Callisto-RTS: Fine-Grain Parallel Loops

Tim Harris (Oracle Labs), Stefan Kaestle (ETH Zurich)

ATC 2015

Callisto-RTS

- OpenMP-like runtime system for (NUMA) shared memory machines
- Aims to scale better than OpenMP, while requiring less manual tuning
- 'Automatic' handling of nested loops
- Scales well to very small block sizes (1K cycles)

Implementation – API

"Our initial workloads are graph analytics algorithms generated by a compiler from the Green-Marl DSL [12]. Therefore, while we aim for the syntax to be reasonably clear, our main goal is performance."

Implementation – API

"Our initial workloads are graph analytics algorithms generated by a compiler from the Green-Marl DSL [12]. Therefore, while we aim for the syntax to be reasonably clear, our main goal is performance."

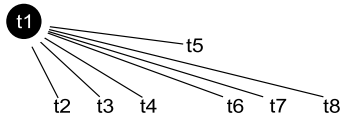
```
struct example_1 {
  atomic<int> total {0};
  void work(int idx) {
    total += idx;
    // Atomic add
} } e1;

parallel_for<example_1, int>(e1, 0, 10);
cout << e1.total;
```

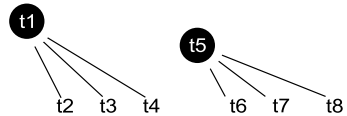
Implemenation – API 2

- Adding functions `void fork(thread_data_t &)` and `void join(thread_data_t &)` allows thread-local accumulation.
- They considered C++ lambdas, but "performance using current compilers appears to depend a great deal on the behavior of optimization heuristics."
- Each parallel loop ends with an implicit barrier (so preemptions are a problem).
- Nested parallized loops need an additional loop level, with 0 being the innermost loop.

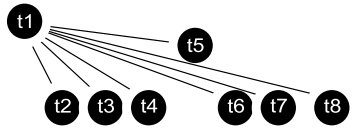
Implementation



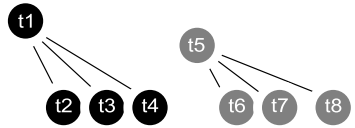
(a) Thread t1 executes sequential code. Other threads wait for work.



(c) Thread t1 enters a loop at level 1. Thread t1 and t5 participate. Threads t6–t8 now wait for work from t5.



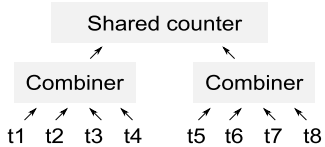
(b) Thread t1 enters a loop at level 0. All threads participate in the loop.



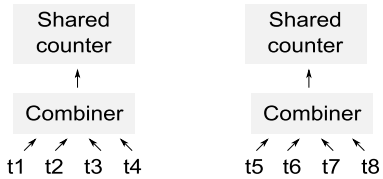
(d) Threads t1 and t5 enter loops at level 0, threads participate in the respective loops.

- Threads are organized in a tree, with a fixed level for each thread.
- A thread at level n becomes a *leader* when it hits a loop with level $k \leqslant n$.
- A *follower* (= not a leader) at level n executes iterations in a loop when its leader hits a loop at level $k \leqslant n$.

Work stealing



(a) Distribution during a level-0 loop led by t_1 in which all threads participate, using separate request combiners on each socket.



(b) Distribution during a level-0 loop led by t_1 (left) and by t_5 (right).

Work distributors, that are combined in a hierarchy:

Shared counter Single iteration counter, modified with atomic instruction.

Distributed counters Work is evenly distributed among 'stripes', threads complete work in own stripe before moving to others.

Request combiner Aggregate multiple requests for work, and forward to higher level.

Request combining

```
my_slot->start = REQ;

while (1) {
  if (!trylock(&combiner->lock)) {
    while (is_locked(&combiner->lock)) ;
  } else {
    // collect requests from other threads,
    // issue aggregated request,
    // distribute work
    unlock(&combiner->lock);
  }

  if (my_slot->start != REQ) {
    return (my_slot->start, my_slot->end);
  }
}
```

Evaluation – Hardware

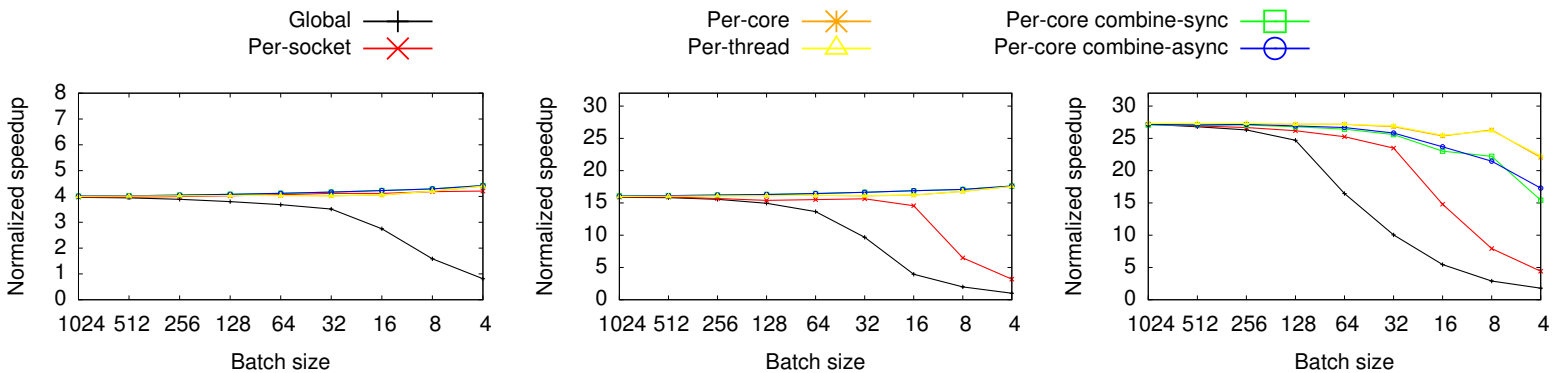
2-socket Xeon E5-2650 IvyBridge

- 8 cores per socket
- L1 and L2 per core
- 2 hardware threads per core

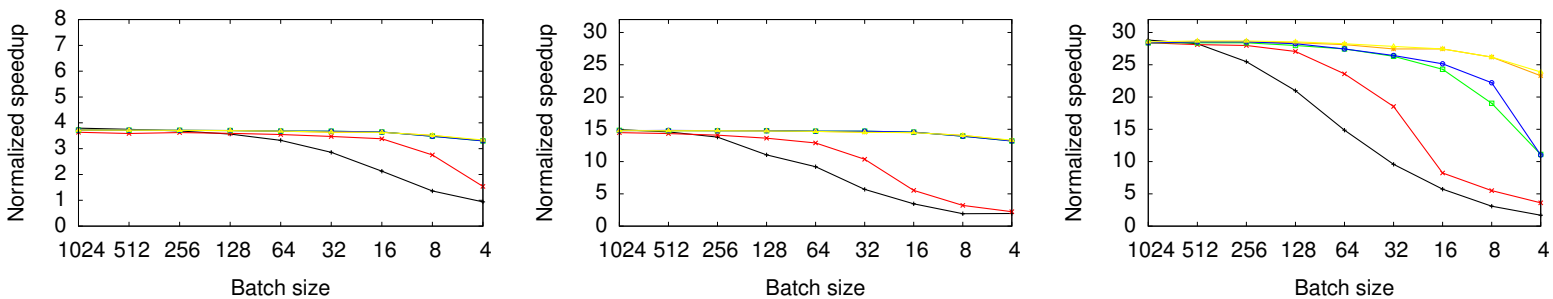
8-socket SPARC T5

- 16 cores per socket
- L1 and L2 per core
- 8 hardware threads per core

Evaluation – Microbenchmarks 1

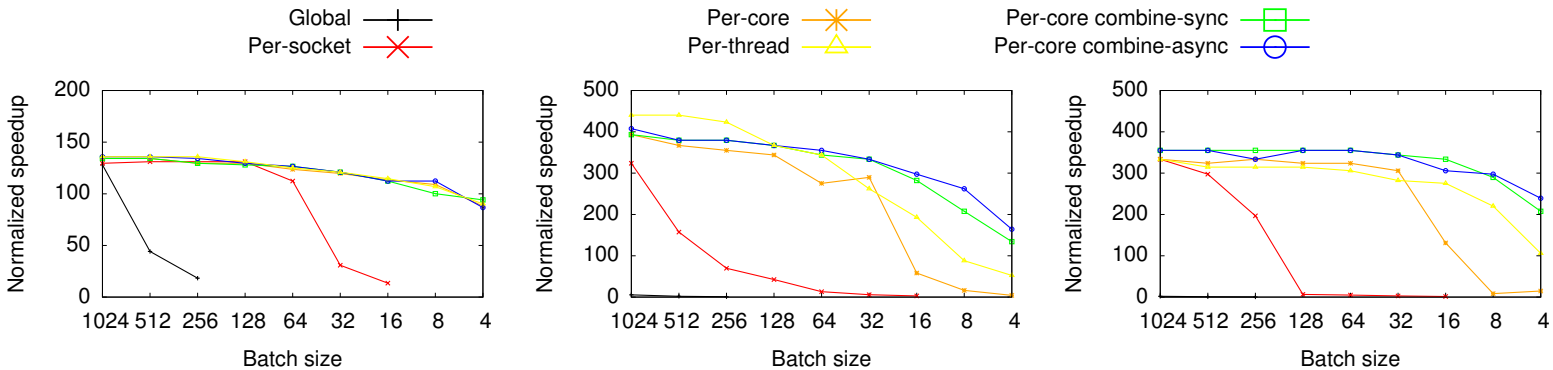


Even distribution: 2-socket Xeon (X4-2), 4 threads (left), 16 threads (center), 32 threads (right)

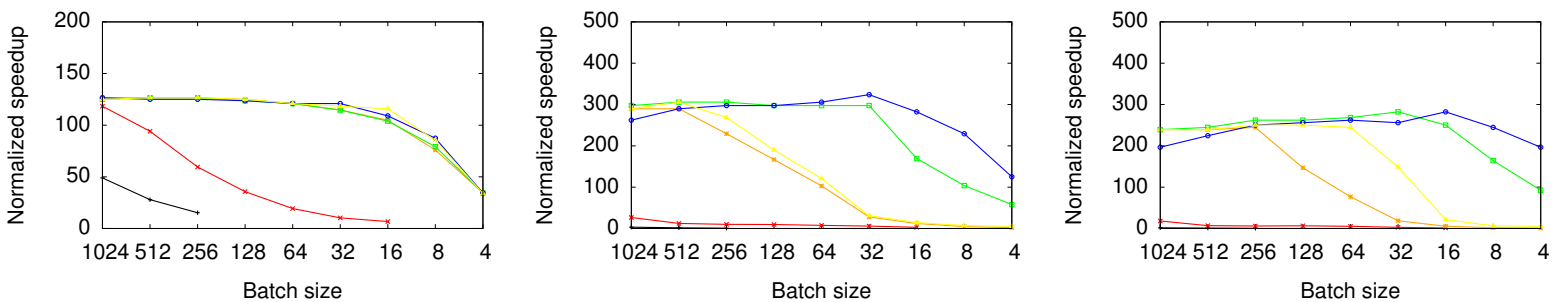


Skewed distribution: 2-socket Xeon (X4-2), 4 threads (left), 16 threads (center), 32 threads (right)

Evaluation – Microbenchmarks 2

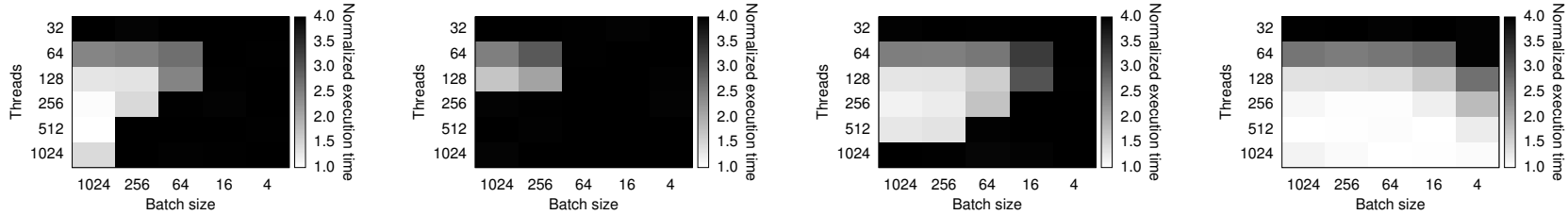


Even distribution: 8-socket T5 (T5-8), 128 threads (left), 512 threads (center), 1024 threads (right)

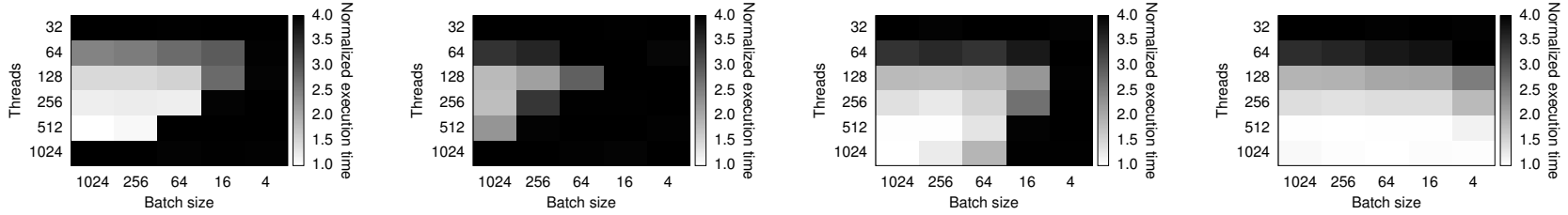


Skewed distribution: 8-socket T5 (T5-8), 128 threads (left), 512 threads (center), 1024 threads (right)

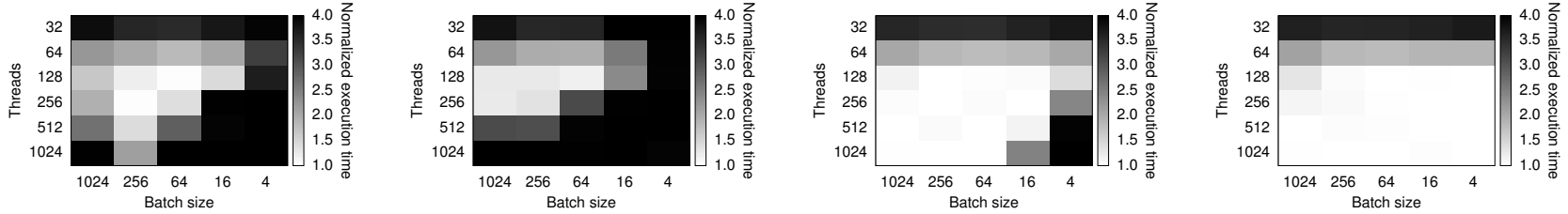
Evaluation – Graph Algorithms



T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), PageRank – LiveJournal. The best OpenMP execution took 0.26s (512 threads, 1024 batch size).

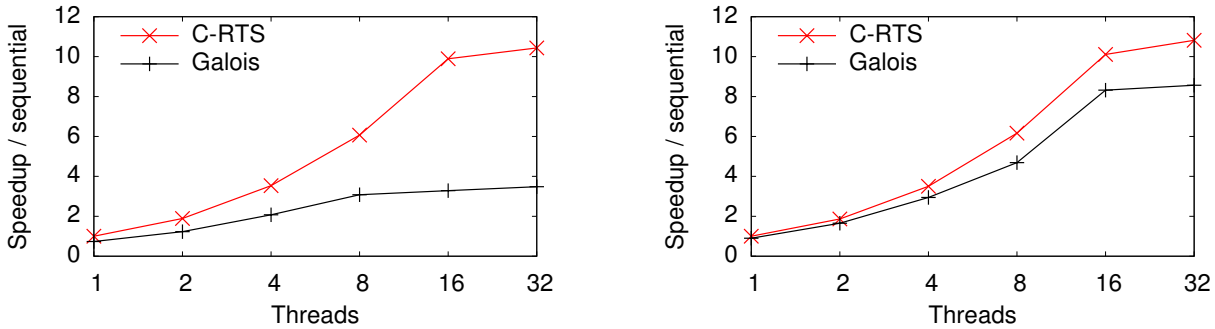


T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), PageRank – Twitter. The best OpenMP execution took 6.0s (512 threads, 1024 batch size).

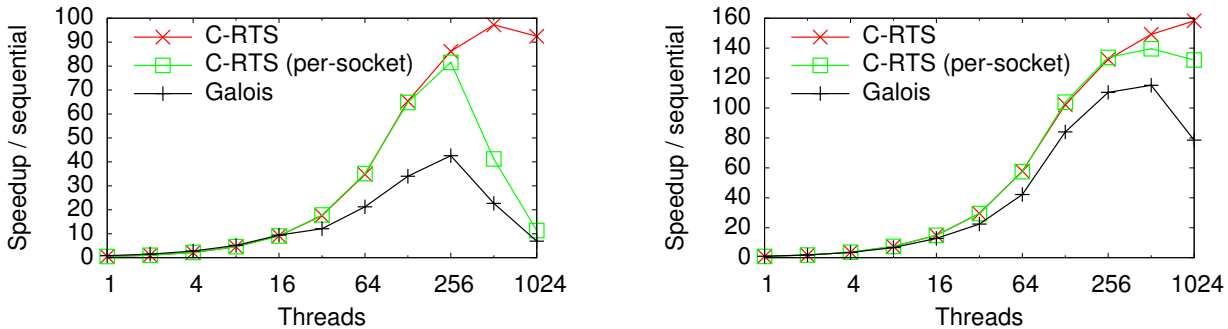


T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), Triangle counting – LiveJournal. The best OpenMP execution took 0.21s (256 threads, 256 batch size).

Evaluation – Comparison to Galois



2-socket Xeon (X4-2), PageRank with LiveJournal input (left) and Twitter input (right).



8-socket T5 (T5-8), PageRank with LiveJournal input (left) and Twitter input (right).

Galios uses per-socket queues to dispatch work blocks, which worker threads draw from.

Evaluation – Nested parallelism

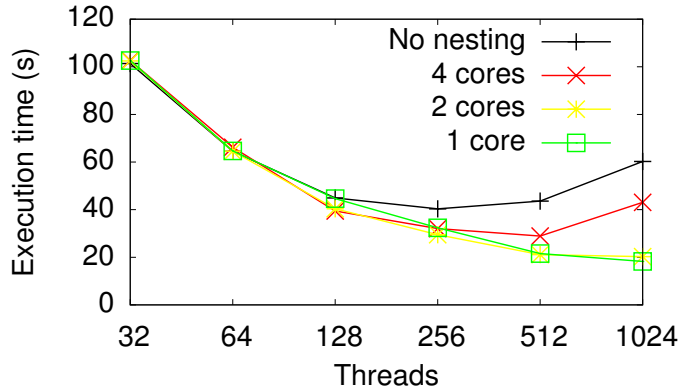


Figure 7: Betweenness centrality using nested parallelism at different levels.

- With 128 threads + flat parallelism: 9.8% misses in L2-D Cache
- With 1024 threads + flat parallelism: 29%
- With 2014 threads + 10.8%

Discussion

- What to do when there are other processes? Busy waiting and barriers don't really work then.
- What is the relation to Callisto?
- What is the problem with C++ lambdas?