Parallel Join Algorithms

@Andy_Pavlo // 15-721 // Spring 2019
PROJECT #2

This Week
→ Status Meetings

Monday April 8th
→ Code Review Submission
→ Update Presentation
→ Design Document
PARALLEL JOIN ALGORITHMS

Perform a join between two relations on multiple threads simultaneously to speed up operation.

Two main approaches:
→ Hash Join
→ Sort-Merge Join
TODAY’S AGENDA

SIMD Background
Parallel Sort-Merge Join
Evaluation
**SINGLE INSTRUCTION, MULTIPLE DATA**

A class of CPU instructions that allow the processor to perform the same operation on multiple data points simultaneously.

All major ISAs have microarchitecture support SIMD operations.

→ **x86**: MMX, SSE, SSE2, SSE3, SSE4, AVX
→ **PowerPC**: Altivec
→ **ARM**: NEON
SIMD EXAMPLE

\[ X + Y = Z \]

\[
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n \\
\end{bmatrix} +
\begin{bmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_n \\
\end{bmatrix} =
\begin{bmatrix}
  x_1 + y_1 \\
  x_2 + y_2 \\
  \vdots \\
  x_n + y_n \\
\end{bmatrix}
\]

\[
\text{for } (i=0; i<n; i++) 
\{ \\
  Z[i] = X[i] + Y[i]; \\
\}
\]
SIMD EXAMPLE

\[ X + Y = Z \]

\[
\begin{bmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n
\end{bmatrix}
+ 
\begin{bmatrix}
    y_1 \\
    y_2 \\
    \vdots \\
    y_n
\end{bmatrix}
= 
\begin{bmatrix}
    x_1+y_1 \\
    x_2+y_2 \\
    \vdots \\
    x_n+y_n
\end{bmatrix}
\]

for (i=0; i<n; i++) {
    \( Z[i] = X[i] + Y[i] \);
}
SIMD EXAMPLE

\[ X + Y = Z \]

\[
\begin{bmatrix}
    x_1 \\
    x_2 \\
    \vdots \\
    x_n \\
\end{bmatrix} +
\begin{bmatrix}
    y_1 \\
    y_2 \\
    \vdots \\
    y_n \\
\end{bmatrix} =
\begin{bmatrix}
    x_1 + y_1 \\
    x_2 + y_2 \\
    \vdots \\
    x_n + y_n \\
\end{bmatrix}
\]

\[
\text{for} \ (i=0; \ i<n; \ i++) \\ 
\quad Z[i] = X[i] + Y[i];
\]

\[
\begin{bmatrix}
    8 \\
    7 \\
    6 \\
    5 \\
    4 \\
    3 \\
    2 \\
    1 \\
\end{bmatrix} +
\begin{bmatrix}
    1 \\
    1 \\
    1 \\
    1 \\
    1 \\
    1 \\
    1 \\
\end{bmatrix} =
\begin{bmatrix}
    9 \\
    8 \\
    7 \\
    6 \\
    5 \\
    4 \\
    3 \\
    2 \\
\end{bmatrix}
\]
SIMD EXAMPLE

\[ X + Y = Z \]

\[
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n
\end{bmatrix}
+ 
\begin{bmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_n
\end{bmatrix}
= 
\begin{bmatrix}
  x_1 + y_1 \\
  x_2 + y_2 \\
  \vdots \\
  x_n + y_n
\end{bmatrix}
\]

```
for (i=0; i<n; i++) {
    Z[i] = X[i] + Y[i];
}
```
**SIMD EXAMPLE**

\[
X + Y = Z
\]

\[
\begin{bmatrix}
X_1 \\
X_2 \\
\vdots \\
X_n
\end{bmatrix} +
\begin{bmatrix}
Y_1 \\
Y_2 \\
\vdots \\
Y_n
\end{bmatrix} =
\begin{bmatrix}
X_1 + Y_1 \\
X_2 + Y_2 \\
\vdots \\
X_n + Y_n
\end{bmatrix}
\]

for \( i = 0; i < n; i++ \) {
  \( Z[i] = X[i] + Y[i]; \)
}
SIMD TRADE-OFFS

Advantages:
→ Significant performance gains and resource utilization if an algorithm can be vectorized.

Disadvantages:
→ Implementing an algorithm using SIMD is still mostly a manual process.
→ SIMD may have restrictions on data alignment.
→ Gathering data into SIMD registers and scattering it to the correct locations is tricky and/or inefficient.
SORT-MERGE JOIN (R \Join S)

Phase #1: Sort
→ Sort the tuples of R and S based on the join key.

Phase #2: Merge
→ Scan the sorted relations and compare tuples.
→ The outer relation R only needs to be scanned once.
SORT-MERGE JOIN (R \(\times\) S)

Relation R

Relation S
SORT-MERGE JOIN (R \Join S)
SORT-MERGE JOIN (R⨝S)

Relation R

MERGE!

Relation S
PARALLEL SORT-MERGE JOINS

Sorting is always the most expensive part.

Use hardware correctly to speed up the join algorithm as much as possible.
→ Utilize as many CPU cores as possible.
→ Be mindful of NUMA boundaries.
→ Use SIMD instructions where applicable.
PARALLEL SORT-MERGE JOIN (R \times S)

Phase #1: Partitioning (optional)
→ Partition R and assign them to workers / cores.

Phase #2: Sort
→ Sort the tuples of R and S based on the join key.

Phase #3: Merge
→ Scan the sorted relations and compare tuples.
→ The outer relation R only needs to be scanned once.
PARTITIONING PHASE

Approach #1: Implicit Partitioning
→ The data was partitioned on the join key when it was loaded into the database.
→ No extra pass over the data is needed.

Approach #2: Explicit Partitioning
→ Divide only the outer relation and redistribute among the different CPU cores.
→ Can use the same radix partitioning approach we talked about last time.
SORT PHASE

Create **runs** of sorted chunks of tuples for both input relations.

It used to be that Quicksort was good enough. But NUMA and parallel architectures require us to be more careful...
CACHE-CONSCIOUS SORTING

Level #1: In-Register Sorting
→ Sort runs that fit into CPU registers.

Level #2: In-Cache Sorting
→ Merge Level #1 output into runs that fit into CPU caches.
→ Repeat until sorted runs are ½ cache size.

Level #3: Out-of-Cache Sorting
→ Used when the runs of Level #2 exceed the size of caches.
CACHE-CONSCIOUS SORTING

Level #1

Level #2

Level #3

UNSORTED

SORTED
Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.

Input  

9
5
3
6

Output  

3

9
9
3
5
6
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.

Input

Output

wires = [9, 5, 3, 6]

wires[0] = \text{min}(\text{wires}[0], \text{wires}[1])
wires[1] = \text{max}(\text{wires}[0], \text{wires}[1])
wires[2] = \text{min}(\text{wires}[2], \text{wires}[3])
wires[3] = \text{max}(\text{wires}[2], \text{wires}[3])
wires[0] = \text{min}(\text{wires}[0], \text{wires}[2])
wires[2] = \text{max}(\text{wires}[0], \text{wires}[2])
wires[1] = \text{min}(\text{wires}[1], \text{wires}[3])
wires[3] = \text{max}(\text{wires}[1], \text{wires}[3])
wires[1] = \text{min}(\text{wires}[1], \text{wires}[2])
wires[2] = \text{max}(\text{wires}[1], \text{wires}[2])
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.

Input

Output

wires = [9, 5, 3, 6]

wires[0] = min(wires[0], wires[1])
wires[1] = max(wires[0], wires[1])
wires[2] = min(wires[2], wires[3])
wires[3] = max(wires[2], wires[3])
wires[0] = min(wires[0], wires[2])
wires[2] = max(wires[0], wires[2])
wires[1] = min(wires[1], wires[3])
wires[3] = max(wires[1], wires[3])
wires[1] = min(wires[1], wires[2])
wires[2] = max(wires[1], wires[2])

1

2

3
LEVEL #1 – SORTING NETWORKS

Instructions:
→ 4 LOAD
LEVEL #1 – SORTING NETWORKS

Sort Across Registers

Instructions:
→ 4 LOAD
LEVEL #1 – SORTING NETWORKS

**Sort Across Registers**

Original:

```
12 21 4 13
9 8 6 7
1 14 3 0
5 11 15 10
```

After 4 LOAD:

```
1 8 3 0
5 11 4 7
9 14 6 10
12 21 15 13
```

Instructions:

→ 4 LOAD

→ 10 MIN/MAX
LEVEL #1 – SORTING NETWORKS

Sort Across Registers

Instructions:
→ 4 LOAD

Transpose Registers

Instructions:
→ 10 MIN/MAX

→ 8 SHUFFLE
→ 4 STORE
LEVEL #2 – BITONIC MERGE NETWORK

Like a Sorting Network but it can merge two locally-sorted lists into a globally-sorted list.

Can expand network to merge progressively larger lists (½ cache size).

Intel’s Measurements
→ 2.25–3.5x speed-up over SISD implementation.
LEVEL #3 – MULTI-WAY MERGING

Use the Bitonic Merge Networks but split the process up into tasks.
→ Still one worker thread per core.
→ Link together tasks with a cache-sized FIFO queue.

A task blocks when either its input queue is empty or its output queue is full.

Requires more CPU instructions, but brings bandwidth and compute into balance.
LEVEL #3 – MULTI-WAY MERGING

Sorted Runs

Cache-Sized Queue

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

Cache-Sized Queue

MERGE

Cache-Sized Queue

MERGE

Cache-Sized Queue

MERGE

Cache-Sized Queue

MERGE

Cache-Sized Queue

MERGE

Cache-Sized Queue

MERGE

Cache-Sized Queue

MERGE

Merge operation.
LEVEL #3 – MULTI-WAY MERGING

Sorted Runs

Cache-Sized Queue

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE
MERGE PHASE

Iterate through the outer table and inner table in lockstep and compare join keys. May need to backtrack if there are duplicates.

Can be done in parallel at the different cores without synchronization if there are separate output buffers.
SORT-MERGE JOIN VARIANTS

Multi-Way Sort-Merge (M-WAY)

Multi-Pass Sort-Merge (M-PASS)

Massively Parallel Sort-Merge (MPSM)
MULTI-WAY SORT-MERGE

Outer Table
→ Each core sorts in parallel on local data (levels #1/#2).
→ Redistribute sorted runs across cores using the multi-way merge (level #3).

Inner Table
→ Same as outer table.

Merge phase is between matching pairs of chunks of outer/inner tables at each core.
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning
### Local-NUMA Partitioning

<table>
<thead>
<tr>
<th>Local-NUMA Partitioning</th>
<th>Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image" alt="Sort" /></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Sort" /></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Sort" /></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Sort" /></td>
</tr>
</tbody>
</table>

**MULTI-WAY SORT-MERGE**
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge  Same steps as Outer Table

Local-NUMA Partitioning

Sort

Multi-Way Merge

Same steps as Outer Table
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning | Sort | Multi-Way Merge | Local Merge Join | Same steps as Outer Table
---|---|---|---|---
| ![Diagram](image1.png) | ![Diagram](image2.png) | ![Diagram](image3.png) | ![Diagram](image4.png) | ![Diagram](image5.png)
MULTI-PASS SORT-MERGE

Outer Table
→ Same level #1/#2 sorting as Multi-Way.
→ But instead of redistributing, it uses a multi-pass naïve merge on sorted runs.

Inner Table
→ Same as outer table.

Merge phase is between matching pairs of chunks of outer table and inner table.
# MULTI-PASS SORT-MERGE

## Local-NUMA Partitioning

<table>
<thead>
<tr>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
<th>Partition 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="CPU" /></td>
<td><img src="image2.png" alt="CPU" /></td>
<td><img src="image3.png" alt="CPU" /></td>
<td><img src="image4.png" alt="CPU" /></td>
</tr>
<tr>
<td><img src="image5.png" alt="Memory" /></td>
<td><img src="image6.png" alt="Memory" /></td>
<td><img src="image7.png" alt="Memory" /></td>
<td><img src="image8.png" alt="Memory" /></td>
</tr>
<tr>
<td><img src="image9.png" alt="Disk" /></td>
<td><img src="image10.png" alt="Disk" /></td>
<td><img src="image11.png" alt="Disk" /></td>
<td><img src="image12.png" alt="Disk" /></td>
</tr>
</tbody>
</table>

## Local-NUMA Partitioning

<table>
<thead>
<tr>
<th>Partition 5</th>
<th>Partition 6</th>
<th>Partition 7</th>
<th>Partition 8</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image13.png" alt="CPU" /></td>
<td><img src="image14.png" alt="CPU" /></td>
<td><img src="image15.png" alt="CPU" /></td>
<td><img src="image16.png" alt="CPU" /></td>
</tr>
<tr>
<td><img src="image17.png" alt="Memory" /></td>
<td><img src="image18.png" alt="Memory" /></td>
<td><img src="image19.png" alt="Memory" /></td>
<td><img src="image20.png" alt="Memory" /></td>
</tr>
<tr>
<td><img src="image21.png" alt="Disk" /></td>
<td><img src="image22.png" alt="Disk" /></td>
<td><img src="image23.png" alt="Disk" /></td>
<td><img src="image24.png" alt="Disk" /></td>
</tr>
</tbody>
</table>
MULTI-PASS SORT-MERGE

Local-NUMA Partitioning    Sort    Local-NUMA Partitioning

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMU 15-721 (Spring 2019)
MULTI-PASS SORT-MERGE

Local-NUMA Partitioning  Sort  Global Merge Join  Sort  Local-NUMA Partitioning
MASSIVELY PARALLEL SORT-MERGE

Outer Table
→ Range-partition outer table and redistribute to cores.
→ Each core sorts in parallel on their partitions.

Inner Table
→ Not redistributed like outer table.
→ Each core sorts its local data.

Merge phase is between entire sorted run of outer table and a segment of inner table.
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sort

Globally Sorted
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

SORT1!

SORT1!

SORT1!

SORT1!
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

Cross-Partition Merge Join
MASSIVELY PARALLEL SORT-MERGE
HYPER’s RULES FOR PARALLELIZATION

**Rule #1: No random writes to non-local memory**
→ Chunk the data, redistribute, and then each core sorts/works on local data.

**Rule #2: Only perform sequential reads on non-local memory**
→ This allows the hardware prefetcher to hide remote access latency.

**Rule #3: No core should ever wait for another**
→ Avoid fine-grained latching or sync barriers.

Source: Martina Cezara Albutiu
Compare the different join algorithms using a synthetic data set.

→ **Sort-Merge:** M-WAY, M-PASS, MPSM
→ **Hash:** Radix Partitioning

**Hardware:**
→ 4 Socket Intel Xeon E4640 @ 2.4GHz
→ 8 Cores with 2 Threads Per Core
→ 512 GB of DRAM
RAW SORTING PERFORMANCE

Single-threaded sorting performance

- **C++ STL Sort**
- **SIMD Sort**

Throughput (M Tuples/sec)

Number of Tuples (in $2^{20}$)

2.5–3x Faster

Source: Cagri Balkesen
COMPARISON OF SORT-MERGE JOINS

Workload: 1.6B × 128M (8-byte tuples)

Source: Cagri Balkesen
M-WAY JOIN VS. MPSM JOIN

Workload: 1.6B ⨝ 128M (8-byte tuples)

Throughput (M Tuples/sec)

- Multi-Way
- Massively Parallel

Number of Threads

Source: Cagri Balkesen
**SORT-MERGE JOIN VS. HASH JOIN**

*Workload: Different Table Sizes (8-byte tuples)*

- **Partition**
- **Sort**
- **S-Merge**
- **M-Join**
- **Build+Probe**

<table>
<thead>
<tr>
<th>Cycles / Output Tuple</th>
<th>SORT 128M × 128M</th>
<th>HASH 128M × 128M</th>
<th>SORT 1.6B × 1.6B</th>
<th>HASH 1.6B × 1.6B</th>
<th>SORT 128M × 512M</th>
<th>HASH 512M × 1.6B</th>
<th>SORT 1.6B × 6.4B</th>
<th>HASH 6.4B × 1.6B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source: Cagri Balkesen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SORT-MERGE JOIN VS. HASH JOIN

Varying the size of the input relations

- Multi-Way Sort-Merge Join
- Radix Hash Join

Source: Cagri Balkesen
PARTING THOUGHTS

Both join approaches are equally important. Every serious OLAP DBMS supports both.

We did not consider the impact of queries where the output needs to be sorted.
Query Compilation
→ Or "Why is DSL Project is Awesome"