Callisto-RTS: Fine-Grain Parallel Loops

Tim Harris, Oracle Labs
Stefan Kaestle, ETH Zurich

7 July 2015
The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.
Setting: parallel loops on shared-memory machines

for (uint64_t node = 0; node < G.num_nodes(); node++) {
    double val = 0.0;
    for (edge_t w_idx = G.r_begin[node];
        w_idx < G.r_begin[node+1];
        w_idx ++) {
        node_t w = G.r_node_idx[w_idx];
        val += G_pg_rank[w] / (G.begin[w+1] - G.begin[w]);
    }
    G_pg_rank_nxt[node] = (1 - d) / N + d * val;
}
Setting: parallel loops on shared-memory machines

```
parallel_for<
    uint64_t>
    (0, G.num_nodes(),
     [&](uint64_t node) {
        double val = 0.0;
        for (edge_t w_idx = G.r_begin[node];
             w_idx < G.r_begin[node+1];
             w_idx ++) {
            node_t w = G.r_node_idx [w_idx];
            val += G_pg_rank[w] / (G.begin[w+1] - G.begin[w]);
        }
        G_pg_rank_nxt[node] = (1 - d) / N + d * val;
    });
```
Setting: parallel loops on shared-memory machines

```cpp
parallel_for<uint64_t>(0, G.num_nodes(),
 [&](uint64_t node) {
     double val = 0.0;
     for (edge_t w_idx = G.r_begin[node];
         w_idx < G.r_begin[node+1];
         w_idx++) {
         node_t w = G.r_node_idx[w_idx];
         val += G_pg_rank[w] / (G.begin[w+1] - G.begin[w]);
     }
     G_pg_rank_nxt[node] = (1 - d) / N + d * val;
 });
```

Loop index type and bounds

Loop body (C++ lambda)
Batch size / load imbalance trade-off

**Diagram Description:**
- **Iteration number** is plotted on the x-axis.
- **Iteration execution time** is plotted on the y-axis.
- The diagram shows a fixed amount of work in each iteration.

**Text:**
Fixed amount of work in each iteration.
Batch size / load imbalance trade-off

Divide iteration space evenly between threads and get good load balancing.

Fixed amount of work in each iteration.
Batch size / load imbalance trade-off

(Actual data – #out-edges of the top 1000 nodes in the SNAP Twitter dataset)
Batch size / load imbalance trade-off

Divide into large batches
- Reduce contention distributing work
- Risk load imbalance

Divide into small batches
- Increase contention distributing work
- Achieve better load balance
Batch size / load imbalance trade-off

Typically, choose manually – but getting this right depends on (1) algorithm, (2) machine, (3) data

Divide into large batches
Reduce contention distributing work
Risk load imbalance

Divide into small batches
Increase contention distributing work
Achieve better load balance
Example performance

OpenMP static & dynamic loops

8-socket SPARC T5
16 cores per socket
8 h/w threads per core

PageRank
SNAP LiveJournal data set

Best performance: 0.26s
Batch size / load imbalance trade-off

Typically, choose manually – but getting this right depends on (1) algorithm, (2) machine, (3) data

<table>
<thead>
<tr>
<th>Divide into large batches</th>
<th>Divide into small batches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce contention</td>
<td>Increase contention</td>
</tr>
<tr>
<td>Risk load imbalance</td>
<td>Distributing work</td>
</tr>
<tr>
<td></td>
<td>Achieve better load balance</td>
</tr>
</tbody>
</table>

Our approach: support efficient small batches
Overview

1. Request combining
2. Asynchronous work requests
3. Non-work-conserving nested loops
4. Results
Overview

1. Request combining
2. Asynchronous work requests
3. Non-work-conserving nested loops
4. Results
Approach, consider a loop 0..65536, batch size 8
Approach, consider a loop 0..65536, batch size 8

8 sockets

16 cores per socket
Approach, consider a loop 0..65536, batch size 8

- 8 sockets
- 16 cores per socket
- 8 h/w contexts per core
Approach, consider a loop 0..65536, batch size 8

- 8 sockets
- Distribute iterations at start of loop down to per-core counters
- 16 cores per socket
- 8 h/w contexts per core
Approach, consider a loop 0..65536, batch size 8

- 8 sockets
- Distribute iterations at start of loop down to per-core counters
- 16 cores per socket
- 8 h/w contexts per core
Approach, consider a loop 0..65536, batch size 8

- 8 sockets
- Distribute iterations at start of loop down to per-core counters
- 16 cores per socket
- Aggregate requests upwards within a core
- 8 h/w contexts per core

Per-core lock
Per-thread request flags
Approach, consider a loop 0..65536, batch size 8

0..512
512..1024

8 sockets
Distribute iterations at start of loop down to per-core counters
16 cores per socket
Aggregate requests upwards within a core
8 h/w contexts per core

Per-core lock
Approach, consider a loop 0..65536, batch size 8

- Distribute iterations at start of loop down to per-core counters
- Aggregate requests upwards within a core
- 8 sockets
- 16 cores per socket
- 8 h/w contexts per core
- Per-core lock
Approach, consider a loop 0..65536, batch size 8

- 8 sockets
- Distribute iterations at start of loop down to per-core counters
- 16 cores per socket
- Aggregate requests upwards within a core
- 8 h/w contexts per core

Per-core lock
Approach, consider a loop 0..65536, batch size 8

- 8 sockets
- Distribute iterations at start of loop down to per-core counters
- 16 cores per socket
- Aggregate requests upwards within a core
- 8 h/w contexts per core
- Per-core lock
Approach, consider a loop 0..65536, batch size 8

- 8 sockets
- Distribute iterations at start of loop down to per-core counters
- 16 cores per socket
- Aggregate requests upwards within a core
- 8 h/w contexts per core
- Per-core lock

0..512
512..1024

3*8
Approach, consider a loop 0..65536, batch size 8

- 8 sockets
- Distribute iterations at start of loop down to per-core counters
- 16 cores per socket
- Aggregate requests upwards within a core
- 8 h/w contexts per core

Per-core lock
Approach, consider a loop 0..65536, batch size 8

- 8 sockets
- Distribute iterations at start of loop down to per-core counters
- 16 cores per socket
- Aggregate requests upwards within a core
- 8 h/w contexts per core

0..8 8..16 16..24

Per-core lock
Hierarchical distribution with request combining

- Combining implemented over flags in a single line in the shared L1 D$
- On TSO: no memory fences
- Synchronization remains core-local if work is evenly distributed
- Threads waiting for combining can use mwait
Overview

1. Request combining
2. Asynchronous work requests
3. Non-work-conserving nested loops
4. Results
Asynchronous combining of requests

Synchronous

- Execute batch
- Set flag
- Wait for / fetch next batch
Asynchronous combining of requests

Intuition: the time taken to execute the current batch provides an opportunity for other cores to service our request without us needing to wait.
Overview

1. Request combining
2. Asynchronous work requests
3. Non-work-conserving nested loops
4. Results
Nested loops

• Abundant parallelism, why use nesting?
Nested loops

• Abundant parallelism, why use nesting?
• Contention between iterations of an outer loop
• E.g., betweenness-centrality:
  – Iterate over vertices
  – BFS traversal from each vertex (plus additional work)
Nested loops

• Abundant parallelism, why use nesting?
• Contention between iterations of an outer loop
• E.g., betweenness-centrality:
  – Iterate over vertices
  – BFS traversal from each vertex (plus additional work)

Better cache locality within each traversal than between (unrelated) traversals
Nested loops

• Abundant parallelism, why use nesting?
• Contention between iterations of an outer loop
• E.g., betweenness-centrality:
  – Iterate over vertices
  – BFS traversal from each vertex (plus additional work)

Better cache locality within each traversal than between (unrelated) traversals
Run at most one of these per L2 D$
Nested loops
Controlling thread -&gt; loop allocation

• Number loops “inside out”
  – Level 0 =&gt; innermost
  – Level 1 =&gt; may contain a level-0 loop

• Each thread also has a level
  – It will execute iterations &lt;= its own level
  – Level 0 thread: only executes inner-most loop iterations
  – …
Nested loops
Nested loops: non-nested level 0 – all threads participate
Nested loops: outer (level 1) – just 1+5 participate
Nested loops: inner (level 0) – help respective leaders
Overview

1. Request combining
2. Asynchronous work requests
3. Non-work-conserving nested loops
4. Results
Microbenchmark results
SPARC T5-8, 1024 threads

Per-core + asynchronous combining (blue)
Per-core + synchronous combining (green)

Even work

Batch size

0 100 200 300 400 500

1024 512 256 64 32 16 8 4

(Approx 1k cycles)

Per-socket counters
Per-core counters
Per-thread counters
Microbenchmark results

SPARC T5-8, 1024 threads

Per-core + asynchronous combining (blue)
Per-core + synchronous combining (green)

Even work (Approx 1k cycles)

Imbalanced work (1024:1)
PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)

OpenMP

Callisto-RTS

Threads

1024
512
256
128
64
32

Batch size

1024
512
256
128
64
32

1.0
2.0
3.0
4.0

Normalized execution time
## PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)

<table>
<thead>
<tr>
<th>Threads</th>
<th>Batch size</th>
<th>Normalized execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1024</td>
<td>1.0</td>
</tr>
<tr>
<td>64</td>
<td>512</td>
<td>1.5</td>
</tr>
<tr>
<td>128</td>
<td>256</td>
<td>2.0</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>2.5</td>
</tr>
<tr>
<td>512</td>
<td>64</td>
<td>3.0</td>
</tr>
<tr>
<td>1024</td>
<td>32</td>
<td>3.5</td>
</tr>
</tbody>
</table>

17% improvement in best-case performance

### OpenMP

- Threads: 32, 64, 128, 256, 512, 1024
- Batch size: 1024, 512, 256, 128, 64, 16, 4

### Callisto-RTS

- Threads: 32, 64, 128, 256, 512, 1024
- Batch size: 1024, 256, 64, 16, 4

Copyright © 2015, Oracle and/or its affiliates. All rights reserved.
PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)
Betweenness-centrality
SNAP Slashdot data set (82.1K nodes, 948K edges), T5-8
Comparison with Galois

SNAP Twitter data set

![Graph showing comparison between Callisto-RTS and Galois for Xeon X4-2 and SPARC T5-8 processors.](image)
Comparison with Galois

SNAP LiveJournal data set

**Xeon X4-2**

**SPARC T5-8**
Future work

• Continuing development of the programming model
• Control over data placement as well as threads
  – Initial examples from graph workloads generally have random accesses: spread data and threads widely in the machine
  – (See “Shoal”, USENIX ATC 2015)
• Interactions between multiple parallel workloads
  – OS/runtime system interaction (ref our prior work at EuroSys 2014)
  – Placement in the machine
  – Control over degree of parallelism
Integrated Cloud
Applications & Platform Services