ADVANCED PARALLEL COMPUTING
LECTURE 05 - SYNCHRONIZATION 2

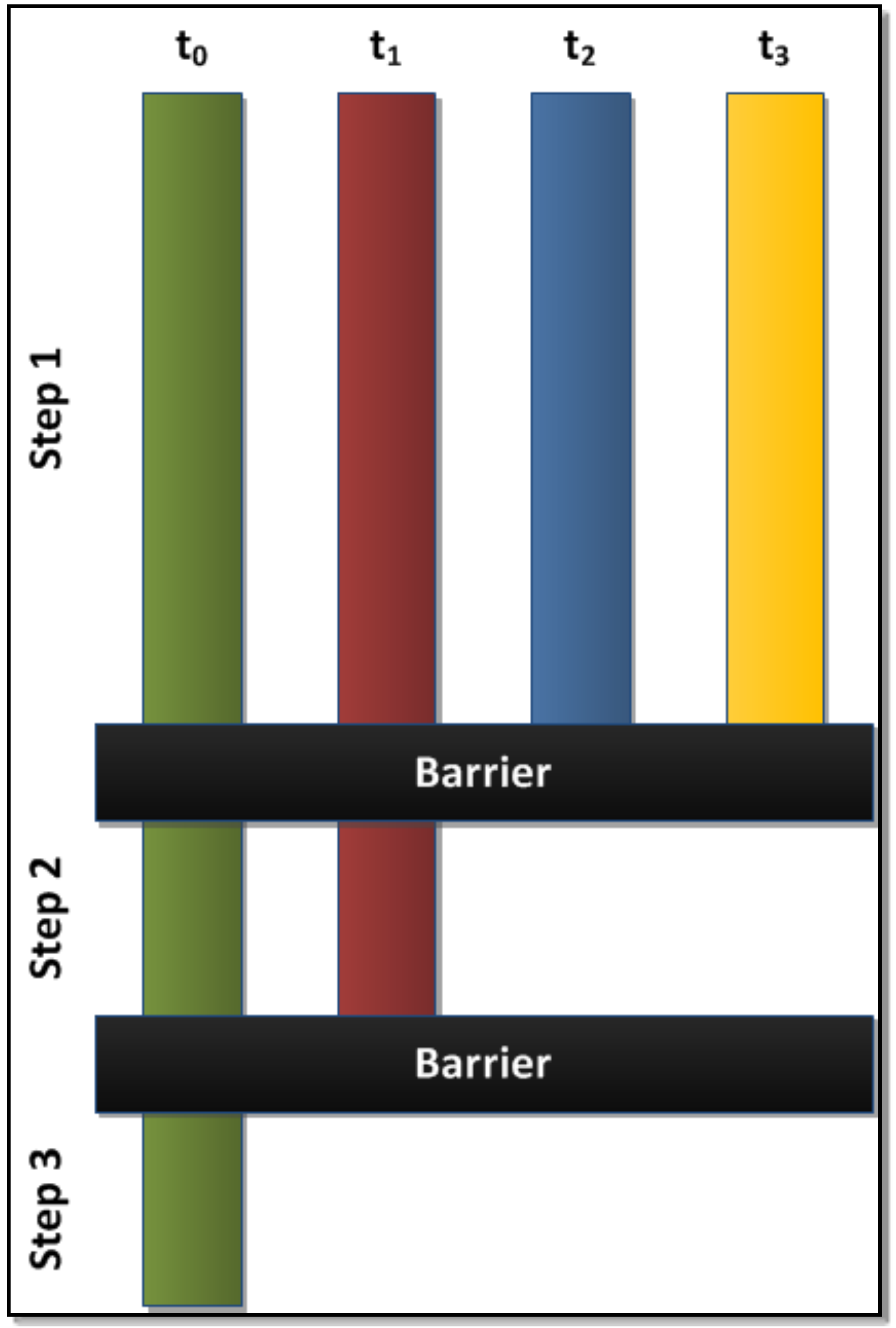
Holger Fröning
holger.froening@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

Some material by Falsafi, Hardavellas, Nowatzyk of EPFL, Northwestern, CMU

BARRIERS

WHY DO WE NEED BARRIERS?

- What is a barrier?
  - Any participating thread has to wait until all threads have reached the barrier
- Why do we need such coarse-grain (collective) synchronization?
  - Any kind of lock-step computation
  - Physics simulation computation
    - Divide up the computation of each time step into N independent pieces
    - Each time step: compute independently, synchronize
  - Essential for BSP-like programming
  - Stencil codes
  - Six-step FFT method



Barrier-based Merge-Sort

BARRIERS VS. LOCKS

Lock

Threads are waiting because they have something to do

A lock master is usually not viable

- No thread/core may be free to take over this part
- Contention would quickly turn the master into a bottleneck

Barrier

Threads are rather waiting because they have finished their work and wait for their external dependencies to be resolved

A barrier master can be viable

- At least one thread is waiting anyway

GLOBAL SYNCHRONIZATION BARRIER

At a barrier, all threads wait until all other threads have reached it

- Strawman implementation (wrong)
- So what's wrong?

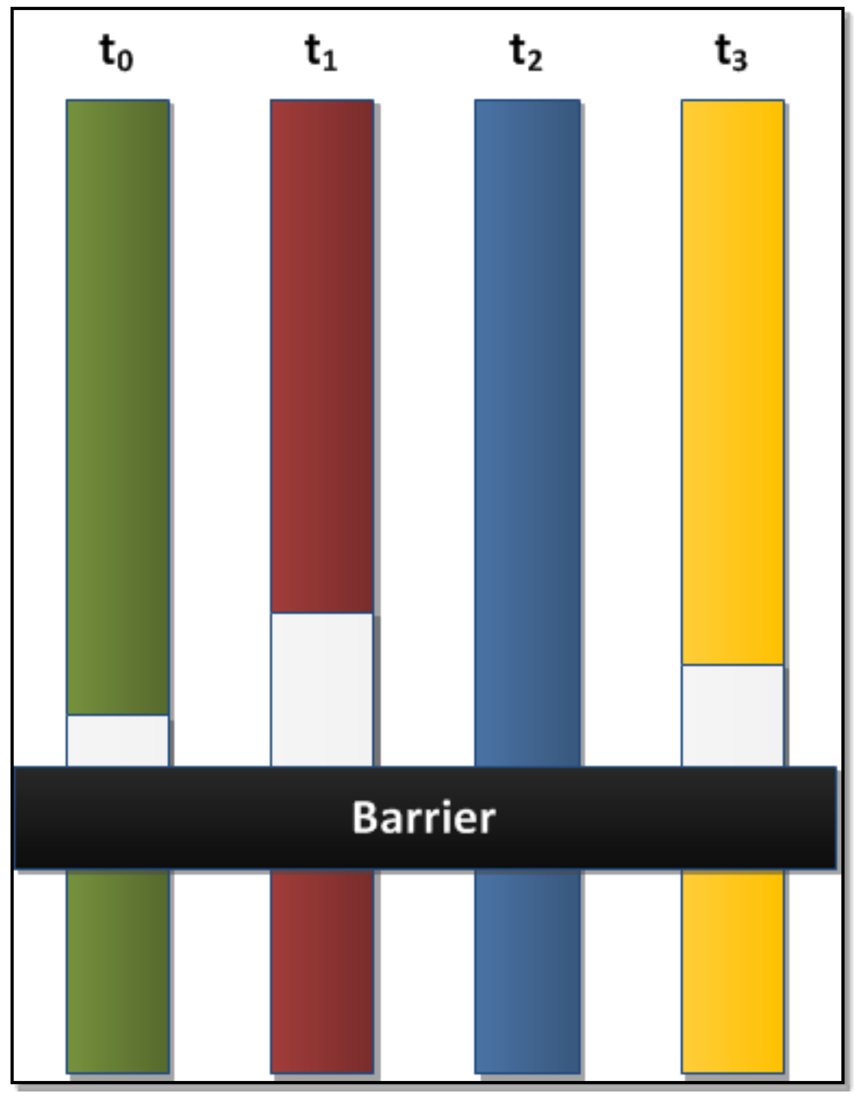
Race condition

- After resetting counter to P, before all threads are notified, another thread could again decrement counter

```
(shared) int count = P;

void strawman_barrier ()
  {
  if ( fetch_and_dec ( &count ) == 1 )
    count = P;
  else
    while ( count != P );
      //spin wait
  }
```

```
...
do_work_1 ();
strawman_barrier ();
do_work_2 ();
strawman_barrier ();
...
```



FUNCTIONALLY CORRECT BARRIER “SENSE-REVERSING”

```
(shared) int count = P;
(shared) bool sense = true;
(local) bool local_sense = true;

void sense_barrier ()
  {
  // each processor toggles its own sense
  local_sense = !local_sense;

  if ( fetch_and_dec ( &count ) == 1 )
    count = P;
    // last processor toggles global sense
    sense = local_sense;
  else
    while ( sense != local_sense ); //spin wait
  }
```

ISSUES WITH CENTRALIZED BARRIERS

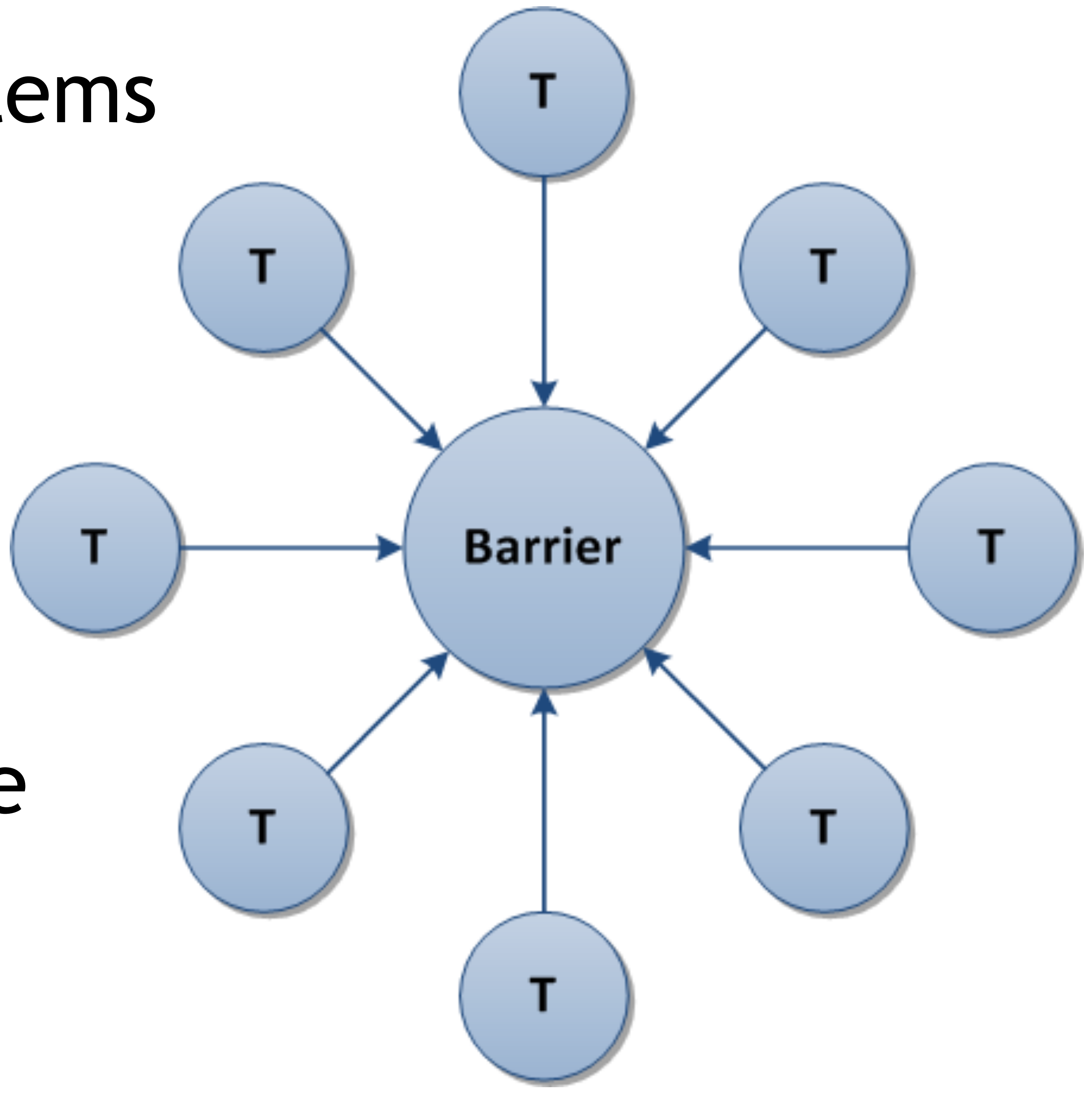
Single counter makes the sense-reversing barrier a centralized barrier

- Highly suitable for cache-coherent shared memory systems
- Cache coherence & polling on sense:
  uniform notification time (hopefully)

Problem with centralized barrier

- All processors must increment the counter
- Each RMW is a serialized coherence action, i.e. a cache miss
- O(n) threads arrive simultaneously, slow for lots of processors

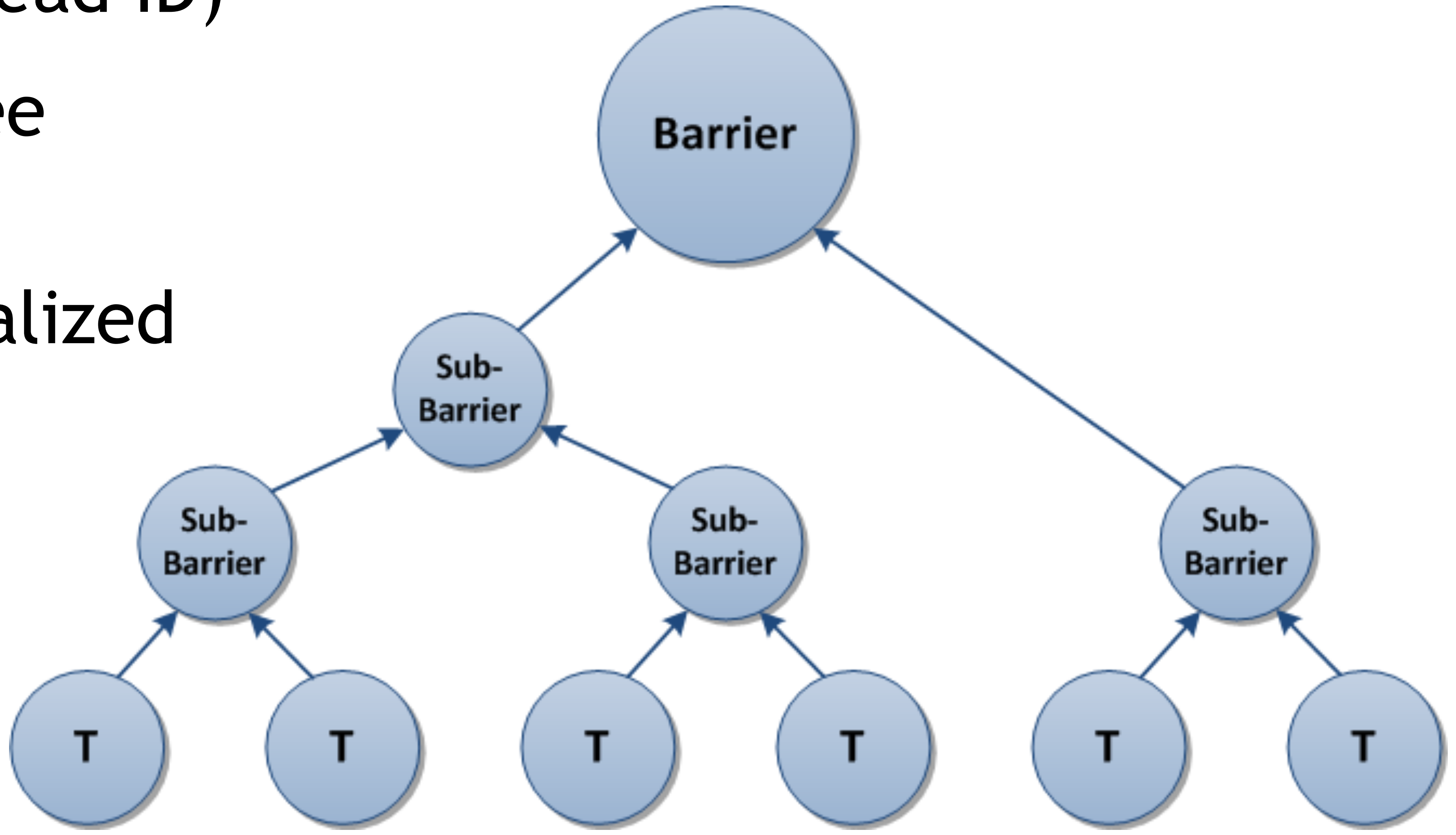
=> Memory contention



COMBINING TREE BARRIER

Combining tree barrier: build a $\log_k(n)$ height tree of counters (one per cache block)

- k = radix
- Each thread coordinates with (k-1) other threads (i.e., by thread ID)
- Last of the k threads coordinates with next higher node in tree (identified by last **atomic** decrement)
- As many coordination addresses are used, misses are not serialized (O (log n) in best case)



Backoff: less traffic but longer delay

Spread memory accesses across multiple (sub-)barriers

=> Reduce contention

- What is faster, decrement a single location or visiting a logarithmic number of barriers?

=> At the cost of increased latency

- Eventually

DISSEMINATION BARRIER

Butterfly barrier: Butterfly-like structure with pairwise signaling based on multiple rounds (k)

- In round k, processor i synchronizes with processor i XOR 2^k

Dissemination barrier: no longer pairwise, instead processor i signals (disseminates) to (i + 2^k) mod P, P=2^k

- No remote spinning

OTHER BARRIER IMPLEMENTATIONS - SW

Separate barrier operation into two phases: arrival and release

Various optimizations

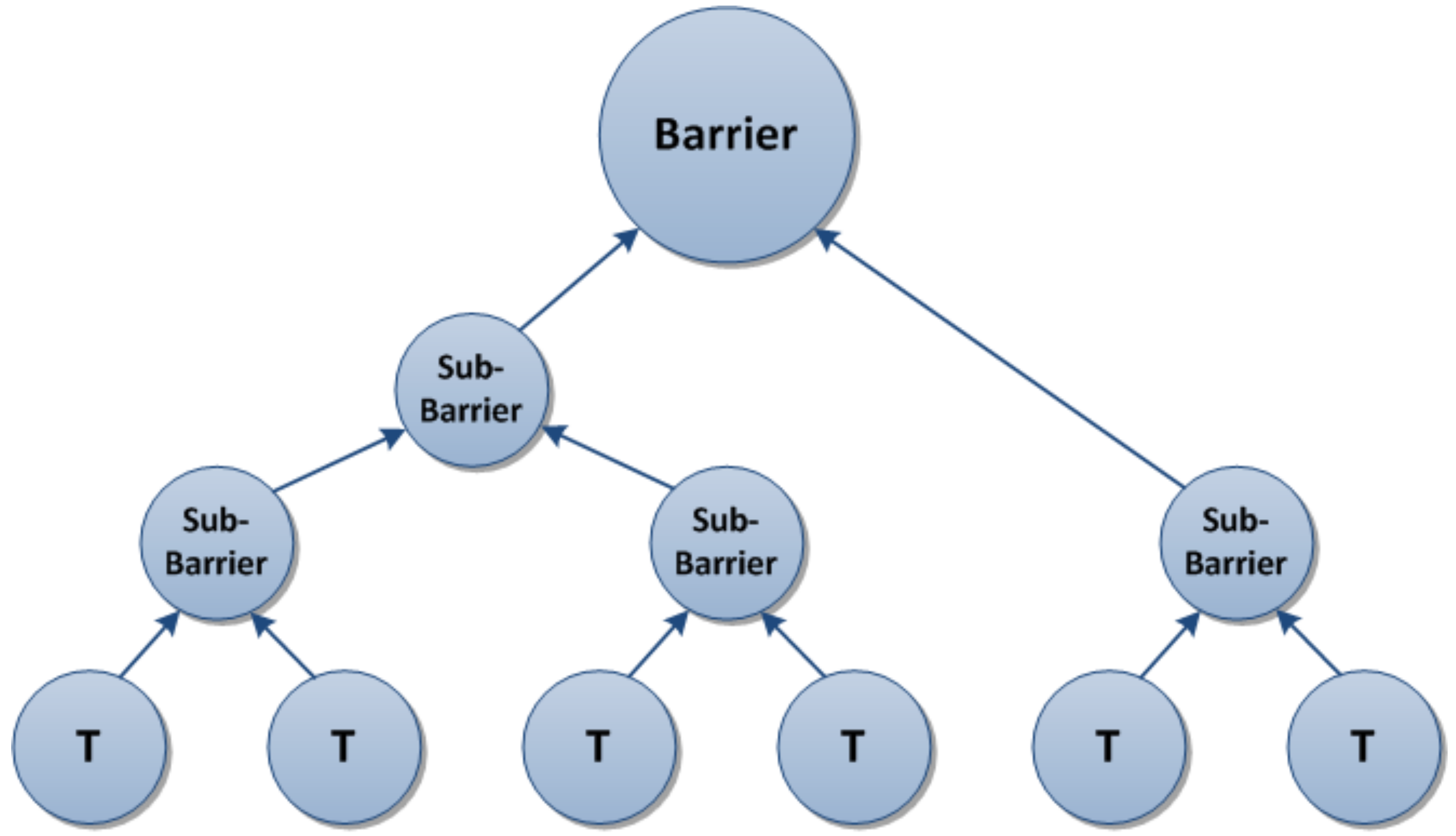
1. (Software) Combining tree

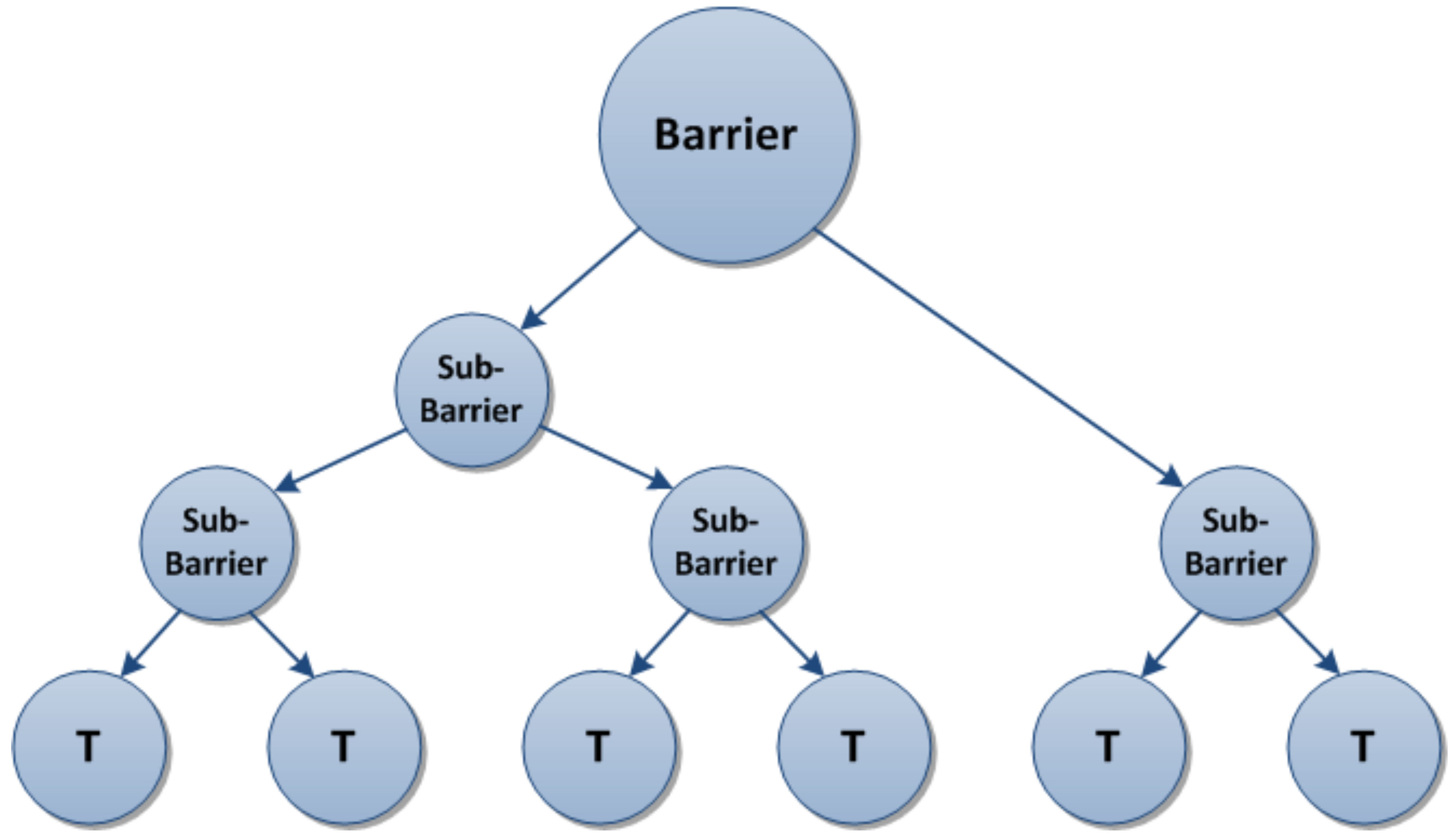
    Different fan-outs, even different for arrival/ release tree

3. Tournament to avoid RMW

    Based on tree, but “winning” processor is determined statically, thus no need for fetch-and-op

And these are only the software implementations





OTHER BARRIER IMPLEMENTATIONS - HW

In principle: AND trees (possibly even hard-wired)

Issues

- No one wants a second dedicated network
- Virtualization: support for more than one concurrent barrier

1. Syncbits associated with each cache block
   - J. Goodman et al., “Efficient Synchronization Primitives for Large- Scale Cache-Coherent Shared-Memory Multiprocessors”, ASPLOS1989
2. E-Registers & synchronization units (32, memory-mapped) (Cray T3E)
   - S.L. Scott, “Synchronization and Communication in the T3E Multiprocessor”, ASPLOS1996
3. Starving cache line requests (postpone responses until barrier condition met)
   - J. Sampson et al., “Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers”, MICRO2006
4. GBarrier for many-core CMPs (G-Line based dedicated network)
   - J. L. Abellán, J. Fernández and M. E. Acacio, "Efficient Hardware Barrier Synchronization in Many-Core CMPs”, TPDS2012

BARRIER PERFORMANCE

Barrier performance on Symmetry (shared-memory, bus-based, cache-based, 80386 system)

Arrival tree

- Tree barrier with a central sense-reversing flag instead of a release tree
- Smaller slope than “counter” as N-1 writes are cheaper than N RMWs

Network transactions

- Dissemination barrier: O (P log P), tournament & tree: O (P), centralized & arrival tree: O(P) if broadcast support, potentially unbounded otherwise

Critical path length

- Assuming network supports parallel transactions
- Centralized: O (P), others: O (log P)

Space

- Centralized: 1, tree: O (P), others: O (P log P)

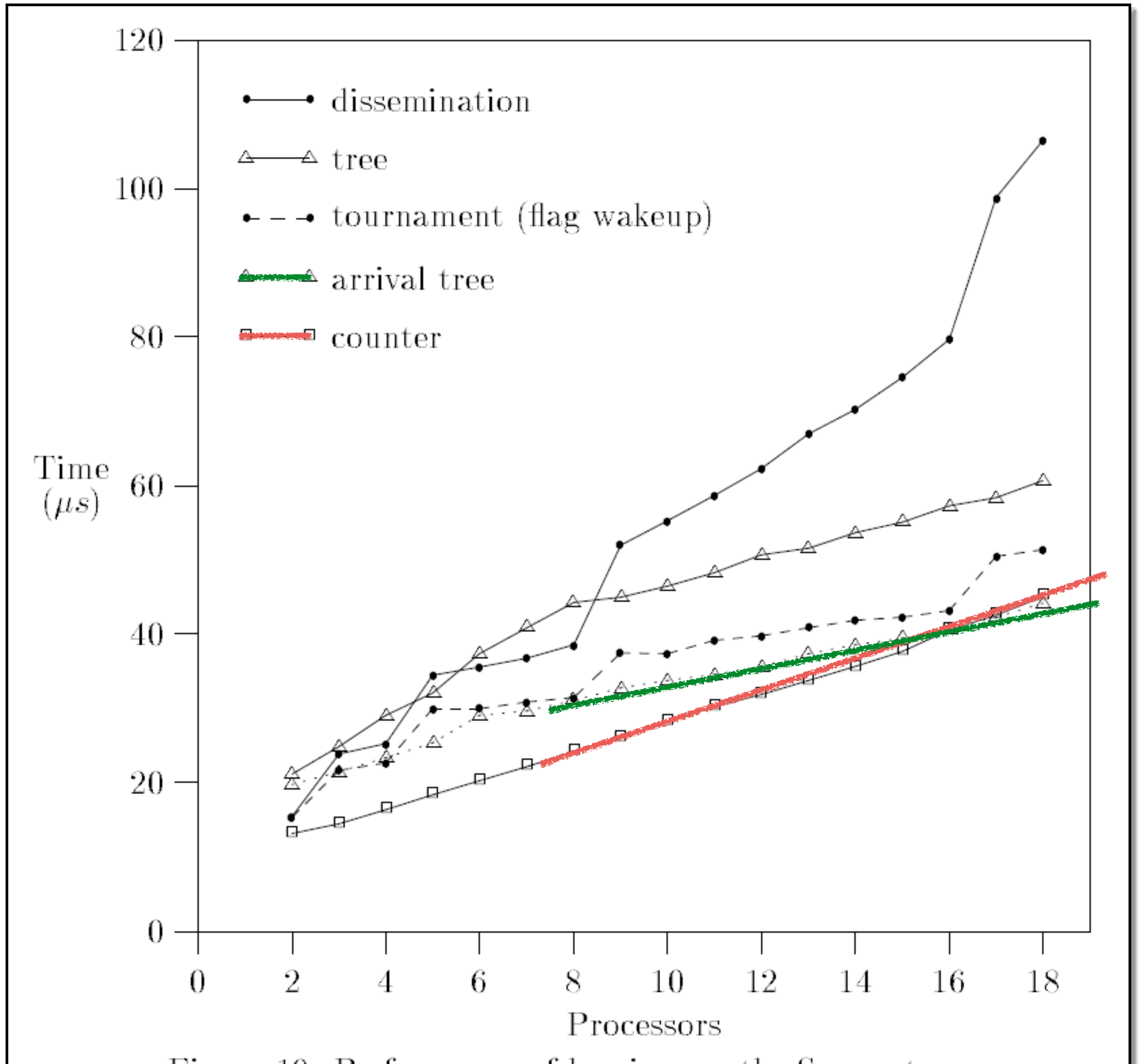
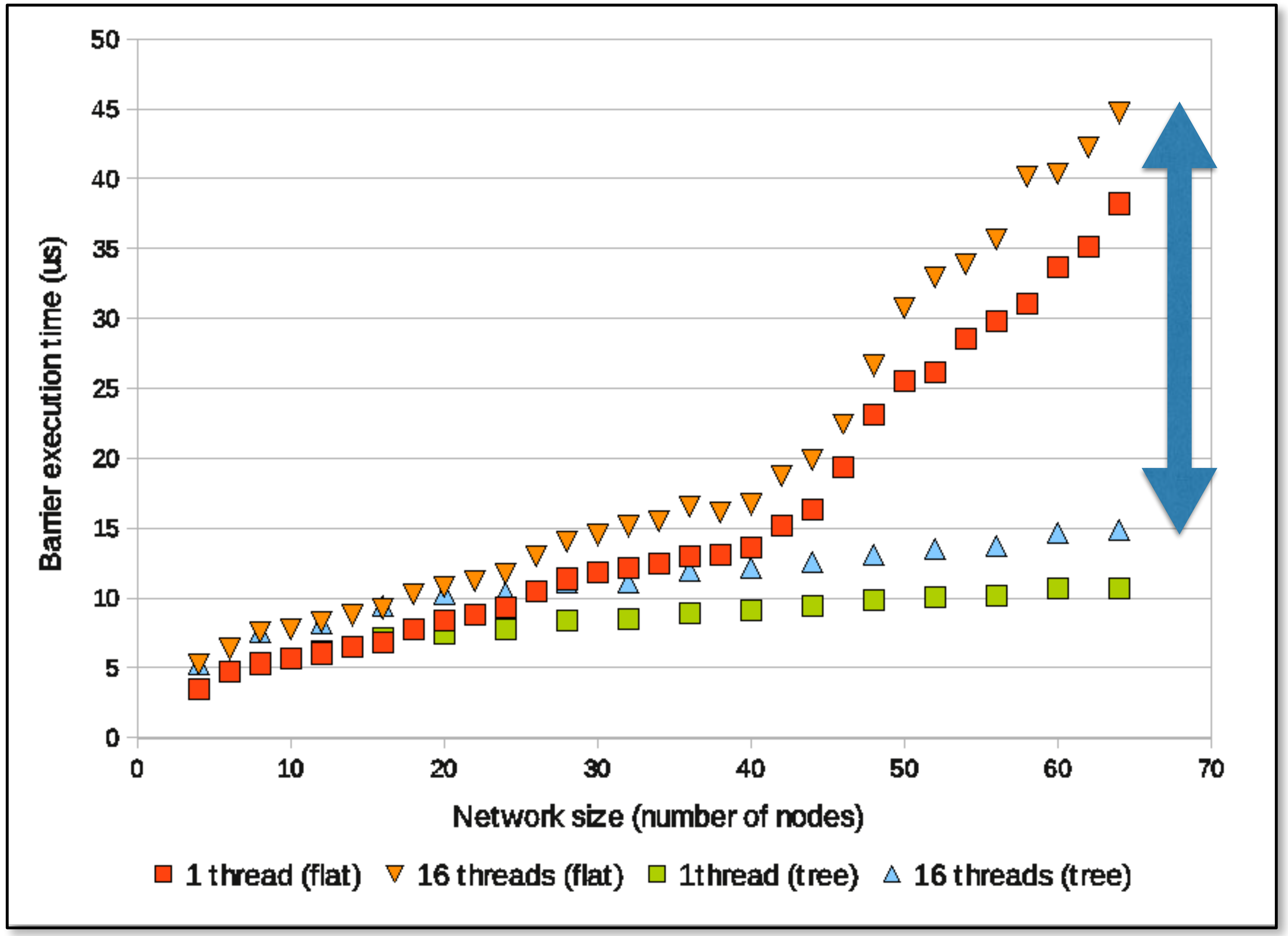


Figure 10: Performance of barriers on the Symmetry.

Mellor-Crummey, Scott, TOCS 1991

BARRIER PERFORMANCE

- Barrier over distributed shared memory (MEMSCALE)
  - No global coherence but local coherence domains
  - Support for remote loads and stores
  - No RMW
- Tree-based: scalability, locality
- Tournament: arrays of flags instead of RMWs
- Dissemination: push instead of pull
- => 1k threads in 15usec
  - 64P @IB: min. 20us



Holger Fröning, Alexander Giese, Hector Montaner, Federico Silla, Jose Duato, Highly Scalable Barriers for Future High-Performance Computing Clusters, 18th annual IEEE International Conference on High Performance Computing (HiPC 2011), 2011, Bangalore, India.

OTHER ASPECTS

Relaxations

Do you really need to synchronize?

Stencil codes with halo exchange: Probably at the cost of an increase in iterations

Training of deep learning: asynchronous weight exchange

Overlap

Hide barrier latency with communication

Which model does that?

Jitter

Large-scale SPMD programs might suffer from varying state
=> increased synchronization time

SUMMARY

Spin Lock

Avoid “Coherence Storms”

Make use of atomic operations if available

- Preferable Compare-and-Swap, or Fetch-and-Store
- Remember the Bakery algorithm as SW alternative

List-based queuing locks usually outperform any other

- In particular true for scalability

HW like QOLB: ROI too low

Barriers

Avoid “Coherence Storms”

Avoiding atomics can be beneficial

Use combining tree to minimize the critical path length, and dissemination to leverage locality effects

- Treat arrival and release structures differently

HW structures

- ROI too low for shared memory
- Potentially reasonable for message passing

PROJECT WORK PROPOSALS

CONSTRUCTION AREA

NVIDIA Fermi-class GPU

- 2 per server, 8 servers

NVIDIA Kepler-class GPU

- 16 per server (1x)
- 1 per server (8x)
- Embedded plattform (Tegra TK1, TX1)

NVIDIA Pascal-class GPU

- 2 per server (1x)

ARM Mali GPU - currently defunct :(

- Embedded plattform

ARM CPU

- Tegra TK1, TX1, TX2, Xavier
- A couple more…

Micron PicoComputing System

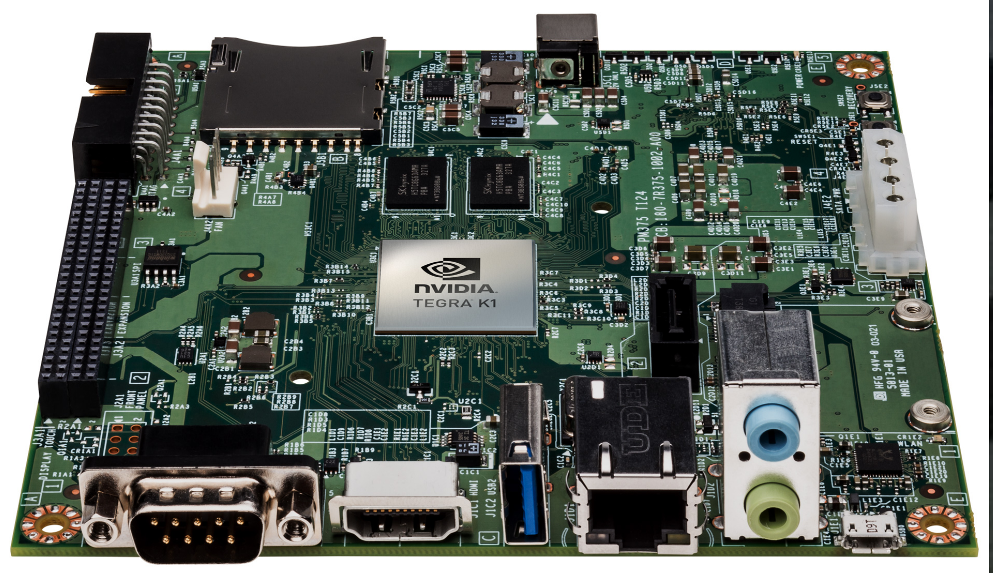
- OpenCL-capable FPGA+HMC

Intel X86

- Sandy Bridge (12 core, 8x)
- Ivy Bridge (16 core, 1x)

Intel MIC

- Possibly, but have to check...



1. ANALYZING DYNAMIC NUMA EFFECTS: LOAD VERSUS LATENCY

Static NUMA effects: memory access latency variations for an unloaded system

Dynamic NUMA effects: memory access latency variations for a loaded vs unloaded system

- GEMM as a simple background load

Method: measure round-trip latency or throughput (OPs/second) with and without background load

Architectures

- Multi-GPU
- x86
- ARM

Goal: understand where this is an issue and quantify the impact

2. IMPLEMENTING A SCALABLE DATA ANALYSIS FOR EXASCALE APPLICATIONS

Repository: https://portal.nersc.gov/project/CAL/doe-miniapps.htm

dumpi tools: https://github.com/sstsimulator/sst-dumpi

Problem: more data than we can analyze in reasonable time

- Python and bash scripts
- Traces for MPI Exascale proxy applications: ~1/2TB analyzed, >25TB left

Open questions

- Do we need ordering? Find cases in the traces where two concurrent messages, identically named (source, destination, tag, communicator), are received explicitly in order (named receive)
- What is the impact of matching on overall performance? - Calculate time for all matches as ratio of overall execution time.

Benjamin Klenk, Holger Fröning, An Overview of MPI Characteristics of Exascale Proxy Applications, International Supercomputer Conference (ISC), June 18-22, 2017, Frankfurt, Germany.

3. EXPLORING UNIFIED MEMORY: PERFORMANCE, ENERGY AND (MAYBE) SCHEDULING IMPLICATIONS

"Downgrade" some CUDA SDK apps to Unified Memory

- Replace cudaMalloc + cudaMemcpy with cudaMallocManaged

Explore performance and energy on NVIDIA Pascal GPUs

- Option: Interact with some multi-GPU guys

Design micro-benchmarks that highlight performance and fallacies

Optionally: prefetch instructions

Goal: understanding + micro-benchmarks

4. IMPACT OF MATRIX MULTIPLY OPTIMIZATIONS ON PORTABILITY

Conventional wisdom: optimizations hurt portability

Approach

- Implement different optimizations for an Open-CL/CUDA matrix multiply
- Explore their impact on different processors

Optimizations

- Compute multiple output values per thread, reducing shared memory pressure
- Address bank conflicts in shared memory
- Vectorize shared memory ops by using compound data types like float2/float4
- Double buffering by overlapping shared memory load for set1 while computing set2

Architectures

- NVIDIA {Fermi, Kepler, Pascal}, embedded-class & server-class
- x86 CPU
- ARM CPU
- ARM GPU Mali? (currently defunct)

5. OPTIMIZE REDUCE-AND-SCALE

Reduce-and-scale concept

- http://www.ziti.uni-heidelberg.de/ziti/uploads/ce_group/2018-ECML.pdf
- https://openreview.net/forum?id=HylDpoActX
- Complexity of DNN inference reduced, using quantization and pruning
- Many additions + only few multiplications instead of many multiply-accumulates (MACs)
- Impressive accuracy results (>= Microsoft's LQNet)
- Results for time missing

Goal

- Implement & optimize reduce-and-scale for ARM (Cortex-A)
- Be fast

6. VALIDATION OF POWER MODELS

- Common power models include NVML, RAPL
    - Accepted by community to be accurate
    - Validation is missing
- Goal: validate power by wall-plug power measurement
    - Using workload spectrum, not a single workload
- Choose suitable platform (x86, GPU, ARM?)

7. PREDICT BLAS PERFORMANCE

Choose processor (x86, ARM, GPU)

Obtain ground truth, understand behavior (size, aspect ratio, architecture)

Create & validate model

Explore model portability

8. PREDICT MEMORY ACCESS PATTERNS

CUDA memory tracer to enumerate all memory accesses

- https://github.com/UniHD-CEG/cuda-memtrace
- http://www.ziti.uni-heidelberg.de/ziti/uploads/ce_group/2019-GPGPU.pdf

Can we predict accurately future memory accesses?

- memtracer with warp ID
- Select ground truth (data available); optional: obtain ground truth for custom applications
- Create and validate model for predictions
- Predictions might be addresses, volume, etc.

NOTES

Make up your mind - we’re open for proposals!

Think about aligning this project assignment with future work

- Seminar talk, project work, MSc thesis