Morsel-Driven Parallelism

Matt Holman and Andrew Ross

CS 265

1. what’s the problem?

(we’re not taking advantage of multi-core, NUMA architectures.)
NUMA NUMA

**Nehalem EX**
- Socket 0: 8 cores, 24MB L3
- Socket 1: 8 cores, 24MB L3
- Socket 3: 8 cores, 24MB L3
- DRAM: 25.6GB/s

**Sandy Bridge EP**
- Socket 0: 8 cores, 20MB L3
- Socket 1: 8 cores, 20MB L3
- Socket 3: 8 cores, 20MB L3
- DRAM: 51.2GB/s

Communication:
- Socket 0 to Socket 3: 12.8GB/s (bidirectional)
- Socket 1 to Socket 2: 16.0GB/s (bidirectional)
2. why is it important?
3. why is it hard?
What is “non-uniform”?  

(also, what is NoSQL?)
What is “non-uniform”?

What does it mean to be NUMA-friendly
What is “non-uniform”?

Keep threads and specific data close together?
Concurrency is hard
Threads don’t know which data is local
4. existing solutions
(and their problems)
The Volcano Model (Graefe 1990)
The Volcano Model (Graefe 1990)

What is it
The Volcano Model (Graefe 1990)

What is it

Vanilla operators unaware of

3 types of parallelism, Goetz picks

Plan everything in advance

Complex plans, simple operators?
Bushy Parallelism

=Threads=

-Task 1-

-Task 2-

-Task 3-

Pros/cons?
Bushy Parallelism

What if each task uses different data?
5. Core intuition for paper
Data

Operators (lock-free, efficient)

Results

Threads

Morsels
Pipelines

Sequential pipelines, concurrent data

Elastic degree of parallelism (# colors)
Figure 2: Parallellizing the three pipelines of the sample query plan: (left) algebraic evaluation plan; (right) three- respectively four-way parallel processing of each pipeline.
Data (and data structures) stay close to threads
6. what the paper does
Table of contents
Dispatcher
Elasticity (+ easy cancel)
Hash join
Lock-free Hash Table
NUMA-friendly Partitioning
Grouping and Aggregation
Parallel Merge Sort
Table of contents

Dispatcher
Elasticity (+ easy cancel)
Hash join
Lock-free Hash Table
NUMA-friendly Partitioning
Grouping and Aggregation
Parallel Merge Sort
The Dispatcher

Locality

Elasticity

Load Balancing

Figure 5: Dispatcher assigns pipeline-jobs on morsels to threads depending on the core
Table of contents
Dispatcher
**Elasticity (+ easy cancel)**
Hash join
Lock-free Hash Table
NUMA-friendly Partitioning
Grouping and Aggregation
Parallel Merge Sort
Workers can decide to help with a different task or query after every morsel.
Table of contents

Dispatcher
Elasticity (+ easy cancel)

Hash join
Lock-free Hash Table
NUMA-friendly Partitioning
Grouping and Aggregation
Parallel Merge Sort
Figure 3: NUMA-aware processing of the build-phase
Figure 4: Morsel-wise processing of the probe phase
Table of contents
Dispatcher
Elasticity (+ easy cancel)
Hash join
Lock-free Hash Table
NUMA-friendly Partitioning
Grouping and Aggregation
Parallel Merge Sort
```c
insert(entry) {
    // determine slot in hash table
    slot = entry->hash >> hashTableShift
    do {
        old = hashTable[slot]
        // set next to old entry without tag
        entry->next = removeTag(old)
        // add old and new tag
        new = entry | (old&tagMask) | tag(entry->hash)
        // try to set new value, repeat on failure
        while (!CAS(hashTable[slot], old, new))
    } while (false)
}
```

Figure 7: Lock-free insertion into tagged hash table
Compare-and-swap (CAS)

1. Destination
2. Compared value
3. Swap value

- If Destination == Compared value:
  - Destination = Swap value
  - Return true
- If Destination != Compared value:
  - Destination unchanged
  - Return false
Simpler example of CAS hashtable
Simpler example of CAS hashtable

Thread 1
*ptr1 → k1, v1 → *ptr0

Thread 2
*ptr2 → k2, v2 → *ptr0

*ptr0

k0, v0

...
Simpler example of CAS hashtable

Thread 1

*ptr1 → k1, v1

*ptr1 → *ptr0

*ptr0 → k0, v0

Thread 2

*ptr2 → k2, v2

*ptr2 → *ptr0

*ptr0 → k0, v0

*ptr0 → k0, v0
Bad way

Thread 1
*ptr1 → k1, v1 → *ptr0

Thread 2
*ptr2 → k2, v2 → *ptr0

*ptr0 → k0, v0 → ...

...
Bad way

Thread 1
*ptr1 → k1, v1 → *ptr0
update

Thread 2
*ptr2 → k2, v2 → *ptr0

*ptr0 → k0, v0 → ...

...
Bad way

Thread 1

*ptr1 → k1, v1 → *ptr0

Thread 2

*ptr2 → k2, v2 → *ptr0

*ptr1 → k1, v1 → k0, v0
Bad way

Thread 1
*ptr0 → k1, v1

*ptr1

Thread 2
*ptr0 → k2, v2

*ptr0
Bad way

Thread 1
*ptr0 \rightarrow *ptr1 \rightarrow k1, v1

Thread 2
*ptr2 \rightarrow k2, v2

update

*ptr0 \rightarrow *ptr1

...
Bad way

Thread 1
*ptr1, v1 → *ptr0

*ptr0

Thread 2
*ptr2 → k2, v2

*ptr2

k1, v1 → k0, v0

k2, v2

...
Bad way

Thread 1
*ptr0: k1, v1
*ptr2

Thread 2
*ptr2: k2, v2
*ptr0

*ptr2

k0, v0
k1, v1
k2, v2
Bad way

Thread 1
*ptr0 \rightarrow k1, v1

*ptr2

Thread 2
*ptr0 \rightarrow k2, v2

*ptr2

k0, v0
Good way

Thread 1
*ptr1 \rightarrow k1, v1 \rightarrow *ptr0

Thread 2
*ptr2 \rightarrow k2, v2 \rightarrow *ptr0

\*ptr0 \rightarrow k0, v0 \rightarrow \ldots

\ldots
Good way

Thread 1

*ptr1 → k1, v1 → *ptr0

Thread 2

*ptr2 → k2, v2 → *ptr0

CAS(i, *ptr0, *ptr1)
Good way

Thread 1

*ptr1 → k1, v1 → *ptr0

Thread 2

*ptr2 → k2, v2 → *ptr0

true

*ptr1

k0, v0

...
Good way

Thread 1
*ptr1 → k1, v1
*ptr0

Thread 2
*ptr2 → k2, v2
*ptr0

true
**Good way**

Thread 1

*ptr1 \( \rightarrow \) k1, v1

*ptr0 \( \rightarrow \) k1, v1

Thread 2

*ptr2 \( \rightarrow \) k2, v2

*ptr0 \( \rightarrow \) k2, v2

*ptr1 \( \rightarrow \) k1, v1

k0, v0

...
Good way

Thread * \( *\text{ptr}0 \rightarrow \text{v1} \rightarrow *\text{ptr}0 \)

Thread 2

\( \text{CAS}(i, *\text{ptr}0, *\text{ptr}2) \)

\( *\text{ptr}2 \rightarrow \text{k2} \rightarrow *\text{ptr}0 \)

\( \text{k1}, \text{v1} \rightarrow *\text{ptr}1 \)

\( \text{k0}, \text{v0} \rightarrow *\text{ptr}0 \)

\( \ldots \)
Good way

Thread 1
*ptr1, v1
*ptr0

Thread 2
*ptr2, v2

false

*ptr1

k1, v1

k0, v0
Good way

Thread 1

*kptr1 \rightarrow k1, v1

*rptr0

Thread 2

*kptr2 \rightarrow k2, v2

*rptr0

reread

\ldots

*kptr1

k1, v1

\ldots

k0, v0
Good way

Thread 1
*ptr0 → k1, v1

Thread 2
*ptr2 → k2, v2

k1, v1

k0, v0
Good way

Thread 1
*ptr1 → *ptr0
k1, v1 → k0, v0
CAS(i, *ptr1, *ptr2)

Thread 2
*ptr2 → *ptr1
k2, v2 → k1, v1
...
Good way

Thread 1
*ptr1 \rightarrow k1, v1
*ptr0

Thread 2
*ptr2 \rightarrow k2, v2
true

*ptr2 \rightarrow k2, v2

k2, v2 \rightarrow k1, v1 \rightarrow k0, v0
Good way
Good way

Thread 1
*ptr0 \rightarrow k1, v1

Thread 2
*ptr1 \rightarrow k2, v2

*ptr2 \rightarrow k1, v1

*ptr2 \rightarrow k0, v0
Table of contents
Dispatcher
Elasticity (+ easy cancel)
Hash join
Lock-free Hash Table
**NUMA-friendly Partitioning**
Grouping and Aggregation
Parallel Merge Sort
Table of contents
Dispatcher
Elasticity (+ easy cancel)
Hash join
Lock-free Hash Table
NUMA-friendly Partitioning
Grouping and Aggregation
Parallel Merge Sort
Figure 8: Parallel aggregation
Table of contents

Dispatcher
Elasticity (+ easy cancel)
Hash join
Lock-free Hash Table
NUMA-friendly Partitioning
Grouping and Aggregation
Parallel Merge Sort
Figure 9: Parallel merge sort
6.5. Does the paper prove its claims?
System gets ~20-30x speedup with first 32 threads
Remaining virtual threads get it to ~30-40x

Figure 11: TPC-H scalability on Nehalem EX (cores 1-32 are “real”, cores 33-64 are “virtual”)
System gets ~20-30x speedup with first 32 threads
Remaining virtual threads get it to ~30-40x

Vectorwise maxes out at 10x speedup

Figure 11: TPC-H scalability on Nehalem EX (cores 1-32 are “real”, cores 33-64 are “virtual”)
System gets ~20-30x speedup with first 32 threads.
Remaining virtual threads get it to ~30-40x.

Vectorwise maxes out at 10x speedup vs.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.28</td>
<td>32.4</td>
<td>82.6</td>
<td>0.2</td>
<td>1.40</td>
<td>1.13</td>
<td>30.2</td>
<td>12.5</td>
<td>0.5</td>
<td>74</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>0.08</td>
<td>22.3</td>
<td>25.1</td>
<td>0.5</td>
<td>15.17</td>
<td>0.63</td>
<td>4.6</td>
<td>8.7</td>
<td>0.5</td>
<td>55</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0.66</td>
<td>24.7</td>
<td>48.1</td>
<td>4.4</td>
<td>25.34</td>
<td>3.83</td>
<td>7.3</td>
<td>13.5</td>
<td>4.6</td>
<td>76</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>0.38</td>
<td>21.6</td>
<td>45.8</td>
<td>2.5</td>
<td>15.32</td>
<td>2.73</td>
<td>9.1</td>
<td>17.5</td>
<td>6.5</td>
<td>68</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>0.97</td>
<td>21.3</td>
<td>36.8</td>
<td>5.0</td>
<td>29.30</td>
<td>4.52</td>
<td>7.0</td>
<td>27.8</td>
<td>13.1</td>
<td>80</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>0.17</td>
<td>27.5</td>
<td>80.0</td>
<td>0.1</td>
<td>4.43</td>
<td>0.48</td>
<td>17.8</td>
<td>21.5</td>
<td>0.5</td>
<td>75</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>0.53</td>
<td>32.4</td>
<td>43.2</td>
<td>4.2</td>
<td>39.38</td>
<td>3.75</td>
<td>8.1</td>
<td>19.5</td>
<td>7.9</td>
<td>70</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>0.35</td>
<td>31.2</td>
<td>34.9</td>
<td>2.4</td>
<td>15.24</td>
<td>6.7</td>
<td>3.9</td>
<td>7.7</td>
<td>6.7</td>
<td>39</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>2.14</td>
<td>32.0</td>
<td>34.3</td>
<td>5.3</td>
<td>48.32</td>
<td>5.7</td>
<td>8.3</td>
<td>7.2</td>
<td>5.7</td>
<td>63</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>0.60</td>
<td>20.0</td>
<td>26.7</td>
<td>5.2</td>
<td>37.24</td>
<td>5.7</td>
<td>8.3</td>
<td>7.2</td>
<td>5.7</td>
<td>63</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>0.09</td>
<td>37.1</td>
<td>21.8</td>
<td>2.5</td>
<td>25.16</td>
<td>6.6</td>
<td>9.6</td>
<td>21.3</td>
<td>4.7</td>
<td>61</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>0.22</td>
<td>42.0</td>
<td>64.5</td>
<td>1.7</td>
<td>5.34</td>
<td>6.2</td>
<td>11.4</td>
<td>46.6</td>
<td>13.3</td>
<td>74</td>
<td>37</td>
</tr>
<tr>
<td>13</td>
<td>1.95</td>
<td>40.0</td>
<td>21.8</td>
<td>10.3</td>
<td>54.25</td>
<td>6.2</td>
<td>11.4</td>
<td>46.6</td>
<td>13.3</td>
<td>74</td>
<td>37</td>
</tr>
<tr>
<td>14</td>
<td>0.19</td>
<td>24.8</td>
<td>43.0</td>
<td>6.6</td>
<td>29.34</td>
<td>2.4</td>
<td>7.3</td>
<td>13.7</td>
<td>4.7</td>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>0.44</td>
<td>19.8</td>
<td>23.5</td>
<td>3.5</td>
<td>34.21</td>
<td>1.6</td>
<td>7.2</td>
<td>16.8</td>
<td>6.0</td>
<td>62</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>0.78</td>
<td>17.3</td>
<td>14.3</td>
<td>2.7</td>
<td>62.16</td>
<td>1.6</td>
<td>8.8</td>
<td>24.9</td>
<td>8.4</td>
<td>53</td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>0.44</td>
<td>30.5</td>
<td>19.1</td>
<td>0.5</td>
<td>13.13</td>
<td>0.8</td>
<td>15.0</td>
<td>16.2</td>
<td>2.9</td>
<td>69</td>
<td>7</td>
</tr>
<tr>
<td>18</td>
<td>2.78</td>
<td>24.0</td>
<td>24.5</td>
<td>12.5</td>
<td>40.25</td>
<td>14.9</td>
<td>6.5</td>
<td>26.3</td>
<td>8.7</td>
<td>66</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>0.88</td>
<td>29.5</td>
<td>42.5</td>
<td>3.9</td>
<td>17.27</td>
<td>2.8</td>
<td>8.8</td>
<td>7.4</td>
<td>1.4</td>
<td>79</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>0.18</td>
<td>33.4</td>
<td>45.1</td>
<td>0.9</td>
<td>5.23</td>
<td>1.9</td>
<td>9.2</td>
<td>12.6</td>
<td>1.2</td>
<td>74</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>0.91</td>
<td>28.0</td>
<td>40.7</td>
<td>4.1</td>
<td>16.29</td>
<td>12.0</td>
<td>9.1</td>
<td>18.2</td>
<td>6.1</td>
<td>67</td>
<td>9</td>
</tr>
<tr>
<td>22</td>
<td>0.30</td>
<td>25.7</td>
<td>35.5</td>
<td>1.3</td>
<td>75.88</td>
<td>3.1</td>
<td>4.3</td>
<td>7.0</td>
<td>2.4</td>
<td>66</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 11: TPC-H scalability on Reniant EX (cores 1-32 are “real”, cores 33-64 are “virtual”).
Morsels don’t need to fit in cache

Best morsel size = min. such that overhead is negligible
Elasticity:

1 query, 64 threads/query ≈ 64 queries, 1 thread/query
7. gaps in the paper?
Figure 11: TPC-H scalability on Nehalem EX (cores 1-32 are “real”, cores 33-64 are “virtual”)

Speed up of 30 on 32 cores? Other architectures?
8. next steps
8. next steps

Fin.
Why morsel-wise?
Figure 1: Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$