An Evaluation of
Morsel-Driven Parallelism:
A NUMA-Aware Query Evaluation Framework for the Many-Core Age

Presented by Matthew Cote
Overview
Setting the Stage:

- Rise of many-core architectures
  - More parallelism, less single-thread optimizations
  - ~10s to 100s of threads in modern architectures
- Main memory capacities have increased
  - Query processing not always I/O bound
- Rise of NUMA architecture
  - Needed to scale throughput from large memories
  - Allows multiple cores to access different memory banks simultaneously
NUMA Architecture

NUMA = Non-Uniform Memory Access

- Memory access cost varies depending on which chip the accessing thread and memory are located.

*Example NUMA Multi-Core Server with 4 Sockets and 32 Cores*
The Problem: Part 1

- Single threaded database applications have already been significantly optimized.
- Modern processor design focuses more on adding more cores instead of improving each core.

What about multi-core database designs?
Previous Approach to Parallelism

Parallel Volcano Model

- Operators unaware of parallelism
- Tuple streams routed among threads executing a pipeline
- Statically determine # of threads, operation for each thread
Previous Approach to Parallelism

Parallel Volcano Model

- Operators unaware of parallelism
- Tuple streams routed among threads executing a pipeline
- Statically determine # of threads, operation for each thread

Weaknesses:

- Poor load-balancing
- Not NUMA-Aware
- Not dynamically elastic
Proposed System

Morsel Driven Query Execution Framework for Main Memory Databases

- Multiple threads execute each pipeline (as before)
- Flexible and dynamic dispatcher assigns tasks at runtime
- Tasks organized by what memory they access
- Tasks work on small morsels of data
Solving the Problem

Perfect Load Balancing

- Threads for a given pipeline finish at the same time
- Possible due to morsel-size work increments and work stealing

NUMA - Awareness

- Thread (primarily) reads/writes to memory in own socket

Elasticity

- Dispatcher determines at run-time what each thread will work on
- Threads can be moved to other pipelines upon finishing a morsel
Depiction of Three-Way Hash Join

- Color = Socket
- Line = Thread
- Line Group = Pipeline
Implementation Details
Execution Overview: Scheduling Pipelines

- QEPObject controls the execution of a query
  - Transfers executable pipelines to the dispatcher
    - Only pipelines with no dependencies
  - Allocates storage for results from threads
  - Creates new morsels from results of past operations
Scheduling Threads

- Dispatcher has list of available pipelines
- Allows for inter-query parallelism
- Pipeline has list of morsels to process, one list per memory bank.
- Assigns each thread a task (pipeline job + morsel)
- Dispatcher = lock free, run by the work-requesting thread.
Scheduling Threads

- Queries dynamically assigned threads
- Preemption occurs at morsel boundary
- All cores working on a pipeline finish at the same time
  - Prevents fast threads from idling
  - Requires work stealing
Solving the Problem

Perfect Load Balancing

- Threads for a given pipeline finish at the same time
- Possible due to morsel-size work increments and work stealing

NUMA - Awareness

- Thread (primarily) reads/writes to memory in own socket

Elasticity

- Dispatcher determines at run-time what each thread will work on
- Threads can be moved to other pipelines upon finishing a morsel
Example Query Execution
Three Way Hash Join
Important First Step: Distributing a Table

- Tables partitioned semi-evenly among the memory banks
- Partition by hashing the primary key/foreign key
Building the Hash Table

- Threads processes morsel at a time
  - Write to own socket’s memory
- All threads finish phase one before going to next.
- Hash table = distributed, lock-free, perfect size

Figure 3: NUMA-aware processing of the build-phase
Building the Hash Table

- Each hash bucket has 16 bit hash tag summarizing contents.
- Probe checks tag before searching list.
- Compare and Set used to atomically grow the hash table.
Finishing the Join

Repeat similar steps as before, but now probing the hash tables.

Figure 4: Morsel-wise processing of the probe phase
Testing
Evaluation Technique

Two different architectures
Different NUMA topologies/BWs

Figure 10: NUMA topologies, theoretical bandwidth
Performance Evaluation

Normalized by execution of single-threaded HyPer

Primarily compared with Vectorwise

Significantly faster than PostgreSQL and commercial column store

Figure 11: TPC-H scalability on Nehalem EX (cores 1-32 are “real”, cores 33-64 are “virtual”)
NUMA Awareness Evaluation

Near maximum bandwidth of 100 GB/s for queries 1 and 6.

% Remote is very low - indicates most memory accesses are local

QPI is lower - indicating lower congestion on most used link

<table>
<thead>
<tr>
<th>TPC-H #</th>
<th>HyPer</th>
<th>Vectorwise</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>rd.</td>
</tr>
<tr>
<td>1</td>
<td>0.28</td>
<td>32.4</td>
</tr>
<tr>
<td>2</td>
<td>0.08</td>
<td>22.3</td>
</tr>
<tr>
<td>3</td>
<td>0.66</td>
<td>24.7</td>
</tr>
<tr>
<td>4</td>
<td>0.38</td>
<td>21.6</td>
</tr>
<tr>
<td>5</td>
<td>0.97</td>
<td>21.3</td>
</tr>
<tr>
<td>6</td>
<td>0.17</td>
<td>27.5</td>
</tr>
<tr>
<td>7</td>
<td>0.53</td>
<td>32.4</td>
</tr>
<tr>
<td>8</td>
<td>0.35</td>
<td>31.2</td>
</tr>
<tr>
<td>9</td>
<td>2.14</td>
<td>32.0</td>
</tr>
<tr>
<td>10</td>
<td>0.60</td>
<td>20.0</td>
</tr>
<tr>
<td>11</td>
<td>0.09</td>
<td>37.1</td>
</tr>
<tr>
<td>12</td>
<td>0.22</td>
<td>42.0</td>
</tr>
<tr>
<td>13</td>
<td>1.95</td>
<td>40.0</td>
</tr>
<tr>
<td>14</td>
<td>0.19</td>
<td>24.8</td>
</tr>
<tr>
<td>15</td>
<td>0.44</td>
<td>19.8</td>
</tr>
<tr>
<td>16</td>
<td>0.78</td>
<td>17.3</td>
</tr>
<tr>
<td>17</td>
<td>0.44</td>
<td>30.5</td>
</tr>
<tr>
<td>18</td>
<td>2.78</td>
<td>24.0</td>
</tr>
<tr>
<td>19</td>
<td>0.88</td>
<td>29.5</td>
</tr>
<tr>
<td>20</td>
<td>0.18</td>
<td>33.4</td>
</tr>
<tr>
<td>21</td>
<td>0.91</td>
<td>28.0</td>
</tr>
<tr>
<td>22</td>
<td>0.30</td>
<td>25.7</td>
</tr>
</tbody>
</table>

Table 1: TPC-H (scale factor 100) statistics on Nehalem EX
Elasticity Evaluation

System works for a large range of number of query streams

Demonstrates threads can be dynamically scheduled and load balanced
Evaluation of the Paper
What the Paper Did Well

- Provided concrete example to follow throughout
- Used helpful and illustrative diagrams
- Clearly stated the weakness of old systems and how this system fixes it
- Justified most architecture decisions in detail and discussed alternatives
- Created experiments to test the key optimizations
What the Paper Could Improve:

- Discuss potential applications to non in-memory databases
- Including priority based scheduling
- Including more detail on lock-free data structures - especially the dispatcher
- Talk about how queries were interwoven during testing
- Present weaknesses of this system - no real sense of a trade-off
- Calculate overhead of scheduling