How to make queries go fast and play nice with parallel languages

Brandon Myers
Department of Computer Science & Engineering
University of Washington
Parallel query processing

```
select * from Author,
(select a_id, count(*) from Article
 where Article.year < 2000
 group by a_id) Pubcounts
where Author.id==Pubcounts.a_id
```

R. S. Xin et al. Shark: SQL and rich analytics at scale. SIGMOD 13
Parallel query processing

```
select * from Author,
    (select a_id, count(*) from Article
    where Article.year < 2000
    group by a_id) Pubcounts
where Author.id==Pubcounts.a_id
```

R. S. Xin et al. Shark: Sql and rich analytics at scale. SIGMOD 13
Eliminating overheads

Eliminating overheads

Eliminating overheads

Eliminating overheads

Compiling for distributed machines

```
select R.a * R.b, R.b, S.b from R,S where R.b = S.a
```

program generation
Compiling for distributed machines

```sql
select R.a * R.b, R.b, S.b from R,S where R.b = S.a
```

```python
for r in R
    store(hash(r))
while pull()
    insert(r.b, r)
```
Compiling for distributed machines

```
for r in R
    store(hash(r))

while pull()
    insert(r.b, r)
```

select R.a * R.b, R.b, S.b from R,S where R.b = S.a

- Program generation
- Sequential compiler
- Machine code
Compiling for distributed machines

```
select R.a * R.b, R.b, S.b from R,S where R.b = S.a
```

```
for r in R
  store(hash(r))
  while pull()
    insert(r.b, r)
```

```
for r in R
  on partition(hash(r))
  insert(r.b, r)
```

program generation

sequential compiler

parallel compiler

machine code
SELECT R.a\*R.b, R.b, S.b FROM R, S WHERE R.b = S.a;

Example
SELECT R.a*R.b, R.b, S.b FROM R, S WHERE R.b=S.a;

Example:

Scan(R)  
Scan(S)  
Shuffle(h(b), (a,b))  
Shuffle(h(a), (a,b))  
HashJoin(R.b=S.a)  
Apply(t = R.a*R.b)  
Emit(t, R.b, S.b)
Partitioned global address space (PGAS)

- programming model
  - shared memory with partitions
  - parallel tasks: loops, spawns
  - remote calls
Example...revisited

```
SELECT R.a*R.b, R.b, S.b FROM R,S WHERE R.b=S.a;
```

for r in R:

\[ t = (r.a*r.b) \]

for s in lookup(r.b)

emit t, r.b, s.b
Example...revisited

```
SELECT R.a*R.b, R.b, S.b FROM R,S WHERE R.b=S.a;
```

```
for r in R:
    t = (r.a*r.b)
    for s in lookup(r.b)
        emit t, r.b, s.b

for r in R:
    t = (r.a*r.b)
    for s in lookup(r.b)
        emit t, r.b, s.b
```
Example...revisited

SELECT R.a*R.b, R.b, S.b FROM R,S WHERE R.b=S.a;

for r in R:
    t = (r.a*r.b)
    on partition [ hash(r.b) ]
        for s in lookup(r.b)
            emit t, r.b, s.b

for r in R:
    t = (r.a*r.b)
    on partition [ hash(r.b) ]
        for s in lookup(r.b)
            emit t, r.b, s.b
for $r$ in $R$:  
\[
\text{on partition [ hash(r.b) ]}
\]
for $s$ in lookup(r.b)  
\[
\text{emit t, r.b, s.b}
\]

<table>
<thead>
<tr>
<th>for $r$ in $R$:</th>
<th>for $r$ in $R$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = (r.a*r.b)</td>
<td>t = (r.a*r.b)</td>
</tr>
<tr>
<td>on partition [ hash(r.b) ]</td>
<td>on partition [ hash(r.b) ]</td>
</tr>
<tr>
<td>for s in lookup(r.b)</td>
<td>for s in lookup(r.b)</td>
</tr>
<tr>
<td>emit t, r.b, s.b</td>
<td>emit t, r.b, s.b</td>
</tr>
</tbody>
</table>
Outline

• Motivation

• New technique for [query -> PGAS]

• Implementation of Radish

• Performance results
Compilation to PGAS

```sql
select * from Author,
       (select a_id, count(*) from Article
        where Article.year < 2000
        group by a_id) Pubcounts
where Author.id = Pubcounts.a_id
```
Compilation to PGAS

```sql
select * from Author,
    (select a_id, count(*) from Article
     where Article.year < 2000
     group by a_id) Pubcounts
where Author.id==Pubcounts.a_id
```
Compilation to PGAS

```sql
select * from Author,
    (select a_id, count(*) from Article
     where Article.year < 2000
     group by a_id) Pubcounts
where Author.id==Pubcounts.a_id
```

```python
spawn task0
    parallel for each t0 in Article
        if t0.year < 2000
            on partition(cnt_table[t0.a_id])
            aggregate t0 in cnt_table[t0.a_id]

spawn task1
    parallel for each t0 in Author
        on partition(join_table[t0.id])
        materialize t0 in join_table[t0.id]

spawn task2
    task0.sync
    task1.sync
    parallel for each t0 in cnt_table
        on partition(join_table[t0.a_id])
        parallel for each t1 in join_table[t0.id]
        output Tuple(t1, t0)

task0.sync; task1.sync; task2.sync
```
parallel for each t0 in cnt_table 
  on partition(join_table[t0.a_id]) 
  parallel for each t1 in join_table[t0.id] 
  output Tuple(t1, t0)
Parallel tasks
Parallel tasks
Parallel tasks
Inter-pipeline coordination

- relation-grain (coarse) synchronization
- tuple-grain (fine) synchronization
Relation-grain synchronization

```
emit
    ↓
Join
  ↓     ↓
Join    Scan
  ↓     ↓
Select  Scan
  ↓
Scan
```
Relation-grain synchronization

```plaintext
emit

Hash table join

Hash table join  Scan

Select  Scan

Scan
```
Relation-grain synchronization

Diagram:
- Build
- Select
- Scan
- Probe
- Scan
- Probe
- Build
- emit
- Probe
- Scan
Relation-grain synchronization

task0.sync

Build

Select

Scan

Probe

Scan

emit

task1.sync

Build

Probe
Tuple-grain synchronization

emit

Join

Select

Scan

Join

Scan
Tuple-grain synchronization

Symmetric hash join

emit

Symmetric hash join  Scan

Select  Scan

Scan
Tuple-grain synchronization

```
emit
I+L left
emit
I+L left
emit
I+L right
```

```
I+L left
Select
Scan
```

```
I+L right
Scan
```
Symmetric hash join

Left

Right

(1,6)

(1,5), (1,9)

core0

core1

core2

core3

pipeL

pipeR
Symmetric hash join

Left

Right

pipeL

pipeR

(1,6)

(1,6)

(1,5), (1,9)

core0

core1

core2

core3

(1,0)
Interface to parallel code

global tuple * results

local

local

local

local

global tuple * results
Implementation of Radish
Evaluation

• How does the performance of Radish compare to another in-memory query processing system for analytical queries?

• *RADISHX* backend built upon Grappa

• Compare to Shark: SQL on Spark
  • input and output tables in memory
Graph data benchmark: Sp$^2$bench

41.4x

small input to joins

5.3x

large input to many joins

16x

{authors} x {authors}
Execution breakdown
Network comparison

- Atomic increment rate
  - Grappa Prefetching:
    - Disabled
    - Enabled
  - Systems:
    - GraphLab (TCP)
    - Spark (TCP)
    - Grappa (TCP)
    - Grappa (RDMA)

- Time (sec):
  - 0.00
  - 0.25
  - 0.50
  - 0.75
  - 1.00

- Cumulative time

- Relative time for each system component:
  - Network/messaging
  - Serialization
  - Iteration
  - Scheduler
  - Application

- Component colors:
  - network/messaging: #2876dd
  - serialization: #b2df8a
  - iteration: #58d6a7
  - scheduler: #f4a582
  - application: #e83e8c
CPU time comparison
Takeaways

• compiling queries is an effective approach to mitigating the CPU bottleneck

• we described a technique to generate code for distributed memory systems by leveraging parallel languages

• our system, RADISH, outperforms a state-of-the-art distributed data analysis platform

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

Grappa talk today
1:30pm CSE403
People

eScience

- Dan Halperin
- Bill Howe
- Andrew Whitaker

Grappa

- Jacob Nelson
- Brandon Holt
- Vincent Lee
- Mark Oskin
- Luis Ceze
- Simon Kahan
backups
Applicability

- no fault tolerance as is, so not a replacement when nodes are likely to fail during query
- fault tolerance akin to RDD lineage could be applied atop shared memory
- RADISHX compilation time averages 19s for Sp2bench. Until this is improved, Radish is most applicable for repetitive (like classifier) or sufficiently long running queries
- also, there’s room for improvement in compilation time