t* me, const size_t startat, const size_t stopat,
    void* arg)
{
  typedef compressed_sparse_row_graph<bidirectionalS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_descriptor_t;

  qt_rank_info<Graph>* qt_rinfo = (qt_rank_info<Graph>*)arg;
  vertex_id_map<Graph> vid_map = get(_vertex_id_map, qt_rinfo->g);
  vertex_descriptor_t* rev_end_points =
      qt_rinfo->g.get_rev_end_points();

  for (size_t i = startat ; i < stopat ; ++i) {
    double total = 0.0;
    int begin = qt_rinfo->g[i];
    int end = qt_rinfo->g[i + 1];

    for (int j  = begin ; j < end ; ++j) {
      vertex_descriptor_t src = rev_end_points[j];
      double r = qt_rinfo->rinfo[src].rank;
      double incr = (r / qt_rinfo->rinfo[src].degree);
      total += incr;
    }

    qt_rinfo->rinfo[i].acc = total;
  }
}
```

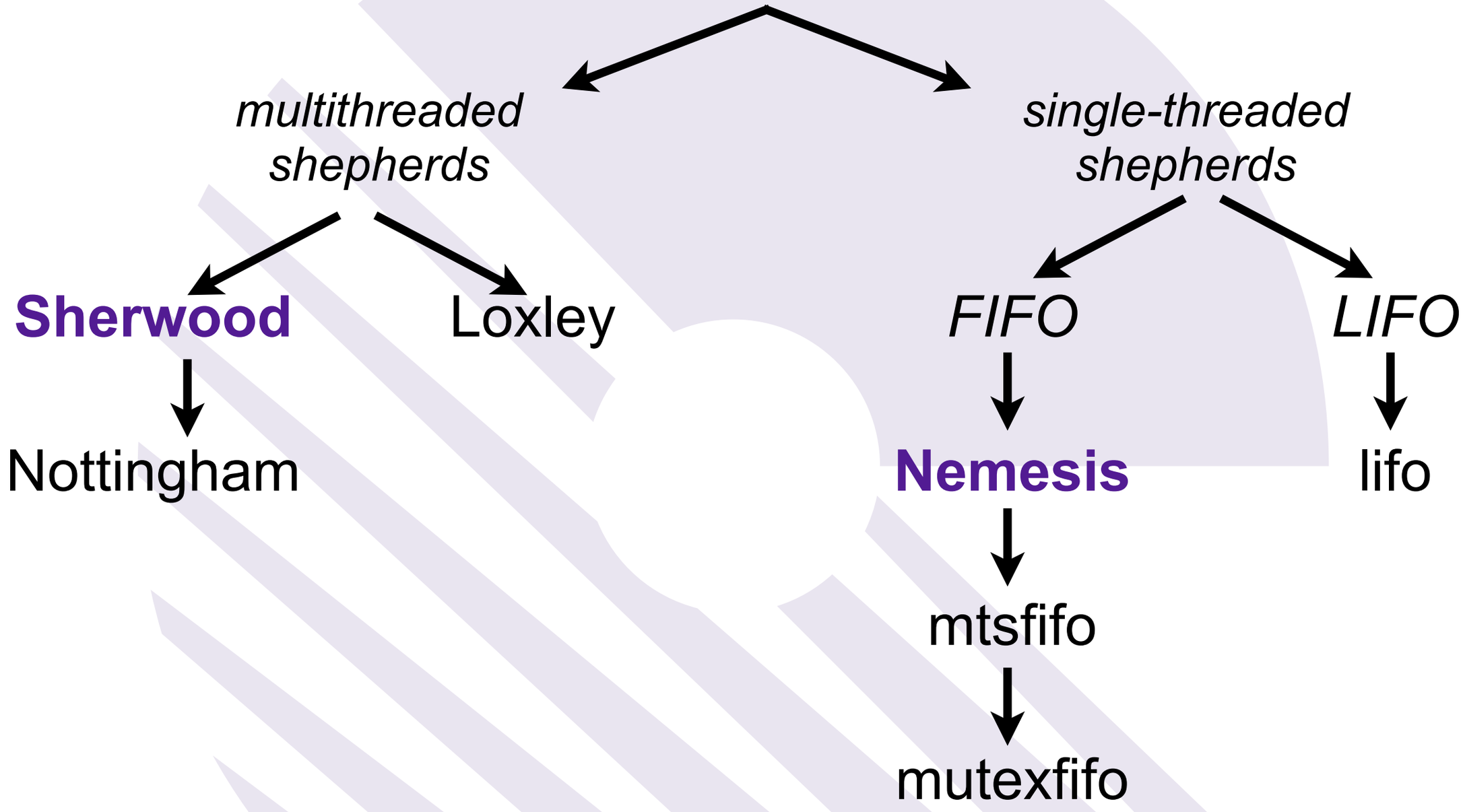
Defines Thread Behavior

Scheduling Requirements

Sandia National Laboratories

- Not all tasks can be stolen and/or migrated
  - For startup/teardown, the original thread context must be restored
  - “Tied” tasks, which specified an affinity...
    - ...must not be moved
    - ...probably shouldn’t have a higher priority than other tasks
- Remote spawns are possible
  - Sending work to the data is a fundamental operation
- Tasks are scheduled co-operatively
  - ...Mostly (can be killed; see eurekas)
- Workers and shepherds may be disabled by the user
  - Except for shepherd 0, worker 0
- All tasks have their own fixed-size stack

Schedulers


multithreaded shepherds
single-threaded shepherds
Sherwood
Loxley
FIFO
LIFO
Nottingham
Nemesis
lifo
mtsfifo
mutexfifo

The Sherwood Scheduler

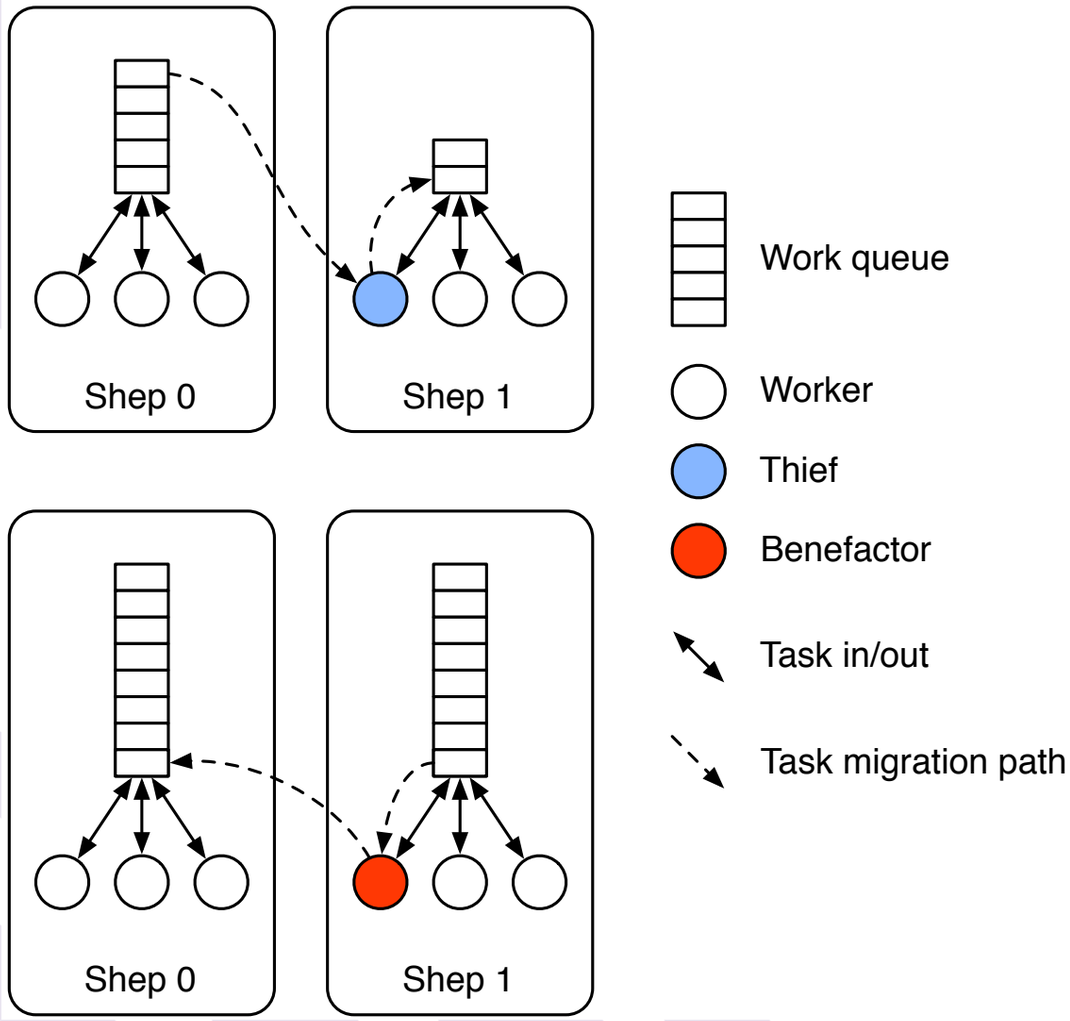
Sandia National Laboratories

- Combines Blumofe/Leiserson's workstealing scheduler with Blelloch's parallel depth-first scheduler
  - Workstealing has load-balancing
  - PDFS has provably nearest-to-sequential cache performance
- Uses topology information for both stealing and for establishing queues and grouping workers
- FIFO work-stealing between caches
  - e.g. to maintain L3 cache locality
- Important details:
  - Only one thief per chip performs work-stealing (avoid unnecessary communication)
  - Thief must steal multiple tasks (preferably half of what's available)
  - Not all tasks are stealable
  - **Contention avoidance is KEY to performance!**

Sherwood Scheduler Picture

Work-stealing

Remote-spawn



Nemesis Scheduler

Sandia National Laboratories

- Basic FIFO scheduler
- Relies on NEMESIS lock-free queue from MPICH2
  - VERY fast enqueue/dequeue
  - Only one worker-thread per queue
- Load distribution is spawn-time round-robin
  - Not adaptive
  - Similar to CHARM++
- Potential for periodic barrier-style load redistribution (see CHARM++)

Other Schedulers

- Shared Queues
  - Nottingham
    - EXPERIMENTAL!
    - Mostly lock-free implementation of Sherwood
    - Uses high-performance R/W lock to differentiate lock-free and locked access to queue
  - Loxley
    - EXPERIMENTAL!
    - Fixed-size-array queue implementation
    - Lock-free
- Queue-per-worker
  - MTSFifo
    - Multi-Threaded-Shepherd FIFO (not currently used for multi-threaded shepherds)
    - Lock-free, with hazard pointers
  - MutexFifo
    - For compatibility with difficult architectures (e.g. TILE64)
    - Baseline
  - Lifo
    - Lock-free
    - For baselining

Task Visibility

- Help-first vs Work-first
  - Some think the distinction is about task visibility to the scheduler
    - They are wrong
  - Help-first does not specify when the spawner gets scheduled again
    - May be similar to having yielded
    - May be prioritized
- Spawn-cache
  - Spawning many tasks is a source of contention
    - Task-lists (e.g. Intel TBB) a way to avoid such contention
    - Compilers can “cheat” by allocating tasks in bulk
    - Implicit task-list management is easy
  - Tasks accumulate until a scheduling operation
    - Blocking & Yielding

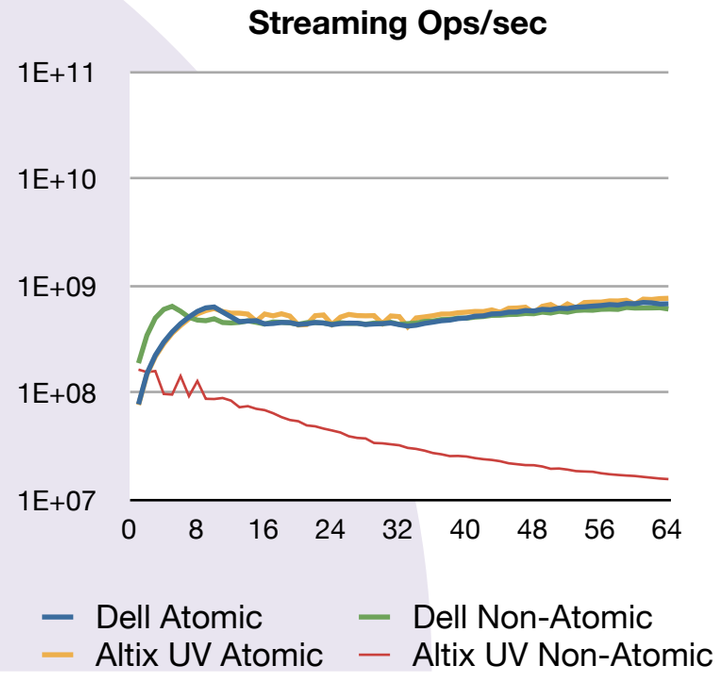
Adaptivity (RCRTool)

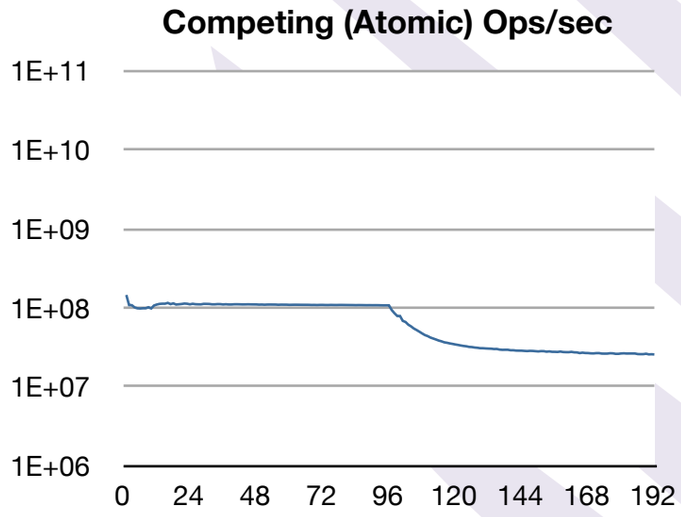
- Shared resource contention is a problem
- Sometimes the cost of contention outweighs the benefit of additional parallelism
- RCRDaemon
  - Performance counters sometimes require root access
  - Monitors the state of shared resources
  - Publishes data via shared memory page
- RCRTool
  - Monitors shared memory page
  - Disables shepherds/workers based on performance data

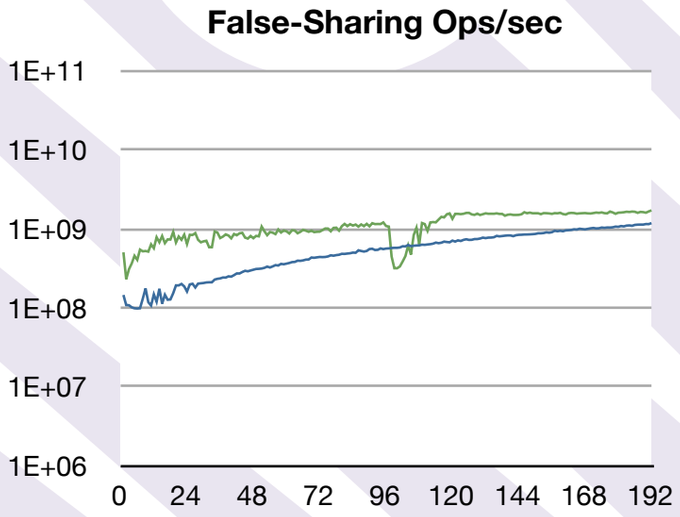
This is highly experimental, but we have had promising results!

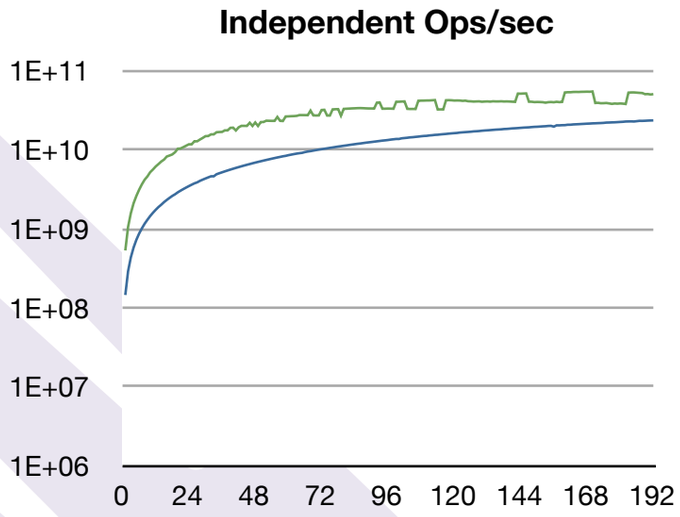
General Purpose Synchronization

- Portable Atomics: Increment and CAS
  - Some platforms harder than others (TILE64)
  - Auto-detection of compiler intrinsics
  - Inlined assembly



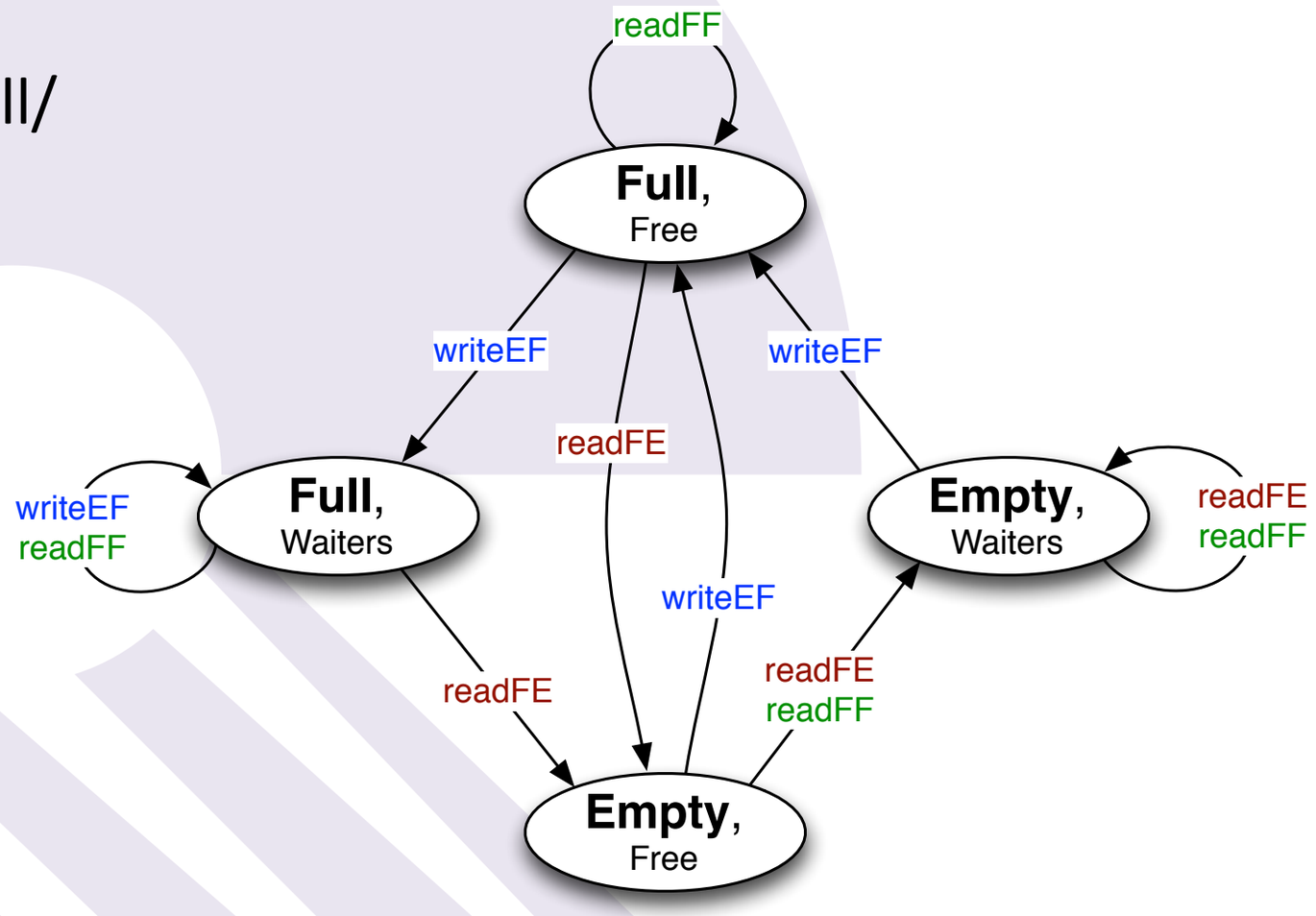






General Purpose Synchronization

- FEBs
  - Every word in memory gets a full/ empty state
  - Reads/writes can:
    - Wait for a precondition state
    - Modify state and data together atomically
  - Previous HW Implementations
    - Tera/Cray MTA, Cray XMT/XMT2
    - MIT Alewife
    - Denlcor HEP
    - Horizon
    - Convey CHOMP



FEB Preconditions

Sandia National Laboratories

- Explicit data dependencies
  - Set of addresses that must be “full” before the task will spawn
  - Task will automatically spawn when all specified addresses are full
- Uses the same infrastructure as arbitrary FEB blocking
  - Simply hooks up a task meta-data struct with no runtime data attached
- Preconditions checked in reverse order specified
  - *Assumed not to flip back to empty*

FEB: Two Implementation Options

Sandia National Laboratories

- Centalized Hash Table
  - Hash key is any aligned 64-bit word address
  - Stores 64-bits of data
  - Presence in hash table means "empty or waiters", absence means "full & no waiters"
  - Blocked threads
    - are enqueued in the hashtable
    - not visible to the scheduler
  - Unsynchronized load/stores are unaffected
  - **Fastest when expecting to block**
- Local State Cache
  - 4-bits of 64-bit word reserved
    - 1 spin-lock bit
    - 1 full/empty state bit
    - 1 waiter/no-waiter state bit
    - 1 reserved for debugging
  - Blocked threads
    - are enqueued in the hashtable
    - not visible to scheduler
  - Unsynchronized load/stores require a bitshift or a mask
  - **Fastest when expecting not to block**

Collective Synchronization

- Collectives
  - Sinc
    - Termination detection
    - Arbitrary reduction operation
    - Flexible membership growth
  - Teams
    - Based on sincs
    - Allow for dependent subteams
- Eurekas
  - Based on team membership

Sincs

- Termination detection
  - First pass: “donecount”
    - Termination means matching in/out counters
    - Atomic increments
      - Potential for contention issues
      - Hardware can help by aggregating increments
  - Second pass: SNZI-style
    - CAS-tree to avoid contention
      - Harder for hardware to accellerate
    - Not yet complete
- Correct use requires announcing new submitters before all previously announced submitters submit
  - Easiest way to guarantee is by requiring that only active (unsubmitted) members can create increase the expect-count

Sincs

- Arbitrary reduction operations
  - Worker-local accumulation buffers
    - Size determined by input
  - May want specialized implementations for common reductions (sum, product, max, min, bag, etc.)
  - Not all members that submit must participate in reduction

Task Teams

- Current Use-cases:
  - Termination detection
    - Basic test of membership
    - Relies on sincs
  - Eureka operations
    - Membership defines “who dies” when the eureka is triggered
- Membership
  - No external team joining (e.g. “spawn into team X”)
  - No multi-membership, except virtually via hierarchy
  - No empty teams - teams are defined by their members
  - Default team: team 0
    - Cannot eureka
- Subteams
  - Existence of a subteam is dependent upon its spawner
  - rephrased: killing the spawner kills the subteam

Eurekas

- Two Use-cases
  - Algorithm “races”
    - Who finds the answer first?
  - Shared data-structure exploration with early-exit
    - “I found the gold nugget!”
    - “This branch is guaranteed uninteresting!”
- Eurekas must support a hierarchy
  - Kill signal propagates down, not up
  - Motivating requirement for subteams
- Eureka operation is **ALLOWED TO BE SLOW**

- Eureka steps:
  1. Use a synchronization operation (e.g. atomic CAS) to pick a “current eureka”
  2. Send signal (SIG_USR1?) to all worker threads
  3. Barrier
  4. All workers stop executing tasks, filter their queue (may nominate just worker 0)
  5. Shepherd 0 will also filter the hash tables for blocked tasks.
  6. Barrier
  7. Back to work!

The Others in the Field

Sandia National Laboratories

	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	✔					✔			✔	✔		

PIM

- Locality is key to performance
  - Qthreads vs just-about-anything-else
- Computational power of a PIM

- State management vs synchronization
  - Not a tradeoff most would expect

Accelerators

- Task granularity & aggregation
- Synchronization model
- Host behavior
- Locality hierarchy extended