Callisto-RTS: Fine-Grain Parallel Loops

Tim Harris, Oracle Labs
Stefan Kaestle, ETH Zurich
Daniel Goodman, Oracle Labs

15 July 2016
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.
In-memory graph analytics

Using a graph representation for your data
In-memory graph analytics

Using a graph representation for your data enables many interesting new analyses:

- **Product Recommendation**
- **Influencer Identification**
- **Community Detection**
- **Pattern Matching**

For example:

- **Purchase Record**
- **Communication**
In-memory graph analytics

Using a graph representation for your data enables many interesting new analyses.

and eliminates repeated join operations, which is much more efficient when you have a lot of relationships to traverse.

- Product Recommendation
- Influencer Identification
- Community Detection
- Pattern Matching
PageRank inner loop
PageRank inner loop
PageRank inner loop
PageRank inner loop
Hardware options
Hardware options

My laptop
• Insufficient RAM
• Insufficient CPU capacity

Non-starter
Hardware options

My laptop
• Insufficient RAM
• Insufficient CPU capacity

Non-starter

Cluster
• Enough RAM
• Enough CPU capacity
• Distributed memory model

Irregular memory accesses make it hard to program
Hardware options

My laptop
- Insufficient RAM
- Insufficient CPU capacity

Cluster
- Enough RAM
- Enough CPU capacity
- Distributed memory model

Large shared memory machine
- Enough RAM
- Enough CPU capacity
- Shared memory model

Non-starter

Irregular memory accesses make it hard to program
In-memory graph analytics

**Domain specific languages**
- Queries expressed in terms of graph concepts
- Tailor for different kinds of workload (e.g., sub-graph isomorphism)

**Generated code**
- Efficient in-memory data representations, e.g., compressed-sparse-row format
- Abundant parallelism

**Runtime system**
- Allocation of resources to a query
- Distribution of work and data within a machine

```cpp
parallel_for<node_t>([&](node_t n) {
    ...
});
```
Batch size / load imbalance trade-off

- Iteration number
- Iteration execution time
- 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
PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)
OpenMP static & dynamic loops

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

Best performance: 0.26s
My laptop

1 socket
My laptop

1 socket

4 cores per socket
My laptop

1(socket)

4(cores per socket)

2(h/w contexts per core)
My laptop

1 socket

4 cores per socket

2 h/w contexts per core
My laptop

1 socket

4 cores per socket

2 h/w contexts per core
My laptop

1 socket

4 cores per socket

2 h/w contexts per core
My laptop

1 socket

4 cores per socket

2 h/w contexts per core
T5-8

8 sockets
T5-8

8 sockets

16 cores per socket
T5-8

- 8 sockets
- 16 cores per socket
- 8 h/w contexts per core
T5-8

- 8 sockets
- 16 cores per socket
- 8 h/w contexts per core
T5-8

- 8 sockets
- 16 cores per socket
- 8 h/w contexts per core
T5-8

- 8 sockets
- 16 cores per socket
- 8 h/w contexts per core
T5-8

- 8 sockets
- 16 cores per socket
- 8 h/w contexts per core
The problem

**My laptop**
- 8 Threads accessing the counter
- The counter is always on the required socket
- 1 time in 4 the counter is on the required core

**T5-8**
- 1024 Threads accessing the counter
- 1 time in 8 the counter is on the required socket
- 1 time in 128 the counter is on the required core
PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)

OpenMP

<table>
<thead>
<tr>
<th>Threads</th>
<th>1024</th>
<th>256</th>
<th>64</th>
<th>16</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>256</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
</tbody>
</table>

Callisto-RTS

<table>
<thead>
<tr>
<th>Threads</th>
<th>1024</th>
<th>256</th>
<th>64</th>
<th>16</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>256</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
</tbody>
</table>

Batch size

1.0 1.5 2.0 2.5 3.0 3.5 4.0
Normalized execution time
PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)

OpenMP

Callisto-RTS

17% improvement in best-case performance

1024 512 256 128 64 32
Threads

1024 256 64 16 4
Batch size

1024 512 256 128 64 32
Threads

1024 256 64 16 4
Batch size

Normalized execution time
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
Risk load imbalance

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

Our approach: support efficient small batches
Techniques

1. Request combining
2. Asynchronous work requests
Techniques

1. Request combining
2. Asynchronous work requests
Approach

- 8 sockets
- 16 cores per socket
- 8 h/w contexts per core
Approach

Per-socket iteration counters, reducing communication between sockets. Steal from other sockets when own work complete.

- 8 sockets
- 16 cores per socket
- 8 h/w contexts per core
Approach

8 sockets

16 cores per socket

8 h/w contexts per core

Per-socket iteration counters, reducing communication between sockets. Steal from other sockets when own work complete.

Aggregate requests within a socket, reducing contention on per-socket counter.
Approach

- 8 sockets
- 16 cores per socket
- 8 h/w contexts per core

Per-socket iteration counters, reducing communication between sockets. Steal from other sockets when own work complete.

Aggregate requests within a socket, reducing contention on per-socket counter.

Further aggregation within a core, exploiting shared cache.
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**

- **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
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

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

0..512 512..1024
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
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
Techniques

1. Request combining
2. Asynchronous work requests
Asynchronous combining of requests
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, and the number of requests batched together will be larger.
PageRank – SNAP LiveJournal (4.8M vertices, 69M edges)

17% improvement in best-case performance
Microbenchmark results

SPARC T5-8, 1024 threads

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

Even work

Normalized speedup

Batch size

Per-socket counters
Per-core counters
Per-thread counters

(Approx 1k cycles)
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)
Comparison with Galois

SNAP LiveJournal data set

Xeon X4-2

<table>
<thead>
<tr>
<th>Threads</th>
<th>Speedup / sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
</tr>
<tr>
<td>8</td>
<td>8.0</td>
</tr>
<tr>
<td>16</td>
<td>16.0</td>
</tr>
</tbody>
</table>

Galois

Callisto-RTS

SPARC T5-8

<table>
<thead>
<tr>
<th>Threads</th>
<th>Speedup / sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
</tr>
<tr>
<td>16</td>
<td>4.0</td>
</tr>
<tr>
<td>64</td>
<td>8.0</td>
</tr>
<tr>
<td>256</td>
<td>16.0</td>
</tr>
<tr>
<td>1024</td>
<td>32.0</td>
</tr>
</tbody>
</table>

Per-socket

Galois

Callisto-RTS
Nested loops
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
Nested loops: default, all threads participate
Nested loops: outer level – just 1+5 participate
Nested loops: inner level — help respective leaders
Betweenness-centrality

SNAP Slashdot data set (82.1K nodes, 948K edges), T5-8
In-memory graph analytics

**Domain specific languages**
- Queries expressed in terms of graph concepts
- Tailor for different kinds of workload (e.g., sub-graph isomorphism)

**Generated code**
- Efficient in-memory data representations, e.g. compressed-sparse-row format
- Abundant parallelism

**Runtime system**
- Allocation of resources to a query
- Distribution of work and data within a machine

```cpp
parallel_for<node_t>([&](node_t n) {
  ...
});
```
RTS use cases

- DSLs
- Libraries
- DSLs
- Libraries
- DSLs
- Libraries
- C++
- Java
- Other Languages

Run Time System
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