Integrating query processing with parallel languages

Brandon Myers
University of Washington
bdmyers@cs.washington.edu
2015/4/21
high-performance computing programming

MPI/CAF/PGAS

high ceiling
high effort

high performance parallel system
high-performance computing programming

MPI/CAF/PGAS

high performance parallel system

SQL database
Applications and query processing

query processing for management of persistent data
Applications and query processing

- Application
- Query processing
- Processing + DRAM

Query processing for management of persistent data
Query processing for productivity and performance
Limited programming model

HadoopMR, Dryad, Spark, Fink, …

Application

Query processing

Data parallel model

Shared memory/message passing
Applications and query processing

How are queries and application programs optimized together?
integration of parallel languages with query processing using code generation and program analysis can improve performance and productivity for parallel programs.
Parallel languages & query processing

**Logical query plan**

**Physical query plan**

- generate code
- source or bytecode program
- compile
- link or inline
- libraries

**Machine code**

**Application**

**Query processing**

**Processing + Memory**

- partitioned global address space languages
- specialization in parallel query processing
- future work
Parallel languages & query processing

partitioned global address
space languages

specialization in parallel query
processing

future work
Partitioned global address space (PGAS) languages
Example PGAS program

global int* histogram = new global int[N] block;
global int events[SIZE] block;
parallel_read(events, “input”);
forall e in events {
    atomic histogram[e] += 1; ← multiple roundtrips!
}
Challenge #1: input data that is hard to partition evenly leads to poor locality and imbalanced load
Example PGAS program

global int* histogram = new global int[N] block;
global int events[SIZE] block;
parallel_read(events, "input");
forall e in events {
  on partition(histogram[e]) {
    atomic histogram[e] += 1;
  }
}
Challenge #2: commodity networks are designed for large packets.
Grappa
PGAS language/runtime

Global data

DRAM

Core

Global Tasks

lightweight RPCs

cooperative multi-threading

cache-efficient message aggregation

Infiniband RDMA (user mode)
Testing aggregation

![Graph showing bandwidth (GB/s) vs. message size (16 B, 1 kB, 64 kB). The graph compares MPI and RDMA (verbs) with two lines: one for aggregated GUPS and another for unaggregated GUPS.]
PGAS for data intensive systems

Grappa – Distributed shared memory

MapReduce
GraphLab
Relational Query Engine
Irregular apps, native code, etc...

Linux x86 node

Commodity network

Linux x86 node

Grappa: 17K lines C++11
Runs on x86 Linux clusters with MPI and fast networks (InfiniBand)

[ATC ’15]
Grappa/GraphLab application performance

We used Grappa to build a distributed backend to Raco, a relational algebra compiler and optimization framework. For large packets discussed in §3.3.1, or 61% of the bandwidth (including delegate operations), over the 85 iterations, the average aggregated size of packets as fast as the network can send them, but is still significantly better than unaggregated bandwidth. Our particular approach for the Grappa backend to Raco supports a variety of relational query language representations.

Figure 9 demonstrates the connection between concurrency and aggregation over time while executing PageRank. We can clearly see that each iteration, the number of concurrent tasks spikes as in bandwidth due to aggregating the many concurrent outgoing edges, which leads to a corresponding spike on outgoing edges, which correlates directly with the concurrency at each time step, peaks decreasing as fewer vertices are being updated. The bottom shows message bandwidth per node, which correlates directly with the concurrency at each time step, peaks decreasing as fewer vertices are being updated.

Figure 10 demonstrates the connection between concurrency and aggregation over time while executing PageRank. We can clearly see that each iteration, the number of concurrent tasks spikes as in bandwidth due to aggregating the many concurrent outgoing edges, which leads to a corresponding spike on outgoing edges, which correlates directly with the concurrency at each time step, peaks decreasing as fewer vertices are being updated. The bottom shows message bandwidth per node, which correlates directly with the concurrency at each time step, peaks decreasing as fewer vertices are being updated.

Figure 10: Grappa PageRank execution over time on 32 nodes. The top shows the total number of concurrent tasks per node. The bottom shows message bandwidth per node, which correlates directly with the concurrency at each time step, peaks decreasing as fewer vertices are being updated.

Average: 1.33x

(31 nodes)
Native Grappa programs

• Implemented Beamer’s direction-optimizing BFS [SC ’12] for low-diameter scale-free graphs

• Not expressible in GraphLab model
Writing against Grappa directly

We see that this workload utilizes a small fraction (4%) of the small-message rate of Grappa (Figure 8). The remaining 2X difference in performance is due to the rate of the 40-byte append delegates (32-byte vectors to Grappa's GraphLab engine, we also show a custom algorithm for BFS implemented natively which employs the populations may be counted.

The dataset is 8.9GB and contains 123M instances. The reduction to Grappa's GraphLab engine, we also show a custom algorithm for BFS implemented natively which employs the populations may be counted.

This provides a good indicator of the raw execution time of its execution time in this step. We attribute this difference (5X) between Spark and Grappa. This difference is likely due to a number of factors – data-structures, object serialization, JVM vs C++, garbage collection, etc.

For BFS, Beamer's direction-optimizing algorithm has been shown to greatly improve performance on the graph market. As a software implementation of multi-threading for mainstream general-purpose processors, Beamer's bottom-up optimization to achieve even better performance. This direction-optimizing algorithm for BFS implemented natively which employs the populations may be counted.

The results are shown in Figure 12a for K = 10000. We find Grappa-MapReduce to be nearly an order of magnitude faster than the comparable Spark implementation. Absolute runtime for Grappa-MapReduce is 0.13s per iteration for K = 10000 and 17.3s for K = 1000000. We find Grappa-MapReduce to be 4.4X faster than the comparable Grappa MapReduce version (Figure 12b). Except for very small numbers of clusters, the problem is compute-bound as most execution time is spent in the reduce step. Here we see, at large numbers of clusters, the problem is compute-bound as most execution time is spent in the reduce step. When we reduce the synchronization costs of updates. The first innovation in DSM over the past 30 years has focused on software distributed shared memory.
Parallel languages & query processing

partitioned global address space languages

specialization in parallel query processing

future work
Parallel languages & query processing

partitioned global address space languages

specialization in parallel query processing

future work
## Improving performance

<table>
<thead>
<tr>
<th>Generality</th>
<th>Execution model</th>
<th>Storage model</th>
</tr>
</thead>
<tbody>
<tr>
<td>• type-specialization</td>
<td>• block-oriented processing</td>
<td>• column storage</td>
</tr>
<tr>
<td>• generation and compilation of for-loop code</td>
<td>• operator-at-a-time</td>
<td>• estimation of efficient vertical partitions</td>
</tr>
<tr>
<td>• compiler optimization across operators</td>
<td>• vector-at-a-time</td>
<td>• compression</td>
</tr>
<tr>
<td>• generative programming techniques</td>
<td>• vectorization+pipelining</td>
<td></td>
</tr>
<tr>
<td>• domain-specific processor</td>
<td>• query optimization that considers v+p</td>
<td></td>
</tr>
<tr>
<td>• per-query accelerator</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Query compilation

[Link to: Krikellas ’10, Neumann ’11, Klonatos ’14, Myers ’14, Nagel ‘14]
Compiling pipelined plans

\[
\text{select } \text{name}, \text{cnt from Author,} \\
(\text{select } \text{author_id}, \text{count(*) as cnt} \\
\quad \text{from Article} \\
\quad \text{where Article.year < 2000} \\
\quad \text{group by author_id}) \text{ Pubcounts} \\
\text{where Author.id==Pubcounts.author_id}
\]
Compiling pipelined plans

```csharp
HashTable<int,ArticleTuple> cnt_table = new
for t in Article {
    if t.year < 2000 {
        cnt_table[t.author_id]++
    }
}
```
Query compilation for distributed processing

[Crotty ’14, Li ’14, Seo ’14, Murray ‘11]

pipeline fragment code

[Myers ’14]

pipeline as parallel code

sequential compiler

sequential compiler

parallel compiler

machine code

machine code
Compilation to PGAS

```c
spawn task0 {
  parallel for t in Article {
    if t.year < 2000 {
      on partition(hash(t.author_id)) {
        atomic cnt_table[t.author_id] += 1
      }
    }
  }
}
```
Optimization over parallel pipeline code
Optimization over parallel pipeline code
Evaluation

• *Radish* performance vs. in-memory data processing system performance?

• How much does whole-parallel-program optimization benefit performance?
Radish and Spark on SP²bench

41.4x

small input to joins

5.3x

large input to many joins

16x

{authors} x {authors}
Execution breakdown
Network comparison
CPU comparison
How much does whole-parallel-program optimization benefit performance?

**Baseline:** *Radish-fragments*

**Treatment:** *Radish-whole*
Parallel languages & query processing

- Physical query plan
  - query optimization
  - generate code
- Source or bytecode program
  - compile
  - link or inline
- Libraries
- Machine code
- Application
- Query processing
- Processing + memory

Partitioned global address space languages

Specialization in parallel query processing

Future work
Parallel languages & query processing

partitioned global address space languages

specialization in parallel query processing

future work
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

var numberGroups = 
    from n in numbers
    group n by n % 5 into g
    select new { Remainder = g.Key, Numbers = g };

foreach (var g in numberGroups) {
    ...
}
Query planning with UDFs

```sql
select priceScore(*)
from order
where name like 'ba%'
```

[Hellerstein '98, Chaudhuri '99]
Related work #1: Analyzing UDFs

\[
\text{select } \text{priceScore(\*)} \text{ from order where name like 'ba%'}
\]

compute time & memory time$^1$

isSelection or isProjection$^2$

$R/W$ sets$^3$

1. Crotty '14
2. Jahani '11, Iu '10
3. Hueske '12
Optimizing queries with UDFs

- **data structure semantics**: e.g., hash table’s inputs are used only on the output”

- **value constraints**: e.g., Array[Int] contains integers in the range [0, 42)
Optimizing queries with UDFs

1. how do you pass semantic information down in a maintainable compiler stack?

2. what is the best assignment of concerns?

3. how much benefit from the larger optimization window?
Related work #2: Impedance mismatch

for (e : employees) {
    for (p : project) {
        if (e.id==p.empid) {
            list.add(e)
        }
    }
}

for (i : range(1,num)) {
    list.add(db.query(
        "select * from items
        where id = ?", i))
}

predicted = apply_model(param1, param2)
db.query("insert into Tests values(?,?,?)", predicted, param1, param2)

inferring queries from imperative code
[Weidermann '07, '08, [Cheung '13]

optimizing multiple invocations
[Giannikis '12, Cheung '14, Guravannavar '08, Chavan '11]

using value constraints from the application
[Cheung '15]
Considering application context in parallel queries

• Application context can advise the query planning

• Particularly important in parallel programs, where data is explicitly distributed

1. what kinds of information are useful? 

2. how can optimization decisions be reused? 

3. how can the system balance programmer and optimizer decisions about layout and data representation?
Applications and query processing

- Query processing for management of persistent data
- Query processing for productivity and performance

application

query processing

processing + DRAM
Conclusion

partitioned global address space languages

specialization in parallel query processing

future work

get the code!
RACO+RADISH: github.com/uwescience/raco
GRAPPA: grappa.io
People

eScience
Dan Halperin
Bill Howe
Andrew Whitaker

Grappa
Jacob Nelson
Brandon Holt
Vincent Lee

Mark Oskin
Luis Ceze
Simon Kahan
Attribution

Created by Christopher Holm-Hansen from the Noun Project
Created by Roman Rusinov from the Noun Project
Created by Eddie Alshehri from the Noun Project